Report 1: Rudy: a small web server

Nils Blomgren
September 11, 2023

1 Introduction

We have been introduced to erlang, a language that I haven't been working with before. Prior to this homework I did the non mandatory assignment, which was very helpful to practice syntax and get used to the language. The goal of this homework was to create a simple web server. The program consists of a HTTP parser file which handles the creation of HTTP requests. It constructs the request and handles the sending to the client. The main program can be found in the rudy file. Here we have functions for starting the programming and then making the application listen on a specific port. Furthermore we have functions for handling an incoming request and replying to the request. I also did the additional task of implementing a functionality for delivering file. This was pretty straight forward. If a specific file is requested the program will try to find the file and return it. But if the server does not have the requested file the request will be valid but “No such file with given name” will be displayed to the user. The program also returns some response header which includes the type, the content length and the encoding.

2 Main problems and solution

The program consists of three different modules.

2.1 Rudy

This is the main part of the program which has exported functions: a start function and a stop function. These are used to start the server on a process, register it and then make it listen on a specific port. Incoming request are being handled in

2.2 HTTP

This module handles a HTTP request and parses it. As it is implemented today the program can only handle GET requests, which can be seen in request_line function. I had to add another function for handling the construction of a HTTP response for the file server task. This function simply constructs a response header that says that the request was OK and returns more information then the response function we were given.

2.3 Test

This function runs on another process and is simulating 100 different calls to the server on a specific URI. With this we can test how long it takes for the server to handle 100 requests and also measure the amount of time it takes for the application to handle these.

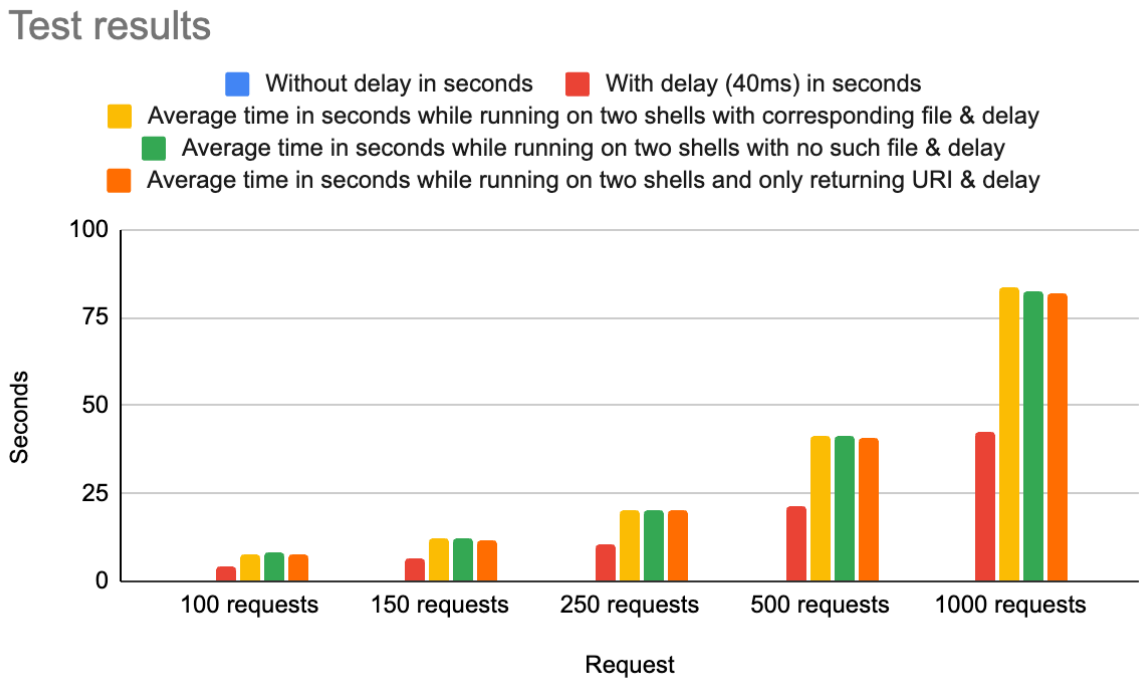
2.4 Problems

Because most of the code was already given to us there weren't any big issues with writing and compiling the code for this homework. But I still spent quite a lot of time reading the documentations to get a better understanding of how some of the functions were working. One thing that I had an issue with was the fact that I didn't have a favicon in the beginning. Because of that, my program was crashing after the first request was sent by the browser, on the other hand this was simple to solve by actually adding a favicon. But I didn't look into if there is any other way of solving this, which there probably is.

