Report 1: Rudy: a small web server

Nikita Gureev

September 7, 2016

1 Introduction

In this homework a simple web server that can handle GET requests was implemented in Erlang. Also an improvement of concurrent request processing was implemented, as it is the most important aspect in respect to distributed systems.

2 Main problems and solutions

The main problems that needed to be resolved were the handling of only one request and then shutting down for the server, which was resolved by having a loop that handled the incoming requests. Another challenge was the sequential execution of the server code, which was first solved by spawning new processes for every new request and then by using a pool of processes to handle the requests. All of the results were measured and documented.

3 Evaluation

The first benchmarks will be run against the simple sequential server, the results are presented in the table below.

	Trial 1	Trial 2	Trial 3	Trial 4	Average
Time, seconds	4.8	4.7	4.7	4.7	4.7
Requests per second	20.9	21.3	21.3	21.3	21.3
Time without the delay, seconds	0.203	0.187	0.172	0.14	0.1755
Requests per second without the delay	492.6	534.8	581.4	714.3	569.8

Table 1: Sequential server, with and without the delay

As can be seen from the table, the number of requests served differs significantly, when the delay is removed, as the average number of requests

	Trial 1	Trial 2	Trial 3	Trial 4	Average
Time to process 100 requests with 2 concurrent sources, sec	9.5	7.4	8.7	8.9	8.6
Requests per second	10.5	13.5	11.5	11.3	11.6
Time to process 100 requests with 4 concurrent sources, sec	17.6	17.5	17.5	17.4	17.5
Requests per second	5.7	5.7	5.7	5.8	5.7

Table 2: Sequential server, with parallel requests

served increases substantially. The parsing time is insignificant compared to the time spent during the call to timer:sleep.

With the introduction of the second process the rate of request processing requests almost halves. With the introduction of two additional processes the time to process 100 requests halves again, which means that as the time is reversely proportional to the number of processes that access the system. Essentially, our system does not utilize concurrent processing of requests, as each request is sequentially processed. Let's introduce a naive implementation of concurrent handling with a new process being spawned every time a request is received.

	Trial 1	Trial 2	Trial 3	Trial 4	Average
Time, seconds	4.81	4.75	4.75	4.72	4.76
Requests per second	20.8	21.1	21.1	21.2	21.1

Table 3: Naive concurrent server server

As can be seen from the table, the handling of requests from one source has become slower, on average by 200 milliseconds. The most probable reason for that is overhead from the spawning new processes for every request. Now, let's examine the effect on handling requests from multiple sources.

	Trial 1	Trial 2	Trial 3	Trial 4	Average
Time to process 100 requests with 2 concurrent sources, sec	4.9	4.75	4.75	4.72	4.76
Requests per second	20.8	21.1	21.1	21.2	21
Time to process 100 requests with 4 concurrent sources, sec	4.9	4.7	4.8	4.8	4.82
Requests per second,	20.5	21	20.7	20.7	20.8

Table 4: Naive concurrent server, with parallel requests

The effect is immediately visible, as now the time to handle 100 requests from multiple sources is much closer to the result for a single source. However, the drop in performance is still detectable, as the rate of handling

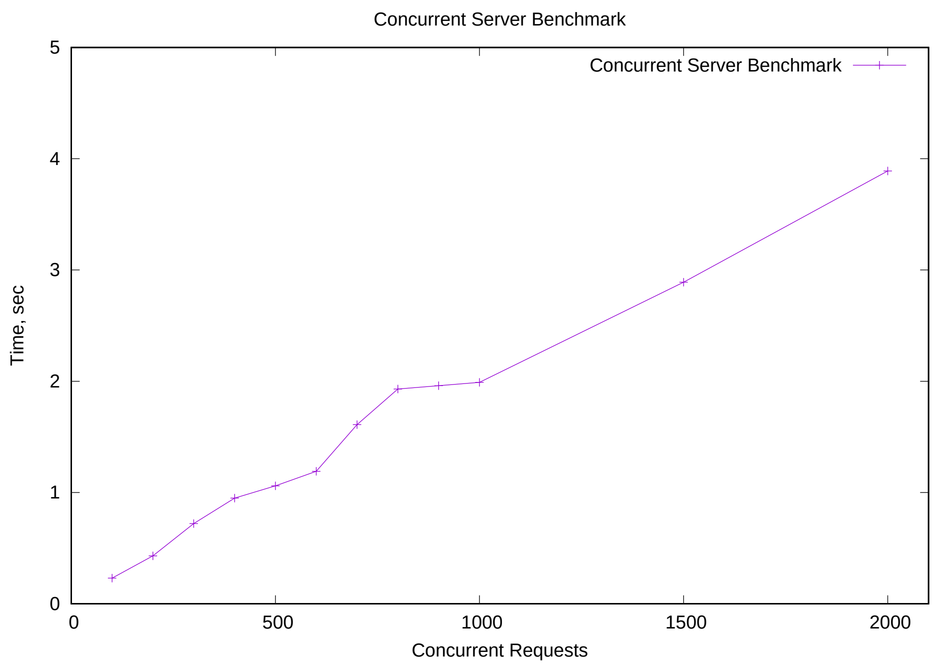

Figure 1: Concurrent server benchmark, pool consists of 50 handlers

decreased on average by 0,34 in the trial with four separate sources in comparison with the trial with only one source. A different approach would be to initialize a pool of handlers and then assign a handler from that pool. The results show an improvement over the previous build, with substantial increase in processing rate, as can be seen from the plot.

4 Conclusions

In conclusion, the most efficient implementation was the concurrent one with a pool of handlers. It was superior to the one with the spawning new processes, as it was more efficient, and could handle large spikes of incoming requests without the risk of having too many processes at the same time. It was far more superior to the sequential server in efficiency, as is evident from tables.