UNIVERSAL WEB APPLICATION SERVER

MAYUMBO NYIRENDA

Department of Computer Science

APPROVED:

_____

_____

_____

to my

MOTHER and FATHER

with love

UNIVERSAL WEB APPLICATION SERVER

by

MAYUMBO NYIRENDA

THESIS

Presented to the Faculty of Science of

The University of Cape Town

in Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF CAPE TOWN

DECEMBER 2007

Acknowledgements

I express my deep-felt gratitude to my advisor and supervisor, Dr Hussein Suleman of the Computer Science Department at the University of Cape Town, for his advice, encouragement, enduring patience, constant support and for his never-ceasing comments that helped me change the way I look at things.

I also acknowledge the support received from the VLIRE project in Zambia, both in terms of financial support and encouragement.

I thank the members of the Advanced Information Management research group and also the members of the High Performance Computing research group, all of the University of Cape Town.

I thank my family and friends for their support and encouragement. Thanks guys for ever believing in me and never failing me.

Abstract

The growth of the World Wide Web has in large part been made possible by the technologies that power it. These are the Web servers and Web browsers that many take for granted. Each has evolved to support the changing needs of users of the WWW from simple static text to highly interactive and dynamic multimedia. The Web servers, in particular, have evolved into a spectrum of different technologies to support what are now known as Web applications. These are usually installed and accessed through a Web server.

Security is a problem in Web server environments and therefore the Web servers are usually run as an un-privileged user. Performance is another problem as some of these technologies require re-initialization of the execution environment with every subsequent request. These security and performance shortcomings have been dealt with by numerous Web application technologies. Most of these technologies are language-centric and seek solutions to the security and performance shortcomings independently of each other.

The universal Web application server is proposed as an alternative solution addressing the security, language dependence and performance shortcomings of existing technologies. It has support for multiple authors in a secure environment with support for multiple implementation technologies (languages) using persistent interpreters to enhance performance.

Test results from the performance evaluation show that the introduction of the layers of processing contributes a small percentage to the total request processing time and that the universal Web application server can perform comparably to other Web application servers. Tests with twenty users also showed that packaging and deploying Web applications in the universal Web application server is an easy and viable approach. Moreover, the installation of PhpBB2 in the universal Web application server demonstrates that it can be used with realistic Web applications.

A universal Web application server that provides an efficient, secure and language independent environment has been developed and thoroughly evaluated demonstrating that a Web application server that addresses the shortcomings of existing technologies is feasible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The importance and need to interlink documents can be traced to as far back as 1945 when Vannevar Bush [12] wrote an article about a photo-electrical-mechanical device called a Memex, for memory extension, which could make and follow links between documents on microfiche. Through years of research, ideas on information sharing have been refined and eventually led to the birth of the World Wide Web (WWW) in the early 90s [8]. The World Wide Web started off as an effort to create a common information space for communication and sharing of information based on standards such as UDIs (now URIs), HyperText Markup Language (HTML) and HyperText Transfer Protocol (HTTP). Much like many other distributed systems, the WWW is built on the client-server model.

The client makes requests which are sent to the Web server which then responds to these requests. Initially the Web servers only served requests for static content. Static content defines responses that are provided by the Web server that change infrequently and only through direct human intervention. Thus a Web server was typically a file server responsible for receiving a client's request, processing it and sending the appropriate response to the client. The response was a file that had special meaning and interpretation to HTTP clients. The power of the Web was in its Universality. The Universality meant that a hypertext link could point to anything, thus making the Web an information space of disparate content and this led to its popularity, widespread use and adoption.

As the use of the Web keeps on growing, the need for content that is more oriented to individual user needs and interaction has become more apparent.

1.1 Web-Based Application Programming

Web-based applications help enhance the interactivity of the Web. They generate dynamic content that is more oriented to individual user needs. They can be broadly categorized into two architectures: client-server applications and client applets. Client-server architectures have a client issuing requests, typically using a browser, to the server that processes the request and sends the response back to the client, usually in the form of HTML for interpretation and display in the browser. In the client applet architecture [45], the client issues a request to the server, typically using a browser. In response to the request, the server sends a whole application to the browser to run locally on the client side. The client application may make requests for data from the server but most of the processing is done locally on the client side. Both approaches have their strengths and weaknesses. This research focusses on the client-server architecture for Web-based applications.

1.2 Client-server Web application architecture

The client side of the client server architecture is composed of the user and the browser. The user is the person or process that invokes the browser to make a request for a resource from the server. The server side is composed of the Web server, the Web application and/or databases. Unlike traditional applications which run locally, this architecture has a transient nature of processing. A request will leave the browser and proceed to the Web server which will then pass on the request processing information to the application. The application then, if necessary, accesses a database and computes a response which it routes to the server. The server then sends the response to the browser and the browser interprets this response for display. The transient nature of processing has a significant impact on both the performance and design of Web-based applications.

The communication between the browser and the server is achieved using the Hyper Text Transfer Protocol (HTTP) [16]. The user instructs the browser on what resource to get from the server using a Uniform Resource Locator (URL). The URL is composed of the protocol that will be used for communication, the server to which the request should be issued, the port to connect to (optional), the path name of the requested resource and

optionally the query information. The server receives this information and then checks to see what kind of request it is. If it is a request that needs an application to process it, the Web server passes the received information to the application. The response from the application is required to have headers which define the response. Multipurpose Internet Mail Extensions (MIME) [41] are used to describe the response. They are composed of type and subtype components. For example, 'text/html' implies the response will be Hyper Text Markup Language (HTML) text [40]. HTML is a language for defining how data will be displayed and formatted for presentation in a window. It is typically transferred using the HTTP protocol and is interpreted by browsers in order to display the response in a human-readable form.

These Web based applications can be categorized as either CGI based, server API based or as Web application server based.

1.2.1 CGI based Web applications

CGI [35] is the traditional protocol for communication between the Web server and Web applications. The Web server and the application communicate using environment variables and the standard input and output. The Web server forks off a CGI process when it receives a request that requires the CGI application to process it. The request processing information is set in the environment of the CGI process. The CGI process can then access this information using environment variables defined by the CGI specification. The Web server collects the response from the standard output of the CGI application. CGI-based Web applications are usually implemented as a collection of stand-alone scripts or binaries.

1.2.2 Server API Web applications

Web server API Web applications are usually implemented as binaries that are forked off as threads and share memory with the Web server. This architecture requires no information passing as the threads share memory with the Web server. The memory sharing makes the applications perform better than CGI applications. These applications access the server memory and functionality using the Web server's API. They can only be implemented using

the programming languages that define the Web server API and cannot be easily ported to other Web servers.

1.2.3 Web application server Web applications

These are applications that use special servers that are not part of the Web server and implemented using some API that is specialized for that kind of application. Generally, applications that are not forked off from the Web server and are not implemented using the Web server API are implemented as Web application server applications. The Web application server is responsible for communication with the Web server. Web applications are implemented using an interface or API that is defined by the Web application server. The Web application server also extends the functionality of the Web server by providing tools that make Web application development easier for the user. These tools are however usually for a specific Web application technology.

1.3 Motivation

CGI based Web applications offer a vast base of implementation languages. In this architecture, processes are forked to handle requests, which terminate after servicing the request. The forking of processes is costly in time and causes degradation in performance with an increase in the number of requests. For this reason, CGI based Web applications do not scale easily. The strength of CGI is that it offers a language independent solution. This makes administration easier as one does not need to configure a myriad of technologies in order to deploy Web applications implemented using different languages. A language independent solution also cuts on resources required to run multiple Web application servers each implementing a single technology. However the performance of CGI applications as the number of requests increases is a major drawback. Technologies like FastCGI [21] try to solve the problem by running the processes persistently. In the FastCGI architecture the process does not exit after serving the initial request. Solutions like FastCGI, however, implement the persistence in the Web application itself. Thus implementing Web applications using such technologies requires some knowledge of parallel processing. FastCGI

comes with libraries which try to make the process of implementing the parallel processing easy. However, the use of such libraries make the FastCGI Web applications less portable. Other efforts like SpeedyCGI [22] are language-centric.

Another issue with Web application technologies is the security in shared environments. CGI processes that process requests are forked by the Web servers. The Web servers hand over request processing control to the processes that they fork. This passing over of control makes the host system vulnerable to attacks as the processes have the privileges of the server that forked them. To solve this problem most Web servers are run as a non-privileged user. Running the Web server as a non privileged user however does not secure the different authors' scripts from one another. This is because all the scripts are run with the same privileges as the Web server and are at times run in the same part of the file system. To solve this problem, technologies like Java servlet engines run the Web applications in controlled environments called sandboxes. Other technologies, like suPHP [28], run the scripts with the privileges of the owner of the script.

Technologies like the Java servlet technology also have APIs that make Java servlets portable. Java servlets can be run in any container that implements the J2EE servlet API. The Java servlet technology is however language-centric and does not meet the need for a technology that is language independent, has good performance and offers a secure shared environment.

Technologies like FastCGI offer a solution to the performance problem while technologies that implement wrapper scripts, like suPHP, focus on security in a multi-user environment. On the other hand technologies like the Java servlet technology focus on portability and security but can only be implemented using the Java programming language. These solutions try to solve the Web application server technologies' shortcomings independently of one another. But there is still need for a Web application server that is a generalization of the existing solutions.

1.4 Aim

The aim of this research was to investigate the possibility of a universal Web application server that is a generalization of existing solutions. In its generalization, the universal Web application server should support multiple back end technologies and also attempt to provide a performance-efficient solution through the use of persistent processes. Processes should be owned and run in the context of the user and thus offer a secure shared environment. This implementation also should support easy deployment of Web applications. The aim of this research was thus to answer the following questions:

1. Is a universal Web application server feasible?
   (a) Is a Web application server that is a generalization of the existing solutions feasible?
   (b) Is a Web application framework for easy deployment that is universal to the back end technologies feasible?
2. Can such a universal Web application server perform comparably to other web application servers?
3. Can such a universal Web application server satisfy developer usability needs?

1.5 Methodology

The research was based on a prototype of the universal Web server which was implemented by two Honours students in partial fulfillment of their BSc. Honours degrees. My contribution to the software included substantial code re-organization, redesign and implementation of the universal Web application server. The Web server used for the research was the Apache Web server and the universal Web application server was implemented in C. The implemented universal Web application server was evaluated using three approaches. The first test involved the installation of phpBB2 [23] in the X-Switch system. The purpose of installing phpBB2 in the X-Switch system was to show that the X-Switch system can be used to deploy and install realistic Web applications. A usability study also was carried out

to test the usability of the universal Web application server. The usability study involved 20 subjects and comprised packaging and installing Web applications in the universal Web application server. The final tests involved performance analysis of the universal Web application server. The purpose of the performance tests was to verify that the universal Web application server can perform comparably to other Web application servers. Conclusions were then drawn based on this implementation of the universal Web application server and the tests that were carried out.

1.6 Dissertation outline

Chapter 1 presents the motivation, aim and methodology of the research.

Chapter 2 presents other work done in relation to Web application server performance and security.

Chapter 3 presents the work done on the prototype of the universal Web application server.

Chapter 4 outlines the design and implementation of the universal Web application server.

Chapter 5 discusses the experiments done to test and evaluate the universal Web application server.

Chapter 6 presents the discussion, concluding remarks and future work related to the universal Web application server.

Chapter 2

Literature review

2.1 HTTP and the World Wide Web

The Web originally started off with HTML as the main choice of developing Web pages and HTTP as the means of distributing the content. The Web is built on a client/server architecture. The server is a computer from which the user retrieves the Web site while the client is the computer that is used to access the Web site. Traditionally all the users would access the same file and thus have the same response. Adding parameters to the request however makes it possible for the server to generate a response that is customised for each individual user. Typically, the client makes the request using a program called a browser and the server uses a program called a Web server to generate the response. The client and the server communicate using the HyperText Transfer Protocol (HTTP) [16]. The client sends the request to the server using a designated HTTP port. The Web server listens on this port for any internet connections. Typically, a request contains the method type, protocol type and version, a request URI and/or request parameters. When a user enters the URL 'http://www.uct.ac.za/test?name=john' into the browser the following request is sent to the Web server at www.uct.ac.za:

GET /test?name=john HTTP/1.0

User-Agent: Mozilla/4.7

The request is sent to port 80 which is the default port for Web servers. 'Get' indicates that the request has no request body data. '/test?name=john' is the URI together with the request parameters. The parameters are separated by the '?' character from the file path. The last part indicates that the browser understands HTTP version 1.0 protocol. The lines that follow contain the headers. In this case the User-Agent is the program was used to make the request. The server processes the request and sends the response back

to the client. The first part of the response are the headers which are separated from the response body by an empty line. The following is a sample response:

```
HTTP/1.0 200 OK
Server: Apache
Content-type: text/html

<HTML>
<HEAD><TITLE>Hello</TITLE></HEAD>
<HTML>
```

HTTP defines codes which indicate the response status to the browser. In this case '200 Ok' implies the request was successfully processed. The 'Content-type' header tells the browser how to render the response in human-understandable form. The empty line signals the end of the headers and the beginning of the response body.

The need for dynamic content has led to the development of various technologies that generate dynamic content based on parameter-driven programs. The parameters are processed by a Web server by invoking an appropriate Web application which then generates information which is formatted into a Web page using HTML. This allows for the generation of dynamic content using HTML which was originally designed for static content.

2.2 Web application technologies

As the need for dynamic Web content grew it became apparent that HTTP lacked the many capabilities that were required to generate dynamic content and exploit the potential of the Web. This led to the development of dynamic, content-driven, Web application technologies. The first of these was the Common Gateway Interface (CGI) [35].

2.2.1 The Common Gateway Interface

HTTP was not originally designed to distribute dynamic content. The Common Gateway Interface (CGI) is an interface between a Web server and application that allows for the generation of dynamic content without redesigning the HTTP specification. The client

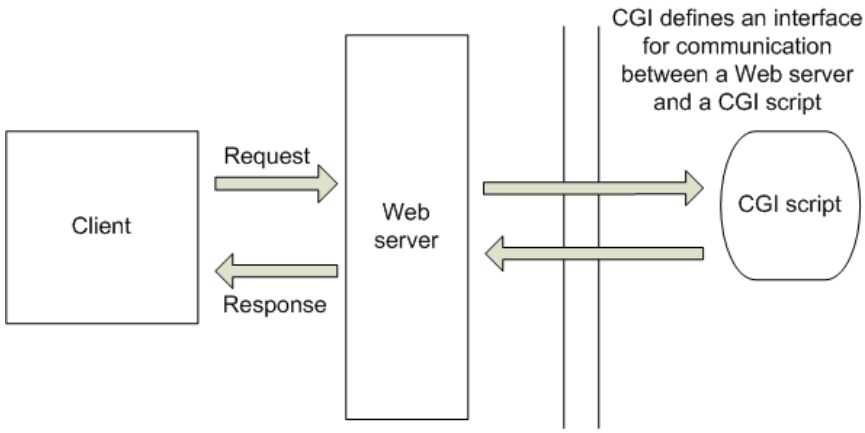

Figure 2.1: CGI defines a protocol for how the Web server and CGI script communicate

can make a request for information from the Web server and the server can then perform an action such as running a Web application to generate this information, which is then formatted into a Web page using standard HTML. CGI is a protocol for communication between Web applications and Web servers [20]. It defines and sets a standard for how the Web application and the Web server communicate with each other. All that CGI does is to specify how the external application will receive the HTTP request from the Web server and how the Web server gets the results from the CGI application (See Figure 2.1). With static content the user enters a URL in a browser to request a file from the server whereas with CGI the URL is a request to run an application. The CGI application generates the response dynamically instead of the Web server reading a static HTML file. When an HTTP request arrives the Web server first sets up the the application's environment and then creates a new process that runs the application. The Web server determines what kind of environment to set up by looking at the extension of the script to be processed and the information that came with the request. The application gets the information required to process the request from the environment variables. Request body data is sent to the standard input stream of the CGI process in the case of POST requests. The CGI process determines how much data to read by checking the 'CONTENT_LENGTH' environment variable. GET request input data is set in the 'QUERY_STRING' environment variable. The CGI process uses this input data to generate a customised response. The Web server

reads the response that the CGI process generates from the standard output of the process. The Web server then sends the response headers together with the response body to the client that made the request. With CGI a Web page user could now fill in forms and request information from the Web server based on the information submitted in the form. Dynamic Web content added user interactivity to the Web. The major strength and success of CGI was due to the fact that CGI Web applications could be written in any language and also because CGI was compatible with various Web server architectures.

However there are a few problems associated with CGI processing. As the number of requests for a CGI Web application increases the performance begins to degrade drastically. This is because every time a request is received a new process is created and the script is then compiled and executed and the process exits (See Figure 2.2). If the same request is received subsequently, the same steps are repeated and this degrades the performance of CGI processes as the request load increases. The degradation in performance is due to the overhead associated with creating the execution environment of the CGI process over and over again. The second pronounced problem with CGI is security. In an attempt to solve these problems other technologies based on CGI have been developed and are discussed in the sections to follow.

2.2.2 FastCGI

To solve the performance degradation problem in CGI Web applications, Open Market Inc developed FastCGI [21], as a variation of CGI, having processes that run persistently.

The FastCGI interface is simply a variation of the CGI interface. FastCGI uses the same protocol as CGI and a FastCGI Web application receives the same information from the Web server as a CGI Web application would have received. FastCGI however does not receive its information through environment variables as CGI does. FastCGI Web applications communicate with the Web server using inter-process communication. When an HTTP request arrives the Web server connects to the FastCGI Web application and passes the request information as defined by the CGI protocol using inter-process communication like (e.g., TCP or UNIX socket) and reads the results from the application using inter-process communication and then passes it back to the client. Using inter-process com-

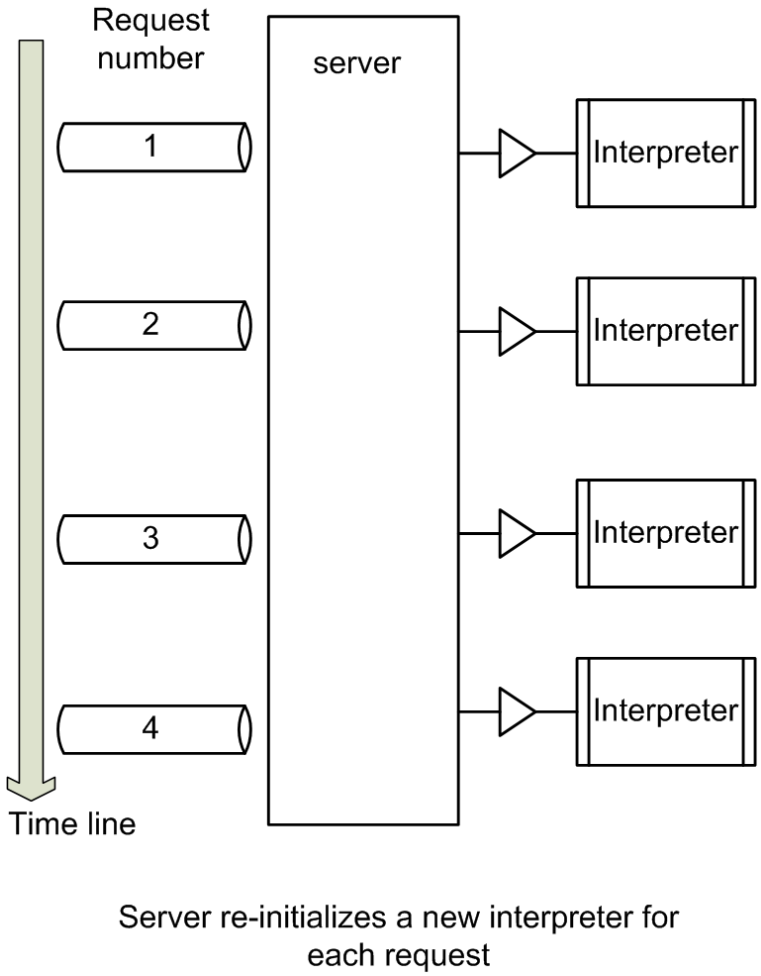

Figure 2.2: Request processing using non-persistent interpreters.

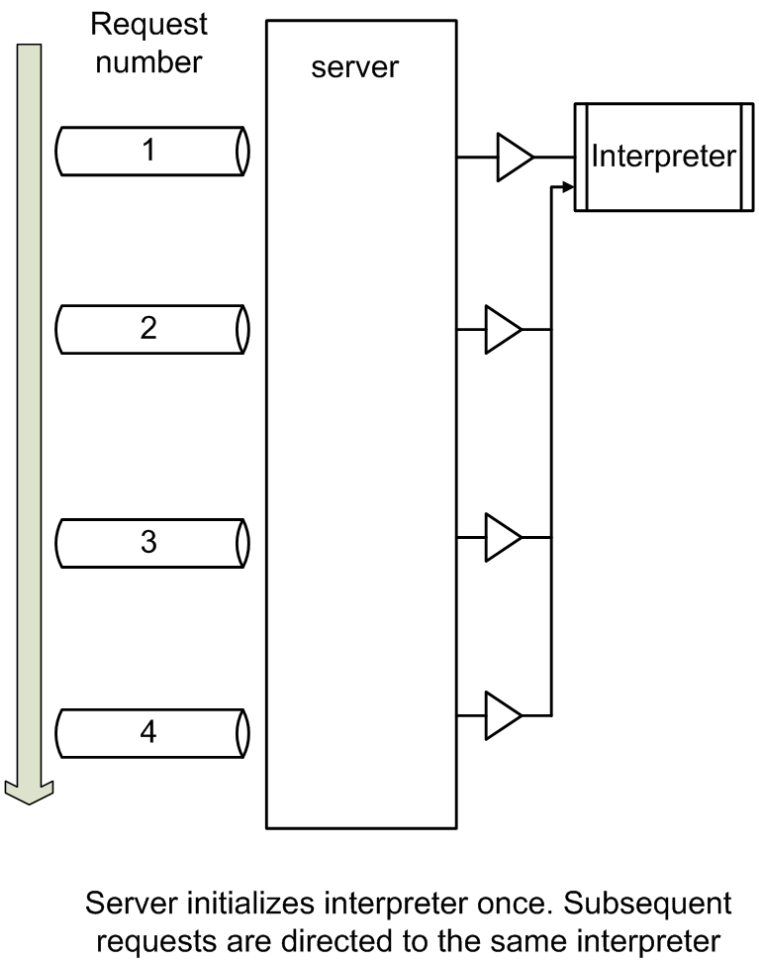

Figure 2.3: Request processing using persistent interpreters

munication for communication between the Web server and the Web application makes it possible to run the Web application persistently (See Figure 2.3). Thus the lifetime of a FastCGI application is not tied to the lifetime of the request as is the case with CGI. A FastCGI Web application once started can potentially serve many requests without terminating.

The strength of FastCGI lies in its efficient use of system resources. FastCGI uses a packet-oriented communication protocol. Every packet sent or received has a header field indicating the request ID. Thus, using one process to multiplex the communication channels, FastCGI can pass information from multiple requests to a single process and handle

the context switching internally in the application. This is a more efficient use of the processing power that is available. FastCGI applications can thus handle an arbitrary number of concurrent requests per connection while handling an arbitrary number of concurrent connections all in one process.

However, to achieve the multiplexing of the communication channels, the developer has to use FastCGI reusable components. FastCGI has a set of libraries that provide a framework that is supposed to make FastCGI application programming as manageable as possible. The use of FastCGI libraries in development of Web applications implies that existing Web applications have to be modified in order to take full advantage of the functionality of FastCGI. The FastCGI Web application framework puts the management of persistence in the application code. The advantage of this is that the application once running persistently does not have to reload shared information such as a database connection every time a new request arrives. FastCGI applications also scale easily as they can communicate over a network leaving room for various techniques to balance the load. FastCGI applications can handle more than one request per process and this makes it easy to cache information pertaining to the request such as results for a given set of values for the request and when pages are reloaded the cached results can be used again.

FastCGI however has some drawbacks. FastCGI multiplexes multiple requests per process and this implies that if a FastCGI application crashes it kills all requests it is processing instead of just killing the one request that caused the crashing of the application. Memory leaks have severe consequences in FastCGI applications because the applications run potentially forever. Thus the memory leaks can lead to instability in FastCGI applications. Developing a FastCGI Web application requires the developer to learn how to write multiplexed FastCGI applications which can be both time-consuming and complex. This is because FastCGI puts the burden of process persistence on the Web application programmer. FastCGI runs one script in each process and thus serving many different scripts requires running many interpreters when using interpreted languages like Perl and this is undesirable under heavy load. It also creates a bias towards some types of monolithic system architectures because implementation is easier when you use the FastCGI libraries. These systems are difficult to change because they typically implement their multiplexing

using the FastCGI libraries and thus cannot be ported easily to other Web application servers.

2.2.3 SpeedyCGI

SpeedyCGI [22] is a utility to improve performance of Perl Web applications by running the Perl interpreter that compiles and executes the Perl scripts persistently (See Figure 2.3). SpeedyCGI conforms to the CGI specification and thus can be used to run Perl scripts that do not make use of SpeedyCGI-Perl libraries. Such scripts however do not run persistently. SpeedyCGI removes the overhead of starting a new script execution environment or process every time that a request that requires the execution of the particular Perl script arrives. By running the processing environment persistently, SpeedyCGI removes the overhead associated with starting the interpreter every time that a request arrives.

To remove the overhead associated with starting a new Perl interpreter every time a request to run a Perl script arrives, SpeedyCGI keeps the interpreter running persistently. Like ordinary Perl during initial execution a new process is created and the script is compiled and executed. However, with SpeedyCGI, during subsequent requests for a particular Perl script the same interpreter is used to handle new executions instead of starting a new process. This means that the Perl process can execute the script right away without re-reading and re-compiling the script, making Perl CGI applications run faster by removing the overhead associated with creating the script's execution environment. Keeping the interpreter persistent makes it possible to cache resources like database connection handles as global variables in the script's execution environment and thus removing the overhead associated with starting a new database connection every time a script that requires the database handle is run.

On arrival of a request a SpeedyCGI front-end sends the request to run the Perl script to the persistent Perl process which would already be running. Each Perl process is run as an independent process and so execution of one Perl script does not interfere with the execution of other Perl scripts. A Perl interpreter also can be used to run more than one Perl script. One of the major strengths of SpeedyCGI is that it does not run Perl code inside the Web server and thus badly written Perl code cannot affect the Web server. This

is because the Perl interpreter runs outside the Web server. This makes the execution environment more secure as failure due to poorly written scripts does not bring down the whole server.

By default SpeedyCGI assigns one or more interpreters to each Perl script to execute the script. This leads to inefficient use of memory when running many different scripts because each of the scripts has its own interpreter. However, SpeedyCGI also offers a capability that groups Perl interpreters and scripts. Scripts in the same group can be run by Perl interpreters which belong to the same group. Thus one Perl interpreter can be used to run different scripts which belong to the same group. With the Perl script and Perl interpreter grouping capability SpeedyCGI helps in creating a secure execution environment by isolating scripts that have a potential security threat by using a separate interpreter to execute them while also providing efficient memory usage by grouping scripts that have no potential security threat. SpeedyCGI however uses the least number of interpreters possible by reusing the same interpreters over and over again even under heavy load.

Unlike CGI, SpeedyCGI is an implementation for a single language. SpeedyCGI works only with Perl script Web applications. Another pitfall of SpeedyCGI is that it can only be run on a local system. SpeedyCGI interpreters use a lot of private memory for each interpreter because they are not forked from a common base interpreter and cannot share pre-compiled code. FastCGI requires each script to run in its own interpreter and this leads to inefficient use of resources when multiple scripts are being served. SpeedyCGI on the other hand has a provision for running different scripts in the same interpreter (See Figure 2.4 and Figure 2.5).

