Implementation of Thread-based Web-Server

Rajvardhan Deshmukh
UMass Amherst
rdeshmukh@umass.edu

Shamanth Kumar
UMass Amherst
sparameshwar@umass.edu

Mohanish Panchal
UMass Amherst
mpanchal@umass.edu

Disha Siddappana Palya
UMass Amherst
dsiddappanap@umass.edu

ABSTRACT

In this project, we have implemented a functional web server using the threaded approach. We have enough references that state thread based systems can provide a simpler programming model that achieves equivalent or superior performance than event based approach. The project has helped us to understand the basics of distributed programming, client/server structures, and issues in building high performance servers.

The threaded approach provides 3 key features[1]: 1) Scalability 2) Efficient Stack Management and 3) Resource aware scheduling. Thread model is useful as it improves implementation to remove scalability barriers. A web server listens for connections on a socket (bound to a specific port on a host machine). Clients connect to this socket and use a simple text-based protocol to retrieve the files from the server. In a multi-threaded approach a new thread will spawn for each incoming connection i.e, once the server accepts a connection, it will spawn a thread to parse the request, transmit the file, etc.

Keywords

Web-Server, Multi-threaded programming, HTTP/1.0, HTTP/1.1

1. INTRODUCTION

Today's Internet services have ever-increasing scalability demands. Modern servers must be capable of handling tens or hundreds of thousands of simultaneous connections without significant performance degradation. [1]Thread based approach give us the ability to handle these large number of connections with ease of implementation. Thread packages provide a natural abstraction for high concurrency programming.

Another method popular nowadays is the event based approach which handles requests using a pipeline of stages. Each request is represented by an event, and each stage is implemented as an event handler[1]. These systems allow precise control over batch processing, state management, and admission control; in addition, they provide benefits such as atomicity within each event handler.

Unfortunately, event-based programming has a number of drawbacks when compared to threaded programming. Event systems hide the control flow through an application making it difficult to understand cause and effect relationships when examining source code and when debugging. For instance, many event systems invoke a method in another module by sending a "call" event and then waiting for a "return" event in response. In order to understand the application, the programmer must mentally match these call/return pairs, even when they are in different parts of the code. Furthermore, creating these call/return pairs often requires the programmer to manually save and restore live state. This process, referred to as "stack ripping" is a major burden for programmers who wish to use event systems[1]. Some sources[1] stated that events are better compared to threads. After closely inspecting these claims, the researchers found that it was only true when threads were not implemented properly.

2. IMPLEMENTATION DETAILS

The project is divided into following modules.

2.1 Path Translation

The server translates the relative filename to absolute filename in a local file-system and the directory from which the files are accessed is passed as an argument while starting the server program. Index.html will be the default file if no explicit filename is mentioned in the HTTP request. Web server translates GET/ into GET/index.html. The sample request sent by the browser is as follows:

GET /index.html HTTP/1.1
Host: 127.0.0.1:8001
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml,application/xml;
q=0.9,/*;q=0.8*

Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive

As shown in above request, we use the second and third field of first line to identify the file name and http version. If no file name mentioned then we translate it to index.html and continue.

2.2 Check presence of file

Our server implementation handles the following error codes[4]. We have supported basic error codes which are required for any web server to function.

a)200-OK: This is a standard response for successful HTTP requests. The actual response will depend on the request method used. Since we are supporting only GET request, for this request, we send the following response.

GET /index.html HTTP/1.1
HTTP/1.1 200 OK
Date: Tue May 9 23:20:28 2017
Content-Type: text/html
Connection: Keep-Alive Keep-Alive: timeout=10, max=5
Content-Length: 20751

The response contains a header with response code, other required fields and an entity in body corresponding to the requested resource.

b)400-Bad Request: The server cannot or will not process the request due to an apparent client error (e.g., malformed request syntax, too large size, invalid request message framing, or deceptive request routing). The sample request and response are as follows:

GET / HTTP1.1

In the above request, HTTP1.0 is not proper as expected by HTTP standards. Our server will return following response for the same.

