Report 1: Rudy

Thomas Galliker

September 7, 2011

1 Introduction

In this exercise we were asked to build a small web server based on the functional programming language Erlang. This web server was used to demonstrate the easy handling of TCP sockets in Erlang. Measurements of the response time of http requests shall be done and expectably lead to discussions about the performance of single and multi threaded processes.

There were three major parts to be analyzed and solved in this exercise: A http parser (http.erl) with a http service (rudy.erl) as well as a testing routine (testrudy.erl). To my surprise, the most confusing code sequences were found in the http parser. There are some tricky pattern matching functions implemented to split up http GET requests. It is of obvious advantage to have good knowledge of the data structure of http requests and responses.

2 Main problems and solutions

The most time-consuming problem remaind in the behavior of how TCP connections are terminated. If a client closes a TCP session, this session is actually not immediately removed after "gen_cp:close" is executed. There is a certain period of time (depending on the operation system) that helps to make sure, no belated/misrouted TCP segments are accidentally forwarded to a wrong process. Microsoft Windows provides 1024 user-addressable TCP ports per process by default. The WAIT_CLOSE period on Windows is approximately 30 seconds. This behavior was mainly responsible for these kinds of "system_limit" errors:

```
1>rudytest:bench("localhost", 80, 5000).
    ** exception exit: badmatch,error,system_limit
```

Windows users can issue the command "netstat -a" to se TCP connection states:

```
TCP 127.0.0.1:9684 activate:http CLOSE_WAIT
TCP 127.0.0.1:9685 activate:http CLOSE_WAIT
TCP 127.0.0.1:9686 activate:http CLOSE_WAIT
...
```

There are two possibilities to clean up Wait-Close connection: Either you wait until the operating system removes them or you make use of an utility (e.g. CurrPorts on Windows). To allow fair testing conditions, Close-Wait connections must be purged before each test run.

3 Tests & Evaluation

To check the performance of this web server, I have specified several test scenarios. Since my testing environment is not absolutely dedicated to the http server process itself, I have performed up to five independent test runs on each test case.

- **Test Scenario 1:** Run rudytest against rudy (the single process web server) resp. rudycon (the multi-worker web server) on localhost with different loads (1000 resp. 5000) of http GET requests. On rudycon, this test was repeated several times: Once with 1, 10, 100 and finally with 1000 parallel workers queues. The aim of this test is to find out the number of worker queues that are still able to work efficiently on a multi-core system. My assumption was that the number of workers depends on the number of physically available cpu cores - and, that more than 10 workers will won't be able to handle huge amount of requests in an efficient way (due to the administrative overhead of the process scheduler).

  Conclusions for Test 1:

  - Serialized processing (using rudy) takes dramatically more time. For some reasons a single "spawned" process was almost 3 times faster than the single process.
  - Much time is wasted for console output.
  - My assumption was not confirmed: 1000 workers seem to be as efficient as 10 workers. The reason for this seems to lay in the behavior of Erland that does not create a real thread on the operating system for each spawned process.

