Designing a Web Server in Rust

Team Members: Arulselvan Madhavan & Jimmy Ly

PDF Version of Project Report: CS5600FinalReport.pdf

PDF Version of Project Presentation: CS5600_Project_Presentation.pdf

Source Code: webserver.tar.gz

Designing a Web Server in Rust

April 27, 2016

CS5600 Final Report

Team Members: Arulselvan Madhavan & Jimmy Ly

0. Abstract

The focus of this project was to leverage the features of the relatively new Rust programming language in order to build a scalable web server. The implementation began as a very basic and naive web server. However, the architecture gradually grew in size as a thread pool was integrated in order to enable parallelization of tasks, priority scheduling was introduced to prevent large file requests from clogging the web server traffic, and server side caching was added to reduce the number of necessary I/O operations. Logging the incoming requests and outgoing response statuses was also implemented across the threads in the thread pool. The ApacheBench tool was used to measure the performance at the various stages of the web server's progression. These benchmarks were compared with those of the Apache web server in order to determine whether a Rust web server is capable of matching the scale an existing web server that is widely used today. This initial evaluation suggests that the Rust web server design without the prioritization is comparable and perhaps slightly better in performance than the Apache web server for smaller files. The priority scheduling effectively serves files in order of file size, however, continued work is required to improve the overall performance. Additional work also includes adding other typical features found in existing web servers.

0.5 Contributions

The project was pair programmed in entirety. We would design and implement the web server during 3-4 hour meeting periods, which would take place several times a week.

1. Introduction

A web server built in Rust would have many advantages including safety, high performance, and easy concurrency over existing web servers. While many high performing applications are written in C and C++ to take advantage of the manual memory manipulation, the program is vulnerable to many memory-related issues involving segfaults, buffer overruns, and memory leaks. Writing applications in other languages, such as Java or Python, rely on built-in memory management that is often not optimized for the purpose of the program.

Rust combines the advantages of both low-level languages and high-level languages by providing both safety and control. Using Rust's built-in ownership system, most bugs that would be found (and difficult to debug) at runtime with a C or C++ program can be found and fixed at compile-time. This potentially makes building the system more extensible as the code base grows larger and more complicated since it can be less prone to newly introduced bugs.

2. Novelty

Although there are other web servers in existence, such as Apache or NGINX, these are written in C and are therefore susceptible to the issues described above. There are not any widely used web servers that are written in Rust, therefore the purpose of this project is to determine the feasibility of implementing a web server in Rust. The goals include to discover whether the performance of the Rust web server can compare with that of an existing popular web server, such as Apache. It is also desirable to explore how extensible the web server is: will adding new features likely introduce new bugs in the code?

3. Design

3.1 Basic Tasks of a Web Server

There are several stages in designing the web server in Rust. The first stage is to develop a bare bones web server implementation. This involves writing a single-threaded program that can listen for incoming requests, determine which file to serve, then serve the file.

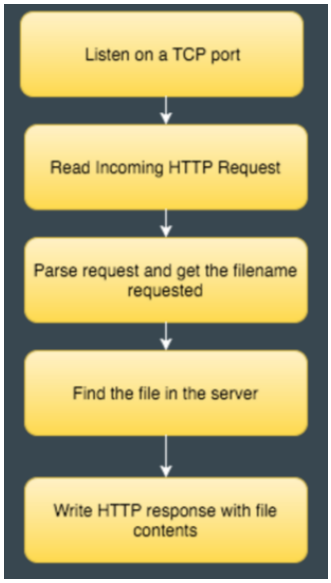

Fig 3.1: Basic tasks of a web server

3.2 Parallelizing the Tasks

This very basic implementation suffices to show that a minimalistic web server implementation is possible using Rust. However, this design is not scalable for numerous concurrent incoming requests. Most of the tasks required to read the request and serve a file can be parallelized. Therefore, the goal in designing the next stage of the web server is to use multiple threads to concurrently handle the incoming requests. The first step in this direction is to have the main thread spawn a thread for each incoming request, having the new thread read the request and serve the response.

