Report 1: Rudy - A Small Web Server

Ziheng Zhang

September 10th, 2021

1 Introduction

Communications between different computers depend on transmission protocol, server processes and client processes etc. The web server has significant contributions to this process. In this seminar, I acquired knowledge of Erlang socket API, how the server process works and how to parse HTTP requests. Based on this, I built a small web server with the basic functions of receiving and sending messages.

2 Main problems and solutions

2.1 How TCP Sockets work

Understanding how TCP sockets works is vital in building this small web server. By reading official documentation and blogs, I learnt that the TCP socket works like the Figure 1[1]. Unlike UDP, TCP needs to *listen* to the socket so that it can set sessions up.

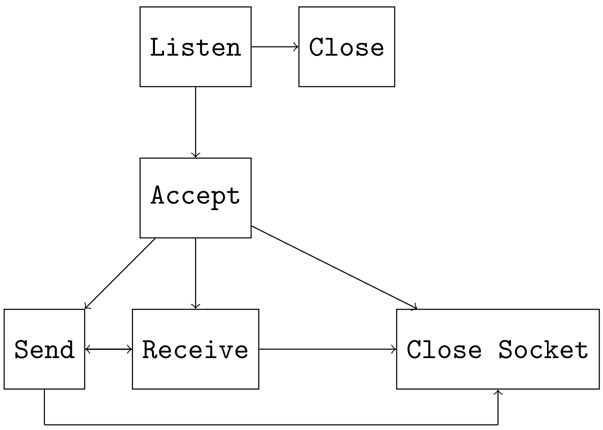

Figure 1: TCP Sockets working flow.

3 Evaluation

3.1 The speed of processing requests (single-process)

The server-side code is in the file `rudy.erl`, and the test code is in the file `test.erl`. First run `rudy.erl` to start the server, and then run `test.erl` for ten times. As shown in Figure 2, the requests' processing time is about $4.6s$ with artificial delay, and $0.04s$ without artificial delay.

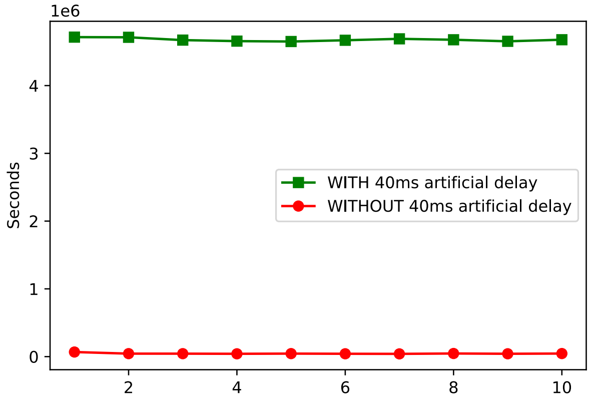

Figure 2: The requests' processing time.

There are 100 requests in `test.erl`, and for each reply of Rudy it has $40ms$ delay. Therefore there are approximately $100 \times 40ms = 4000ms = 4s$ delay. The test results shown in Figure 2 conform to this expectation. This also show that artificial delay has a significant impact on handling requests.

3.2 The speed of processing requests (multi-process)

When using multiple processes, we need to make several code changes, specifically creating processes to send messages in a loop as shown in the following code.

```
test(0) ->
    ok;
test(N) ->
    spawn(fun() -> bench(localhost, 8080) end),
    test(N - 1).
```

When the number of processes created from 1 to 4, the results are shown in Table 1.

Number of processes (with 100 requests)	**Execution time per process (seconds)**
1	4.977358
2	9.650795
	9.701278
3	13.843462
	13.888825
	13.935110
4	18.426471
	18.474291
	18.530509
	18.582323

Table 1: The time required by each request process on the client when **one** process on the server is processing a request.

Take creating 4 process as an example. Each reply has $40ms$ artificial delay, therefore theoretically there should be a delay of approximately $4 \times 40ms \times 100 = 16000ms = 16s$ delay. According to Figure 2, the total time is around $(\approx)4 + (\approx)16 \approx 20s$. Table 1 shows that the experimental results are in the line with expectations.

3.3 Increasing throughput

When there is a large number of requests coming from clients, one process is not enough to handle them. Therefore we can create several processes, specifically handler processes, to serve the requests. I learned about the implementation of creating a handlers pool to deal with requests in example the file `rudy4.erl`, and mimicked the implementation in the file `rudyPool.erl`.

First run `rudyPool.erl` to start the server, and then run `test.erl` to evaluate the performance of Rudy. For $1 \sim 4$ processes, each process sends 100 requests, and the server process pool size is 3, the experimental results are shown in Table 2.

Number of processes (with 100 requests)	**Execution time per process (seconds)**
1	4.674867
2	4.663501
	4.663808
3	4.698214
	4.698419
	4.698726
4	5.564928
	5.981491
	6.073651
	6.489600

Table 2: The amount of time required by each request process on the client side when **four** processes are processing requests simultaneously.

As we can conclude from Table 2, the server's ability to handle requests increases dramatically when multiple processes are processing requests on the server side.

4 Conclusions

Through this project, I learned how to use and build a server with basic functions, knew that the server in the processing of files or server-side scripts has a significant impact on the transfer of messages, knew that created multi-process pool on server can remarkably improve the performance of server and realized that multi-process requests have higher requirements for the server to process information.

References

[1] Learn You Some Erlang. *TCP Sockets*. URL: `https://learnyousomeerlang.com/buckets-of-sockets#tcp-sockets`.