Report 1: Rudy: a small web server

Corentin LEROY

September 11, 2018

1 Introduction

The goal of this exercise was to implement a simple web server. In this application I created three programs, a HTTP parser, a server handling client requests, and a benchmark to send requests. All of this was coded in Erlang language, using the *gen_tcp* API to manipulate sockets with TCP.

The main objectives were to understand the structure of a server process, have basic knowledge of the HTTP protocol, practice with the functional language Erlang and familiarize with its characteristics and with the use of a socket API. This simple two-tier architecture is a good introduction to distributed systems.

2 Main problems and solutions

The main difficulty of this task was to understand Erlang's philosophy and subtleties. Especially for multi-threaded programs and TCP sockets. The fact that multiple processes can listen to a single socket is a bit destabilising when you are not used to it, but it is very useful, because it can increase the efficiency of the server, as we will see in the Evaluation section.

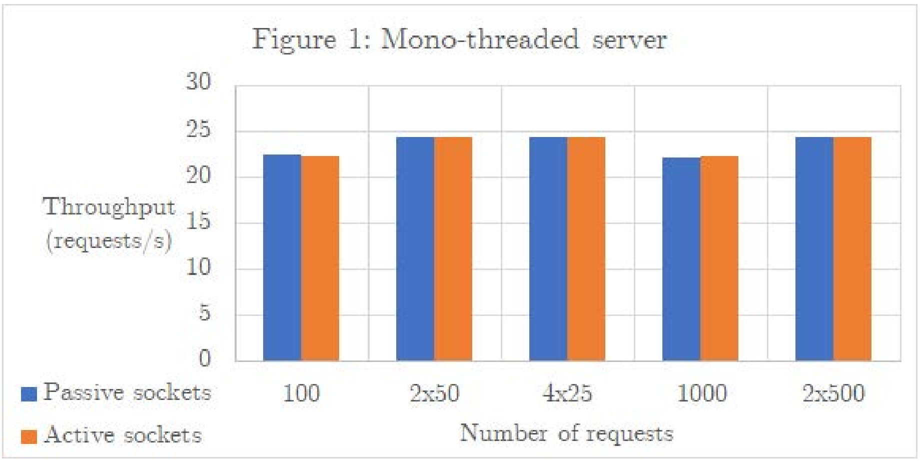
A big part of the work was reading Erlang documentation, particularly for sockets modes. I created a server using only active TCP sockets and I had to understand how to use them. However this implementation did not have an effect on the server's speed as shown in the next section.

One other problem was for me to figure out how to end the program properly and to avoid having TCP sockets errors when closing it. Each time I stopped the *rudy* process by calling the `stop()` function, I had errors informing that the sockets were closed. I fixed this by checking, when we have this error in the handler function, if the controlling process has received a *stop* message. If this is the case, we make sure the socket is closed and end the program without reaching the print of the error reason.

3 Evaluation

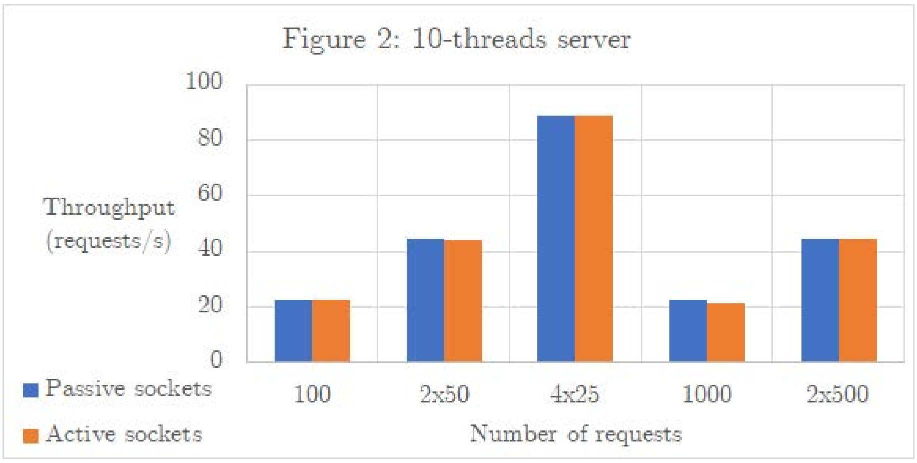
I tested Rudy servers in 5 scenarios: 100 requests from a mono-threaded client, 2x50 requests from a 2-threads client, 4x25 requests from a 4-threads client, 1000 requests from a mono-threaded client, and 2x500 requests from a 2-threads client. For each case, I used a mono-threaded server, a 10-threads server via *rudy4*, and mono-threaded and 10-threads servers with active TCP sockets via *activeRudy*. Each server has a 40 ms delay in its reply method to simulate file handling, server side scripting, etc. I ran the servers on my machine and used the KTH SSH tool to run the clients. All the values found here are an average of at least 5 experiences each.

1. Mono-threaded servers



The first thing we can notice is that the throughput does not vary a lot (less than 10%) with the different scenarios (figure 1). The throughput is slightly bigger with multi-threaded clients. I suppose this tiny difference is due to the greater speed on the client side with multi-threading, because requests are treated in the same way by the server: they are put in a single queue before being handled. We can also see that the throughput does not depend on the mode (active or passive) of the sockets.

