Development and performance evaluation of multi-threaded and thread-pool based web servers

Badrish Chandramouli
Department of computer science
Duke University, Durham NC 27708
badrish@cs.duke.edu

***Abstract*—A web server is a software that allows clients to connect to it using a well-defined protocol (HTTP) and retrieve web content. The client is typically a browser such as Internet Explorer or Mozilla. There are various design alternatives for web servers, some of which include multi-threaded, thread-pool based, and event-driven web servers. This paper describes the development of two flavors of web servers – a multi-threaded web server and a thread-pool based web server. A detailed experimental performance analysis of the multi-threaded web server is presented. The thread-pool based web server is also evaluated comparatively. The web server is also compared to Apache, a commercial-grade web server. Comparison of HTTP 1.0 and HTTP 1.1 is also discussed.**

***Index Terms*—web, server, HTTP, performance, Internet**

I. INTRODUCTION

THIS paper describes the design, implementation, and performance evaluation of two designs of web-servers. A web server is a software that allows clients to connect to it using a well-defined protocol (HTTP) and retrieve web content. The client is typically a browser such as Internet Explorer or Mozilla. There are various design alternatives for web servers, some of which include multi-threaded, thread-pool based, and event-driven web servers.

The rest of this paper is structured as follows. Section II provides a brief overview of web servers and discusses the motivation behind this project. Section III gives a detailed account of the design and implementation of the multi-threaded web server, while section IV does the same for the thread-pool based web server. Section V presents the experimental methodology and section VI explains the experiments performed and results obtained. Section VII discusses various aspects of the project, and section VIII presents the conclusions.

This project was undertaken as a part of the advanced computer networks course requirements at Duke University, under the guidance of Dr. Adolfo Rodriguez.

II. OVERVIEW AND MOTIVATION

A. Overview

The web is perhaps the most important part of the Internet today. The common protocol responsible for the web is the HyperText Transfer Protocol (HTTP). The HTTP protocol specifies how web clients (browsers) should talk to servers and vice versa. Any server program that implements the HTTP protocol is called a web server, and can be accessed by any browser.

I have implemented two basic flavors of web servers in this assignment – the multi-threaded web server and the thread-pool based web server. The main focus of this project is on the multi-threaded web server and its performance evaluation. A comparison of the two types of web servers is also presented, as is a discussion of the performance of HTTP 1.0 versus HTTP 1.1. I also profiled the code of the web server to analyze the time spent in various activities such as creating the socket, setting socket options, binding, listening, processing the request, etc.

B. Motivation

The project helped to understand network programming techniques, and the issues that arise in the same. The concurrency problems that can occur in a multi-threaded web server are endless and their resolution needs a robust programming methodology.

The main motivation behind this assignment was to understand the HTTP protocol and implement the same. It created an understanding of how diverse clients and servers can easily interact and exchange information, just by defining a complete and robust language for the interchange.

Another motivation behind this project was to investigate the various factors that contribute to web server performance. A popular web server may have to handle millions of client requests of differing object sizes. A deeper understanding of the issues and costs of various aspects of a web server would immensely help in developing a robust web server.

