Application Level Differentiated Services for Web Servers

By

Gautam Rao

A Master's Project Report

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major : Computer Science

Under the Supervision of Professor Byrav Ramamurthy

University of Nebraska-Lincoln

April 2000

Acknowledgements

I would like to express my gratitude to Dr. Byrav Ramamurthy for helping me at all steps, including choosing the topic and research, leading to the successful completion of my project. His guidance and advice have been invaluable to me. I sincerely thank my committee members, Dr. Steve Goddard and Dr. Hong Jiang who have taken their time to review my report and for their very helpful comments about the report. I also wish to thank S. Sankaravadivelu and T. Schroeder for their valuable help and advice.

Gautam Rao

University of Nebraska, Lincoln

April 2000

Table of Contents

List of Figures iv

List of Tables v

1. Introduction 2

2. Background and Previous Work 5

3. Design and Implementation of DiffServer 10

3.1 Design of the DiffServer Architecture 11

3.2 Advantages of using Threads over a Process based Model 12

3.3 Purpose of the Options.conf file 13

3.4 Parsing the URL.....14

3.5 Priority Determination.....15

3.6 Program Flow 17

4. Evaluation 21

4.1 Experimental Setup 21

4.2 Performance Measurement 22

5. Conclusions and Future Work.....30

References:.....32

Appendix A (options.conf listing).....35

Appendix B (Data Structure Overview) 38

List of Figures

Figure 1: Logical Model of the Diffserver.....10
Figure 2 : Conceptual Model of the DiffServer.....12
Figure 3 : Program Flow for Main Thread and Scheduler Thread.....19
Figure 4 : Program Flow for Child Threads.....20
Figure 5 : Connection Time and Response Time for HTML Connections.....27
Figure 6 : Connection Time and Response Time for *cgi-bin* Connections 28
Figure 7 : Average Waiting Time for File and Domain Based Priority.....29

List of Tables

Table 1 : Performance of HTML connections.....25
Table 2 : Performance of *cgi-bin* connections.....25
Table 3 : Average Waiting Time for File Based Priority 26
Table 4 : Average Waiting Time for Domain Based Priority.....26

Application Level Differentiated Services for Web Servers

Gautam Rao

Department of Computer Science and Engineering

University of Nebraska-Lincoln

Advisor : Dr. Byrav Ramamurthy

Abstract

Web content hosting, in which a Web server stores and provides Web access to documents for different customers, is becoming increasingly common. For example, a web server can host webpages for several different companies and individuals. Traditionally, Web Service Providers (WSPs) provide all customers with the same level of performance (best-effort service). Most service differentiation has been in the pricing structure (individual vs. business rates) or the connectivity type (dial-up access vs. leased line, etc.). This report presents DiffServer, a program that implements two simple, server-side, application-level mechanisms (server-centric and client-centric) to provide different levels of web service.

Chapter 1

Introduction

Due to the enormous growth of the Word Wide Web and the ever-increasing resource demands on the servers, Web content hosting is an increasingly common practice. Continuous growth of the Internet and emerging multimedia applications place demands for higher bandwidth on the Web Service Providers (WSPs). Contemporary networks are required to support higher and higher volumes of best-effort data as well as to differentiate among various classes of traffic that may include voice, music, and video. Companies expect that the requests from their potential clients (users that access their web pages) are serviced with a quality proportional to the amount of money they pay for these hosting services. Also, the system administrators may choose to give preferential services to requests from certain hosts and domains. As a result, WSPs are finding it necessary to offer their customers alternative levels of service. Besides meeting new customer expectations, this allows the WSPs to improve their revenues through premium pricing and competitive differentiation of service offerings, which in turn can fund the necessary expansion of the network. There is a need to improve the web hosting technology in terms of performance, scalability and delivery of new functionalities. Quality of Service (QoS) has become a rallying cry for all forms of communications on the Internet.

Three basic levels of end-to-end QoS can be provided across a heterogeneous network:

- **Best-effort service**---Also known as lack of QoS, best-effort service is basic connectivity with no guarantees.
- **Differentiated service (also called soft QoS)**---Some traffic is treated better than the rest (faster handling, more bandwidth on average, lower loss rate on average). This is a statistical preference, not a hard and fast guarantee.
- **Guaranteed service (also called hard QoS**)---An absolute reservation of network resources for specific traffic.

Deciding which type of service is appropriate to deploy in the network depends on several factors such as-

1. The application or problem the customer is trying to solve.
2. The rate at which customers can realistically upgrade their infrastructures.
3. Last, but not the least, the cost involved in implementing these services.

In general, the cost of implementing and deploying guaranteed service is likely to be more than that for differentiated service.

There is a clear need for relatively simple and coarse methods of providing differentiated classes of service for Internet traffic, to support various types of applications, and specific business requirements. This project report presents, DiffServer, a program that implements two simple, server-side, application-level mechanisms (server-centric and client-centric) to provide different levels of web

service. The rest of this report is organized as follows: Chapter 2 discusses background and previous work. Chapter 3 discusses our implementation, including the DiffServer model, priority determination and the program flow. Chapter 4 examines the performance and presents the results from our experiments of the DiffServer. Finally, Chapter 5 presents our conclusions and future work. Appendix A contains a complete listing of the `options.conf` file. Appendix B touches upon the programming aspects of DiffServer like the source and header files, the data structures used as well as instructions for compiling and launching the software.

Chapter 2

Background and Previous Work

The behavior of HTTP servers is quite unpredictable in cases where there is a large burst of requests. Apart from the fact that under such situations the servers tend to drop requests indiscriminately, the requests to popular pages have a tendency to overwhelm the requests for others, possibly more important pages. Apache [1], one of the most used Web servers, handles incoming requests in a first-come, first-served (FCFS) manner. Furthermore, most implementations of HTTP servers do not perform any discrimination between requests. Therefore a site cannot enforce any kind of priority scheme it may wish to implement. Since the possibility of the backbone providing differentiated services is becoming all the more promising these days, it is imperative that the end-servers also provide differentiated services to circumvent some of the typical problems such as indiscriminate dropping of requests, overwhelming of certain requests [2] etc.

This chapter provides an overview of common approaches for providing differentiated services. Most network servers in Unix are written in the traditional model -- when a process needs something performed by another entity, it spawns a child process by invoking the `fork()` system call and lets the child perform the processing. While this paradigm has served well for many years, there are problems

with `fork()`. `fork()` is expensive since memory contents are copied from the parent to the child and all descriptors are duplicated in the child. Also, some form of interprocess communication like shared memory or message queues are required to exchange information between the parent and the child. T.S. Sankaravadivelu implemented a process-based model (Quibs) [3] to incorporate domain and file based differentiated services in a web server. This module is functionally in between the client requesting a web page and the web server servicing the request. Shared memory was used for the requests queue, process id queue as well as other data structures that were required to be shared by all the processes. Shared memory was chosen since it allows for quick access to the memory and hence fast interprocess communication was possible. Also, multiple processes can read from the shared location concurrently. One process is created for each incoming request, so it is fundamentally a process-per-request model. The *child* process passes the request from the client to the server, returns the response from the server to the client and then exits. If another client is connected to the server, the *child* places its request in a queue and then goes to sleep. This queue is sorted using *Quicksort* based on the priorities assigned to the domain name and file name by the *manager* thread. When the server is "free", the *manager* awakens the processes in the new sorted order. However, the use of shared memory limits the size of the data structures like the requests queue size. More importantly, since there is a huge overhead in creating a process, the performance of the system was poor and even 7-8 requests/sec inundated the Quibs Server [3].

Jussara Almeida et al. [4] have explored priority-based request scheduling by Providing Differentiated Levels of Service at both the user and kernel levels. In the user-level approach they modified a popular web server program, Apache, by adding a scheduler process, which decides the order in which requests should be serviced. In the kernel-level approach they modified both the Apache and Linux kernel by adding two new system calls that provide a mapping from request priorities into process priorities and kept track of which processes are running at which priority level. They found that simple strategies such as controlling the number of processes can improve the response time of high-priority requests notably while preserving the system throughput. They also found that the kernel-level approach tends to penalize low-priority requests less significantly than the user-level approach, while improving the performance of high-priority requests similarly. Though, the performance results of the implementation is impressive, developing these types of systems require a detailed working knowledge of the Apache architecture and the Unix kernel.

Raju Pandey et al. [2] have presented a notion of a quality of service (QoS) model that enables a site to customize how an HTTP server should respond to external requests by setting priorities among page requests and allocating server resources. As part of the QoS model, they have devised a notation, which they call WebQoSL that supports specifications such as-

i. Allocation of a specific and relative amount of server resources to specific page requests.

ii. Availability of groups of pages at all time.

iii. Time-based and link-relation-based allocation of resources.

iv. Specification of guarantees about byte transfer and page request rates.