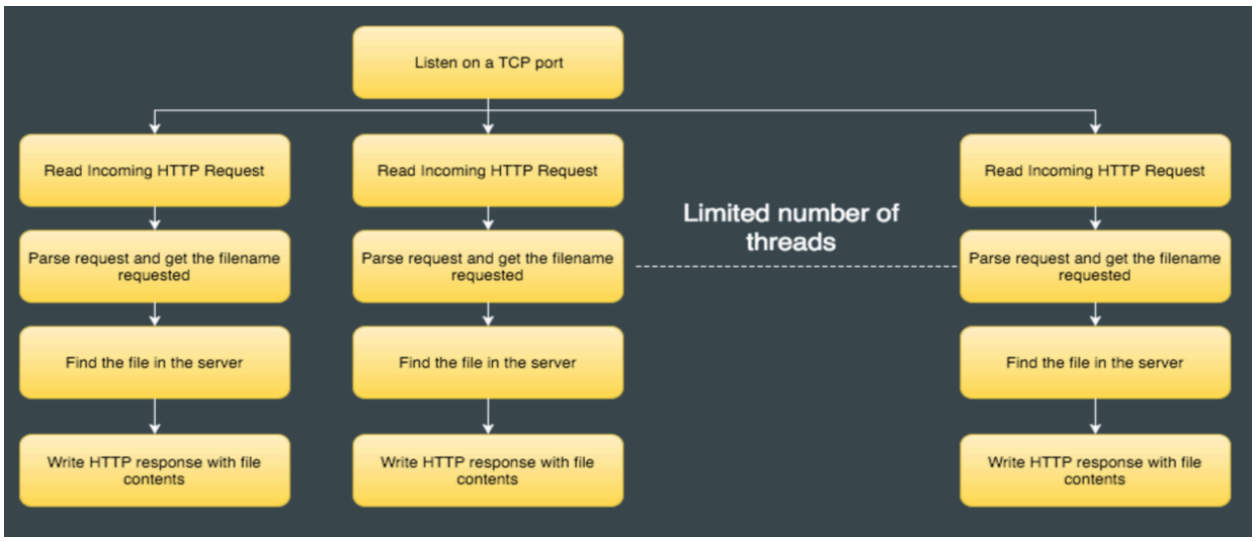

Fig 3.2 Parallelizable blocks of a web server

Although this naive approach satisfies the parallelization task stated previously, this implementation still cannot scale for three reasons:

1. There is a limit on the number of threads that can be spawned
2. Spawning new threads is rather expensive
3. Scheduling many threads also adds a large amount of overhead.

3.3 Thread Pool Integration

The intuitive solution to this problem is to integrate a thread pool into the web server. A fixed number of threads are spawned at the start of the web server and remain idle until requests are received. For each request, a thread is woken and tasked with both reading the request and serving the requested file. An existing Rust thread pool library is used for this purpose.

Challenge: Safely share the incoming requests among the threads in the thread pool. Each request should be served by exactly one thread. No incoming request should be dropped without being served.

Solution: Safely share the incoming requests among threads by enforcing the threads to acquire lock before processing an incoming request.

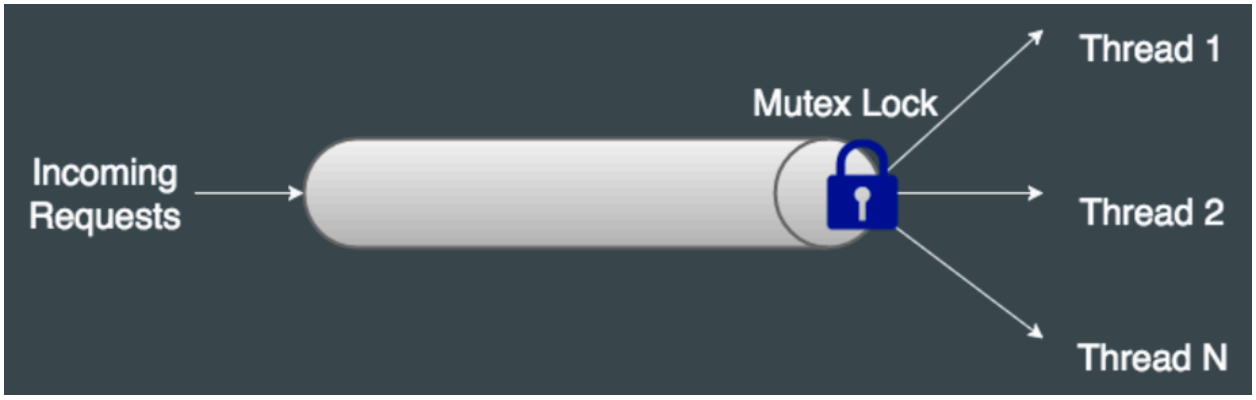

Fig 3.3 Rust Channels

3.3.1 Rust's support for Concurrency:

Rust supports interprocess communication between threads via channels. A channel has two ends: Transmitter and Receiver. There can be multiple receivers with a single transmitter or multiple transmitters with a single receiver. When a channel is created, a transmitter and receiver are created in memory. In the case of multiple receivers, each thread is given an atomic reference to the receiver end of the channel. The receiver threads acquire a mutex lock on the receiver end to receive incoming requests. The requests sent by the transmitter sit in the channel buffer, in FIFO order, waiting to be processed by the receiver end. The buffer in the channel is unlimited. When there are no requests in the channel buffer, the receivers go to sleep and yield their CPU.

3.4 Prioritized Thread Pool

A web server gets several hundreds of requests every second. It cannot treat each request with the same priority. For example, consider a scenario in which there is high priority file 'A' of size 10KB and a low priority file of size 100MB. If the web server receives the request for the low priority file first before the high priority file and starts serving it, the high priority file may have to wait until the full low priority file is served. A web server could lose business if it fails to serve the high priority jobs ahead of low priority jobs.

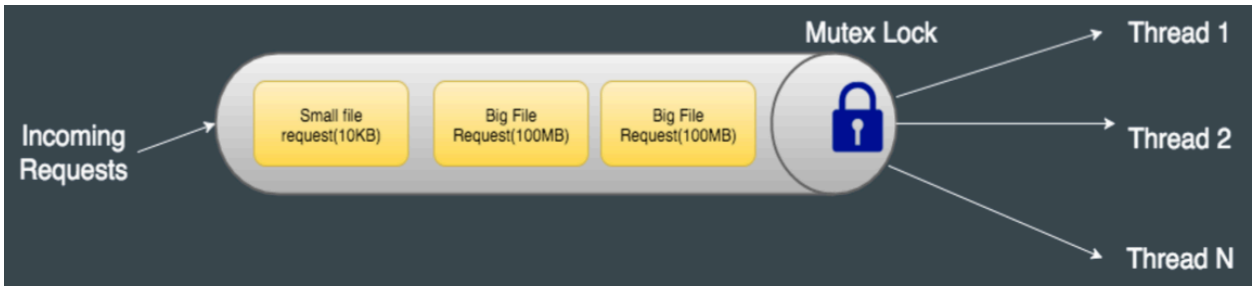

Fig 3.4 Problem with non-prioritized web servers

Solution: We achieved prioritization of incoming file requests by adding an additional layer of indirection to our web server architecture.