2.3 Web server embedded technologies

2.3.1 Implementing Web applications using a Web server API

Modern Web servers allow developers to have access to the Web server core functionality and structures through an Application Programming Interface (API) [43, 31, 4, 3]. The API makes it possible to develop modules which extend the Web server functionality. The Apache Web server API and architecture makes it possible for a developer to participate

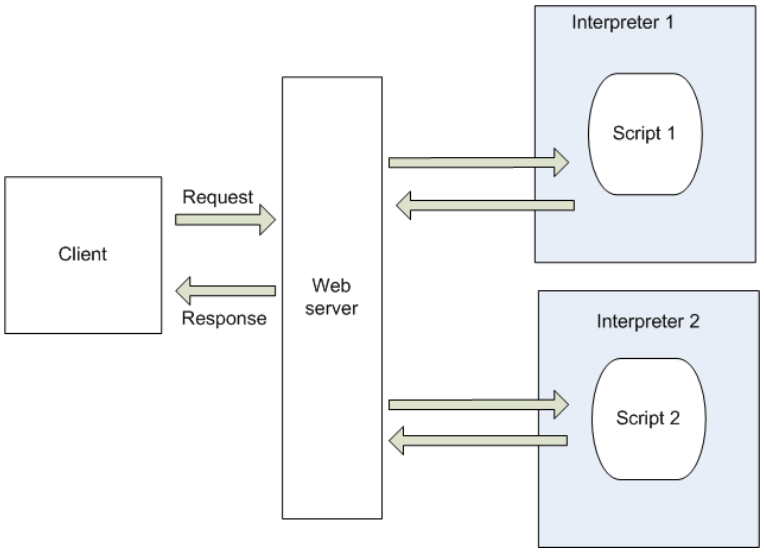

Figure 2.4: FastCGI can only use the interpreter to run one script

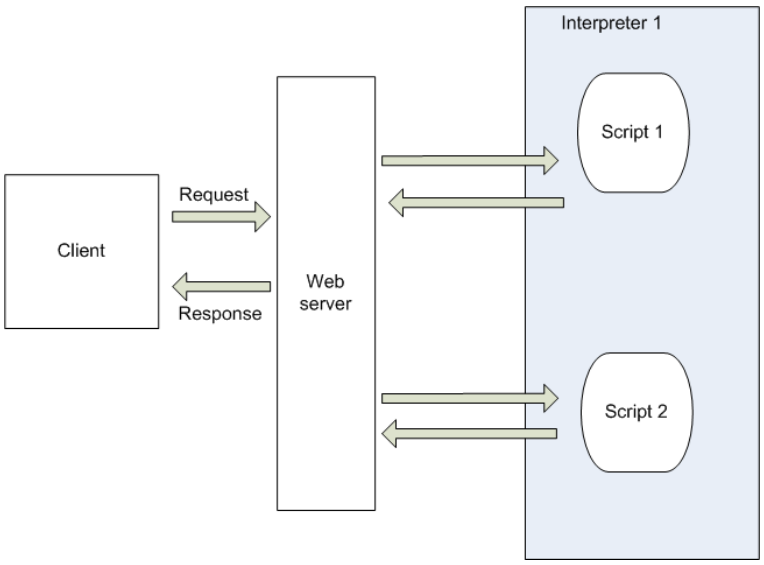

Figure 2.5: Speedy can use the same interpreter to run more than one script

in various stages of request processing [13]. The Apache Web server architecture and API is based on a pool-of-processes Web server model. In this model a Web server has one parent process and a pool of child processes. On arrival, a request is randomly dispatched to an idle child process from the pool for processing. The child process will only accept to process another request after finishing execution of the current request. On startup the server creates a defined number of child processes to handle requests. The Web server creates more child processes when there are no idle child processes in the pool to allocate incoming requests to. The server will only stop creating child processes when it reaches the maximum defined number of child processes.

Developers can use the Web server API to implement a Web application as a module of the server. For example mod_oai [36] is an Apache module that implements the Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH). Apache Web server modules can be either built into the Web server executable at compile time or implemented externally as a dynamic shared object (dso) on UNIX-based Operating Systems or as a dynamic link library (dll) on Microsoft Windows Operating Systems. However, because each child process that handles incoming requests runs independently of the other processes it is difficult to cache global variables associated with the application because the child processes are chosen at random on arrival of a request. Implementing an application with in-memory caching using the Web server API is difficult because incoming requests are randomly allocated to child processes. The Web server would have to keep a copy of a frequently used file in all of the child processes to achieve in-memory caching. Keeping a copy of the file in each of the child processes would mean that when the file is updated all the copies would have to be updated. An alternative would be using a shared memory library like the Linux MM shared memory library [33]. The aim of developing the MM shared memory library was to incorporate it into the Apache Web server so that modules like mod_php and mod_perl could easily use shared memory pools instead of using heaped memory pools.

Some Web server architectures are implemented using a multi-threaded internal structure and API. With multi-threading it is possible to develop a Web server module Web application that handles several requests at the same time. This makes it possible to cache shared information without paying any costs for inter-process communication. However the flaw

with developing Web applications using a multi-threaded Web server API is that multi-threading is compounded by lack of isolation between different applications and between the Web server and the applications. Web server API applications also leave the Web server vulnerable as a bug in the module can bring down the whole server and a bug in the module can compromise the security of the whole server. Implementing Web applications using Web server APIs requires carefully fine-tuning the Web server so that the operating system context switches to the server often enough to address an increasing number of hits. This is because the threads only run during the time slice available to the Web server. In addition Web applications developed using Web server APIs do not port to other Web servers since Web server APIs are usually proprietary. Web server API applications are not a desirable solution in a multi-user environment with users of varying user privileges because they make Web server administration complex if not impossible.

2.3.2 Server Side Includes

Some Web servers like Apache can be configured to support server side includes [26]. Server Side Include (SSI) technology allows a developer to add dynamic content to an HTML page without having to use an external dynamic content-generating technology to generate the whole page. The code for generating the dynamic content is included in the HTML page. The SSI directives are evaluated on the server as the HTML page is being served [19]. The developer adds the dynamic content to the HTML page by placing SSI directives inside the HTML page. SSI technology is a result of an effort to generate Web pages which are mostly static but also have some dynamic content. SSI is however not appropriate for generating dynamic content when most of the page is not static.

Languages like PHP are a result of SSI technology development. PHP [24] is a specialized language whose interpreter is usually compiled into the Web server. It has routines and functionality that allow Web developers to read data, process the data and display the results as part of a dynamically-generated Web page. This makes dynamic content generation easier. For example Web developers do not need to worry about parsing POST or GET headers when using PHP. PHP focuses on generation of dynamic content for display in an HTML Web page. However Web application development is shifting towards the

direction of Web applications which function more or less like traditional applications but can be interfaced with the Web. Such applications are better developed using languages which were developed and are more suited for application development.

2.4 Security in Multi-user Web server environments

Web applications have brought user interactivity to Web pages. The Web applications are driven by parameters which are received via the Web. CGI-based Web applications are the most popular due to their simplicity. The CGI protocol makes it possible for scripts to accept input from Web page forms. By sending the input to the script the Web server passes the responsibility to generate the Web page to an external program. Errors in the CGI script can thus make the host system prone to external attacks.

To reduce the eventuality of an external attack due to poorly written or intentionally harmful scripts, most Web servers are run in a restricted environment. Conventionally, spawned processes run with the same privileges as the parent process that spawns them. Script processes spawned by a server running in a restricted environment are thus restricted to the privileges of the Web server process that spawns them. Web server administrators can thus carefully limit the directory and file permissions of the Web server and reduce the damage that harmful CGI scripts can cause. Most Web servers are run as an un-privileged user without login permissions, such as the user '*nobody*'. The security on the host system can be further enhanced by placing the entire Web server in a restricted directory using the *chroot* command, thereby limiting the processes that it spawns to one part of the file system.

Running the Web server as the unprivileged user '*nobody*' works well in a single user environment. In multiple author environments, such as a Web server run in an academic environment or by an Internet service provider (ISP), a Web server running as the restricted user '*nobody*' compromises the security of the authors' scripts. Scripts running in such an environment run under the same user and are *chrooted* to the same directory hence there is nothing to protect one author's scripts from interfering with other authors' scripts. In such an environment one author can compromise the security of another author inten-

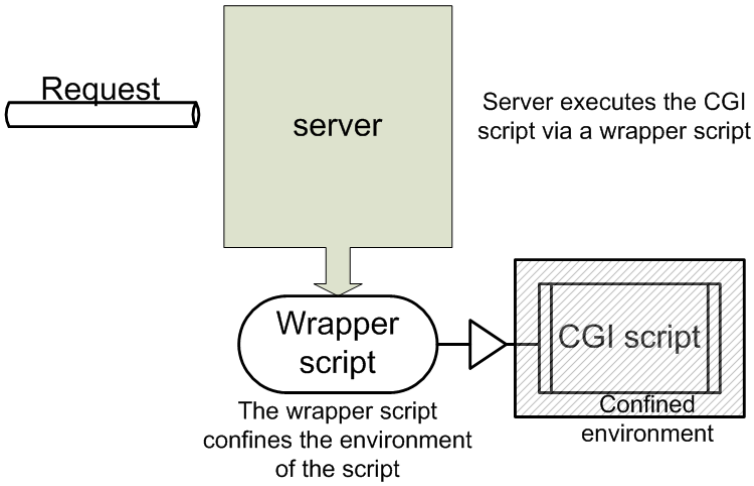

Figure 2.6: Executing the CGI script using a wrapper script makes it possible to confine the environment in which it runs.

tionally or due to a poorly written script. For example one author can have a script that needs to be highly available for corporate reasons. Another author can intentionally write a script that continually forks and this can result in the server's resources being consumed and result in an eventual denial of service attack. Web server administration of such errors is difficult in such an environment as it is difficult to trace an attack back to the owner as all the scripts are executed with same privileges.

To address the problem of security in a multiple author environment, a Web server administrator can have all scripts submitted for evaluation before deploying them but this is however not viable in a busy environment with many authors. Another solution could be pre-installing scripts which the users can link to but this would likely not be popular. One of the viable solutions is using a wrapper script. The Web server uses the wrapper script to run the CGI scripts (See Figure 2.6) instead of running them directly (See Figure 2.7). The wrapper can then be used to enforce security measures in the environment that the CGI script is to be run in.

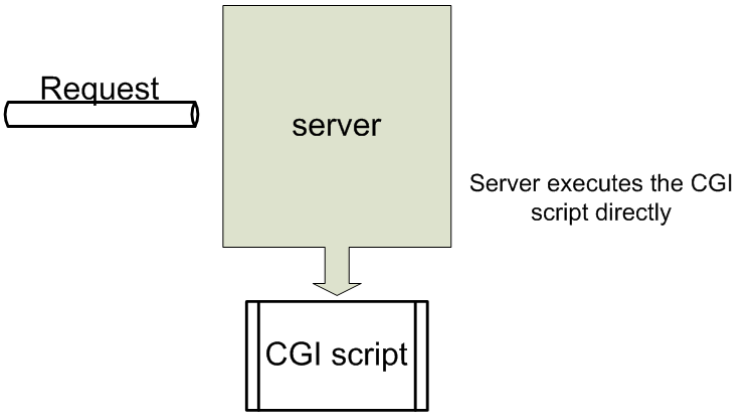

Figure 2.7: Executing the CGI script without a wrapper makes the script run with the privileges of the Web server usually the user nobody.

2.4.1 cgiwrap

cgiwrap [2] is a wrapper program that changes the context under which a CGI script is run from that of the shared Web server account to that of the owner of the script. *cgiwrap* uses the UNIX '*setuid()*' call to run the user-maintained CGI script under the user and group ID of the owner of the CGI script. CGI scripts running under the context of the owner can thus not write data to other Web authors' files. In addition it is possible to trace harmful scripts as they are run under the context of the owner of the script. *cgiwrap* can also prevent denial of service attacks by limiting the amount of resources available to the user-maintained scripts using the Berkeley '*setrlimit()*' call.

cgiwrap runs scripts under the context of the owner and thus the scripts have the privileges of the owner. While this insulates one user form the others it does not insulate the user from him/her self. A poorly written script can be tricked into causing damage to the owner's files such as editing the owner's HTML documents simply because the script has access to the owner's resources.

2.4.2 Apache suEXEC

suEXEC [1] is a wrapper that comes with the Apache Web server. It is used to run scripts under the context of owner of the script and not that of the shared Web server account. In order to invoke a script using *suEXEC*, the Apache server creates a child process that runs the *suEXEC* binary and then passes it the particulars of the script to execute. Before executing the script, *suEXEC* runs a security check to assess if the CGI script request has potential to harm the host system or not. *suEXEC* then loads the script into an edited secure environment using an '*execv()*' call and replaces *suEXEC* itself. Like *cgiwrap*, *suEXEC* also changes its userId and groupId to that of the owner of the script. However *suEXEC* also logs the userIDs together with the groupIds used to execute the scripts making it possible to track attacks. *suEXEC* also does require changing any URLs in order to use it like *cgiwrap*.

During its security check, *suEXEC* edits the available environment variables and reduces them to a list of trusted variables. *suEXEC* can only be executed with the correct number of arguments and should be able to change identity to that of the requested username and group. Using the wrong number of arguments implies an attempted system attack and *suEXEC* simply logs an error and does not run the script. The username/ID invoking *suEXEC* must be valid and listed in the '*/etc/passwd*' file and the requested script cannot be run under the context of the '*root*' user. *suEXEC* can only be executed under the username that was compiled into it when it was built and the requested script cannot be an absolute file system path. A script that is not from the ∼*username* (URL pattern) directory must be under the DOC_ROOT specified during the configuration of *suEXEC*. This ensures that all scripts that are executed using *suEXEC* are owned by a user with an account on the server or are owned by a user who has access to the Web server's DOC_ROOT. The requested script and the directory it belongs to should have file permissions to write to the group it belongs to and should be owned by the group and user under which it is to be executed. The requested script and the directory it belongs to should exist and *suEXEC* should be able to '*chdir()*' to this directory.

suEXEC's control parameters can only be changed at compile time and changing the parameters can only be done by recompiling the Apache Web server. A single misconfiguration

of *suEXEC* can be potentially harmful to all the CGI scripts being run by the server. While *suEXEC* works by insulating one author from another it reduces the security of the authors in their own environments. The script is run under the context of the owner and a poorly written script can be tricked into deleting the owner's home directory because it has the same privileges as the owner.

2.4.3 suPHP

suPHP [28] is a wrapper for executing PHP scripts under the context of the owner of the PHP script being executed. It has an Apache module (mod_suphp) that invokes a *setuid-root* binary (*suPHP*) to change the uid of the process executing the interpreter to the uid of the owner of the script. uid is short form for User ID and is the means by which a user is identified to various parts of Linux. Using *suPHP* limits the privileges of the executed script and the interpreter to that of the owner of the script being executed. *suPHP* is however a tool that was designed and implemented for use with PHP scripts only.

2.4.4 sbox

sbox [44] is a wrapper script that is used to change process privileges of CGI scripts from that of the shared Web server account to that of the owner of the script. *sbox* limits the amount of resources available to the CGI script being run and thereby makes it possible to prevent denial of service attacks on the host system.

sbox runs the script with the privileges of the owner of the script or the owner of the directory in which the script resides. *sbox* also has options for running scripts with the privileges of the group that owns the script or the group that owns the directory in which the script resides. Unlike *suEXEC* and *cgiwrap*, *sbox* also performs consistency checks on the script file and directory ownerships to ensure that the script being executed is not world writable to prevent potential damage to the script and/or the files in the directory in which the script resides. *sbox* establishes and sets limits on the script's use of the CPU and memory and runs the script in a *chroot-ed* directory located within the home directory of the owner of the script to prevent denial of service attacks. The use of *sbox* can be made transparent when used with the Apache Web server's mod_rewrite. Using Apache's

mod_rewrite makes it possible to remove the wrapper's name from the path of the requested script by translating or rewriting URIs of requests.

sbox performs a series of consistency checks to make sure that it is not being tricked into running a process that might compromise the security on the host system. For example, in its consistency checks, *sbox* ensures that it is not run as a special user such as the bin or root user to increase the security on the host system. *sbox* consistency checks ensure a secure execution environment that insulates the author from other authors and also ensures that the security of the author who owns the script that is being executed is not compromised. *sbox* thus allows for a secure environment in which un-trusted users are allowed to upload user maintained scripts onto the server because *sbox* runs the scripts in a sandbox. The sandbox acts as a form of script isolation. The isolated script is run in an environment where it can cause least or no damage at all to the host system.

2.5 Java Servlet technology

Java servlet technology [37] is a server side technology that is used to reflect the state of the client and the server based on a client request and server response. The servlets work as a type of extension to the Web server functionality and are simply Java objects that implement the javax.servlet.Servlet interface. The servlets plug into the existing server and process a request on behalf of the server by performing the work of an application. The Java Servlet technology is a protocol and platform-independent technology that is used to build, distribute and install binary Web applications that can be hosted on any Operating System.

Servlets communicate with the Web server using the Java Servlet API that defines the link between the servlets and the hosting system. Servlets receive requests from a client's Web browser over the HTTP protocol via a server (servlet server) that implements the Java Servlet API. Most servlet servers are can also serve standard Web pages but they do not have the range of functionality and efficiency that full Web servers like Apache offer. Likewise Web servers like Apache cannot provide the facilities that dedicated servlet servers like Apache Tomcat offer. Servlets are thus usually served by a servlet server that is

configured to receive requests via a full Web server like Apache. The dedicated Web server is configured to direct servlet requests to the servlet server.

2.5.1 Web application component architecture

Web applications developed using the Java servlet technology are distributed in the form of a directory structure which is packaged into a '.war' file. The war file is a single archive file which is used to distribute and install the application on a servlet server. The directory structure is a single directory that contains all the Java class files, libraries, Java Server Pages (JSPs), HTML pages, data files and configuration information in their correct format and structure. Java Web applications distributed using the standard .war file can be installed and served by any standard servlet server on any platform. The .war file is copied to the Web application directory of the server and the server will unpack it, load the servlets and start serving them. The Java Web application architecture arguably makes Web application installation easier and more convenient.
The directory structure has one extra directory at the top level called *WEB-INF*. The *WEB-INF* directory is protected from Web access and contains the servlet classes, servlet configuration information and libraries that must be loaded and be available for the servlets to work. The *WEB-INF* directory also contains any other data files or information required for execution of the servlets.

2.5.2 Java Server Pages

JSPs [14] are HTML pages which have embedded Java statements in the HTML code. JSPs remove the difficulty of embedding HTML statements in the servlet. This helps separate program logic from Web page design. Developers can thus design the Web page using any Web page design tools and then program the servlets independently and embed calls to the servlet into the HTML code. The servlet server translates the JSP into a standard servlet with embedded HTML statements and all the necessary escaping of characters the first time that it loads the JSP. Many technologies have been developed which make embedding of Java code into the HTML code easier and more secure.

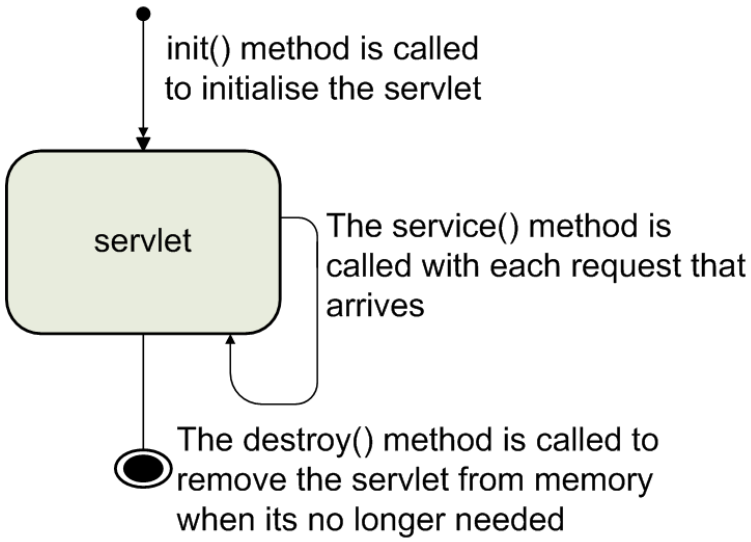

Figure 2.8: A servlet is loaded into memory using the *init()* method. Once loaded each request is processed by calling the *service()* method call. Finally the servlet is unloaded from memory using the *destroy()* method call.

2.5.3 Servlet lifetime

One of the strengths of the Java servlet technology is the fact a servlet can be loaded once into memory and then be reused several times to service requests. The servlet is then unloaded from memory when it is no longer needed. This is achieved by using the *init()*, *service()* and *destroy()* method calls (See Figure 2.8) which are discussed in the following sections.

init()

The *init()* method call is used to load the servlet into memory. It is passed a ServletConfig parameter that has variables from the deployment descriptor that are used to initialize the servlet. The ServletConfig attribute is an important attribute since servlets have no main method or constructor method and thus it is the attribute that is used to pass the various initialization parameters to the servlet.

service()

The *service()* method call is called every time that a new request arrives. It is passed two main parameters, the *ServletResponse* and *ServletRequest* objects. The *ServletResponse* object is used to initialize an output stream for the servlet's response and outputting the response while the *ServletRequest* object is used to initialize an input stream that is used to read in the information that is required to process the request.

destroy()

Finally, when the servlet is no longer needed the *destroy()* method is called to remove the servlet from memory. It is called once the servlet is garbage-collected and is used to release any resources like files that the servlet was using for processing its requests.

2.5.4 Java Web application security

Like all other Web application technologies security is also a major concern with Java servlets. A servlet with enough privileges can perform a function such as '*System.exit()*' which can exit the servlet server process causing the server to stop. A servlet having network access can open a port for data reading and writing which can lead to potential damage to the host system. Servlets use access control lists and the servlet sandboxes to increase security on the host system (See Figure 2.9).

Servlet sandbox

Servlets can cause potential damage to the host system if not properly managed. Servlets can be obtained from various sources, both reliable and non-reliable. The servlet could have been developed by the Web administrator or could have been downloaded from the Internet. Servlets can be associated with different levels of trust depending on where they were sourced from. Servlet servers can associate different levels of trust to different servlets using a servlet sandbox [45]. A servlet sandbox is an environment of execution where servlets are given restricted and defined authority on the server. The sandbox determines how much access the servlet has to system resources such as the file system, network and

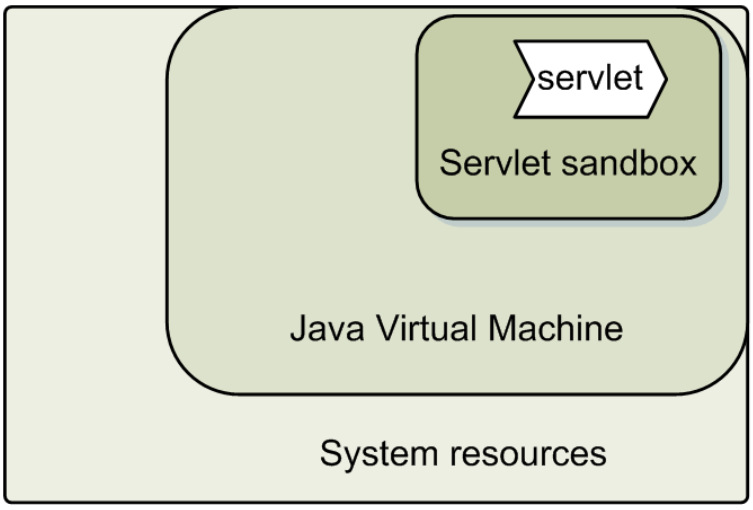

Figure 2.9: The servlet's execution environment is controlled using a servlet sandbox

writing authority. The amounts of resources that are granted in the sandbox are defined by the Web administrator.

Access control lists (ACLs)

Access control lists are a way of restricting access to servlets by defining what kind of access is allowed, to what objects the access applies and which users are granted access. The ACL is specified by the server and used with either server-defined names or server defined user groups.

2.6 Server performance optimization techniques

The growth and widespread use of the Web has led to performance degradation due to congestion and server overload. The developer managing the Web application has little control over the congestion of the Internet while on the other hand the server load can be prepared for and forecast during implementation. One of the major causes of server overload is the performance of the Web application server. For this reason, the Web server needs to be tuned in terms of performance. Web servers can serve hundreds or even

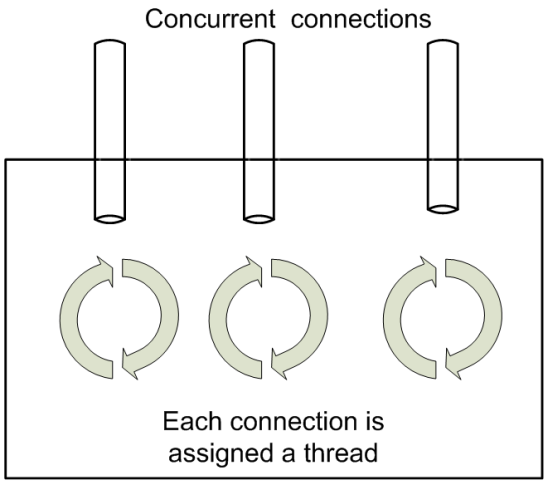

Figure 2.10: Each request is assigned a single thread to process it. The connections on the threads are in a blocking mode.

thousands of connections concurrently and thus one connection's IO blocking can degrade the performance of the server. There are several architectures for managing concurrency on Web servers. The main levels of classification are event-driven, thread-per-connection and the hybrid which utilizes both threads and event-driven approaches.

2.6.1 Thread based servers

In thread based servers, a request is assigned to a thread that handles its IO (See Figure 2.10). Thus when the IO for one thread blocks, other threads can still execute. Thread based servers rely on either the context switching of kernel threads by the Operating System or thread libraries to do the context switching when IO blocking occurs. However context switching and contention for locks leads to performance degradation in architectures that require many threads [38]. For this reason the number of threads is restricted in most of such architectures thereby restricting the number of concurrent connections.

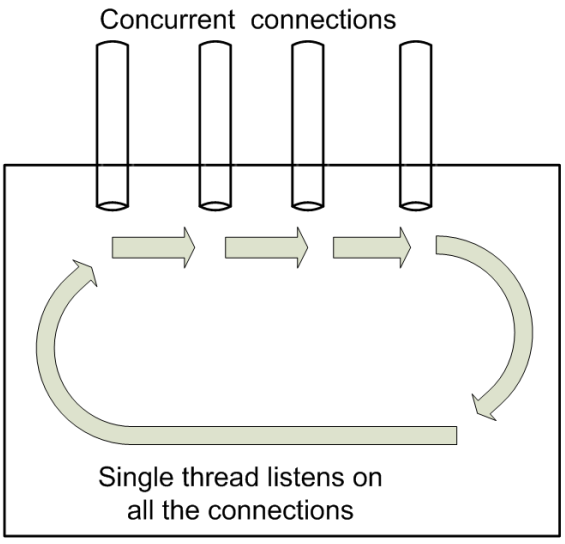

Figure 2.11: One single thread is used to multiplex all the requests. The connections are in non-blocking mode

2.6.2 Event driven servers

In this architecture, a single process multiplexes all the connections by having all the sockets in a non blocking mode and issuing system calls only on those sockets that have events on them and are thus either ready to be read from or written to. The single thread uses system calls like *kqueue*, *epoll*, *poll* and *select* to identify sockets that are ready for reading or writing. However this architecture does not cater for blocking IO system calls such as reading a disk file. Thus it is not appropriate for connections which involve a lot of disk access (See Figure 2.11).

2.6.3 Hybrid servers

To overcome the failures of the two architectures discussed in the previous sections, hybrid architectures use multiple threads and each thread also multiplexes multiple connections. Thus connections are classified into groups and each group is handled by a single thread (See Figure 2.12).