Differentiated Services at the Network layer

The need for providing differentiated classes of service for Internet traffic, to support various types of applications lead to the formation of the Differentiated Services Working Group [6]. The differentiated services approach to providing quality of service in networks employs a small, well-defined set of building blocks from which a variety of aggregate behaviors may be built. The Differentiated Services architecture [7] [8] offers a framework within which service providers can offer each customer a range of network services, which are differentiated on the basis of performance in addition to pricing tiers used in the past. Customers request a specific performance level on a packet by packet basis, by marking the DS field of each packet with a specific value. This value specifies the Per-hop Behavior (PHB) to be allotted to the packet within the provider's network. Typically, the customer and provider negotiate a profile (policing profile) describing the rate at which traffic can be submitted at each service level. Packets submitted in excess of this profile may not be allotted the service level requested. A salient feature of differentiated services is its scalability, which allows it to be deployed in very large networks.

Our DiffServer implementation runs at the application space level and hence is very easy to setup and run. It eliminates the need to deploy and provision expensive network devices like routers, which are very challenging tasks. Also, unlike Raju Pandey's [2] and Jussara Almeida's [4] approaches no modifications have been made to either the client (Netscape Navigator, Internet Explorer) or the Apache Web Server source code. Chapter 3 deals in detail with the model and program flow of DiffServer.

Chapter 3

Design and Implementation of DiffServer

We implemented a user-level program (DiffServer), which runs in application space and acts as a module around the web server. Our implementation included both a server-centric as well as a client-centric differentiated services scheme.

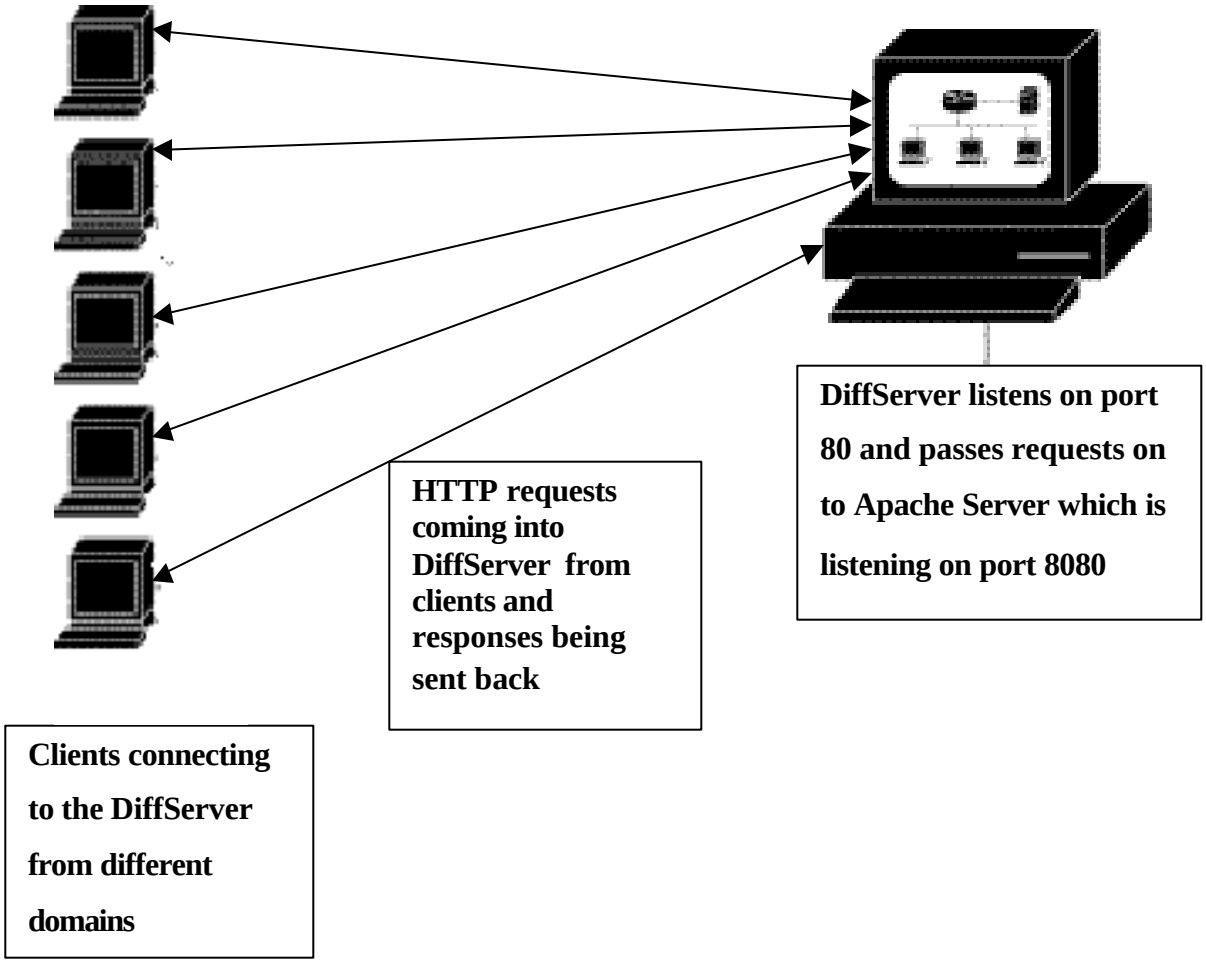


Figure 1: Logical Model of the Diffserver

A priority scheme based on the hostname and domain of the client (client-centric) is enforced on the incoming burst of requests. For each set of incoming requests, a file name based priority (server-centric) scheme is also enforced. Both these priority orders are "hard coded" into a configuration file named `options.conf`.

Figure 1 shows the Logical diagram of the DiffServer architecture. Essentially the program involves, three kinds of threads. A *parent* thread, *Child* threads and a *Scheduler* thread. The parent thread listens to port 80 for incoming requests. The parent first creates a *Scheduler* thread, which later picks a Child Thread from the pool to handle an incoming request. The parent thread assigns priorities to the incoming requests according to the criteria listed in the `options.conf` file and then the requests are inserted into a queue in the ascending order of Priorities. The requests are assigned priorities based on the document they are requesting as well their hostname and domain. The scheduling algorithm is a non-preemptive scheme, since at the user level we cannot interrupt a running process and block it in order to allow a new process to run.

3.1 Design of the DiffServer Architecture

Figure 2 shows the conceptual model of the DiffServer. The DiffServer module listens at port 80 for requests coming in from the clients. When a request is received, it is assigned a priority based on its domain (client-centric approach) as

well the filename requested (server-centric approach). The position of the request in the queue is determined by its priority. The Apache web server listens on port 8080 for the sorted requests that are forwarded by the DiffServer module. Apache returns the page requested to DiffServer, which in turn returns the response to the client.

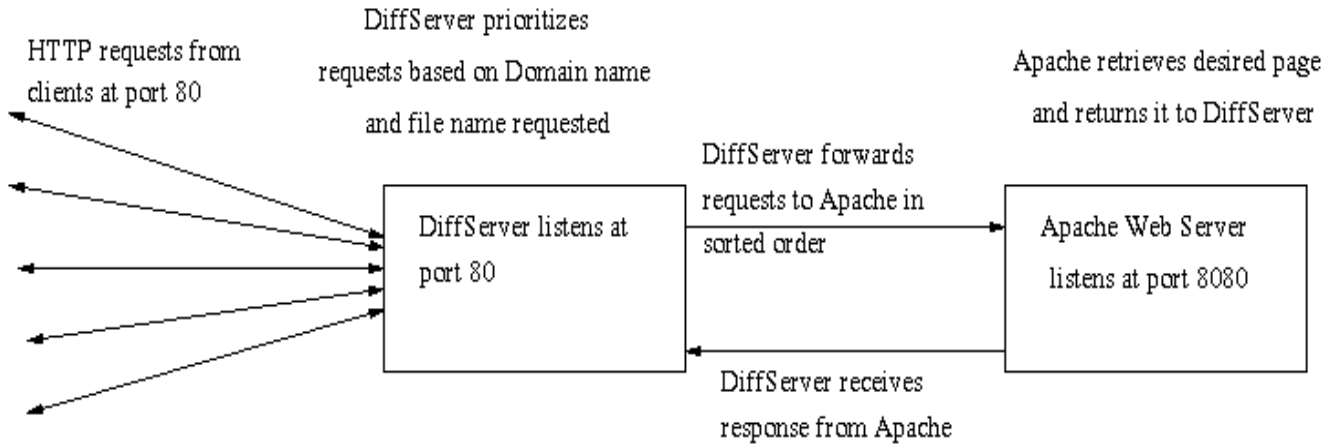


Figure 2 : Conceptual Model of the DiffServer

3.2 Advantages of using Threads over a Process based Model

Threads [11] are often called lightweight processes and while this term is somewhat of an over simplification, it is a good starting point. Threads have many advantages over processes. It takes less time to create a new thread than a process, because the newly created thread uses the current process address space. Terminating a thread takes comparatively less time and also context switching between two threads within the same process is less time consuming, partly because the newly created thread uses the current process address space. Finally the communication overheads between the threads of one process are simple. Due to aforementioned reasons we decided to go in

for a thread based approach to our solution architecture. We used the *pthread* interface to Linux kernel threads (POSIX threads) [9] [10] [11] to create and manage a multi-threaded architecture. The POSIX threads Application Progam Interface (API) provides a set of interfaces and semantics for creating and controlling threads.

