CSCI 339 Project 1: Web Server

Erik Kessler, Kevin Persons
Williams College

Abstract

We describe the architecture, implementation details, and design decisions of our web server which accepts and responds to HTTP GET requests. In this paper, we also explore and evaluate the performance of our implementation. Furthermore, we discuss factors that impact performance such as bandwidth, latency, and file size.

1 Introduction

With this project we aimed to build a functioning web server in order to learn the basics of distributed programming and understand a client/server architecture. Furthermore, it allowed us to experience the difficulties of creating a well-performing service in both the ability to respond quickly and the ability to handle large numbers of requests and degrade gracefully. The problem of providing high-availability means we need to be able to serve multiple clients at once. Possible solutions to this problem include threading, separate processes, or event queues. In this paper we outline the solution we chose and detail the project's architecture. By the end, we evaluate the performance of our server in various conditions and discuss how these conditions apply to HTTP in general.

2 Architectural Overview

We opted to create a multithreaded web server. Our decision to go with threads was motivated by the fact that we could get a single threaded server working and then easily expand to having a new thread for each connection, and given the lightweight nature of threads (as compared to processes) they seem like a good fit for connections as connections are usually short-lived and we want to have many of them to enable high-availability. In practice, we found that going from one thread to multiple threads was straightforward although it did introduce race conditions that were difficult to debug. Our choice to use threads was also motivated by a desire to gain experience with threads which neither of us had. In deciding on how many threads to allow, we went with 100 as our benchmarking suggested that the server could handle that many threads without excessive strain on the system.
Another design decision we made was choosing to use a set sized buffer (8192 bytes) for sending chunks of data to the client. Sending in chunks should allow us to support arbitrarily large files without having to create a massive buffer.
A last design decision worth mentioning is our choice to use C++ for its object oriented nature. Thinking about the server in an object oriented fashion allowed us to encapsulate functionality in a way that makes understanding the working of the system easier. Having a Request object made it easy to reason about request parsing in isolation and having a Connection object made it easy to bundle a Connection to its thread. We discuss our architecture in more detail below.

Three main components makeup our architecture: the WebServer, Connections, and the clock thread (see Figure 1 for a visual representation).

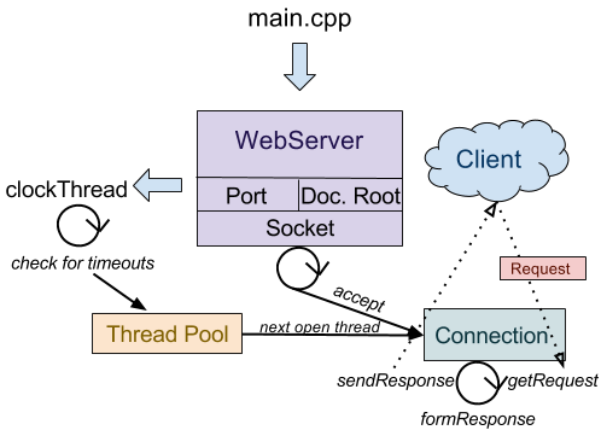

Figure 1: The high level structure of the system

When the server starts, we create a WebServer object which coordinates the activity of the other components and holds a thread pool. The first job of the WebServer is to create and bind to a socket on the specified port number. The WebServer also spawns a new thread for the timeout clock.

The timeout clock thread sits in an infinite loop that loops through the thread pool looking for Connections that are open but not active. The timeout heuristic we used has two levels of need: moderate and critical. Moderate-need occurs when over half the threads in the thread pool have an open connection and critical-need occurs when every thread in the pool has an open connection. When the server is in moderate-need state, it closes any connection that has been idle for three or more seconds, in critical-need, it closes any idle connection, and in the default case, it closes any connection that has been idle for ten or more seconds. We chose this heuristic because when we are using fewer than half of the threads in the pool there is no pressing reason to close a connection because an idle connection doesn't consume many resources, thus a ten second timeout seems reasonable. If we get to the moderate-need case it suggests that our server is under somewhat of a load, so we allow for the connection to stay open for three seconds to give the client plenty of time to reuse the connection if desired while respecting that the server is under a load. In the critical-need, the server is under its heaviest load and it is likely that there are clients waiting to connect so leaving a connection idle is not a good use of resources when there are other clients who want to connect.

