X-Switch: An Efficient, Multi-User, Multi-Language Web Application Server

Mayumbo Nyirenda, Hussein Suleman, Andrew Maunder, Reinhardt van Rooyen

Department of Computer Science, University of Cape Town

ABSTRACT

Web applications are usually installed on and accessed through a Web server. For security reasons, these Web servers generally provide very few privileges to Web applications, defaulting to executing them in the realm of a guest account. In addition, performance often is a problem as Web applications may need to be reinitialised with each access. Various solutions have been designed to address these security and performance issues, mostly independently of one another, but most have been language or system-specific. The X-Switch system is proposed as an alternative Web application execution environment, with more secure user-based resource management, persistent application interpreters and support for arbitrary languages/interpreters. Thus it provides a general-purpose environment for developing and deploying Web applications.

The X-Switch system's experimental results demonstrated that it can achieve a high level of performance. Furthermore it was shown that X-Switch can provide functionality matching that of existing Web application servers but with the added benefit of multi-user support. Finally the X-Switch system showed that it is feasible to completely separate the deployment platform from the application code, thus ensuring that the developer does not need to modify his/her code to make it compatible with the deployment platform.

KEYWORDS: Web application servers, scalability, context switching, process persistence, modularity

1 INTRODUCTION

Web applications that once were fairly monolithic are slowly making the transition to collections of cooperating services. This trend is being spurred on by the steady increase in availability of standard Web Services interfaces to many popular services. As a step further, Web Services also may be aggregated to create new, possibly more useful, services. In this environment every service interface is mapped onto a component but some larger components (e.g., learning management systems) can provide multiple Web-based interfaces to downstream services (e.g., university portals). In the limiting case, each component provides at least one externally-accessible service (e.g., an API to a search engine component). Now, given that each component has well-defined external interfaces that operate over the Web, there is no longer any requirement for standardisation in the choice of programming language or Web technology. As such, different components could be developed in different languages. Unfortunately most current Web servers do not provide a mechanism to support Web components/applications in multiple languages easily. For example, if one component of a larger system is coded in Java, another in PHP and a third in Perl, it is non-trivial (if at all possible) to have a single Web server software system cater for all languages and, additionally, keep all interpreters and virtual machines in memory for faster execution.

To further complicate matters, there is frequently no simple correspondence between Web applications and physical machines. Given the complexity of managing a Unix server, it is often the case that a more powerful machine is shared among many users, to amortise the cost of server management. In this case, many users may be using a single machine for Web application/component development and deployment. These Web applications are usually applications belonging to a particular user account and they may need to read from or write to files on the disk. Since the Web applications are executed by the Web server (rather than by the user), the Web server would need access to write to every user's home directory but, for security reasons, the Web server instance is run as an 'unprivileged' system user, the user 'nobody'. Thus, to allow access to the users' files, Web application directories and files need to be world-readable or world-writable as necessary. This is clearly undesirable as the Web application of one user may write to the home directory of another user and intentionally or inadvertently overwrite or delete data or applications. This solution is untenable in many situations, including teaching and learning environments in Web Programming. Ideally, no Web application should need to be world-writable.

Email: Mayumbo Nyirenda `mnyirenda@unza.zm`, Hussein Suleman `hussein@cs.uct.ac.za`, Andrew Maunder `amaunder@cs.uct.ac.za`, Reinhardt van Rooyen `rvanrooy@cs.uct.ac.za`

Solutions currently exist that allow a Web server to access files or directories that are not world readable or world writable. They achieve this by switching user context from 'nobody' to the user context of the owner of the Web component. Examples include suExec [1] and suPHP [2], where the 'su' prefix denotes 'switch user'. While these solutions are attractive they typically only support a single Web technology (implementation language or server) and cannot provide the performance required by industrial Web applications. High levels of performance have been achieved by Web applications implemented using technologies like FastCGI [3] and SpeedyCGI [4] but SpeedyCGI in particular only supports a single implementation language, namely Perl.

This paper thus presents the X-Switch system as an alternative - a framework that allows multiple users the option of deploying Web components written in one of a set of languages on a single Web server, while maintaining a high level of security and scalability - essentially a 'universal' Web application server.