3.3 Purpose of the Options.conf file

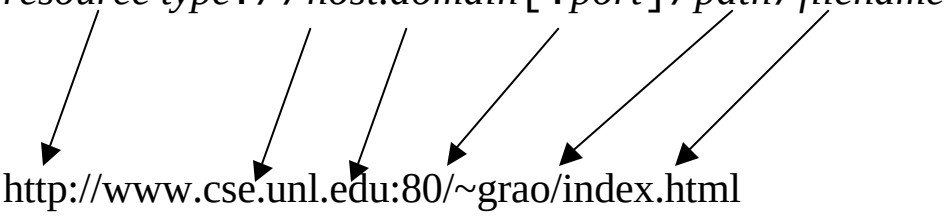
The DiffServer at startup parses the `options.conf` file. This functionality allows the administrator to manipulate the values of certain QoS parameters that govern the performance of the DiffServer and eliminates the need for recompilation of the source code. Another viable alternative to the `options.conf` file would have been to allow the user to specify a *command-line* option or to specify these values as *environment variables*. However, we decided to model the DiffServer along the lines of the Apache Web Server, which has a configuration file (`httpd.conf`). `options.conf` primarily contains the list of hostnames and domains along with the list of directories and files that should be provided differentiated services. It also contains parameters that can be used to tune the performance of the DiffServer. The administrator can change the Port on which the DiffServer listens, or vary the MAX_NUM_HANDLER_THREADS and NUM_HANDLER_THREADS parameters within acceptable limits. The LISTENQ and the BUFF_SIZE can also be varied. The LISTENQ defines the maximum length of the queue that the number of pending

connections may grow to. The BUFF_SIZE specifies the size of the *buffer* that is used for reading in data from the Apache server, hence a larger BUFF_SIZE leads to faster screen refresh rates. In order to provide the administrator the flexibility of either providing higher domain based priority than file based priority or vice versa, the DOMAIN_NAME_FACTOR and FILE_NAME_FACTOR parameters can also be changed. Appendix A contains a complete listing of the `options.conf` file.

3.4 Parsing the URL

In the forthcoming discussion, since we will be referring to the different parts of a URL to a great extent, let us understand what the different parts of a URL are. URL stands for "Uniform Resource Locator". It is a draft standard [12] for specifying an object on the Internet, such as a file or newsgroup.

resource type`://`*host.domain*`[:`*port*`]/`*path*`/`*filename*



The first part of the URL, before the colon, specifies the access method or protocol. The access method can be of various types such as ftp, file, gopher etc but we will confine our discussion to http. The part of the URL after the colon is interpreted specific to the access method. In general, two slashes after the colon indicate a

machine name (machine:port is also valid). For an HTTP or FTP URL the next part is a pathname which is usually related to the pathname of a file on the server. The last part refers file that is to be accessed.

3.5 Priority Determination

The system administrator assigns priorities to the domains and files in the `options.conf` file. The priorities are assigned on a scale of 1 (Lowest) to 10 (Highest). They may also specify any subset of the *hostname.domain* entry in the URL, for example, entries in the `options.conf` file could be of the form :

andes.unl.edu	10
unl.edu	8.5
edu	5.1
cnn.com	6

Hence, the administrators can provide differentiated services to a very fine granularity since they are not just restricted to a domain name when specifying priorities but can also specify a host or machine name within that domain. Priorities are also assigned to filenames and directories of the requested documents. For example:

/images/background.gif	9.5
/images/list.html	7

/images/	4
/index.html	8
/projects/	3

In the above example, the file *background.gif* and *list.html* in the */images* directory are treated differently (assigned priorities 9.5 and 7 respectively) than any other files under */images* (assigned priority 4). If a priority value is not entered next to the domain or file entry, it is assigned the Lowest Priority of 1. On receiving a request, the main program extracts the hostname, domain, directory and filename from the URL. It then compares the extracted hostname and domain combination for a sub-string match to its domains list and assigns the request a *domain based priority*. If no match is found, the request is assigned the LOWEST_PRIORITY of 1. While entering information in the `options.conf` file, it is important to take the structure of the *hostname.domain* part of the URL into account, since the comparison algorithm quits when it has the first match rather than the best match. In the above example, the administrator wants to treat requests from the host *andes.unl.edu* differently from the other requests coming from the *unl.edu* domain. So, the hostname *andes.unl.edu*, should occur prior to *unl.edu* in the `options.conf` file listing. Otherwise, any request coming from *andes.unl.edu* will be assigned the same priority as requests coming in from the *unl.edu*., based on the best-match approach. Besides a *domain based priority*, the request is also assigned a *file based priority*. This is accomplished by comparing the directory and filename from the incoming URL to the filename

priority list. As in the previous case, it is important to take into account the structure of the directory tree while assigning file based priorities. So, any listings of sub-directories and their files should occur before their parent directories in the `options.conf` file. The final priority of the request is determined by its *domain based priority* and *file based priority*. The administrator specifies a DOMAIN_NAME_FACTOR and FILE_NAME_FACTOR, which are multiplied by the domain based priority, and the file based priority respectively to compute the *Overall Priority* of the request. This *Overall priority* determines the request's position in the queue.

3.6 Program Flow

The parent initially creates the *Scheduler* thread and also *Child* threads to handle the requests. The number of Child threads created is determined by the NUM_HANDLER_THREADS parameter in `options.conf`. The parent listens on port 80 for any incoming requests. On receiving a request, the parent determines its Priority based on the hostname and domain and also the filename in the URL and inserts the request in the requests queue. Each node in the requests queue also contains the file handle of the incoming request. The *Scheduler* constantly checks the queue for any new requests inserted and whether there is a *Child* Thread available to handle it. It selects the first "free" *Child* and signals it to handle the request. The

signaled *Child* Thread, retrieves the first request from the queue and connects to the Apache Web server and passes the request on to it. Apache retrieves the desired page and gives it to the *Child* Thread, which in turn returns the response to the client. The Child Thread then begins to wait for the next request on the condition `child_wake_up()`. The *Scheduler* also checks to see if the number of requests in the queue is greater than the HIGH_REQUESTS_MARK, if so, it adds another *Child* Thread to the pool of Request Handlers. This addition of Request Handling Threads is only carried on till the number of threads in the pool reaches the MAX_NUM_HANDLER_THREADS value.

The important aspects of our scheduling policy is that it caters to increase in the hit rate by increasing the number of Handler threads only until the threshold level (MAX_NUM_HANDLER_THREADS) is reached. Though, this limits the total number of clients who can simultaneously connect to the DiffServer, it is intended as a brake to keep a large burst of incoming requests from taking the system down. Also, all incoming requests are tunneled through the sorted queue of requests and so the administrator can manipulate the order in which the requests are given to the Apache server. Figures 3 and 4 below give a diagrammatic representation of the roles played by the main thread, *Child* Thread and the *Scheduler* Thread.

