Report 1: Rudy - a small web server

Emile Le Gallic

September 10, 2024

1 Introduction

In this first homework we set up a small web server covering several topics :

1. The HTTP protocol : how to parse HTTP request in order to extract the relevant fields of the request.
2. The concept of a web socket : The `gen_tcp` module provides an interface to TCP/IP sockets which allows us to open connections between two host and to exchange messages.

Furthermore, web servers are really related to distributed systems because they often need to handle many client requests concurrently, distribute workloads across multiple servers or nodes, and ensure high availability and fault tolerance.

2 Main problems and solutions

2.1 First implementation

In order to simulate a benchmark on several machines simultaneously, I wrote the `bench/4` function which is designed to measure the time it takes to execute a number of parallel requests to a server. It starts by recording the current time and then spawns multiple parallel processes (using `spawn_parallel/4`) to handle the requests. After initiating these parallel processes, it waits for all of them to complete using `wait_for_parallel/1`. Finally, it calculates the elapsed time for the whole operation.

Initially, I encountered an issue with the run/4 function in the bench/4 implementation. The function used `self() !` done to signal the completion of tasks. However, since `run/4` is executed within a spawned process, `self()` refers to that process rather than the parent process that initiated the parallel requests. This led to problems because the done message was not being sent to the correct process.

To resolve this, I modified the implementation to pass the `Parent` parameter (which is the parent process that originally called `bench/4`) to the `run/4` function. This way, the `run/4` function sends the done message to the correct parent process, ensuring that the `wait_for_parallel/1` function accurately waits for all child processes to complete. This adjustment ensured that the completion signals were received properly, allowing the `bench/4` function to correctly measure the total time for all requests.

Also, as I was conducting experiments with several machines, and everything worked fine until I reached 7 machines; beyond that point, I encountered the error {`error, etimedout`} and {`error, closed`}. This error occurs because, with 7 machines running in parallel, the server struggles to handle the high number of simultaneous connections, leading to timeouts and closed connections. The server likely becomes overwhelmed due to resource constraints or network limitations, causing it to drop some requests or fail to respond in time, resulting in these errors.

2.2 Going further

After the first implementation of rudy (file `rudy.erl`), I moved on to implementing additional functionalities (file `rudy_further.erl`).

I started with **increasing throughput**.

With this new functionality, the `init` function actually uses a new `P` parameter to define the number of handler processes, creating a pool that allows the server to manage multiple concurrent connections.

The `Pids` array of the `init` function holds the process IDs of all spawned handler processes.

The `Exit()` function, passed as a callback (e.g., `exit_rudy/0`), ensures a graceful shutdown by performing clean-up actions once the server finishes its tasks.

Signal exchange between the `rudy` process and its parent coordinates the server's lifecycle, with the `done` signal triggering the shutdown sequence, including subprocess termination and exit routines.

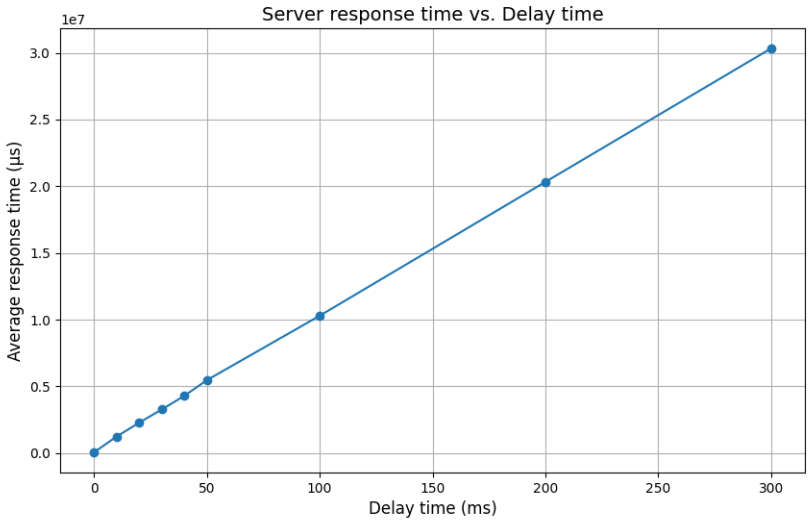
3 Evaluation

I created a Python program (`plot.py`) that reads data from `experiment_single.dat` and `experiment_parallel.dat` files, calculates the average response time from the last three values of each line, and plots the server response time against delay time and the number of parallel processes, respectively, using `matplotlib`.

Here are the raw data and the plots :

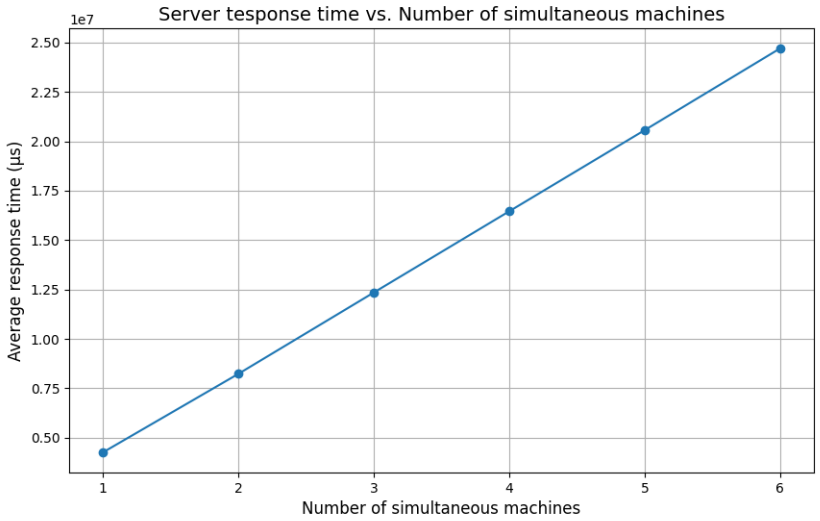
Delay time (ms)	Average response time (µs)
0	53796, 48950, 59911
10	1214230, 1214793, 1272529
20	2264821, 2279670, 2283438
30	3297435, 3237666, 3257649
40	4317210, 4299828, 4239350
50	5803364, 5283446, 5304433
100	10304341, 10264634, 10310753
200	20349431, 20320442, 20312783
300	30319490, 30334644, 30346009

