Report - Homework 1

Emil Karlsson

September 29, 2022

1 Introduction

This homework included evaluating benchmarks, discussing multithreading, and improving the web server. Benchmarking was done using an empirical method of testing different scenarios, and discussing the outcome.

2 Benchmarks

Using the supplied benchmarking module we can compile the following result.

2.1 Benchmark 1x10

10 sequential request, from one thread (1x10)

1 thread - delay	10 threads - delay	1 thread - no delay	10 threads - no delay
750 ms	750 ms	50 ms	50 ms

Result

The fact that *1 thread* and *10 threads* reported identical results was expected since the client waited for the result to come back before sending another request. The artificial delay is clearly noticeable, however, it is not unreasonable to expect around 70 ms per request that accumulates to 700 ms over our 10 sequential requests.

2.2 Benchmark 5x10

1 thread - delay	10 threads - delay	1 thread - no delay	10 threads - no delay
3500 ms	800 ms	60 ms	60 ms

Result

As expected, when parallel requests are made to a single threaded server, delays will grow multiplicatively. Having more threads than concurrent requests ensures essentially the same performance as in the first bench mark, eg. 10 threads handling 5 concurrent requests is fine. Again, the artificial delays are reasonable, but at these few requests, the parsing overhead is barely noticeable.

2.3 Benchmark 20x10

1 thread - delay	10 threads - delay	1 thread - no delay	10 threads - no delay
30 000 ms	1700 ms	600 ms	400 ms

Result

This benchmark shows clear benefits of a multithreaded server. When the amount of requests scales up, a single threaded server is provably not sufficient. However, the fact that *10 threads - delay* took twice as long as the previous benchmark shows a weakness of a multithreaded server, namely: if the concurrent requests are more than the available thread pool size, single threaded server issues arise. Further, when handling these many requests, it is now obvious that the parsing overhead is no longer negligible. Even more interesting is the fact *10 threads - no delay* without delays still struggled in relation to *1 thread - no delay.* This is probably best explained by the overhead of scheduling.

3 Evaluation

3.1 Multithreading

This server, as implemented from the given PDF, lacks support for concurrent processing. When handling requests coming from many users simultaneously, delays and even timeouts can occur when the server is unable to handle the incoming request. While this problem is always present, multithreaded or not, certain issues arise when a server is single threaded.

As we can see in the code, the socket waits for an incoming request, and handle the request in some fashion. However, in this single threaded server, while it is processing and handling this request, it is unable to accept new incoming requests on the socket - it is simply busy doing something else. Moreover, this problem is even more obvious when external actors are added, such as another API or a database, as this creates inevitable delays that the single threaded server is not in control of.

A multithreaded server could be implemented as a pool of workers waiting to be assigned a task. Contrary to a single threaded server, a request can be processed, queries can be made to a database etc., all while new requests are coming in. This parallel nature enables the server to serve more clients, since every request is essentially independent for one another.

Practical implementation

Implementing this kind of multithreading in Erlang is a simple task. Using spawn and a dedicated function for listening to a socket, it boils down calling the spawn function multiple times.

3.2 Deliver Files

As mentioned in the PDF, a web server is not much of a web server if it cannot server files to the client, thus, a major improvement includes adding this feature. The idea of how this is implemented is simple (albeit not the most efficient), and it is demonstrated below.

The URI includes a path to the requested resources, and this path is just relative to a folder called webroot. If the file exists, such as index.html return it in the response with status code 200. If it does not exist, return a response with status code 404. It is also convenient to have the index.html be returned when URI is empty; "/". Moreover, since, in this specific case, the web server acts both as a frontend and a backend, it is useful to distinguish between file requests and other requests, such as a REST call. This can be naivley implemented as inspecting the suffix of the request, eg. lists:suffix(".html", URI).

Practical implementation

In order to extend the base functionality, a new module rudyfile was created. This was meant to encapsulate a single function called get_file_from_uri(URI). The order it was going to fetch the file was:

- Check if file exists
- Load file content
- Determine file type
- Determine http content header

Checking if a file exists, loading its content and size are trivial operations in Erlang. However, determining the filetype and using it to set the content header requires more thought. In a http response, the Content-Type header tells the receiver how to interpret the data sent. For example a browser requires Content-Type: text/html in order to interpret the data as a renderable html page. This meant that filetype are translated to Content-Header, for example: *.html -¿ text/html, *.css-¿text/css. A base case of text/plain is useful too, if an unknown filetype exists in the webroot and is requested.