Designing a ‘universal’ Web application server.

ANDREW MAUNDER, REINHARDT VAN ROOYEN AND HUSSEIN SULEMAN
University of Cape Town

Modern Web server systems typically consist of a single Web server instance capable of utilising various backend technologies. For security reasons this Web server instance is run as the unprivileged user, the user ‘nobody’. This has the implication of having users make their Web components world-accessible so that such an unprivileged Web server instance may access them. World accessible files or directories are open to many threats including modification and removal by any system user, authorised or unauthorised. The X-Switch system attempts to provide a solution to this problem by allowing Web components to be run with an identical set of privileges as the component owner, an essential feature for maintaining secure multi-user server environments. The X-Switch system is a generalisation of existing solutions but attempts to provide a higher level of performance and scalability while maintaining the benefits of being independent of the implementation language used.

The X-Switch system’s experimental results demonstrated that a Web server that utilises run-time context switching can achieve a high level of performance. Furthermore it was shown that an X-Switch compatible engine can be developed to 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 issues of performance from the Web component code thus ensuring that the developer is free from the task of modifying his/her code to make it compatible with the deployment platform.

Categories and Subject Descriptors: H5.2 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia - *Architectures*; D4 [**Operating Systems**]: Security and Protection - *Access controls;* D.4 [**Operating Systems**]: File Systems Management - *Access methods*
General Terms: Design, Experimentation, Performance, Security
Additional Key Words and Phrases: Web application servers, scalability, context switching, process persistence, modularity

1. INTRODUCTION

Modern Web server systems typically consist of a single Web server instance capable of utilising various backend technologies (e.g., PHP, Java, Perl). For security reasons, the Web server instance is run as an ‘unprivileged’ system user, the user ‘nobody’. Any directories or files that are to be accessible by the Web server instance must be set as world readable and similarly any directories or files that need to be written must be all be set as world writable. For the most part an unprivileged Web server instance can perform the necessary reading and writing of files in a world readable and writable directory, typically the ‘public_html’ directory, without any problems. But what if a situation arises where allowing all Web components to reside in a single, world accessible directory is not desirable, such as on a multi-user Web server where Web components are owned by various users. Any system user may access the Web components and the unauthorized modification or removal of a component is a constant threat, whether deliberate or accidental.

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’[Apache Software Foundation, 2005] and ‘suPHP’ [Marsching, 2005], 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 such as FastCGI [Open Market Inc., 1996] and SpeedyCGI [Horrocks, 2001] but do not incorporate any context switching component and, in the case of SpeedyCGI, only support a single implementation language, namely Perl. This paper presents the X-Switch system, a framework that allows multiple users the option of deploying Web components written in any language 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

Author Addresses:
A. Maunder, Department of Computer Science, University of Cape Town, Private Bag Rondebosch, 7701, South Africa; amaunder@cs.uct.ac.za.
R van Rooyen, Department of Computer Science, University of Cape Town, Private Bag Rondebosch, 7701, South Africa; rvanrooy@cs.uct.ac.za.
H. Suleman, Department of Computer Science, University of Cape Town, Private Bag Rondebosch, 7701, South Africa; hussein@cs.uct.ac.za.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, that the copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than SAICSIT or the ACM must be honoured. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2005 SAICSIT

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) [World Wide Web Consortium, 1999]. CGI driven applications would typically 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.

Open Market Inc. responded with the development of FastCGI [Open Market Inc., 1996], a persistent implementation of CGI, which provided a mechanism for reusing existing application instances to service future requests. FastCGI managed to maintain 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. Unfortunately the FastCGI solution includes a hidden drawback, which is that a Web component must use FastCGI libraries in order to be compatible with the FastCGI framework. The result is that a Web component that was previously accessible via CGI would have to be modified for it to be compatible with a FastCGI enabled Web server.