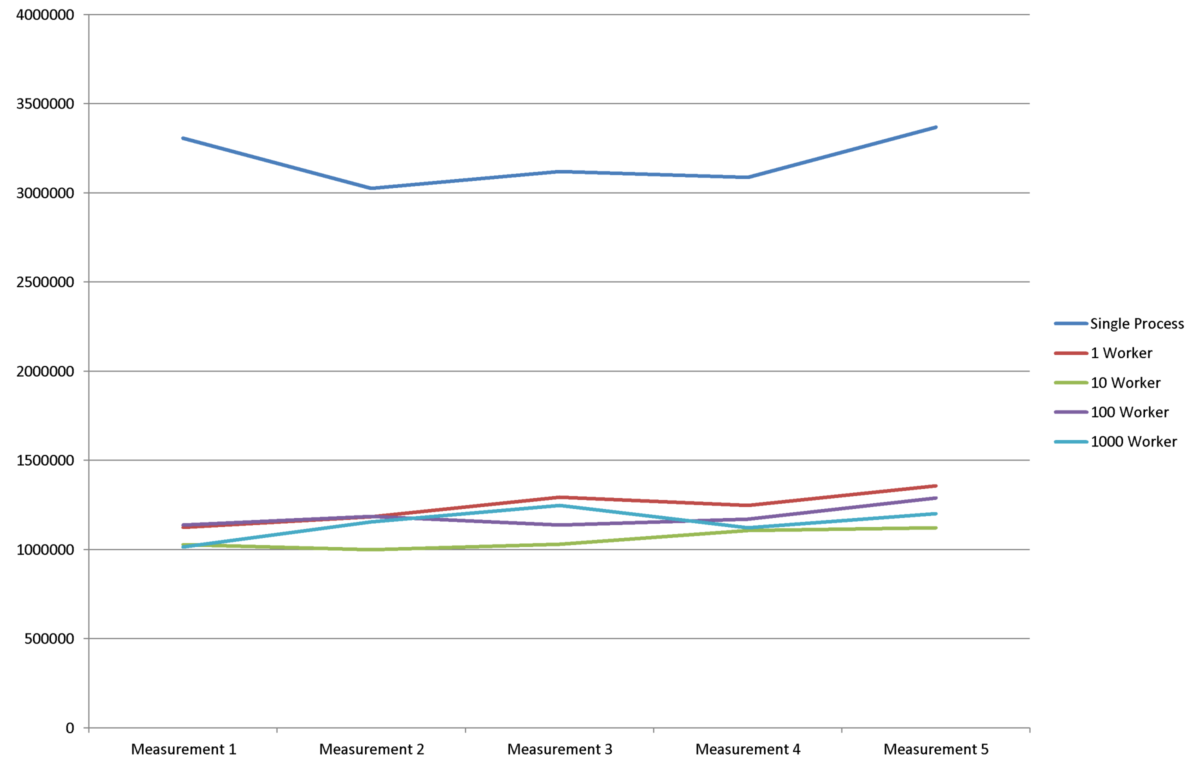

Figure 1: Measurements of test 1 with 1000 successive http requests. (Values in microseconds).

In figure 1 and 2 you can find the visualized outcomes of the test measurements of test scenario 1. As you can see, there is no significant deviation between the rudycon tests with different numbers of worker queues. Only the non-parallelized process rudy took by far more execution time. Interesting to note is, that most tests had shorter execution times at the beginning rather than at the end of the test. This was probably a side effect of Wait-Close connection removal (see above).