HTTP1.1 400 Bad Request
Date: Tue May 9 23:15:20 2017
Connection: keep-Alive
Content-Type: text/html
Content-Length: 109

<html>
<body>
<h1>Bad Request</h1>
<p>This server did not understand your request.
</p>
</body>
</html>

c)403-Forbidden: Our server will send this response in following scenario.
1) Authentication was provided, but the authenticated user is not permitted to perform the requested operation.
2) The operation is forbidden to all users. For example, requests for a directory listing return code 403 when directory listing has been disabled.Sample response is as follows:

HTTP/1.1 403 Forbidden
Date: Tue May 9 23:18:48 2017
Connection: keep-Alive
Content-Type: text/html
Content-Length: 111

<html>
<body>
<h1>Forbidden</h1>
<p>403 Forbidden You don't have permission to access it on this server.</p>
</body>
</html>

d)404-Not Found: Server sends this response if the requested resource could not be found but may be available in the future. Subsequent requests by the client are permissible. Sample response is as follows:

HTTP/1.1 404 Not Found
Date: Tue May 9 23:18:48 2017
Connection: keep-Alive
Content-Type: text/html
Content-Length: 111

<html>
<body>
<h1>Not Found</h1>
<p>The requested URL was not found on this server.</p>
</body>
</html>

2.3 HTTP Support

We have supported both HTTP/1.0 and HTTP/1.1. For HTTP/1.0 a separate connection is used for each requested file, whereas for HTTP/1.1, the server keeps connections to clients open, allowing for "persistent" connections and pipelining of client requests i.e., after the results of a single request are returned (e.g., index.html), the server will by default leave the connection open for some period of time, allowing the client to reuse that connection to make subsequent requests. We have configured the timeout needed to keep the connection open based on the number of other active connections the server is currently supporting. Thus, if the server is idle, it can afford to leave the connection open for a relatively long period of time. If the server is busy servicing several clients at once, it may not be able to afford to have an idle connection sitting around (consuming kernel/thread resources) for very long[3].

We have chosen 10 seconds as the default timeout period, however we are calculating our timeout dynamically depending on number of open connections. We have implemented a simple heuristic to vary the timeout dynamically. Formula for the same is as follows:

$$DELAY = DEFAULT - (int)(2 \cdot log_{10}(iActiveThreads)) \quad (1)$$

In the above formula, DELAY is dynamically updated based on variable iActiveThreads and constant value DEFAULT which is defined as 10. As per our observations we found that the maximum number of Active threads at a given time is less than the order of 10^5, so we assign the delay of zero seconds once it exceeds this limit.

As we want to decrease the delay gradually over the Active threads range from 1 to 10^5, we have used the best fitting function i.e. log to the base 10 which is defined in math.h header.

2.4 Thread Support

Thread is the heart of this project. We have used pthread library to support thread.

We have used the socket function 'accept' in an infinite while loop which allows us to accept series of requests. Once the socket accepts a connection, a thread is spawned. A shared variable called number of active threads is updated i.e. iActiveThread. Number of thread depends on number of connections accepted by server.

Once the thread begins execution, it reads from socket using read system call. It identifies the required components from the request and performs following validations. 1) First word of first line is GET or not (since we are supporting only GET request). 2) HTTP version (we support only HTTP/1.0 and HTTP/1.1). If it does not pass this validations then we send bad request 400 in response.

If the request is proper we create appropriate response message. We also check for the HTTP version and handle the HTTP/1.1 by keeping the socket open as long as the timeout doesn't occurs and requests keep coming. After servicing the requests the shared variable called number of active threads is updated, the socket is closed and the thread dies.

Deciding the critical section depends on the use of the shared variables and appropriate care has been taken to limit the use of locks to only those sections, so as to reduce performance overhead and performance degradation.
Example:If a webpage contains 3 objects then 4 separate threads/connections are established, one for each of the 3 objects and one for the original webpage[3].

In short, our web server will function as follows:[3]
Forever loop:
1.Listen for connections
2.Accept new connection from incoming client
3.Parse HTTP request
4.Ensure well-formed request (return error otherwise)
5.Determine if target file exists and if permissions are set properly (return error otherwise)
6.Transmit contents of file to the client (by performing reads on the file and writes on the socket)
7.Determine timeout (if HTTP/1.1)
8.Close the connection

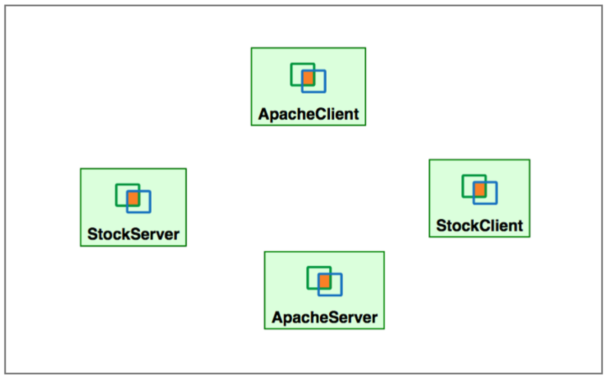

Figure 1: Test Bed in Geni Network showing Apache Client/Server and Client/Server nodes

3. DESIGN AND IMPLEMENTATION

We used the berkelee socket library to get socket functionality.