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 most definitely a major security risk. As an example, commercial Web application servers such as Jakarta Tomcat [Chopra and Turner, 2003; Sun Microsystems, 1999] were not designed to support secure, multi-user environments and the Web components deployed on them can be exploited in exactly the manner mentioned above. Furthermore, if a Web application component, deployed by a user, does not validate its request data the component may crash, possibly leading 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 [OWASP, 2004] has listed a buffer overflow attack as one of the ten most common Web application security problems.

The Apache Software Foundation produced 'suExec' [Apache Software Foundation, 2004] 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' [Neulinger, 2001] was later developed to provide similar functionality to 'suExec' but aimed to provide context switching abilities while being Web server independent.

Sun Microsystems [Sun Microsystems, 1999] 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 container access to them. A typical 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.

Stein [1999] presented his implementation of a script isolation technique called 'SBox' [Stein, 1999], a CGI wrapper script that executes CGI scripts (target script) on behalf of the Web server process. The wrapper script is SUID to 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.

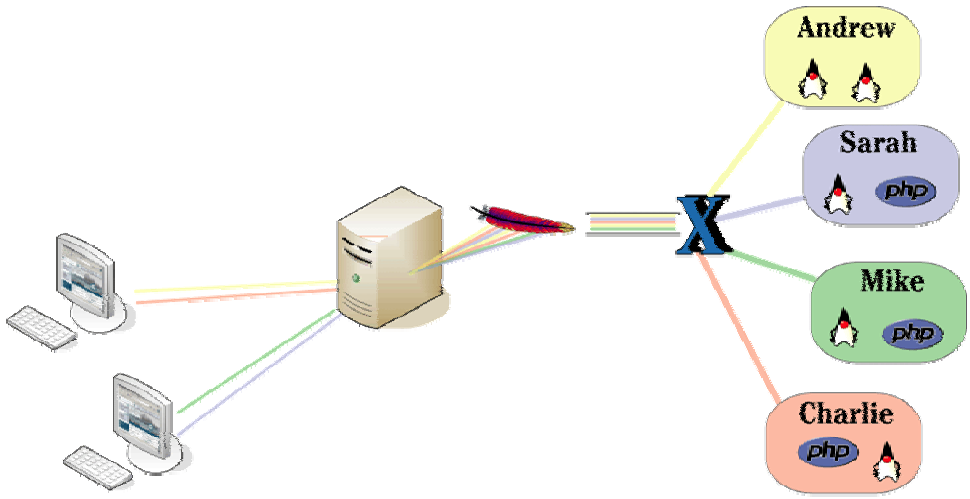

Figure. 1. *The X-Switch universal Web-application server system.*

In addition, Stein [1999] 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 Sun Microsystems' sandboxing technique.

The X-Switch system aims to combine these 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. In addition the X-Switch system aims to allow Web developers to be free from using special libraries to ensure compatibility with persistent Web server implementations. Finally the X-Switch system aims to support multiple back-end technologies (Java, PHP and Perl) and provide an interface for the inclusion of additional technologies in the future.

3. DESIGN AND METHODOLOGY

As mentioned, the X-Switch system aimed to address a particular flaw that was evident in current Web servers, that is the lack of a multiple user Web application server that can provide a high level of performance while maintaining simple Web component deployment. The X-Switch system extends the ideas proposed in solutions such as FastCGI [Stein, 1999] and suExec [Apache Software Foundation, 2004], such as process persistence and multi-user support, but introduces the concept of separating issues of performance from the Web component code as well as run-time component deployment. 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.

The system is based on a three sub-tier system:

- The first sub-tier consists of the Web server module. The X-Switch system is designed to be independent of the Web server used. The Apache Web Server was chosen due to its popularity as a Web server and for the ability to include custom Apache modules via the server API. The X-Switch Apache module (Mod_X) routes requests and responses between the Apache Web server and the core X-Switch system.

- The second sub-tier consists of the core X-Switch system. This tier is responsible for managing the requests and responses passed between the Web server and the processing 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.

- The third sub-tier consists of the processing engines. The key feature of such engines are 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. The X-Switch system (tier 2) initialises separate processing engines for each user and for each type of back-end technology as they are required (see Figure 2).