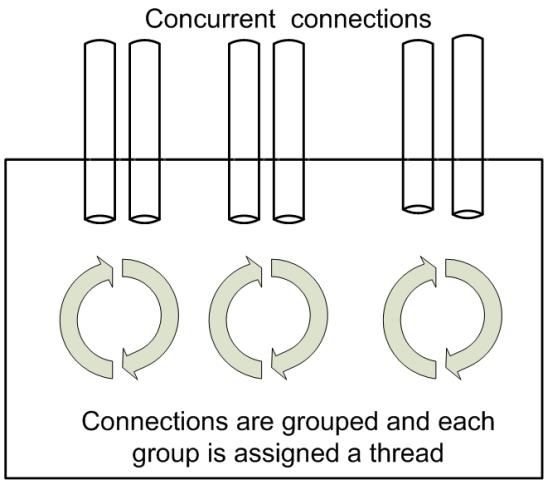

Figure 2.12: Multiple requests are assigned to a single thread. Multiple threads are used. The connections are in a non-blocking mode

2.6.4 Design considerations

Server performance can be improved and enhanced by choosing the correct architecture for concurrency. According to Lauer et al [25], the performance of threaded and event driven architectures is the same. However, von Behren et al [46] argue that the event driven architecture is bad for high-concurrency servers. They argue that threads are a more natural and simpler style for programming. They also claim that threads can achieve all the strengths of events including support for high concurrency, low overhead and a simple concurrency model. In contrast to the view that events are a bad idea, Ousterhout [38] recommends use of events. Ousterhout suggests that threads be used only when true CPU concurrency is needed. In addition, Bradel and Drula [11] also recommend the use of events over threads. They claim that the two architectures have similar performance and speculate that events are a more natural representation for complicated designs. Recent studies by Pariag et al [39] reveal that in their experiments the event driven server and hybrid based server achieved up to 18% higher throughput than the best implementation of the thread based server. In a detailed analysis of their experiment they revealed that the throughput of the thread based server was hampered by overheads (6% to 10% of available

CPU time) in the Capriccio thread library. They claim that user level thread libraries like Capriccio are built on an event driven foundation. Thus the user level thread library also incurs overheads associated with event driven applications in addition to the threading layer context-switching, scheduling and synchronization.

2.7 Frameworks for easy Web application deployment

Web applications have varying complexity ranging from trivial to large and critical Web applications. Most of the Web applications are packaged for ready deployment. Deploying Web applications on a dedicated server requires the customization of the server configuration files. However the deployment of a Web application on a shared hosting environment or multiple deployment environment is more complicated. In addition developing Web applications for distribution makes the process of packaging for easy deployment even more complicated. Developers of the Web application try to make the process of deployment as easy as possible. Many frameworks have been developed to try to make the process of packaging and deploying a Web application as easy as possible. The following sections outline some of the techniques for Web application deployment.

2.7.1 Using Apache's .htaccess

There are many ways of configuring directory access when using the Apache Web server to host your Web applications. The Apache '*.htaccess*' [15] can be used to configure directory configurations for deploying a Web application. The Apache Web server on which the Web application will be deployed should however have similar configurations to the configurations of the Web server on which the Web application was developed. Using the *.htaccess* configuration file allows for deploying a Web application without having access to the configuration files of the Web server on which the Web application is being deployed. The *.htaccess* file is packaged and distributed together with the Web application. This makes it possible to package a Web application for distribution without worrying about the Web application failing after deploying it into a different environment. The drawback with this method is the fact that the method is limited to the Apache Web server let alone the fact

that the configurations of the Web server on which the Web application will be deployed need to be similar to the configurations of the Web server on which the Web application was developed.

2.7.2 Python Paste deployment

Paste Deployment [10] is a system for finding and configuring Python Web Server Gateway Interface (WSGI) [9] Web applications and servers. It uses a simple function to load Python WSGI Web applications from a configuration file or Python egg. In order to deploy the Web application, the user has to load the Web application using the URI of the *config.ini* file. The *config.ini* file or Python egg contains the filenames of the scripts and also the names of the *config.ini* files of the sub-applications that compose the application being deployed. For example consider the following portion of a sample config file:

[composite:main]

use = egg:Paste#urlmap

/ = home

/blog = blog /cms = config:cms.ini

The composite key word means that the requests for the given path prefixes will be forwarded to other Web applications. The Web applications are addressed using a filename or another *config.ini* file. Paste Deployment also can be used to build a Web application from pre-existing components. Users can deploy Python WSGI Web applications without having much knowledge about the WSGI interface. Thus with Paste Deployment the user does not have to go into the Web server config files to configure the Web application - all they need is to load the config file or Python egg that came with the Web application using the Paste Deployment system. Paste Deployemnt comes with many options which allow a user to install more than one Web application using one config file. In order to use Python Deployment the server administrator needs to install Python Paste. The major drawback is that Python Paste is used to deploy Python WSGI Web applications only.

2.7.3 WebLogic Server Deployment

WebLogic server deployment [42] extends the J2EE 1.4 deployment API specification. To make the process of deployment easier, WebLogic comes with deployment tools which include the weblogic.Deployer, Administrative Console and WebLogic Scripting Tool (WSLT).

weblogic.Deployer

The weblogic.Deployer tool provides a commandline interface for the WebLogic deployment functionality. It supports some functions which cannot be performed using the Administrative console.

Administrative Console

The Administrative Console is a Web-based deployment assistant that can be used in the deployment process. It also allows accessing and changing the values of the deployment descriptor while the deployment unit is running.

WSLT

The WSLT is a commandline tool that is used to automate application deployment configuration and deployment operations.
The WebLogic server uses a deployment descriptor *web.xml* to deploy Web applications. The *web.xml* file is a standard configuration of the .war file that is used to configure WebLogic server specific configurations for the Web application. The Web application is packaged as a .war file ready for deployment on any Web application server that implements the J2EE deployment API specification. However WebLogic server deployment is specifically for Java Web applications. Other servlet containers like Apache Tomcat, JBoss and Jetty also implement and extend the J2EE deployment specification. Apache Tomcat also has a manager interface which can be used to deploy applications via the Web.

2.7.4 JBoss Deployer Architecture

JBoss deployer architecture is based on the JBoss JMX microkernel [17]. It has an extensible architecture that allows incorporation of components into the microkernel. It uses a main deployer called the 'MainDeployer' that is called every time there is a component to deploy into JBoss. The MainDeployer then checks to see if there is a subdeployer that can handle the deployment and delegates the deployment to the subdeployer. JBoss comes with extensive subdeployers like the JARDeployer which is used to deploy files that end with a '.jar' file extension. It also has the AbstractWebDeployer that is used to deploy '.war' files. The .war files must have a WEB-INF/web.xml deploy descriptor. The subdeployers are a mechanism that JBoss uses to load the classes and libraries of the application. When the MainDeployer receives a deployment request it iterates through its list of subdeployers and invokes the accept(DeploymentInfo) method of the subdeployers. The first subdeployer to return true is chosen. The MainDeployer then delegates the init, create, start, stop and destroy deployment life cycle operations to the subdeployer.

2.7.5 Tomcat Deployer

Web applications can be deployed into Apache Tomcat either statically or dynamically. Apache Tomcat [27] has to be restarted when an application is deployed statically into it. Apache Tomcat deploys all .war files located in its 'webapps' directory on startup. Dynamic deployment allows applications to be deployed into a running Apache Tomcat container without the need for a restart. The applications are deployed as .war files into the Apache Tomcat server. Dynamic deployment can be done using either the Tomcat Manager or Client Deployer Package.

Tomcat Manager

Tomcat Manager is a Web application that can be use to remotely or locally install applications in Apache Tomcat. It has methods for deploying, undeploying, reloading, stoping and starting Web applications in Apache Tomcat. It can be used with either the manager HTML interface or by sending HTTP GET requests directly.

The Client Deployer Package

The client deployer package allows the use to validate, compile, compress to .war, and deploy the application onto a running development or production Apache Tomcat server. The target Apache Tomcat server should however be running in order to use this package.

2.7.6 Wombat Web application deployment

Wombat [32] is a servlet container for Perl that is inspired by the Java Servlet 2.3 Specification. A Wombat Web application directory structure follows the J2EE Servlet specification. The Web application is deployed using a *web.xml* deployment descriptor. The *web.xml* deployment descriptor defines the configuration of the Web application. Libraries that the Web application may require are packaged in the lib directory of the Web application under the *WEB_INF* directory and are added to Perl's *@INC* when the application is loaded so that application classes are visible. Wombat demonstrates the implementation of the J2EE servlet specification on a language other than Java.

2.8 Summary

The qualities of a universal Web application server are process persistence with efficient resource usage and performance, a secure environment for multiple authors and support for multiple Web application development languages. CGI has no process persistence and hence does not meet the requirements of a universal Web application server. SpeedyCGI has no support for multiple development languages as it only supports Perl scripts and Java servlet technology is limited to Java technology and thus both do not meet the requirements of a universal Web application server. A possible solution could be using FastCGI with a wrapper like sbox. However FastCGI runs one script per execution per interpreter when used with interpreted languages. Thus in a multi-user environment serving an arbitrary number of authors each having an arbitrary number of scripts the use of FastCGI with sbox could result in an inefficient use of resources because each script for each user would require its own interpreter. FastCGI also does not support Web application packaging. A universal Web application server should be able to allocate more than one script to an

execution environment for efficient resource usage. Using Web server APIs is not viable for a universal Web application server because this solution does not scale easily and thus would result in performance degradation as the request load increases. SSIs are meant more for dynamic Web pages than for Web applications. None of the technologies discussed so far meet the criteria of a universal Web application server, although they each have some of the desirable characteristics.

In like manner, most of the effort at making Web application deployment easier is language-centric. Wombat on the other hand demonstrates that the servlet specification can be extended and used with other Web application technologies. The use of an XML file for defining the configuration of the Web application for deployment is more viable than using a file like Apache's *.htaccess* file which is more server-dependent. Also defining directory structures makes it possible to add libraries which can be loaded once into the interpreter to reduce the overhead associated with loading the application into the interpreter. This is again demonstrated by Wombat.

Chapter 3

X-Switch prototype server

3.1 Overview

This chapter discusses the prototype of the universal Web server developed by Maunder et al [29] in partial fulfillment of their Honours degrees. The chapter also outlines the design motivation, evaluation results, conclusions and proposed future work of their research.

3.2 Introduction

In their research, Maunder et al [29] revealed that almost all major South African Web hosts supported more than one Web application technology. Amongst the technologies supported were PHP, Microsoft's ASP, CGI-scripts and Sun Microsystems' Java servlets. They also stated that despite the variety of technologies, PHP, ASP and Java servlets were the most predominant Web applications and that no single Web server implementation provided concurrent support for all these Web application technologies. The Web hosts ran each technology on separate server machines and redirected requests for a particular technology to the corresponding server that implemented the technology. In an effort to implement a Web server that had support for multiple development languages in a shared environment, they proposed the X-Switch prototype.

3.3 Design and implementation

3.3.1 Design motivation

The design of the X-Switch prototype was motivated by the need for the multi-user and multi-language requirements of a conceptual universal Web server. The following requirements were taken into consideration while designing the X-Switch prototype.

1. A Web application server that supports multiple implementation languages without integrating them into the core system of the Web application server.
2. A Web application server that can perform user/group context switching.
3. Simplifying processing engines so as to have lightweight components that still retain the necessary functionality.
4. A Web application server into which existing Web applications can be deployed without having to recode the Web application or using complex descriptors.
5. Developing a Web application server that can perform comparably with industry-grade single user Web application servers.

3.3.2 The X-Switch system

In order to implement a multi-development language platform the X-Switch prototype design took a modular approach. It used lightweight engines which each supported a single language for processing requests. The engines were almost completely decoupled from the core system except for a request message that was passed between the core system and the component engine.

The X-Switch prototype component engines used persistent interpreters for efficient request processing. This was inspired by FastCGI [21] and SpeedyCGI [22] which both demonstrate that using persistent processes helps in achieving significant performance improvements.

For security in the multi-user environment, the X-Switch prototype used a X-Switch wrapper script '*suexecme*' to execute the processing engines in a controlled and audited environment. Each user that had scripts installed on the server was provided with processing

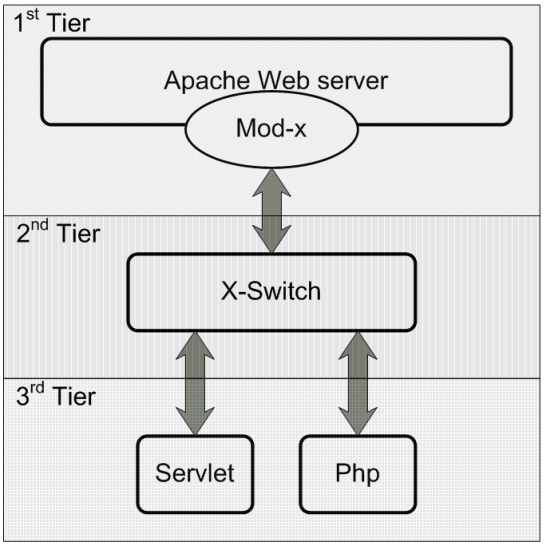

Figure 3.1: X-Switch1.0: Modular design of the universal Web server protype

engines that executed the user's scripts.
The modular design had three main levels of processing as shown in Figure 3.1.

The Web server module (Mod_x)

In their design Mod_x was implemented as a simple module that routed the URL for the request to the core X-Switch system. It did not do any header passing and only handled GET requests. It was implemented as a Dynamic Shared Object (DSO) that gets called and included in the Apache core during the Web server startup phase via the DSO interface. The module utilized two main features of the Apache API:

1. **Custom handlers** The Apache core and API allows users to register handler routines that are called whenever the Apache Web server receives a request that it can associate to the particular handler. In the design of the universal Web server each language installed in the X-Switch prototype, e.g. Java servlets, had a handler defined in Mod_x. Thus for each processing language Apache called a handler that was defined for that particular language.

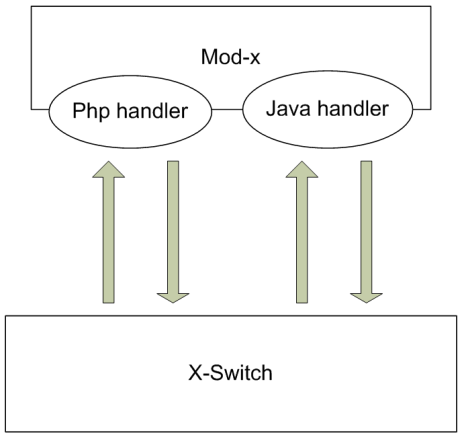

Figure 3.2: Mod_x had handlers for each language type. It established a TCP socket connection with X-Switch for sending request processing information and closed it after receiving all of the response from X-Switch

2. **Configuration directives** An Apache module can register a per-URI configuration directive that matches a string pattern. Apache then analyzes requests and calls the handler when it encounters the string pattern. In the design of the X-Switch prototype, the specific handlers each registered a string pattern. For example the PHP handler registered a '/php/' string pattern. Thus when Apache encountered a request URI which had a '/php/' string pattern, it called the PHP handler for Mod_x. Mod_x then established a TCP socket connection with X-Switch and routed the request URI to the Switch main module. The connection was closed after receiving the response from X-Switch module (See Figure 3.2).

X-Switch main module

The X-Switch main module (X-Switch) was responsible for routing requests between the processing engines and Mod_x. It also was responsible for running and managing the engines. In this implementation of the universal Web server the X-Switch module was responsible for interpreting the requests. It had the supported engines and execution paths to the processing engines hard-coded into the main module. Upon determining the author of the script being requested, X-Switch then spawned a processing engine and started

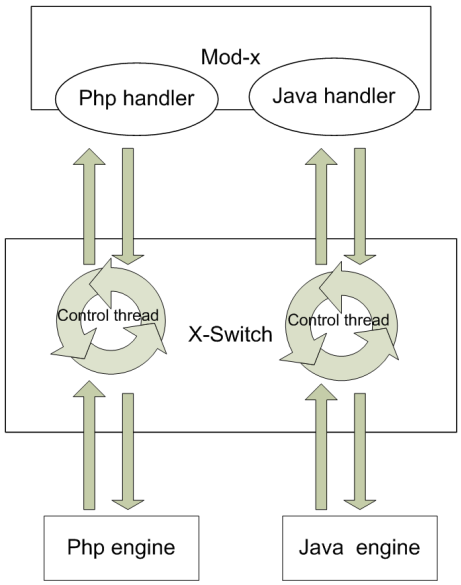

Figure 3.3: Each processing engine is assigned a control thread that is responsible for communication between the processing engine and X-Switch

a thread to handle communication between X-Switch and the processing engine. The main module only handled linear requests and had no mechanism for killing engines when they were no longer needed and starting up extra engines when request loads went up. Processing engine execution was done via a wrapper script, *suexecme*. *suexecme* was a X-Switch wrapper script that executed the processing engine with the privileges of the owner of the Web script that the engine was to run. The X-Switch module assigned a thread to each processing engine (See Figure 3.3). The X-Switch engine communicated with the processing engine using standard input and output. The request information was sent to the processing engine via standard output and the response was read from standard input. The communication protocol used the '$' char to signal the end of the output from the processing engine. X-Switch then closed the TCP socket with the Mod_x handler that sent the request.

Processing engines

The initial implementation had two sample processing engines. The engines only had support for simple HTTP GET requests. The engines were implemented as persistent processes and communicated with the X-Switch module via standard output and standard input. The following engines were implemented

- **PHP processing engine**-The PHP processing engine was implemented as a lightweight processing engine that used the PHP commandline SAPI. It had no support for any headers. It however had support for parameterized GET requests.
- **Servlet processing engine**-The servlet processing engine, like the PHP processing engine, was lightweight and only had support for HTTP GET requests. It implemented a threaded model for handling the requests. It did not implement the servlet API for basic servlet functionality.

3.4 Evaluation

The evaluation of the X-Switch prototype proved that the prototype had performance that was comparable to industry standard Web application servers [30]. The performance of the universal Web server was better than that of CGI. In the realistic scenario which was carried out with the servlet engine, the results revealed that the performance of the X-Switch prototype in combination with the X-Switch servlet engine was comparable with known servlet engines.

3.5 Summary

The implementation of the universal Web server works for trivial cases. Evaluation studies which were carried out prove that the performance of the universal Web server is acceptable in the limited tests that where conducted. The universal Web server however did not have most of the basic functionality that comes with a Web application server. The universal Web server prototype did not support multiple connections. It also did not address Web

application packaging. The following chapters outline my contribution to the universal Web application server research project and modifications made in trying to meet the basic needs of a Web application server. Ongoing research on architectures for concurrency reveals that the universal Web server can benefit from a single threaded architecture for concurrency, as discussed in the preceding chapter. Factors like servicing concurrent connections, load management and server information management are taken into account. This and other factors are considered in the current design of the universal Web application server and are discussed in the following chapter.

Chapter 4

Design and Implementation

4.1 Introduction

The previous chapter presented the first prototype of the universal Web application server, X-Switch-1.0, while the current chapter will present my contribution to a more complete and robust universal Web application server, X-Switch-1.1.

4.2 Design overview

X-Switch-1.1 (See Figure 4.1) follows the modular design of X-Switch-1.0. It has three main levels of processing. The Apache Web server module, Mod_x, receives requests from the Apache Web server and forwards them to X-Switch. Mod_x implements a single handler that is used to handle all request types. Upon receiving a request it establishes a socket connection with the X-Switch module. It uses this socket to send the headers and any other information needed for request processing to the X-Switch module. The X-Switch module then determines the owner of the Web application that is required to process the request. If no engine that is required to process the request exists, X-Switch then spawns an engine to process the request. X-Switch establishes two socket connections with the processing engine: the control socket and the input-output socket. The control socket is used for transmitting the request information to the processing engine and also for receiving the signal for the completion of request processing by the processing engine. The input-output socket is used for sending the request body data, in the case of a POST request, to the engine's standard input and also for reading the response from the engines standard output. Upon establishing the connection, X-Switch sends the request processing information to the processing engine and relays the response from the processing engine to Mod_x. Mod_x

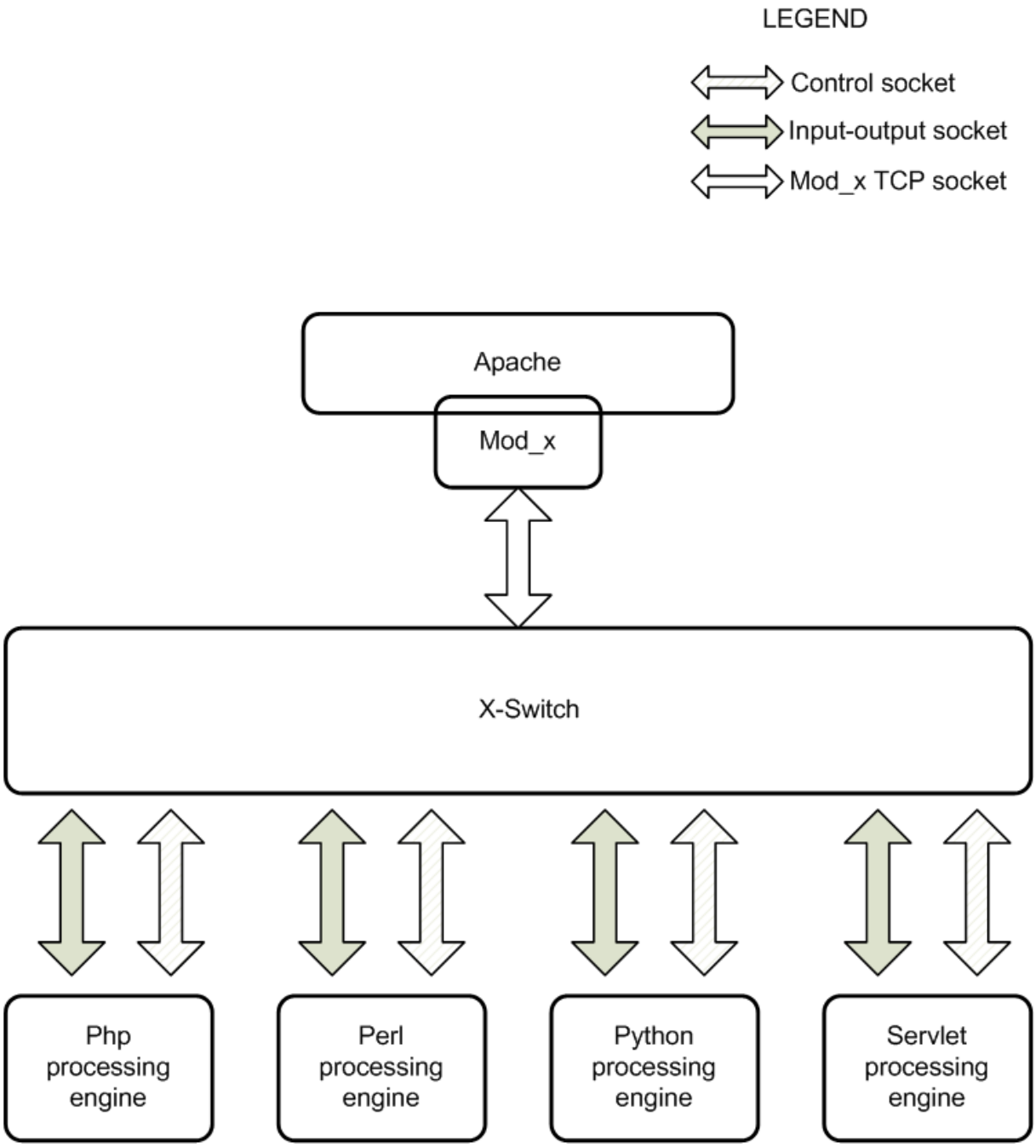

Figure 4.1: Modular design of X-Switch-1.1

then routes the response to the client that made the request (See Figure 4.2).

4.2.1 Advantages of the modular design

The modular design of the universal Web application server has the following advantages:

1. **Web server independence of the core functionality**
   The modular design of the universal Web application server ensures the separation of the core functionality of the universal Web application server from Mod_x. This makes it easier to implement different Web server module front ends for the universal Web application server, thus reducing the dependence of the universal Web application server core functionality on the Web server front end. This essentially makes the universal Web server less coupled to the Web server, in this case Apache.

2. **Ease of maintenance**
   The Mod_x module implements less of the functionality of the X-Switch system. Thus changes in the Web server or its API would result in minor changes in the universal Web application server as a whole because the consequential changes would only have to be made in Mod_x. The universal Web application server can be implemented with different Web server module front ends. A Web server that has an API, that can be used to access the Web server's core functionality, can be used to implement Mod_x. Thus it is possible to have more than one kind of Web server module that interfaces with the X-Switch system. In such a scenario removing the core functionality from Mod_x makes maintenance easier. This is because changes to the core functionality of the universal Web application server would have to be made to X-Switch only, preventing a situation where all the Web server modules would have to be updated. The modular architecture also makes it easier to incorporate other processing engines into the X-Switch system at later stages.

3. **Web server stability**
   Though Web servers are designed to be stable and robust, failure in Mod_x can potentially result in the whole server going down. Thus Mod_x is designed to be as simple as possible, shifting the core functionality to X-Switch. Such a modular design

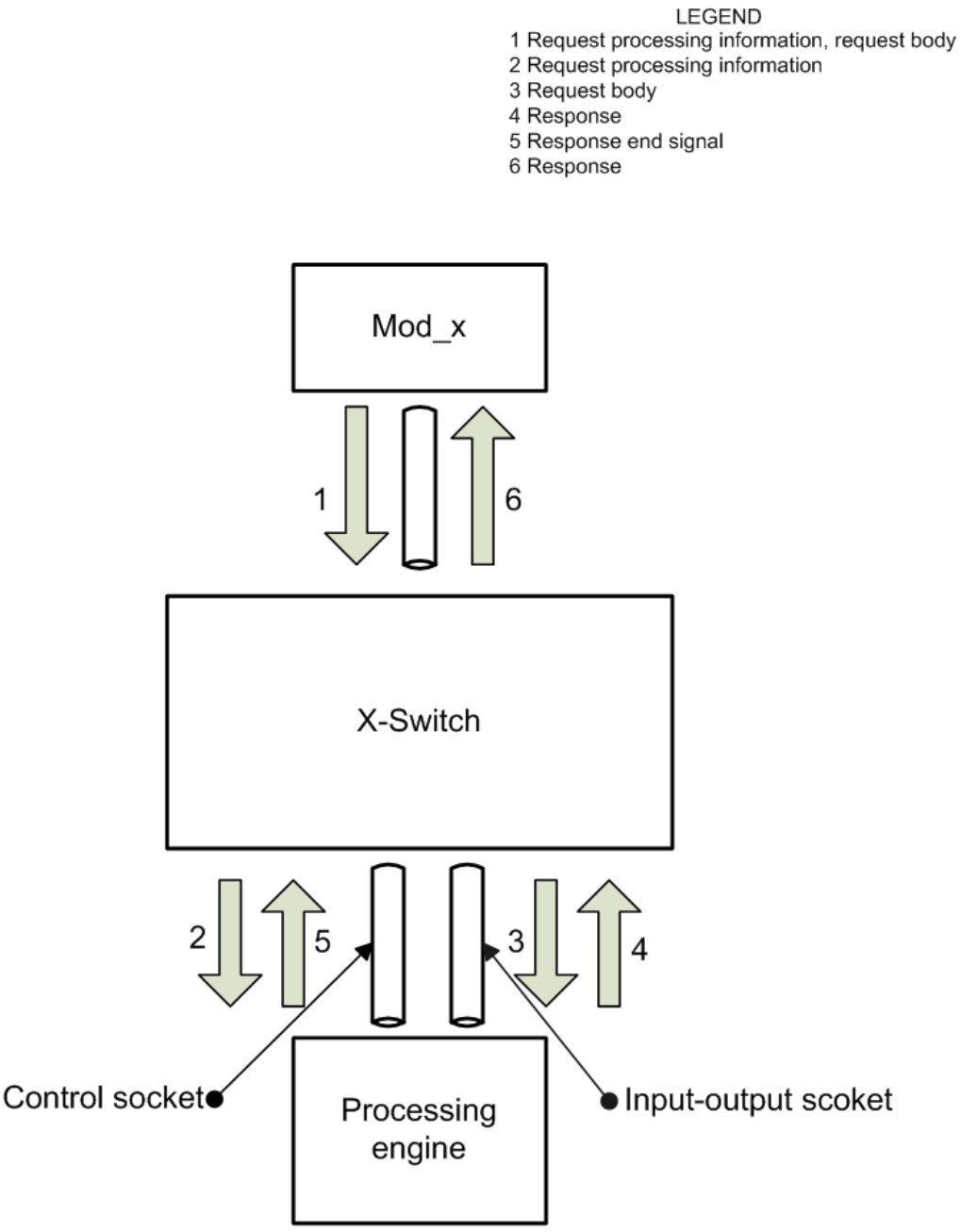

Figure 4.2: Information flow in X-Switch-1.1

helps reduce potential failure of Mod_x and the Web server. Failure of the main X-Switch module would leave the Web server still servicing other requests. On the lower level the separation of the processing engines from the main X-Switch module implies that failure in the processing engine does not stop the X-Switch system from servicing other requests. This ensures that failure of a corrupt application for one author does not cause the failure or non-availability of another author's Web applications.