Figure 3 : Program Flow for Main Thread and Scheduler Thread

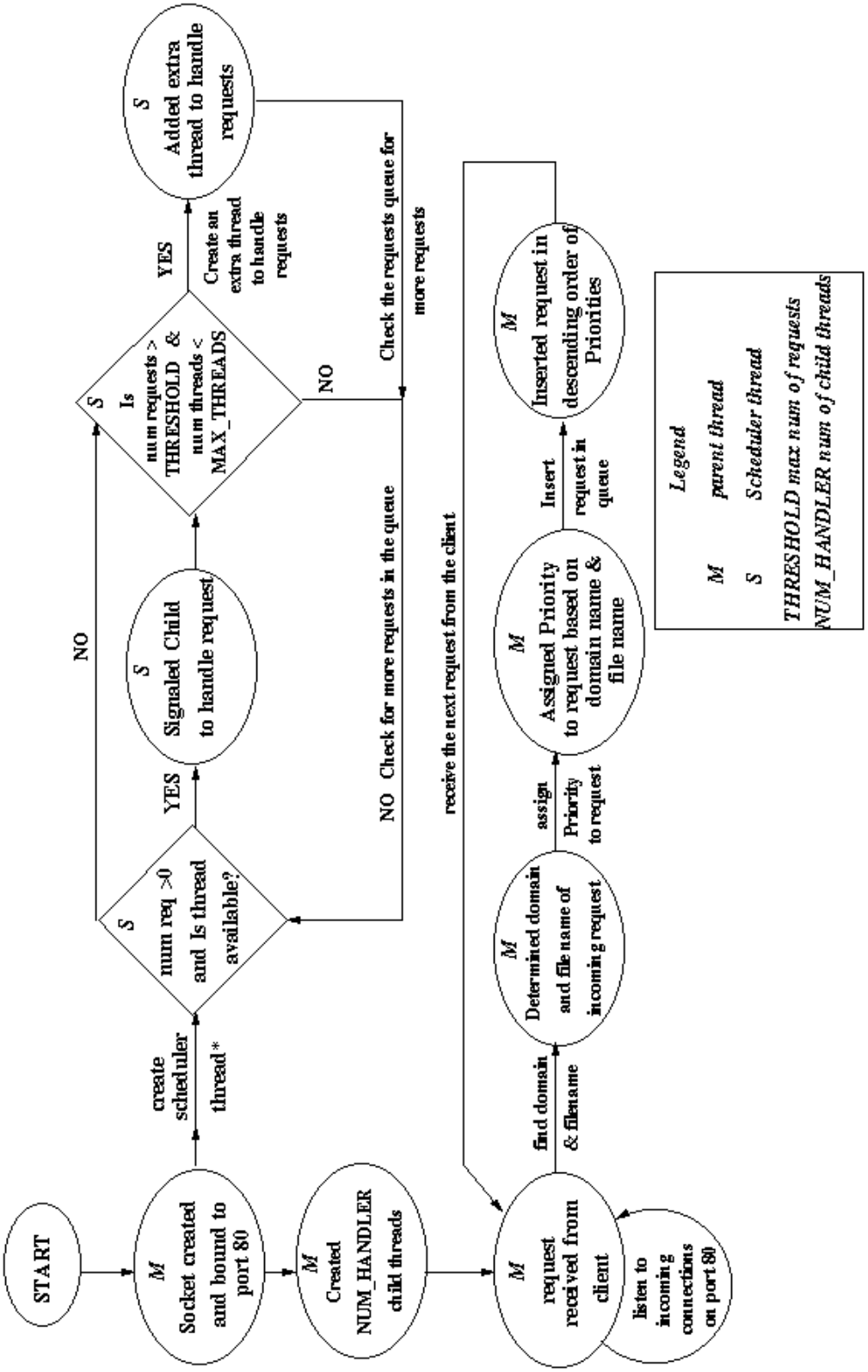
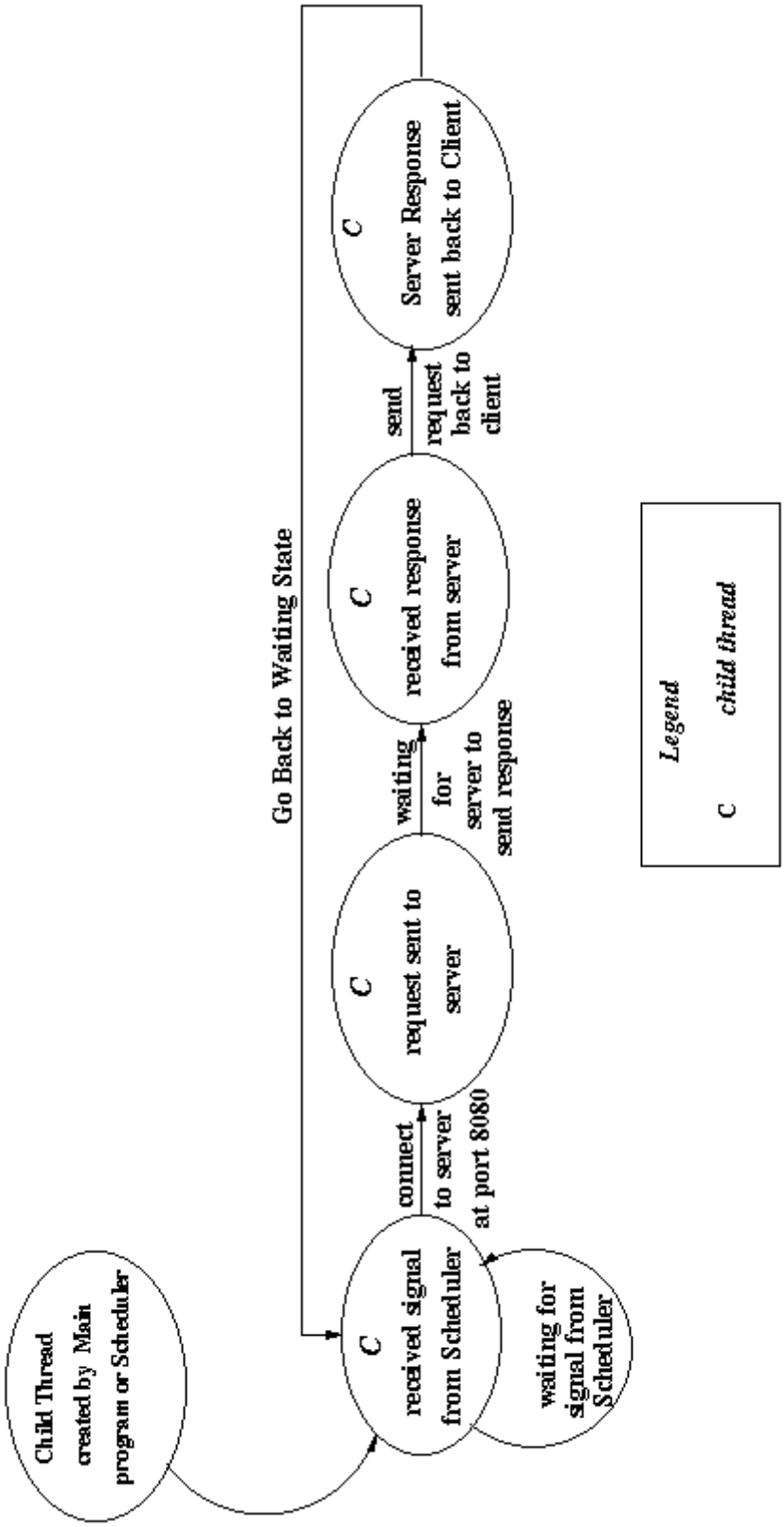


Figure 4 : Program Flow for Child Threads



Chapter 4

Evaluation

4.1 Experimental Setup

To generate the WWW workload, we used *httperf* [13], a configurable load generator from Hewlett-Packard. It provides a flexible facility for generating various HTTP workloads and for measuring server performance. The three distinguishing characteristics of *httperf* are its robustness, which includes the ability to generate and sustain server overload, support for the HTTP/1.1 protocol, and its extensibility to new workload generators and performance measurements. *httperf* also provides the user the ability to set a timeout, in the event that the web server has not responded within a reasonable amount of time, *httperf* will abort the connection and retry. *httperf* allows the user to select the connection rate manually. This provides the ability to examine the effect of increasing load on the web server in a controlled fashion. The development and experiments for this project was carried out in the Advanced Networking and Distributed Experimental Systems (ANDES) Lab in the Department of Computer Science and Engineering (CSE) at the University of Nebraska-Lincoln (UNL). The network used for our tests was a 100 Mb shared Ethernet with a network file system. Each of the client nodes that ran *httperf* was an Intel Pentium II 266 MHz or AMD 400 MHz with 128 MB of RAM running Red Hat Linux 5.2 with version

2.2.10 of the Linux kernel. The DiffServer and the Apache web server were run on a Pentium Celeron 333 MHz with 256 MB RAM.

4.2 Performance Measurement

The performance metric used to evaluate the effectiveness of our approaches was the average Connection time (elapsed time) of servicing a request, and the Response Time. The Connection Time gives the lifetime statistics for successful sessions. The lifetime of a session is the time between a TCP connection was initiated and the time the session was closed. Another important metric to take into account is the server Connection Rate which is expressed as connections per second. This is an indication of how fast the server can establish a connection and start communicating with the clients. The Response Time gives information on how long it took for the server to respond and how long it took to receive the reply.

