Homayoun

Reference 39

A Self-Adapting Web Server Architecture: Towards Higher Performance and Better Utilization

Farag Azzedin Khalid Al-Issa
Information and Computer Sciences Department
King Fahd University of Petroleum & Minerals
Dhahran, Saudi Arabia
fazzedin@kfupm.edu.sa *khalid.issa@aramco.com*

ABSTRACT

The way at which a Web server handles I/O operations has a significant impact on its performance. Servers that allow blocking for I/O operations are easier to implement, but exhibit less efficient utilization and limited scalability. On the other hand, servers that allow non-blocking I/O usually perform and scale better, but are not easy to implement and have limited functionality. This paper presents the design of a new, self-adapting Web server architecture that makes decisions on how future I/O operations would be handled based on load conditions. The results obtained from our implementation of this architecture indicate that it is capable of providing competitive performance and better utilization than comparable non-adaptive Web servers on different load levels.

KEYWORDS: Operating systems; Internet and Web computing; Synchronous and asynchronous I/O; Concurrency

1. INTRODUCTION

Performance is a vital factor behind the success of Web-based services. As a result, working towards improving the performance of Web servers becomes a critical issue. As a matter of fact, the tremendous growth of Web-based services and applications over the past several years, the growth of network bandwidth, and the presence of a very demanding, large and growing community of Web users, are expected to put more heat on Web servers [16] [5] [2] [19] [18].

In general, performance can be looked at as either *macro* or *micro* performance [20]. A Web server's macro performance refers to the side of performance observed by clients, including throughput and response time. Micro performance, on the other hand, represents the server's internal performance, including lots of metrics like clock Cycles Per Instruction (CPI) and cache hit rate. While both of the two classes of enhancement contribute to the overall performance of a Web server, they differ in complexity, significance and effectiveness. For instance, a simple approach towards improving the overall performance is to use replication, which is precisely used to give better macro performance by providing multiples of the original throughput. This approach, however, only provides a workaround that would still suffer from the same set of issues that exist in the server architecture [19]. A more effective alternative would be to look into enhancing a Web server's micro performance, which would not only improve the overall performance, but also allows for eliminating major limitations and defects. As an example, some earlier work has shown that a server would be able to satisfy its clients with better throughput and response time if its cache locality is improved [11], or if more clients requests are taken every time the server accepts new requests [5] [3].

The primary task of Web servers is to deliver Web contents in a concurrent fashion. In order to deliver contents, frequent disk I/O operations have to take place, and that hits concurrency significantly. Concurrency is generally achieved either through asynchronous system calls to avoid blocking the server, or through multiple server instances using threads or sub-processes in which case the use of synchronous system calls becomes acceptable. The former approach is typically used in the Single-Process Event-Driven (SPED) Web server architecture. As shown in Figure 1, a client request is first accepted by the SPED server process. Then, the server performs all necessary computations as requested by the client. In case I/O operations are needed, the server would enqueue these I/O requests against an asynchronous system call like *select()*, which should relieve the server from having to block waiting for the I/O operation to be completed. While the I/O is being performed, the server may start processing computations associated with another

978-1-4244-4907-1/09/$25.00 ©2009 IEEE

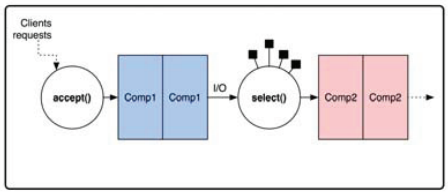

Figure 1. The SPED Web Server Architecture

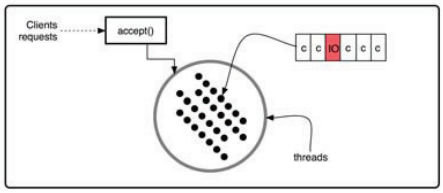

Figure 2. The Multi-Threaded Web Server Architecture

client request.

While the SPED model works pretty well when serving cached contents, it is not efficient when contents are to be fetched from disk, in which case the server is expected to interleave the serving of requests with slow I/O operations [16] [13]. In addition, asynchronous system calls available for use to implement concurrency in this model causes the server process to actually block in certain cases [16] [8]. The alternative for achieving concurrency through asynchronous calls is to run multiple server instances through either multiple processes or multiple threads. As shown in Figure 2, in a multi-threaded server, a thread is responsible for processing all computations as well as any needed I/O associated with the client request. While the multi-process and multi-threaded architectures are rather easier to implement [13] [4], they exhibit relatively low utilization of a server's resources since they allow processes and threads to block. In addition, the fact that every connection gets assigned a unique server thread or process has a negative impact on scalability. Moreover, the introduction of persistent connections in HTTP 1.1, which permits a connection to stay active while different objects are transferred, allowed for even less efficient utilization of the server [4].