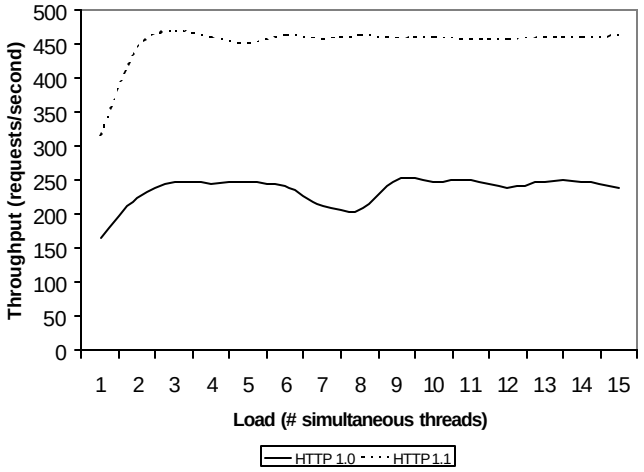

Fig. 1. Throughput (req/sec) vs. load for 8kb file size

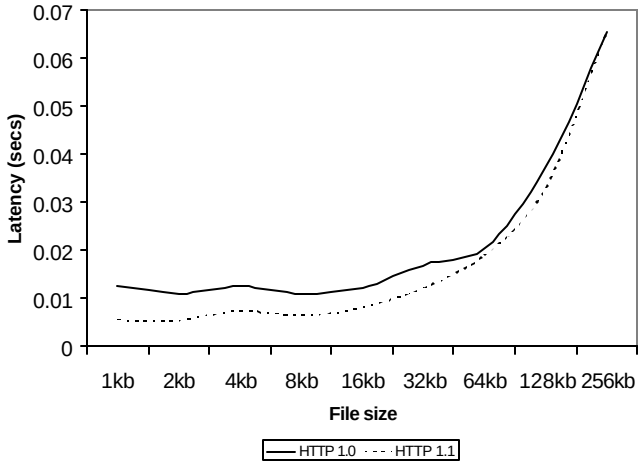

Fig. 2. Latency (secs) vs. file size

III. The Multi-threaded Web Server

The main focus of this assignment was in building a multi-threaded web server. The web server was implemented in C++. A server class is created for handling the main loop which waits for incoming connections. The server first creates a TCP socket and binds to it after setting the required options. For each incoming connection, a new thread is spawned for processing the same. The spawned thread creates an instance of a HTTP handler (HttpHandler object) and tells it to take care of the connection.

The server supports HTTP 1.0 as well as HTTP 1.1. It creates persistent connections when the client requests a HTTP 1.1 connection. The connection is kept open for http_timeout seconds, and if no request is received from the client in this timeframe the connection is closed. The timeout value is adaptive, based on the load on the server. For light loads, the timeout stays at a constant configurable value. But, as the load increases beyond a threshold, the timeout is reduced linearly to compensate for the increased load. Any other function (such as exponential) can easily be fitted into the system with minor modifications.

The server has security features such as disallowing access to directories outside the defined web server root. Compilation instructions are provided in the included README.

IV. The Thread-pool based Web Server

A thread-pool based web server was also implemented in C++. Since the design is object oriented, the main change compared to the multi-threaded server was to replace the Server class with a PoolServer class. A pool of threads is pre-created by the server in order to process incoming requests. Each incoming connection is accepted by exactly one thread in the pool. This is accomplished using a mutex for accept within each thread. Other than this, the server is quite similar to the multi-threaded web server.

V. Experimental Methodology

A. The client

In order to test the HTTP server, I wrote a load generator in C++. The load generator takes four arguments – the number of threads to execute in parallel, the time duration of the experiment, the file size (in bytes) to retrieve from the server, and whether or not each thread's connection to the server should be persistent i.e. whether HTTP 1.1 or HTTP 1.0 should be used. The client supports both versions of the HTTP protocol. It spawns the specified number of threads and repeatedly makes requests and reads responses for the specified time duration.

B. Experimental Setup

The server was run on a Sun-Blade-100 (Solaris) system running the SunOS 5.9 operating system. The CPU speed of the system was 500MHz and the machine had 512MB RAM. The client was run on a Sun-Blade-150 (Solaris) system running the SunOS 5.9 operating system. The CPU speed of the system was 650MHz and the machine had 256MB RAM. Both machines were on the same LAN.

VI. Experiments and Results

I ran a number of tests under this configuration, to analyze the performance of the web server. Following are the experiments that I performed, along with the results.