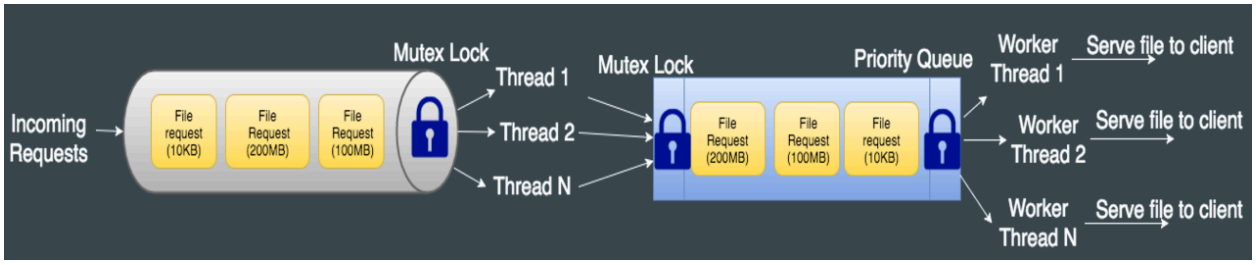

Fig 3.5 Web server with priority based scheduling

Customizable Prioritization: In our implementation, we have prioritized the incoming requests based on the file size. In practice, there maybe multiple factors based on which a request may have to be prioritized. Our design is flexible enough to be extended to any prioritization scheme.

Types of threads:

1. Scheduler Threads: The main task of a scheduler job is
    a. Acquire the lock on the incoming requests channel
    b. Calculate the priority of the request.
    c. Assign a priority value to the request in hand
    d. Acquire the lock on the priority queue.
    e. Push the request into the priority queue
    f. Release the lock

1. Worker threads:
    a. Acquire the lock on the priority queue.
    b. Pop the request with the highest priority
    c. Process the request
    d. Find the requested file
    e. Return an appropriate HTTP response.

3.5 Web Server Cache

File I/O is a costly operation. A web server could benefit from a carefully designed cache that reduces the number of file IO operations carried out by a web server. There are several factors that can be taken into consideration for caching a file. The type of requests that go into the cache is usually dictated by the size of the cache and the type of contents served by the web server. In our implementation, a file is cached in the memory if its file size is less than 50KB. This is an arbitrary number. Any number can be chosen as a threshold for caching the file contents.

Design: A concurrent hashmap is used as the cache. A RWlock protects each entry of the hashmap. Each worker thread will have access to the cache. In the event of a cache hit, the worker thread accesses the cache contents and returns it to the client. In the event of a cache miss, the worker thread accesses the file on the disk and writes the file contents as an HTTP response. After writing the response to the stream, the worker thread decides if the contents needs to be pushed into the cache. If yes, then the worker thread waits to acquire the writer lock. Once it gains write access to the corresponding entry in the hashmap, it writes the file contents to the cache.

Key: Path to the file

Value: Contents of the file

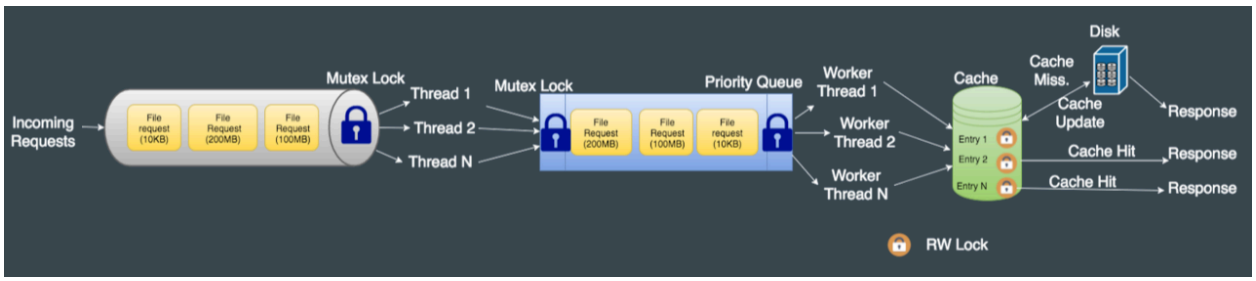

Fig 3.6 Web Server with Cache

Challenge: The worker thread waits for a long time to acquire the lock on a cache item. Since worker threads are one of the highly used threads, it would be better if the worker threads are better used by not making them wait for a lock.

Solution: Add an additional layer of indirection for cache update requests. All worker threads will have to send their cache update requests to a designated cache update thread.