2 BACKGROUND

The development of modern Web servers has always been driven by the requirements of its primary users. Initially these users included military scientists and engineers as well as university scholars and academics. The early requirements for a Web server hinged largely around the displaying of simple static content, but as the demand for commercial Web applications increased so did the requirements for higher Web server performance, scalability and dynamic content generation.

One of the first standards available that provided a mechanism for server side applications to service Web requests was the Common Gateway Interface (CGI) [5]. CGI-driven applications initialise a new application instance for each request received by the server. The overhead of repetitive process initialisation severely hampered the performance of CGI so the World Wide Web Consortium (W3C) proposed that a more efficient technique be developed to overcome this shortcoming.

Another shortcoming of the CGI based Web server was the lack of Web component isolation when used in a multi-user environment. Web servers using a standard implementation of CGI are typically unprivileged for security reasons and are only allowed to access world-readable Web components. This is most certainly undesirable for the client's sake as their components will be placed in a world-readable directory (typically 'cgi-bin'), accessible by all the system users and open to unauthorised modification or even removal. Such a scenario is a major security risk. As an example, commercial Web application servers such as Jakarta Tomcat [6] were not designed to support secure, multi-user environments and the Web components deployed on them can be exploited in exactly the manner described above. Furthermore, if a Web application component, deployed by a user, does not validate its request data the component may possibly lead to a hacker taking control of a component process. This is commonly referred to as a buffer overflow attack. The Open Web Application Security Project [7] has listed a buffer overflow attack as one of the ten most common Web application security problems.

The Apache Software Foundation produced suExec [1] in a response to the security risks posed above. Web components can be accessed via an Apache Web server with an identical set of privileges as the owner of a Web component. Essentially, a suExec-enabled Web server has the ability to switch its user context from 'nobody' (an unprivileged user) to the user context of the component, e.g., 'Andrew'. The Web server process running as Andrew can access any files or directories owned by the user Andrew and is prohibited from accessing any other user's files or directories. Therefore even if a security breach, such as a buffer overflow attack, did occur, the rogue process would only be able to access a single user's files and directories, effectively protecting other system users as well as the rest of the system files and resources.

CGIWrap [8] was developed to provide similar functionality to suExec but aimed to provide context switching abilities while being Web server independent. SBox [9] is another script isolation technique that executes CGI scripts (target scripts) on behalf of the Web server process. Like suExec and CGIWrap, the wrapper script is set up to be SUID ROOT, which makes it possible for it to change its process ID to match that of the target script, the context switch. In addition to being able to perform a context switch, the SBox wrapper script performs a series of checks on the target script and prepares the script execution environment. The pre-execution checks include ensuring that the script is non-world-writable and that the script is run as the user and not anybody else. In addition, Stein [9] felt it necessary to include component isolation when preparing the component execution environment. This includes performing a 'chroot' to the directory that contains the user's scripts and thereby effectively sealing the script off from the rest of the server and limiting the memory, CPU time and disk space available to the target script before the script is finally invoked, similar to a sandboxing technique used by Sun Microsystems.

Sun Microsystems [10] introduced a sandboxing technique as an integral part of their Java Applet security framework. The Java SDK 1.0 used the sandbox metaphor to explain the principle behind the security features of the Java Virtual Machine (JVM). An untrusted Servlet is loaded into the JVM dynamically and executed within a very restricted environment. The restrictions apply to memory, file I/O privileges and priority of the Servlet's execution thread. An important point to note is that a single Servlet engine (container) typically services many Servlets belonging to various users. Untrusted Servlets must be accessible by a container that typically runs as 'nobody' or a similarly unprivileged user. This implies that all Servlets must be globally accessible to allow the con-