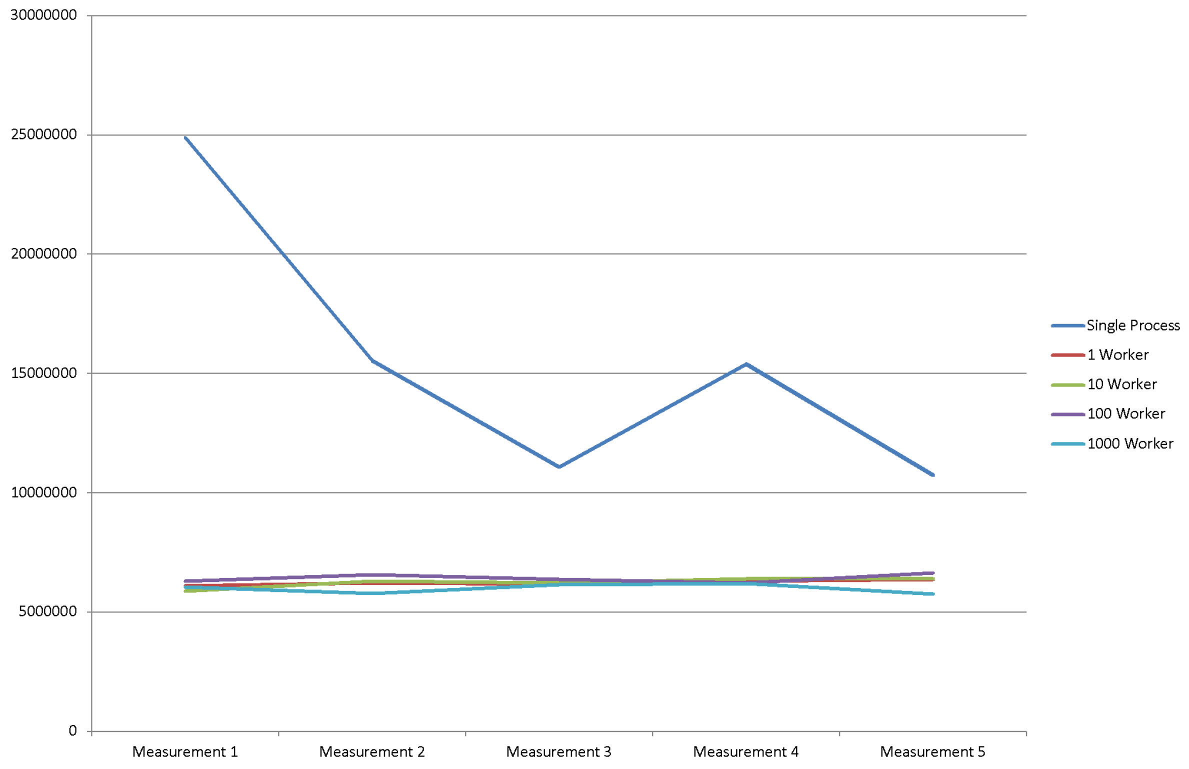

Figure 2: Measurements of test 2 with 5000 successive http requests. (Values in microseconds).

- **Test Scenario 2:** Scenario 1 nicely shows that there is an essential performance gain with the introduction of spawned worker queues. But there is no significant difference between 1 and 1000 workers in terms of performance. This made me wonder why... Could the problem maybe remain in the way we request rather than in the way we respond http messages? I tried to rewrite the rudy test script "rudytest" to handle requests and responses in an asynchronous fashion.

  Conclusions on Test 2: The script works. Not perfectly, but it does what it is supposed to do. Thus, the test cases of scenario 1 were repeated - but again, without success. Sequentially requested messages arrived much earlier than the parallelized ones. Bad luck and hours for nothing?
  There are very rare cases where these kind of errors occurred with the parallelized version of rudytest:
  =ERROR REPORT==== 6-Sep-2011::19:15:47 === Error in process <0.846.0> with exit value: badmatch,error,econnrefused,[rudytest,request,5]

- **Test Scenario 3:** In a further scenario I tried to run a simple multi-node test. Two rudytest nodes were used to issue 5000 requests each, again in a serial manner.

```
erl -sname bench1 -env ERL_MAX_PORTS 5030 +P 250000
rudytest:bench("localhost", 80, 5000).
```

```
erl -sname bench2 -env ERL_MAX_PORTS 5030 +P 250000
rudytest:bench("localhost", 80, 5000).
```

Conclusions on Test 3: Unfortunately there was definitely not enough time to build a fair launch pad for this test. I started them manually; one after the other. The processes with 2x 5000 requests didn't take more time to finish than one single console with 1x10000 requests. The reason for this is maybe in the very short processing time of http requests and responses. Parallel execution is growing priority if processes waste much time by waiting for other processes and/or resources. Those blocked processes must not block the rest of the system!

4 Conclusions

It has cost me some (!) hours to understand every single detail of the given code fragments but I think it was worth spending it. The implementation of this web server was a good exercise to get to know Erlang a bit better. It would be helpful for me to discuss the open unclarities of the tests with the worker queues.

5 Links

http://www.erlang.org/doc/man/gen_tcp.html
http://www.nirsoft.net/utils/cports.html
http://forum.trapexit.org/viewtopic.php?p=38717sid=4410c9dd63bbe6c05edc9ade4e92edf2
http://msdn.microsoft.com/en-us/library/ms819739.aspx
http://stackoverflow.com/questions/337115/setting-time-wait-tcp
http://www.thomasgalliker.ch