2. Multi-threaded servers

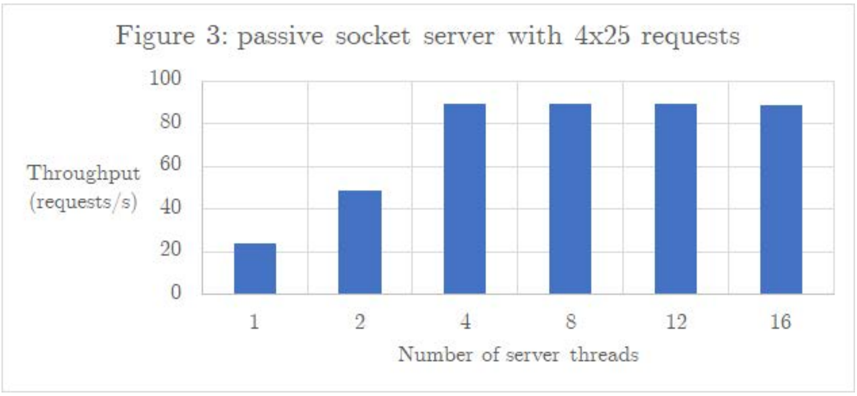


Here we can see a much bigger difference in the throughput with mono-threaded or multi-threaded clients (figure 2). For the same total number of requests, we can quadruple the throughput by running a 4-threads client instead of a 1-thread client. Moreover, for a given number of client threads, the number of requests sent by each thread influences of course the running time, but not the average throughput.

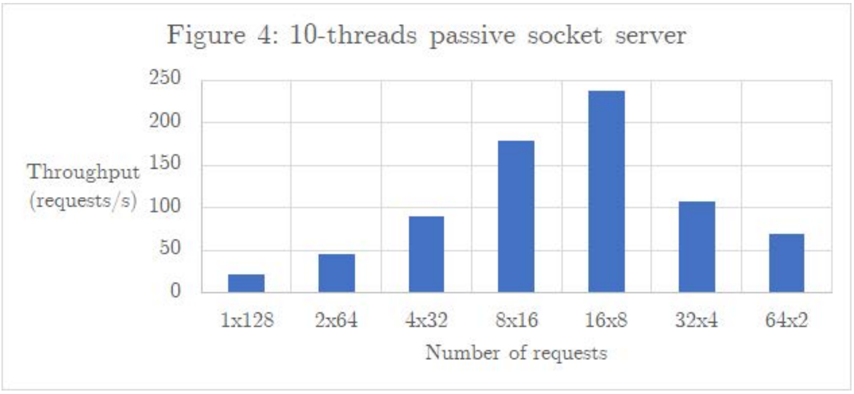
We can once again see that it does not vary with the mode of the sockets. This may be because in both case, the server does not do much apart from handling the requests. Active sockets are useful if the program is going from receiving messages to doing something else, alternatively[1]. Here there is no server side scripting, so it is not of a great interest to use active sockets, and the results show it.

[1]Hebert F., *Learn You Some Erlang for Great Good!: A Beginner's Guide.*, No Starch Press, 2013.

On the remote machine, the Erlang virtual machine was running with 8 schedulers, so I assumed KTH machines have more cores than my PC, which has 2 of them. To measure the impact of the number of server threads on the throughput (figure 3), I ran the server on the remote computer and the client on my machine. We can see that the throughput increases with the number of threads up to 4, but remains constant for more than 4 threads. I guess that means that the KTH machines have 4 cores.



However, I tried to run the Erlang virtual machine with different numbers of schedulers and it did not affect the throughput at all. I was a bit surprised by that, because I expected the efficiency to increase with the number of schedulers until it reaches the number of cores in the processor.



I was intrigued by the fact that the throughput increases with the number of client threads (cf. figure 2), so I tried to see if there was an optimal number of threads on the client side (figure 4). I ran the server on the remote machine, with 8 schedulers, and 10 server threads. The throughput increases until there are 16 client threads, and then decreases again. I was also surprised by this result, as I see no correlation with the number of cores of the processor nor with the number of server threads.

4 Conclusion

Through this exercise I learnt the basic structure of a web server, which is very useful for network applications. It was a good way to start practicing with Erlang and learn more about the semantic of HTTP requests and how they are handled by a server. I learnt how to deal with TCP sockets in Erlang, and the difference between an active and a passive socket. I also learnt more about the concurrency allowed by this language. The most surprising and interesting thing I have learnt with Rudy is that multiple processes can listen on a single socket in Erlang, which is not the case with basic C socket libraries for example.

I highlighted in the Evaluation section that a multi-threaded server is more efficient than a mono-threaded one, especially when multiple client threads are sending requests. I also found that there is an optimal number of client threads in order to have the greatest throughput, and that the number of requests sent by each client thread does not influence the average throughput.

The HTTP parser is rudimentary and could be improved and be more robust to deal with split requests for example. The server would be a much better server if it could deliver files. It would also be useful to write a script to launch the server and multiple clients for example, and this without having to find the IP address and type it by hand for each Erlang node.