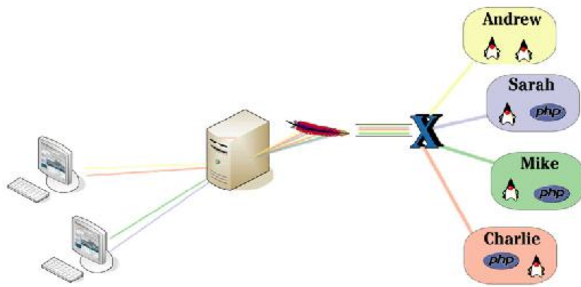


Figure 1: The X-Switch universal Web-application server system

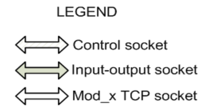
tainer access to them. An untrusted Servlet is therefore severely restricted, firstly by the operating system and secondly by the JVM, resulting in a Servlet that is completely exposed with little or no file access rights. Open Market Inc. developed FastCGI [3], a persistent implementation of CGI, which provided a mechanism for reusing existing application instances to service future requests. FastCGI maintains all the existing benefits of CGI, such as process isolation and language and architecture independence, while minimising the delay between request arrival and request process initialisation. However, a Web component must use FastCGI libraries in order to be compatible with the FastCGI framework and take advantage of its component use and reuse. The result is that a Web component that was previously accessible via CGI would have to be modified, albeit only slightly, for it to run persistently on a FastCGI enabled Web server. Consequently FastCGI architecture based Web components cannot trivially be run by other Web application servers. The X-Switch project aims to solve a different fundamental problem from FastCGI, which focuses on performance, but the system created is similar. It may be argued then that X-Switch validates the FastCGI approach while attempting to support different Web application types natively.

3 DESIGN AND METHODOLOGY

The X-Switch system(see Figure 1) is designed to combine the principles of process persistence and context switching into a single solution while maintaining the benefits of CGI (process isolation, language and architecture independence) and process isolation via a sandboxing technique. The X-Switch system also introduces the concept of separate support for multiple languages and development frameworks. Thus the X-Switch system was designed to meet three primary goals: efficiency, multiple user support and multiple technology support, without modification to the Web component code to ensure compatibility.

In brief, the major design objectives included the following aspects:

- Modular design
- Support for multiple users
- Independence of different backend technologies
- Scalability
- Efficiency



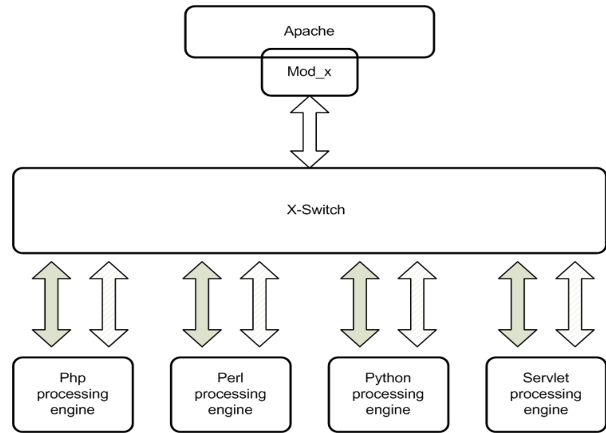


Figure 2: Performance of the universal Web application server with an increasing number of concurrent connections

- Security

The system is based on three sub tiers: the Web server module that connects the Web server to X-Switch; the X-Switch module that manages and processes requests; and a set of Web application processing engines for different languages and frameworks. These are illustrated in Figure 2.

3.1 Web Server Module

The first sub-tier is the Web server module. This tier is responsible for routing requests together with any additional information from the Web Server to the core X-Switch system(See Figure 3). The X-Switch system takes a modular approach in order to make it less dependent on the Web server implementation approach. Therefore the Web server module implements less functionality and can be implemented using any Web server that has an external plugin API. The Apache Web server was used to implement mod_x because of it's wide spread use and popularity [11]. The X-Switch Apache module (mod_x) routes requests from the Apache Web server to the core X-Switch system.

3.1.1 Web server module/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 the Web server module and the X-Switch main module.

The X-Switch main module receives the

1. request method type and content length
2. filename and request arguments
3. request headers
4. default engine type required to process the request

from mod_x using this protocol. After the request has been processed the X-Switch main module then sends the

1. response headers
2. response body

to mod_x using the Web server module/X-Switch communication protocol.

3.1.2 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 also defines an *ENGINE_TYPE* custom configuration directive that determines the default engine type for that particular directory. Using the *ENGINE_TYPE* makes it possible to register one handler 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 *ENGINE_TYPE* 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 [12].
After being invoked, the mod_x handler routine first establishes a TCP socket connection with the X-Switch main module. If successful mod_x then proceeds to process the request otherwise it sends a 'service temporary not-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 the X-Switch main module using the communication protocol presented in section 3.1.1. mod_x receives the response from the X-Switch main module on the already established TCP socket and thereafter closes the connection.