Table 1: Average response time for various server-side delay times (3 values computed for mean approximation)



Number of parallel processes	Average response time (µs)
1	4231048, 4288257, 4249868
2	8224184, 8219808, 8246277
3	12346226, 12349116, 12345901
4	16468580, 16447298, 16459012
5	20579276, 20561990, 20554868
6	24669340, 24738464, 24701291

Table 2: Average response time for varying numbers of parallel processes (3 values computed for mean approximation)



I also measured the response time with a pool of handlers of increasing size. Here are the results for :

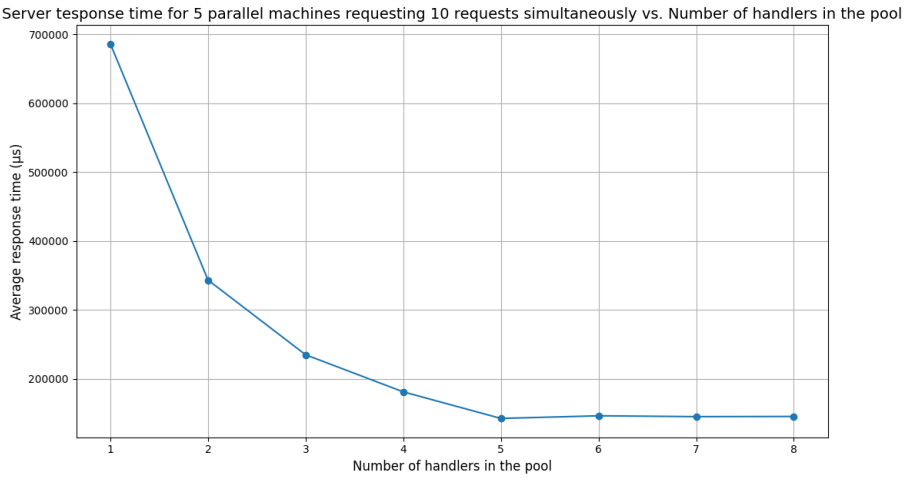
- The response time of a server to 5 parallel machines sending 10 messages vs. the size of the pool of handlers
- The response time of a server to 5 parallel machines sending 100 messages vs. the size of the pool of handlers

In both of those experiments, we notice that reaching a number of 5 handlers in the pool, increasing the number of handlers does not affect the response time. This is because with reached maximum capacity and so we have a response time equal to the response time for 1 handler.

Here are the raw data and the plots :

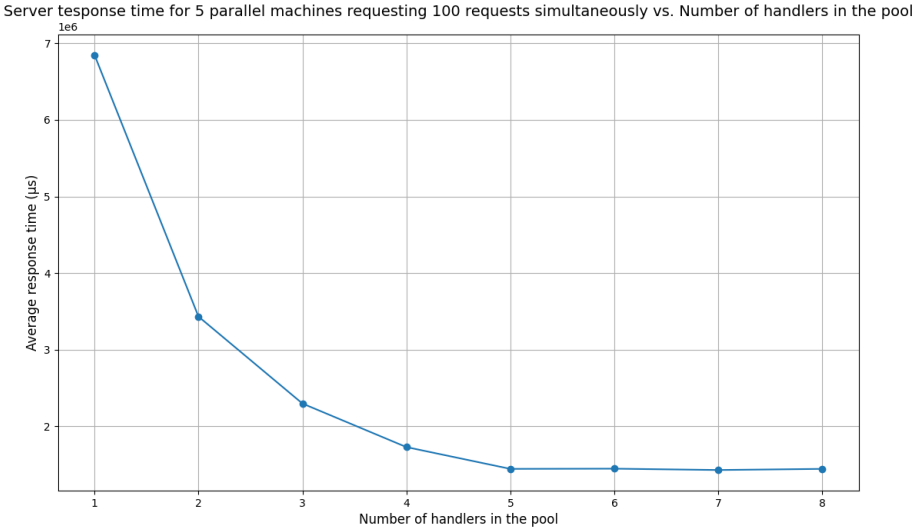
Number of handlers in the pool	Response time (µs)
1	2058843
2	1029816
3	704189
4	542706
5	426966
6	438456
7	434799
8	435631

Table 3: Response time for different numbers of handlers in the pool with P=5 parallel machines sending N=10 messages.



Number of handlers in the pool	Response time (µs)
1	20543227
2	10296534
3	6890910
4	5185432
5	4332015
6	4339803
7	4287650
8	4331901

Table 4: Response time for different numbers of handlers in the pool with P=5 parallel machines sending N=100 messages.



4 Conclusions

This homework was a great way to really implement parts of the HTTP protocol that we often use in software engineering but that we never really dissected that way. Also, it was a good introductory to Erlang and especially the subtleties of functionnal programming which is something I'm not used to.