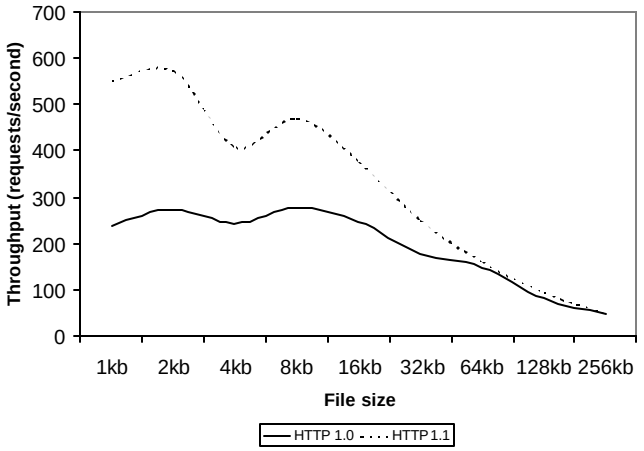

Fig. 3. Throughput (req/sec) vs. file size

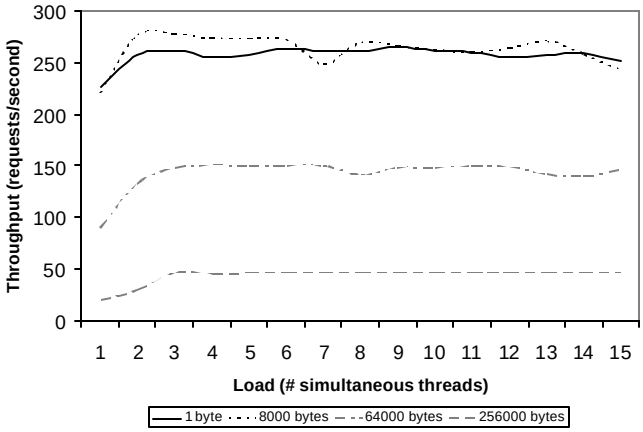

Fig. 4. Throughtput (req/sec) vs. load for different file sizes (HTTP 1.0)

A. Measuring the saturation point

In this experiment, I kept the file size constant at 8kb, and varied the load (number of threads) from 1 to 15. I then plotted the throughput as measured by the client, for both HTTP 1.0 and HTTP 1.1. The intent was to determine the load at which the server get saturated i.e. the point at which the server's throughput plateaus and the server is performing at its maximum. The result is shown in figure 1. It is seen that the server reaches its saturation point when three clients are making requests simultaneously. The results were similar for both HTTP 1.0 and HTTP 1.1, though HTTP 1.1 gave higher throughput as expected.

B. Latency vs. file size

In this experiment, I varied the size of the file retrieved. The load was kept at saturation (three client threads). The result is plotted in figure 2. The x-axis shows the file size in bytes. We see that for both protocols, the latency increases linearly with file size (the graphs show exponential curves because the x axis is logarithmic). Moreover, the performance of HTTP 1.1 is seen to be better than HTTP 1.1. This is because HTTP 1.1 incurs the overhead of creating and destroying the connection only once per session. However, we also notice that as the file size increases, the latencies of HTTP 1.0 and HTTP 1.1 begin to coincide. This is because a large fraction of the overall latency is for the transfer of the large file, and so the cost of connection setup and destruction is not as important in the overall latency.

C. Throughput vs. file size

In this experiment, I varied the size of the file retrieved and plotted the server throughput in requests per second. The load was kept at saturation (three client threads). The result is plotted in figure 3. The x-axis shows the file size in bytes. We see that for both protocols, the throughput decreases with file size. Moreover, the throughput of HTTP 1.1 is seen to be higher than that of HTTP 1.1. This is because HTTP 1.1 incurs the overhead of creating and destroying the connection only once per session. However, we also notice that as the file size increases, the throughputs of HTTP 1.0 and HTTP 1.1 begin to coincide. This is because a large fraction of the overall service time is for the transfer of the large file, and so the cost of connection setup and destruction is not as important, leading the two throughputs to converge.

D. Throughput and latency vs. load for various file sizes

In this experiment, I varied the load from 1 to 15 for each of a number of file sizes. For each experiment, I computed the average throughput and latency of each request. Latency here is measured from the time the client issues a request until the time the response has been completely received by the client. Throughput is the number of requests per second as seen by the client. The results of this experiment are shown in figures 4 and 5. As seen, with increasing file sizes, the throughput decreases and latency increases. Also, beyond the saturation point, the throughput is more or less steady while the latency increases linearly with increase in load.