3.2 X-Switch

The second sub-tier is the core X-Switch system. This tier is responsible for managing the requests arriving from the Web server module and routing responses from the processing engines to the Web server(See Figure 3). This tier reads the core X-Switch system configuration file for supported processing engines and the information needed to run the engines. It is also responsible for creating, monitoring and destroying heterogeneous processing engines for each user, based on available system resources and the current traffic load.

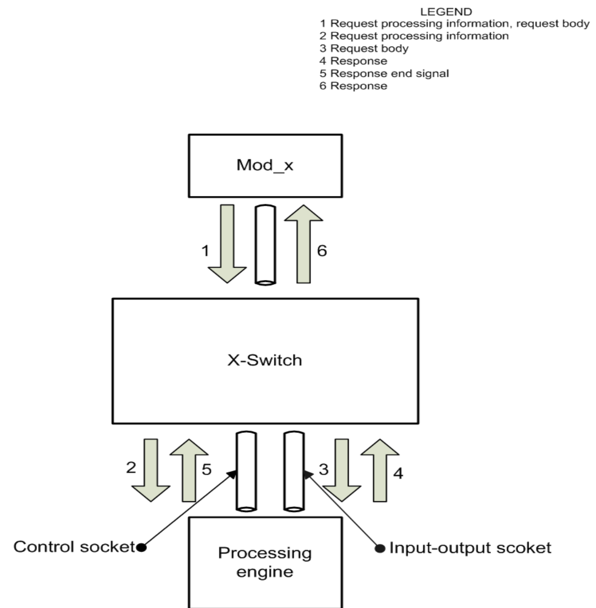


Figure 3: X-Switch request routing

3.2.1 X-Switch main module/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 the particular request, the X-Switch main module establishes communication with a processing engine using two socket pairs. One is for data sending while the other is for controlling and monitoring the request processing phase. The X-Switch main module 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 present. The request body is sent using the X-Switch main module's data input-output socket while the engine reads it in using its '*STDIN*' input stream. After request processing has started the X-Switch main module 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. The X-Switch main module then relays this response to the Web server module using that request's connection socket with the Web server module. The processing engine then signals the end of the response by sending a response end control character via the control socket to the X-Switch main module.
The X-Switch system uses a polling mechanism to support multiple connections. Studies by Pariag et al [13] revealed that an event-driven or hybrid server achieved up to 18% higher throughput than the best implementation of the thread based server. Therefore the X-Switch system uses a single thread to listen to and poll all its connections.

3.2.2 X-Switch system wrapper script (suexecme)

X-Switch uses a lightweight application, suexecme, to perform the context switching of the engines. Suexecme runs as a root process and thus has its 'suid' bit set by the root user to allow any person to execute the application. Suexecme is run each time an engine is being created and sets the engine to belong to the user whose script is being executed. Thus all scripts run using a processing engine are run with the privileges of the owner of the processing engine.

3.2.3 Request processing

A central requirement of the X-Switch system is the ability to support multiple users on a single Web server implementation. To achieve this X-Switch evaluates an incoming request to determine the user and checks if the user owns any existing engines that are available to process the request for that particular type/language of component. If an appropriate engine is identified the request is forwarded to the existing engine, otherwise the X-Switch main module spawns off a child process via suexecme.

The X-Switch system is independent of the back-end processing technology. Any processing language that can read from standard input and write to standard output can be used as a processing engine. Such an engine should be added to the X-Switch configuration file together with the path of execution and X-Switch restarted for the processing engine to be included in the list of processing engines supported by X-Switch. There are only two requirements that a processing engine has to adhere to: the processing engine must remain persistent and it must use the defined X-Switch/Processing engine communication protocol to correctly interact with the X-Switch module. This leaves great scope for the processing engine to be as complex as it needs to be, without adding much overhead when creating a processing engine.

