Table of Contents

List of Figures

List of Tables

1. Introduction

The World Wide Web (WWW or Web) [20] is rapidly increasing, with the number of users expected to reach 320 million by the year 2002. Correspondingly, the diversity of Web applications continues to increase, making Web Quality of Service (QoS) an increasingly critical issue in Web services [39].

With the term QoS, we refer to non-functional requirements, such as performance or availability requirements. QoS requirements deal with how an application or service will behave at run-time. QoS requirements may differ for different invocations of a service, based on factors such as the user or time of day. This is referred to as a differentiated QoS [1, 2].

Today, most Web servers do not provide differentiated QoS. The Apache Web server [12], one of the most widely used Web servers, handles incoming requests on a first-come, first-serve basis. All requests correctly received are eventually handled, regardless of the type of request on the Web server. In this situation, premium users cannot be protected from overload in the Web server. Further, the throughput in a general Web server is decreased during a peak period. Consequently, this general Web server does not provide differentiated QoS. The service outline of a differentiated quality of Web service is shown in Figure 1.

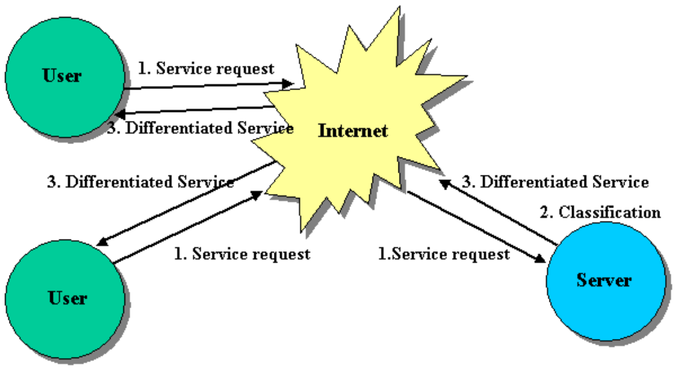


Figure 1. Differentiated Web Service

A diverse number of Internet users send service requests to the Web server, which provides quality of service in proportion to the level of payment. The incoming requests in the Web server are classified into different categories, and different levels of service are applied to each category. Using priority-based request scheduling, each request is served at different levels of quality. In this situation, even if many users request service at the same time, premium users can be protected from overload in the Web server because the requests of premium users are serviced first. This approach is termed a differentiated Web service.

In this thesis, we propose two approaches, a user-level and a kernel-level, to provide differentiated QoS using priority-based scheduling. In the user-level approach, the Apache Web server is modified to include a classification process, priority queues and a scheduler. It is difficult to achieve portability when modifying a specific Web server, such as the Apache Web server. We propose a new user-level approach to add a module, including classification and the scheduling of user requests. In the kernel-level approach, a Linux kernel [42] is modified by adding a real-time scheduler to support priorities among the HTTP processes which handle user requests.

Using the httperf benchmark tool [11] developed by Hewlett-Packard Research Labs, we measure the effect of the approaches on request latency, throughput and error rate per class. Further, both the general Web server and the differentiated Web server are compared using performance metrics.

In this thesis, we limit our investigation to the Web server only. However, we can easily extend current work to other Internet servers, such as the FTP server, the VOD server and the RealAudio server, and so on.

The rest of this thesis is organized as follows. Chapter 2 presents functional and non-functional requirements for design and implementation issues. Chapter 3 presents the system design architecture and classification approaches. Chapter 4 describes two prototype implementations. Chapter 5 describes the experimental environment and the workload used in this study and presents the performance evaluation results. Chapter 6 describes related work. Finally, chapter 7 summarizes the results and discusses some possible future work.

2. Requirements

In this chapter, we discuss requirements that must be considered during design and implementation. Requirements are divided into two parts – functional requirements and non-functional requirements.

2.1 Functional Requirements

In order to design and implement approaches to provide differentiated quality of Web service, certain functional requirements must be considered. The differentiated Web server requires the following functions: classification of requests, scheduling, and execution of requests.

The basic role of a general Web server is processing HTTP [20] packets on a first-come, first-serve basis. The general Web server listens to a signal which indicates that a connection has been made. After accepting the connection, the server must serve the user request. After serving the request, the server awaits the next signal.

Unlike the general Web server, the differentiated Web server must provide differentiated Quality of Service (QoS) using priority-based scheduling. Therefore, we modify the general Web server into the differentiated Web server to add components that support differentiated QoS.

The differentiated Web server listens to a signal which indicates a connection. After accepting the connection, the server classifies the user request. The classification of requests is the first concern in the implementation of a differentiated Web server. The classification methods are as various as can be imagined. For example, in a server based method, we can categorize a URL required by the user. In a client based method, we can classify the user request with the client IP address.

After classification of the user request, the server must save it into a priority queue. A scheduler which employs a specific scheduling method assigns the user request in priority queues. The scheduling method is the second concern in implementing a differentiated Web server. As a specific scheduling method is applied in this Web server, the throughput remains constant during peak demand. In addition, the error rate also remains constant during peak demand. After scheduling the user request, the server must serve the user request. Finally, after serving the request, this server is ready to receive another connection.

2.2 Non-Functional Requirements

To deliver differentiated Quality of Service (QoS), a Web server can be modified to include a classification process, priority queues and a scheduler. Yet it is difficult to achieve portability when modifying a specific Web server, such as the Apache Web server. A module which supports a differentiated QoS must use a variety of general Web servers, such as the Apache Web server and IIS Web server [43]. In addition, a differentiated Web server must be portable on various operating systems, such as Solaris, HP-UX, IRIX, AIX, Windows and Linux. Portability is the major concern among non-functional requirements.

General Web servers have evolved toward a multi-threaded architecture that either dedicates a separate thread to each incoming connection, or uses a thread pool to handle a set of connections with a smaller number of threads. A greater number of threads is used to process user requests, so more CPU capacity and memory usage is needed in general Web servers. Further, this condition is applied to a differentiated Web server equally. Because modules are added to support differentiated QoS, more CPU capacity and memory usage is needed in a differentiated Web server. Most Web servers do not serve user requests during peak demand or run short of CPU capacity and memory volume. The Web servers display a ‘service denied’ message in this situation. If both a general Web server and a differentiated Web server have an equal CPU capacity and memory volume, the latter must use as little CPU and memory usage as possible. Therefore, the resource requirements are a secondary concern in non-functional requirements.

3. Design of Two Approaches

In this chapter, we discuss two approaches towards a differentiated Quality of Service (QoS) in the Web server, and present a differentiated Web server architecture and process structures.

3.1 Differentiated Service System Architecture

In this section, we discuss the major components of these prototypes. A differentiated Web server for supporting differentiated QoS consists of a listen queue, a classification module, priority queues and a schedule module, as illustrated in Figure 2.

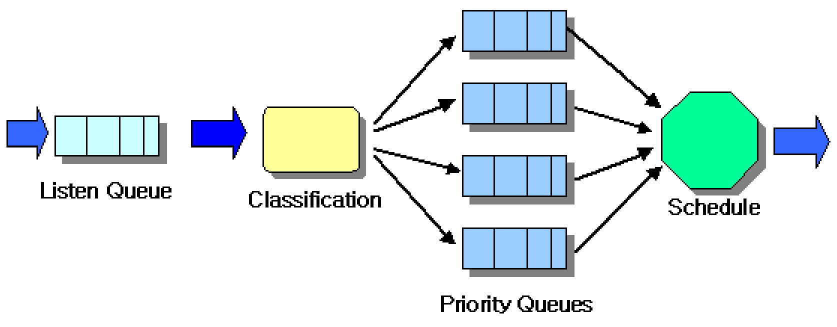


Figure 2. Differentiated Service System Architecture

All incoming user requests are saved in regular order at the listen queue. The classification module intercepts all requests that arrive at this server. This module classifies the request and places the request on the appropriate priority queue. The number of priority queues implies the number of differentiation levels. The schedule module receives requests from these queues instead of directly from the HTTP socket. A schedule module selects the next request, based on the scheduling method. For example, the requests from the highest priority queue will be processed first, according to priority-based scheduling. In the next sections, we present two approaches to support a differentiated QoS.

3.2 User-Level Approach

In this section, we present a user-level approach for supporting differentiated quality of Web service. In the user-level approach, the general Web server, such as Apache Web server is only modified to include components for supporting differentiated QoS. These components consist of connection & classification processes, priority queues, a schedule process and execution processes, as illustrated in Figure 3. The components such as classification processes, priority queues and the schedule process do not exist in a general Web server.

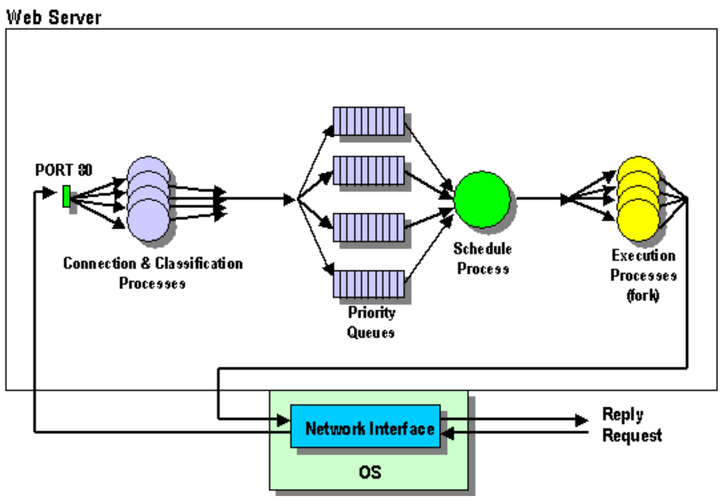


Figure 3. User-Level Approach

First, incoming user requests from the network interface in the operating system are received through port 80 by connection & classification processes. Port 80 is a well-known port for supporting World Wide Web (WWW) [20] service. These processes classify the requests and place the requests on the appropriate priority queues.

Several methods are used to classify requests. These classification mechanisms can be divided into two categories, a server-based and a client-based approach. In a later section, we explain the classification method in detail.

The number of priority queues implies the number of differentiation levels. Priority level is assigned per priority queue. After requests are classified according to classification methods, the server must actually realize different service levels for each class of requests. This is accomplished by selecting the order of request execution.

A schedule process selects the next request, based on the scheduling policy. For example, the requests from the highest priority queue will be processed first. Execution processes forked by the schedule process may be able to execute requests from any class, and will run until completion. Finally, execution results are sent to the user through a network interface.

Since we must modify a general Web server source code for supporting differentiated QoS in this approach, the prototype using this approach cannot handle the large variety of general Web servers. Because of this, we propose a new user-level approach to support portability in the next section.

3.3 New User-Level Approach for supporting portability

We describe the user-level approach to support a differentiated QoS.

However, the user-level approach can be only used in a specific Web server, and this approach does not support portability. For this reason, we propose a new user-level approach to support portability. The new user-level approach consists of a differentiate module and a general Web server, as illustrated in Figure 4.