E. Comparison of multi-threaded and thread-pool based web servers

I compared the two types of web servers as follows. I kept the file size constant at 8000 bytes, and varied the load from 1 to 15. The thread pool of the second server was kept at a size of 10. The experiment was performed for HTTP 1.0. The results are shown in figure 6. As seen in the figure, both server perform similarly. However the thread-pool based web server shows lesser fluctuations as compared to the multi-threaded server. Also, its throughput is marginally higher. This can be

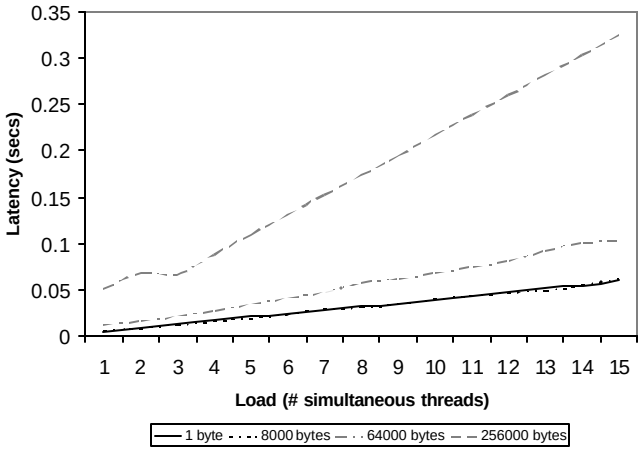

Fig. 5. Latency (secs) vs. load for different file sizes (HTTP 1.0)

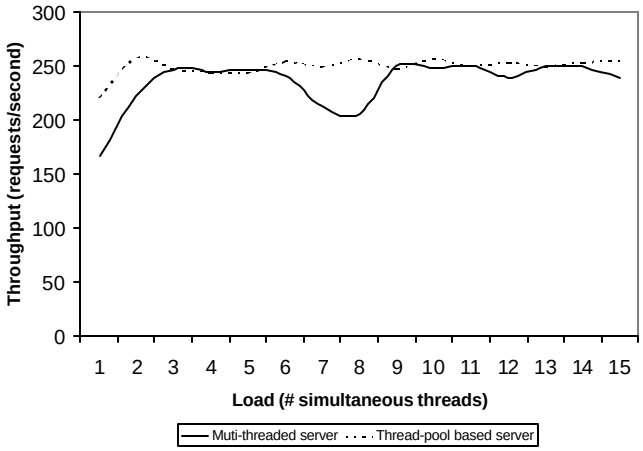

Fig. 6. Throughtput (req/sec) vs. load for 8kb file size

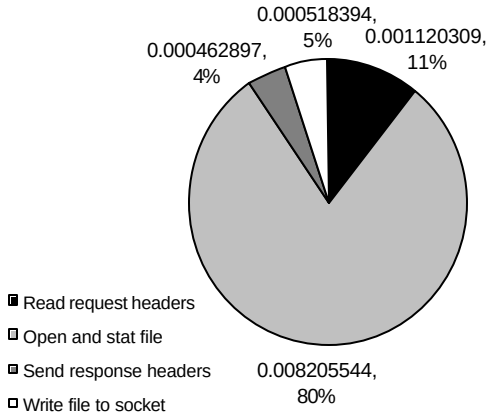

Fig. 7. Profile of each request handled (average time taken).

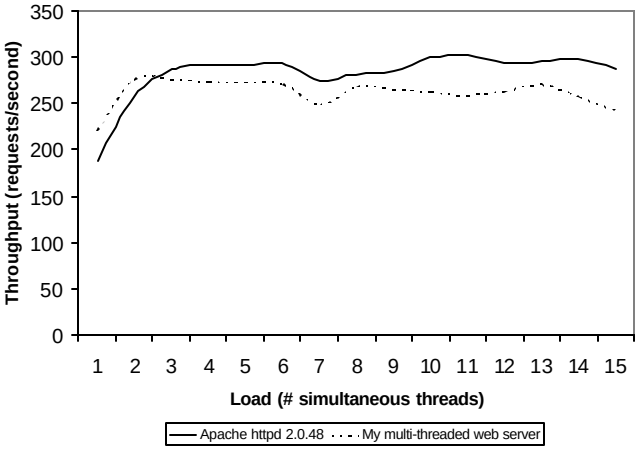

Fig. 8. Comparison of throughputs of Apache web server and my multi-threaded web server.

attributed to the fact that there is no overhead of creating and destroying threads for each incoming connection.