A successful Web server system should be able to handle a heavy request load and be able to allocate resources to users who require them, while still allowing all users to gain access to a processing engine. As more requests enter the X-Switch system, more processing engines are created on a per-user basis. This ensures that the X-Switch system provides sufficient processing power for the users who have a greater request load. Should the system resources become scarce, the X-Switch system will destroy unused engine processes in order to provide users with pending requests an available engine for processing. This mechanism ensures that the X-Switch system remains efficient and degrades gracefully even under extreme volumes of requests.

Security and process isolation was an integral part of the X-Switch system design. The lack of per-user file and process isolation in conventional Web server systems formed one of the primary motivating factors for the development of the X-Switch system. By isolating each process engine on a per-user basis, each process is thereby granted a user equivalent set of permissions, thus creating a separate and secure user environment. Finally the X-Switch modular architecture utilises TCP/IP socket communication for IPC (interprocess communication). This means that the X-Switch system can easily be extended to run on a distributed system where the Web server, X-Switch module and the user environments are all running on separate machines as part of a LAN.

3.3 Web Application Processing Engines

The third sub-tier is the processing engines. The key feature of such engines is that they are persistent implementations, thus avoiding the overhead associated with repeated process initialisation. The processing engines are responsible for processing requests for particular users(See Figure 3). The X-Switch main module (tier 2) initialises separate processing engines for each user and for each type of back-end technology as they are required.

All issues regarding process persistence are handled by the processing engines and not the Web components, thus any existing Web components can be used unlike existing solutions which require that the code be compatible with the technology. The Web component code should be written using the standard libraries and APIs that would be used in a regular Web server environment. Ultimately Web developers are not required to undergo any additional training in order to utilise X-Switch.

Processing engines read the request body using the standard input and write the response to a request to standard output (see Figure 3). Request processing information and the request processing phase are controlled using the communication control socket. Perl, servlet (Java), Python and Php engines have been implemented.

The Java servlet engine provides only part of the functionality provided by industrial Java Web application servers. X-Switch starts a processing engine on arrival of the first request. Unlike most industrial solutions the X-Switch servlet engine does not preload the deployed servlets and thus allows for run time deployment and management of the servlets.

The Php engine provides an environment for running Php scripts in X-Switch persistently by providing a wrapper to the command line interface version of Php. The wrapper sets up an execution environment and variables needed for the scripts to execute successfully. The scripts write the output to the standard output.

The Perl and Python processing engines are simple and lightweight implementations of persistent interpreters that read input from the communication control socket. The parsed information is then used to retrieve the filename that is used to execute the appropriate script. The script is then run using this processing engine. The processing engine sends a signal to X-Switch to signify the end of the response

after the script finishes generating the response.

4 EXPERIMENTS AND RESULTS

For the X-Switch system to be accepted as a feasible solution it would not only have to meet the requirements outlined earlier in this paper but must do so efficiently. The experimental section of this paper will examine the efficiency of the X-Switch system in servicing requests for simple Web components written in PHP, Perl, Python and Java. The results will then be compared with the results obtained from existing solutions. Further, the performance implications of the modular architecture will be investigated.

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 [14] and Jakarta-jmeter-2.2 [15].

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 startup. A constant throughput timer controls the amount of time between requests issued by the thread. JMeter also has a 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 stress-testing the universal Web application server. The experiments focus on the performance of the universal Web application server under varying conditions.

4.1 Number of Concurrent users

4.1.1 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.

4.1.2 Methodology

The experiment was carried out by conducting a series of trials 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.

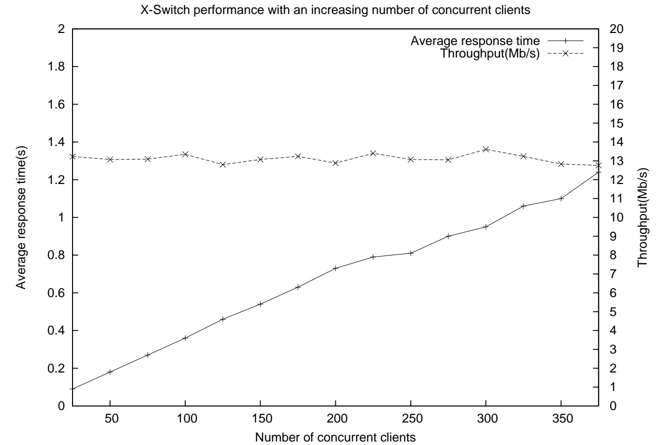


Figure 4: Performance of the universal Web application server with an increasing number of concurrent connections

4.1.3 Results

The results of this experiment are show in Figure 4

4.1.4 Discussion

The server throughput was more or less constant, which means that 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 as expected - a linear degradation in the response time was observed and is arguably ideal.