The details of the design of each of the levels of processing are presented in later sections of this chapter.

4.3 X-Switch system communication protocol

The X-Switch system has three layers of processing. In order for the various layers of processing to communicate successfully and efficiently, the X-Switch system uses defined communication protocols between the layers. The use of defined protocols also makes it possible to evolve each of the layers of processing independently of one another.

4.3.1 Mod_x /X-Switch communication protocol

To achieve the use of different kinds of Web server modules developed using different kinds of Web server APIs, the X-Switch system defines a simple protocol for communication between Mod_x and X-Switch.

Communication between Mod_x and X-Switch begins when Mod_x establishes a TCP socket connection with X-Switch. X-Switch then receives the information required to process the request using the established TCP socket connection. The information is received as four separate lines of text from Mod_x. The first line contains the method type of the request as well as the content length of the request. X-Switch uses the content length and method type to determine whether or not to read in request body data and the amount of data to read in. The second line that Mod_x sends to X-Switch contains the filename of the request as well as the arguments that came as part of the URL for the request. The third line is a string of '&x' token separated headers that are required to process the request. The last line is the default engine type required to process the request. After reading the last line, X-Switch reads the

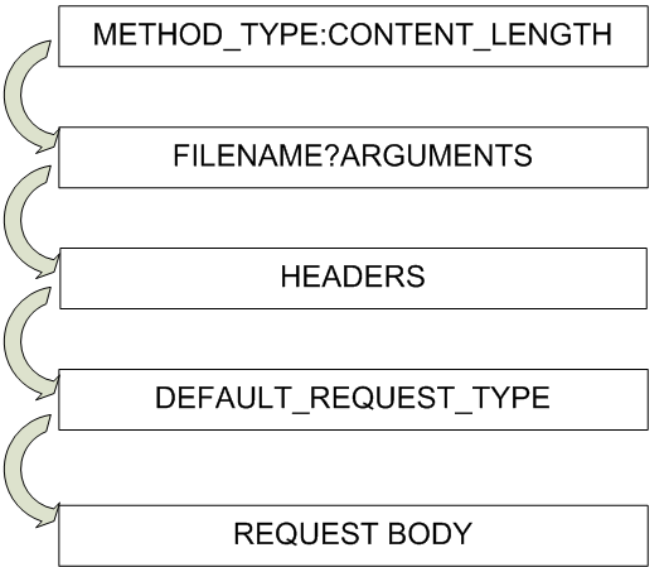

Figure 4.3: The request information is sent as four separate lines of text. The request body if present is sent last.

request body data using the same established socket if and when there is any request body data to read (See Figure 4.3). When receiving the reply from X-Switch, Mod_x first reads in the headers and checks for an empty line in the response. After reading the empty line, Mod_x then reads the rest of the data as the response body. Mod_x then waits for X-Switch to close the socket connection to signal the end of the response. For example consider a simple GET request for a URL 'http://localost:8080/∼mayumbo/perl/t3.pl?name=john' and an Apache Web server configured to call the Mod_x handler with 'EngineType' Perl, for directory patterns that correspond to '/home/mayumbo/public_html/perl'. Mod_x would then send the following lines of text to X-Switch:

1. GET:0
2. /home/mayumbo/public_html/perl/t3.pl?name=john
3. User-Agent: Mozilla/4.7 &x Accept: image/gif, image/jpeg, image/png
4. perl

X-Switch would then pass the relevant request processing information to the processing engine and relay the following response from the processing engine to Mod_x:

Last-Modified: Sun, 26 Dec 2007 12:43:24 GMT

Content-type: text/html;

<HTML>

<HEAD>

<TITLE>Hello World</TITLE>

</HEAD>

<BODY>Hello World</BODY>

</HTML>

Mod_x first reads the headers and then after detecting the empty line it parses the headers and reads the rest of the response as the response body.

4.3.2 X-Switch/processing engine communication protocol

The X-Switch system is designed to interface with multiple engines implemented using different programming languages. After identifying the processing engine that will process a particular request, X-Switch establishes communication with a processing engine using two socket pairs. One is for data while the other is for controlling and monitoring the request processing phase. X-Switch sends the information required to process the request to the processing engine as two lines of text. The first line is the interpreted request filename and any arguments that came with the request while the second line is the set of headers that are required to process the request. This information is sent to the processing engine using the control socket. Only after reading in this information can the processing engine read in the request body if there is one. The request body is sent using X-Switch's data input-output socket while the engine reads it in using its '*STDIN*' input stream. After request processing has started, X-Switch reads in the response from its data input-output socket while the script being run by the processing engine writes its output to its '*STDOUT*' output stream. X-Switch then relays this response to Mod_x using the request's connection socket with Mod_x. The processing engine then signals the end of the response by sending a response end control character via the control socket to X-Switch as illustrated

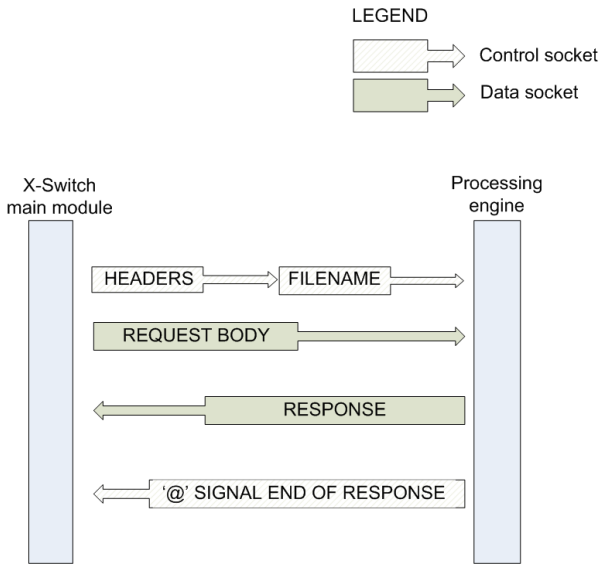

Figure 4.4: X-Switch communicates with the processing engine via two socket pairs. One is for data sending and the other for controlling the request processing

in Figure 4.4 .

The universal Web application server achieves its request and response routing through the various layers of processing as well as communication between these layers using the protocols just discussed. The following sections now present the detail of the design and implementation of the various layers of processing of the universal Web application server.

4.4 Mod_x

The modular design of the universal Web application server makes it possible to use different Web servers to develop the Web server module that interfaces with the main X-Switch module. Such a server would however need to have an API that can be used to access the Web server's core functionality. According to a survey carried out by Netcraft [34], the Apache Web server hosts more than fifty percent of the sites that responded to the survey. The Apache Web server also has contributed a lot to the growth of the World Wide Web and has a well documented API. For this reason the Web server module for the X-Switch

system, Mod_x, was implemented using the Apache Web server.

4.4.1 Design considerations

An Apache Web server module is a piece of compiled code that is either built into the Web server executable at the time the Web server is compiled or as an external piece of compiled code that is incorporated into the Web server at runtime. The external code is called a Dynamic Shared Object (DSO) on UNIX-based platforms or Dynamic Link Library (DLL) on the Microsoft Windows platform. Mod_x is built as a DSO so that installation of the universal Web application server does not require recompilation of the Apache Web server. It is involved in both the configuration and request processing operations of the Apache Web server. In the design of X-Switch-1.0, Mod_x was designed to have a handler for each kind of Web application implementation technology that the X-Switch system supported. The flaw with this design is that introducing a new Web application technology into the X-Switch system requires hard-coding its handler routine into Mod_x. Therefore Mod_x was redesigned to have a single handler that handles all of the requests for the different technologies that the X-Switch system supports. Achieving this required the use of the Apache Web server's command record and per-directory configuration structures.

The Apache Web server API defines a command record structure that can be used to define module specific directives. The module specific directives can then be used in the Apache Web server configuration file. Thus a module can define its command record table and define functions that are used to pass and interpret its directives. The Apache Web server allows the administrator of the Web server to define directives which apply only to specific directory patterns. Mod_x uses a combination of the per-directory configuration directives and the command records to control how the requests are processed. Consider the following configuration directives in Figure 4.5. 'SetHandler' is an Apache defined directive that tells Apache to call the Mod_x handler, x-handler, when ever Apache receives a request with a URI that matches the '/home/*/public_html/xswitch' directory pattern. The scope of the rules is determined by the 'Directory' directive. The 'Directory' directive also determines the URI pattern that the handler is responsible for. 'EngineType' is a Mod_x defined directive that sets the default processing engine type for the URI pattern.

```
<Directory ``/home/*/public_html/xswitch">
    Options ExecCGI
    SetHandler x-handler
    EngineType php
</Directory>
```

Figure 4.5: Apache directory configuration for a X-Switch system processing engine

These configuration directives therefore determine the scope of the directives, URI pattern to match, the request handler and default engine type.

4.4.2 Mod_x request processing

Mod_x performs the first and last tasks of the request processing phases of the universal Web application server. It participates in the content handling phase of the Apache Web server's request processing phases. It is called when Apache encounters a per-directory configuration directive that the Mod_x handler has registered a configuration directive hook for. Mod_x defines an *EngineType* configuration directive that determines the default engine type for that particular directory. Using the *EngineType* configuration directive makes it possible to register just one handler to handle all requests for all engine types for Mod_x. This is because the X-Switch processing engine that is used to handle scripts in that directory is determined using the defined *EngineType* configuration directive for that directory. The Web server administrator can define the directory patterns for which the Mod_x handler is invoked using Apache's 'Directory' configuration directive. The first pattern is for directories which can contain only non-bundled Web applications. This pattern is used to define directories under which simple scripts can be deployed. The default X-Switch system processing engine for scripts in this directory is defined using the *EngineType* configuration directive. The second kind of a directory pattern needs the string '/xswitch/' in the pattern. Directories with this pattern are used to deploy bundled Web applications. This directory pattern has a default engine type which can be overridden in the deployment descriptor of the bundled Web application. Thus it can be used to deploy Web applications of various implementation languages.

After being invoked, the Mod_x handler routine first establishes a TCP socket connection

with X-Switch. If successful Mod_x then proceeds to process the request else it sends a service temporary non-available response to the client that requested the resource. After establishing the connection Mod_x then determines the request method type. Mod_x currently only implements the HTTP POST and GET methods. After computing the method type Mod_x then reads in the rest of the headers that came with the request and sends the request information to X-Switch using the communication protocol presented in section 4.3.1. The request sending phases of Mod_x are illustrated in Figure 4.6.
After sending the request information, Mod_x then starts reading the response from the X-Switch main processing module. When reading the response Mod_x strictly checks for an empty line to signify the end of the header section of the response. Only then does it set the headers in the request's record structure. If the response does not return a header section Mod_x interprets this as a server internal error. After reading the headers, Mod_x then checks if the response is an internal redirect. If it is an internal redirect Mod_x first clears the POST method request data, if the main request was a POST request, and then sets the method type to the default GET method and redirects the request. If not an internal redirect Mod_x proceeds to read in the rest of the response. After finishing reading the response Mod_x then closes the connection with X-Switch and hands over control of the rest of the request processing phases to the Apache Web server. The response processing phases of Mod_x are illustrated in Figure 4.7.

4.5 X-Switch main module

The core of the X-Switch system is X-Switch. Most of the functionality of the universal Web application server is implemented in this module. It acts as the bridge between Mod_x and the processing engines, abstracting the functionality of the Web server from the processing engines and vice versa. It is implemented using the C programming language and is adapted from X-Switch-1.0. It however uses a single thread concurrency architecture as opposed to X-Switch-1.0, which used a multi-threaded architecture. Considering the fact that the actual processing and file input-output blocking operations occur in the processing engines, using a single processing thread helps improve the performance of the X-Switch

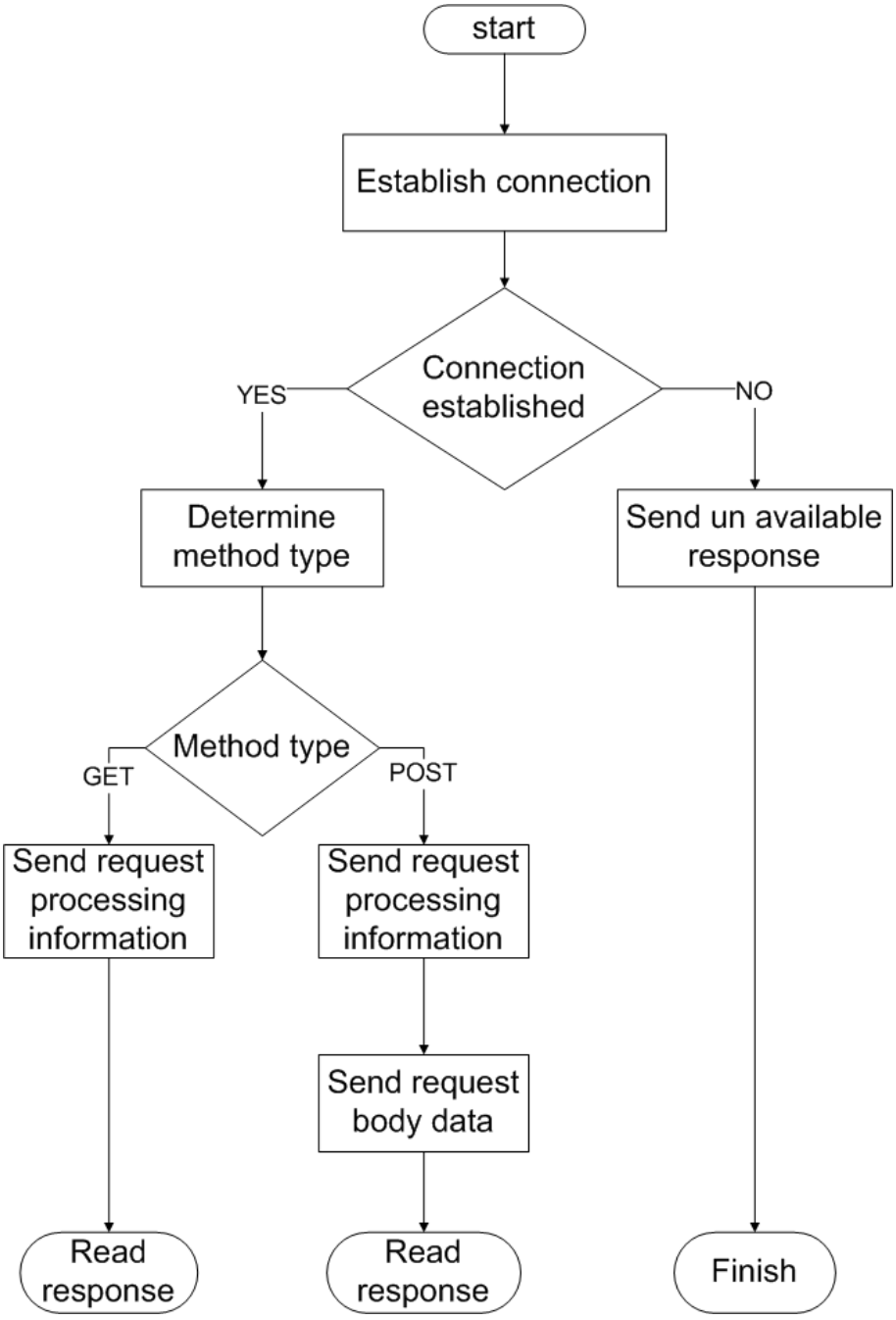

Figure 4.6: Flow chart for Mod_x's request processing phase

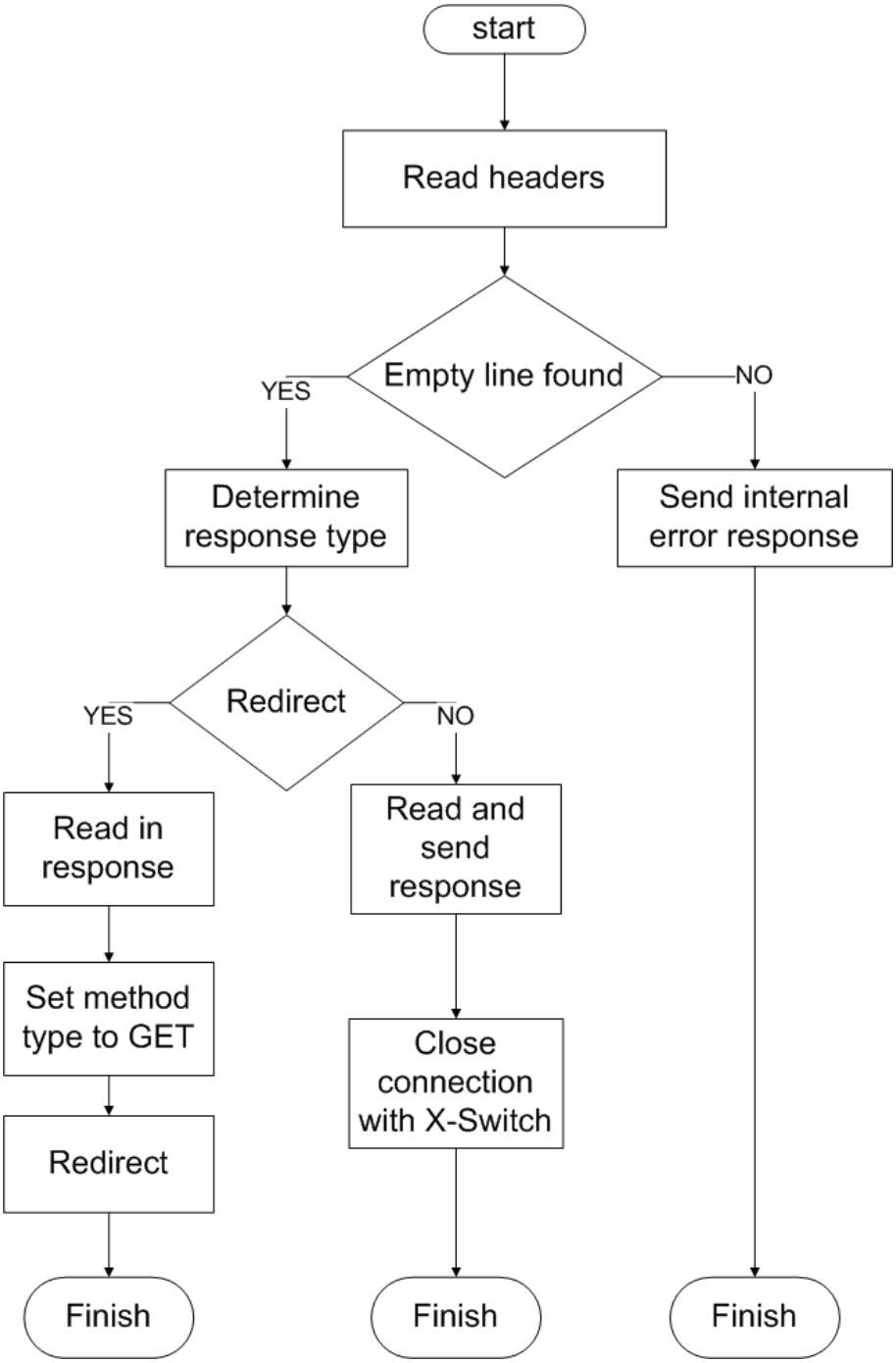

Figure 4.7: Flow chart for Mod_x's response processing phase

system. In addition, studies by Pariag et al [39] reveal that in their experiments the event driven server and hybrid based server achieved up to 18% higher throughput than the best implementation of the thread based server. Therefore X-Switch-1.1 uses a single thread to listen to and poll all its connections.

After accepting a connection, X-Switch reads in the request processing information and then checks if a Web application or script that is registered to handle the URI pattern exists. X-Switch then computes the owner of the Web application or script required to process the request and then looks up or starts a processing engine to handle the request. If the load is not high enough to require the startup of another processing engine the request is queued for processing. X-Switch then creates a timer for the request which is reset every time there is activity on the connection. Connections time out if no processing engine is made available after a specified period of time. If or after a processing engine is made available, X-Switch assigns the processing engine to the connection and then sends the request processing information to the processing engine and starts listening for a response on the connection. X-Switch then relays the response to Mod_x and closes the connection after reading the rest of the response. X-Switch request processing stage is illustrated in Figure 4.8.

4.5.1 User information management

The user in this context refers to the owner of a script or bundled application that is deployed in the universal Web application server. The fundamental unit of information management in X-Switch is the 'User Information' unit. A typical user information unit has a queue for requests and a set of processing engines and is uniquely identified by the owner of the request and the type of requests that it is responsible for. After reading in the request processing information, the first step in request processing is determining who owns the script or bundled application and the second step is determining what type of application it is. For example, a user who owns Perl and Java Web applications will have two corresponding 'user information' units, one for the Perl Web applications and the other for the Java Web applications. The user information for a particular user and Web application type is created once when the server receives the first request for that type of

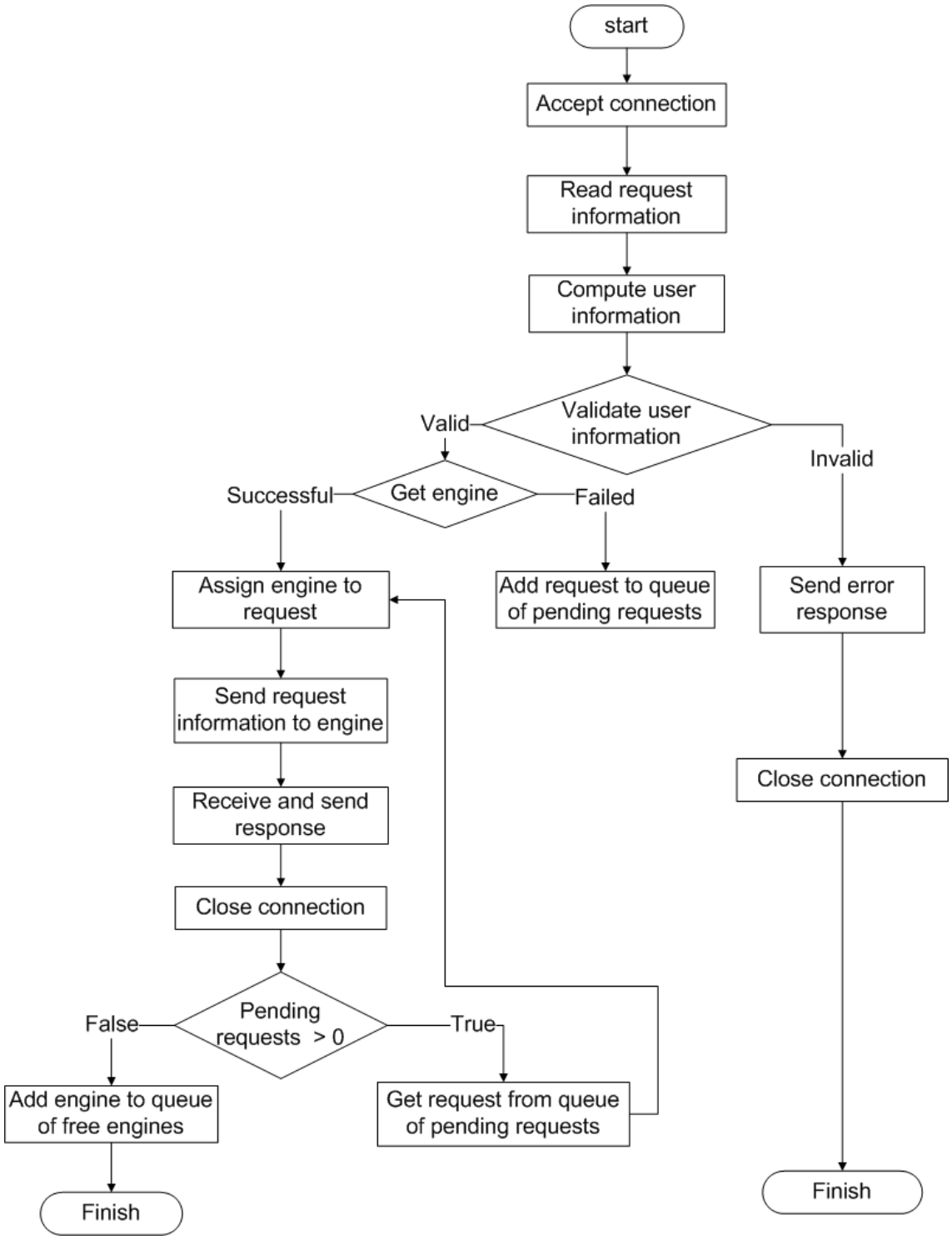

Figure 4.8: Flow chart for X-Switch request processing stage

Web application owned by that user. After the creation of the user information a processing engine is created and assigned to service the request. Upon serving a request the engine is added to a queue of free engines that is owned by the user information unit. If there are pending requests the engine is assigned to the first request in the queue of pending requests. The queue of pending requests is the queue to which requests are added if they arrive when there is no free processing engine and load levels do not justify engine creation. Requests are assigned according to user information units. If there are no pending requests when a processing engine finishes servicing its requests, the processing engine is added to the queue of free processing engines. The queue of free processing engines is owned by the user information unit. When subsequent requests arrive one of the free processing engines is assigned to service the request.

4.5.2 Processing engine management

Processing engines are the back-end of the X-Switch system. They are responsible for the actual execution of the Web application scripts. A processing engine can be implemented using any programming language in order to service requests of that language. The processing engine should however adhere to the protocol for communication presented in section 4.3.2 in order to communicate with X-Switch successfully. X-Switch-1.1 also attempts to advance processing engine management in the universal Web application server to make it more robust. X-Switch-1.0 had hard-coded processing engine information in X-Switch. In the current implementation, X-Switch uses an XML configuration file to read in information required to run the processing engines efficiently. This change makes it possible to add and remove processing engines without having to recode or recompile X-Switch. X-Switch reads the XML configuration file on startup. The configuration of an engine contains the path for execution of the engine, the type of technology that the engine supports and the default welcome page for that type of technology. Figure 4.9 is a sample x_conf.xml file.

```
<?xml version="1.0"?>
<main>
    <engine>
        <type>perl</type>
        <path>perl ../con_perl/perlmod.pl</path>
        <welcome>index.pl</welcome>
    </engine>
    <engine>
        <type>php</type>
        <path>php ../con_php/phpmod.php</path>
        <welcome>index.php</welcome></engine>
    <engine>
</main>
```

Figure 4.9: A sample X-Switch XML configuration file

Processing engine life cycle

In its life cycle a processing engine will serve a specified number of requests after which it is destroyed. Killing the engines after they serve a specified number of requests helps clean up memory leaks which can potentially be in either the processing engine itself or the scripts that it runs. The life cycle of the processing engine can further be cut short if it remains idle for a specified amount of time. A timer is created whenever a processing engine is added to a queue of free processing engines. When this timer expires the processing engine is destroyed. The timer is only cancelled when the processing engine is assigned a request to service. Similarly, a processing engine assigned a request to service can be destroyed if there is no activity recorded on the processing of the request after a specified amount of time. Figure 4.10 illustrates the life cycle of a processing engine.