Along with the starting the timeout clock, the WebServer also accepts new connections. In an infinite loop, the WebServer first loops through the thread pool looking for the first available thread. The thread pool is an array of ConnectionThread structs. The ConnectionThread struct holds a thread id, a reference to the Connection object, and an integer indicating which connection number it is. When a ConnectionThread is unused, we set the connection number value to a constant, AVAILABLE, so the WebServer is looking for one of these AVAILABLE threads. Once it has an available thread, it calls `accept` which blocks until a client attempts to connect. When a client connects, `accept` returns a connection and we use that result to create a Connection object. The WebServer then starts a new thread for that Connection object and goes back to trying to connect to the next client with another thread from the pool.

On its own thread, the Connection object calls `Connection::getRequest` which reads the raw request from the socket as a string. We then use `Request::parseRawRequest` to convert the raw request into a Request object. Each Request holds the method, requested-uri, version, and headers. If a request is malformed, the Request will have a flag `isValid` set to false. Once `Connection::getRequest` returns a Request, we then use `WebServer::formResponse` to create a response to the request. The `formResponse` method creates the response string ("HTTP/1.0 200 OK"), the headers, and returns a file descriptor of the file to send. If there is an error (malformed request, file not found, missing permissions) `formResponse` will form the response to reflect the error and will return the file descriptor for an HTML page indicating the error. The `Connection::sendRequest` method then sends to the client. If the connection should stay open we repeat this process, otherwise we close the connection.

We designed the server in this way to enable nice

encapsulation of the tasks that need to occur. The WebServer doesn't worry about reading from the socket, parsing requests, or sending to the client; a Connection doesn't worry about detecting errors in the request; a Request only has to focus on correctly parsing the raw request. This encapsulation not only makes reasoning about the code easier but also would allow us to easily add functionality. For example, we could add .htaccess IP-Restriction functionality.

To implement IP based restriction using .htaccess files we could add a subroutine to `WebServer::formResponse`. Since .htaccess files apply down to subdirectories, this method would have to take the file path from the request and traverse up toward the server's root directory looking for an .htaccess file. To improve this lookup on repeated requests we could add a cache that maps file paths to .htaccess files. Such a lookup could be recursive: at a given level, we check for an .htaccess file at the current level, if its not there we check the cache, and if its not there we ask the parent directory which repeats the process. Once we have access to the applicable .htaccess file, we would read and parse the file. We might want to improve performance by keeping a cache from .htaccess files to their parsed representation. With the parsed file we would then get the IP address from the connection and check if it against the rules of the .htaccess file and proceed if the rules accept it, and otherwise send an error response.

3 Evaluation

Here we analyze the performance of our server. One of the first tests we did on our server was trying to load the sysnet homepage. We were able to load the page in both Firefox and Chrome so we tried adding more to the page. We added a couple of 14 MB images along with a CSS stylesheet and javascript file. Our server was able to send the images, stylesheet and script file indicating we had basic functionality.

Next, we wanted to test the impact of threading on our server's performance. To test how adding or removing threads impacts performance we used Apache Benchmark to test various thread capacities of our server. We used a 1 MB file and both our server and Apache Benchmark were running on Unix lab machines. The threading configurations we tested were thread pools of size: 1, 2, 20, and 200. For each thread pool size we tested 1, 5, 25, 50, 100, 150, and 200 concurrent requests by setting the -c flag in Apache Benchmark. For each condition we ran 200 requests and took the average request time. Figure 2 shows the results. First, notice that for every thread pool size except 200 there are points with no data. These missing points indicate there were too many concurrent connections and the server started dropping requests. The large spikes in request time as the number of concurrent requests moves above the number of threads is what we expected given that at those points there are too many requests to handle concurrently so some requests have to wait a significant amount of time before the server frees a thread to handle them.