4.2 Impact of processing layers on Response time

4.2.1 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 layers contributes to the total response time.

4.2.2 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

Table 1: The percentage of time that each processing layer contributes to the total processing time

Processing layer	Time (%)	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

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 recorded the total time it took to service the requests 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. It recorded an approximation of the time spent in the Apache and Mod_x processing layers.
3. In the third run of the experiment, a simple script was written that replaced the Apache Web server and Mod_x and directly issued requests to X-Switch. It recorded the time taken to process the requests through X-Switch and its lower layers.
4. In the fourth run, a script that spawned a processing engine and issued requests directly to the engine was written and used. It recorded the time taken to process the requests through the engine processing layer and its lower layers.
5. Lastly, the time taken to run the script was measured by running a 'Hello World' perl script 100,000 times in a persistent interpreter and recording the time taken.

The recorded times were then used to compute the time taken to process requests through each of the request processing layers.

4.2.3 Results

The results of this experiment are tabulated in Table 1

4.2.4 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 substantially 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 because 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.

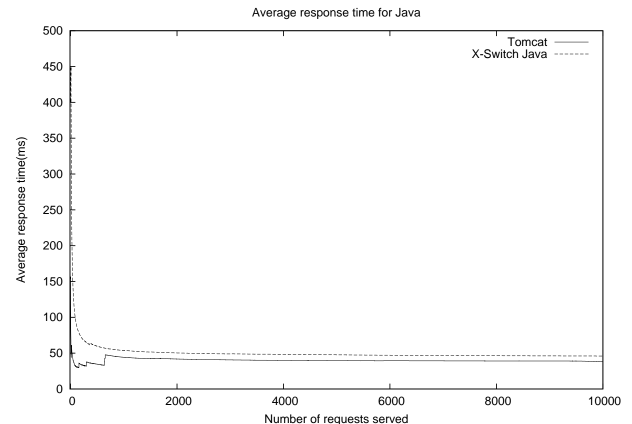


Figure 5: Average response time for Apache Tomcat and the Java processing engine

4.3 Average response time

4.3.1 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 ensure repeatability and and a controlled environment.

4.3.2 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, mod_php, mod_python, FastCGI and SpeedyCGI.
Metric: Average response time

4.3.3 Results

The results are graphed in groups of programming languages. Figures 5,6,7 and 8 are graphs of the results from the experiment.

4.3.4 Discussion

The response time of the universal Web application server's Java engine averages to a slightly higher value than that of Apache Tomcat (See Table 2). The Java processing engine had an initial startup time of

