Distributed Systems - Rudy

Francis Gniady

September 13, 2022

1 Introduction

In this assignment we will be creating a web server that will parse very basic http requests, and respond to them. We will then measure its performance and find potential solutions to the performance issues that arise.

2 Main problems and solutions

To start of, we follow the instructions in the assignment and end up with a simple but working web-server. It can receive GET requests and respond with some value.

The issue with the server that we have so far is that it is only one process, processing one request at a time, one after another. This means that the amount of requests scales linearly with the time it takes to process them. Assuming we are doing some heavy computations that make each request take at least 40ms just to process the result, this will only allow us to serve 25 requests a second (very slow).

So how do we improve the performance of the server? We could try and run the server multiple times, but that would mean multiple processes try to create their own listening socket on the same port, which would fail. We could also listen on different ports, but how would the client know where to connect to? We can't really give them a list of ports and tell them to figure it out.
Sure we could use some external load-balancer to split the traffic to the different ports, but there is a much better solution, parallelization!

Why have one process when we can have multiple all handling requests on their own. We will have a pool of worker processes that can be given a request and respond to it on its own, this way we can process 25 requests a second, per worker, which we can have hundreds off.
We could do this by having one process listen, and then send those requests

via messages to our workers to do the heavy computation. However luckily for us, the gen_tcp library we are using allow us to have multiple processes accept requests from the same socket, in parallel. Meaning all we have to do is open the socket, and share it to all of our workers which can then use it as a sort of FIFO queue. This was implemented in the server_improved.erl module.

There were also some problems when it came to testing, for one, if there were too many clients connecting then some of the requests would timeout and return a "connection refused" error. This was solved simply by setting the timeout on the requests to infinity.

Custom parallel tests were also made so that a single test would spawn X processes all making Y requests, these would then communicate the times to make the requests back, and the results would be compiled.

3 Results

Note: Times denote the time it took a single client to complete all of its requests, not a single request.

Clients	Requests	Total	Avg	Max	Min	Artificial Delay
1	200	9280	-	-	-	Yes
1	100	4641	-	-	-	Yes
1	50	2329	-	-	-	Yes
1	25	1163	-	-	-	Yes
2	50	4655	4632	4655	4609	Yes
1	5000	4367	-	-	-	No
2	2500	6725	3363	3363	3363	No
4	1250	6264	1566	1585	1549	No

Table 1: Performance of non-parallel server

Clients	Requests	Workers	Total	Avg	Max	Min
1	100	10	4641	-	-	-
2	100	10	4639	4639	4639	4639
5	100	10	4641	4631	4641	4640
10	100	10	5650	4998	5650	4645
11	100	10	7692	6826	7692	5651
20	100	10	11161	9227	11161	7271

Table 2: Performance of parallel server

4 Conclusions

To start off, in the first table we can see that for the non-parallel server, time to complete X requests is linearly proportional to the total number of requests. Regardless of if these requests all come from the same client or from multiple clients. We can also see that the majority of the time, the server spends waiting for the artificial delay. This can be seen by the fact that 5000 requests without an artificial delay take approximately the same time as 100 requests with a delay. Without the delay however, it would seem that the server is more sensitive to the amount of clients connecting. However this could also be a result of how we measure the time taken.

As for the parallel server (which always had the artificial delay), we can see that as long as there are more workers than clients, the time to process the requests stays approximately the same regardless of client count. With some slowdown then clients = workers. However, as soon as the number of clients is greater than the number of workers then we will start to see the requests taking longer and longer. We can approximate the time it would take the parallel server if $clients \geq workers$ as $(clients/workers) \times requests$. With some overhead of course.