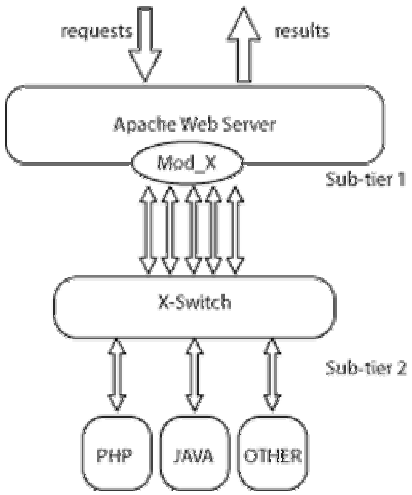

Figure. 2. *The X-Switch architecture*.

There were many considerations when designing the system. In particular, the system had to be:

- Modular in design
- Supportive of multiple users
- Independent of different backend technologies
- Scalable
- Efficient
- Secure

The modular approach stems from the initial requirement that the X-Switch system had to be able to handle multiple backend processing technologies and could be coupled with the system during run-time without recompilation. It was also required that the Web Server component be decoupled so as to provide system administrators with the freedom to include updated or alternative Web servers to serve as the X-Switch Web server front end. As mentioned in section 2, previous solutions such as FastCGI required users to alter existing code in order to make the code compatible. X-Switch users can use existing Web components that were written using standard libraries and APIs, thus shifting all concerns about engine persistence to the X-Switch module. Ultimately Web developers are not required to undergo any additional training in order to utilise the X-Switch performance- and usability-enhancing features.

An implicit requirement of the X-Switch system was the ability to support multiple users on a single Web server implementation. This was achieved by evaluating incoming requests to determine if the requested user owns any existing engines that are available to process a request for a particular type of component. If an appropriate engine is identified the request is forwarded to the existing engine via a TCP/IP request pipe - if not, the X-Switch module process spawns off a child process that has 'root' privileges. The child process then immediately switches its user ID to that of the owner of the requested component. The child process now possesses the privileges of the Web component owner and begins loading and executing the appropriate processing engine code. The minimal lifetime of a root process is an essential part of the X-Switch security policy.

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. There are only two requirements that a processing engine has to adhere to: the processing engine must remain persistent and it must use the proper X-Switch commands 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 overhead limitations 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, whilst still allowing all users to gain access to a processing engine. As more requests enter the X-Switch Web server, 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 servers 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) as well as message passing via well defined interfaces. 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.

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 (in this case at least as efficient as CGI which we regard as the minimum industry standard). The experimental section of this paper will examine the efficiency of the X-Switch system in servicing requests for simple 'Hello World' Web components written in PHP and Java. The results will then be compared with the results obtained from existing solutions, in particular CGI, mod_PHP and various commercial grade Java Web application servers. The feasibility of implementing a custom X-Switch engine that emulates an existing Web application server (e.g., Jakarta Tomcat) will also be evaluated.

4.1 Methodology

The experimental series compared the performance of the X-Switch system in conjunction with the X-Switch Java Servlet engine (essentially the X-Switch interface wrapped around a basic X-Switch Java Servlet container) and the X-Switch PHP engine (essentially the X-Switch interface wrapped around a command line PHP interpreter) with existing industrial solutions that do not provide context switching facilities.

As the aim of the experiment was not to compare the efficiency of the X-Switch backend engines with their industrial counterparts but to evaluate the performance implications of a run-time context switch, a simple 'Hello World' script was processed in an attempt to negate any performance advantages that the industrial solutions had over the X-Switch engines. This allowed us to effectively compare the total time taken for request delivery and the return of an appropriate response, excluding the internal processing time, thus isolating the overhead of a run-time context switch. Should the results fall within an acceptable range (see below), the X-Switch system's multi-user functionality would overshadow a minimal loss of performance, thus suggesting that the X-Switch 'universal' Web server is a feasible concept.