F. Profile of server code

I instrumented the server code with logging information to note the time taken in various stages of the request handling. I wrote a Perl script to extract information from the logged data. The resulting profile is shown in figure 7. As seen from the figure, the majority of request service time is spent in opening and 'stat'ing the file (80%). This is because we are inundating the server with requests, and the file system becomes the bottleneck. Most of the time is spent trying to open and get the statistics about the file before reading and sending the file over the socket connection. I also profiled the cost of other operations such as creation of socket, socket setup, binding the socket, listening, etc. These were found to add minimal overhead to the web server.

G. Comparison of my web server with a real one (Apache)

The Apache Project [3] is a collaborative software development effort aimed at creating a robust, commercial grade, feature rich, and freely available source code implementation of a real web server. Apache accounts for more than 55% of all web domains on the Internet. I compared the performance of Apache with my web server by using the same client. I installed the Apache httpd 2.0.48 on the same machine on which my web servers were evaluated. The client machine was also chosen to be the same as in the previous experiments. The results of the comparison are shown in figure 8. We see that for very low loads, my web server performs slightly better. This is because my server is not as feature-rich as Apache and hence its performance would be better. However for higher loads, the performance of Apache is marginally better. This is to be expected, as Apache is a commercial-grade server.

VII. Discussion

A. Effect of thread model on performance

The thread model can have an impact on web server performance. If we have a basic multi-threaded server, it would perform well under saturating load, but if the load is increased too much, the server will become slow and unresponsive. This is because there is no limit to the number of threads spawned. As a result, a large number of threads would be active and the system will be overloaded. The latency for individual requests will increase exponentially and the throughput will decline rapidly. However, if we use a thread-pool based design for the server, the problem of unbounded number of threads would be eliminated. However, when the request rate increases beyond the capability of the thread pool, queues will build up and the latency will degrade. But, the bandwidth would not deteriorate as in the multi-threaded case since the load of the threads would be limited by the number of threads in the thread pool. An event-based approach would be the best model, because it would provide excellent isolation of various requests and ensure that the performance degraded fairly and uniformly with increasing load.

B. HTTP 1.0 versus HTTP 1.1 for varying RTT

For small round trip times, the difference between HTTP 1.0 and HTTP 1.1 would not be pronounced. This is because the main advantage of HTTP 1.1 is the elimination of the tearing and reestablishment of connections for each object being requested. If the round trip time is small, the SYN-ACK exchange sequence would not take much time and the performance improvement of HTTP 1.1 would not be too much. But as the round trip time increases, the contribution of connection establishment and tearing to the overall latency would increase, and hence HTTP 1.1 would perform better than before compared to HTTP 1.0.

C. HTTP 1.0 versus HTTP 1.1 for varying file sizes

This was illustrated in figure 3. For small file sizes, the relative time to establish and tear the connection would be large as the file transfer itself is short. Hence HTTP 1.1 would do very well as compared to HTTP 1.0. However, for large file sizes the improvement would not be as apparent because the request service time would be dominated by the file transmit time and not the time for establishment/tearing of the connection.

D. HTTP 1.0 outperforming HTTP 1.1

One situation in which HTTP 1.0 could outperform HTTP 1.1 is when we have a large number of requests from *different* clients coming in to the server, and the requests are for just a single object in the server. In case of HTTP 1.1, since the connection is retained until the timeout, the thread would be idle and unable to process any other new request. Moreover the benefits of caching the connection are lost since the client only desires a single object. In such a situation, HTTP 1.0 would outperform HTTP 1.1.

VIII. Conclusions

Two models of web servers were designed, implemented, and evaluated in this project. It was found that HTTP 1.1 outperformed HTTP 1.0 in all the test scenarios that I experimented with. A number of interesting results were obtained and these were discussed in detail. The server code was also profiled to determine the bottlenecks. The web server performance was also compared to that of a commercial-grade server (Apache). In conclusion, it is clear that web server performance is of prime importance. It is necessary to design a web server using a robust model so that performance does not deteriorate with increasing load, and the web server is fair to all clients in the event of overload.

References

[1] L. L. Peterson and B. S. Davie. Computer Networks, A Systems Approach. Morgan Kaufmann, 2000.

[2] W. R. Stevens. UNIX Network Programming. Prentice Hall, 1997.

[3] The Apache HTTP Server Project. Online http://www.apache.org, 1997.