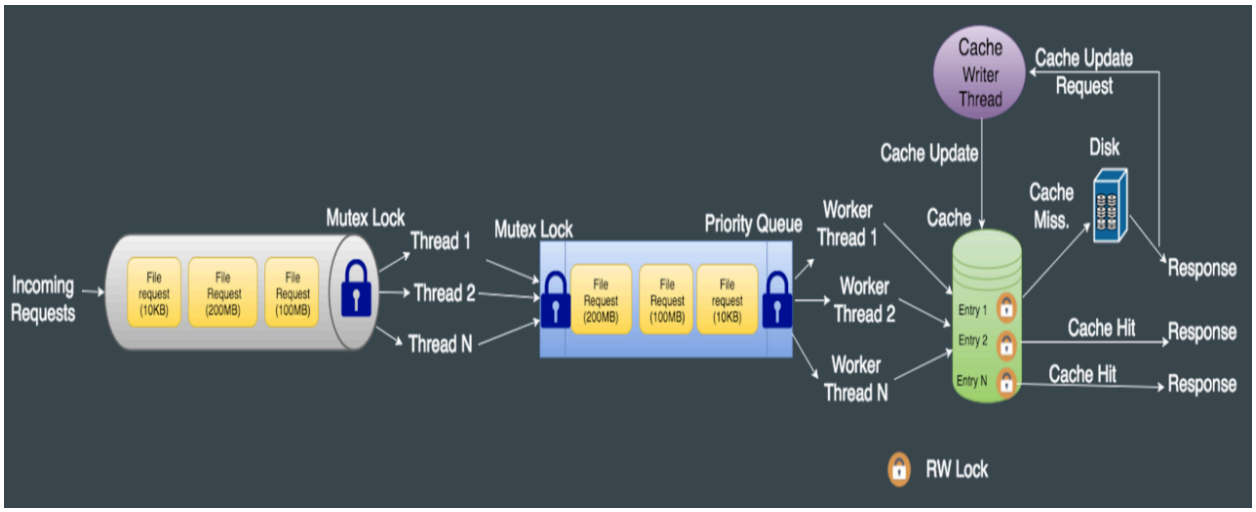

Fig 3.7 Web Server with Cache thread

3.6 Logging

We have several thread pools in our web server architecture. It would be better if all threads write to the same log file. So we have a separate logger thread to which all the log file requests are routed. The logger thread owns the log file and processes all log write requests in the order it received them.

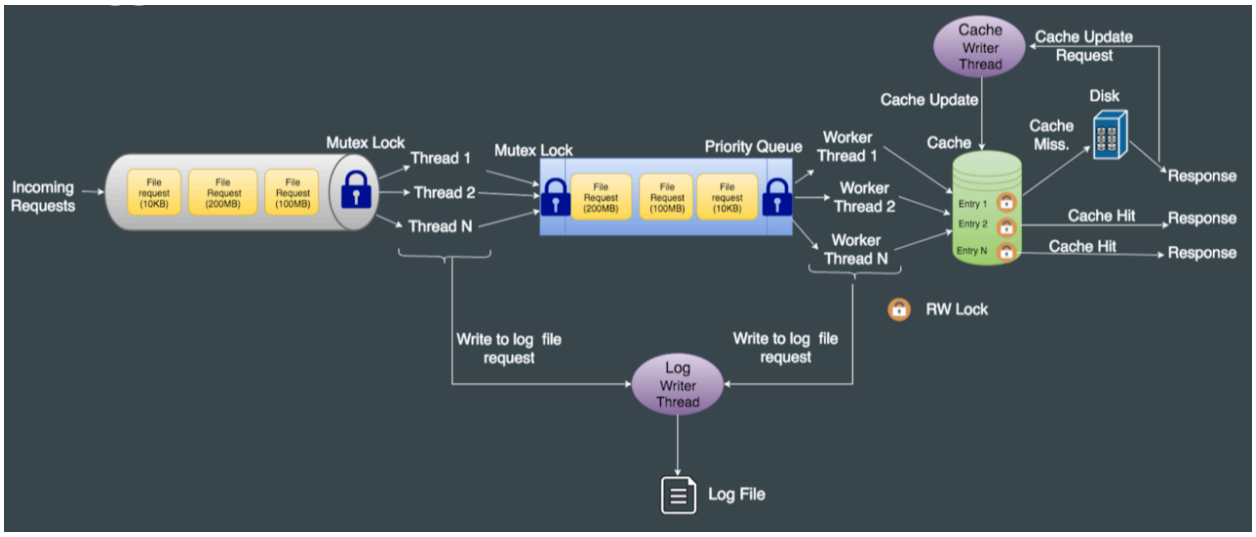

Fig 3.8: Web server with logger thread

4. Implementation / Technical Challenges

4.1 Basic Web Server Implementation

For the very basic web server implementation, everything was executed on the main thread of the program. Rust's built-in standard network module was utilized to bind to a TCP port and listen for incoming TCP connections. The program enters an infinite for-loop where an incoming HTTP request is represented as a TcpStream instance. The TcpStream can be read to obtain the details of the HTTP request from the client. The request is then parsed to determine the path to the requested file. Rust provides a standard file system library that allows the user to open a file, similar to the open system call C. A match statement is used for the result of the open call to handle the case if the file is found with one branch of

code and to handle the case if the file is not found with a different branch of code. If the file exists in the file system, then the file is opened, and its contents are written to the TcpStream in order to send the response back to the client. The file contents are prepended with a response status line containing the protocol (HTTP/1.1) and the status of the response. In this case, the status is 200 OK. If the file is not found, then the server will write a 404 NOT FOUND error response back to the stream, which sends this different response back to the client.

4.2 Spawning New Threads for New Requests

As mentioned in section 3, the next phase of the web server involves spawning a new thread for each incoming request. These threads would serve the file and then exit. In Rust, spawning a new thread is a just matter of calling a static “new(...)” constructor method for the thread and passing in a closure as an argument. This closure describes the routine that the thread will execute, and it also contains all of the necessary data that the thread will be responsible for. The new thread is given the current TcpStream instance by the main thread. Just as before, the new thread then reads the request from the stream, searches for the file, and serves the file if found. When the thread reaches the end of the closure, the thread stops executing and is dropped. At that point, its resources are freed. There is no need for the parent thread to wait for the child thread.