Load control

Processing engines are created only when the load cannot be sustained and before the user reaches the maximum number of processing engines that they can own. Each user can only have a specified number of each type of processing engine. User loads for different units of user information are determined independently of each other. Load checks are triggered by two conditions collectively. The first condition is that a request arrives when there are no

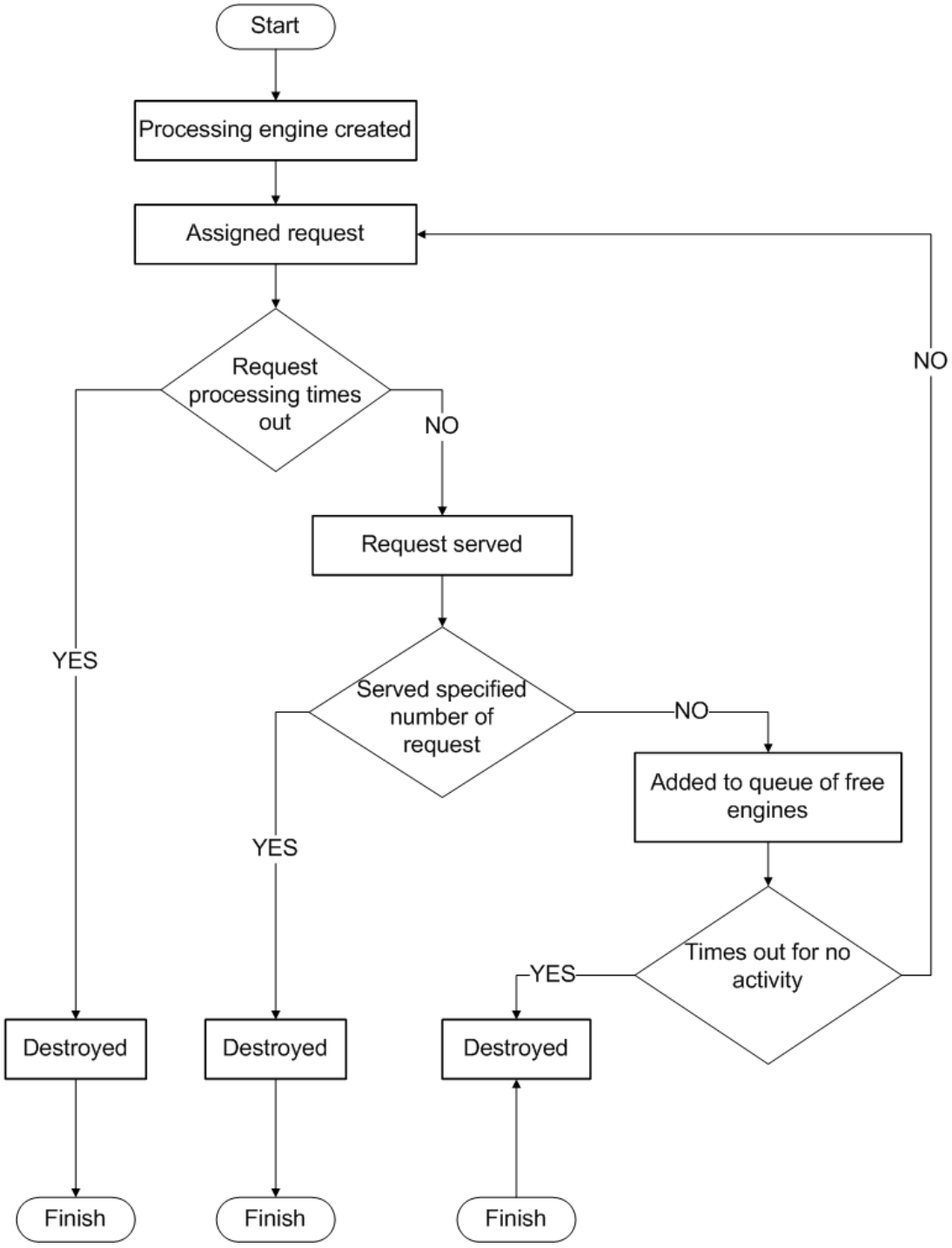

Figure 4.10: The life cycle of a processing engine

Table 4.1: A summary of the conditions that trigger the creation of a new processing engine

Condition 1	Condition 2	Condition 3
No engine	Num of requests > critical value	Request arrival > departure
Engine destroyed	Num of requests > critical value	Request arrival > departure

processing engines available. The second condition is that the queue of pending requests is more than the critical value. The fulfillment of these two conditions collectively results in a load check. The load is determined by comparing the request arrival rate to the request departure rate. If the arrival rate is greater than the departure rate then a new engine is created. The arrival and departure rate are determined using a sliding window of the five previous values recorded. A load check also can be triggered when a processing engine is destroyed. In such a case the length of the queue of pending requests is checked and if it is higher than the critical value, a load check is enforced. This helps prevent overloading the system as a result of killing faulty engines when the load is still high. Table 4.1 summarizes the conditions that lead to the creation of a new engine.

4.5.3 Timers

X-Switch uses timers to keep track of dormant processes or resources. Timers help terminate non-responsive connections as well as engines that are not in use. X-Switch keeps track of two kinds of dormant resources; processing engines and connections.

Dormant connections

X-Switch creates and assigns a timer to a connection the moment that the connection is established and assigned resources. The connection timers are reset every time there is activity on the connection such as a socket read or write event. These timers are checked periodically and connections which do not have activity are closed and the resources they were using released. In addition X-Switch also checks the connection timers every time that it receives requests after reaching the critical number of connections that it can handle. This

number is 100 connections by default. This helps release and make available connection resources when demand is high.

Dormant processing engines

The X-Switch system is a multi-user environment that implements multiple Web application technologies. This requires an efficient usage of resources and as a result dormant processing engines have to be destroyed. When a processing engine finishes processing a request X-Switch checks to see if there are any pending requests in the queue of pending requests for the user who owns the engine. If there are no pending requests the processing engine is added to the queue of free processing engines and a timer is created that is only cancelled when the processing engine is assigned another request to service. Processing engines that are dormant for more than 10 seconds are destroyed.

4.5.4 The X-Switch wrapper script

X-Switch uses a wrapper script, suexecme, to execute its processing engines in the context of the owner of the scripts that the processing engine will run. It runs the processing engine with the groupId and userId of the owner of the scripts that it will be servicing. It considers attempts to run processing engines with root privileges as an attack on the system security. suexecme does not implement many security checks. However cgiwrap [2], suPHP [28] and other wrapper scripts demonstrate that such security checks can be enforced with a wrapper script.

4.5.5 Web application deployment

The universal Web application server's modular design uses multiple implementations of Web application technologies. Each of these technologies use different approaches for deploying Web applications. However, in its generalization, the universal Web application server places most of the core functionality in X-Switch and not in the various back-ends. Achieving this requires Web application deployment to be handled in X-Switch. In addition, the X-Switch module runs the processing engines in the context of the person who

owns the Web application. This implies that the process of determining the owner of the Web application or script should be done in X-Switch. The universal Web application server therefore implements a Web application framework for easy Web application deployment that is general and can be extended to the various back-end technologies. The Web application framework for easy deployment puts the management of deployment information in X-Switch thus making it possible to do the path translation at X-Switch level.

Bundled Web applications

Web applications are deployed in the universal Web application server as either simple scripts or as a bundled Web application. Bundled Web applications are archived and deployed as a '*.xar*' zip archive. A *xar* archive has a simple directory structure with an XML deployment descriptor (*weblet.xml*) at root level and *weblet* directory that contains all the scripts that implement the functionality of the Web application. Other resources such as images should be placed at the root level of the directory whereas resources like databases can be placed anywhere within the archive in directories defined by the author of the Web application. The Web application can be bundled using any zip archiving tool. The archives are exploded the first time that the first request for the Web application is handled. X-Switch uses the 'unzip' application to explode the archive. The unzip application is executed using a wrapper script in order to explode the application with the same privileges as the owner of the archive. Archives also are used as a reference to the most current version of the Web application. The time stamp of the exploded Web application is not supposed to be older than that of the archived Web application. Therefore, while searching for the deployment descriptor for the Web application, X-Switch checks whether or not the archive is more recent than the exploded application. If the archive is more recent then it is exploded to get the most recent version of the Web application. The major difference between bundled applications and scripts is that URIs for scripts are received as actual paths to the file while URIs for bundled applications have to be translated into actual paths to the file. The bundled applications are grouped into a directory structure and have a deployment descriptor which has the configuration of the application. The processing engine type for bundled applications is configured in the deployment descriptor whereas

the processing engine type for a script is determined by the 'EngineType' directive in the Apache configuration file. Therefore scripts are grouped into directories according to type whereas bundled applications can be grouped anyhow within a 'xswitch' directory. The number of 'xswitch' directories is not limited but a 'xswitch' directory cannot be located with the path of another 'xswitch' directory.

Path translation

X-Switch receives two kinds of URIs from Mod_x. The one is a path to the script while the other is a URI for a resource that is part of a bundled Web application. After reading in the request processing information, X-Switch first checks if the request is associated with a bundled application or if it is a simple script. If the request is associated with a bundled Web application, X-Switch then translates the path. On the other hand, a path that cannot be associated with a bundled Web application is interpreted as a path to the actual script. X-Switch then proceeds to check if the path is valid by verifying the file's existence. To determine whether or not a path is associated with a bundled Web application, X-Switch first checks for the existence of the '/xswitch/' string within the URI. If it exists, X-Switch then recursively checks for the first occurrence of a deployment descriptor in directories that fall under the 'xswitch' directory. If a deployment descriptor is found X-Switch checks if the configuration of the Web application is already cached, if not, X-Switch then reads and parses the deployment descriptor. Using this deployment descriptor, X-Switch then does the URI pattern and path mapping and caches the result. X-Switch then checks to see if there is a URI pattern that corresponds to the path read in from Mod_x to get the translated path. If the URI pattern is not found, the path is considered to be a request for a resource that is not available. Figure 4.11 illustrates part of the path translation process. For example, consider the deployment descriptor in Figure 4.12 and the URI '/home/mayumbo/public_html/xswitch/sample/pi_finder'. X-Switch receives the URI from Mod_x. The first step in path translation is locating the '/xswitch/' string pattern in the URI. In this case it is there, so X-Switch proceeds to locate the deployment descriptor. It first checks in the '/home/mayumbo/public_html/xswitch/sample' directory. If the deployment descriptor is not located in this directory, X-Switch checks the lower directory

which in this case would be the '/home/mayumbo/public_html/xswitch/sample/pi_finder' directory. X-Switch keeps checking the lower directories until it locates the first occurrence of the file descriptor. In our example the deployment descriptor is located in the '/home/mayumbo/public_html/xswitch/sample' directory. X-Switch reads the deployment descriptor and then maps URI patterns to the actual paths to the scripts. Thus the pi_finder URI pattern will be mapped onto the 'weblets/example.php' script path. Therefore the URI '/home/mayumbo/public_html/xswitch/sample/pi_finder' is translated to '/home/mayumbo/public_html/xswitch/sample/weblets/example.php'.

Web application deployment descriptors

Web application deployment descriptors can currently be implemented as a simple XML file. They are used to store the Web application's configuration information. They contain information about the processing engine that is required to run the Web application's scripts. Apart from the Web application type information, the deployment descriptor also has information which maps URI patterns to actual paths for script execution. The processing engine type from the deployment descriptor overrides the processing engine type that comes from the request information from Mod_x. Using the deployment descriptor removes the restriction of grouping Web applications according to the Web application technology that they implement. Thus you can put Java, PHP, Perl and any other Web application type in the same folder. This makes Web server administration and Web application deployment easier as it reduces the number of directories to configure and manage in the Web server. The design of the deployment descriptors is similar to that of Java servlet descriptors except for aspects that are specific to Java servlets like initialisation parameters. Figure 4.12 is a sample deployment descriptor that was used to deploy a simple 'Hello World' application.

4.5.6 X-Switch main module summary

In a typical request processing scenario, X-Switch accepts a connection from Mod_x and reads in the request processing information which includes the path for the requested resource. The path is first translated and the translated path is then used to compute the

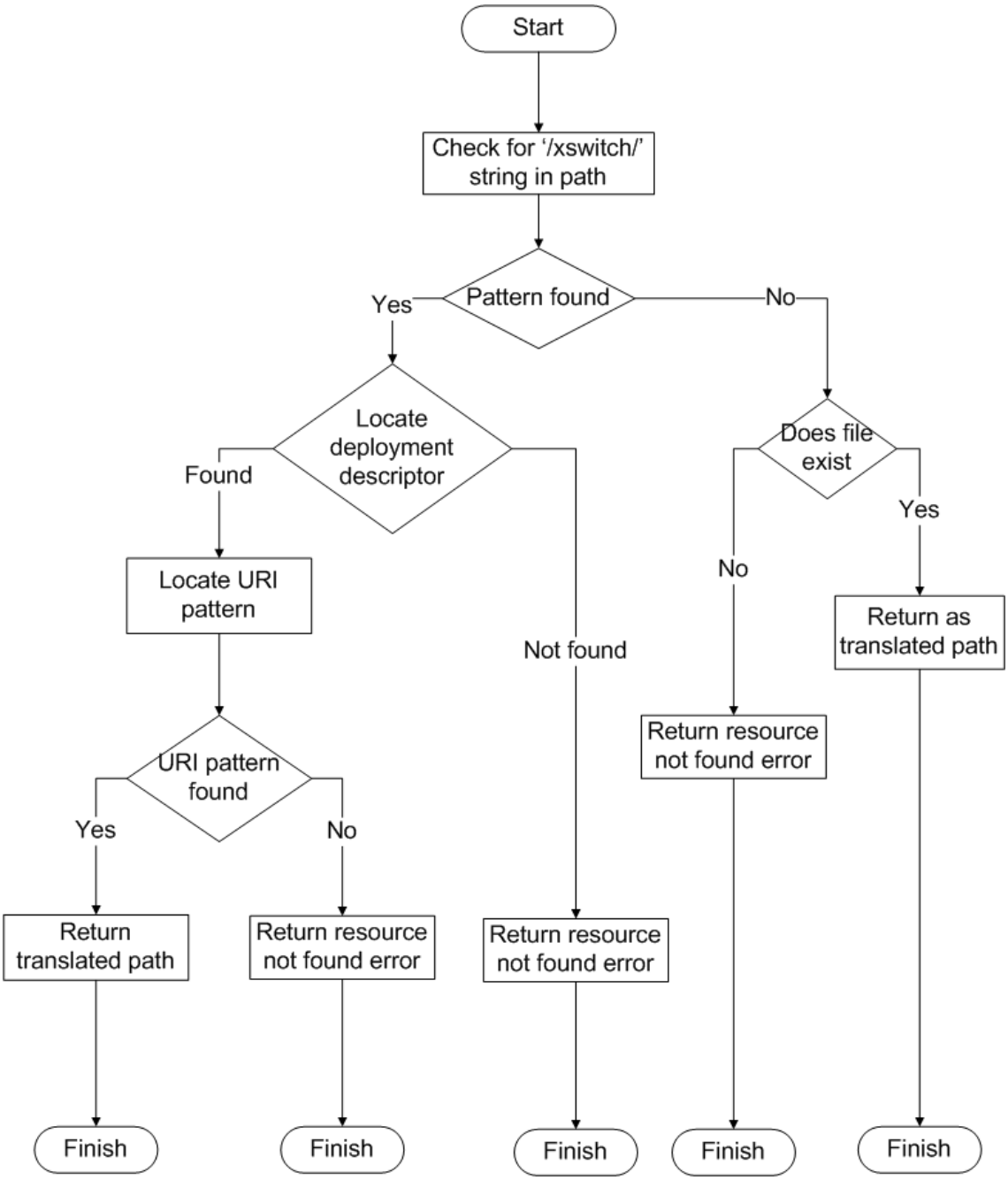

Figure 4.11: X-Switch request path translation flow chart

```
<web-app>
    <type>php</type>
    <weblet>
        <name>sample</name>
        <path>weblets/example.php</path>
    </weblet>
    <weblet-mapping>
        <name>sample</name>
        <url-pattern>/pi_finder</url-pattern>
    </weblet-mapping>
</web-app>
```

Figure 4.12: A sample deployment descriptor for a simple 'Hello World' application

owner of the script. The type of processing engine can either be read from the deployment descriptor of the Web application or comes with the request processing information in the case of simple scripts. The owner and request type are then used to manage the user information that is used to control the user's connections as well as the processing engines. The Web applications are deployed either as simple scripts or as bundled Web applications. X-Switch then sends the request processing information to the processing engine and relays the response from the processing engine to Mod_x.

4.6 Processing engines

Processing engines implement the back-end functionality of the universal Web application server. Any programming language that can write to the standard output stream and read from the standard input stream can be used to implement a processing engine. The processing engines are run via suexecme, the X-Switch wrapper script. They are executed with an argument which has the file descriptor for the socket that is used for request processing control. Upon execution the processing engine opens the socket using the file descriptor in order to receive request processing information. The processing engines are run persistently and are only destroyed after they serve a specified number of requests or if they remain dormant for more than 10 seconds. The current X-Switch system implements

four types of processing engines: Perl, PHP, Java and Python processing engines.

4.6.1 Java servlet engine

Java Web applications are processed using the X-Switch servlet engine in the X-Switch system. X-Switch-1.0 implemented a multi-threaded servlet processing engine. Using threads in the servlet engine shifted the responsibility of load control from X-Switch to the servlet processing engine. For this reason, X-Switch-1.1 uses a single threaded processing engine and all the load control and monitoring is done in X-Switch. X-Switch-1.1 therefore uses multiple processing engines when load is high as opposed to X-Switch-1.0 which would just spawn more threads in the single processing engine. The X-Switch Java servlet engine is implemented as a lightweight processing engine and has a partial implementation of the servlet API (See Appendix A.1 for the implemented methods). The Java programming language does not by default allow a programmer to access file descriptors and it is recommended not to try to override this encapsulation. However, the communication protocol between X-Switch and the processing engines requires two sockets, the control socket and the data socket. The data socket uses the standard input and output whereas the control socket requires opening the stream using a file descriptor which was passed to the processing engine as a command-line argument. To achieve this the Java servlet engine has a class that implements a native method that is used to open input/output streams of file descriptors inherited from the process that spawned the processing engine. The class first verifies that the socket exists and is open for reading and writing before creating the file descriptor. The file descriptor is then used to open the stream for the control socket. This class is loaded into the Java virtual machine as the servlet engine is loaded. After being loaded, the servlet engine then establishes communication with X-Switch on the control socket. The servlet engine then blocks on the control socket, waiting for request processing information. Upon receiving the request processing information, the servlet engine then loads the servlet class and future requests for the servlet are served using instances of the already loaded class. The time stamp for the loaded servlet is always supposed to be more recent than that of the servlet class when processing a request. If not, the servlet is reloaded. The execution environment is then prepared by creating instances of the request and response objects using

the request processing information received. The servlet instance is then serviced using the instances of the request and response objects. After servicing the servlet, the servlet engine then signals the X-Switch module for the end of the response. It then blocks on the control socket waiting for another request. The negative aspect of not using threads within a single process is the difficulty that comes with implementing Http sessions. Requests use sessions to store session variables. This is easier to implement and manage when all requests are served from the same engine that uses threads to serve different requests. Threads share memory and this makes it easier and possible to share session variables. The X-Switch design requires spawning more processes under high load which do not share memory and thus making the sharing of session variables difficult.

4.6.2 PHP processing engine

The PHP processing engine is used to serve PHP Web applications. It uses an interpreter that is executed via the PHP commandline SAPI. It has a partial implementation of the PHP global variables required for processing requests using PHP Web applications. For the implemented PHP GLOBALS see Appendix A.2. Like the Java programming language, PHP also does not allow the programmer to access the file descriptors inherited from a process that was implemented using another programming language. To solve this problem the PHP processing engine uses the 'xreadsock' PHP module for reading and writing to a socket using an inherited file descriptor. xreadsock is a X-Switch PHP processing engine module. The PHP processing engine is also implemented as a persistent lightweight processing engine. The xreadsock module can be compiled into the PHP interpreter or loaded at runtime using PHP's *dl()* function for loading modules at runtime. The PHP processing engine loads the xreadsock module before it goes on to establish the communication channel on the control socket with X-Switch. It then blocks on the control socket, waiting for request processing information. After receiving the request processing information the PHP processing engine then proceeds to set up the PHP GLOBALS which contain the information required for processing the request. It then proceeds to run the script. When the script completes running, the PHP processing engine then signals X-Switch for the end of the response. The signal is sent using the control socket. It then proceeds to clear the

PHP GLOBALS and blocks on the control socket waiting for another request.

4.6.3 Perl processing engine

The Perl processing engine is a lightweight engine that is used to run Web application scripts implemented using the Perl programming language. There were less complications in implementing the communication protocol between the Perl processing engine and the X-Switch module as Perl applications easily interface with applications programmed using C. The Perl programming language has a library that can be used to open socket streams using file descriptors inherited from a parent process. Perl Web applications use the CGI environment variables to access information that came with the request. The X-Switch Perl processing engine implements most but not all these environment variables. For the implemented variables see Appendix A.3. On startup, the Perl processing engine first opens the communication channel with X-Switch and then blocks, listening for request processing information on this control socket. After reading the request processing information, it proceeds to set up the CGI environment variables and then runs the script. When the script completes running, the Perl processing engine sends a signal to X-Switch, on the control socket, indicating the end of the response. It then clears the CGI environment variables and blocks on the control socket waiting for the next request.

4.6.4 Python processing engine

The Python processing engine is a simple lightweight processing engine that is used to run Python Web applications. In its implementation it does not parse the headers for the Web application. It however does support and implement HTTP GET and POST methods. Python ,like Perl, also supports using file descriptors inherited from the parent process to read and write on sockets. The Python processing engine begins by first establishing a control socket communication channel with X-Switch. It then blocks on the control socket waiting for request processing information. After reading the request processing information it then runs the Web application script. When the script finishes running, the Python processing engine then signals X-Switch that the response has ended. The signal

is sent on the control socket. It then blocks on the control socket and waits for the next request.

4.7 Summary

X-Switch-1.1 is a single threaded Web application server that uses a single thread to poll and listen on all its sockets. It receives request processing information from Mod_x an Apache Web server module. The actual execution of the Web application scripts is done in the processing engines. X-Switch-1.1 currently implements Perl, Python, PHP and Java processing engines. Requests are received by Mod_x and routed to X-Switch which then computes which processing engine to assign the request to. The processing engine then runs the script which writes its response to the standard output. X-Switch then reads the response from the engine's standard output and relays the response to Mod_x.

Chapter 5

Evaluation

5.1 Overview

The aim of building the universal Web application server is to support Web applications in a secure environment for multiple authors and without compromising the performance of such a server. This chapter discusses how the universal Web application server was tested and evaluated. The evaluation includes performance tests and analysis, user testing and a case study which involved the installation of phpBB2 in the universal Web application server. The following sections of this chapter each outline how the Web application server was tested and evaluated.

5.2 Performance analysis

The design and implementation of the universal Web application server takes a modular approach which introduces layers of processing that can potentially affect the request processing time. The performance testing and analysis compares the performance of the universal Web application server to known Web application servers. The performance evaluation also seeks to find the impact of the layers of processing on the total processing time. Two machines were used to create a simple network using a crossover cable. The client was run on a Pentium IV 3.2 Ghz desktop with 512 Mb RAM while the server was installed on a Pentium M 1.73 Ghz laptop with 512 Mb RAM. The software used to simulate user transactions and connections was Siege 2.65 [18] and Jakarta-jmeter-2.2 [6].
Jmeter was used in most of the experiments as it logs the results better than Siege. In addition JMeter has the capability of configuring each simulated connection with different properties. Thus with JMeter it is easy and possible to simulate different popularity for

applications. In Jmeter terminology, a ramp up defines the amount of time between thread start up. A constant throughput timer controls the amount of time between requests issued by the thread. JMeter also has a random Gaussian timer that issues requests randomly simulating typical user patterns. Siege on the other hand tries to start as many connections as possible per client until the server goes down. For this reason Siege was used in experiments that focussed on stressing the universal Web application server. The experiments focus on the performance of the universal Web application server under varying conditions.

5.2.1 Experiment 1

Aim

A desirable solution for a universal Web server needs to use resources efficiently. The number of concurrent users that a Web application server can handle also helps in determining the return on the investment in hardware. The higher the number of concurrent users, the higher the return on the hardware and also the more efficient the use of the hardware is. This test measured the number of concurrent clients that the Web application server can support.

Methodology

The experiment was carried out by conducting a series of runs and varying the number of concurrent clients with each subsequent run. Siege2.65 was used to manage the client connections. A simple Perl Web application was used in this experiment. The application produced a 43Kb response of randomly generated characters. The first run had 25 concurrent clients. The number of concurrent connections was increased by 25 with each subsequent run until Siege could not allocate memory to run the test. The maximum number of concurrent clients that was used was 375 which was the maximum that Siege could allocate memory for.

Metric 1: Average response time.

Metric 2: Throughput of the Web application server.

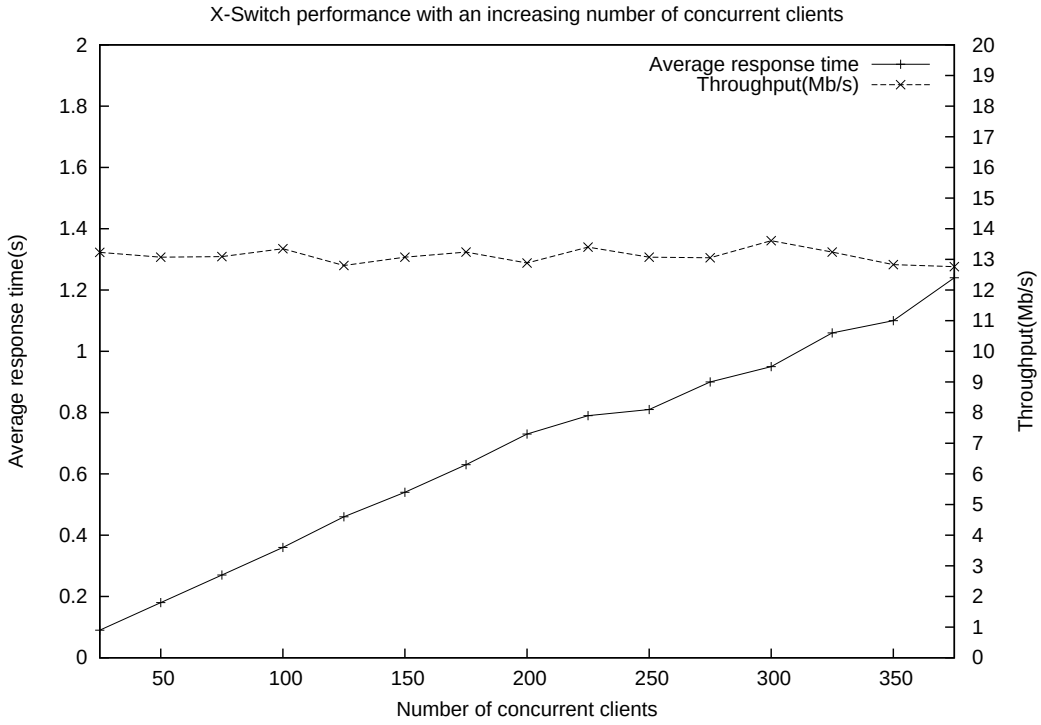

Figure 5.1: Performance of the universal Web application server with an increasing number of concurrent connections

Results

The results of this experiment are show in Figure 5.1

Discussion

The server throughput was more or less constant, which means that at least there was a more or less consistent network transfer with the increase in the number of concurrent clients. In addition, the increase in the response time as the number of concurrent clients was increased was not drastic. Moreover, a linear degradation in the response time was observed and is ideal. This shows that the universal Web application server has support

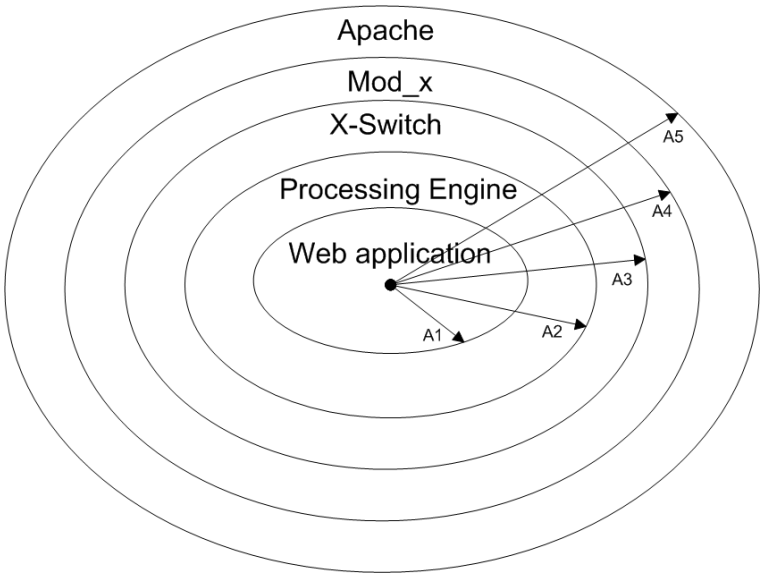

Figure 5.2: Request processing layers for the universal Web applications server

for concurrent connections and that the performance does not degrade drastically with an increase in the number of concurrent connections.