We initiated sessions of 1500 connections each with both HTML (static) as well as CGI (dynamic) content. The average HTML file sizes were 6401 bytes whereas the average CGI file sizes were 2204 bytes for the experiments. Testing with dynamic files is necessary since more and more dynamic content is appearing on the Web. In order to evaluate the overhead of the search and insertion of the request in the sorted linked list, we ran test cases for the DiffServer's scheduling algorithm both as a FIFO list as well as a sorted priority list. Figures 5 and 6 show the Comparisons of the Connection Time and the Response Time for HTML requests. Figures 7 and 8 depict

the same for the *cgi-bin* case. A `/var/log/httpd/access_log` file from the ANDES server provided us with a trace that was used by *httperf* to perform our experiments for different rates of requests/sec. In order to assign a domain based priority to the incoming requests, we simulated hits from different domains by implementing a function that generates a random number which is mapped to a domain in the domaininfo array (see Appendix B). A domain based priority was assigned based on this pseudo domain to the incoming request.

We notice that the Connection time and Response times are much higher in the CGI case, since a CGI program runs as a separate process in the server machine every time a CGI document is requested and therefore is very costly. We compared the average waiting time for a request in the queue if it is serviced in the traditional FIFO way (no priority assigned to requests) against the average waiting time after it is assigned a priority based on the domain name and filename. The average waiting time for a high priority request in the queue was significantly decreased (76% and 88% for file based and domain based priority respectively). The average waiting time for low priority requests increases by 115% (file based priority) and 60% (domain based priority) as compared to the case where no differentiated QoS policy is used. The graphs below (see Figures 5, 6 and 7) show the results obtained.

Proxy Mode

The DiffServer can also be run in a proxy mode. In this mode the DiffServer runs on a different machine than the web server. The Administrator specifies the IP address of the machine on which the Apache web server is being run at startup. Hence, the administrator can provide differentiated services on virtually any web server on the Internet. The administrator can either specify the IP address of the web server on the command line as an additional parameter to the DiffServer or include it in the *options.conf* file. The Connection times as well as the Response times in the proxy mode were significantly higher than in the non-proxy case. Further investigation is necessary to quantify the benefits of using the DiffServer in a networked environment with one or more ‘back-end’ web servers.

Table 1 : Performance of HTML connections

1500 HTML Connections from ANDES server log, Average File Size = 6401 B

	Requests/sec	Connection Rate	Connection Time(ms)	Response Time(ms)
	80	80	3	2
	100	100	3.1	2.2
Apache	120	119.9	3.2	2.2
	140	139.8	3.5	2.5
	160	159.6	3.9	2.9
	180	179.4	4.3	3.3
	Requests/sec	Connection Rate	Connection Time(ms)	Response Time(ms)
	80	80	6.4	5.4
	100	99.5	8	7
FIFO Module	120	119.5	10.8	9.8
	140	139.2	12.8	11.7
	160	157.4	18	17
	180	178.5	39.3	38.3
	Requests/sec	Connection Rate	Connection Time(ms)	Response Time(ms)
	80	79.9	9.8	8.8
	100	99.5	10.2	9.3
DiffServer	120	117.8	11.7	10.6
	140	137.1	14.9	13.9
	160	158.7	18.2	17.1
	180	179.4	39.4	38.4

Table 2 : Performance of *cgi-bin* connections

1500 CGI-BIN Connections from ANDES server log, Average File Size = 2204 B

	Requests/sec	Connection Rate	Connection Time(ms)	Response Time(ms)
	80	54.9	2752.1	2637
	100	39.8	4793.9	3760
Apache	120	37.2	5538.4	3865
	140	38.1	6601.1	4253
	160	38	6768	4325
	Requests/sec	Connection Rate	Connection Time(ms)	Response Time(ms)
	80	40.8	5772.8	5772.4
	100	40.2	8541	8540.6
FIFO Module	120	38.6	9619.8	9619.4
	140	36.8	10628.3	10627.9
	160	38.9	10860.2	10859.8
	Requests/sec	Connection Rate	Connection Time(ms)	Response Time(ms)
	80	39.2	7392.6	7392.2
DiffServer	100	34.9	9213.4	9213.1
	120	37.6	10323.3	10322.9
	140	36.5	11024.6	11024.3
	160	37.9	11306.3	11306

Table 3 : Average Waiting Time for File Based Priority

1500 Connections from ANDES Server Log♠

	No Priority	File based Priority	
Requests/sec	Average Delay(ms)	High Priority requests (ms)	Low Priority Requests (ms)
80	2.09	1.98	5.1
100	4.38	2.06	5.87
120	10.61	3.89	13.83
140	13.06	4.86	16.71
160	22.92	5.92	50.66
180	74.1	17.71	155.82

Table 4 : Average Waiting Time for Domain Based Priority

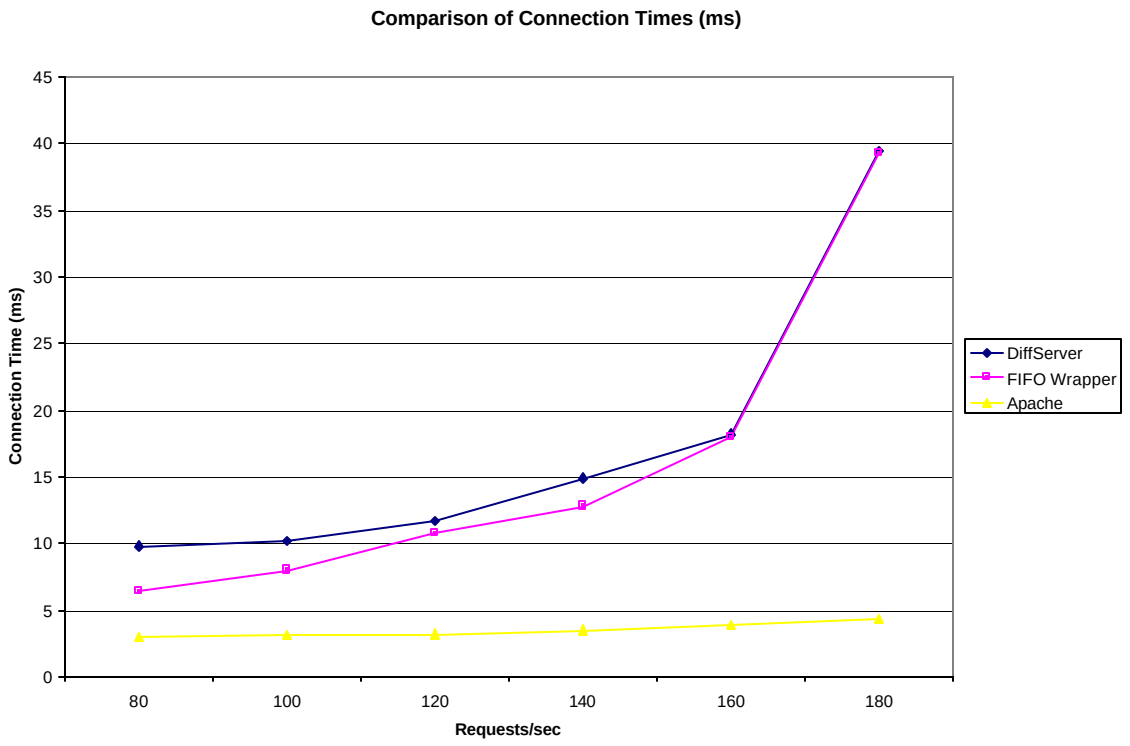
1500 Connections from ANDES Server Log♣

	No Priority	Domain based Priority	
Requests/sec	Average Delay(ms)	High Priority requests (ms)	Low Priority Requests (ms)
80	3.74	2.49	3.93
100	3.43	2.66	4.66
120	9.23	4.19	15.55
140	14.62	4.61	20.3
160	40.3	6.99	50.18
180	71.99	8.31	115.88

♠ : For Priority case, 750 connections with high priority and 750 connections with low priority requests were used

♣ : For Priority case, 750 connections with high priority and 750 connections with low priority requests were used