4.3 Integrating the Thread Pools

To fix the number of threads that the web server could spawn, an existing Rust thread pool library was utilized. The configuration file for Rust’s package manager called Cargo was updated to include the dependency. To use the library, an initialization method was called, passing in the number of threads to spawn in the pool. Then instead of passing the closure to a new thread, the closure is passed to the thread pool. The thread pool wakes a blocked thread and has the thread execute the closure to server the file request.

4.3.1 Prioritization and Rust’s Support for Concurrency

Introducing the priority scheduling of file requests became one of the harder tasks of the projects due to Rust’s ownership model. In summary, the ownership model works by initially allocating memory for variables in the stack and stating that each piece of data has a single owner. When passing a variable directly to a function, for instance, this passes ownership of that variable to the function. Since the ownership of that variable has moved, the caller cannot use that variable unless it the variable is the return value of that function. With only one owner, the variable is freed whenever the variable goes out of scope without worrying that it will being used in another location.

Although these rules help to enforce programming safety, the priority queue which enforces the file serving order must be shared and updated across many threads. Rust provides an Arc (atomic reference counting) wrapper type for sharing data across thread boundaries. When calling the clone() method on a variable wrapped in an Arc, a new reference to the variable, and the reference count is incremented. Once all of the references no longer exists, then count drops to 0 and the data is freed.

As a result, the receiver end of the channel where jobs were passed to the scheduler threads was wrapped in an Arc and also a Mutex type. The Arc type solved the issue of ownership across threads, and the Mutex type solved the issue of synchronization across threads. This way, all of the scheduler threads had access to the receiver end of the channel to take jobs from the channel buffer, and there were not any concurrency issues or data races since a thread would need to acquire the lock before taking a job. For similar reasons, the priority queue was wrapped in an Arc and a Mutex type since both the scheduler threads and the worker threads needed to access and update the queue.

For the priority queue itself, the binary heap implementation from Rust’s standard collections module was used as the implementation. A FileJob struct was created to represent a task for the worker threads, containing the TcpStream, the size of the requested file, and the details of the request. The Ord trait was implemented for the FileJob struct, which essentially allows the priority queue to order the FileJobs by file size.

4.4 Caching Small Files

As mentioned in section 3.5, a concurrent hashmap was used from an existing library. Rust’s file system standard module includes a Metadata struct which contains the length of a file given the path to the file. This file size was stored in the FileJob struct and used to determine whether the file contents should be stored in the hashmap cache or not. The most complicated issue that was found while implementing the cache was determining the datatypes to hold the file contents when reading or writing the file contents and when storing the file contents in the cache. In the end, the file contents are stored as bytes in a Rust Vector type since the size of Vectors can be dynamically allocated, but the file contents must be stored in an array of bytes when writing to the TcpStream. Ideally only one data type would be used for both situations to avoid the overhead of copying the data, but Rust does not currently support writing from Vector types. Also, the size of arrays must be determined by compile time so it isn’t possible to allocate byte arrays of the file size to store in the cache since the file sizes vary and are not known until runtime.

4.5 Logging Across Threads

For logging, the logger thread owned a single receiver end of a channel while all of the other threads owned a reference to an Arc protected transmitter. When the logger thread is spawned, it opens the log file. Whenever a thread sends a logging request through the channel, the logger thread writes the request to the open log file.

5. Evaluation

The different web server configurations were benchmarked using the ApacheBench tool on an Ubuntu 14.04 instance running on an Oracle

VirtualBox virtual machine. The performance of the Apache web server was also observed to compare the Rust web server with a popular existing web server implementation. The load tests for the various server configurations consist of 10 concurrent open HTTP connections that send a total of 100,000 HTTP requests to the web server. For each load test, a specific file is requested from the server. The file is either 10kb, 40kb, or 80kb in size. Recall that the all files smaller than 50kb in size are cached by the Rust web server. The 10kb file is chosen to be a small cached file, the 40kb file is chosen to be a larger cached file, and the 80kb file is chosen to be an uncached file.

5.1 Choosing the Number of Scheduler and Worker Threads

Since the Rust web server is configurable in the number of scheduler and worker threads that are spawned, the the first step to benchmarking the prioritized thread pool configurations is to determine the optimal number of schedule and worker threads. This is done by fixing the number of worker threads to 4 and running load tests against varying numbers of scheduler threads. The configuration that can handle the most number of requests per second is selected. The procedure is then repeated by fixing the number of scheduler threads to 4 and varying the number of worker threads to find the optimal number of worker threads to spawn. The values in the tables 5.1 and 5.2 below are the requests per second for each configuration

Scheduler Thread Count	File Size		
	10kb File	40kb File	80kb File
1	2940.00 requests/sec	2947.35 requests/sec	2565.05 requests/sec
2	4295.18 requests/sec	4200.18 requests/sec	3339.98 requests/sec
4	6334.83 requests/sec	5588.38 requests/sec	4133.2 requests/sec
6	6532.16 requests/sec	5983.55 requests/sec	4086.58 requests/sec
8	7029.61 requests/sec	5962.65 requests/sec	4005.15 requests/sec
10	7117.44 requests/sec	6800.31 requests/sec	4558.33 requests/sec
12	7172.09 requests/sec	6670.28 requests/sec	4630.43 requests/sec
14	7097.49 requests/sec	6537.1 requests/sec	4048.33 requests/sec

Table 5.1 Load Test Results for 4 Worker Threads and Various Scheduler Thread Counts