5.2.2 Experiment 2

Aim

The design of the universal Web application server introduces several layers of processing which can potentially degrade the performance of the Web application server. The purpose of this experiment was to measure the percentage that each of the universal Web application server processing layers contributes to the total response time. Web application servers are responsible for routing information to the Web application and also for setting up the environment for request processing. Therefore an efficient design of such a Web application server contributes a small fraction to the total processing time. Figure 5.2 illustrates the various layers of processing.

Methodology

This experiment was conducted by issuing 100,000 requests to each layer of processing and the time required to service the requests was measured and recorded. The script used in this experiment had a simple 52 byte response. The experiment was conducted in the following stages.

1. A simple application for issuing requests directly to the Web server was implemented in C. It issued 100,000 requests and recorded the total amount of time it took for the Web server to service the requests. In this run the requests were issued to the Web server front end which then passed the requests on to the lower processing layers. The time was recorded as A5 (time taken to service the request through all the layers of processing).

2. In the second run, Mod_x was replaced with a module that did not connect to the X-Switch system. The module did not do any processing apart from returning the Http status OK (200) response. Thus the time taken for the request to be processed is an approximation of the request processing time without the overhead of the X-Switch system. 100,000 requests where then issued directly to the Apache Web server. The total processing time was then recorded as A5 - A4 (time taken to process the request through the Apache Web server layer).

3. For the third run of the experiment, a simple script was written that interfaced with X-Switch and used the protocol that Mod_x and X-Switch use to communicate. The script was used to measure the amount of time it took to process 100,000 requests without the Apache Web server and Mod_x module processing layers. The time was recorded as A3 (time taken to process the request through X-Switch and the lower layers).

4. In the third run, a script that spawned an engine and issued 100,000 requests directly to the processing engine was written and used. The total time it took to process the request was recorded as A2 (time taken to process the request through the engine processing layer and its lower layers).

Table 5.1: The percentage of time that each processing layer contributes to the total processing time

Processing layer	Percentage of time (%)	Measured time (s)
Web application	24.52	14.957005
Apache	40.13	24.479877
Mod_x	14.40	8.781802
Processing engine	11.06	6.033242
X-Switch	9.89	6.748551

5. Lastly, the time taken to run the script was measured by running the script 100,000 times in a persistent interpreter and recording this time as A1.

The following formulae were used to calculate the percentage of time spent in each of the processing layers. From Figure 5.2, and taking total response time as A5,

$$T_x = ((A3 - A2)/A5) * 100 \ \textit{(X-Switch processing layer)}$$

$$T_p = ((A2 - A1)/A5) * 100 \ \textit{(processing engine layer)}$$

$$T_w = (A1/A5) * 100 \ \textit{(Web application processing layer)}$$

$$T_a = ((A5 - A4)/A5) * 100 \ \textit{(Apache Web server processing layer)}$$

$$T_m = ((A4 - A3)/A5) * 100 \ \textit{(Mod\_x processing layer)}$$

Metric: Percentage of processing time per layer

Results

The results of this experiment are tabulated in Table 5.1

Discussion

The results show that most of the processing time is spent in the Apache and Web application processing layers. Thus the modular design of X-Switch does not degrade the performance of the Web application server. Moreover, the X-Switch processing layer contributes the least percentage of time to the total processing time of the requests. The response time for the Web application used in this experiment was small cause a trivial response was used. Therefore as the response size increases the amount of time that the generation of the response takes would also increase and the Web application would contribute the bigger proportion of time.Therefore the request response time is mostly affected by the time it takes to run the Web application and not the routing of the request and the response.

5.2.3 Experiment 3

Aim

The aim of this experiment was to measure the response time of the universal Web application server using a synthetic work load and compare it to other Web application servers. The synthetic workload was used in order to allow the Web application server to perform at its best.

Methodology

In this experiment Jakarta-jmeter was used to issue requests with ten concurrent connections (threads). Each of the threads issued 1,000 requests. The threads each had a constant timer of 0 seconds and the ramp up period for the threads also was 0 seconds. Thus all the threads started issuing requests at the same time while each thread had a 0 lapse between consecutive requests. Simple 'Hello World" applications were used in this experiment. The setup was repeated with each of the four processing engines, that is, the Perl, PHP, Python and Java processing engines. The same setup also was repeated with Apache Tomcat, Modphp, Modpython, FastCGI and SpeedyCGI.
Metric: Average response time

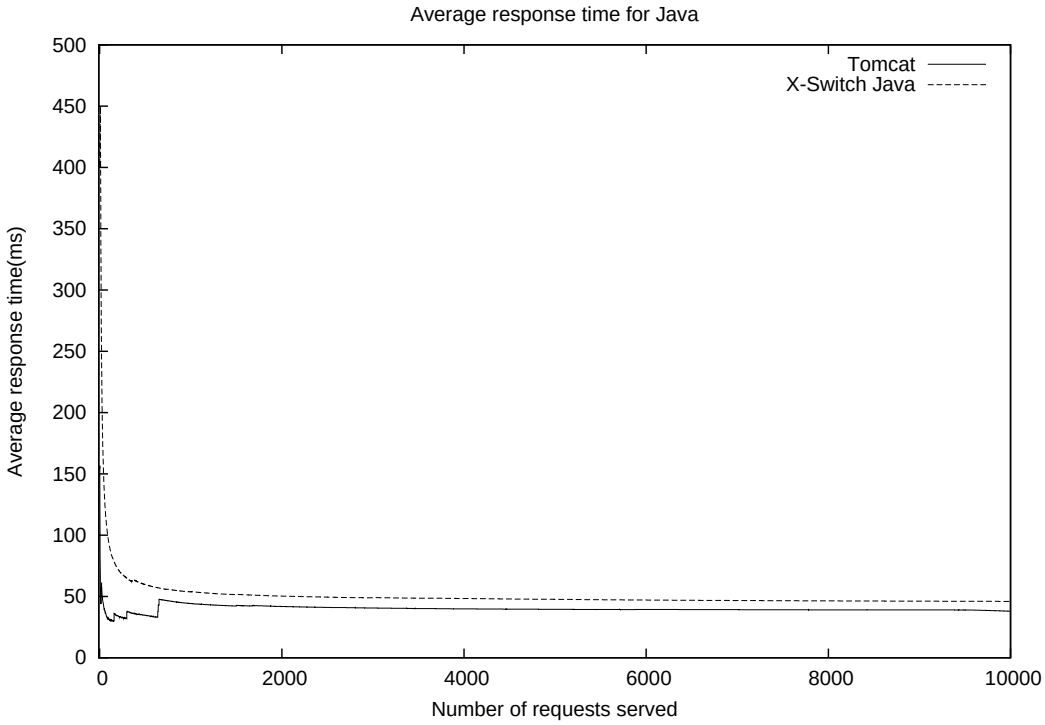

Figure 5.3: Average response time for Apache Tomcat and the Java processing engine

Results

The results are graphed in groups of programming languages. Figure 5.3 shows the results for Apache Tomcat and the Java processing engine for the universal Web application server. Figure 5.4 is graph of the results from Modphp and the universal Web application server's PHP processing engine. Figure 5.5 is a graph of the results for SpeedyCGI, FastCGI and the Perl processing engine for the universal Web application server. Figure 5.6 is a graph of the Python processing engine for the universal Web application server and Modpython.

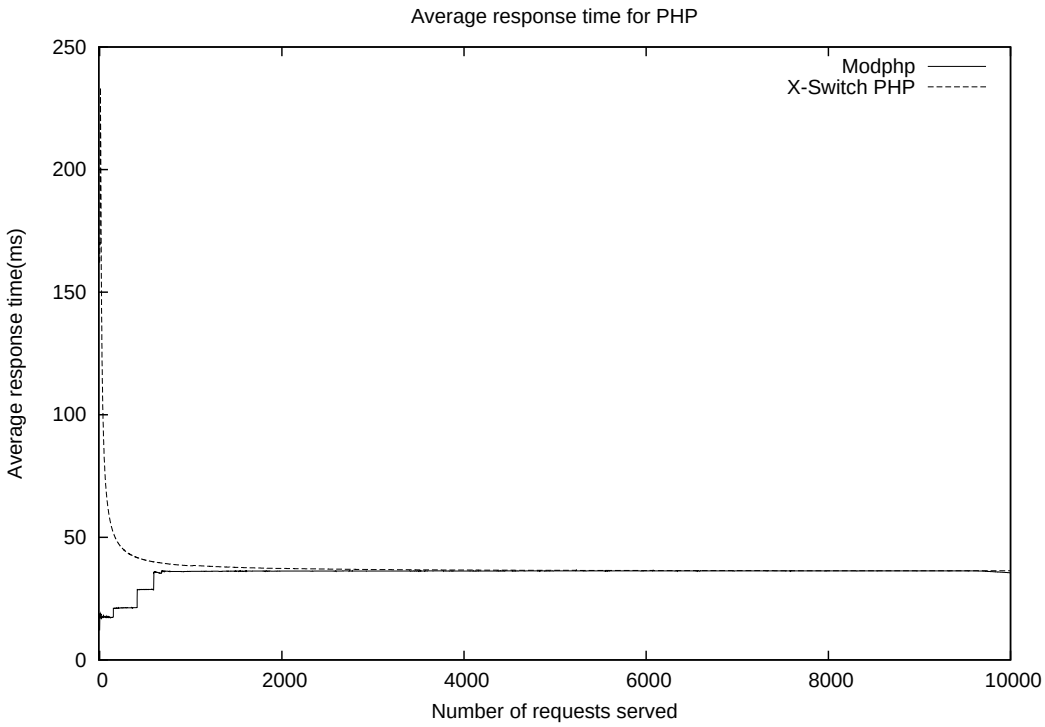

Figure 5.4: Average response time for Modphp and the PHP processing engine

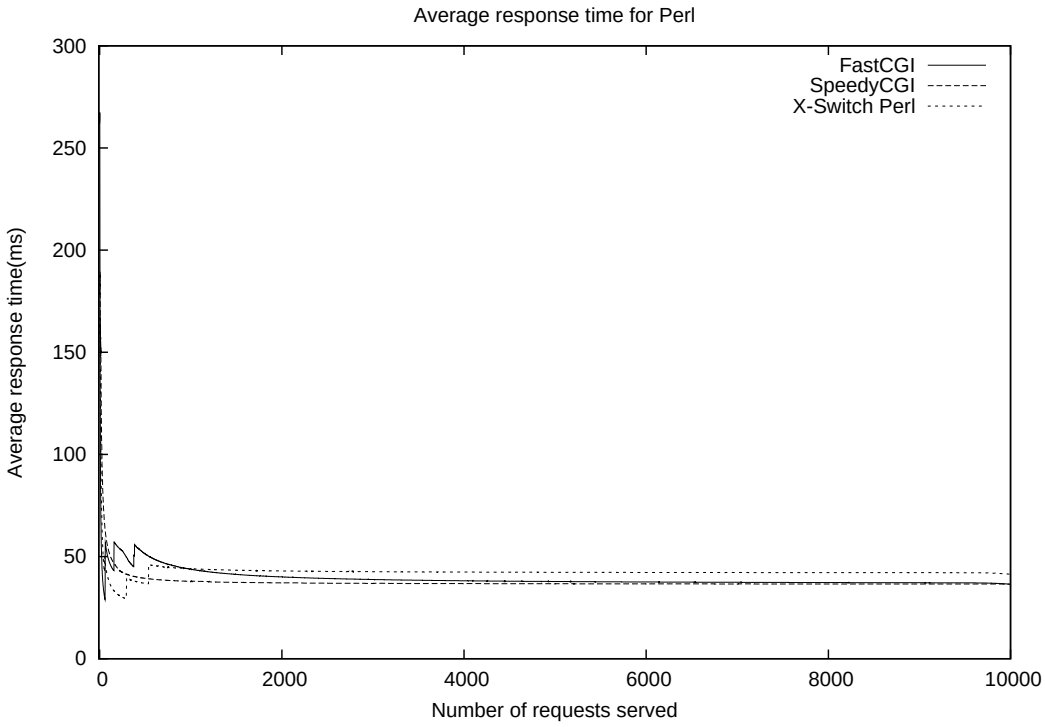

Figure 5.5: Average response time for the Perl processing engine, SpeedyCGI and FastCGI

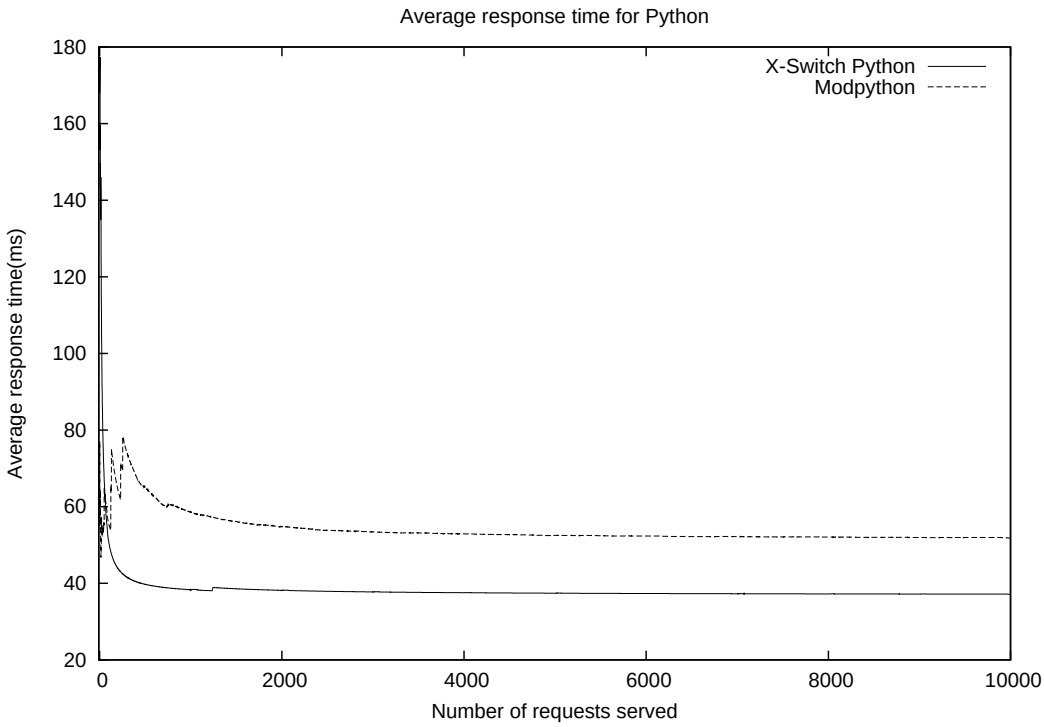

Figure 5.6: Average response for the python processing engine and Modpython

Discussion

The response time of the universal Web application server's Java engine averages to a slightly higher value than that of Apache Tomcat. The Java processing engine had an initial startup time of about 450ms and a final average response time of 54ms whereas Apache Tomcat had a final average response time of 48ms. The Python engine for the universal Web application server had a lower response time as compared to Modpython. The Python processing engine had an initial response time of about 180ms and a final average response time of 42ms whereas Modpython had a final average response time of 52ms. The performance of the universal Web application server Perl engine was similar to that of FastCGI and SpeedyCGI. The Perl processing engine had an initial startup time of about 243ms and final average response time of 39ms. FastCGI had a final average response time of 43ms and SpeedyCGI had a final average response time of 38ms. The PHP processing engine had a startup time of 234ms and a final average response time of 36ms whereas Modphp had a final average response time of 35ms. On average, the universal Web application server can perform comparably with other Web application servers. The high values for the initial response times is the result of the preparation and engine setup costs. The persistence of the processing engines however leverages this. It was anticipated that the extra layers of processing and the generality would reduce the performance but not to a great extent and thus the results confirm what was anticipated.

5.2.4 Experiment 4

Aim

The main objective of building the universal Web application server was to have a Web application server implementation that has support for multiple Web application technologies. In this experiment the performance of the universal Web application server was evaluated first by comparing how it performs when serving multiple back-end implementation languages to how it performs when serving a single back-end implementation language. Secondly the performance of the universal Web application server serving multiple implementation languages was compared to that of a set of standard Web application servers

serving different implementation languages but installed on the same instance of the Apache Web server.

Methodology

In this experiment five Jakarta-jmeter concurrent connections (threads) were used. Simple 'Hello World' Web applications were used in this experiment. The threads had a zero second ramp up period and each thread had a zero second constant timer. Thus the threads were started at the same time and there was no lapse between requests in each thread. The first set of this experiment measured the average response time for the universal Web application server serving four different kinds of back-end technologies. PHP, Python, Java and Perl back-end technologies were used. The universal Web application server served requests from four thread groups each group having five concurrent connections. Each concurrent connection issued 1000 requests for each type of back-end technology. The average response time was then computed as the average of the four thread groups' concurrent connections. In the second set, the same setup was run four times and in each run the universal Web application server served just one kind of back-end technology. The average response time was computed as the average of the four runs. In the last setup, Apache Tomcat, Modphp, Modpython and SpeedyCGI were used to serve requests from five concurrent connections. Each connection again issued 1000 requests for each kind of technology served by the different Web application servers.
Metric: Average response time

Results

The results of this experiment are graphed in Figures 5.8 and 5.7.

Discussion

The results of the experiment show that the performance of the universal Web application server does not degrade when serving multiple processing engines. The minor difference can be attributed to the fact that when serving one kind of back-end technology the universal Web application server has fewer processes residing in memory and thus less context

switching whereas when it was serving multiple kinds of back-end technologies, each back-end technology had a process residing in memory, thus increasing the context switching. There was not much of a difference between the performance of the universal Web application server and the standard Web application servers. Thus the universal Web application server can serve multiple back-end technologies as well as it can serve a single back-end technology.

5.2.5 Experiment 5

Aim

Web application responses vary in size though it is recommended for performance that the size be limited to about 50Kb [5]. In this experiment the performance of the universal Web application server was evaluated with regard to the size of the response. A good Web application server does not degrade drastically in performance with an increase in the size of the response.

Methodology

For this experiment Siege2.65 was used to simulate users. Siege issues as many concurrent requests as possible for each simulated user and each user can have more than one concurrent transaction. Seige has a metric, *concurrency*, which is an average measure of how many concurrent transactions existed for each client in order to serve all the requests that Siege issued. Siege tries to make as many transactions as possible for each client. An efficient Web server can serve the transactions fast enough for them not to increase. Siege keeps on increasing the transactions until the server goes down. As the performance degrades each client has more transactions existing concurrently. Therefore an increase in the value of Siege's *concurrency* attribute signifies degradation in server performance. Siege can therefore be configured to simulate a certain number of concurrent clients and load each concurrent client with an increasing number of concurrent transactions until the server goes down. In this experiment, 10 concurrent users were simulate and each experiment lasted 60 seconds. The initial response size was 200Kb, which is higher than the recommended

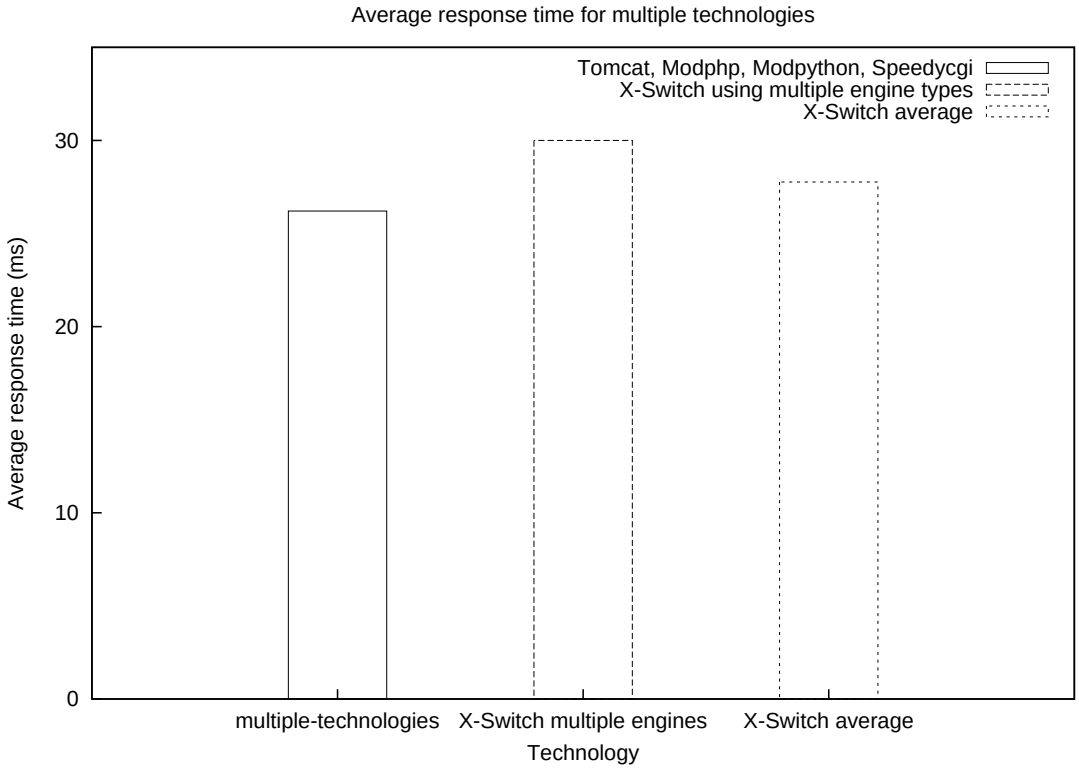

Figure 5.7: The first bar starting from the left is the average response time for a set of standard Web application servers while the second bar is for X-Switch serving multiple engines at the same time. The last bar is the average of X-Switch serving one technology at a time.

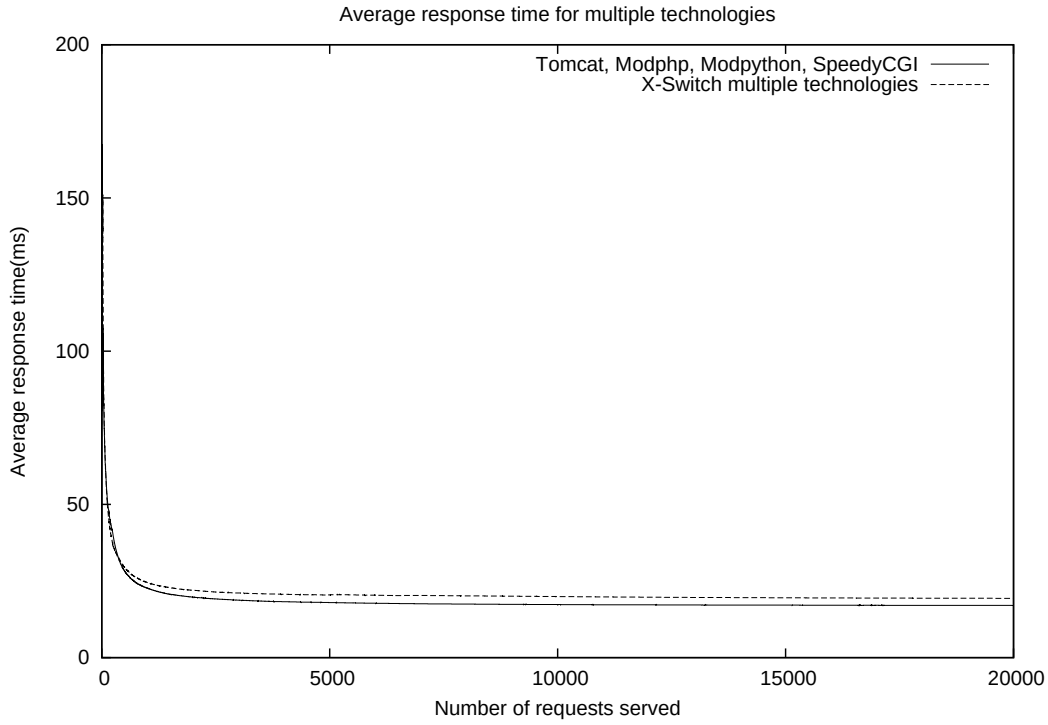

Figure 5.8: Average response time for the universal Web application server serving multiple languages and for a set of standard Web application servers

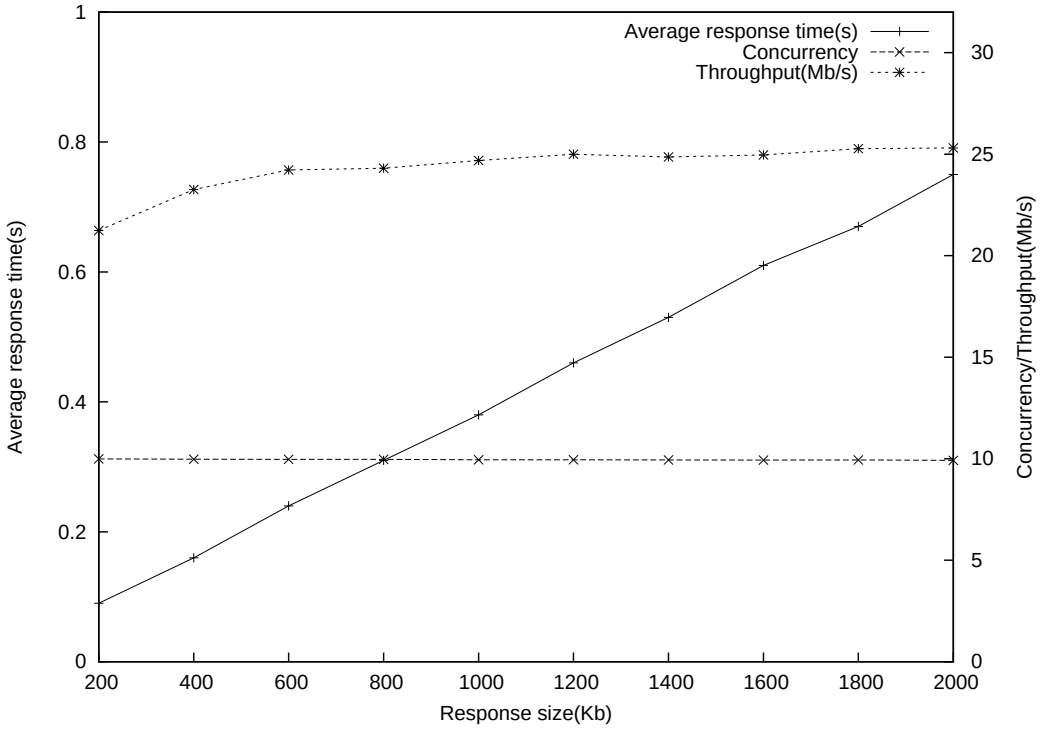

Figure 5.9: Performance of the universal Web application server with an increasing response size

file size for good performance. In each successive run, the value of the response size was increased by 200Kb. The final response size was 2Mb. For each run the response time, throughput and *concurrency* were measured and recorded.

Results

The throughput, *concurrency* and response time for the runs were measured, recorded and graphed. See Figure 5.9.

