Report 1: Rudy: a small web server

Justas Dautaras

September 5, 2019

1 Introduction

In the first task a small web server is built with Erlang to understand the main principles of distributed systems. Erlang is a functional programming language used to build massive scalable real-time systems. To understand how it works (server processes, HTTP protocol, tcp communication) a small web server is built and it's capabilities are tested with specific performance measurement.

Main topic of this report is concurrent processes and what effect it has to server performance. Furthermore, it is important to understand these characteristics, since most of the systems nowadays are distributed and it is one of the solutions to supply the high demand of various online services and their requirments on high availability. In the end, one possible improvement is discussed.

2 Main problems and solutions

One of the main problems that distributed systems try to solve over the centralised ones are performance issues or in other words - throughput of data. That is, how many requests per second the system can serve. Although with small amounts of data this should not be a problem, but with millions of users it can cause a lot of problems. That's why concurrent processes are used to handle many different users requests independently. In this paper we compare concurrent processes influence over a single processor.

3 Evaluation

Evaluation is done by running a number of performance tests. In total, six tests are carried out. First, three to show the difference between having 1, 2 and 10 concurrent request handlers with a simulation of 5 users with 10 requests each. Secondly, three tests of 1, 2 and 10 concurrent processes with a simulation of 1 user with 50 requests. It is important to notice that a 40 ms timer is used in the server when handling requests to simulate real world use case, as well as that the server and users are both run locally, meaning that there is no latency.

Processes	Time (ms)
1	2343
2	1171
10	485

Table 1: Results of using different numbers of concurrent processes for handling of 5 users with 10 requests each

Processes	Time (ms)
1	2344
2	2344
10	2344

Table 2: Results of using different numbers of concurrent processes for handling 1 user with 50 requests

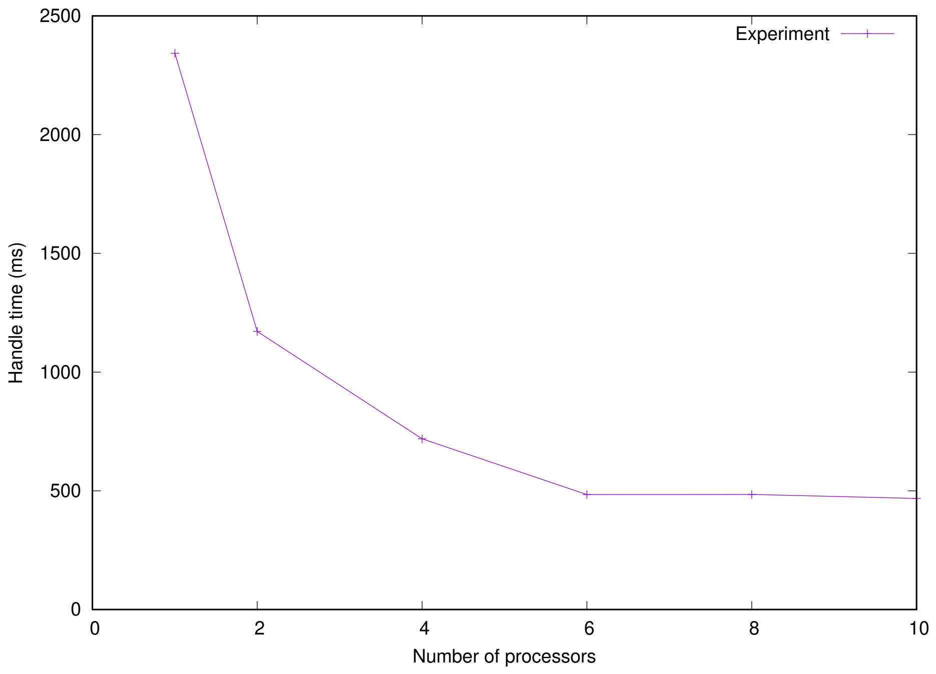

Figure 1: Table 1 expanded results

Figure 1 shows that distributing a system can increase throughput for high number of users, but to increase performance for single users, internal request processing changes need to be made. Furthermore, from the tables it shows that 1 process handles 5 users with 10 requests in same time as 1 processor handles 1 user with 50 requests. From this example, it is clear that every request is handled in a row, one by one.

Right away we can see one of the improvements that could be made to this system. And that is multi threading. Concurrent processes can solve the problem of many users using the system, whereas multi-threading would solve the single users requests handling speed.

Another important part is that we have a 40 ms simulation delay and the messages are in a queue, that means that we can expect to handle a single request, if it is not delayed because of other users, in 40 ms with a minor overhead allocated for executing the benchmarks. If let's say two users try to do this on the same single processor, the time would double. So, in a perfect world, we would expect to handle 25 messages per second (1000 / 40) on a single processor.

4 Conclusions

Solving this task has taught me the basics of functional programming and helped to understand underlying network characteristics and the fundamentals of distributed systems: how socket API's, server processes, handling of HTTP requests work and how all of this effects performance, latency, avalibility, etc.