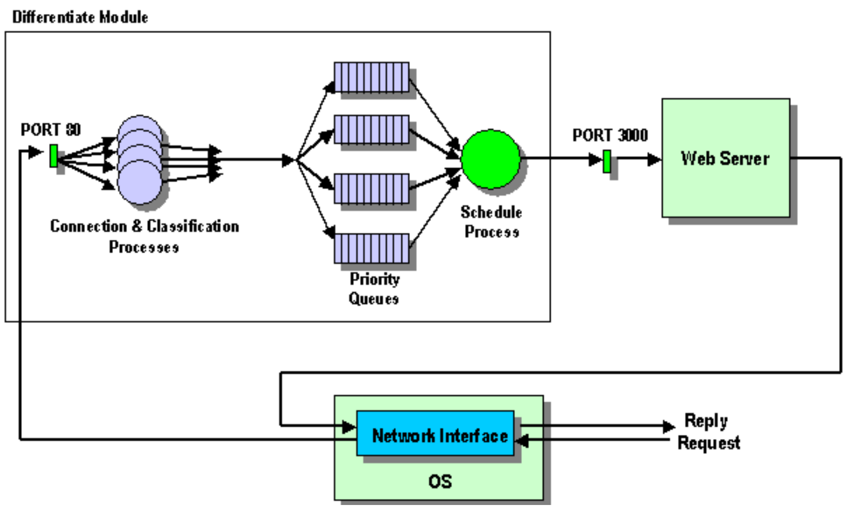


Figure 4. New User-Level Approach for supporting portability

The differentiate module is an independent program for the realization of a differentiated QoS. The components of the differentiate module are similar to that of the user-level approach in the previous section. Similarly, we do not completely modify a general Web server, such as the Apache Web server.

First, user requests arriving from the network interface in the operating system are received through port 80 by connection & classification processes in the differentiated module.

These processes classify the requests and place the requests on the appropriate priority queues in this module. Several ways exist to classify requests.

In a later section, we explain the classification method in detail.

After requests are classified according to classification methods, the server must actually realize different service levels for each class of requests. This is done by selecting the order of request execution.

A schedule process selects the next request based on the scheduling policy. A selected request by a schedule process is sent to a general Web server through a specific port, such as 3000. To accomplish this, the Web service port in a general Web server is configured into a specific port number, except for port 80. The selected request is processed in a general Web server. Finally, the execution result is sent to the user through a network interface.

3.4 Kernel-Level Approach

The motivation to use a kernel-level approach is that, processes which act directly on the priorities assigned to the HTTP request might be more effective in controlling their executions. Therefore, we use direct mapping from the user-level request priority to a kernel-level process priority.

The kernel-level approach is based on the instrumentation of both a general Web server modification and an operating system with real-time module, as illustrated in Figure 5.

This approach to support differentiated QoS consists of a modification of a Web server to support request classification and an operating system with a real-time module. A general Web server is modified to have each HTTP process call the kernel to record the priority of the request currently being handled. The kernel is responsible for mapping this priority into the process priority. The kernel scheduler decides which process should use the CPU next. The kernel must keep track of all processes currently using the priority, along with their current state.

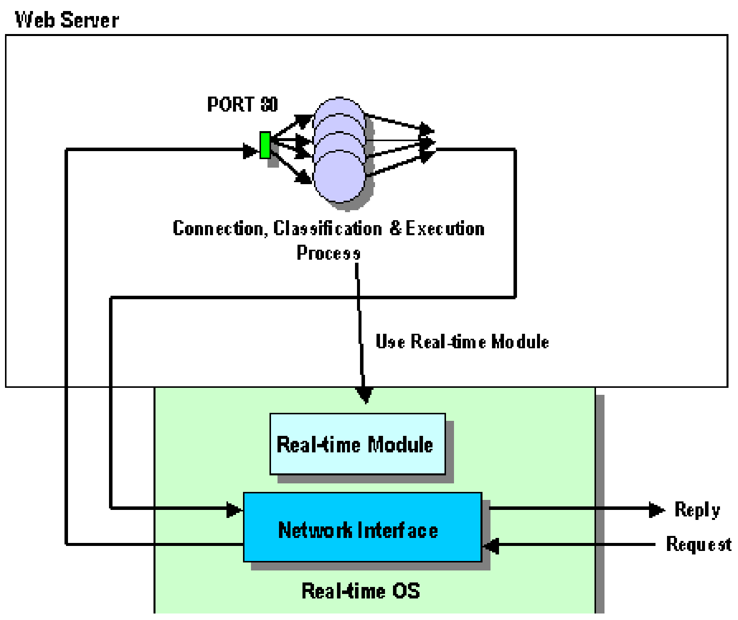


Figure 5. Kernel-Level Approach

User requests arriving from the network interface in the operating system are first received through port 80 by connection, classification and execution processes in a modified version of the Web server for supporting request classification. These processes classify the requests, and adjust the priority level by themselves.

The real-time module in an operating system is responsible for the assignment of request priority level. In generally, the real-time module, such as a specific real-time scheduler, can be added into an operating system that lacks a real-time property. After processing user requests, execution results are sent to the user through a network interface.

3.5 Classification Approaches

A key requirement to support differentiated QoS is the ability to identify and classify the incoming requests of each class. Several methods exist to classify requests. Classification mechanisms can be divided into two general categories: server-based and client-based approaches. The server-based approach classifies according to the content or destination of the request. However, the client-based approach characterizes requests according to the source of the request. First, we present the server-based approach, as illustrated in Figure 6.

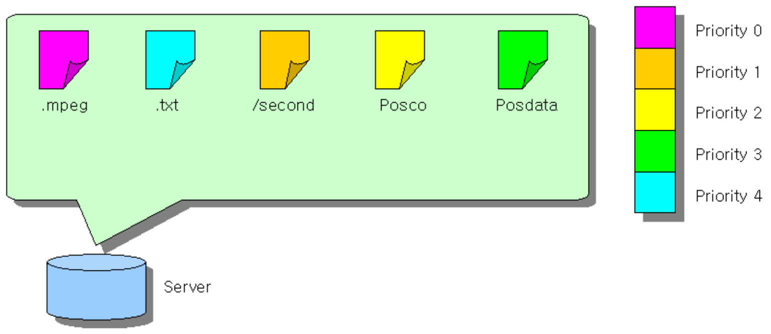


Figure 6. Server-based approach

We regard the URL [20] in the server as crucial. A URL consists of a protocol header, a server address, a directory name, a file name and an extension. This information can be used to classify the relative importance of the request. In this case, the sender of the request is irrelevant. Content can be classified into different priority levels. This allows e-commerce purchase activities, for example, to be given higher priority than browsing activities. Destination IP [20] addresses can be used by a server when the server supports co-hosting of multiple destinations (Web sites) on the same node. Another approach to classify user requests is the client-based approach.

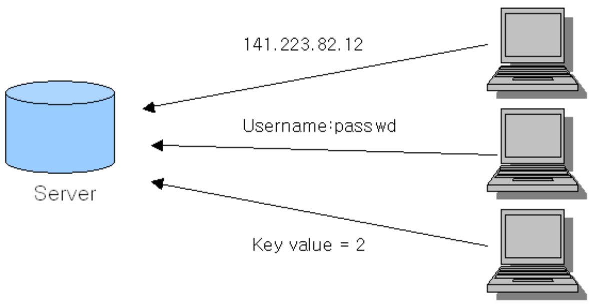


Figure 7. Client-based approach

The client-based approach characterizes requests by the source of the request, as illustrated in Figure 7. The IP address is used to distinguish clients from one another. This method is the simplest to implement. However, the client's IP address can be masked due to proxies or firewalls, so this method has a limited application. Further, user authentication through a username and password is used to classify a request. Finally, we use a key value to classify user requests. A key value is equal to a specific priority level.

3.6 Priority-driven scheduling methods in User-Level approach

After the requests are classified according to one of the above-mentioned classification schemes and admitted by the classifier, the server must actually realize different service levels for each class of requests. This is done by selecting the order of request execution. Execution processes are autonomous and select requests to process based on the scheduling policy. The scheduling policy may depend on queue lengths. Execution processes may be able to execute requests from any class. Alternatively, to reserve a capacity for higher-class processes, they may be restricted to executing higher-class traffic. This thesis presents several potential policy guidelines.

- Strict priority – This policy schedules all higher-class requests before lower-class requests, even when low-priority requests are waiting.
- Weighted priority – This policy schedules a class based on its weight importance. For example, one class will receive twice as many requests scheduled if its class weight is twice that of another.
- Shared capacity – This policy schedules each class to a set capacity and any unused capacity can be given to another class. The class may also have a minimum reserve capacity that cannot be assigned to another class.
- Fixed capacity – This policy schedules each class to a fixed capacity that cannot be shared with another class.
- Earliest deadline first – This policy schedules based on the deadline for completion of each request. This can be used to provide a guaranteed predicted response time.

In this thesis, we use a strict priority scheduling method to provide differentiated Web service. The strict priority scheduling method is very simple and requires less CPU capacity, memory volume than other priority scheduling methods. This condition is sufficient for non-functional requirements. If we use other priority scheduling methods, this prototype needs a resource management mechanism. An efficient resource management mechanism is used to economize resources in the Web server.

4. Implementation of the prototypes

We have implemented two prototypes for two distinct approaches to a differentiated Quality of Service (QoS). Specifically, one is the prototype of a new user-level approach, which is portable on various Web servers and operating systems. The other is a kernel-level approach, based on Linux and using a real-time scheduler. The C programming language, commonly used in the computing field, is used throughout the implementation of the two prototypes. We have implemented these prototypes on the Linux with kernel version 2.2.14 using the Intel Pentium III processor.

4.1 Development Environment

We have implemented two approaches on the development environment as summarized in Table 1. Obviously, the development environment of kernel-level approach is similar to that of new user-level approach, except that it uses the Montavista [40] real-time scheduler to support process priority on the operating system kernel level. The Montavista real-time scheduler is a specific scheduler for supporting soft real-time kernel, as presented in the previous chapter.

	New User-Level Approach	Kernel-Level Approach
OS	Linux Kernel 2.2.14	Linux Kernel 2.2.14
Realtime Kernel	None	Soft Realtime Kernel (Monstavista Realtime Scheduler)
Web Server	Apache 1.3.12	Apache 1.3.12
Language	C, PHP	C, PHP

Table 1. Development Environment

We have commonly implemented the prototypes of these approaches on Linux kernel 2.2.14 and Apache Web server 1.3.12. We use C language for programming the prototypes of these approaches as a whole. Further, PHP [41] script is used to configure both the classification policy and user information in these prototypes.

4.2 Montavista Real-time Scheduler

We have implemented a prototype of the kernel-level approach on the Linux kernel, version 2.2.14, with the Montavista real-time scheduler. Montavista's real-time scheduler replaces the kernel sched.c code with rtsched.c. This code separates real-time tasks in such a way that the scheduler overhead used to context switch real-time tasks has a nearly fixed load. This scheduler supports three scheduling algorithms or policies. These are SCHED_RR, SCHED_FIFO and SCHED_OTHER. SCHED_RR and SCHED_FIFO are real-time policies. SCHED_OTHER is the standard policy that this scheduler uses for non real-time tasks.

