MULTI-THREADED WEB SERVER

Parmananda Banakar, Kaushik Sadashiva Reddy, Chethan Ramesh, Aditya Calambur
University of Massachusetts, Amherst

ABSTRACT:

Web Server refers to a server software or hardware dedicated to running said software, that can serve contents to the World Wide Web. Our implementation of webserver is multithreaded, where the processing of each incoming request takes places inside a separate thread of execution. We have analyzed the performance of our web server with the performance of Apache’s web server based on throughput and latency for concurrent requests.

Keywords: Multithreading, Webserver, Concurrency.

INTRODUCTION:

A web server processes incoming network requests mainly over the HTTP protocol and several other related protocols. The web server primarily stores, processes and delivers web pages to clients. The communication between client and server takes place using the HTTP. Pages delivered are most frequently HTML documents, which may include images, style sheets and scripts in addition to the text content. The client which is usually a web browser initiates communication by making a request for a specific resource using HTTP and the server responds with the content of that resource or an error message if unable to do so. The resource is typically a real file on the server's secondary storage.

Web server application must be concurrent to service multiple requests simultaneously from substantial number of clients. Web server application is also expected to provide high throughput with extremely low response time for each request while servicing multiple clients. This can be implemented using event-based programming as in SEDA architecture. We believe it is difficult to write concurrent applications using event-based programming as the application must keep track of all the state transitions and call backs of the events. It is also hard to schedule events to hardware resources in multiprocessor environment to achieve concurrency.
So, we propose to implement a thread-based web server. The web server application is implemented as a single process with multiple threads, each thread servicing a request from client.

Improvements in CPU architecture like Simultaneous Multithreading (SMT) have effectively doubled the instruction level parallelism available in each processor. We believe our implementation of web server application can better exploit these hardware resources and maximizing CPU utilization for the application in multiprocessor environment without introducing large overhead.

Threads abstract the underlying hardware and provide simple programming model for the application developer.

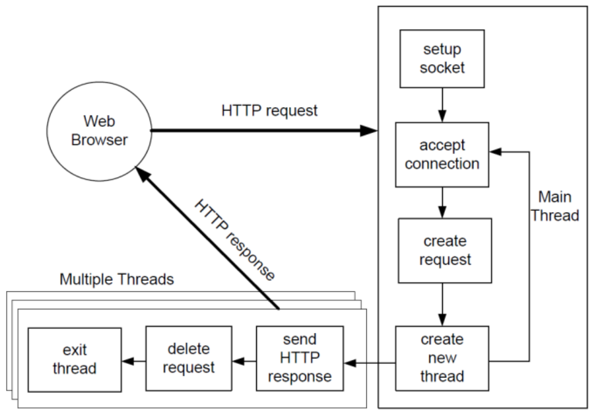

Figure 1: Multi-threaded architecture

Threads share the same address space of the parent process, this allows light weight context switching between threads. For example, when a thread is waiting on page fault, another thread can process different client request. Figure1, depicts the architecture of our multithreaded server.

IMPLEMENTATION:

Hardware Specifications: Server will execute on Intel Core i7-7500U CPU @ 2.70GHz x 4, 64-bit processor with 16 GB RAM.

TCP connection is established using socket programming for the communication between client and server as shown in figure 2. We will be using the POSIX thread (pthread) library to implement threads in the application. The POSIX threads are implemented as light weight processes in Linux kernel which provide fast context switch with small overhead.

As shown in figure TCP connection will be established between server and client with 3-way handshaking. Service thread reads the HTTP commands received from the client. Command will be parsed and validated. If the request is valid, HTTP OK response is sent back to client, else an error message is sent.

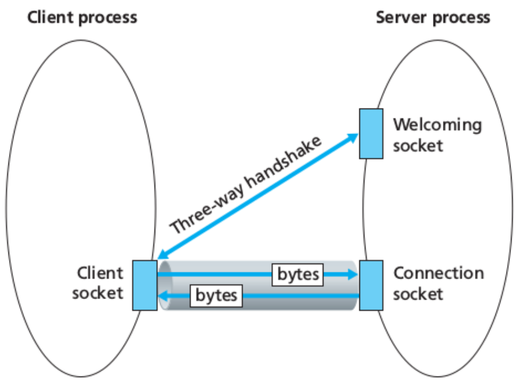

Figure 2: Socket creation

We will support different file formats like html, txt, jpeg, gif etc. If it is GET request, then requested file is looked up in directory relative to the configured root directory of the web server. If the file is not present in the directory, error message is sent back to client. If GET request is received without any file name, then the file name is interpreted as index.html.

After servicing the requested file, connection will be closed with finish ACK and thread will exit in HTTP 1.0 protocol. In case of HTTP 1.1, connection is kept open until timeout waiting for any new request from the same client. This timeout is dynamically changed based on number of existing connections and maximum number of connections allowed. Threshold will be set based on the number of active connections, which will reduce the response time of the requests being serviced. After reaching threshold server declines any new requests received until any existing request closes.

ANALYSIS:

The web server is evaluated by measuring throughput and latency at different load conditions. We have used httperf tool to generate and measure web traffic. Httperf was used for rigorous analysis of web server with real time inputs and we have analyzed results about response rate and latency.

The web server was tested by running client and the server on different machines connected via LAN ethernet cable. The setup also included a few system configuration changes for full use of hardware and giving us better results. The changes included increasing the number of open file descriptors from 1024 to 65535; increasing the range of TCP ports from 1024 to 65535

We also have compared throughput of web server for increasing number of client requests. Ideally the throughput is expected to increase linearly with the load in the initial phase and then with growing requests it should remain constant.