Discussion

The performance of the universal Web application server did not degrade as the response size increased. As can been seen from Figure 5.9, the *concurrency*, which is a measure of performance, did not fluctuate much. In addition the throughput increased together with the response size. As was anticipated the response also increased as the file size increased. However the increase in response time was linear and this is ideal. Thus the universal Web application server did not noticeably degrade in performance as the response size increased.

5.2.6 Experiment 6

Aim

The universal Web application server was designed to serve Web applications for a particular user using a set of processing engines (interpreters) which belong to the user and correspond to the appropriate back-end technology. Thus both popular and unpopular Web applications are served by the same set of processing engines. The aim of this experiment was to measure the effect of document popularity on response time.

Methodology

For this experiment a 43Kb Web application was deployed under five different contexts. Using Jakarta-jmeter, five concurrent connections (threads) were used in each setup. The threads had a ramp up period of zero seconds and thus started at the same time. In the first run four of the applications were made popular by giving them a zero constant timer and thus there was no lapse between consecutive requests. One application was made unpopular by giving one of the threads a constant timer of 100ms and thus making it issue its requests after a periodic delay. Thus in this setup there where four popular applications and one unpopular application.

For the second run, four applications were made unpopular and one was made popular. For this setup the threads issuing requests for the unpopular applications had a 100ms constant timer whereas the one thread issuing requests for the popular application had a zero constant timer.

The average response times for each thread in each setup was measured and recorded.

Results

The results of this experiment were recorded and graphed in two separate graphs. Figure 5.10 is for one popular application with four unpopular applications and Figure 5.11 is for one unpopular application and four popular applications.

Discussion

The results demonstrate that the popular applications performed better than the unpopular applications. This was the case when requests for one unpopular application were being made while four popular applications were being served. The same effect was noticeable even when serving requests for four unpopular applications while serving requests for one unpopular application. The popular Web applications have a better response time because they are in memory most of the time as they are served more frequently. The difference in response time was small and insignificant.

5.2.7 Experiment 7

Aim

The experiments presented thus far use synthetic load. It is however necessary to test applications using a realistic load. The realistic load follows actual load patterns which can be obtained from existing log files. The aim of this experiment was to measure the performance of the universal Web application server using realistic load patterns.

Methodology

An Apache access log file for the Web server that serves sites for the Department of Computer Science at the University of Cape Town was used in this experiment. The log file was for the last quarter of the year 2007. The log file was analysed to get the busiest time using Webalizer [7]. The busiest month was selected by getting the month with the

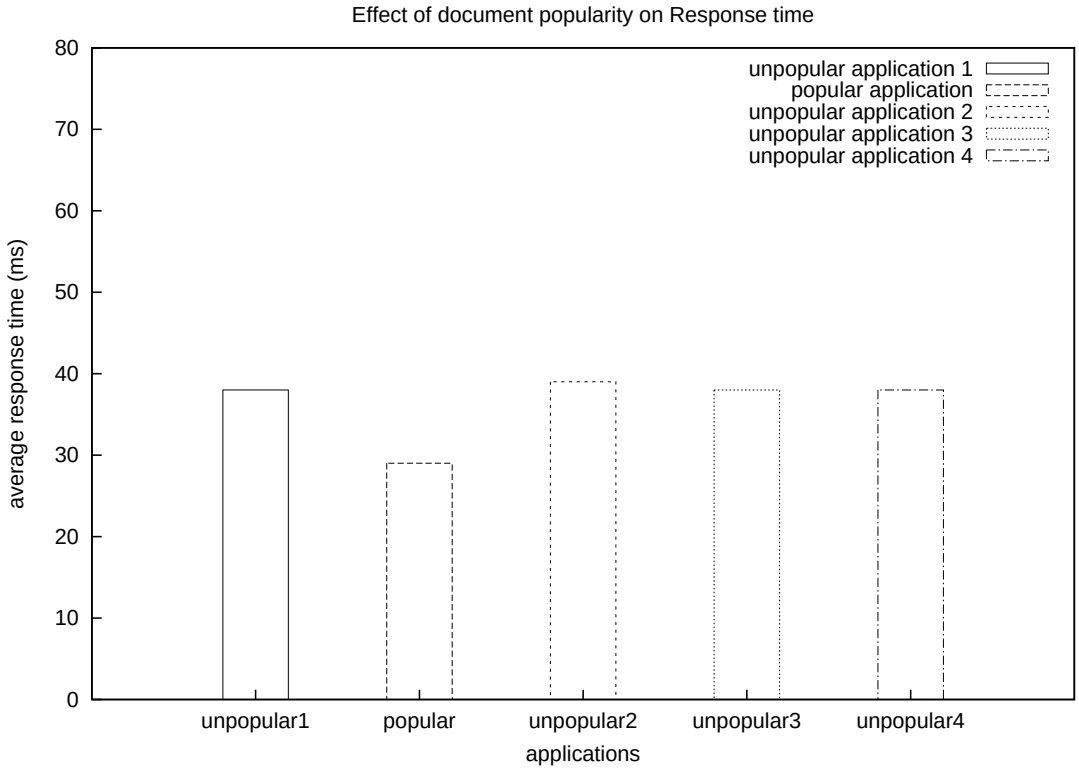

Figure 5.10: The effect on the response time of a popular document while unpopular documents are being served

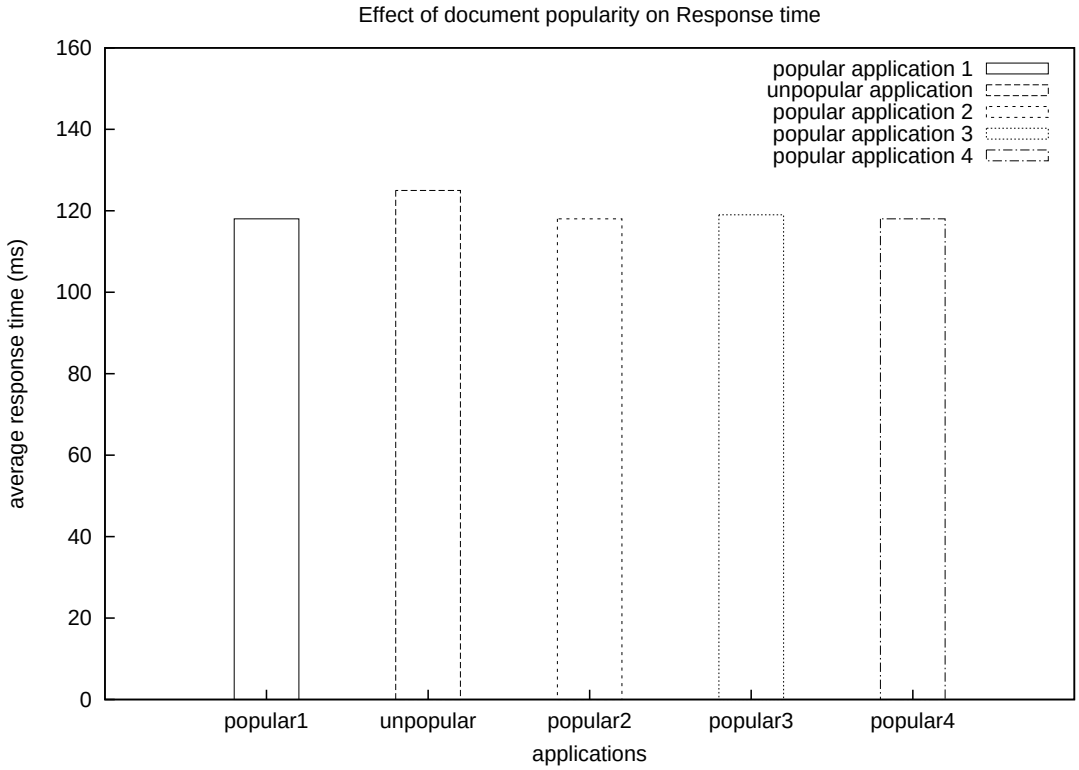

Figure 5.11: The effect on the response time of an unpopular document while popular documents are being served

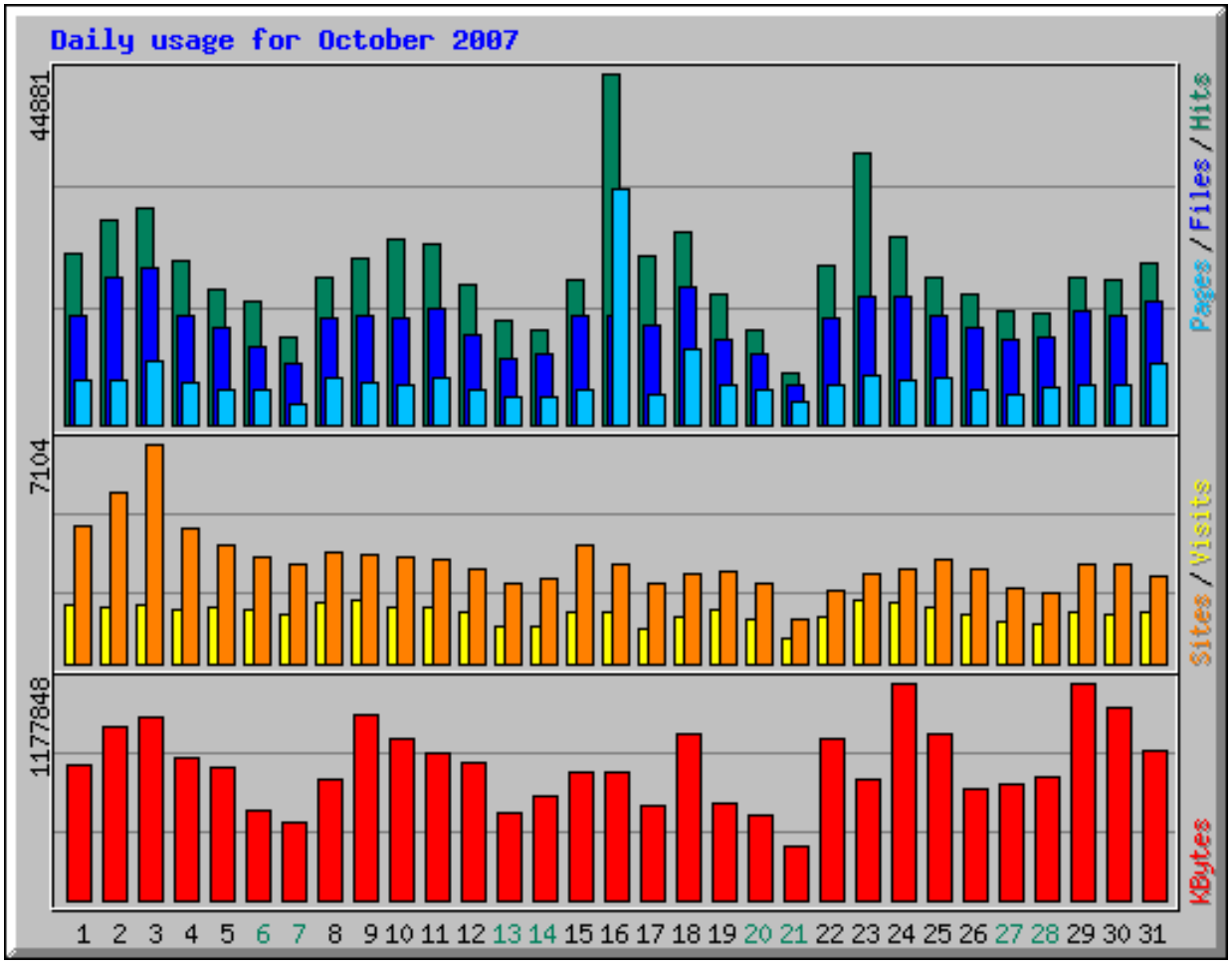

Figure 5.12: Histogram showing the busiest day of the month

maximum number of hits. Then the busiest hour, illustrated in Figure 5.13, of the busiest day, illustrated in Figure 5.12, of the month was selected. The busiest hour of the day was then analysed using a simple Perl script to get the busiest 10 minutes. The busiest 10 ten minutes were used to make a JMeter test plan that was used to run the tests. The load pattern was used on a simple Perl script that generated a 5Kb response.
The average response time was measured and recorded.

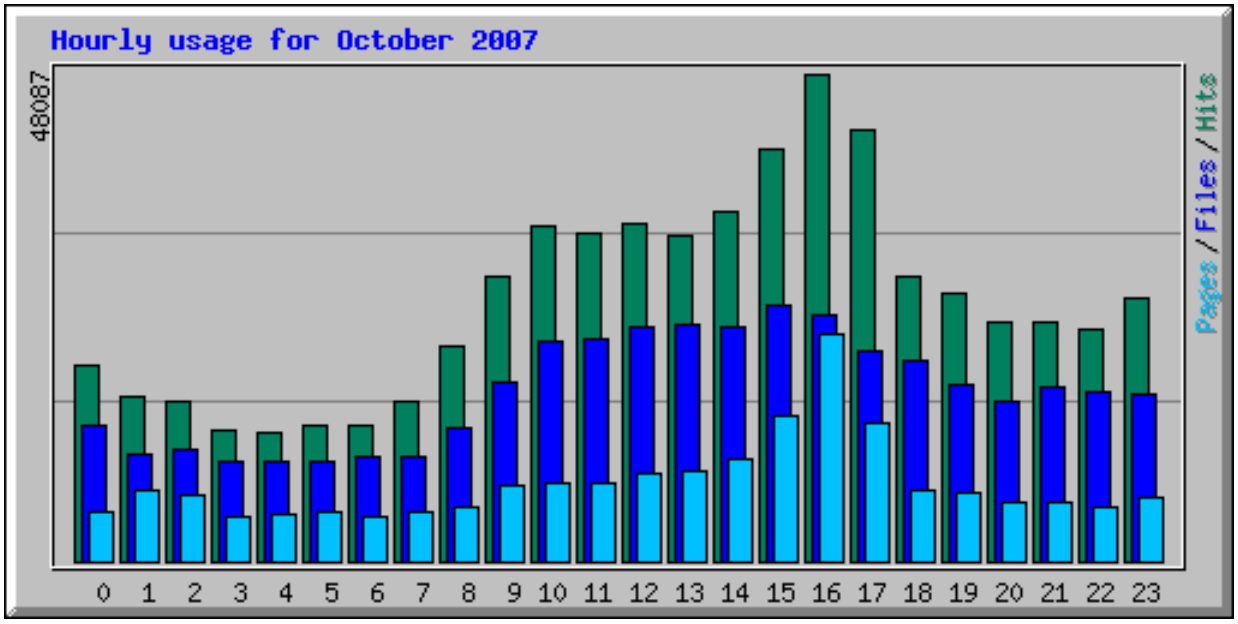

Figure 5.13: Histogram showing the busiest hour of the day

Results

The results to this experiment are presented in Figure 5.14.

Discussion

The initial response time was high as the universal Web application server was starting up processing engines. After that the average response time starts going down until it reaches a steady level and remains almost constant at 8ms. The response time pattern for the realistic load is the same as that of synthetic load patterns and thus the universal Web application server can serve realistic loads just well as it can serve the synthetic load patterns that were used.

5.2.8 Summary of performance analysis

The Perl processing engine performed as good as the SpeedyCGI and FastCGI. The Python processing engine performs much better than Modpython and the PHP processing engine

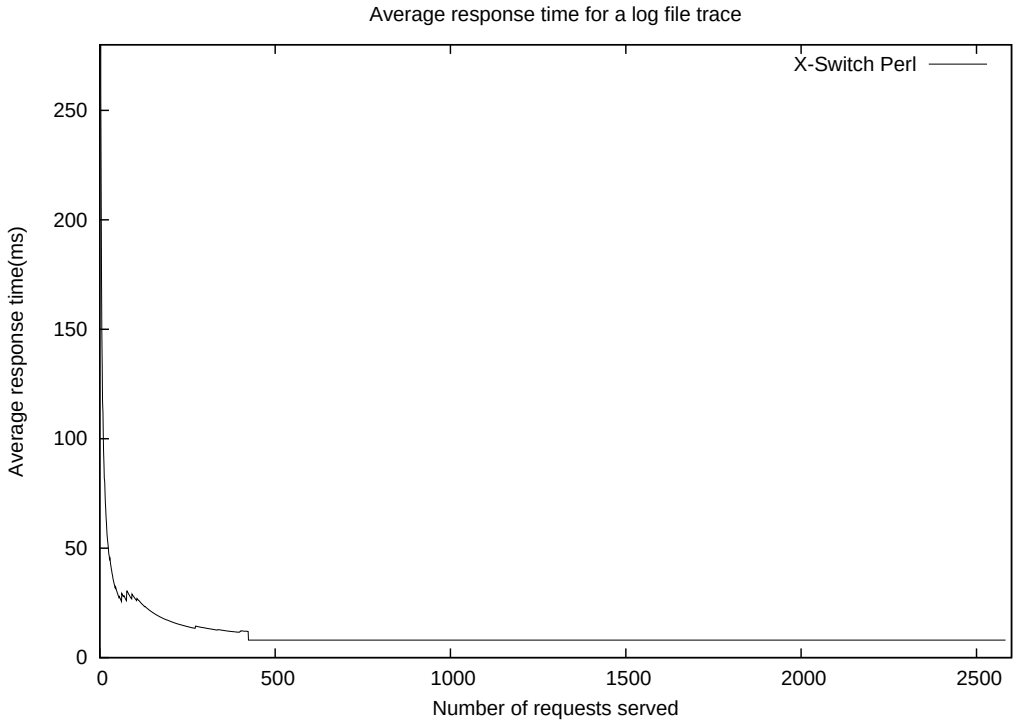

Figure 5.14: Average response time of the universal Web application serve when serving realistic load

performs just as good as Modphp. The Java processing engine also did perform competitively to Apache Tomcat. The initial response time for the universal Web application server's processing engines is high because of engine start up which involves forking a process and also creating and parsing user information. This however is leveraged by the persistence of the processing engines as subsequent requests are served. The universal Web application server also can support large responses and concurrent clients. Its performance does not vary much whether serving a single technology or multiple technologies. On average, response time for the universal Web application server serving Perl, PHP, Python and Java technologies concurrently was 19ms and that of Apache Tomcat, Modpython, Modphp and SpeedyCGI running concurrently was 17ms. Therefore the results of the performance tests prove that a universal Web application server can perform comparably to known Web application servers.

5.3 Usability testing

The design of the universal Web application server separates the logic of Web application context management from the actual execution of the Web application scripts. This requires that the logic of packaging and deploying applications be the same for all the back-end technologies. The main aim of the usability study was to evaluate how viable packaging and deploying applications for the universal Web application server is. The study involved twenty users who were required to deploy a Web application in the universal Web application server and also package and deploy another Web application in the universal Web application server. The users who comprised of undergraduate and postgraduate students were then required to fill in a questionnaire in order to get their feedback. The usability study targeted people who had knowledge of the Linux command-line.

The usability testing first started with a pilot study which involved two users before going on to the actual testing. The feedback from the pilot study, with the two users, was then used to reorganize the questionnaire and the approach to the whole test. The following sections outline the steps taken in the usability study.

5.3.1 Methodology

The users were required to begin by reading a brief background on what the usability study was all about and also what they were required to do. After this, the users were required to answer some questions which were used to collect their background information.
The first part of the exercise required the users to deploy a Web application in the universal Web application server. After this, the users were then required to install a similar Web application in Tomcat using the Tomcat Web manager interface. In both cases the users were deploying the Web applications on a remote machine.
In the second part of the exercise, the users were required to package a Web application of their choice from a list of Web applications. The users were then provided with pre-written components of a Web application and all they had to do was package the application for deployment. The users then deployed the applications after packaging them.

5.3.2 Pilot study

The pilot study involved two users and was conducted using the procedure and steps outlined in the previous section. Even though the study targeted users who had knowledge of the Linux command-line it was noticed that the two users needed help with the correct syntax for most of the commands. Thus the questionnaire was revised to include the syntax for the commands that were required to complete each step of the exercise. The changes to the questionnaire are the only changes which were made before proceeding on to the actual user testing.

5.3.3 Experiment

There was no difference between the way the pilot and the actual study was conducted except for the changes in the questionnaire.

5.3.4 Results

The users who participated in the study had various backgrounds. Most of the users who participated in the exercise had a computer science background (see Table 5.3). Of the 18

Table 5.2: The number of users who participated by year of study

Year of Study	Number of users
1st	5
2nd	4
Honours	2
Masters	7

Table 5.3: The number of users by field of study

Field of study	Number of users
Computer Science	15
Electrical engineering	1
Civil engineering	2

users 4 had no experience with installing Web applications. All the users had knowledge of Web applications and the majority, that is 11 of the users, were familiar with Java Web applications while the least known Web application technology was Python. The responses to the non-free style questions are tabulated in Table 5.2, 5.3, 5.4 and 5.5

5.3.5 Discussion

The results show that most of the users found the deploying of Web applications in the universal Web application server easy. In addition most of the users were satisfied with the amount of time it took to complete the deploying of a Web application. It took two minutes, on average, for the users who were comfortable with the command-line to complete the deployment. Users who were less comfortable with the command-line took 8 minutes on average to finish the deployment of a Web application. More than half of the users agreed that installing Web applications using the universal Web application server was easier than using the Apache Tomcat manager interface. In their comments, the users said navigating

Table 5.4: Responses to questions related to deploying of Web applications

Question	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
Installing Web applications in the universal Web application server is easy	8	7	2	1	0
Overall, I am satisfied with the ease of installing Web applications in the universal Web application server	9	6	0	2	1
Overall, I am satisfied with the amount of time it took to complete the installation of the Web application in the universal Web application server	10	5	1	2	0
Installing Web applications in the universal Web application server would make it easier to install Web application	5	8	3	1	1
Installing Web applications in the universal Web application server is easier than installing Web applications in Tomcat	7	2	8	1	0

Table 5.5: Responses to questions related to packaging of Web applications

Question	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
Packaging Web applications for deployment in the universal Web application server is easy	6	3	6	3	0
Overall, I am satisfied with the ease of packaging Web applications in the universal Web application server	5	7	4	1	1
Overall, I am satisfied with the amount of time it took to complete the packaging of the Web application	7	5	2	3	0

to the directory in which they deploy the Web application made the installation easier than Apache Tomcat where they lost control of the Web application. Some users suggested that a Web interface similar to that of Apache Tomcat be developed to make deploying Web applications easier. However, the majority of the users were satisfied with the overall process of deploying Web applications in the universal Web application server.

The packaging was not as easy as the deployment to most of the users. Half of the users said that the packaging process was easy. Most of the users were also satisfied with the amount of time it took to complete the packaging of a Web application. Amongst the negative aspects noted was the number of steps it took to complete the packaging. The users also suggested that the process of packaging should be reduced to a single command-line command which should invoke a script that automates the steps of packaging. It was also observed that using the command-line for packaging was irritating to the users as it was easy for them to make syntax errors which were not easy to retrace. On average it took 4 minutes for the users to finish packaging a Web application.

Thus, in general, packaging Web applications as well as deploying the Web applications in the universal Web application server can be deemed to be a viable approach to package and deploy Web applications.

5.4 Case study

In all the experiments described in the previous sections, simple Web applications were developed and used for the experimentation. This section describes the packaging and deploying of phpBB2 in the universal Web application server. The purpose of deploying phpBB in the universal Web application server was to show that the universal Web application server can host real applications.

5.4.1 What is phpBB?

phpBB is a widely used customizable open source bulletin board package [23]. It is a winner of many awards amongst which is SourceForge's 'Best Project for Communications' in July 2007. phpBB is written in the PHP programming language.

5.4.2 Implementation

phpBB, like most PHP Web applications, is designed to process requests for Web applications components by including the component name in the URL that invokes the request. In addition, internal redirects require that the component names be added to the redirected URL. In order to package and deploy phpBB in the universal Web application server, the redirects for phpBB code had to be changed. The modified phpBB required that a URL pattern that corresponds to the component that was to handle the request be included in the URL that was used to invoke the Web component. For example, the original phpBB's index page URL was '*http://localhost:8080/backslash/∼username/phpBB2/index.php*' whereas for the modified phpBB the URL was '*http://localhost:8080/∼username/phpBB2/index*'. This required the writing of a '*weblet.xml*' for phpBB. The *weblet.xml* file mapped URL patterns to actual paths to the components that handled the requests (See Figure 5.15).

```
<web-app>

     <weblet>
          <name>main_index</name>
          <path>weblets/index.php</path>
          <type>php</type>
     </weblet>
     <weblet-mapping>
          <name>main_index</name>
          <url-pattern>/index</url-pattern>
     </weblet-mapping>

     <weblet>
          <name>member_list</name>
          <path>weblets/memberlist.php</path>
          <type>php</type>
     </weblet>
     <weblet-mapping>
          <name>member_list</name>
          <url-pattern>/memberlist</url-pattern>
     </weblet-mapping>

     <weblet>
          <name>faq</name>
          <path>weblets/faq.php</path>
          <type>php</type>
     </weblet>
     <weblet-mapping>
          <name>faq</name>
          <url-pattern>/faq</url-pattern>
     </weblet-mapping>
```

Figure 5.15: A snippet of the weblet.xml file for the modified phpBB

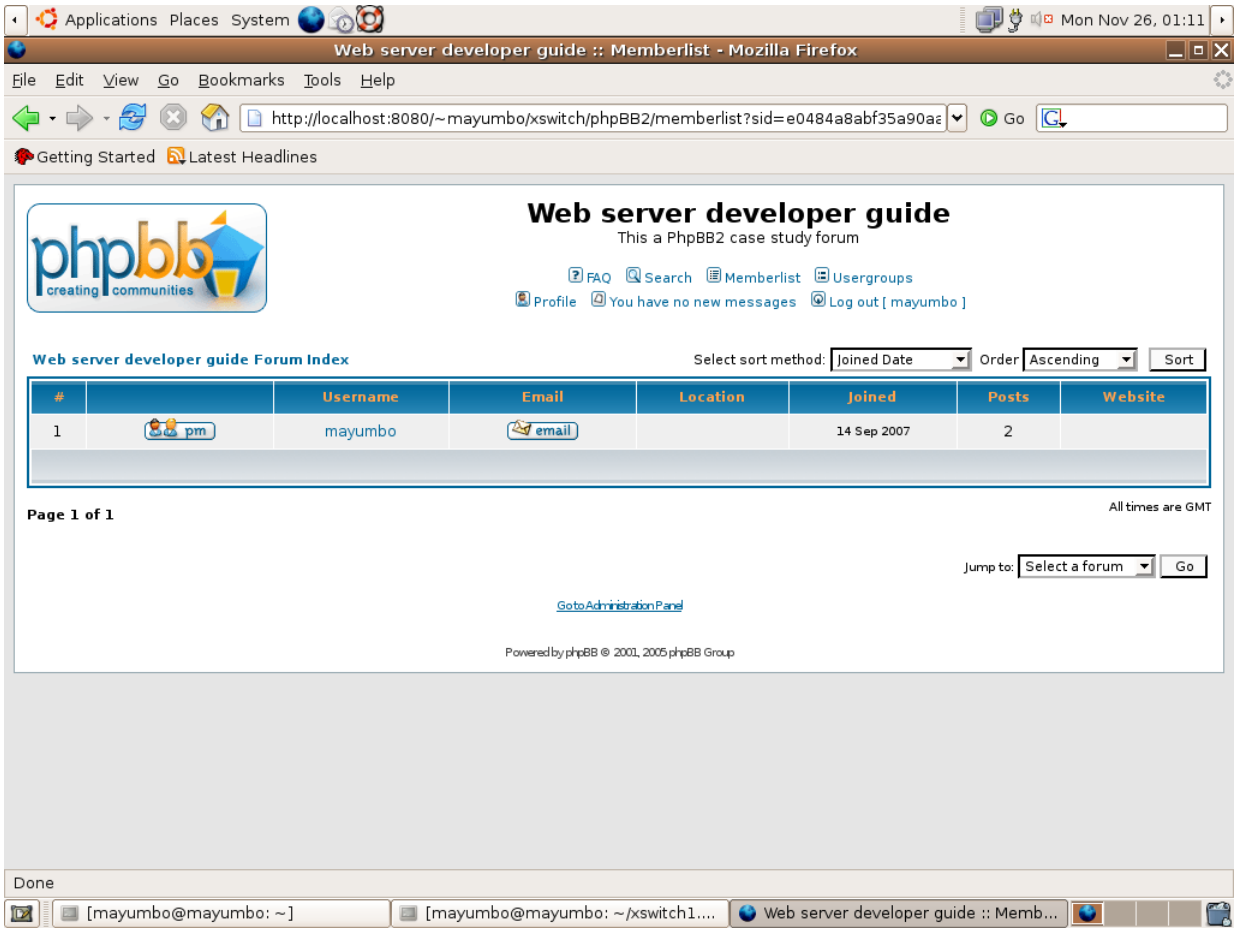

Figure 5.16: A screen shot of the modified phpBB showing a typical URL pattern

In addition to this change, the directory structure for phpBB also was changed. All the components that were invoked in order to process requests were placed in the *weblets* directory. The PHP processing engine for the universal Web application server uses the command-line SAPI for PHP and thus the phpBB headers had to be hard coded into the application. The packaged application was then archived into a xar file ready for deployment. A screenshot of the deployed phpBB is shown in Figure 5.16 and also shows the URL patterns which do not expose the directory structure of the Web application and the implementation technology.

