Report 1: Rudy

Alexandru Matcov

September 11, 2024

1 Introduction

This seminar covers the implementation of a small web server in Erlang. The aim of the assignment is to comprehend:

- procedures for using a socket API;
- structure of a server process;
- HTTP protocol.

Also, this seminar provides a good first challenge for testing the knowledge gathered in Erlang from the "Getting Started" tutorial.

2 Main problems and solutions

The main problem presented in this seminar was implementing a small web server that is capable of handling multiple client requests either sequentially or in parallel. Each implementation is discussed separately in the subsequent subsections.

In order for the server to open up a connection and listen to the client requests, socket API was used with the implementation of the **gen_tcp** library for establishing TCP connection. The most important functions are:

- `gen_tcp:listen(Port, Opt)`
- `gen_tcp:accept(Listen)`

The former implies the opening of a new listening socket and the latter the action of listening to the opened socket.

2.1 Sequential Server

The idea of the basic implementation of a sequential server is to wait for a request, serve it, and wait for the next. This was easily achieved by implementing a *handler()* function that recursively calls itself to handle the next client request in the queue. Thus, the size of the queue increases linearly with the number of client requests.

2.2 Concurrent Server

A solution to increase the throughput of the server is to have each request handled concurrently. A possible solution would be to create a new process for each incoming request. However, this puts a high load on the server with the increase of client requests. With thousands of client requests, the server most likely would run out of available hardware resources to create new processes to handle the requests and eventually crash.

A better solution would be to spawn a predefined number of handlers and assign the incoming requests to the available handlers. This would limit the load put on the server by concurrently handling a number of requests. In case there are no available handlers, the requests would be simply put on hold in a queue until a handler becomes available again.

3 Evaluation

3.1 Sequential Implementation

The implementation of a sequential server is done by handling all incoming client requests by a single process. This creates a bottleneck for handling multiple clients.

In Figure 1 are presented the results of the response time of the sequential server. A linear growth is observed in the plot which is explained by the linear relationship between the number of clients and server process time of incoming requests. A benchmark with the following server-side and client-side parameters was conducted:

- 1 handler
- 1, 2, 4, 10 simultaneous clients
- 100 requests per client
- 40 ms delay per request

From the obtained results, it can be concluded that approx. 25 requests per second can be served with an artificial delay of 40 ms. However, once we remove the artificial delay an improved time of approx. 2000 requests per second can be served. This indicates that even if the injected artificial delay seems insignificant it adds up 0,04 seconds to each request. Serving 100 requests increases the delay by 4 seconds!

3.2 Concurrent Implementation

To test the concurrency of handling the requests in parallel and compare it with the sequential implementation the same benchmark was run, but

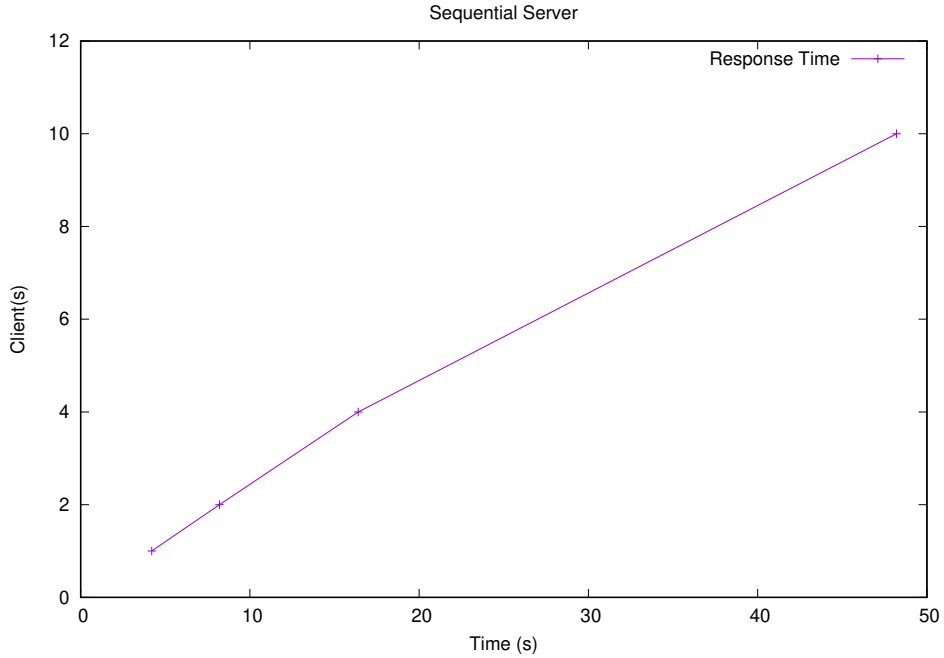

Figure 1: Sequential Server Benchmark

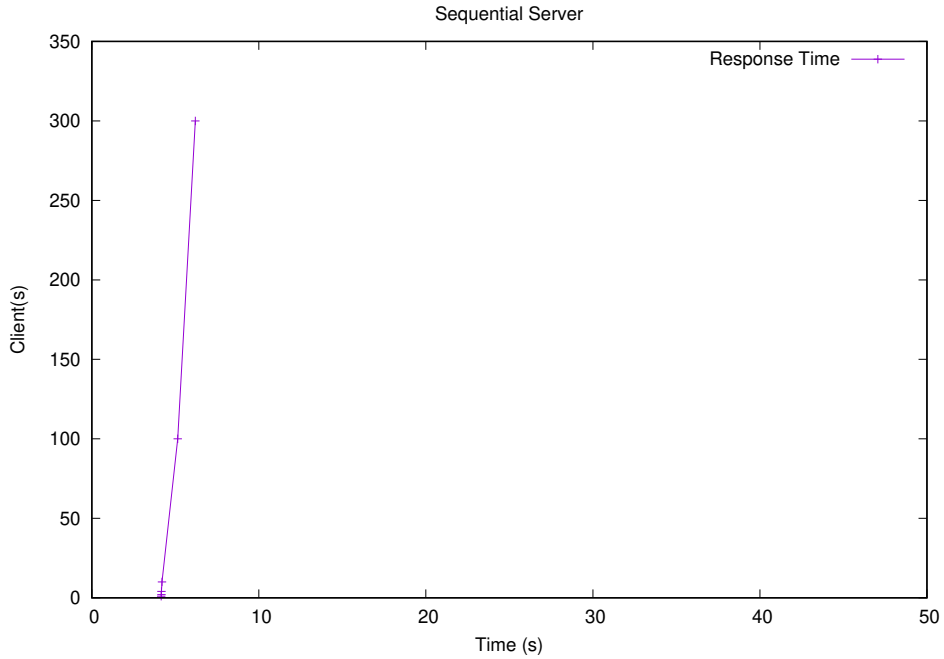

Figure 2: Concurrent Server Benchmark

with the number of handlers matching the number of clients. Thus, for each client we have 1 handler in the concurrent server. Moreover, 2 additional tests for 100 and 300 simultaneous clients was conducted to investigate if it takes time to create new processes .

Figure 2 plots the results of the benchmark. Here we can clearly see an improvement of the processing time of the server with an almost unchanged time for different number of clients. However, with the increasing number of spawned handlers the time also increases insignificantly, demonstrating that it takes time to create a process.

4 Conclusions

In conclusion, this assignment was a nice practical introduction to concurrent process and communication between them with the help of Socket API in Erlang.