The fact that both of the two alternatives have significant limitations in concurrently handling I/O operations and clients requests has motivated researchers to: (1) come up with hybrid architectures that combine certain features of the two approaches [16] [19] [4], (2) implement libraries to replace available, less efficient system calls [21] [8] [12], and (3) suggest and implement enhancements to these two architectures [11] [2] [5] [1] [9] [13] [3], . What is common about all the proposals we survey is that their technique for handling I/O operations is pre-determined, and can not adapt to the continuously changing state of the server. However, there are situations in which allowing a server thread to block is more cost-effective. There are also cases in which a server thread is so expensive that it should be allowed to only perform computations, and no I/O. As a result, we believe that a server should be allowed to determine what is best I/O scheme to follow based on load conditions.

This research work proposes the algorithm and structure of a self-adapting, multi-threaded server architecture that has the ability to switch between two different I/O schemes depending on load conditions. To the best of our knowledge, this is the first proposed adaptive Web server model that can follow more than one I/O scheme.

1.1. Motivation

Enhancing the way at which a Web server handles I/O has been an active topic in literature over the last decade. The key motivation was to propose ideas for higher concurrency, despite the long latency of disk I/O. Hybrid architectures, which rely on combinations of the two original approaches, are among the most interesting proposed ideas. For instance, one of the first hybrid models suggests employing a pool of helper processes whose role is to perform I/O. This relieves SPED servers from the need for inefficient asynchronous I/O, and greatly increases these servers' capacity.

While different hybrid models employed different techniques for achieving the primary goal of improving performance through increasing utilization and scalability, they are common in that they enforce a fixed I/O scenario. Proposed hybrid models that allowed blocking I/O to take place would allow that even under overloaded conditions. Similarly, models that utilize helper processes would pass I/O requests to helper processes even under lower load conditions, in which case blocking might both faster and less of an overhead. This motivates us to propose a self-adapting Web server architecture and evaluate its effectiveness to outperform non-adaptive architectures in both throughput and response time on different load conditions.

1.2. Objectives

The main objective of this research work is to outline two major limitations present in today's widely used I/O model in Web servers, the synchronous blocking I/O model. Scalability is highly affected due to allowing server threads to block for I/O. This would consequently have negative effects on a Web server's overall performance as it limits

throughput and increases response time. In addition, allowing server threads to perform blocking I/O operations represents an inefficient utilization of this valuable resource. Utilization becomes a critical issue as the rate of incoming clients requests increases while server threads are idle over I/O.

We still believe that blocking for I/O is the right choice under certain circumstances. Therefore, we propose in this paper a Web server architecture that can use both the blocking and the non-blocking I/O models under different load conditions to enhance performance and provide better utilization of server threads.

1.3. Contributions

This research work contributes to literature by first providing a survey and classification of current Web server architectures. Over the past several years, a number of architectures have been proposed to overcome limitations in the original models, as well as to improve performance and cope with the increasing popularity of Web-based services. In this research, we present a survey of these models along with a classification that is based on how they handle I/O requests.

Second, this research work brings to attention the need for adaptability in Web servers. This feature will enable the server to choose a more practical work scenario depending on past, current, or foreseeable circumstances.

Third, this research work promotes the use of asynchronous I/O for multi-threaded servers. The highly improved scalability obtained with this technique compared to the very common contender justifies it very well.

Last, this research work introduces a performance evaluation of an implementation of the proposed Web server architecture.

The paper is organized as follows: Section 2 presents a survey and a classification of existing Web server architectures. In Section 3, we explain the I/O models and outline strengths and weaknesses of each one of them. Section 4 describes the advantages of implementing self-adaptability in Web servers. Then, we describe the internals of the self-adapting Web server architecture in Section 5. In Section 6, we present the results obtained from our experiments in which we compare the performance of an implementation of the self-adapting Web server model to non-adaptive Web servers. We then explain how utilization is enhanced in the self-adapting Web server architecture in Section 7. Finally, Section 8 presents a conclusion of this paper, along with plans for future work.

Table 1. Some Existing Web Server Architectures

Year	Contribution	Class	Remarks
1999	AMPED	1	This is a SPED server that passes I/O requests to helper processes or threads
2001	Cohort scheduling	3	In this server, the order of executing threads is changed in order to execute similar computations consecutively, which reduces cache misses.
2001	SEDA	1	A pipelined server that consists of multiple stages, each is associated with a pool of threads.
2001	Multi-Accept	3	Instead of accepting a single incoming connection, a bulk of incoming connections are taken every time *accept() is called.*
2003	Cappriccio	2	A multi-threaded package that uses asynchronous I/O and provides high scalability.
2004	Lazy AIO	2	A new asynchronous I/O library that is meant to resolve issues with the available asynchronous libraries.
2005	Hybrid	1	A multi-threaded server that employs an event-dispatcher to resolve issues with allowing persistent HTTP connections.
2007	SYMPED	1	This server employs multiple SPED instances.
2008	MEANS	2	A software architecture that uses micro-threads for scheduling event-based tasks to Pthreads.