During experiments, a set of 5000 sequential requests were issued from a single client to the server, to study the pattern of response times for sequences of requests for the same resource. The requests were not sent in parallel because the effect of multitasking within/among different applications could be a confounding factor for the experiment. As mentioned above the 'Hello World' Java program and PHP script represented the simplest cases and were used to establish a benchmark value using CGI, where the benchmark value represents the total time (response time) taken from initialisation of a client request until the client received an appropriate or successful response. It was decided that the CGI benchmark value would represent the minimum acceptable value. A single, clean experimental machine was used to run the all the experiments, thus eliminating any system variability e.g., processor and hard-disk speed. The variability of traffic on a congested network was eliminated by running the client and server on the same machine, implying that the time lapse between the issuing of a client request and the arrival of the response is primarily due to server process instantiation.

4.2 Results

The data collected from the experiments are summarised in Tables 1 and 2 and on the graphs in Figure 3-5. Table 1 highlights the large variation between the initial response times of the evaluated Java Web application servers. In order correctly to interpret the data set it is important to understand the various components that constitute the initial response time value.

In the case of CGI the initial response time consisted primarily of process initialisation (for the Java back end interpreter), loading the program (script) from disk and processing the actual request (which is minimal for the simple 'Hello World' example). When considering the initial response times for the four commercial Java Web application servers the value implies something slightly different. The interpreter processes are initialised and the scripts are loaded during the Web server start-up phase and thus the primary component of the initial response time in this case would be the processing of the request itself.

The X-Switch system showed the longest initial response time of all the servers. This observation can be attributed to the fact that the X-Switch start-up phase does not include interpreter process initialisation or script loading from disk. Both these actions are performed at run-time when a request for a particular script (Web component) arrives, in some sense "just-in-time instantiation". X-Switch endures an initial response time of more than double that of some of the other servers, the price paid for avoiding component deployment during the server start-up phase.

This notion of run-time deployment is ideal for educational purposes where a student may wish to deploy several versions of his/her Web components within the space of a few hours and see the effects of modifications immediately.

The X-Switch system successfully implements run-time deployment while maintaining a high level of performance through a combination of process persistence and minimising disk access (see Figure 3 and 4). Attempting to minimise disk access and provide run-time deployment may seem contradictory but the X-Switch Servlet engine included a mechanism to evaluate an object situated on disk for changes before going through the expense of loading it.

Engine	Initial Response
CGI	118ms
Tomcat	86ms
Jetty	77ms
Resin	54ms
Orion	67ms
X-Switch	**143ms**

Table. 1. *The initial response times of CGI, X-Switch and commercial Java Web application servers.*

Subsequent requests issued to the experimental Web applications servers revealed that CGI could not improve upon its initial response time, understandably so, as a new process is initialised and the Web component reloaded from disk for every request. Figure 3 shows the drastic improvement of X-Switch system performance as subsequent requests arrive for processing. The X-Switch system exhibits a significant performance improvement when compared with CGI. The four commercial Java Web application servers (Tomcat, Jetty, Resin and Orion) show a significant improvement on their initial values (see Figure 4). The X-Switch system follows with a similar improvement trend and, as seen in Figure 4, provides performance that is comparable to the industrial Web application servers.

The results of the X-Switch PHP engine evaluation again showed that the X-Switch system provides a significant performance improvement over that of CGI. The initial response times were consistent with the results of the Java benchmark experiments in that the X-Switch system exhibits the longest initial response value but shows a dramatic performance improvement when subsequent requests are sent – these results are summarised in Table 2.

Engine	Initial Response
CGI	12ms
Mod_PHP	1ms
X-Switch	**18ms**

Table. 2. *The initial response times of the CGI, X-Switch and mod_PHP*.

The initial response timings shown in Table 2 again exhibit a high level of variation among engines. The X-Switch system again showed the longest initial response time of all the engines. This observation can also be attributed to the fact that the X-Switch start-up phase does not include interpreter process initialisation or script loading from disk, consistent with our Java experimental findings. The low initial response time of mod_PHP is made possible through the Apache Web server modular interface. mod_PHP is included as part of the Apache Web server core and initialised during the start-up phase of the Apache Web server. On arrival of the first request a process is immediately available to generate a response – the only delay would occur when a script is initially loaded from disk.

