Rudy: A small web server

Report by Aksel Uhr

September 2021 Aksel Uhr
Royal Institute of Technology

1. Introduction

The program consists of a small web application, including a server. In the program, the user may send messages (requests) to the waiting server, which in turn replies to the client. To make this communication work, HTTP protocols are used for the communication. A socket (endpoint) is also opened on the server due to the socket api (gen_tcp). By listening to the socket, the server is open for requests from the client. Once the socket is closed, the server will stop listening to requests. Additionally, it is possible to measure the performance of the server in the program.

2. The program and main problems

The program is divided into four different modules described below. The program description may be overlooked as it is mainly for my own learning purpose.

2.1 The http module

This module handles the parsing of a request, namely a GET request which is used to retrieve data. When parsing the request, the program interprets a byte stream according to the incoming http request. The request follows the HTTP protocol standard, which consists of a method (e.g. GET), a path which identifies the resource of the server (e.g. /index.html) and the version of the HTTP protocol (e.g. HTTP/1.1). It may also include headers which tell the server how to interpret the data. The HTTP protocol standard is a blueprint of how communication should take place between two or more parts.

2.2 The reply module

This module is initiating a server with a given port number (input, e.g. 8080) and opens a listening socket.

The module consists of four functions.

1. init(Port) takes a port number, opens a socket for incoming connection. Once listened to, the handler function is called.
2. handler(Listen) listens to the incoming connection. Afterwards it calls the request function. By calling itself again, it allows multiple incoming connections / requests from the client. Otherwise, the connection will be closed.
3. In case of receiving a client request, request(Client) parses the request by using the http:parse_request from the previous module. Afterwards, it we send a reply to the client.
4. In the last function, reply(args), we decide what to reply to the client.

2.3 The run3 module

In this module, we run the server as a separate erlang process. This is useful since we can decide when the server should terminate.

The module consists of two functions.

1. start(Port) takes a port number, registers the process (e.g. the activity/task) and spawns (e.g. initializes) the process by calling the init function from the previous module. Hence, the server is initialized.
2. stop() simply closes the server.

2.4 The benchmark module

The purpose of this module is to provide functionality to measure the time it takes for the server to reply.

The module consists of three functions.

1. bench(Host, Port) which takes the host of the server (e.g. localhost) and the port number. Afterwards, the time it takes to reply to N requests to the server is measured by timing the start and finish time.
2. run(N, Host, Port) takes N requests from the client and sends them to the server. After calling the actual request function, the function calls itself recursively with N-1 as a parameter. This means that if N = 100, the function will call itself 100 times until the base case (N = 0) and hence produce 100 requests to the server.
3. request(Host, Port) connects to the server, sends the server information (e.g. “localhost”, “8080”, Opt) and calls the get function with “foo” being the URI. Then, the function receives the socket information and if succeeded, the server is closed.