First, SCHED_RR (round-robin) is a time-sharing policy. SCHED_RR tasks are scheduled first by priority, and also have a time slice. Once a SCHED_RR task finishes its time slice, it transfers to the tail of its priority queue and is given a new slice, thus allowing other tasks of the same priority the opportunity for processing. If there are no other tasks with the same priority, the task will continue running with a new slice. The actual time slice used is not easily calculated, but is somewhere between 1 and 40 centiseconds.

Second, SCHED_FIFO (first-in, first-out) is a run-until-blocked policy. SCHED_FIFO tasks are scheduled by priority, and once started will run until they finish or block for some resource. They do not share the processor in the same manner that SCHED_RR tasks do.

Finally, SCHED_OTHER is a throughout scheduling policy. SCHED_OTHER tasks are scheduled in such a way as to attempt to maximize processor throughput and to provide reasonable responsiveness to interactive users. These two conflict, of course, as the best throughput is obtained by running a task until it blocks or finishes. SCHED_OTHER tasks will not run if any real-time tasks are ready.

4.3 New User-Level Approach Flow

We have implemented a prototype of a new user-level approach with no modification of the Apache Web server 1.3.12, as described in the previous chapter. Because a module for differentiating user requests resides in front of a general Web server, such as the Apache Web server, this prototype is portable on various general Web server and operating systems. This module consists of a request parser, a classifier and a request scheduler. The operation of this module is a sequence of functional flow, as illustrated in Figure 8.

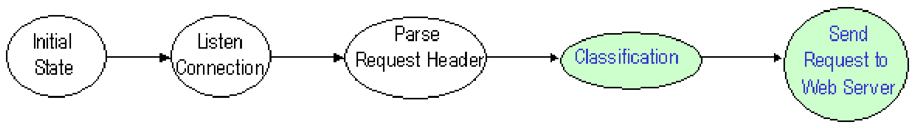


Figure 8. New User-Level Approach Flow

At first, this module is in the initial state, which exists for socket initialization, and so that the configuration file can be read. After initialization of the socket, a number of sockets listen to the user requests. If user requests enter the differentiation module, the request parser in this module reads the HTTP message. This parser parses the HTTP header and stores the parsing result. The classifier classifies user requests according to classification methods. At this time, the classifier uses the parsing result to classify user requests by placing the requests on appropriate priority queues in this module. The module must actually realize different service levels for each class of requests. This is accomplished by selecting the order of request execution. A scheduler selects the next request based on the scheduling policy. A selected request by a schedule process is sent to the Apache Web server through port 3000. In Figure 8, the shaded section is not in the Apache Web server.

4.4 Kernel-Level Approach Flow

We have implemented a prototype of the kernel-level approach with modification of the Apache Web server 1.3.12 and the Linux kernel 2.2.14 using the Montavista real-time scheduler, as described in the previous chapter. The priority can be set by API. The operation of this prototype is a sequence of functional flow, as illustrated in Figure 9.

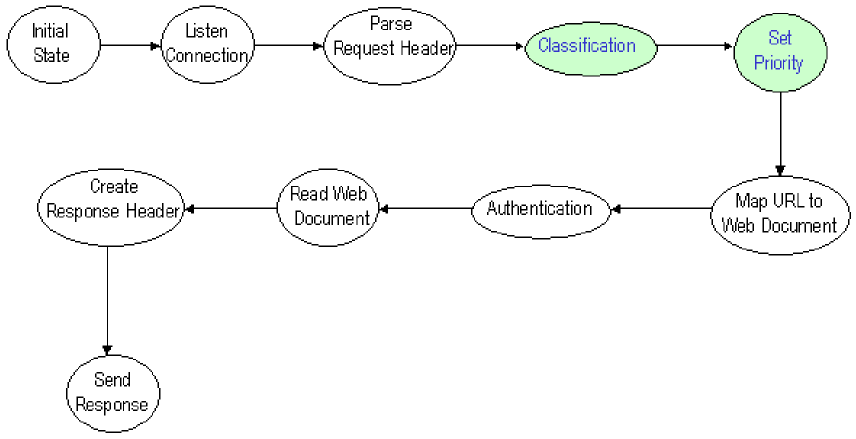


Figure 9. Kernel-Level Approach Flow

At first, this module is in the initial state, which exists for socket initialization, and so that the configuration file can be read. After initialization of the socket, a number of sockets listen to user requests. If the requests come into a modified Apache Web server, the request parser in this server reads the HTTP message, then parses the HTTP header and stores the parsing result. The classifier classifies user requests according to classification methods. At this time, the classifier uses the parsing result to classify user requests, and sets a priority for this executed process by using the API of the Montavista real-time scheduler. The Montavista real-time scheduler is responsible for realizing different service levels for each class of requests. The Montavista real-time scheduler selects the next execution process based on the scheduling policy. After prioritizing this process, the prototype determines the type of application interface and stores a pointer to the handler. Further, the prototype forces authentication of the user upon the URL, username and password. In the following sequence, this prototype read Web documents from file system and creates an HTTP response header. Finally, the execution result is sent to the user through the network interface. In Figure 9, the shaded section is not in the Apache Web server and Linux kernel 2.2.14.

5. Performance Evaluation

In this section, we evaluate the performance of two prototypes. We compare response times, throughput, and error rates for premium and basic client running with priority scheduling. The general Web server was run on a 600Mhz Pentium III with 128 MB of main memory running Linux 2.2.14. Further, The differentiated Web server was run on a 133Mhz Pentium III with 128MB of main memory running Linux 2.2.14. The test clients were run on a 233Mhz Pentium III with 128MB of main memory running Linux 2.2.14 and a 866Mhz Pentium III with 256 MB of main memory running Linux 2.2.14.

5.1 Performance Metrics

In this thesis, we use Httperf to measure Web server performance. It provides a flexible facility for generating various HTTP workloads and for measuring server performance.

Performance of a Web server is dependent upon a number of variables : the server hardware and operating environment, the server application, the network protocol and the network, hardware, bandwidth, and traffic load. Perception of this performance depends also on variables on the client side : the client platform and operating environment, and the Web client.

Three metrics are most often used to measure the capacity of Web servers.

- Throughput in reply rate per request rate, which is dependent upon the processing capacity of the Web server.
- Response time, which is a measure of the time it takes for a packet to be sent from the client, along with the time it takes for a response

to be received by the client, completing the request.

- Error rate, which is a measure of how many HTTP requests were lost or not handled by the Web server

5.2 Httperf

A Web system consists of a Web server, a number of clients, and a network that connects the clients to the server. The protocol used to communicate between the client and the server is HTTP.

In order to measure the server performance in such a system, it is necessary to run a tool on the clients that generates a specific HTTP workload. Currently Web server measurements are conducted using benchmarks such as SPECweb [22] or Webstone [24] which stimulate a fixed number of clients. Given that the potential user base of an Internet-based server is on the order of hundreds of millions of users, it is clear that stimulating a fixed and relatively small number of clients is often insufficient. For this reason, Banga and Druschel [14] argued the case for measuring Web servers with tools that can generate and sustain overload, which is effectively equivalent to stimulating an infinite user base. They also presented a tool called "s-clients" that is capable of generating such loads.

The s-clients approach is similar to Httperf in that both are capable of sustaining overload, but they differ significantly in design and implementation. For example, Httperf completely separates the issue of how to perform HTTP calls from such issues as what kind of workload and measurements should be used. Consequently, Httperf can readily be used to perform various types of Web-server related measurements, including SPECweb / Webstone-like measurements, s-client-like measurements, or new kinds of measurements, such as session-based measurements.

5.3 An example using Httperf

To show how the procedure is actualized using Httperf, this section presents a brief example.

The simplest way to achieve this is to send requests to the server at a fixed rate, and to measure the rate at which replies arrive. Running the test several times and with monotonically increasing request rates, one would expect the reply rate to level off when the server becomes saturated.

To execute such a test, it is necessary to invoke Httperf on the client machines. Ideally, the tool should be invoked simultaneously on all clients, but since the test runs for several minutes, startup differentials measured in seconds will not produce significant errors in the end result. A sample command line is shown below.

```
Httperf –server hostname –port 80 –uri /index.html –rate 150 –num-conn 27000 –num-call 1 –timeout 5
```

This command causes httperf to use the Web server on the host with IP name hostname, running at port 80. The Web page being retrieved is "/index.html" and, in this simple test, the same page is repeatedly retrieved. The rate at which requests are issued is 150 per second. The test involves initiating a total of 27,000 TCP [23] connections and on each connection one HTTP call is performed (a call consists of sending a request and receiving a reply). The timeout option selects the number of seconds that the client is willing to wait to receive a response from the server. If this timeout expires, the tool regards the corresponding call to have failed.

With a total of 27,000 connections at a rate of 150 per second, the total test duration is approximately 180 seconds, independent of the load the server can actually sustain.

When a test is completed, several statistics are printed. An example from Httperf is shown in Figure 10. The figure shows that there are six groups of statistics, separated by blank lines. The groups consist of the overall results, results which pertain to TCP connections, results for the requests that were sent, results for the replies that were received, CPU and network utilization figures, and a summary of the errors that occurred (timeout errors are common when the server is overloaded).

- Overall results – number of connections, number of requests, number of replies, test-duration(seconds)
- Connection – connection rate(connections/s), connection time(ms), connection length(replies/connections)
- Request – request rate(requests/s), request size(byte)
- Reply – reply rate(replies/s), reply time(ms), reply size(byte), reply status
- Cpu time(seconds) and network utilization(KB/s)
- Summary of the errors – total number of errors, cause of error

```
httperf --timeout=5 --client=0/1 --server=jordan.postech.ac.kr --port=80 --uri=/1.html --
rate=480 --send-buffer=4096 --recv-buffer=16384 --num-conns=25000 --num-calls=1
Maximum connect burst length: 4

Total: connections 19350 requests 3973 replies 3386 test-duration 52.081 s

Connection rate: 371.5 conn/s (2.7 ms/conn, <=1022 concurrent connections)
Connection time [ms]: min 6.0 avg 1830.1 max 6193.9 median 918.5 stddev 1464.8
Connection time [ms]: connect 1165.5
Connection length [replies/conn]: 1.000

Request rate: 76.3 req/s (13.1 ms/req)
Request size [B]: 77.0

Reply rate [replies/s]: min 0.0 avg 67.7 max 146.0 stddev 52.5 (10 samples)
Reply time [ms]: response 960.2 transfer 0.0
Reply size [B]: header 229.0 content 1358.0 footer 0.0 (total 1587.0)
Reply status: 1xx=0 2xx=3386 3xx=0 4xx=0 5xx=0

CPU time [s]: user 4.47 system 47.61 (user 8.6% system 91.4% total 100.0%)
Net I/O: 106.5 KB/s (0.9*10^6 bps)

Errors: total 21614 client-timo 4131 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 5650 addrunavail 0 ftab-full 0 other 11833
```

Figure 10. Statistics Example

5.4 General measurement architecture

In this section, we show the general measurement architecture for performance evaluation in both a general and a differentiated Web server. The experimental setup is illustrated in Figure 11. Test results are based on two client machines simulating many clients issuing uniform size HTTP GETs.