Figure 5 : Connection Time and Response Time for HTML Connections



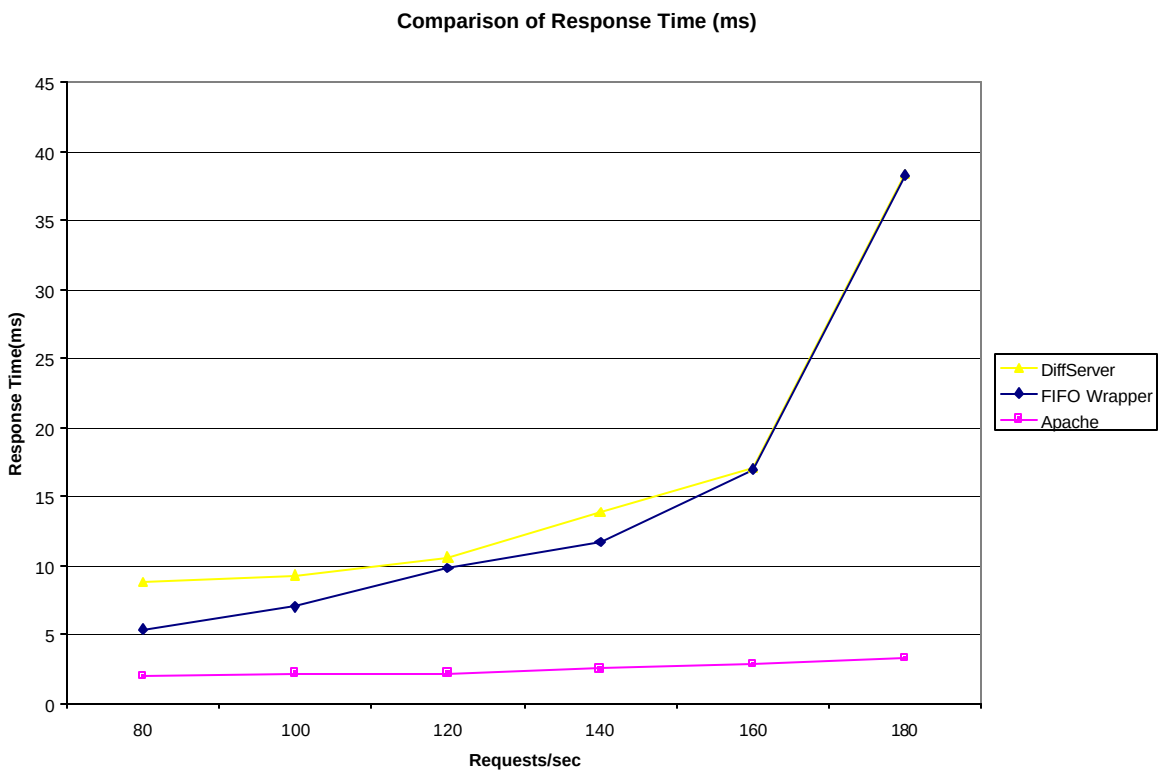
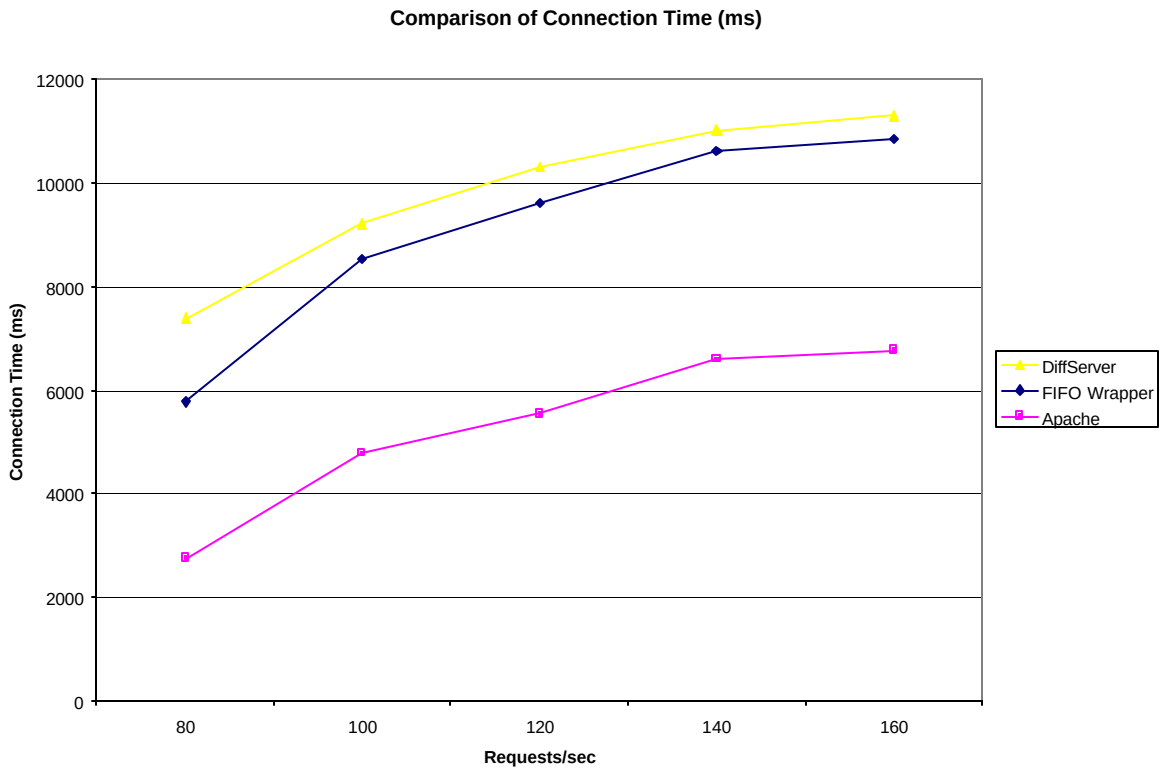