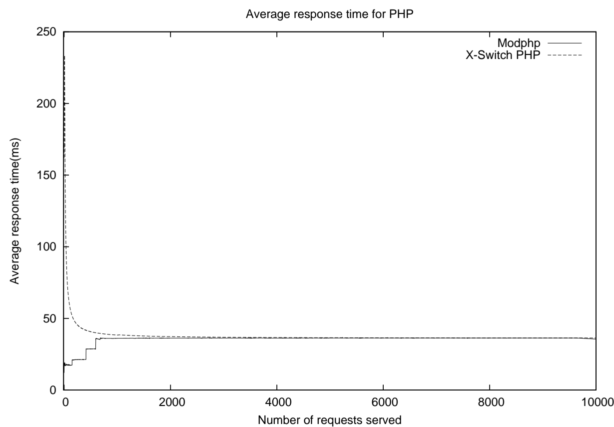


Figure 6: Average response time for mod_php and the PHP processing engine

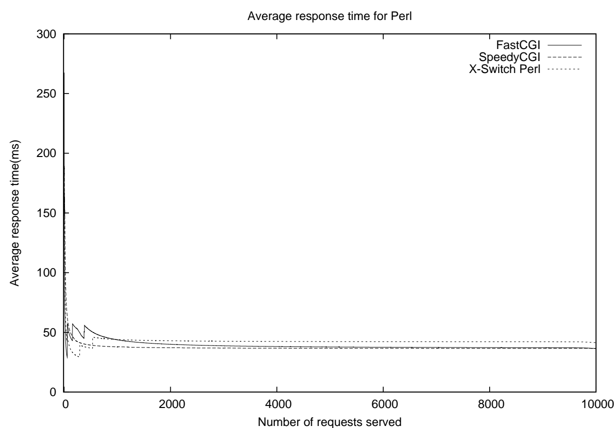


Figure 7: Average response time for the Perl processing engine, SpeedyCGI and FastCGI

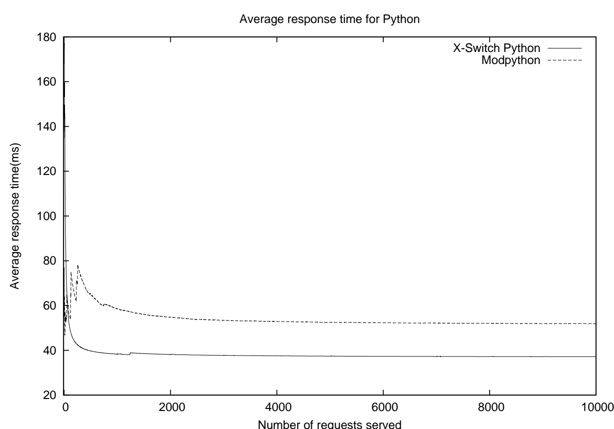


Figure 8: Average response for the python processing engine and mod_python

Table 2: The initial (IRT) and average (ART) response times of X-Switch and other Web application technologies

Technology	IRT (ms)	ART (ms)
X-Switch Java	450	54
Apache Tomcat	180	48
X-Switch PHP	234	36
Apache mod_php	20	35
X-Switch Perl	243	39
FastCGI Perl	238	43
Speedy Perl	236	38
X-Switch Python	180	42
Apache mod_python	88	52

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 mod_python. The Python processing engine had an initial response time of about 180ms and a final average response time of 42ms whereas mod_python 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 mod_php 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 are 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 CONCLUSIONS

The X-Switch system has confirmed that it is indeed feasible to create a multi-user and multi-language Web server extension mechanism without sacrificing performance or the security framework that is an implicit feature of less capable systems. Techniques such as process persistence (reuse) and component caching enhanced the overall performance of the X-Switch system. The servlet engine test results showed that in the simplest case ('Hello World' Web component) the X-Switch system produced performance results that were comparable with commercial grade Java Web application servers. In addition, the X-Switch system maintained multi-user support as well as run-time deployment - these features were not supported by any other Java Web application servers.

The modular design of the X-Switch system allows for

the Web server module to be replaced by an updated or alternative solution and similarly for processing engines, provided that the updates adhere to the X-Switch interface definitions. This design feature makes it feasible to allow third party developers to implement and maintain X-Switch engines. The inclusion of TCP/IP pipes as part of the X-Switch communications protocol makes it possible for the system to be hosted on a distributed system where the Web server module, processing engines and the X-Switch module may be located on separate machines, thus providing an effective mechanism to achieve system scalability. With more testing and further implementations of additional processing engines, the X-Switch system can possibly fill a niche left out by conventional commercial Web Servers. While other systems have provided subsets of these features, X-Switch attempts to unify Web server requirements into a single universal system and provide further evidence that this is indeed possible.

6 FUTURE WORK

Since the aim of this project was largely to develop a proof-of-concept prototype, there are various optimisations that can be incorporated into the code base. At a micro-level, the individual processing engines could use a common pool of shared libraries so that memory efficiency is not sacrificed for processing time efficiency. At a macro-level, the existing process pool can be interrogated to optimise the management of processing engines e.g., the system can maintain a dynamic profile of combined historical and past use to prime engines to match an expected request pattern. These are both examples of internal improvements - the impact of the system is greater when it is considered for its relationship to other projects.