Once again the X-Switch system shows a drastic reduction in response times (performance improvement) when subsequent requests are sent to the server – see Figure 5. The mod_PHP results remain consistent throughout the experimental series of 5000 requests, which were attributed to the absence of process initialisation as mentioned above. Essentially, mod_PHP represents the most efficient means of executing a PHP script as it is included as part of the Web server but has been shown to be undesirable for two reasons. Firstly, should the Web server process that is executing the PHP script crash (due to a buffer overflow attack for example), the Web server system may be brought down. Secondly, mod_PHP does not provide any context switching support, which is essential for a secure, multi-user environment. It is clear that the X-Switch system provides performance that is a significant improvement over CGI and comparable with mod_PHP, while providing essential context switching abilities and process isolation.

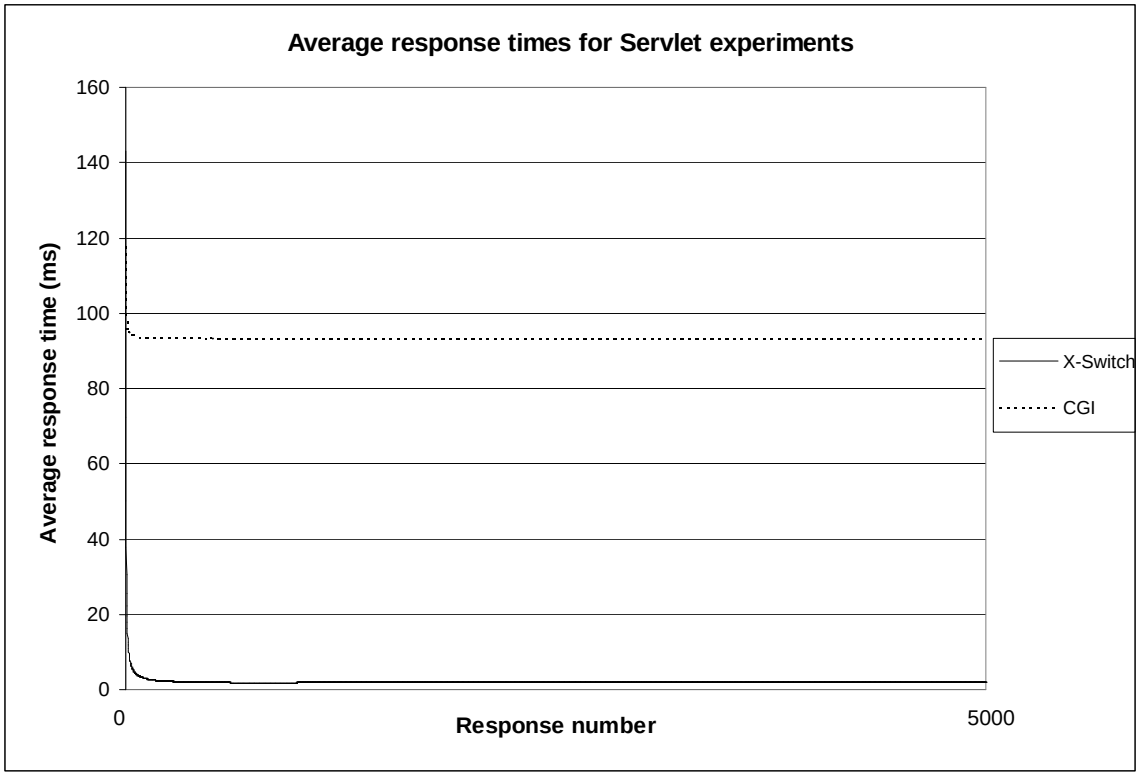

Figure. 3. *Comparison of X-Switch Servlet engine to CGI.*

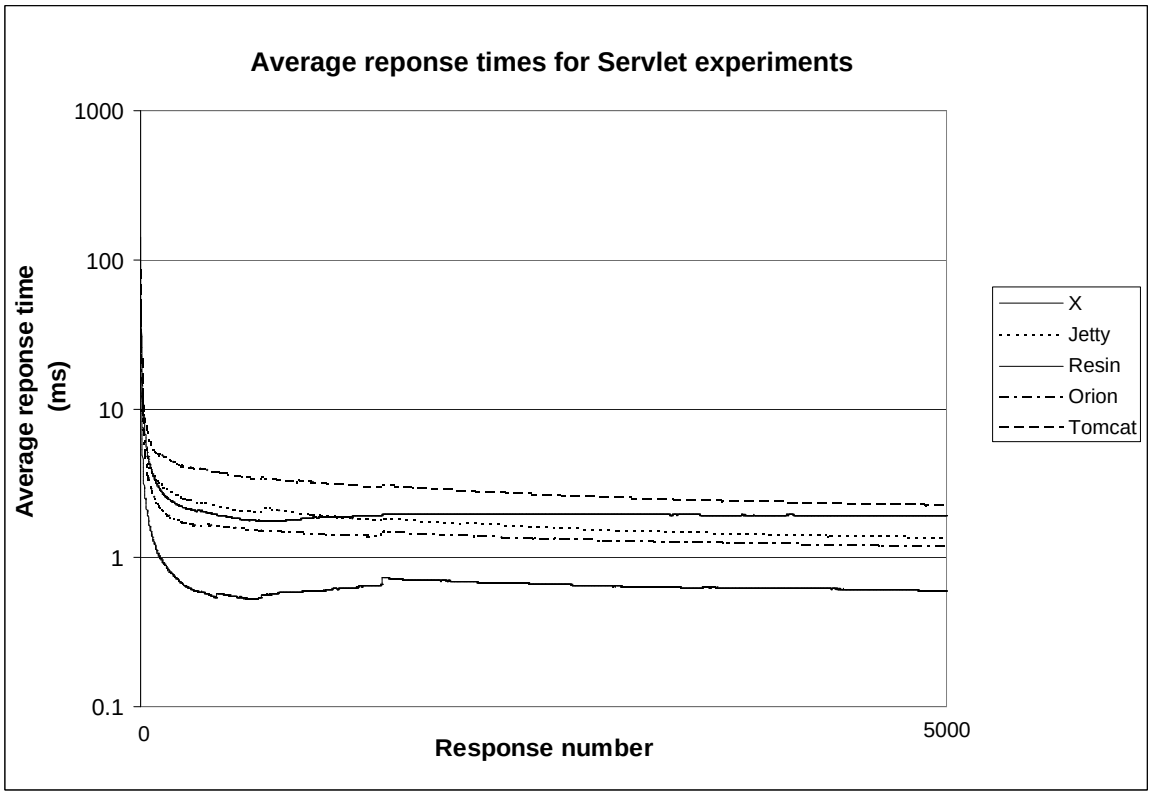

Figure. 4. *Comparison of X-Switch Servlet engine to other Java Web application servers.*