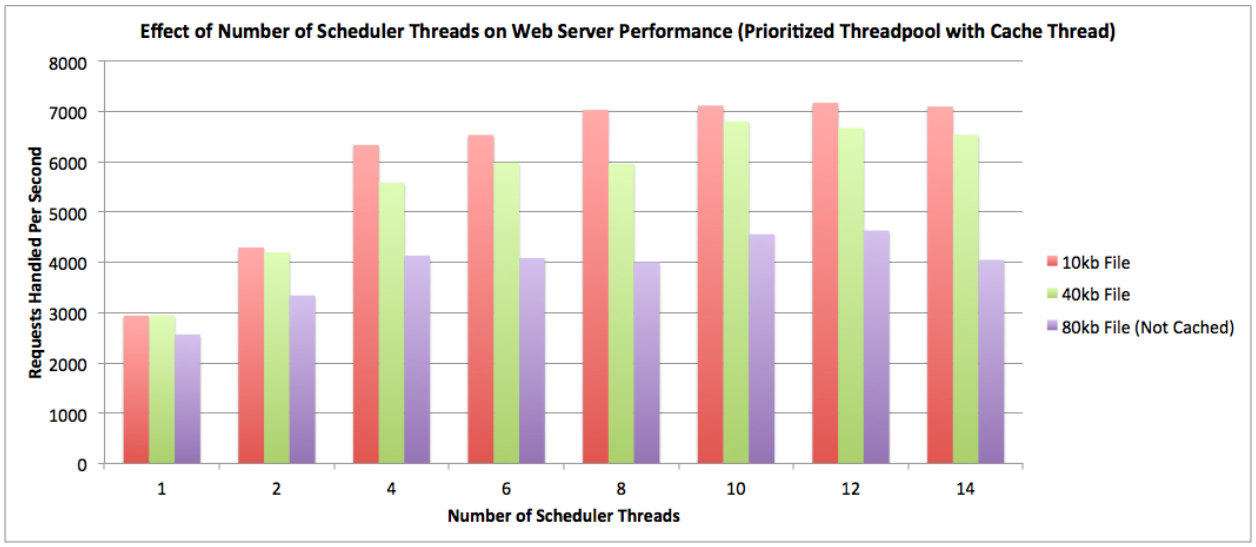

Figure 5.1 Load Test Results for 4 Worker Threads and Various Scheduler Thread Counts

Worker Thread Count	File Size		
	10kb File	40kb File	80kb File
1	7470.79 requests/sec	6575.76 requests/sec	3551.67 requests/sec
2	7347.41 requests/sec	6449.22 requests/sec	4402.22 requests/sec
4	6199.36 requests/sec	5989.15 requests/sec	4378.04 requests/sec

6	5998.17 requests/sec	5820.64 requests/sec	3776.65 requests/sec
8	4606.35 requests/sec	5184.86 requests/sec	3817.18 requests/sec

Table 5.2 Load Test Results for 4 Scheduler Threads and Various Worker Thread Counts

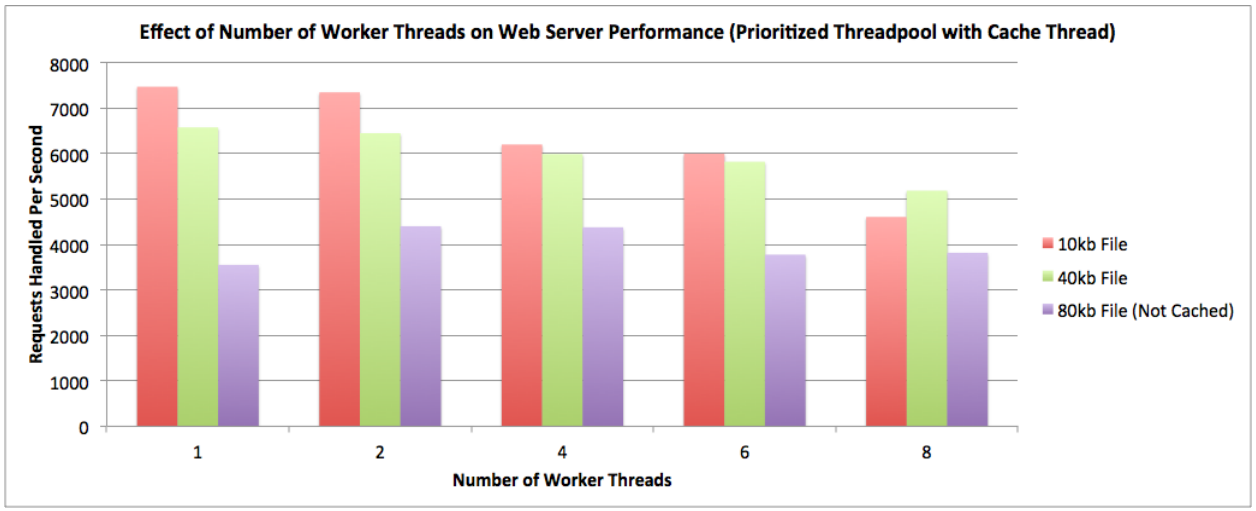

Figure 5.2 Load Test Results for 4 Scheduler Threads and Various Worker Thread Counts

Based on these initial load test results, the optimal configuration for the prioritized thread pool is to have 12 scheduler threads spawned and 2 worker threads spawned. Although the configuration with 1 worker thread performed slightly better than the configuration with 2 worker threads when serving the 10kb and 40kb file, the configuration with 2 worker threads was significantly better when serving the 80kb file.

5.2 Benchmarking the Various Web Server Designs

With the number of scheduler and worker threads to spawn determined, load tests were then performed across various designs of the Rust web server, as well as with the Apache web server. The results can be seen in table 5.3 and figure 5.3.