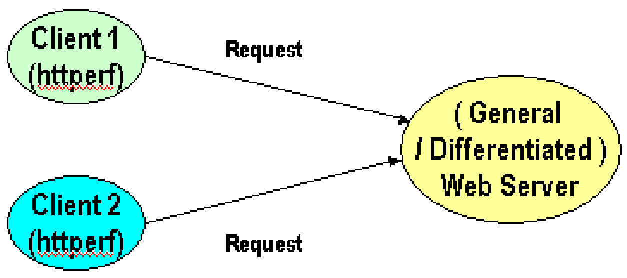


Figure 11. General measurement architecture

- Server : Two PCs (using one Pentium 133Mhz and one Pentium III 600mhz) running Linux 2.2.14
- Clients : Two PCs (using one Pentium II 233Mhz and one Pentium III 866Mhz) running Linux 2.2.14.
- Network : 100-base T Ethernet

The two clients communicate with the server over the 100-Base T Ethernet. On each client, the httperf application is configured to issue requests at a given rate and for a given static Web page. Each request has a five second timeout, and if no response arrives from the server within that time period, the client aborts the connection and logs an error. At the end of the test, statistics are collected for each client (number of successfully completed requests, average response time for there requests, number of requests that did not complete successfully). The client request rates are then increased and the process is repeated, yielding a graph that represents the performance as a function of an increasing offered load.

5.5 Measurement testbed for General Web Server

In this section, we show two measurement testbeds for performance evaluation in both a slow and a fast general Web server. The experimental setup is illustrated in Figure 12 and Figure 13. The measurement testbed consists of two servers and two clients. One server evaluates the slow general Web server and the other server evaluates the fast general Web server.

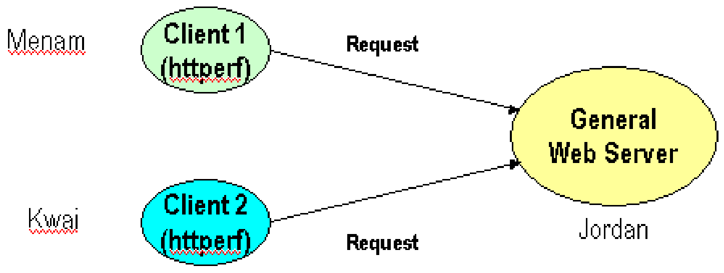


Figure 12. Measurement testbed for slow General Web Server

First, the slow server (Jordan) tested for performance evaluation is a Pentium 133Mhz PC, running Linux 2.2.14. The two clients (Menam and Kwai) communicate with the server over the 100-Base T Ethernet. Menam is Pentium III 866Mhz PC running Linux 2.2.14 and Kwai is Pentium II 266Mhz running same Linux version. On each client, the httperf application for performance evaluation is configured to issue requests at a given rate and for a given static Web page.

In the second, the fast server (Indus) tested for performance evaluation is a Pentium III 600Mhz PC, running Linux 2.2.14. The performance evaluation of the two clients is the same as with the slow general Web server.

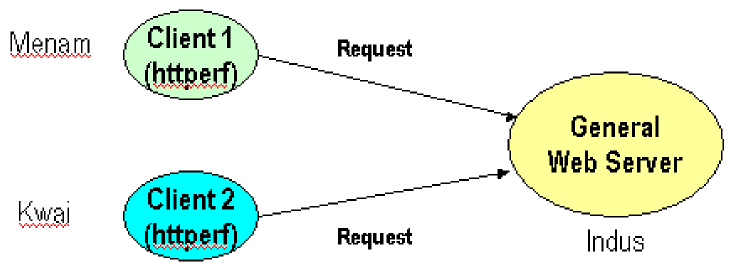


Figure 13. Measurement testbed for fast General Web Server

5.6 Measurement testbed for Differentiated Web Server

In this section, we show a measurement testbed for performance evaluation in differentiated Web server. The experimental setup is as illustrated in Figure 14. Pentium 133Mhz PC running Linux 2.2.14 (Jordan) is also used as the server for this performance evaluation.

In the following Figure 14, we show the two types of priority testbeds for differentiated Web server. The measurement testbed consists of one server and two clients. On each client, the httperf application is configured to issue requests at a given rate and for a given static Web page. One client (Menam) is configured for generating a high priority request and the other client (Kwai) is configured for generating a low priority request. In the first stage, the request rate for this evaluation is fixed as a specific number. The request rate increases gradually. In this evaluation, we examine the effect of priority in differentiated Web server. We use the same server for the above evaluation.

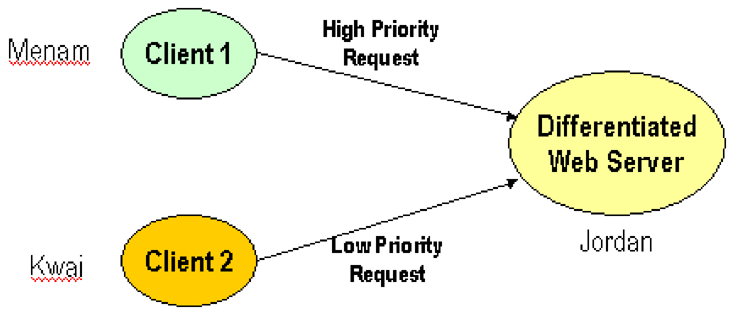
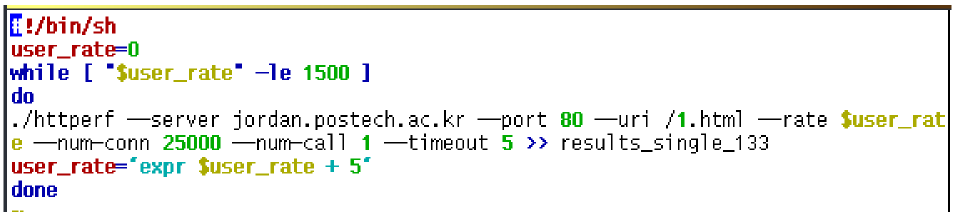


Figure 14. Measurement testbed for Differentiated Web Server

5.7 Method of measurement

The method of measurement is divided into two groups. One measurement method is for one client and the other is for two clients. With two clients, a synchronization problem can occur when generating requests. For example, httperf in two clients must begin experimentation simultaneously. To begin, we show the measurement methods for one client.

In performance measurement using one client, the httperf application is configured to issue requests at a given rate and for a given static Web page. Each request has a five second timeout, and if no response arrives from the server within that time period, the client aborts the connection and logs an error. At the end of the test, statistics are collected at the client (the number of successfully completed requests, the average response time for there requests, and the number of requests that did not complete successfully). The client request rates are then increased and the process is repeated, yielding a graph that measures performance as a function of the increasing offered load. This performance evaluation method is manually laborious. Therefore, we use shell script to repeat the above operation. The content of shell script is illustrated in Figure 15.



```
#!/bin/sh
user_rate=0
while [ "$user_rate" -le 1500 ]
do
./httperf --server jordan.postech.ac.kr --port 80 --uri /1.html --rate $user_rat
e --num-conn 25000 --num-call 1 --timeout 5 >> results_single_133
user_rate=`expr $user_rate + 5`
done
```

Figure 15. Shell script for performance evaluation in one client

In performance measurement using two clients, we have a synchronization problem. Because of this problem, "rdate" is used for timing synchronization. A sample command line is shown below.

```
Rdate –s [system]
```

[system] is a specific time server for timing synchronization. We use the "crontab" utility to execute a shell script periodically. We show sample content from crontab below.

```
00 09 * * * * rdate –s fraser.postech.ac.kr

01 09 * * * * perf.sh 100

10 09 * * * * rdate –s fraser.postech.ac.kr

11 09 * * * * perf.sh 150

20 09 * * * * rdate –s fraser.postech.ac.kr

21 09 * * * * perf.sh 200
```

In this content, the first columns are explained as a specific minute to execute this command line. The second columns are explained as a specific hour. The method for performance evaluation consists of a timing synchronization phase and a shell script execution phase. The two phases are executed alternately.

5.8 One client test in General Web Server

In this section, we evaluate both a slow and a fast general Web server using one client. The slow general Web server (Jordan) runs on a Pentium 133Mhz and the fast general Web server (Indus) runs on a Pentium III 600Mhz, as illustrated in Figure 16. Each client for this performance evaluation is Kwai and Menam. In 5.8.1 and 5.8.2, we show the performance evaluation results in graphs.

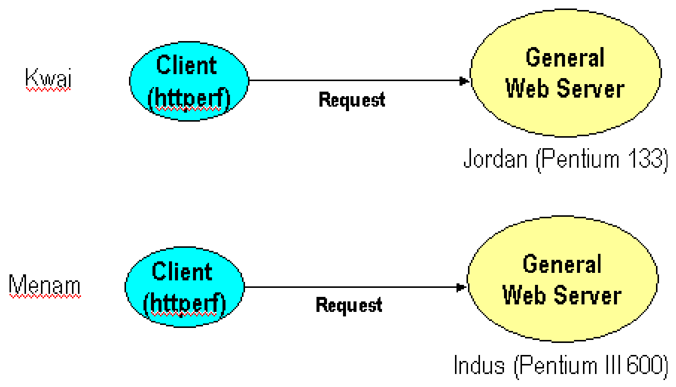


Figure 16. Testbed using one client for General Web Server

5.8.1 One client test in slow General Web Server

Our results are from the Apache Web server in Pentium 133Mhz machine (Jordan). Three graphs illustrate the experimental results for performance evaluation in a slow general web server. To test this server, we use only one client in this section. In the tests we analyze throughput, response time, and the error rate of a server handling one client whose rates monotonically increase from 5-850 requests/s. The client requests are for a fixed object of 1035 bytes. This experiment models a situation with a fixed number of clients. The first, Figure 17, plots the data throughput in a reply rate as a function of the total offered client demand rate. As shown in Figure 17, at low request rates the server can easily satisfy all client requests. As the offered load increases, the reply rate increases linearly until the request rate comes to approximately 60 requests/s. However, if the request rate is over 60 requests/s, reply rate is nearly fixed at 65 replies/s.

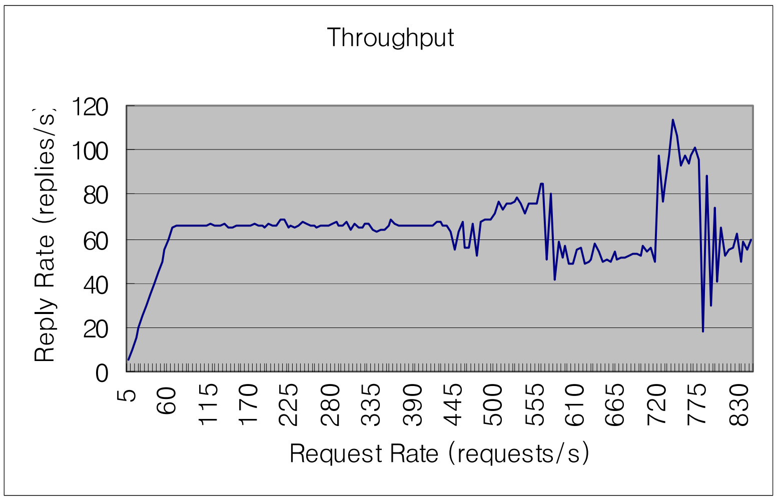


Figure 17. Throughput in slow General Web Server (one client)

The second graph, Figure 18, plots the average response time in milliseconds for all client requests completed successfully as a function of the total offered client demand rate. As the offered load increases, the response time does not increase rapidly until request rate approaches 225 requests/s. But if the offered load increases over 225 requests/s, the response time increases rapidly. At this time, response time ranges from 400 to 1700 ms. This graph shows the average response time for successful calls.