We also tested the impact of file size on request time using Apache Benchmark. We tested files of size (in KB) 1, 5, 10, 15, 20, 50, 100, 200, and 300. Figure 3 shows the results. We expected that as the file size got larger the request time would go up mostly linearly which is what we saw. For smaller files we can see that the overhead of establishing the connection, parsing the request, and forming the response overshadows the time it takes to send the file. Since this is a local connection it is fast with low latency so the overhead for setting up the connection is small so it doesn't take long for the file size to become the dominant factor. We predict that for more latent connections, the overhead of setting up a connection would be the dominant factor for larger files and that for low latency connections, sending the file would consume the most time.

To test the impact of latency we used Google Chrome's network throttling feature to simulate different network characteristics. We simulated 4 connections all 30 Mb/s but with different latencies: 10ms, 25ms, 50ms, and 100ms. For this test we used file sizes of (in KB) 1, 5, 10, 20, 50, 100, and 200. Using Chrome's network timeline feature, we calculated the percentage of the total request time spent sending for each condition. The results, in

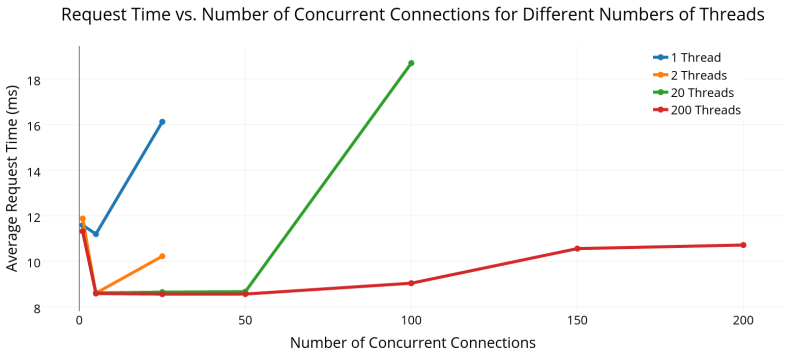

Figure 2: Benchmarking our server with thread pool sizes and levels of concurrency

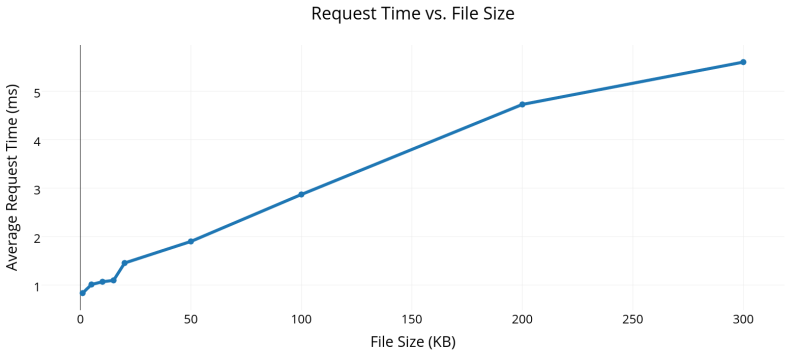

Figure 3: Benchmarking our server with different file sizes

Figure 4, show how for low latency connections, it doesn't take very large files before the majority of the total time is spent sending, and for latent connections, it takes larger files to make the time spent sending be the major factor indicating that latency is the main bottleneck.

The results from the latency test led us to consider the differences between HTTP/1.0 (connection closed) and HTTP/1.1 (connection kept open). A client using 1.0 would setup multiple connections to make requests in parallel while a client using 1.1 would setup a single connection to make requests in serial. Our latency tests show that for low latency connections even small files spend a majority of time sending, so the bottleneck it reducing the time spent sending. Thus if there is high bandwidth so that we can send multiple files in parallel, 1.0 would perform better because sending in parallel allows us to fill the bandwidth, and since the latency is low, creating a connection does not impose a significant overhead. However if bandwidth is low compared to our file sizes there is little to gain from parallelism as a single file can keep the pipe full. In this case, 1.1 might perform a bit better than 1.0 as it can avoid the small overhead of setting up a connection.