	Existing Thread pool Library	Prioritized Thread pool (No Cache)	Prioritized Thread pool (Cache, no cache thread)	Prioritized Thread pool (Cache thread)	Apache
10kb File	10027.43 requests/sec	6583.26 requests/sec	7529.2 requests/sec	7517.02 requests/sec	9732.3 requests/sec
40kb File	10717.15 requests/sec	4051.38 requests/sec	6791.37 requests/sec	7092.27 requests/sec	8910.72 requests/sec
80kb File	10221.92 requests/sec	4927.44 requests/sec	4892.3 requests/sec	5048.13 requests/sec	6932.61 requests/sec

Table 5.3 Load Test Results for Various Web Server Designs

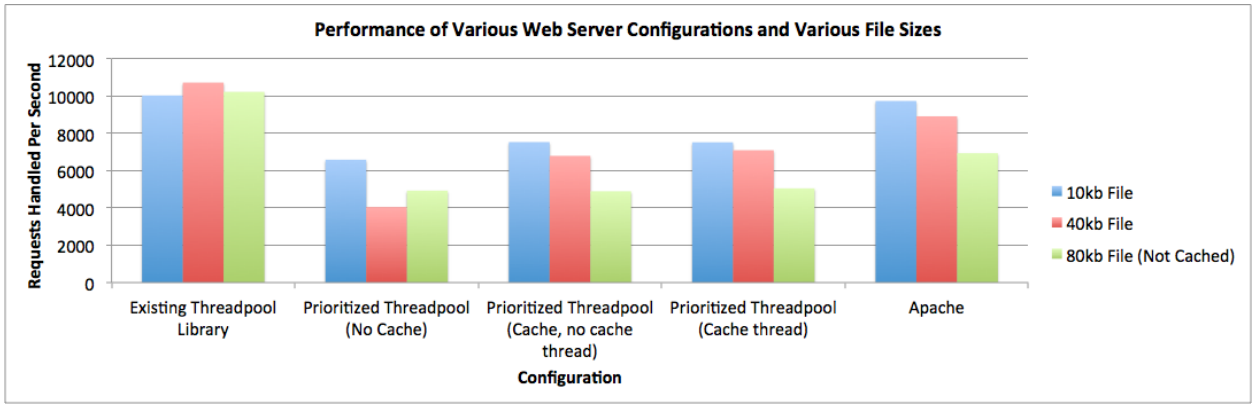

Figure 5.3 Load Test Results for Various Web Server Designs

These results suggest that the design of the Rust web server that includes the existing thread pool library without prioritization outperformed the Apache web server and all of the other Rust web server designs for these load tests. Apache also outperformed the Rust web server designs that include prioritization. The results also suggest that the web server does perform better when serving cached files since the test results for the 10kb and 40kb files improved from the design without the cache to the designs with the cache. However, the performance of the server with the cache thread did not do significantly better than the implementation where each thread updates the cache.

Since the three file sizes chosen are relatively close in size and rather small, load tests were performed where a 10mb file was requested. The results are found in Table 5.4 below.

	Existing Thread pool Library	Prioritized Thread pool (No Cache)	Prioritized Thread pool (Cache, no cache thread)	Prioritized Thread pool (Cache thread)	Apache
10mb File	278.60 requests/sec	273.31 requests/sec	266.92 requests/sec	276.85 requests/sec	320.92 requests/sec

Table 5.4 Load Test Results for a Large File

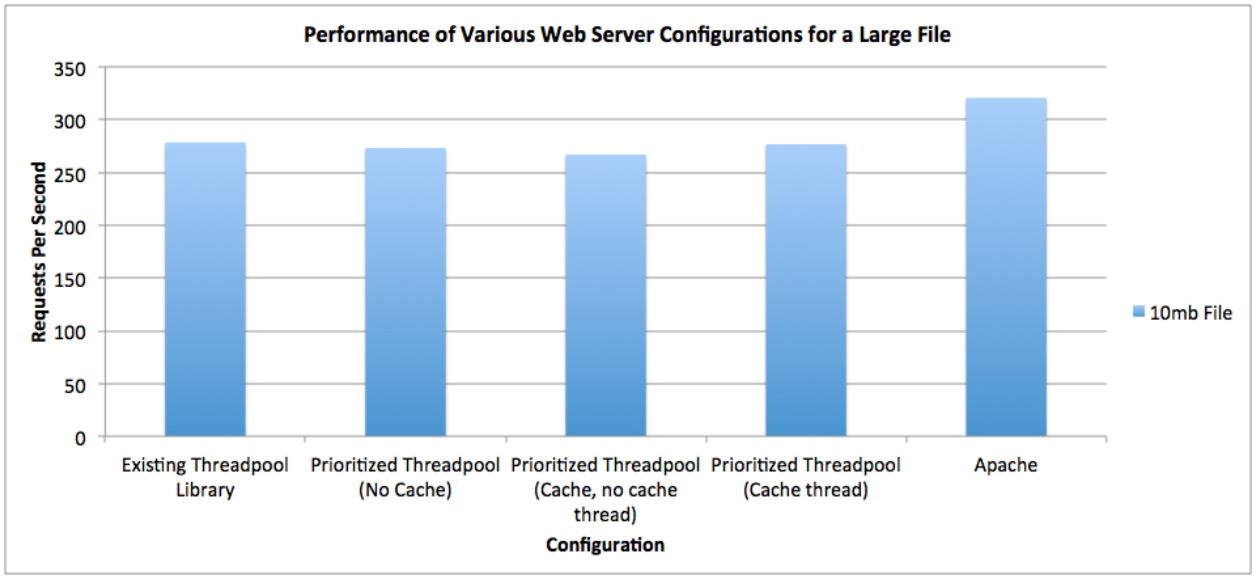

Figure 5.4 Load Test Results for a Large File

After load testing with the large file, the requests handled per second was much more similar across the different web server configurations. The range of the results was approximately 54 requests per second, as opposed to the range of about 6000 requests per second when serving the 10kb file. In this scenario, Apache had the best performance, although the results were rather close. The duration of the large file load tests was most likely dominated by the I/O operations of reading and writing the file. This may explain the similarity in results. In the other case with the small files, the portion of time spent on I/O was probably less in comparison to the other web server tasks, such as managing the various threads.