In an almost trivial example, X-Switch can be used as part of a Web server installation to teach students how to develop Web applications. The architecture of X-Switch is fundamentally one where multiple users can share a single Web server such that each user has a privileged, i.e., with full access to that user's resources, and distinct sandbox. This is ideal where students need to be experimental but do not have access to dedicated computing, a scenario especially suited to large classes of undergraduate students and students in developing countries, where it cannot be assumed that every student has a computer at home! This use of X-Switch will further vindicate its design philosophy as well as bring to the fore possible extensions such as individual user resource allocation and administrative control systems to monitor large installations.

Primarily, however, the X-Switch system was designed to serve as a platform for future generations of adaptive Web applications and Web services/Services. New generations of digital library systems (aka Web-based information management systems) have to deal with both flexibility of systems and the need for arbitrary scalability - users of such Web-based systems have been known to ask for additional pluggable services post-deployment and data sets can easily range from 5 items to 5 million items. Hence the need for a flexible Web-based deployment container was identified and X-Switch was designed. There is still much work to be done on how generic Web application components can be deployed on demand, replicated and migrated in both cluster and grid computing configurations. X-Switch can provide the language-agnostic platform as one starting point for this research, but additional work is required to incorporate support for service deployment mechanisms, security models for controlled access to individual suites of components, labelling and management of service endpoints, component configuration and local resource allocation in distributed environments. The anticipated end-result is a system that allows a non-privileged user community to easily install and make accessible software components that are in essence Web Services, without having to deal with a myriad of different technologies and without having to hardwire hooks into the Web server and similar system-level resources, while gaining flexibility, security and scalability.

X-Switch can serve as a useful common base environment into which Web applications or components can be installed and executed without complex user-specific configuration. This could make it simpler to install Web applications and Web application components in general. Current open source packaging systems (such as Portage) make it feasible to define X-Switch and mod_x as dependencies of a processing engine, which is in turn a dependency of a Web component that executes in that environment. Thus, the installation and use of X-Switch may be completely transparent to an end-user.

Ongoing work on the X-Switch system is focussed on attempts to define the structure of a Web application or component, independently of language or environment, as a redeployable package. This package would then be supported directly by X-Switch as a language-independent application format, with drop-in deployment, supporting not only rapid installation for single machine systems but simple relocation and replication in a high performance multi-user compute cluster.

REFERENCES

[1] APACHE SOFTWARE FOUNDATION. *suEXEC Support*, 2005. URL `http://httpd.apache.org/docs/2.0/suexec.html`.

[2] S. MARSCHING. *suPHP*, 2005. URL `http://www.suphp.org`.

[3] OPEN MARKET INC. *FastCGI: A high performance Web server interface*, 1996. URL `http://www.fastcgi.com`.

[4] S. HORROCKS. *Speedy CGI/Persistent Perl*, 2001. URL `http://daemoninc/speedyCGI`.

[5] WORLD WIDE WEB CONSORTIUM. *The common gateway interface*, 1999. URL `http://www.w3c.org/cgi/`.

[6] V. Chopra et al. *Apache Tomcat Security Handbook*. WROX Press Ltd, Hoboken, NJ, 2003.

[7] OPEN WEB APPLICATION SECURITY PROJECT. *OWASP Top Ten Most Critical Web Application Security Vulnerabilities*, 2004. URL `http://www.owasp.org`.

[8] N. NEULINGER. "CGIWrap: User CGI access", 2001. URL `http://cgiwrap.unixtools.org`.

[9] L. D. Stein. "Putting CGI scripts in a box", 2003. Technical Report, Cold Spring Harbour Laboratory.

[10] SUN MICROSYSTEMS. *The Java Security architecture*, 1999. URL `http://www.java.sun.com/j2se/1.3/docs/guide/security.html`.

[11] P. Mutton. "November 2007 Web server survey [online]". Available: http://news.netcraft.com/. [December 2007].

[12] "Apache Web server 1.3 API Dictionary [online]". Available: http://httpd.apache.org/dev/apidoc/. [December 2007].

[13] A. H. P. B. A. S. R. D. D. Pariag, T. Brecht. "Comparing the performance of Web Server Architectures". In *Proceedings of EuroSys'07*. Lisboa, Portugal, 2007.

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

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