Report Homework 1

Riccardo Fragale

September 8, 2025

1 Introduction

The aim of this homework is to implement a small web server in Erlang. This server acts as an HTTP parser; it receives certain HTTP message (through socket API), parses them and produces a reply. The server should be able to stay active after parsing each arriving HTTP message. It is also asked to do some tests on the performances and robustness of this implementations but also to try to highlight possible improvements or extra features.

2 Main problems and solutions

The server, called *rudy*, opens a listening socket, waits for an incoming request, handles it, delivers a reply and then waits for new messages. The whole procedure is implemented using methods contained in the **gen_tcp** library. The server should run until it is manually terminated using a function called **stop()**. Since you don't want to terminate the Erlang shell while terminating the server, we are running the process as a separate shell and then register this process under a name in order to kill it properly. We also have to consider the fact that the server is not able to deal with errors and exceptions while receiving a request, so it's very rudimental. A **test** module was developed and it is responsible of verifying some aspects related both to performances and robustness such as the number of requests the server is able to serve per second. Another aspect to consider is that the benchmark is run on a single machine, what if I do it on multiple machines simultaneously? A third important detail is that, we introduce an artifical delay of 40ms while handling the request to simulate file handling, server-side scripting etc. In my opinion this delay is quite significant, as I expect a parsing overhead of few milliseconds, but I will run all the tests to verify my expectations.

3 Evaluation

I did a first run of the benchmark to verify roughly how many request it is able to do in 1 second. I found out that in **1049400** μs (roughly **1.04** s) it serves **25** requests. This value running the benchmark on a single machine. Then, to analyze the effect of our artificial delay I decided to reduce it first to $20ms$ and then erase it totally.

Delay	Number of requests/second(roughly)
$40ms$	25
$20ms$	50
$0ms$	7300

Table 1: The effect of artificial delay

This behaviour shows first that our artificial delay is reducing a lot the throughput of the server as, simply halving the delay causes a 100% increase on the number of requests served per second. What sounds particular to me is the exponential increase when we eliminate our delay. It turns down that the server deals with almost 7300 requests per seconds, showing that the HTTP parsing procedure is really fast, in the order of μs. This confirms my expectations.

Regarding the effect of sending the HTTP messages from multiple machines, it is quite clear from the graph below that the server is able to react only to a message at a time(as I expected) and so its throughtput is the same.

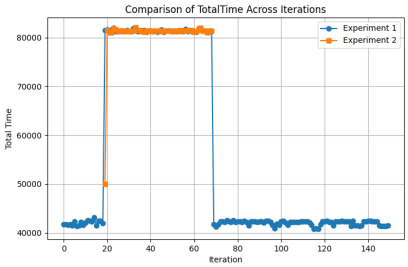

Figure 1: Comparison of TotalParsingTime while two machines are sending requests to the server

Clients have to wait more when multiple machines are interacting from the server. In our case the time for a single reply is almost doubling, this means that with 10 machines (as example) it can increase to 10x times. This is a big issue and a first solution might be to open a new process for each incoming request to "parallelize" the handling of the requests. This can be a good idea, but we need to correctly close the newly created process after

they serve the HTTP message. Another useful solution might be to create a pool of handlers to which assign the requests when they are arriving in order to reduce the average waiting time.

4 Optional task: Increasing throughput

I decided to implement this task as guaranteeing a multi-threading functionality is essential for this server to have acceptable performances. I created for this purpose a module called **rudy_pool** which maintains many parts of the initial **rudy**. The main difference is that the server now creates a pool of handlers (the number of which is decided by the user through a parameter of the function **start**. This handlers are selected with a *round robin approach*. There is a function called **accept_loop** that assigns each incoming request to a handler. After assigning the job of parsing the message accept_loop the selected handler is moved to the end of the queue. The queue is realized through a simple list. The following graph shows that when two processes are sending requests to the server the time needed for a reply is not changing significant nor doubling as in the original server.

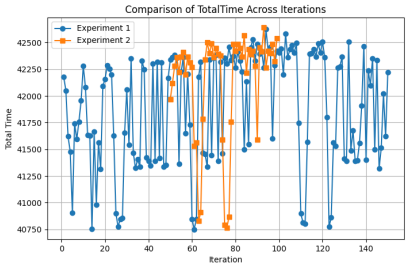

Figure 2: Comparison of TotalTime across two experiments.

5 Conclusions

I have definitely improved the initial server by implementing the multi-threading feature but there are still possible improvements to be done. First of all, there is no error or exceptions handling when receiving a message. Moreover, sockets are left open when things go wrong which is real issue for a server like this operating online. A second interesting aspect is that real HTTP request might not be finished at the first line. We might improve by adding new features to the parsing procedure so that even complex HTTP messages are handled by rudy. I found this homework very useful to understand the concept of sockets and to start thinking about distributing a service that we are giving to a user into multiple machines or processes that are running together and improving the performances by splitting the job.