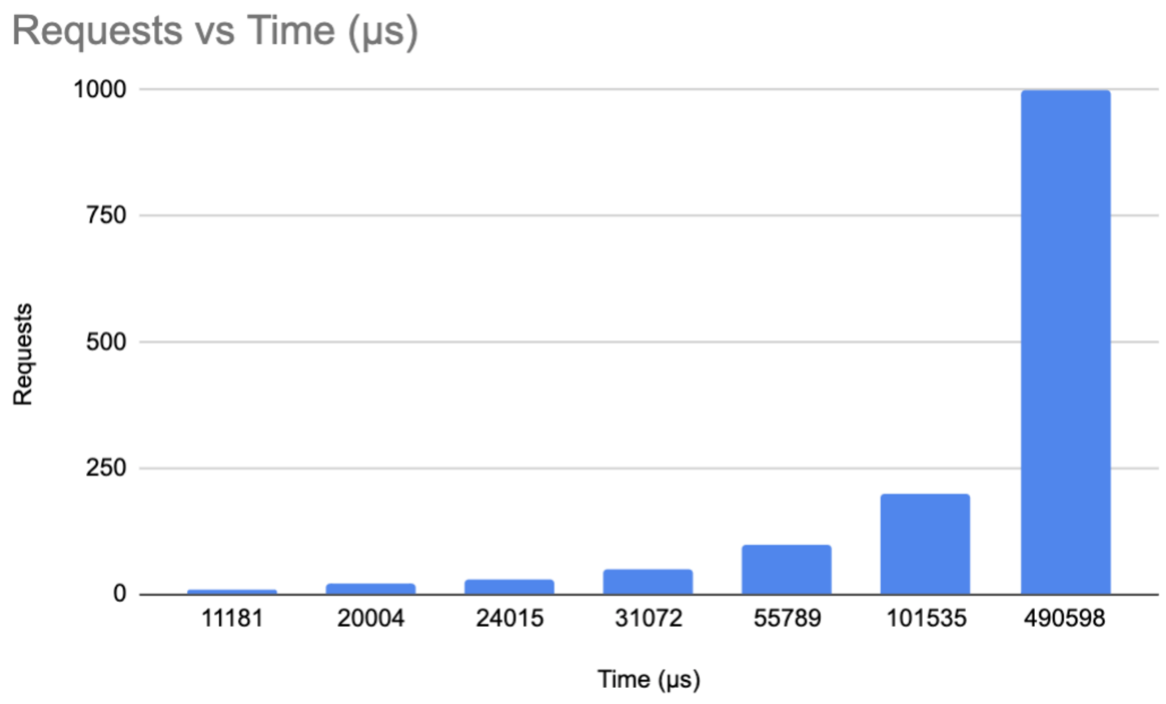
2.5 Main problems

The main problems encountered were interpreting the given code, the functional way of building an application and Erlang’s syntax, although the fundamental idea of client-server communication is somewhat straight-forward.

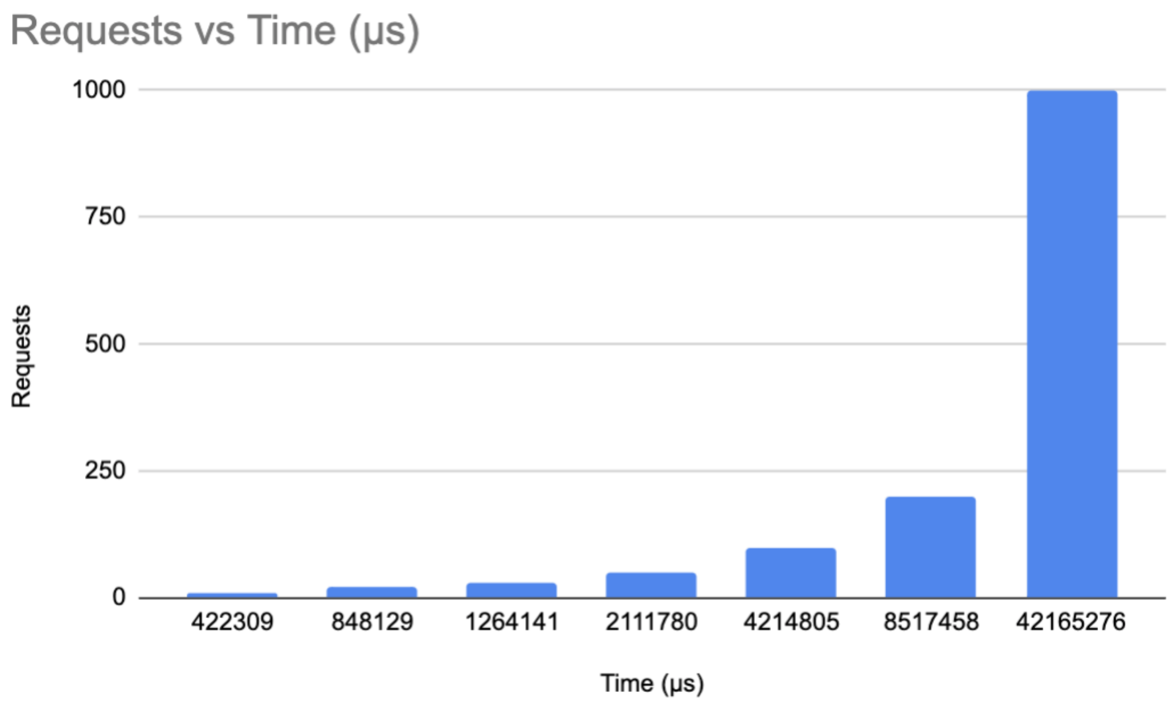
The task of completing the code in task 2 (the first reply) was however quite straightforward. Since we needed to initiate connection to server, listen, request the server and ultimately reply to the client in the following order, it was pretty clear that function calls were missing. As the connection was closed after one request to the server, it made sense to call the handler(Listen) again in order to not close the connection after one request.

3. Benchmark results

A graph of the test is shown below. The diagrams are created in google sheets.



Please note that the following test was done using the artificial delay regarding simulation of a file handling.



4. Conclusion

The more requests sent to the server, the more time it takes for the server to reply to the client.

4.1 Potential improvements

- The benchmark would be more accurate if more tests were conducted and if the mean was calculated, since the time varied:

```
(Rudy@localhost)24> benchmark:bench("localhost", 8080).
431704
(Rudy@localhost)25> benchmark:bench("localhost", 8080).
461826
(Rudy@localhost)26> benchmark:bench("localhost", 8080).
448080
```

- I only used two different shells, not two different machines. It would be interesting to deploy the server in another location on a different machine to see how the distance and network connections affect the time.
- Handling requests concurrently will most definitely increase throughput. By increasing the amount of threads between the server and the client(s), the execution will be much faster and score better time in the execution.
- The HTTP request is now static, and the body is now empty. By figuring out the body length, the program will be able to handle further requests, for example from a form.