2. LITERATURE REVIEW

The multi-threaded and the event-based architectures are the original approaches for implementing a server. As each of the two has its own limitations and areas for improvement, many proposals over the last several years came to suggest and implement enhancements to overcome limitations and improve performance. We classify these proposals into three classes: (1) proposals for hybrid architectures, (2) proposals that suggest replacement libraries, and (3) proposals that disregard the I/O issue and focus on other aspects to improve performance. Table 1 summarizes our classification of the available Web server architectures.

2.1. Proposals for Hybrid Approaches

The first class of these proposals focused on deriving hybrid architectures that would combine features from the two original models. One of the early attempts was the asymmetric multi-process event-driven (AMPED) architecture [16], which provides a more effective solution for performing I/O operations in SPED servers, in which asynchronous system calls like *select()* are used. The AMPED

architecture is similar to SPED in that it typically runs a single thread of execution. However, instead of performing asynchronous I/O using descriptors through *select()*, AMPED passes I/O operations to helper processes. As a result, if blocking for I/O ever takes place, only helper processes will have to set idle and not the server's process.

A more recent hybrid model is the *Staged event-driven architecture* (SEDA) [19]. In SEDA, the entire work-flow of processing requests is re-structured into a sequence of stages, which makes it very similar to a simple pipeline. The main motivation behind introducing SEDA is to provide massive concurrency by allowing pools of threads to handle specific sets of tasks at the same time.

D. Carrera et. al. [4] proposes a solution for the case in which an idle client continues to hug a server thread, which is an undesirable consequence of allowing persistent connections in HTTP/1.1 for multi-threaded servers. To resolve this issue, they introduce a hybrid model in an event-dispatcher is used to identify sockets with readable contents and assign them to a server thread, which would read and process the request.

D. Pariag et. al [17] proposes the Symmetric Multi-Processor Event Driven (SYMPED) architecture. The SYMPED model consists of multiple SPED instances working together to increase the level of concurrency. Whenever one of these instances blocks for disk accesses, other instances can take over processing clients requests.

2.2. Proposals for Replacement Libraries

The second class of proposals started from the fact that available asynchronous system calls are not efficient, and suggested that they should be replaced. For instance, the way *select()* works requires it to block when used on disk I/O [16] [8]. Proposals in this area focus on providing replacements to these limited libraries. Elmeleegy et. al. [8] proposed Lazy Asynchronous I/O (LAIO), an asynchronous I/O interface to better support non-blocking I/O that would be more appropriate for event-driven programming. LAIO basically provides a non-blocking counterpart for each blocking system call. It handles blocking I/O operations for the application setting on top, while the application is allowed to move on with processing other requests.

Capriccio [1] is a scalable thread package that has the ability to scale up to 100,000 threads. This package was designed to resolve the scalability issue of the multi-threaded architecture. This high scalability in this solution was achieved through the use of *epoll()*, an asynchronous I/O interface that has proven to perform better than both the *select()* and the *poll()* interfaces with the right optimizations [9].

Lei et. al. [12], proposes MEANS, a micro-thread software architecture that consists of two thread-layers setting between the application and the operating system. An application that makes use of MEANS will assign work to MEANS micro-threads, which assign tasks in an event-based scenario to Pthreads interacting directly with the operating system.

2.3. Proposals for Modifying Other Architectural Components

The last class of proposals focused on enhancing the way the original models operate, while allowing the same I/O scenarios to take place. Chandra et. al. [5] and Brecht et. al. [3] suggest modifying the way a Web server accepts new connections. In the case of SPED, instead of accepting only a single new connection every time the server checks for incoming connections, they suggest accepting multiple connections [5]. This method increases the rate of accepting new connections, and increases concurrency of the server by providing more work that is ready to be processed at any instance.

Larus et. al. [11] suggest enhancing Web server performance by increasing locality and minimizing cache misses. They proposed a server model in which different requests are analyzed to identify similar computations. The order at which these requests are processed would be altered to allow similar computations to be processed consecutively. Executing similar computations as a group increases locality, and consequently improves performance.

3. I/O MODELS

Whenever an I/O operation needs to be performed by a Web server, the operating system actually takes care of it. This is due to many reasons, including maintaining a layer of security through which only privileged applications are granted access to certain files. While the I/O operation is being carried out, a server could either be blocked waiting for it to complete, or is free to process other requests. This depends entirely on the type of I/O the server initiated.

3.1. Synchronous I/O Operations

Synchronous I/O could be blocking or non-blocking to the calling process [10]. While both are performed through the same *read* and *write* system calls, the non-blocking requires the "O_NONBLOCK" option to be set when the *open()* system call is issued. The main issue with the synchronous non-blocking model is that it would require the calling process to send numerous calls to get the status of the requested I/O. As a result, this model is known to be extremely inefficient [10].

The synchronous blocking model, on the other hand,

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.