Report 1: Rudy, a Small Web Server

Luxiang Lin

September 11, 2024

1 Introduction

Through the Homework 1, we were expected to develop a small web server called 'Rudy' which should be able to receive GET requests from the terminal or a browser. It requires us to have an understanding of the HTTP protocol , the structure of a web server and how to use socket API.

In this assignment, I have completed the basic version of GET requests parsing, implemented a multiprocessor server, and extended the server to let it be able to deliver files

2 Main problems and solutions

The first issue that has troubled me for quite a while is using recursion instead of loops for traversal. This is quite different from the object-oriented programming which I am familiar with, and it took me a lot of effort to adapt. To get used to the recursive function, I tried to trace the execution from the top of the stack to the bottom to understand what really happened during each recursion.

The second problem I encountered was when I tried to construct the parallel version of the client(multiple processes are generated, and all process sends requests to the server at the same time). After I switched the client from the non-parallel version to the parallel one, the time it took dropped to 0. This issue was ultimately resolved by adding a `collect()` function to synchronize the processes.

I have also faced difficulties with server-side parallelization. There was no problem running the parallel version of the code; however, the speed at which the server processed the requests increased very little as the number of parallel processes increased. After taking several experiments, I found that the parallel performance improved as the server load increased, this will be discussed in detail later.

3 Evaluation

The server was initially tested with a client that generated 100 sequential HTTP GET requests in three different scenarios: simple HTTP GET requests, HTTP GET requests with a 40ms delay before the response, and HTTP GET requests that involve transmitting a file. The total time to receive the responses was measured in each case

Scenarios	Average time(µs)	Requests/sec
Simple reply	35634	2806.3
Reply with 40ms delay	4650703	21.5
Reply with a 523KB file	80077	1248.8

Table 1

I also done some experiment of the parallel version server. The server was tested on a client which sent 20 sequential requests on 5 parallel threads. It was first tested with simple HTTP GET requests, results are shown below:

Number of Process	1	2	3	4	5	6	7	8	9	10
Response Time	13009	12928	12956	13025	12984	12956	13023	12974	12986	13014

Table 2

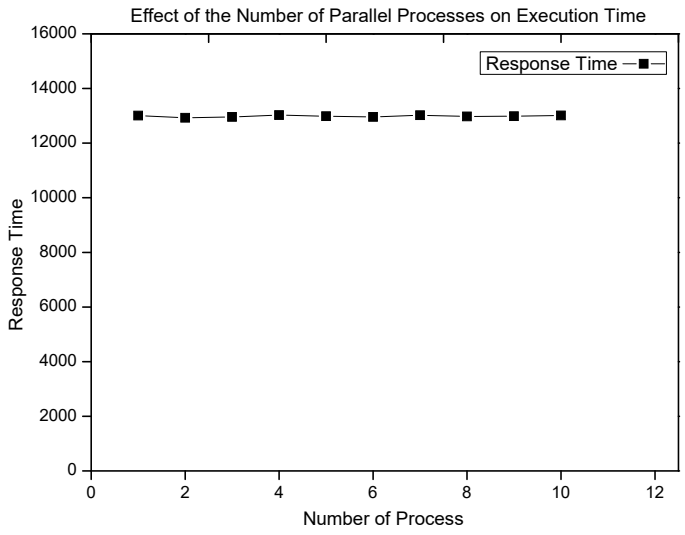

Figure 1

It is clear that the parallelization of the server has little effect in this situation. I hypothesized that the lack of impact from parallelization was due to insufficient load, so I modified the server to reply with files instead of just a few simple words. This change resulted in some noticeable effects, as shown in the results below:

Number of Process	1	2	3	4	5	6	7	8	9	10
Response Time	52428	41677	36352	30208	28775	29593	28467	29081	28262	28263

Table 3

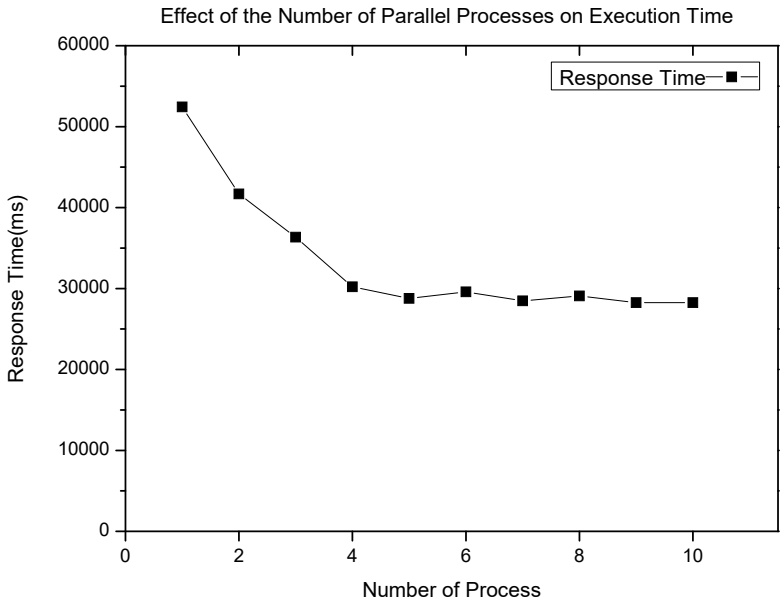

Figure 2

The parallelization begins to affect the process speed. As shown in the graph, before the number of processes spawned by the server reached 5, the response time decreased as the number of processes increased. Then, I attempted to further increase the server's load, I added a0 `timer:sleep(10)` before the reply as part of the workload, here are the results:

Number of Process	1	2	3	4	5	6	7	8	9	10
Response Time	1538151	779059	538317	400179	311501	309453	311910	315188	318157	313139

Table 3

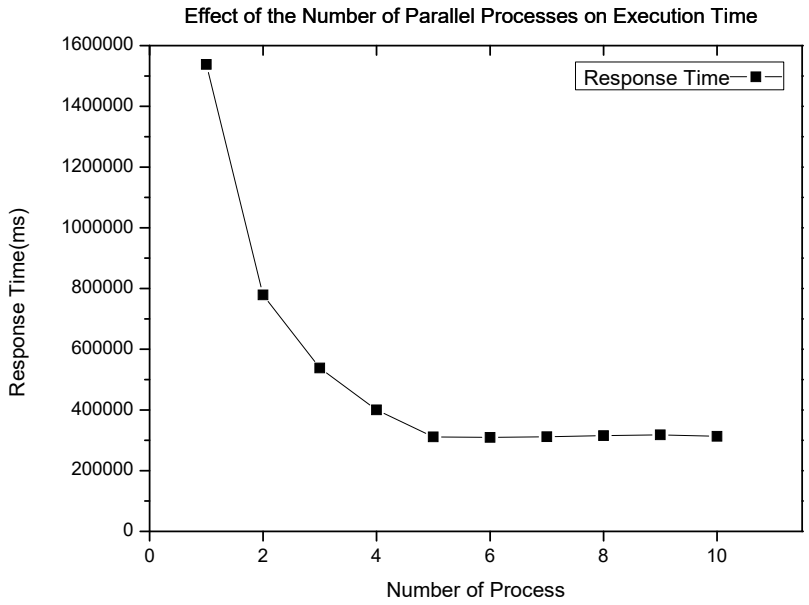

Figure 3

As shown in the graph, when the number of processes is fewer than 5, increasing the number of processes significantly reduces the response time.

4 Conclusions

Through this exercise, I practiced setting up a TCP connection and establishing a small server using the socket API provided by Erlang. Furthermore, I also learned how to build a parallel server and how to transmit files using TCP connections.