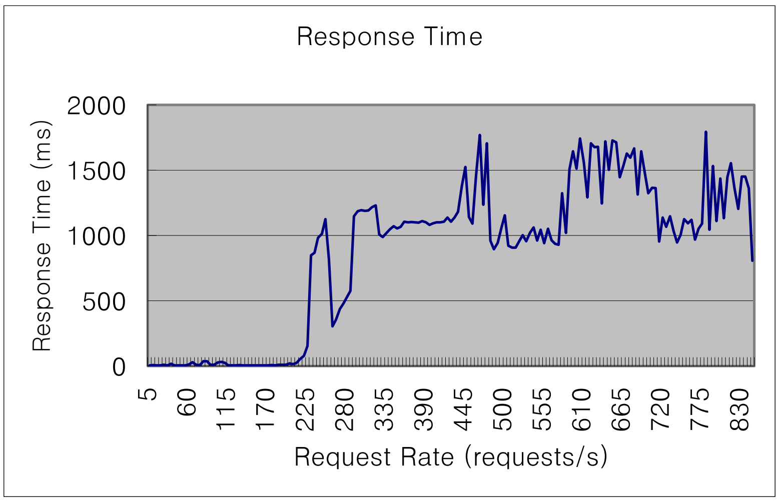


Figure 18. Response time in slow General Web Server (one client)

Finally, the third graph, Figure 19, plots client requests that encounter an error as a function of the total offered client demand rate. As shown in the figure, at low request rates the server can easily satisfy all client requests. As the offered load increases, the client consumes more and more bandwidth until the server's capacity is exhausted, at around 60 requests/s. At this point, the client requests begin to be rejected.

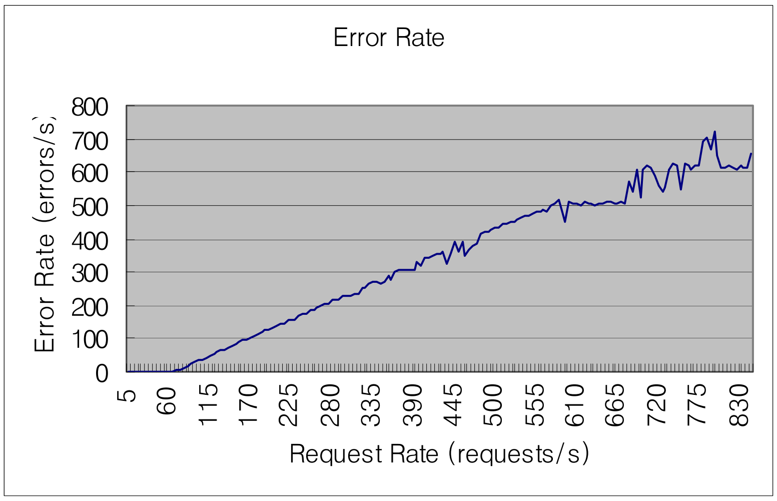


Figure 19. Error rate in slow General Web Server (one client)

5.8.2 One client test in fast General Web Server

In this evaluation, we use a faster server compared to the 5.8.1. Our results are from the Apache Web server in Pentium III 600Mhz machine (Indus). Further, three graphs illustrate experimental results for performance evaluation in a fast general Web server. To test this server, we use only one client in this section. Our tests analyze the throughput, response time, and error rate of a server handling one client, at rates that monotonically increase from 5-1400 requests/s. The client requests are for same size in 5.8.1. We show the data throughput in reply rates as a function of the total offered client demand rate in Figure 20. As offered load increases, the reply rate increases linearly until the request rate reaches approximately 75 requests/s. However, if the request rate exceeds 75 requests/s, the reply rate is nearly fixed at 65 replies/s.

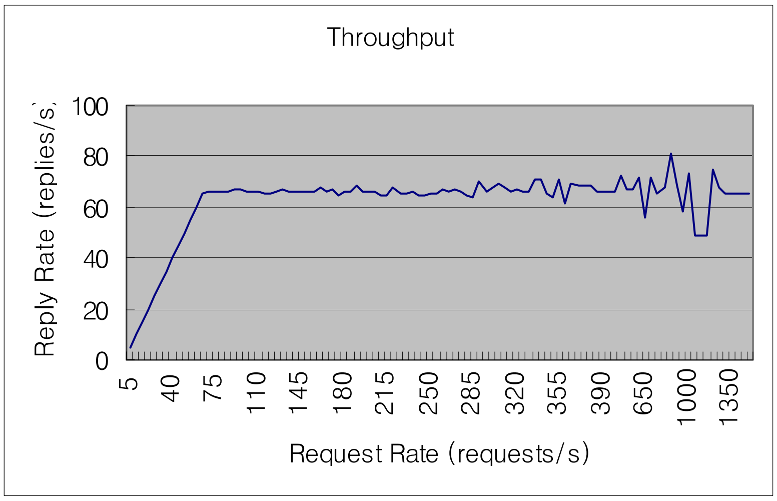


Figure 20. Throughput in fast General Web Server (one client)

The second graph, Figure 21, plots the average response time in milliseconds for all client requests completed successfully as a function of the total offered client demand rate. Contrary to the feature of Figure 18, this graph mysteriously flattens. This situation arises from high-speed client request processing and the insufficiency of the file descriptor in the Web server. This error is mostly due to the insufficiency of the file descriptor. Because of this, a few requests are queued in the Web server, which will be processed a few moments later. Consequently, the average response time is nearly fixed at 0.8 ms. However, if the request rate exceeds 1000 requests/s, the average response time increases monotonically.

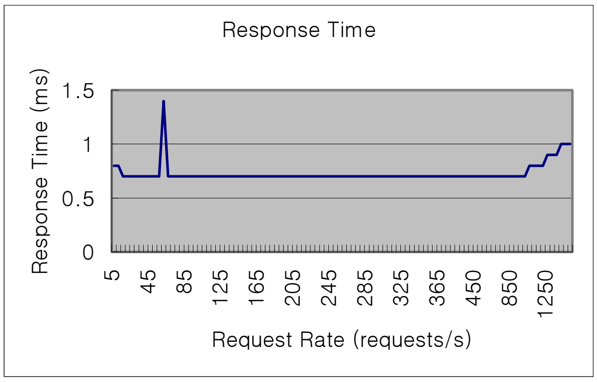


Figure 21. Response time in fast General Web Server (one client)

Finally, the third graph, Figure 22, plots the client requests that encounter an error as a function of the total offered client demand rate. As shown in Figure 22, at low request rates the server can easily satisfy all client requests. As the offered load increases, the client consumes more and more bandwidth until the server's capacity is exhausted, at approximately 75 requests/s. At this point, the client requests begin to be rejected. As the client requests is over 500 requests/s, the error rate linearly increases.

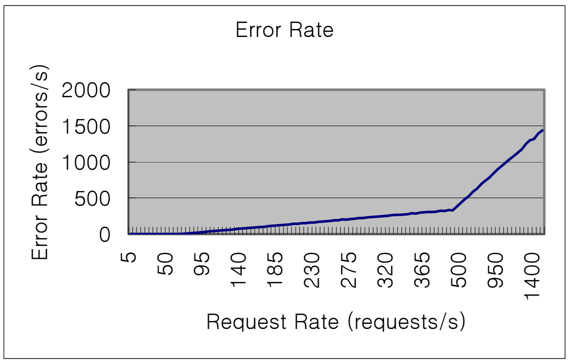


Figure 22. Error rate in fast General Web Server (one client)

5.9 Two clients test in General Web Server

In this section, we evaluate both a slow and a fast general Web server using two clients. The slow general Web server (Jordan) runs on a Pentium 133Mhz and the fast general Web server (Indus) runs on a Pentium III 600Mhz, as illustrated in Figure 16. Each client for this performance evaluation is the Kwai and the Menam. In 5.9.1 and 5.9.2, we will show the graph of performance evaluation results.

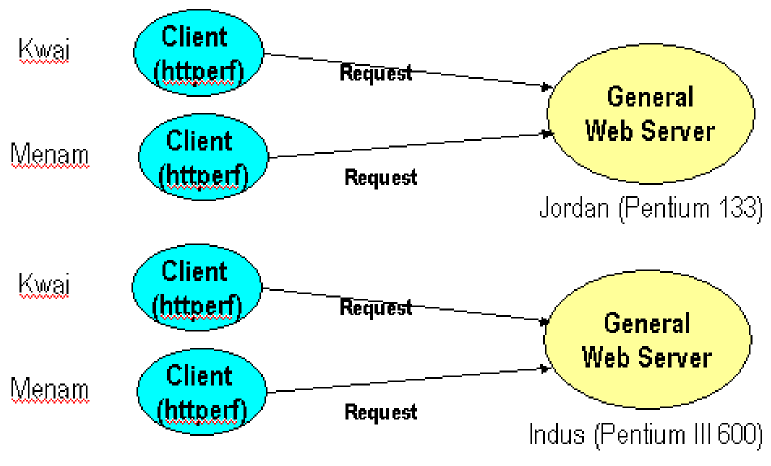


Figure 23. Testbed using two clients for General Web Server

5.9.1 Two clients test in slow General Web Server

In contrast to section 5.8, we use two clients in this evaluation. Our results come from the Apache Web server in the Pentium 133Mhz machine (Jordan). Three graphs illustrate the experimental results in a slow general Web server. In the tests we analyze the throughput, response time, and error rate of a server handling two clients at rates that monotonically increase from 10-750 requests/s.

The client requests are for same size used in section 5.8. The first, Figure 24, plots the data throughput in reply rate as a function of the total offered client demand rate. As shown in Figure 24, at low request rates the server can easily satisfy all client requests. One client is similar to the other client in the throughput graph. As the offered load increases, the reply rate for each client increases linearly until request rate on these clients come to approximately 70 requests/s. However, if the request rate for each client exceeds 70 requests/s, the reply rate to these clients is nearly fixed at 60 replies/s.