Figure 6 : Connection Time and Response Time for *cgi-bin* Connections



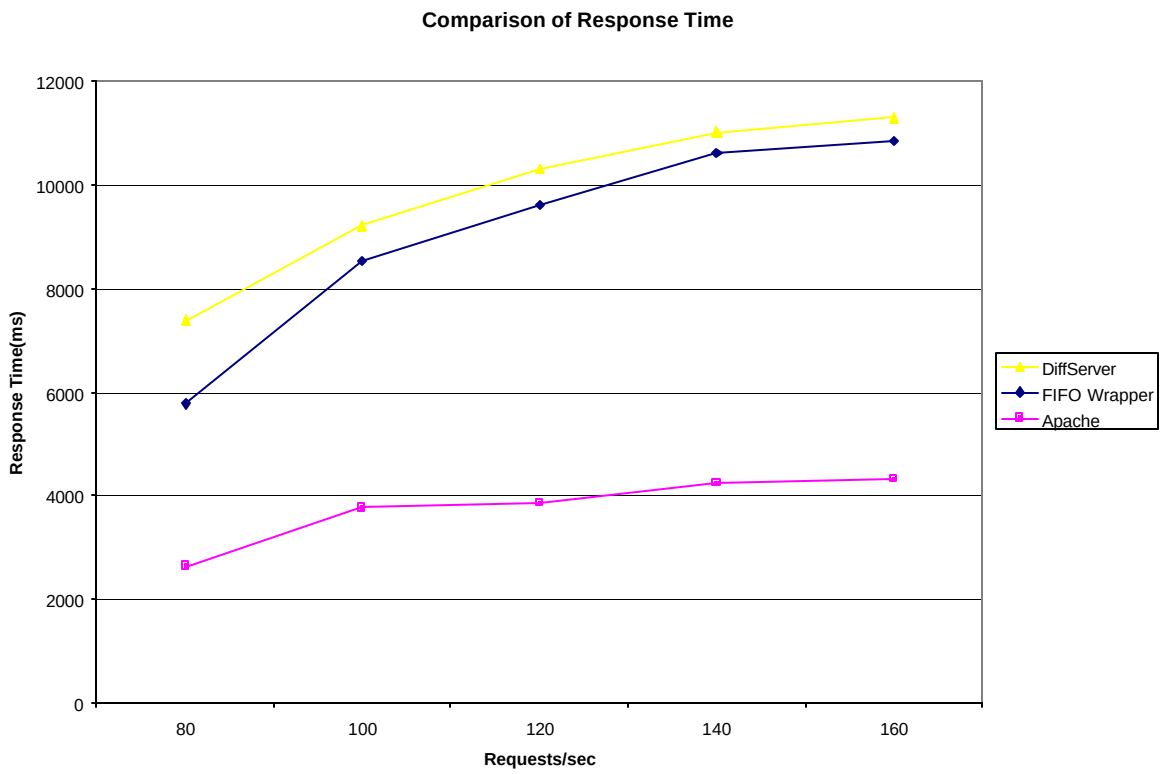
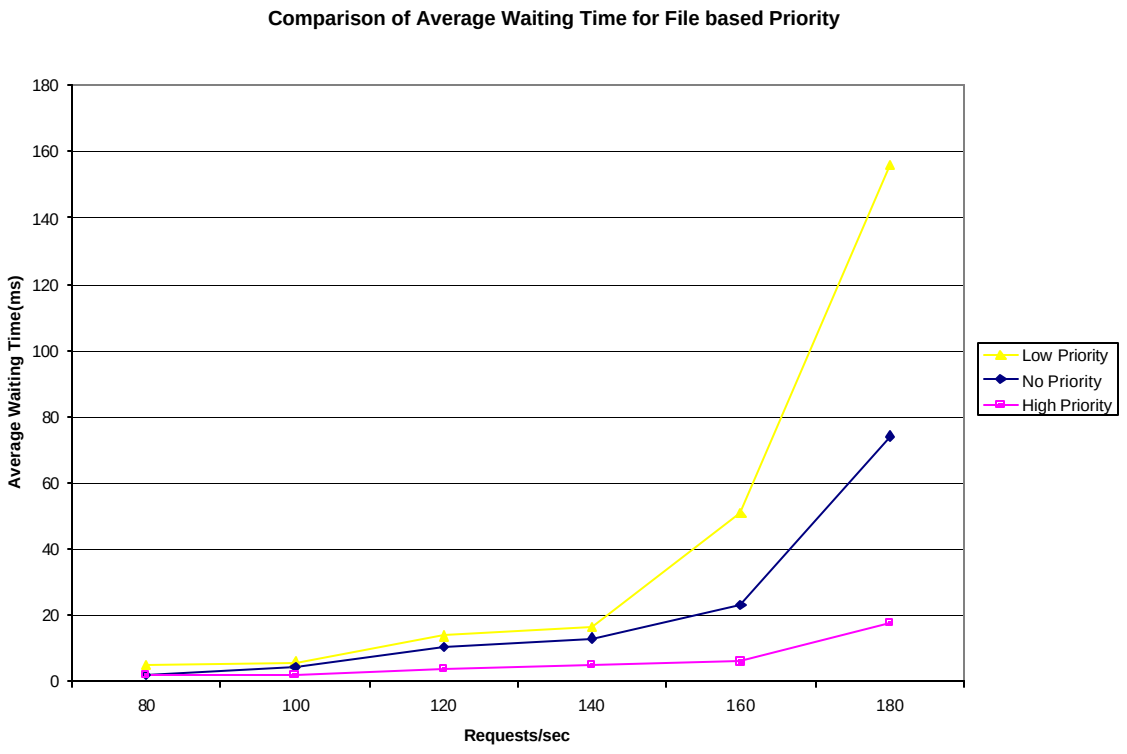
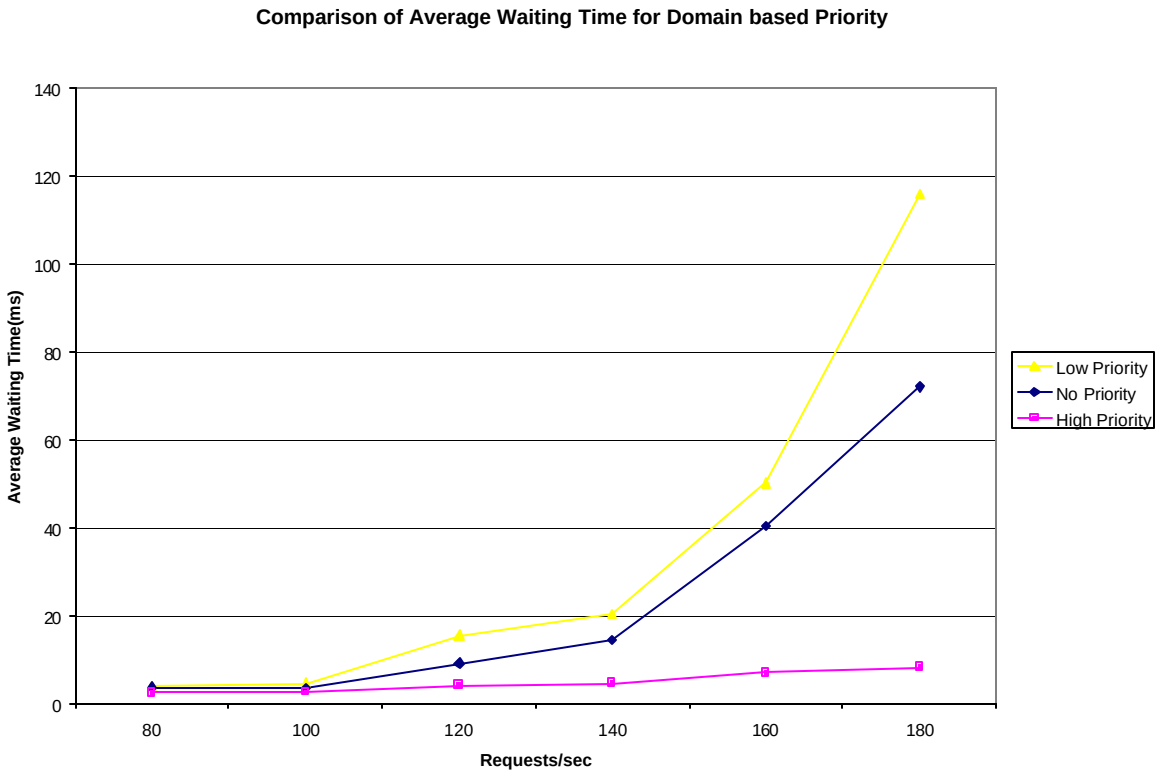


Figure 7 : Average Waiting Time for File and Domain Based Priority





Chapter 5

Conclusions and Future Work

The DiffServer architecture presented in this report is a multi-threaded, application-space and very scalable module. DiffServer is very easy to setup and run and also has user configurable QoS parameters (Refer Appendix A). This project implements an original approach to provide differentiated services. The results of the experiments show that there is not much overhead due to the addition of this additional layer of abstraction between the client and the Apache web server under light load conditions. Also, the average waiting time for high priority requests decreases significantly after they are assigned priorities as compared to a FIFO approach.

This DiffServer implementation has opened vistas for further enhancements. One area of research would be to explore the possibility of a weighted priority queue to ensure that all requests are handled and no starvation occurs.

In this report, we limit our investigation to Web server systems only, without addressing the different services issues in the Internetworking infrastructure. In other words, we assume that the order in which the request packets arrive at the server and in which the packets are transmitted over the Internet are completely out of control of the Web server. Obviously, a complete solution for this type of networking quality of

service would require a combination of networking differentiated services [7] [8] and differentiated services supported by the DiffServer. Web servers now support several means to create dynamic content (e.g. CGI, FastCGI, vendor-dependent Web server APIs and Java servlets). These methods involve complex computation on the Web server and hence slow down its Connection time and Response time. Hence, in cases like this where Web server processing per request becomes more time-consuming, our differentiated services approach can be complemented with a Web Clustering approach [14] [15] [16], which could help alleviate the problem.

References:

[1] The Apache Server Foundation. URL:http://www.apache.org

[2] R. Pandey, J.F. Barnes and R. Olsson, "Supporting Quality of Service in HTTP servers", *Proc., Seventeenth Annual SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Puerto Vallarta, Mexico, June 1998.

[3] T.S. Sankaravadivelu, "Quibs: Incorporating QoS into Web Suring", *A Project Report* for the Advanced Computer Networks course, Dept. of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, December 1999.

[4] J. Almedia, M. Dabu, A. Manikutty and P. Cao, "Providing differentiated levels of service in web content hosting", *1998 Workshop on Internet Server Performance*, Madison, Wisconsin, June 1998.

[5] X. Chen and P. Mohapatra, "Providing Differentiated Services from an Internet Server", *1998 Workshop on Internet Server Performance*, Madison, Wisconsin, June 1998.

[6] Differentiated Services Working Group (diffserv) URL:http://www.ietf.org/html.charters/diffserv-charter.html

[7] Y. Bernet, J. Binder, S. Blake, M. Carlson, B. Carpenter, S. Keshav, E. Davies, B. Ohlman, D. Verma, Z. Wang and W. Weiss, "A Framework for Differentiated Services", IETF Internet Draft, February 1999. URL:http://search.ietf.org/internet-drafts/draft-ietf-diffserv-model-02.txt

[8] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss, "An Architecture for Differentiated Services", *RFC 2475*, Dec 1998.
URL: http://www.hypermail.org/rfcs/rfc2475.html

[9] "Multi-Threaded Programming With POSIX Threads".
URL:http://www.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html

[10] T. Wagner and D. Towsley, "Getting Started With POSIX Threads". URL:http://www2.cs.ucr.edu/~sshah/pthreads/tutorial.html

[11] B. Lewis and D. Berg, *Multithreaded Programming with Pthreads*, New Jersey, Prentice Hall, December 1997.

[12] T. Berners-Lee, R. Fielding and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", *RFC 2396*, August 1998.

