Nicolas Estrada
Dr. Mattox Beckman
CS421 - Programming Languages and Compilers
07/28/2024

Summary of "Developing a high-performance web server in Concurrent Haskell"

In an effort to explore Haskell's ability to provide real-world tools in the Internet age by utilizing lightweight concurrency, high I/O throughput, and fault tolerance, this paper describes in detail how to build a prototypical web server in less than 1,500 lines of Haskell, excluding GHC-provided library tools.

Getting the basic functionalities of a web server out of the way, this paper is only concerned with basic file-serving functionalities. There is a client that initiates a connection to the web server over a network, in which it sends requests containing information about the file it is requesting from the server. The server then responds to the client connection with the file if it exists in the file system. Accomplishing this seemingly small but arduous task in Haskell presents challenges in the following forms if it were to run a live site:

1. The ability to serve clients simultaneously, at scale. This can be made possible through some form of concurrency.
2. Low latency for each newly established connection, at an increasing rate, along with the ability to defend against denial-of-service attacks.
3. Robustness in error handling and recovery in such events. Fault tolerance will be considered as important as performance with minimal downtime.

With these challenges now in mind, there are also some notable extensions that will make the web server implementation smoother to help facilitate them, such as Concurrent Haskell (Peyton Jones et al., 1996), extensions that support exceptions (Peyton Jones et al., 1999), asynchronous exceptions (Marlow et al., 2001), and last but not least, a wide range of libraries like a networking library, a parsing combinator library, an HTML generation library, and a POSIX system interface library. All are even distributed with the GHC compiler.

Given the importance of which concurrency model to implement in this web server, there are some notable forms worth comparing advantages and disadvantages with:

1. Separate processes
2. Operating-system threads
3. Monolithic process with I/O multiplexing
4. User-space threads

The actual implementation of the web server can be constructed with two major components: an implementation of the HTTP protocol and a top-level main loop which continually listens for connections to accept in order to initialize new transactions.

A functional web server needs mechanisms to support timeouts and logging. A client that experiences an interrupted connection should not cause the entire server to enter a panic and should free up resources back for the system. This is achieved with a generic time-out combinator. Additionally, a web server should produce log files which record:

- A timestamp when the request was received

- The requestor's address
- The URL
- The Response code (success or error)
- The number of bytes transferred
- The time taken for the request to complete
- Client software type & version
- The referring URL

A key thing to note here is that the actual generation of these log files is done by separate threads. Usually, a worker thread is responsible for recording a log entry by calling the function logAccess. Some advantages of placing logs in separate threads are that it can reduce the total load on the system because the worker threads can be garbage collected once they're done. Each thread can serve multiple requests and proceed with later requests without waiting for the completion of others. This design is fault-tolerant, as mentioned earlier.

Since the web server is configured by editing a text file, there should be the ability to make the server run-time configurable to ensure high availability. Rather than restarting the entire server to refresh the config, which would make the site offline, it is important to remember the challenges which the paper wants to address, which include fault tolerance with minimal downtime. When refreshing a runtime config, existing connections will continue to use the old settings, whereas new connections will use the most updated server configuration.

By gathering the implementations described so far, there are some metrics that reveal areas for improvement to make the web server even better in regards to performance. For example, GHC's scheduler context switches about 5000 times/sec. By reducing that to 50 times/sec, it helps decrease the load on the system when the scheduler performs a select on every context switch to determine which I/O-bound threads can be woken up. Increasing the allocation size of the garbage collection settings also improved performance. Rewriting the "hGetLine" function using tail recursion improved the reading of requests by 10%. Nonetheless, the web server has been a useful source of insight into the performance bottlenecks in GHC's concurrency and I/O support. The connection latency results also display an O(n) behavior as the number of threads in the system increases, thus the costs of garbage collection increase since it must traverse the active thread queues to determine where the threads are live. The paper notes a drop-off limit due to limiting the number of active connections that the server can process and stopping listening for new connections once that limit is reached. It is a bit higher for Apache, but this web server in Haskell still proved to be "more than adequate".

In conclusion, this paper demonstrates the feasibility of developing a high-performance web server using Concurrent Haskell. The implementation from the original authors successfully addresses key challenges such as scalability, low latency, and fault tolerance. By leveraging Haskell's concurrency model and various language extensions, the authors were able to create a web server that performs competitively with established solutions like Apache, while maintaining a concise codebase of less than 1,500 lines! This paper highlights areas where Haskell excels, such as in managing concurrent operations and providing robust error handling. The paper also identifies potential bottlenecks and areas for improvement, particularly in garbage collection and scheduler performance. This work not only serves as a proof of concept for using Haskell in production-level web servers but also contributes to the ongoing development of GHC by exposing areas where runtime system optimizations could yield significant performance gains. As such, it represents an important step in establishing Haskell as a viable option for developing high-performance, concurrent applications in the rapidly evolving landscape of web technologies.

My Implementation:

The scope of this work is daunting at first to approach, but in my opinion it goes to show the potential of Haskell and utilizing it for web technologies such as a web server. Below you will find the most basic implementation that I could configure using Haskell, this is intentional. I did not strive to build a similar web server as the author of this work, as it is 1,500 lines of code and my studies at the moment are devoted to the final of this class. However, after this class I do intend to iterate more and play with this to see what it can do.

In the git repo, there is a basicWebServer directory with a repo inside of my implementation of a basic web server in haskell. My thoughts and comments on this are below.

The components include the main function, main loop and client handler. Starting with the main function, it creates a socket, and then binds this socket to port 8080, and then it starts listening for connections. Once this is established, it kicks off the main loop.

In the main loop, it's basically waiting and listening for new connections. When a connection is detected and it comes in, it prints a message to the terminal and spawns a new thread to handle that client. This works well and fast. I have launched some stress tests on my machine by opening new connections on various web browsers and tabs and monitoring what is logged on the terminal about information about each individual connection. I prefer not to share these files but I have also provided a basic test file which goes through the process of creating and starting the web server. In the future, these tests can be more complicated, as the server matures and develops. When running 'stack build' then 'stack run' the server launches. Information such as User-Agent, Host, what type of connection is received and logged to the terminal where the server is running, (and in my case a GET / HTTP/1.1 is provided in these logs) which is similar to what the author mentioned above about the basic implementation of a web server is an implementation of the HTTP protocol and a main loop.

The client handler is simple - this is the code responsible for receiving a message, printing it out, and then sending back a "Hello, World!" HTTP response before closing the connection. Some missing features that I am interested in toying around with are some more functionality like serving static web files (HTML, CSS, JS), rate limiting to prevent abuse, and even support for HTTPS. But for now and the scope of this project, I am only tasking this with printing out the HTTP protocol, the response code and a Hello World to let the client know that a connection is established and taken care of. Pretty neat for a small bit of code! I thoroughly enjoyed this project and look forward to continuing it for my learning purposes.

Works Cited

Marlow, Simon. "Developing a High-Performance Web Server in Concurrent Haskell." Journal of Functional Programming, vol. 12, no. 4-5, 2002, pp. 359-374. Cambridge University Press, doi:10.1017/S095679680200432X.