3.1 Problems faced

Finding a heuristic for the timeout for persistent connection. Sending proper header in reply message. Sending image over TCP connection. Evaluating performance of web-server.

3.2 Specifications

Globally routable GENI nodes are used with system Ubuntu 16.04 LTS 64 bit, processor Intel ® Xeon ® CPU X5650 @2.67 GHz, Memory 988MiB System memory

3.3 Tests and Evaluation

We used httperf to generate http traffic from the client to corresponding servers. Command used to run httperf on client:
httperf –server=IP –port=9000 –num-calls= –num-conn= –rate=100

We started with simple htttperf command by giving just num-conns as argument, for which our server was able to run without crashing. Then we performed the test with different values of num-conns, num-calls and rate to generate different req/sec values. The ece570 home page is used as the default index.html page for both the servers. Our graph for performance is reply/sec v/s requests/sec. Graphs for both the severs are shown in figure 2 and 3.

The graph is linear up-to certain throughput values .

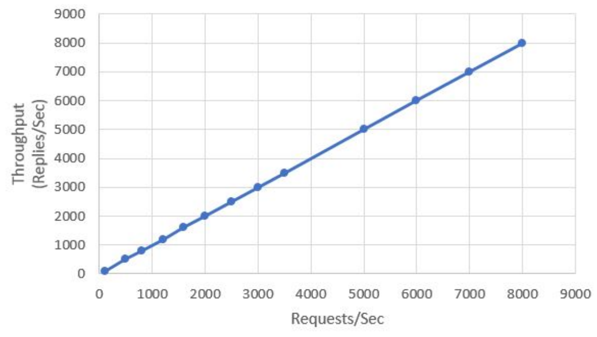

Figure 2: Response rates for different request rates in Apache webserver

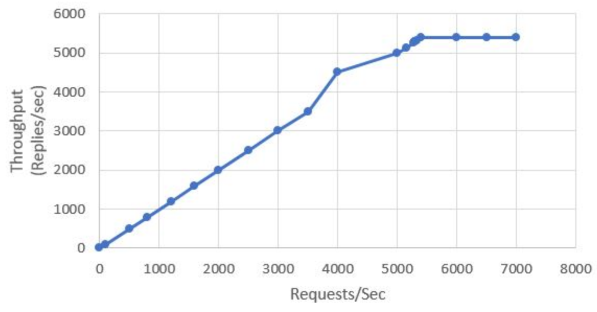

Figure 3: Response rates for different request rates in webserver

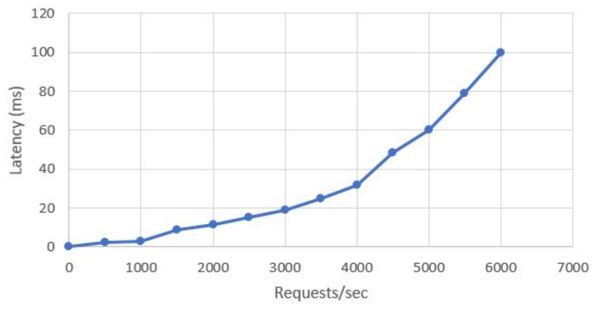

Figure 4: Latency for different request rates in webserver

4. STARTING THE SERVER

./Server <directory_name> <port>

5. CONCLUSION

The threaded approach is simpler to implement. It provides scalability and efficient stack management. The design of the code is less complex and easy to understand. The Webserver that we developed performed as per our expectations. We could open the webpages stored in our directories using Mozilla Firefox web browser. Both the HTTP/1.0 and HTTP/1.1 versions performed as per the norm. The webserver also displayed different status codes that we had defined as per the conditions given. It worked considerably well under high loads. But our evaluations showed Apache performs better as compared to our Webserver for higher loads. Possible reasons are:
1) we might not have optimized the performance in all possible area.
2) we used kernel level thread library.
3) Our heuristic for timeout is not efficient one.

6. ACKNOWLEDGEMENT

We would like to thank Prof. David Irwin for his valuable comments, guidance and assistance throughout the project.

7. REFERENCES

[1] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. `"Capriccio: Scalable Threads for Internet Services"` in SOSP'03:October 19-22, 2003, Bolton Landing, New York, USA.Copyright 2003 ACM.

[2] Matt Welsh, David Culler, and Eric Brewer. `"SEDA: An Architecture for Well-Conditioned, Scalable Internet Services"` in Operating Systems Principles (SOSP-18), Chateau Lake Louise, Canada, October 21-24, 2001.

[3] Web Server Project. `http://www.ecs.umass.edu/ece570/assignments/server.html`

[4] List of HTTP status codes. `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`