On the other hand, when the connection is latent, setting up each connection is costly as it incurs the cost of a full RTT, which is high due to the latency in the network. Thus, it takes larger files for sending to take the majority of the total time. Therefore, if the file sizes mean the pipe will be mostly full, HTTP/1.1 will perform better as it is able to avoid the cost of setting up new connections while HTTP/1.0 would not be able to parallelize enough to overcome the high price of setting up a connection. However if the files are very large and the bandwidth is good there might be benefits to parallelism as the large file size might make it worth the cost of setting up connections.

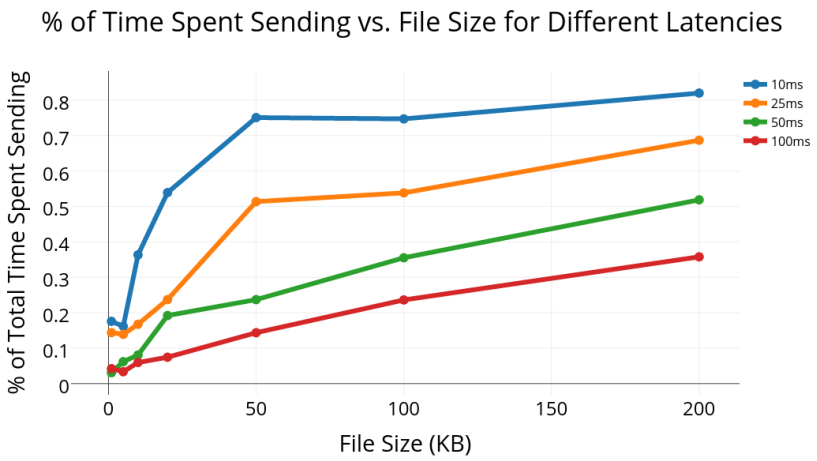

Figure 4: Testing the impact of latency using Google Chrome's Network Throttling

It is clear that both HTTP/1.0 and 1.1 have benefits and it seems like modern browsers like Chrome and Firefox try to take advantage of both by setting up multiple HTTP/1.1 connections. Thus, the browser can gain the parallelism of 1.0 and avoid the setup overhead as provided by 1.1.

4 Problems & Limits

While stress testing our server using Apache Benchmark we found that at high loads the server sometimes crashes due to various errors. We tried to investigate the problem using Valgrind however we were unable to find anything. Debugging these errors proves to be very difficult as they are difficult to reproduce.

One error we were getting was TOO MANY OPEN FILES when accepting a connection. This problem seemed to be the basis of other problems as well. At first we didn't even know that there was an explicit problem: we observed leaking memory and seemingly random crashes. First we figured out how to print error codes for the accept failing and learned there were too many open files which suggested we were missing a `close(connection)`. We used shell commands to determine that we were indeed leaving file descriptors open. After adding the missing close, the problem seemed to go away and we confimed that with the shell commands which indicated that the number of open files wasn't increasing.

Our stress testing also revealed the limits of our server. We found that at around 200 concurrent connections with a 1 KB file, the server is unable to service all of the requests.

5 Conclusion

Building a high-performance web server is difficult even when only supporting basic functionality. We found the main challenge to be scaling our server to handle multiple requests at once because the threading introduced bugs that were difficult to detect, reproduce, and fix. Furthermore, it was difficult to ensure that the server could recover from an error on one thread without bringing the entire system down. Another difficulty is that certain bugs only show up under high loads, so it takes lots of testing to find errors.

Overall the project provided a challenging, yet worthwhile, introduction to C++, threading, sockets, and HTTP.