RESULTS:

Based on the above performance analysis we plotted two graphs from the log files generated from httperf and compared the performance of our web server implementation with that of Apache.

Figure 3. shows the server throughput and response time, Initially the throughput increases linearly with the traffic load. At the point (6000 requests/sec), our implementation starts to saturate because of disk I/O and CPU utilization limitations and then it services a constant number of requests (approx. 5000 requests/sec).

The graph also shows the throughput and response time for Apache server. It indicates the throughput of Apache server is higher than our implementation. The saturation point for Apache server is (8000 requests/sec) and then it services the requests constantly at an approximate rate of 8000requests/second.

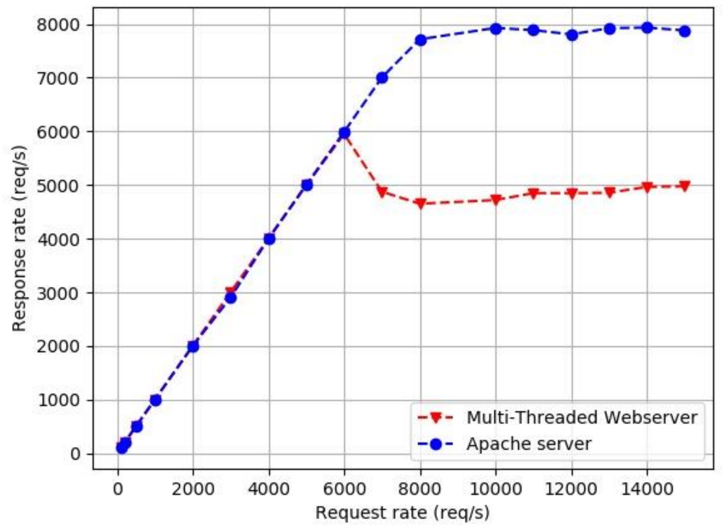

Figure 3: Throughput

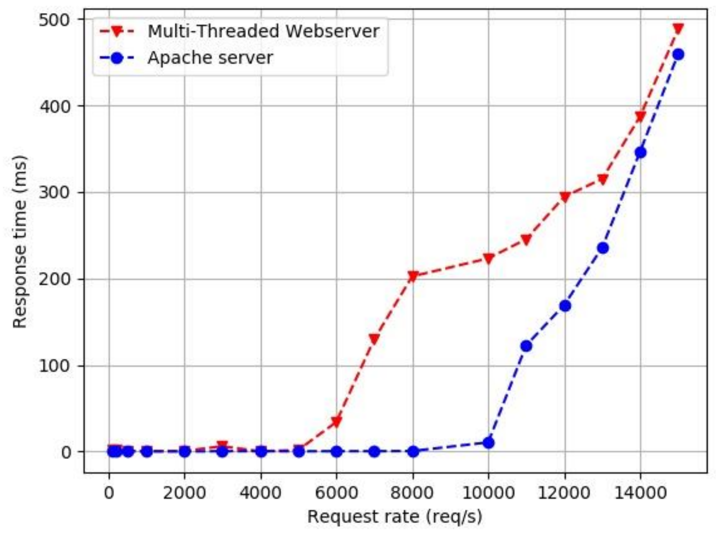

Figure 4: Response Time

Figure 4. shows the average response time versus the load. For small number of client requests, the web server services the requests almost immediately as indicated in the graph.

As the number of requests approaches the threshold, the requests will wait indefinitely to be served resulting in exponential increase in response time. In our implementation, the response time increases exponentially at the point (6000 requests/sec).

The response time of Apache server is also considerably better than our implementation. The response time hits a threshold at 8000 requests/sec after which it increases exponentially. But for higher number of requests (greater than 14000 requests/sec), the response time is almost equal to our implementation.

Analysis clearly indicates that the Apache server is highly optimized for better performance.

FUTURE WORK:

We could have cache implementations in our Web Server to increase the performance. We believe there will be a considerable increase in the throughput and response rate especially if same request files are requested repeatedly.

CHALLENGES FACED:

- Debugging is complex.
- Results are sometimes unpredictable and inconsistent as we cannot predict exactly which thread is running at a time and how much CPU time each thread gets.

CONCLUSION:

We have successfully implemented multi-threaded webserver application which supports HTTP 1.0/1.1. The results obtained were satisfactory and as expected. We have also compared our webserver performance with Apache2 server on the same machine. Apache2 performed better and this shows us that there is still room for improvements in the future. This project has greatly helped us in understanding the concepts of socket programming, pthreads multithreading, synchronization, web server and the HTTP protocol.

ACKNOWLEDGEMENT:

We would like to thank Prof. David Irwin for his comments and timely suggestions during this project.

REFERENCES:

[1] The Apache Software Foundation. 2017. Apache HTTP Server. (2017). *https://httpd.apache.org*

[2] Rob von Behren, Jeremy Condit, and Eric Brewer. 2003. *Why Events Are a Bad Idea (for High-concurrency Servers).* In Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9 (HOTOS'03). USENIX Association, Berkeley.

[3] Matt Welsh , David Culler , Eric Brewer SEDA: *An Architecture for Well-Conditioned, Scalable Internet Services (2001)*

[4] Chandramouli, Badrish, *Development and performance evaluation of multi-threaded and thread-pool based web servers*.

[5] David Mosberger. 2017. A tool for generating HTTP traffic. (2017). https://github.com/httperf