5.4.3 Discussion

The major challenge faced in the implementation was the fact that, like most PHP Web applications, phpBB closes the interpreter. Also, the fact the command-line SAPI for PHP does not parse headers meant that all the locations in the code that had headers be hard-coded to generate the right headers. The second complication which came with the headers was getting to determine when to parse the last headers so as to include all the headers that are to be generated for a particular response. Generating the headers early resulted in errors involving header generation when the headers were already parsed. Using the universal Web application server framework for Web application deployment also made it unnecessary to know the Web application path that phpBB uses to compute the redirect URLs. This is because using URL patterns made the requests independent of the directory paths and thus the X-Switch module is the one that did determine which component to run for the request and not the Web server that did the redirecting. The installation of phpBB, a widely recognized application, shows that the universal Web application server can serve real world applications.

5.5 Summary

The installation of phpBB in the universal Web application server showed that the universal Web application server can host real applications. In addition the usability study showed that installation of Web applications in the universal Web application server using the universal Web application server's framework for Web application deployment is a viable approach. The performance tests showed that although the universal Web application server does not outperform known single technology Web application servers, it can still perform comparably to these servers. Moreover, the performance tests also showed that the universal Web application server has support for at least 375 concurrent connections and that its performance does not degrade drastically with an increasing response size. Therefore, from the results of the evaluation, it could be argued that the universal Web application server is feasible, easy to use and can perform comparably with other Web application servers.

Chapter 6

Discussion and concluding remarks

The widespread use of the Web and its fame has led to the development of a variety of Web application technologies. These technologies each in their own way try to solve different aspects of the problems associated with Web application technologies independently of one another. The main problem is performance degradation and security in multi-user environments. This research presented the universal Web application server as a generalization of the existing solutions. This chapter analyses the implementation of the universal Web application server and presents conclusions drawn from the evaluation studies. It also puts forward some shortcomings of the implementation and possible solutions.

6.1 Generalization of the universal Web application server

The modular architecture of the universal Web application server proves to be a viable approach for a Web application server that supports multiple Web application implementation technologies. Performance tests on the impact of the layers of abstraction show that they have little impact on the total processing time. Sandboxing of Web application scripts is achieved by assigning each user a processing engine. The processing engines are run with the privileges of the owner of the script and therefore offering a secure shared environment. This is achieved by invoking the processing engines using suexecme, the X-Switch system wrapper script. The performance of the X-Switch system is not compromised because the processing engines are run persistently. PHP, Perl, Java and Python processing engines were implemented successfully proving that the X-Switch system can be used to implement multiple Web application technologies. Web application deployment is made easy in X-Switch by using the framework for easy Web application deployment.

6.2 Conclusion

The implementation of the X-Switch system demonstrates the feasibility of a universal Web application server. In addition, the installation of phpBB2 in the X-Switch system shows that the framework for easy Web application deployment is a feasible approach and that the universal Web application server can serve realistic applications. Performance tests show that the performance of the universal Web application server is comparable to known Web application servers. The usability study also reveals that the packaging and deploying of Web applications in the universal Web application server is easy. User comments and responses show that the packaging and installation is an easy to learn process. The implementation and evaluation of the universal Web application server therefore shows that

1. A universal Web application server that is a

   (a) generalization of the existing solutions, and

   (b) implements a framework for easy Web application deployment that is universal to the back end technologies

   is feasible.

2. A universal Web application server can perform comparably to other Web application servers.

3. A universal Web application server is easy to use.

The evaluation also shows that separating the core functionality of a Web application server from Mod_x using a modular design has little impact on the performance of a Web application server.

The universal Web application server makes the programming language used to implement the Web application transparent to the user. The framework for Web application deployment makes Web application programming easier to learn in that a developer would only need to master one framework as opposed to mastering different frameworks and programming languages. In addition a user can easily implement multiple languages without going through the process of configuring the different technologies. Component based systems

can become easy to manage as there is no need to configure all the Web servers on which the components are deployed. This is because the components carry their configuration in the deployment descriptor. Implementing load balancing using migrating components in High Performance computing becomes easier with the universal Web application server. This is because the components of varying implementation languages can easily migrate between different servers without the developer having to worry about allocating and configuring special directories for each technology/component. These components can be implemented using any programming language thus giving the developer a wider choice of implementation languages.

The universal Web application server also can be used in education institutions and training environments where multiple implementation technologies are required and used. The universal Web application server also makes administration easier in shared environments. In addition Web hosts can use a single server to provide services for multiple implementation technologies as opposed to dedicating machines to each single technology.

6.3 Obstacles

The implementation of the universal Web application server relies on an efficient communication protocol amongst its modules. To achieve efficient communication between X-Switch and the processing engines, the X-Switch system uses two channels of communication. The processing engines are mutated from a process that has a C programming language parent process. These engines inherit file descriptors for communication from this parent process. However not all programming languages have libraries that can be used to access these file descriptors. This was evident with the Java and PHP programming languages. Accessing these file descriptors required implementing specialized modules.

Most of the Web application technologies have evolved to conform to orthodox programming approaches. For example, most PHP Web applications are programmed with the view of terminating processes on completion. This was revealed during the installation of phpBB2 in the universal Web application server. In addition the PHP commandline SAPI which was used to implement the PHP processing engine does not parse the headers.

Implementing session handling capabilities with the servlet API was difficult because the servlet API requires requests to be handled by threads that share memory. X-Switch servlet engines are forked from a parent process that does not know the state of the running servlet engines. Thus session variables cannot be shared between processes when there are multiple servlet processing engines.

Another obstacle that was encountered was keeping the connection between Mod_x and X-Switch open for subsequent requests. The current implementation closes the connection after a request is served. The connection is closed to signal the end of the response from X-Switch. However, establishing the TCP connection is expensive and hence keeping it open would improve the performance of the universal Web application server.

6.4 Future work

6.4.1 Persistent TCP connections

Mod_x currently waits for X-Switch to close a connection as a signal for the end of the response. Developing a protocol that can signal the end of a response without closing the connection would improve the performance of the universal Web application server.

6.4.2 Web application packaging and deployment tools

It was noted from the comments and feedback from the users during the usability study that implementing tools that automate the packaging of Web applications for deployment in the universal Web application server would make packaging much easier. In addition most users are familiar with graphical user interfaces. Hence a Web-based interface for Web application deployment would also help in making the process of deploying Web applications in the universal Web application server easier.

6.4.3 Session information management

Web applications implemented using technologies like CGI and PHP store their session variables in databases or flat files. This makes it possible to access the session variables

from processes that do not share memory. Such a solution is easy to implement as a general solution for different back end technologies. This could be done in X-Switch which manages all requests. Session variables could be stored in a database and the database could be managed by X-Switch. The processing engines can then access the session variables from the database during the request processing stage. Access to these variables and databases can be controlled using access keys that are unique to groups of processing engines.

6.4.4 Python processing engine

The Web Server Getway Interface (WSGI) seeks to solve the problem of the many existing frameworks for Python Web applications. It defines an interface that Python Web application servers should implement in order to make Python Web applications portable between the servers. The X-Switch Python processing engine can potentially be extended to implement WSGI interface.

References

[1] Apache suexec support. http://httpd.apache.org/docs/1.3/suexec.html.

[2] Cgiwrap - user cgi access. http://www.unixtools.org/cgiwrap.

[3] Web aim api reference [online]. http://developer.aim.com/ref_api. 2007.

[4] Netscape server api (nsapi) introduction. http://support.zeus.com/, December 2005.

[5] Optimizing web pages- maximum page size. http://www.webdevelopersnotes.com/, 2007.

[6] Apache Software Foundation. *Apache JMeter user's manaul*, 2007.

[7] Barrett, B. The webalizer - what is your web server doing today? http://www.mrunix.net/webalizer/, November 2006.

[8] Berners-Lee, T. www: past, present, and future. *Computer 29-10* (August 1996).

[9] Bicking, I. *A Do-It-Yourself Framework*, July 2006. http://pythonpaste.org/.

[10] Bicking, I. *Paste Deployment*, July 2006. http://hoohoo.ncsa.uiuc.edu/.

[11] Bradel, B., and Drula, C. A study of the thread and event concurrency models for web servers. Tech. rep., University of Toronto, 2003.

[12] Bush, V. As we may think. *Atlantic monthly* (July 1945).

[13] Coar, K. Writing modules for apache 1.3. In *Proceedings of the OReilly Open Source Conference* (Monterey, California, August 1999).

[14] Delisle, P., Luehe, J., and Roth, M. *Java Server Pages Specification Version2.1.* Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, California 95054, U.S.A, May 2006.

[15] Dickerson, N. C. Apache htaccess for php web application deployment. http://www.wmtips.com/Apache/apache-htaccess-for-php-web-application-468.htm, July 2006.

[16] Fielding, R., Irvine, U., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. *RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1.* The Internet Soceity, 1999.

[17] Fleury, M., Stark, S., and Norman, R. *JBoss 4.0 The Official Guide.* Sams Publishing, 2005.

[18] Fulmer, J. Siege readme. http://www.joedog.org/Siege/Reamde, May 2006.

[19] Gundavaram, S. *CGI Programming on the World Wide Web*, first ed. OReilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472., 2006.

[20] Hanegan, K. *Custom CGI Scripting with Perl.* John Wiley & Sons, Inc, 605 Third Avenue, New York, U.S.A, 2001.

[21] Heinlein, P. Fastcgi. *Linux journal* (November 1998).

[22] Horrocks, S. Introduction to speedycgi. In *Proceedings of Yet Another Perl Conference North America* (2001).

[23] Jason, K. What is phpbb? http://www.phpbb-design.com/.

[24] Knudsen, C. Php version 4. *Linux journal* (November 1999).

[25] Lauer, H., and Needham, R. On the duality of operating systems structures. In *Proceedings of the 2nd International Symposium on Operating Systems, IR1A, Oct. 1978, reprinted in Operating Systems Review* (April 1979), pp. 3–19.

[26] Laurie, B., and Laurie, P. *Apache: The Definitive Guide*, second ed. OReilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472., 1999.

[27] Lerner, R. At the forge: Server-side java with jakarta-tomcat. *Linux journal* (April 2001).

[28] Marsching, S. suphp. http://www.suphp.org/, 2006.

[29] Maunder, A., and van Rooyen, R. Universal web server: The x-switch system. Tech. Rep. CS04-20-00, Department of Computer Science, University of Cape Town., 2004.

[30] Maunder, A., van Rooyen, R., and Suleman, H. Designing a 'universal' web application server. In *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries SAICSIT '05* (July 2005).

[31] Microsoft Corporation. *Internet Server API (ISAPI) Extensions*, 2007.

[32] Moseley, B. Wombat: Servlets for perl. In *Proceedings of the O'Reilly Open Source Convention 2001* (San Diego, California, USA, July 2001).

[33] Mourani, G., and Maddy, M. *Securing and Optimizing Linux RedHat Edition -A Hands on Guide*. Open NA, 2000.

[34] Mutton, P. November 2007 web server survey. *Netcraft news* (October 2007).

[35] ncsa. *The Common Gateway Interface specification*, 1999. http://hoohoo.ncsa.uiuc.edu/cgi/overview.html.

[36] Nelson, M. L., de Sompel, H. V., Liu, X., Harrison, T. L., and McFarland, N. mod_oai: An apache module for metadata harvesting. Tech. Rep. arXiv:cs/0503069v1 [cs.DL], Cornell University, March 2005.

[37] Nourie, D. Books to shorten your learning curve. Tech. rep., Sun microsystems, March 2001.

[38] Ousterhout, J. Why threads are a bad idea (for most purposes). In *Proceedings of USENIX Annual Technical Conference* (January 1996).

[39] Pariag, D., Brecht, T., Harji, A., Buhr, P., Shukla, A., and David, R. Comparing the performance of web server architectures. In *Proceedings of EuroSys'07* (Lisboa, Portugal, March 2007).

[40] Raggett, D., Hors, A. L., and Jacobs, I. *HTML 4.01 Specification W3C Recommendation*. W3C, 1999.

[41] Ramsdell, B. *RFC 2633 - S/MIME Version 3 Message Specification*. The Internet Soceity, 1999.

[42] Searls, R. *Java 2 Platform, Enterprise Edition Deployment API Specification Version: 1.1*. Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, California 95054, U.S.A, November 2003.

[43] Stein, L., and MacEachern, D. *Writing Apache Modules with Perl and C*, first ed. OReilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472., March 1999.

[44] Stein, L. D. Sbox: Put cgi scripts in a box. In *Proceedings of the USENIX Annual Technical Conference* (Monterey, California, USA, June 1999).

[45] Sun Microsystems, Inc. *JAVA SECURITY OVERVIEW White Paper*, April 2005. http://hoohoo.ncsa.uiuc.edu/cgi/overview.html.

[46] von Behren, R., Condit, J., and Brewer, E. Why events are a bad idea (for high-concurrency servers). In *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems* (Lihue, Hawaii, USA, May 2003), pp. 19–24.

Appendix A

X-Switch Web application implemented interfaces and APIs

A.1 X-Switch Java servlet engine

A.1.1 ServletRequest

public String getUri()

public boolean isSecure()

public String getCharacterEncoding()

public int getContentLength()

public String getContentType()

public ServletInputStream getInputStream()

public String getParameter(String name)

public Map getParameterMap()

public Enumeration getParameterNames()

public String[] getParameterValues(String parameter)

public String getProtocol()

public BufferedReader getReader()

public String getRemoteAddr()

public String getRemoteHost()

public String getScheme()

public String getServerName()

public int getServerPort()

public String getRemoteUser()

public String getQueryString ()
public String getMethod()
public int getIntHeader(String name)
public Enumeration getHeaderNames()
public String getHeader(String name)
public long getDateHeader(String name)
public Cookie[] getCookies()
public String getAuthType()
public static String parseCharacterEncoding(String contentType)

A.1.2 ServletResponse

public String getCharacterEncoding()
public ServletOutputStream getOutputStream()
public PrintWriter getWriter()
public void setContentLength(int len)
public void setContentType(String type)
public void addCookie(Cookie cookie)
public boolean containsHeader(String name)
public String encodeUrl(String url)
public String encodeRedirectUrl(String url)
public void sendRedirect(String location)
public void setDateHeader(String name, long date)
public void addDateHeader(String name, long date)
public void setHeader(String name, String value)
public void setIntHeader
public void addIntHeader
public void setStatus(int sc)
public void setStatus(int sc, String sm)
protected void sendHttpHeaders()

public static String encodeCookie(Cookie cookie)

public static final String findStatusString(int sc)

A.2 PHP GLOBALS implemented by modphp

$_GET[]

$_COOKIE[]

$_POST[]

$_SERVER['QUERY_STRING']

$_SERVER['SERVER_SOFTWARE']

$_SERVER['HTTP_REFERER']

$_SERVER['REQUEST_METHOD']

$_SERVER['HTTP_USER_AGENT']

$_SERVER['HTTP_ACCEPT_CHARSET']

$_SERVER['HTTP_ACCEPT']

$_SERVER['HTTP_ACCEPT_ENCODING']

$_SERVER['HTTP_ACCEPT_LANGUAGE']

$_SERVER['CONTENT_TYPE']

$_SERVER['CONTENT_LENGTH']

$_SERVER['SERVER_NAME']

$_SERVER['PORT']

$_SERVER['REMOTE_ADDR']

$_SERVER['REMOTE_HOST']

$_SERVER['REMOTE_USER']

$_SERVER['SERVER_PROTOCOL']

$_SERVER['AUTH_TYPE']

A.3 Perl processing engine CGI environment variables

$ENV{‘QUERY_STRING’}
$ENV{‘AUTH_TYPE’}
$ENV{‘CONTENT_LENGTH’}
$ENV{‘CONTENT_TYPE’}
$ENV{‘GATEWAY_INTERFACE’}
$ENV{‘HTTP_ACCEPT’}
$ENV{‘HTTP_USER_AGENT’}
$ENV{‘PATH_INFO’}
$ENV{‘PATH_TRANSLATED’}
$ENV{‘REMOTE_ADDR’}
$ENV{‘REMOTE_HOST’}
$ENV{‘REMOTE_IDENT’}
$ENV{‘REQUEST_METHOD’}
$ENV{‘REMOTE_USER’}
$ENV{‘SCRIPT_USER’}
$ENV{‘SERVER_NAME’}
$ENV{‘SERVER_PORT’}
$ENV{‘SERVER_PROTOCOL’}
$ENV{‘SERVER_SOFTWARE’}

Appendix B

X-Switch Web application usability study

B.1 X-Switch Web application server usability questionnaire

B.1.1 Introduction

Thank you for agreeing to participate in the evaluation of the usability of the universal Web application server.

The first section of the questionnaire outlines a brief background of the universal Web application server. You can read the background information at your own time. The sections thereafter outline how to package and deploy Web applications in the universal Web application server. You will be required to deploy a Web application in the universal Web application server in **section A** of the installation manual and thereafter deploy a similar Web application in Tomcat in **section B** of the installation manual. In **section C** you will be required to package an application (webapp) for deployment in the universal Web application server. Finally you will be required to answer questions related to the deployment and packaging of Web applications in the universal Web application server.

B.1.2 Background information

The universal Web application server is a Web application server implementation that has support for multiple authors in a secure environment with support for multiple implementation technologies. It uses a framework for Web application deployment that is universal

to the implementation technologies and thus abstracts the implementation technology from the user and also aims at making Web application deployment in such a universal environment easy.

The Web application framework for deploying Web applications uses a directory structure that has a deployment descriptor file (weblet.xml) in the root of the directory and is deployed as an archice (.xar archive). The scripts which are run to implement the various functionalities of the Web application are located in the weblets direectory. The other resources required by the Web application can be stored in user defined directories at the root level of the directory structure.

The author of the Web application deploys the Web application in the xswitch directory in their public_html directory as a xar archive. The universal Web application server then loads the Web application when processing the first request for the Web application. With the universal Web application server and its Web application deployment framework the author still has access to their Web application while retaining the advantages of easy deployment of Web applications.

Please try to respond to all the items. Begin by answering the following questions about your background with Web applications.

1. Year of study:

2. Program of study e.g MSc Computer Science:

3. Rate your experience with using Web applications on a scale of 1 to 5 where 1 is NONE and 5 is good.

   .....

4. Have you ever installed a Web application?
   YES NO

5. Which Web application technology do you have knowledge of?
   PERL PYTHON JAVA PHP NONE

6. Rate your experience with installing Web applications on a scale of 1 to 5 where 1 is NONE and 5 is GOOD.

Table B.1: Table of sample Web applications that you can download and deploy

Web application name	Type	Download command
java	Java	**wget 137.158.59.245/java.xar**
perl	Perl	**wget 137.158.59.245/perl.xar**
PHP	PHP	**wget 137.158.59.245/php.xar**
python	Python	**wget 137.158.59.245/python.xar**

.....

B.1.3 Installation manual

Section A

This section of the manual outlines the installation of a simple Web application in the universal Web application server.

1. ssh into test@137.158.59.173 with password 'xswitch'.
   ssh test@137.158.59.173

2. Change directory to the '/home/test/public_html/xswitch' directory
   cd /home/test/public_html/xswitch

3. Download a sample Web application of your choice from the locations listed Table B.1 using the Linux/Unix 'wget' command.

4. In your Web browser enter the URL from Table B.2 that corresponds to the Web application that you installed to view your installed Web application.

Section B

This section of the manual outlines the installation of a simple Web application in Tomcat using Tomcat's html manager interface.

Table B.2: Table of URLs that correspond to the type of Web application insalled

Web application	Type	URL
java	Java	**http://137.158.59.173:8080/~test/xswitch/java/pifinder**
perl	Perl	**http://137.158.59.173:8080/~test/xswitch/perl/pifinder**
PHP	PHP	**http://137.158.59.173:8080/~test/xswitch/php/pifinder**
python	Python	**http://137.158.59.173:8080/~test/xswitch/python/pifinder**

1. In your Web browser enter the URL http://137.158.59.173:8085/manager/html and when prompted for authentication, enter username 'test' and password 'test' to access the manager interface.

2. Under the war file to deploy section of the deploy section in the manager interface enter the URL of the sample Web application to deploy and click on the deploy button to deploy.
   URL: **/home/mayumbo/tomjava.war**

3. In your browser enter the URL http://137.158.59.173:8085/tomjava to view your installed Web application.

Section C

This section of the manual outlines how to package a Web application (webapp) for intallation in the universal Web application server.

1. ssh int tes@137.158.59.173 with password 'xswitch'
   ssh test@137.158.59.173

2. Change directory to the /home/test directory
   cd /home/test

3. Create a directory webapp
   mkdir webapp

4. Change to the webapp directory
   cd webapp

5. Copy the sample weblet.xml file into the current directory using the 'cp' command

   (a) java: **cp /home/test/components/java/weblet.xml weblet.xml**

   (b) perl: **cp /home/test/components/perl/weblet.xml weblet.xml**

   (c) PHP: **cp /home/test/components/php/weblet.xml weblet.xml**

   (d) python: **cp /home.test/components/python/weblet.xml weblet.xml**

6. Create directory weblets and change directory to the weblets directory
   mkdir weblets
   cd weblets

7. Copy the sample script file into the current directory using the 'cp' command

   (a) java: **cp /home/test/components/java/example.class example.class**

   (b) perl: **cp /home/test/components/perl/example.pl example.pl**

   (c) PHP: **cp /home/test/components/php/example.php example.php**

   (d) python: **cp /home.test/components/python/example.py example.py**

8. Change directory to the /home/test directory
   cd /home/test

9. Package the applicatio using the zip application
   zip -r webapp.xar webapp

10. Copy the application to the /home/test/public_html/xswitch directory to deploy the application in the universal Web application server.
    cp /home/test/webapp.xar /home/test/public_html/xswitch

11. In your browser enter the URL
    http://137.158.173:8080/~test/xswitch/webapp/sample to view your installed Web application.

B.1.4 Questions related to the installation process

1. Installing Web applications in the universal Web application server is easy
   STRONGLY-AGREE
   AGREE
   NEUTRAL
   DISAGREE
   STRONGLY-DISAGREE

2. Overll, I am satisfied with the ease of installing Web applications in the universal Web application server
   STRONGLY-AGREE
   AGREE
   NEUTRAL
   DISAGREE
   STRONGLY-DISAGREE

3. Overll, I am satisfied with the amount of time it took to complete the installation of the Web application in the universal Web application server
   STRONGLY-AGREE
   AGREE
   NEUTRAL
   DISAGREE
   STRONGLY-DISAGREE

4. Installing Web applications in the universal Web application server would make it easier to install Web applications **STRONGLY-AGREE**
   AGREE
   NEUTRAL
   DISAGREE
   STRONGLY-DISAGREE

5. Installing Web applications in the universal Web application server is easier than installing Web applications in Tomcat

   STRONGLY-AGREE

   AGREE

   NEUTRAL

   DISAGREE

   STRONGLY-DISAGREE

6. List the most negative aspect(s) of the installation process in section A.

   .....

   .....

   .....

   .....

7. List the most positive aspect(s) of the installation process in section A.

   .....

   .....

   .....

   .....

8. Comment on the installation of Web applications in the universal Web application server as compared to installing Web applications in Tomcat.

   .....

   .....

   .....

   .....

9. List any suggestions you have in regard to the Web application installation process.

   .....

   .....

   .....

   .....

B.1.5 Questions related to the packaging process

1. Packaging Web applications for deployment in the universal Web application server is easy.
   STRONGLY-AGREE
   AGREE
   NEUTRAL
   DISAGREE
   STRONGLY-DISAGREE

2. Overall, I am satisfied with the ease of packaging Web applications in the universal Web application server
   STRONGLY-AGREE
   AGREE
   NEUTRAL
   DISAGREE
   STRONGLY-DISAGREE

3. Overall, I am satisfied with the amount of time it took to complete the packaging of the Web application
   STRONGLY-AGREE
   AGREE
   NEUTRAL
   DISAGREE
   STRONGLY-DISAGREE

4. List the most negative aspect(s) of the packaging process
   .....
   .....
   .....
   .....

5. List the most positive aspect(s) of the packaging process

   .....

   .....

   .....

   .....

6. List any suggestions you have in regard to the Web application packaging process

   .....

   .....

   .....

   .....

B.2 Responses to the free style questions of the usability questionnaire

Similar responses to the questions are not repeated.

1. **Question**. List the most negative aspect(s) of the installation process in section A.
   (a) None
   (b) an inexperienced person could be confused with the directory paths
   (c) Too short to comment on
   (d) Did not find it very user friendly
   (e) it requires the knowledge of the Linux terminal
2. **Question**. List the most positive aspect(s) of the installation process in section A.
   (a) It is quick
   (b) Straightforward
   (c) Anyone is capable of performing it
   (d) very simple, works like most Web containers I have used
   (e) it appear to be more abstract/general
3. **Question**.Comment on the installation of Web applications in the universal Web application server as compared to installing Web applications in Tomcat.
   (a) Installing in Tomcat takes a significantly longer time than installing in the universal Web application server
   (b) very similar in concept. Perhaps a little more complicated but the functionality makes the effort worth it
   (c) In terms of simplicity of installation, I think ease of use of Tomcat and that of the universal Web application server is relatively the same
   (d) Tomcat is a bit more explicit, hence making it much easier to follow

(e) Takes less time and easy to learn

(f) Tomcat looks more feasible because it even has a gui thing going on and you just type the destination of the file and press a button and it is installed whereas with the other case average users will suffer

4. **Question**. List any suggestion you have in regard to the installation process.

   (a) None

   (b) animated help!

5. **Question**. List the most negative aspect(s) of the packaging process.

   (a) Copying the sample weblet.xml into a directory using the 'cp' command was long and you could easily make errors

   (b) the update of a file in the package seems to require a new package to be created

   (c) takes quite some time

   (d) requires a lot of commands to complete

   (e) it requires the zip application to be previously installed

   (f) Creating directories and copying classes with commandline can easily cause typo errors

   (g) one has to have the background of the application

6. **Question**. List the most positive aspect(s) of the packaging process.

   (a) Tasks carried out quickly

   (b) Ease to use

   (c) works like most containers

   (d) well packaged for first time users

   (e) unambiguous

   (f) simple

(g) Did not really find anything simple positive

(h) its easy to understand

(i) saves a lot of time anyone can do it

(j) after it is packaged can copy into public_html domain directly

7. **Question**. List any suggestion you have in regard to the Web application packaging process.

   (a) shorten the commands

   (b) nothing I can think of

   (c) the packaging process should be handled by a graphical manager

   (d) Maybe ideal, but if there could be some way of automating the process of generating the application descriptor such as the weblet.xml file

   (e) The process could be shortened if the zipping occurred in the position where it should be copied

   (f) Keep the contents of the two directories together

   (g) Make the process shorter

   (h) script as well as GUI interface would be good and easier