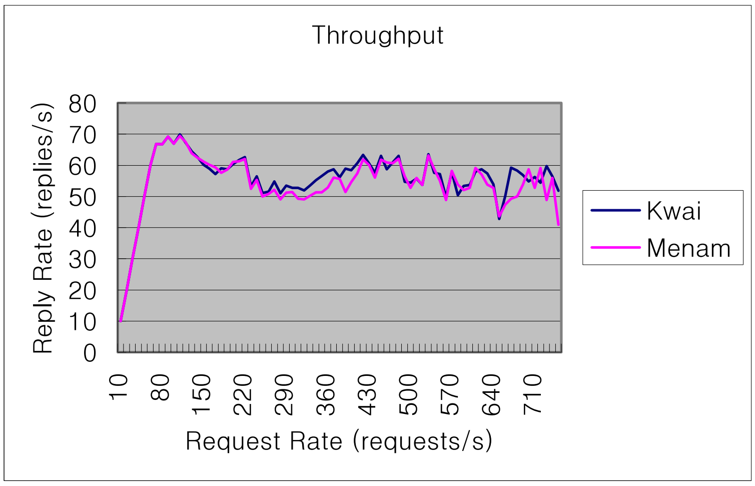


Figure 24. Throughput in slow General Web Server (two clients)

The second graph, Figure 25, plots the average response time in milliseconds. One client is similar to the other client in the average response time graph. As the offered load on each client increases, the response time does not increase rapidly until the request rate on each client reaches about 100 requests/s. If the offered load on each client increases over 100 requests/s, however, the response time increases rapidly. At the present time, the response time range is between 700 to 1200 milliseconds. This graph shows the average response time for successful calls.

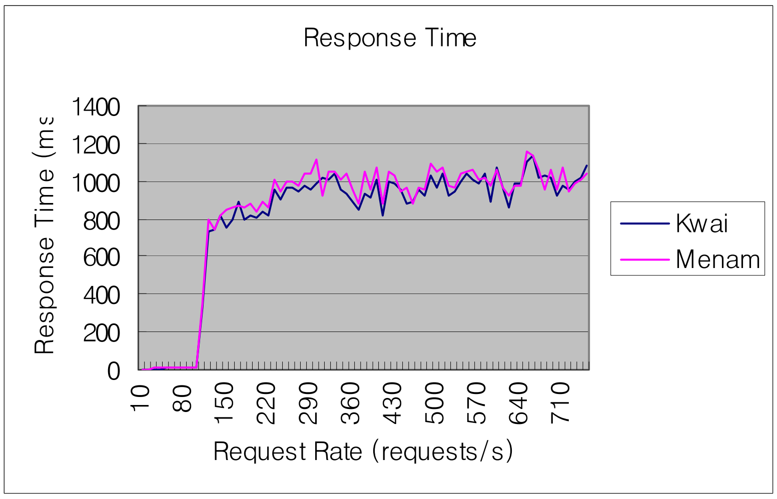


Figure 25. Response time in slow General Web Server (two clients)

Finally, the third graph, Figure 26, plots the client requests that encounter an error as a function of the total offered client demand rate. As shown in Figure 26, at low request rates, the server can easily satisfy all client requests. As the offered load on each client increases, each client consumes more and more bandwidth until the server's capacity is exhausted, at about 70 requests/s. At this point, the client requests begin to be rejected. At this time, the error rate increases monotonically.

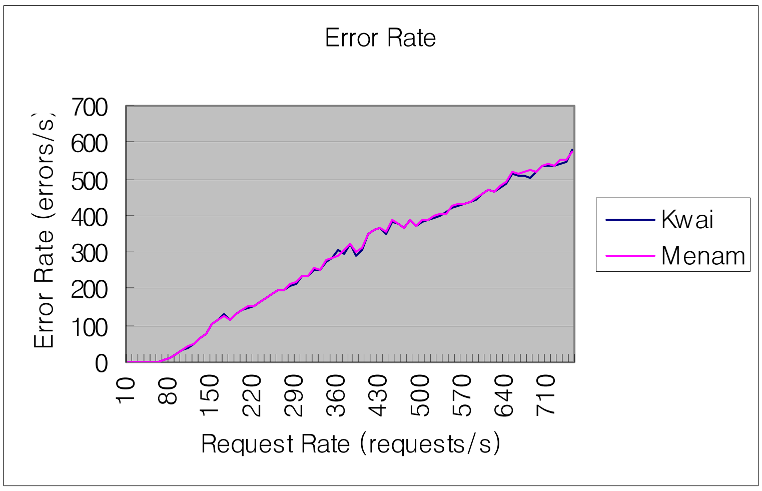


Figure 26. Error rate in slow General Web Server (two clients)

5.9.2 Two clients test in fast General Web Server

In this section, we use a fast server for performance evaluation. Our results are from the Apache Web server in Pentium III 600Mhz machine (Indus). Three graphs illustrate experimental results for performance evaluation in a fast general Web server. To test this server, we use two clients in this section. In the tests we analyze the throughput, response time, and error rate of a server handling one client at rates that monotonically increase from 10-1600 requests/s. The client requests are for a fixed object size of 1035 bytes. The first graph, Figure 27, plots the data throughput in reply rates as a function of the total offered client demand rate. As offered load increases, the reply rate increases linearly until the request rate reaches about 75 requests/s. However, if the request rate exceeds 75 requests/s, the reply rate ranges from 70 to 260 replies/s.

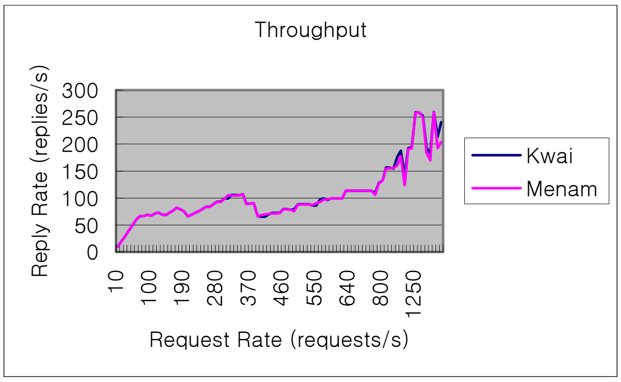


Figure 27. Throughput in fast General Web Server (two clients)

The second graph(Figure 28) plots the average response time in milliseconds for all client requests completed successfully as a function of the total offered client demand rate. One client is similar to the other client in the average response time graph. As the offered load on each client increases, the response time does not increase rapidly until the request rate on each client reaches approximately around 400 requests/s. However, if the offered load on each client increases by over 400 requests/s, the response time increases rapidly. At the present time, the response time ranges between 25 to 350 ms.

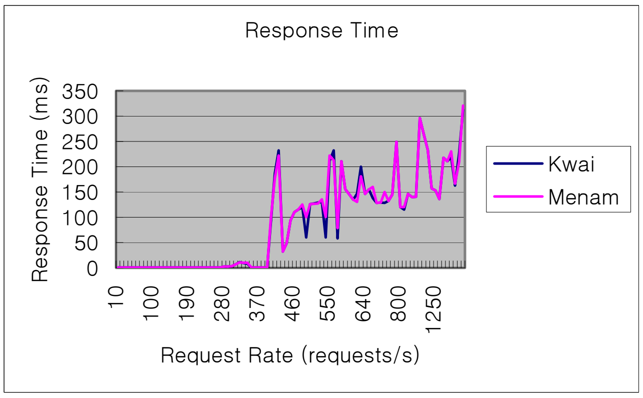


Figure 28. Response time in fast General Web Server (two clients)

Finally, the third graph, Figure 29, plots the client requests that encounter an error as a function of the total offered client demand rate. As the offered load on each client increases, each client consumes more and more bandwidth until the server's capacity is exhausted, at approximately 90 requests/s. At this point, the client requests begin to be rejected, and the error rate increases monotonically. Requests which exceed 1000 cause the error rate to increase rapidly.

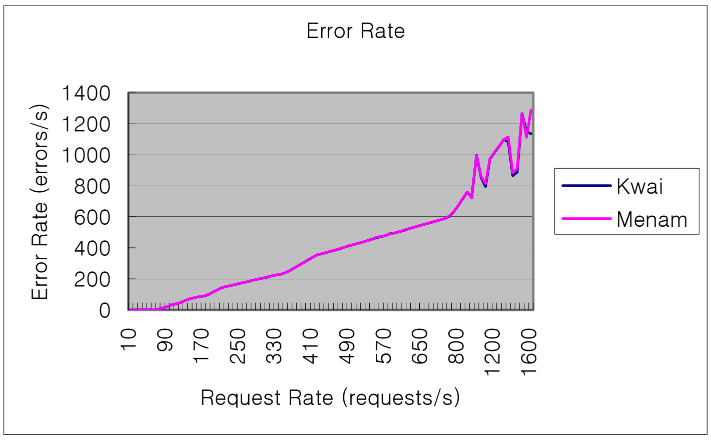


Figure 29. Error rate in fast General Web Server (two clients)

5.10 Comparative study between user-level and kernel-level approach

In Figure 14, we showed two types of priority testbeds for the differentiated Web server. In this section, we present a comparative study for both user-level and kernel-level approaches. We use a Pentium 133Mhz machine (Jordan) as differentiated Web server. To test this server, we use two clients, such as Menam and Kwai. The request rate in Httperf monotonically increases from 10-750 requests/s. The client requests are for a same size of previous evaluation. We hoped to verify if a high priority load can be protected as more and more traffic is added. The performance measurements are for high and low priority clients that monotonically increase with each client issuing an equal number of requests. Recall that the performance metrics are the throughput, the response time and the error rate of a request as perceived by the server. The proportion of high to low priority requests is 1:1. In Figure 30, 31 and 32, the high priority requests have a better throughput, response time and somewhat better error rates in both user-level and kernel-level approaches.

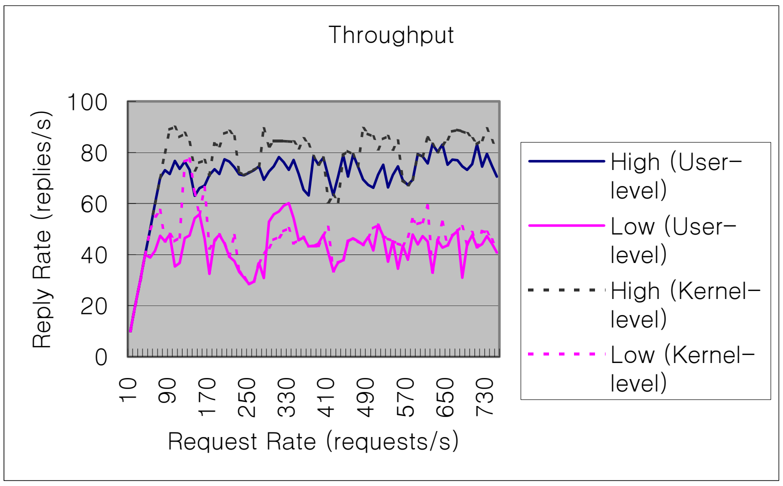


Figure 30. Throughput in Differentiated Web Server

In Figure 30, we show the throughput when we vary the request rate for the user-level and kernel-level approaches. As the offered load increases, reply rate increases linearly until request rate reaches about 90 requests/s in the high priority requests using kernel-level approach. Further, if the request rate is over 90 requests/s, the reply rate is nearly fixed at 90 replies/s. However, as seen in Figure 30, reply rate increases linearly until request rate reaches approximately 75 requests/s in the high pririty requests using a user-level approach. If the request rate exceeds 75 requests/s, the reply rate is nearly fixed at 75 replies/s. This status occurs in low priority requests using both user-level and kernel-level approaches, too. For the most part, the throughput for requests using the kernel-level approach is greater than for requests using user-level approach in both high and low priory requests.