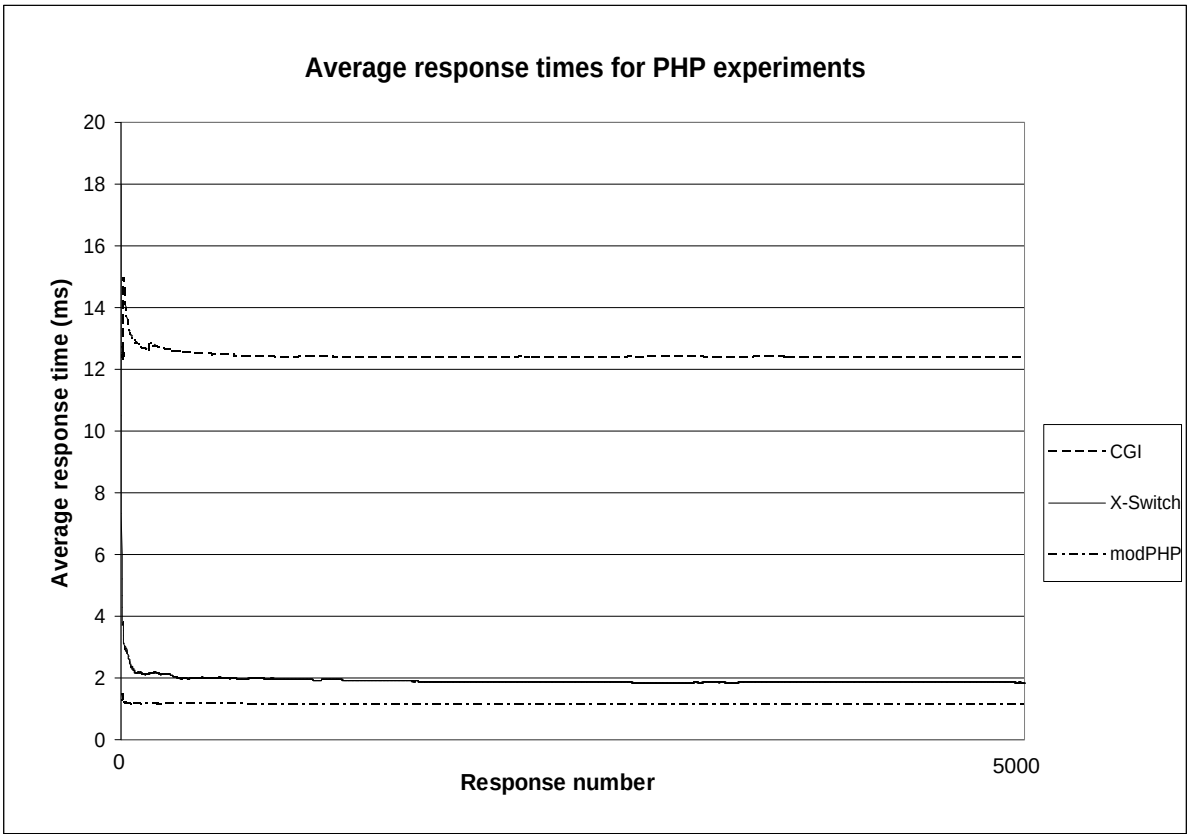

Figure 5. *Comparison of response times of CGI, X-Switch and mod_PHP for PHP processing.*

5. CONCLUSIONS

The X-Switch system has demonstrated 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 but far superior to that of CGI. 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.

6. FUTUREWORK

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 born. 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.

7. REFERENCES

APACHE SOFTWARE FOUNDATION. 2005. *suEXEC Support*. http://httpd.apache.org/docs/2.0/suexec.html

CHOPRA, V. et al. 2003. *Apache Tomcat Security Handbook*. Wrox Press, Hoboken, NJ.

HORROCKS, S. 2001. *Speedy CGI/Persistent Perl*. http://daemoninc/speedyCGI

LAURIE, B. AND LAURIE, P. 1999. *Apache: The definitive guide*. O'Reilly, Sebastopol, CA.

MARSCHING, S. 2005. *suPHP*. http://www.suphp.org

NEULINGER, N. 2001. *CGIWrap: User CGI access*. http://cgiwrap.unixtools.org

OPEN MARKET INC. 1996. *FastCGI: A high performance Web server interface*. http://www.fastcgi.com

OPEN WEB APPLICATION SECURITY PROJECT. 2004. *OWASP Top Ten Most Critical Web Application Security Vulnerabilities*. http://www.owasp.org

STEIN, L. 2003. *sBox: Put CGI scripts in a box*. Technical Report, Cold Spring Harbour Laboratory.

SUN MICROSYSTEMS. 2003. *The Java Servlet API: White Paper*. http://www.java.sun.com/products/servlet/whitepaper.html.

SUN MICROSYSTEMS. 1999. *The Java Security architecture*. http://www.java.sun.com/j2se/1.3/docs/guide/security.html

WORLD WIDE WEB CONSORTIUM. 1999. *The common gateway interface*. http://www.w3c.org/cgi/