[13] D. Mosberger and T. Jin, "httperf - A tool for measuring web server performance". URL:http://www.hpl.hp.com/personal/David_Mosberger/httperf.html

[14] X. Gan, "A Prototype of a Web Server Clustering System", *M.S. Project Report*, Dept. of Computer Science and Engineering, University of Nebraska - Lincoln, April 1999.

[15] X. Gan, T. Schroeder, S. Goddard and B. Ramamurthy, "LSMAC vs. LSNAT: Scalable Cluster-based Web Servers", *Cluster Computing Journal 2000* (to appear).

[16] T. Schroeder, S. Goddard, B. Ramamurthy, "Scalable Web Server Clustering Technologies", *IEEE Network 2000* (to appear)

[17] L Eggert and J. Heidemann, "Application-Level Differentiated Services for Web Servers", *World Wide Web Journal*, vol. 3, 1999.

[18] R. Stevens, *Unix Network Programming*, vol. I & II, New Jersey, Prentice Hall, December 1998.

[19] Beej's Guide to Network Programming. URL:http://www.ecst.csuchico.edu/~beej/guide/net/

[20] V. Metcalfe and A. Gierth, "Programming UNIX Sockets in C – Frequently Asked Questions", April 1997. URL:http://www.kmutt.ac.th/knowledge/programming/c/unix_socket/faq/unix-socket-faq.html

[21] A.D. Marshall, "Programming in C, UNIX System Calls and Subroutines using C", March 1999. URL:http://www.cm.cf.ac.uk/Dave/C/CE.html

[22] D. Butenhof, *Programming With Posix Threads*, Addison-Wesley Professional Computing Series, May 1997.

Appendix A (options.conf listing)

```
#options.conf file
# Author : Gautam Rao
# Copyright 2000 , All Rights Reserved
# Configuration file for the diffserver
# The Port on which the DiffServer is going to listen, is set to 2026 by default. This can be
# changed to port 80 or any other port that is not currently being used by other services
DIFFSERVER_PORT 2026

# The Port on which the apache web server is listening is set to 80 by  default.
SERVER_PORT 80

# The IP address for the server on which the Web Server is running.. Default is andes.unl.edu
PROXY 129.93.33.141

# The NUM_HANDLER_THREADS & MAX_NUM_HANDLER_THREADS defines the
# number of threads when the DiffServer starts up as well as the total number of  threads that the
# system will create during heavy loads
NUM_HANDLER_THREADS 10
MAX_NUM_HANDLER_THREADS 40

# The HIGH_REQUESTS_MARK defines the number of requests in the queue after which the
# scheduler decides to add another thread to handle the queue of requests
HIGH_REQUESTS_MARK 80

# The DOMAIN_NAME_FACTOR and FILE_NAME_FACTOR, decides whether the domain
# name or the file name receives higher priority before insertion into the list. To  guarantee a
# higher file based priority or domain based priority order of processing,  their ratio should be
# large
DOMAIN_NAME_FACTOR 100
```

```
FILE_NAME_FACTOR 1

# The BUFF_SIZE parameter is used to specify the buffer size for the data to be transferred from
# the server to the client. Larger values are recommended for refreshing the screen faster
BUFF_SIZE 10000

# The LISTENQ sets the size of the queue for the LISTEN command. The LISTENQ defines the
#  maximum length of the queue that the number of pending connections may grow to
LISTENQ 200

# Enter the domain names along with their priorities. If no priority is entered, the domain is
# assigned the lowest priority. If you do not enter a priority for the domain, hit the enter button as
# soon as you type in the domain name. Also, since the algorithm is "First Match"  rather than
# "Best  Match", it is the administrator's responsibility to  ensure that the order of occurrence is
# correct. So, if you wish to  assign different priorities to the host "andes.unl.edu" than the
# "unl.edu"   domain  make sure that it appears *before* the "edu" domain in the list

# Rate the priorities on a scale of 1 to 10, where 10= HIGHEST 1 = LOWEST. Real Numbers
# may also be used e.g. : 3.3. If no priority is entered, the domain is assigned the LOWEST
# priority

BEGIN
andes.unl.edu 10
edu 5.9
com 1.9
cnn.com
msnbc.com
apache.org
whitehouse.gov
timesofindia.com
rediff.com
```

```
END

# Enter the file names along with their priorities. If no priority is   entered, the file is assigned the
# lowest priority. If you do not  enter a priority for the file, hit the enter button as soon as you
# type in the file name  As in the previous case, it is important to take into account the structure
# of the directory tree while assigning file based priorities. So, priorities assigned to a
# subdirectory should # occur *before* its parent directory.

# Rate the priorities on a scale of 1 to 10, where 10= HIGHEST  1 = LOWEST. Real Numbers
# may also be used e.g. : 3.3. If no priority  is entered, the domain is assigned the LOWEST
# priority
BEGIN
/images/andes.jpg 9
/frame.html 10
/list.html 10
/list2.html 10
/home.html 10
/images/banner.jpg 5
/images/red_home.gif 5
/images/background1.jpg 5
/images/machines1.jpg 5
/images/projects1.jpg 5
/search.html 10
/feedback.html 9.7
/linux.html 9.9
/install-linux.html 10
/lab.html 9.9
/images/lab-layout.gif 5
/images/ 5
/projects/
/ 10
END
```

Appendix B (Data Structure Overview)

Appendix B touches upon the technical aspects of DiffServer such as the compilation, program usage , files and the data structures used.

DiffServer is a multi-threaded program and uses the pthread library [18] [19]. There are 2 source files, diffserver.c and parser.c, 4 header files, list.h, params.h, semaphore.h and misc.h. `options.conf` is a flat file that contains the priorities assigned to the various domains and filenames as well as a few QoS parameters

To compile the program just type

make

To start the DiffServer type:

diffserver <portname> <IP address>

The *<portname>* and <IP address> are optional, if not specified the program picks up the default DIFFSEVER_PORT and PROXY defined in `options.conf`.

list.h contains the data structures to hold the incoming request , along with its domain and file based priority and the file descriptor.

```
struct request
{
   float dom_priority;    //domain based priority of the request
   float file_priority;  //file based priority of the request
   float priority;     // Overall priority of the incoming requests
   char *reqbuff;       // the buffer[] containing the request
```

```
    int fd;            //file descriptor associated with the request
    char dom[DOMAIN_SIZE]; //the domain of the incoming request
    struct request *next;
};
```

list.h also contains the data structures that are used to read the filenames and host and domains along with the priorities assigned to them from the `options.conf` file. These structures are called fileinfo and domaininfo respectively.

```
struct fileinfo {
        char filename[FILE_NAME_SIZE];
        float priority;
} **filep;
struct domaininfo {
        char domain[25];
        float priority;
} **domainp;
```

parser.c contains a function Parse_File() that is used to parse the `options.conf` file and read information from it. In the `options.conf` file there is *no upperbound* on the number of entries that can be entered for either the domains and their priorities or the filenames and their priorities. The only stipulation is that in either case the list should start with the keyword “BEGIN” and the last entry should be followed by the keyword “END”. The keywords are case in-sensitive. This virtual limitless capacity

for the number of domains and filenames was achieved through the use of the *realloc()* in the code. So, if the maximum *malloc*ed*()* limit is reached while the data is being read in from the `options.conf` file, more memory *realloc*ed*()* in increments of 10.

Following is the code segment that performs the operation

```
//if the maximum malloced memory is reached, time to realloc some memory
        if(count==maxlimit)
        {
            maxlimit +=increment_factor;
            filep = (struct fileinfo**) realloc(filep, maxlimit * sizeof(struct fileinfo*));
            for(i =count; i<maxlimit; i++)
            {
                filep[i] = (struct fileinfo*)malloc(sizeof(struct fileinfo));
            }
        }
```

The disadvantage to having a large list of files and domains is that it affects the performance of the DiffServer, since the parent thread has to do a lot more comparisons before determining the *domain based priority* and *file based priority* of the URL. Also, we defined the *domaininfo* and *fileinfo* structures as arrays rather than linked lists since traversal through linked lists is much slower than arrays. Hence, the usage of arrays will contribute to a faster search at run time.

params.h contains the structure *thread_params{}* that s used to hold the *thread_id* for each thread created and also its availability. This parameter is passed to the *Child* thread through the *pthread_create()* command. Using the *thread_id* the *Child* thread can wait on the condition *pthread_cond_wait(&child_wake_up[thread_id], &child_lock)* for the *Scheduler* to *signal* it to handle the next request.

```
//  thread parameters structure.
struct thread_params {
   int thread_id;                          // 'id' of thread
   int thread_available;            // thread availablity
   struct thread_params *next;
};
```