The response time is optimistic since it does not include requests which are rejected or which encounter an error. The second graph, Figure 31, plots the average response time in milliseconds for all client requests completed successfully as a function of the total offered client demand rate. The differentiated Web server which uses the user-level approach is longer than one which uses the kernel-level approach in the response time. A new module for supporting portability in the differentiated Web server causes many processing delays.

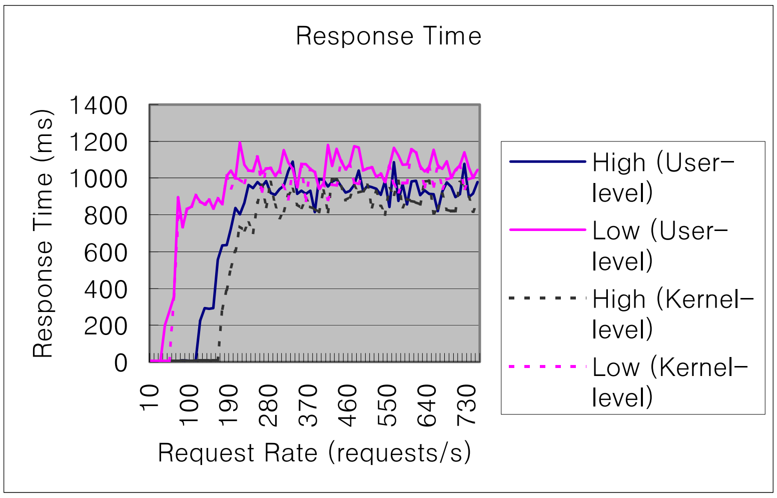


Figure 31. Response time in Differentiated Web Server

Finally, the third graph, Figure 32, plots client requests that encounter an error as a function of the total offered client demand rate in both the user-level and the kernel-level approaches. The error rate for requests using user-level approach is nearly equal to one using kernel-level approach in both high-level and low-level priority requests.

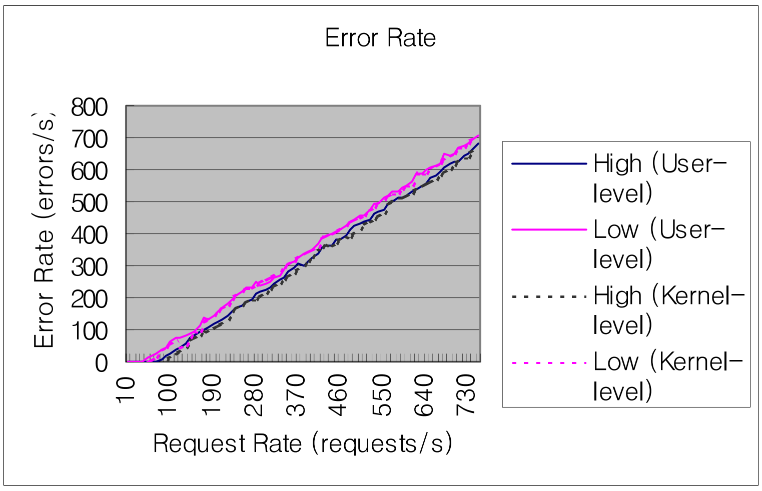


Figure 32. Error rate in Differentiated Web Server

Overall, a differentiated Web server which applies a user-level approach becomes lower than one using a kernel-level approach in performance. However, supporting portability in such a user-level approach is an important factor. Most Web server administrators wish that the facilities to support differentiated service are applicable to various Web server and operating systems. It is likely that, with the increasing complexity of Web page design, the differentiation between high and low priority requests will increase as well. In the performance evaluation of the differentiated Web server, the response of high priority requests are occasionally slower than one of low priority requests. In our view, the drop rate of low priority requests will be faster than one of high priority requests. In this situation, the length of the low priority queue will be temporarily shorter than that of the high priority queue. So this problem results. Future work will be devoted to ascertaining the exact cause of the problem.

6. Related Work

Much previous work has focused on the concept of a Quality of Service (QoS) for Web servers. To cite some examples, R.Pandey presents a notion of QoS that enables a site to customize how an HTTP server should respond to external requests. It sets priorities among page results and allocates server resources by using distributed Web servers [6]. M.Banatre uses profile-based predictive prefetching to anticipate user requests, based on access history [3]. Kevin Bayer presents a novel approach, which allows an organization to offer external access to their data while ensuring a desired level of service for local users [4]. The local users are given high priority and the external users a lower priority. A complete architecture for Web server QoS is given by Bhatti and Friedrich in [5]. Their goals were to manage peaks in client request rates and to support differentiated QoS for users. Their solution essentially entails scheduling and admission control to improve the performance of high priority requests. Their architecture includes a management component so that configuration parameters can be remotely set and the server's operation monitored. This closely resembles our work. Yet, there are several differences. First, changes to the main Apache server code were required for both request classification and scheduling. Our user-level prototype performs similar tasks without modifying the main Apache code. Second, they do not have the kernel-level approach concept. Nikolaos presented the design of a QoS architecture which can be added to any Web server to allow the server to provide a differentiated QoS that can also be externally managed at run-time. They do not have the kernel-level approach concept, however [1].

6.1 HP WebQoS

This architecture for a Web server QoS is presented by Bhatti and Friedrich in [5]. HP WebQoS version 2 is an enhancement to the HP-UX operating environment, providing a unique and advanced platform for assuring high service quality for e-services on HP 9000 Enterprise Servers.

HP's WebQoS stabilizes service delivery, assuring fast and consistent service quality to customers even under the extreme operating conditions found on the Internet. WebQoS optimizes resources and permits developers to build more cost-effective solutions. WebQoS also enables businesses to prioritize service-levels allowing higher service quality for the most important users and applications.

The WebQoS version2 technology creates a New Web Order by transforming the server operating system from a passive to an active work engine. WebQoS proactively monitors service-levels and adjusts workload scheduling and processing rates based on policies configured by the administrator. It enhances network services on the HP-UX operating system. WebQoS schedules http requests into three different priority queues.

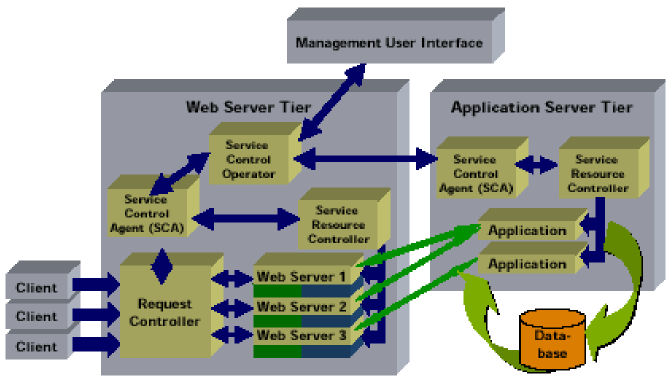


Figure 33. WebQoS Architecture

HP WebQoS v2 is comprised of four components, as illustrated in Figure 33.

- Request Controller
- Service Resource Controller
- LD Controller
- Management System

Request Controller classifies requests, controls admission and queues requests. Requests are classified into high, medium, or low priority, based on the configured policy. Three priority levels are used to determine the admission policy and performance-level. Once the initial request is classified, a session is marked and all subsequent requests associated with that session continue to be classified at the same priority until the entire session is reclassified at a higher priority. For example, a user who adds contents to a shopping basket and begins the purchasing process can be assigned a higher priority, after a request is sent to the "add to basket" URL.

Admission decisions for new sessions are based on the configured policy. Configurable admission policies are based on the user or service class and on various performance or resource conditions, such as CPU utilization, or length of the high, medium or low priority http request queues. HTTP requests for admitted sessions are then queued into high, medium, and low priority queues. The queues are serviced, based on the configured policy, resulting in different performance levels at each tier.

The Service Resource Controller is used to configure the minimum resources a group of processes is allocated. More resources would be allocated per unit of workload for higher priority services.

The LD Controller runs on Web server systems and controls the Cisco LocalDirector. The LD Controller is the basis of the HP & Cisco Dynamic LocalDirector (DLD) Solution. DLD is comprised of HP 9000 servers running WebQoS with Netscape Enterprise Server and Cisco LocalDirector.

The management system provides a GUI for creating, editing, and deleting Service Level Objectives (SLOs). SLOs are the capacity and performance objectives configured per user or service class.

6.2 Nikolaos's Work

This work discusses the services needed to provide a differentiated QoS to clients of a Web site, based on the client's identity and attributes. Nikolaos's work [1] integrated an implementation of the services with the Apache Web server and describes a service that can be used to create algorithms to suit a specific company's goal.

This work categorizes the components (graphically shown in Figure 34) as follows: Scheduling and Management. Requests can be categorized or grouped into classes. The requests which arrive at the Web server are not processed immediately, but are placed into a queue associated with the class to which the request belongs. Later, the request is allowed to be processed. A scheduler is responsible for deciding which and how many requests of each class will be processed at any one time. The components of the QoS Module architecture that directly support this strategy are called scheduling components. They include the Resource Table object, QoS Board, Registration and the Scheduler and Scheduler algorithms (these last two are encapsulated in the Scheduler Component).

The QoS module must also support changing scheduling algorithms and the values of parameters that are used by the scheduling algorithms. The components of the QoS Module architecture that support this are called management components. They include Auxiliary Functions (encapsulated in the Scheduler Component), Resource Manager and Communication.

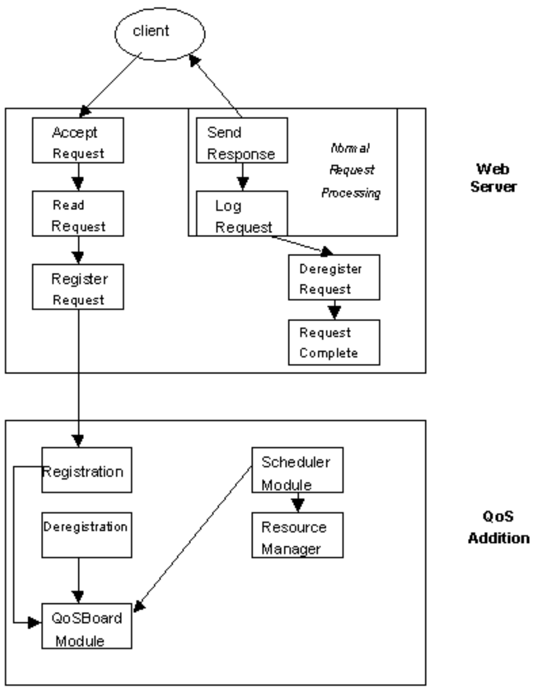


Figure 34. Nikolaos's work Architecture

6.3 Jussara's work

Jussara's work [2] investigates one method to provide a differentiated Quality of Service: priority-based scheduling. The main metric for the quality of service is latency in handling the HTTP request to the Web page. This work shows both a user-level and a kernel-level approach. However, These approaches differ from the approaches which are presented in this thesis. In the user-level approach, Apache is modified to include a scheduler process, and is responsible for deciding the order in which the requests should be handled. The scheduler restricts the maximum number of concurrent processes servicing requests of each priority. In the kernel-level approach, the Linux kernel is modified so that request priroities are mapped into priorities of the HTTP processes handling them. Using the Webstone benchmark suite, this work measures the effect of the approaches on request latency. Scheduling policies can be preemptive or non-preemptive. This work have implemented preemptive scheduling at kernel-level, and non-preemptive scheduling at user-level, since, at the user-level, we cannot interrupt a running process and block it in order to allow a new process to run. There are two important aspects of the scheduling policy. First, upon receiving a request, the scheduling policy must decide whether to process the request immediately, or to postpone the execution (Sleep policy). Secondly, the scheduling policy must also decide when a postponed requests must be allowed to continue (Wakeup policy). This work allows a request to continue only in place of a completed request. When a request is completed, the scheduling policy must decide which of the postponed requests, if any, should be selected to execute in its place. If the policy allows lower priority requests to execute in the absence of higher priority requests, the scheduling policy is said to be work-conserving. Otherwise, it is said to be non-work-conserving. A work-conserving policy tries not to allow processes to block while there are waiting requests. In this work, the Sleep and Wakeup policies have been implemented, by using thresholds for maximum number of requests that can be concurrently handled at each priority level. Thus, a fixed number of slots exist for each priority level, and each incoming request must either occupy a slot (executes), or wait in a queue until it is allowed to execute (blocks).