Note that each load test only involved serving one file at a time. Additional tests were run where 3 concurrent HTTP connections sent 33,333 requests for the 10kb file, 3 concurrent HTTP connections sent 33,333 requests for the 40kb file, and 4 concurrent HTTP connections sent 33,334 requests for the 80kb file. All three components of the test were run simultaneously to simulate the situation where the web server must serve files of varying sizes at the same time. The results are shown in table 5.5 and figure 5.5.

	Existing Threadpool Library	Prioritized Threadpool (No Cache)	Prioritized Threadpool (Cache, no cache thread)	Prioritized Threadpool (Cache thread)	Apache
10kb File	3659.71 requests/sec 9.108 sec total	2575.21 requests/sec 12.944 sec total	2734.61 requests/sec 12.189 sec total	2785.85 requests/sec 11.965 sec total	2885.02 requests/sec 11.554 sec total
40kb File	3665.25 requests/sec 9.094 sec total	1899.65 requests/sec 17.547 sec total	2289.95 requests/sec 14.556 sec total	2205.26 requests/sec 15.115 sec total	2950.97 requests/sec 11.296 sec total
80kb File	4447.61 requests/sec 7.495 sec total	1585.92 requests/sec 21.019 sec total	1856.65 requests/sec 17.954 sec total	1783.35 requests/sec 18.692 sec total	2722.04 requests/sec 12.246 sec total

Table 5.5 Load Test Results With Multiple File Sizes

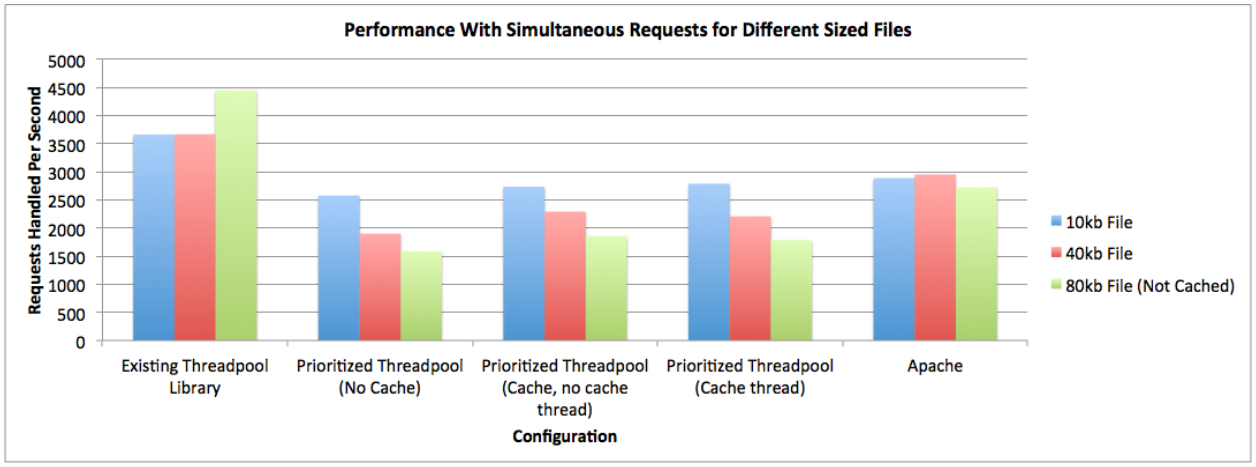

Figure 5.5 Load Test Results With Multiple File Sizes

The table 5.5 includes the time taken to complete each component of the test. For example, it consider the results for the design using the existing thread pool library. It took 9.108 seconds to serve all of the 10kb files, 9.094 seconds to serve all of the 40kb files, and 7.495 seconds to serve all of the 80kb files. In total, the test took 9.108 seconds since this is the maximum time. Although the existing thread pool library implementation without prioritization still completed the tests in the shortest amount of time, notice how the 80kb files were finished being served before the 10kb and 40kb files. With the prioritized thread pool implementations, the files were finished being served in order of increasing file size. This provides evidence that the prioritization is working. Again, the goal with the prioritization is to avoid situations where all of the worker threads are occupied with serving large files and all of the other requests sit waiting in the queue. This could potentially happen with the existing thread pool library, but is less likely to occur with the prioritized thread pool.

6. Conclusion & Future Work

We have built a web server that performs reasonable well compared to the state of the art web servers like Apache. As next steps, we would like to finetune the performance of our prioritized thread pool library to make it perform as well as Apache. Currently, our cache size is not limited. In practice, this is not possible. Our web server could use a cache with size limitation and a carefully designed eviction policy for replacing the old/irrelevant cache entries. Also,we would like our web server to be able to automatically detect the changes in the files cached in memory. We could also add support for HTTP/2, IPv6, web sockets, directory browsing and virtual hosts.