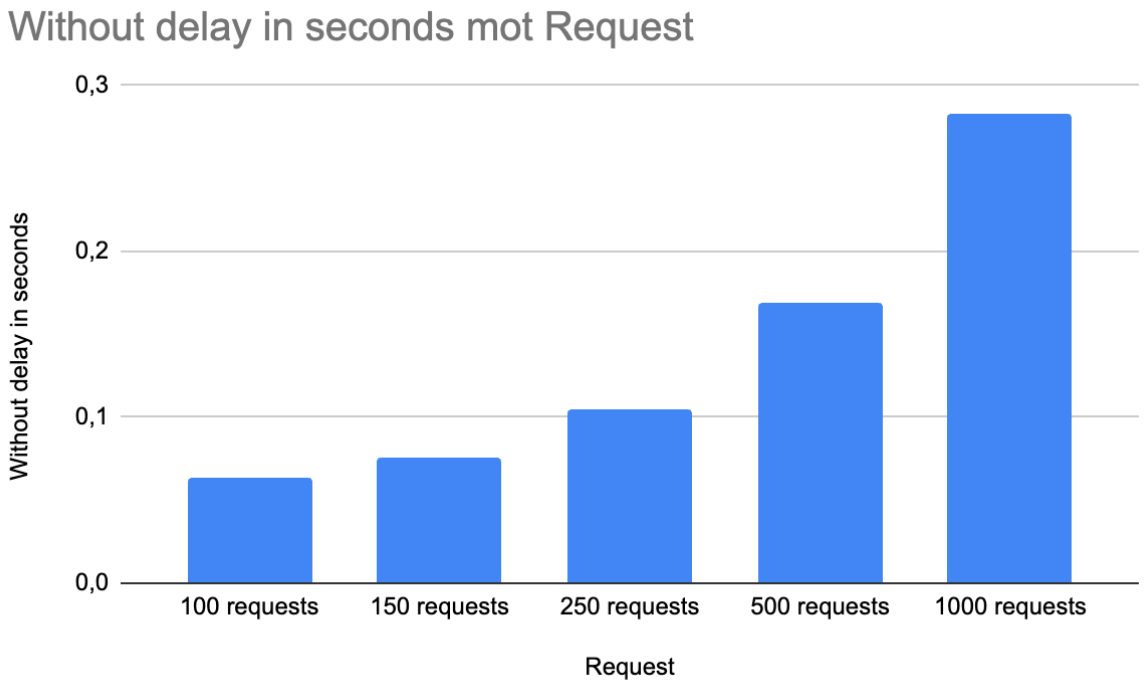
I did the delivering files task which of course required a bit more of me then the rest of the assignment. I picked this one because it sounded the most fun. I had some struggles with creating a HTTP response header. The actual creation of the response header wasn't the tricky part but knowing what to add was hard. I'm still not sure if I did the contentEncoding correctly, because now it is simply hardcoded with an encoding. Also this part of the homework was trickier because I had to understand the syntax better to actually be able to write the functions needed.

3 Evaluating

Using my test file I was able to simulate a lot of requests coming in and how my server handled them with no delay or with a simulated slow server. The results of the test I ran makes sense. Returning only the given URI without response headers is always faster then returning a specified file and constructing response header for the reply.
Note that I was always using a delay of 40 ms in every test that has a delay mentioned.



As seen above the result of running requests without delay is a lot faster and not even visible in the graph above. Below are the results for “Without delay in seconds” displayed by themselves.



Requests	Without delay in seconds	With delay (40ms) in seconds	Average time in seconds (ATiS) while running on two shells returning a file & delay	ATiS while running on two shells with no such file & delay	ATiS while running on two shells and only returning URI & delay
100	0,063102	4,2605	7,9275675	8,0393035	7,5239245
150	0,075852	6,376138	12,1092445	12,088382	11,90771
250	0,104956	10,623869	20,4702435	20,3112435	20,2715895
500	0,16842	21,234843	41,5996335	41,111288	40,9045965
1000	0,282905	42,399315	83,366751	82,6418255	81,696475

3 Conclusion

From the test we can see that the more requests the longer it takes to handle them, which is in line with what I believed would happen before running the tests. The application can handle txt and html files with the current implementation. To add more file compatibility you simply have to add them to the switch case statement.

One improvement to make the total response time faster is to make the application handle requests concurrently. That would make a significant difference.

As always more testing should have been done to more exactly determine the average response time for each operation.

More improvement could be done with error handling, sometimes my server kills itself after a while and the client has no way of being noticed of that with the current implementation.