7. Conclusion and Future Work

We have presented prototypes for two approaches to support Web Quality of Service (QoS) which allows the general Web server to provide a differentiated QoS. Our scheme categorizes HTTP requests into classes based on classification methods, with the requests of each class handled differently by the desired scheduling method. The results obtained clearly demonstrate that providing a differentiated QoS is possible.

A high priority client will be given better throughput, response time, and slightly better error rates. The response time of the differentiated Web server which uses the user-level approach is slower than one which uses the kernel-level approach. A new module to support portability in a differentiated Web server causes many processing delays. Overall, a differentiated Web server which applies a user-level approach shows a lower performance than one using kernel-level approach. However, supporting portability in a user-level approach is considered an important factor. Most Web server administrators intend to implement a facility to support differentiated service in a variety of Web servers and operating systems.

In this thesis, performance measurements are for high and low priority clients that monotonically increase with each client issuing an equal number of requests. Our next research project focuses on analyzing the errors, throughput, and response time of a server handling one high priority client at fixed rate, and one low priority clients with request rates that monotonically increase. This experiment models a situation with a fixed number of clients with a subscription for high quality service. Further, we will provide comparative performance evaluation results of architectures supporting differentiated quality of Web services, as presented in Chapter 6. In the performance evaluation of the differentiated Web server, the response of high priority requests are occasionally slower than one of low priority requests. The cause of this problem will be ascertained in future work.

More work is needed on scheduling algorithms. There are many approaches that can perform the scheduling of requests. The open-queue system of the Web will require complex algorithms to balance the processing of requests in each class while maximizing system utilization. Designing a Web server framework to support server QoS is also a non-trivial task. The Web server framework consists of an admission controller, a resource manager, a disk scheduler and a request scheduler. More work is also needed to integrate a server QoS with a network QoS. It is necessary to unify their QoS parameters. Finally, we will extend current work on the Web server to other Internet servers such as the FTP server, the VOD server, the RealAudio server, and so on.

References

[1] N.Vasiliou and H. Lutfiyya., "Providing a Differentiated Quality of Service in a World Wide Web Server", Proc. of the Performance and Architecture of Web Servers Workshop, Santa Clara, California USA, June 2000, pp. 14-20.

[2] J. Almeida, M. Dabu, A. Manikutty, and P.Cai., "Providing Differentiated Levels of Service in Web Content Hosting.", Proc. of the Workshop on Internet Server Performance, Madison, Wisconsin USA, March 1998, pp. 91-102.

[3] M. Banatre, V. Issarny, F. Leleu, and B.Charpiot., "Providing Quality of Service over the Web: A Newspaper-based Approach.", Proc. of the sixth International World Wide Web Conference, Santa Clara, California USA, April 1997, Paper 149-Tec 110.

[4] Kevin Beyer, Miron Livny, and Raghu Ramakrishnan., "Protecting the Quality of Service of Existing Information Systems.", Proc. of the Cooperative Information Systems, 1998, pp. 74-83.

[5] N. Bhatti and R. Friedrich., "Web Server Support for Tiered Services.", IEEE Network, September/October 1999, pp. 64-71.

[6] R. Pandey, J. Barnes, and R. Olsson., "Supporting Quality of Service in HTTP Servers.", Proc. of the SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Puerto Vallarta, Mexico , June 1998, pp. 247-256.

[7] W. Feng, D. Kandlur, D. Saha, and K. Shin., "Adaptive Packet Marking for Providing Differentiated Services in the Internet.", University of Michigan Technical Report, CSE-TR-347-97, 1997.

[8] C. Aurrecoechea, A. T. Campbell, and L. Hauw, "A survey of QoS Architectures", ACM/Springer Verlag Multimedia Sys. J, Special Issue on QoS Architecture, vol. 6, no. 3, May 1998, pp. 138-151.

[9] Y. Bernet at al, "An Architecture for Differentiated Services", IETF, October 1998.

[10] T. Abdelzaher and N. Bhatti, "Web Server QoS Management by Adaptive Content Delivery", Proc. of the 7th International Workshop on Quality of Service, London, England, June 1999, pp. 216-225.

[11] D. Mosberger and T. Jin, "httperf: A tool for Measuring Web Server Performance", Proc. of the Workshop on Internet Server Performance,

Madison, Wisconsin USA, June 1998, pp 59-67.

[12] Apache Group, http://www.apache.org.

[13] V. Srinivasan et al., “Fast and Scalable Layer Four Switching”, Proc. of the ACM SIGCOMM '98 Conference, Vancouver, British Columbia Canada, September 1998, pp. 191-202.

[14] G. Banga and P. Druschel, “Resource Containers: A New Facility for Resource Management in Server Systems”, Proc. of the Symposium on Operating Systems Design and Implementation, New Orleans, LA USA, February 1999, pp. 45-58.

[15] M.Andrews, “Probabilistic end-to-end delay bounds for earliest deadline first scheduling”, Proc. of the IEEE INFOCOM 2000, Tel Aviv, Israel, vol. 2, March 2000, pp. 603-612.

[16] J. Mogul and K. K. Ramakrishnan, “Eliminating Receive Likelock in an Interrupt-Driven Kernel”, ACM Transactions of Computer System, Aug. 1997, pp. 217-252.

[17] C. Darst and S. Ramanathan, “Measurement and Management of Internet Services”, Proc. of the 6th IFIP/IEEE International Symposium of Integrated Network Management, Boston, MA USA, May 1999, pp. 125-140.

[18] Y. Bernet at al., “A Framework for End-to-End QoS Combining RSVP/Intserv and Differentiated Services”, IETF, March 1998.

[19] Gaurav Banga and Peter Druschel, “Measuring the capacity of a web server”, Proc. of the USENIX Symposium on Internet Technologies and Systems, Monterey, California USA, December 1997, pp. 61-71.

[20] R. Fielding, J. Getys, J. Mogul, H. Frystyk, and T. Berners-Lee, “Hypertext Transfer Protocol – HTTP/1.1”, IETF, January 1997.

[21] Rai Jain, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, NewYork, NY, 1991.

[22] SPEC. “An explanation of the SPECweb96 benchmark”, December 1996, http://www.specbench.org/osg/web96/webpaper.html.

[23] Richard W. Stevens, *TCP/IP Illustrated: The Protocols, volume 1*, Addison-Wesley, Reading, MA, 1994.

[24] Gene Trent and Mark Sake, “WebSTONE: The first generation in HTTP server benchmarking”, February 1995,

http://www.mindcraft.com/webstone/paper.html.

[25] V. Sivaraman, F. Chiussi, “Providing end-to-end statistical delay guarantees with earliest deadline first scheduling and per-hop traffic shaping”, Proc. of the IEEE INFOCOM 2000, Tel Aviv, Israel, vol. 2, March 2000, pp. 631-640.

[26] A. Bouch et.al, “Quality is in the Eye of the Beholder: Meeting Users’s Requirements for Internet Quality of Service”, Proceedings of ACM conference on Human Factors and Computing Systems, April 2000, Hague, Netherlands, pp. 297-304.

[27] L. Cherkasova and P. Phaal, “Session-based admission control – a mechanism for improving performance of commercial web servers”, Proc. of the 7th International Workshop on Quality of Service, London, England, June 1999, pp. 226-235.

[28] K. Li and S. Jamin, “A measurement-based admission controlled web server”, Proc. of the IEEE INFOCOM 2000, Tel Aviv, Israel, vol. 3, March 2000, pp. 651-659.

[29] R. Bless, K. Wehrle, “Evaluation of differentiated services using an implementation under Linux”, Proc. of the 7th International Workshop on Quality of Service, London, England, June 1999, pp. 97-106.

[30] J. Qiu and E. Knightly, “Inter-class resource sharing using statistical service envelopes”, Proc.of the IEEE INFOCOM’ 99, New York, NY, vol. 3, March 1999, pp. 1404-1411.

[31] M. Crovella and A. Bestavros, “Self-similarity in world wide web traffic: Evidence and possible causes”, IEEE/ACM Transactions on Networking, December 1997, pp. 835-846.

[32] P. Cao and S. Irani, “Cost aware WWW proxy caching algorithms”, Proc. of the USENIX symposium on Internet Technologies and Systems, Monterey, California USA, December 1997, pp. 193-206.

[33] E. Knightly and N. Shroff, “Admission control for statistical QoS: Theory and practice”, IEEE Network, March 1999, pp. 20-29.

[34] R. Cruz, “Quality of service guarantees in virtual circuit switched networks”, IEEE Journal on Selected Areas in Communications, August 1995, pp. 1048-1056.

[35] A. Myers, P. Dinda, Hui Zhang, “Performance characteristics of mirror servers on the Internet”, Proc.of the IEEE INFOCOM’ 99, New York, NY,

vol. 1, March 1999, pp. 304-312.

[36] S. Schechter, M. Krishnan and M.D. Smith, “Using path profiles to predict http requests”, Proc. of the 7th International WWW Conference, Brisbane, Australia, April 1998, pp. 457-467.

[37] E.Levy-Abegnoli, A. Iyengar, Junehwa Song, D. Dias, “Design and performance of a Web server accelerator”, Proc.of the IEEE INFOCOM’ 99, New York, NY, vol. 1, March 1999, pp. 135-143.

[38] A. Iyengar, E. MarNair, and T. Nguyen, “An analysis of web server performance”, Proc. of the IEEE Global Telecommunications Conference, volume 3, Phoenix, AZ, Nov 1997, pp. 1943-1947.

[39] N. Bhatti, A. Bouch, and A. Kuchinsky, “Integrating User-Perceived Quality into Web Server Design”, Proc. of the 9th International World Wide Web Conference, Amsterdam, Netherlands, May 2000, pp. 92-115.

[40] Montavista Software, www.montavista.com.

[41] PHP script, www.php.net.

[42] Linux Online, www.linux.org.

[43] Microsoft TechNet, http://www.microsoft.com/technet/iis.