UCLouvain - EPL

LINGI2241

Architecture and performance of computer systems

Project : Measurements and Modeling

Professor :
Sadre Ramin

Assistant :
Kabasele Ndonda Gorby

Authors :
Hennebo Eliot - 43762000
Ben Haddou Mehdi - 19912000

1 Introduction

As part of the course `LINGI2241`, we carried out a project to evaluate the differences in performance between a simple server and an optimised server which allows different clients to query a database according to a regex.

To implement this server, we rely on a queuing model that will allow us to compare the differences between the theoretical and practical model.

2 Implementation

In this section, we will discuss the particularities of our implementation for both versions of the servers.

Before that, we can talk about the client application used, which is the same for both versions. The client application launches X (from 10 to 100) client threads which each launch Y (from 5 to 50) threads opening a TCP connection to the server in order to send a request. Each client launch its Y threads following an exponential distribution.

The requests sent by the client are generated with a small engine that we implemented (in the RegexGenerator class) according to different difficulties (easy, medium, hard, network intensive, mixed), the number of requests generated is of course selectable and all the requests are stored in separated files directly usable in the project.

In order to better understand the implementation, we can also say that the database includes more than 2 million lines which each have a type ranging from 0 to 5 inclusive, so we can assume that there are approximately 2,xxx,xxx/6 lines of type 0, 1, etc...

We can add that in order to be able to answer several requests simultaneously, the two servers uses thread pools to be multi-threaded (from 2 to 16 threads). Therefore, the corresponding model queue of our server is a M|M|m queue.

2.1 Simple version

First, to represent the database, the simple version uses an array of N rows and two columns that represent the type and the associated string. To answer a query, I browse through the N rows.

In order to solve the query, the protocol checks that the query is consistent and then scans the N rows for rows that match the types and the regex.

2.2 Optimized version

For this version, we have implemented several processes to improve performance. Firstly, we have changed the structure and we are using an ArrayList<String> array of size 6 since there are only 6 types.

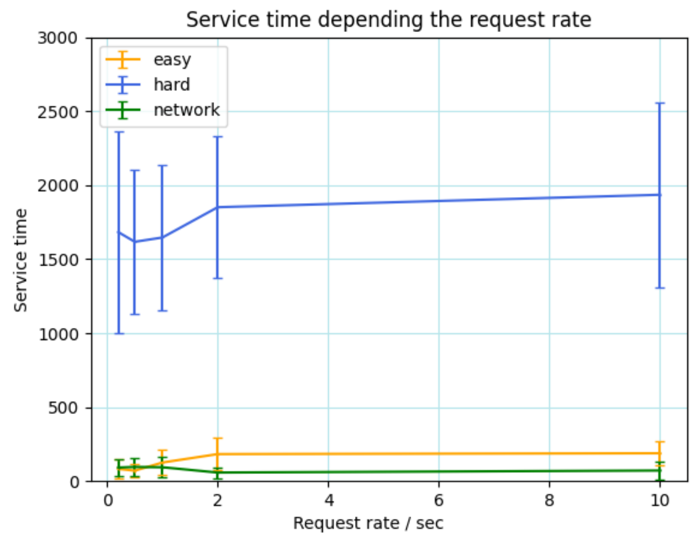
Indeed, the less different types there are in the request, the faster the process will be. In fact, if there are as many rows for each type in the database, the comparison number will be N/6 * nTypes where nTypes is equal to the number of types in the client query.

In addition, we use a cache that implements the LFU (least frequently used) algorithm to avoid having to resolve the query if the query has been requested before.

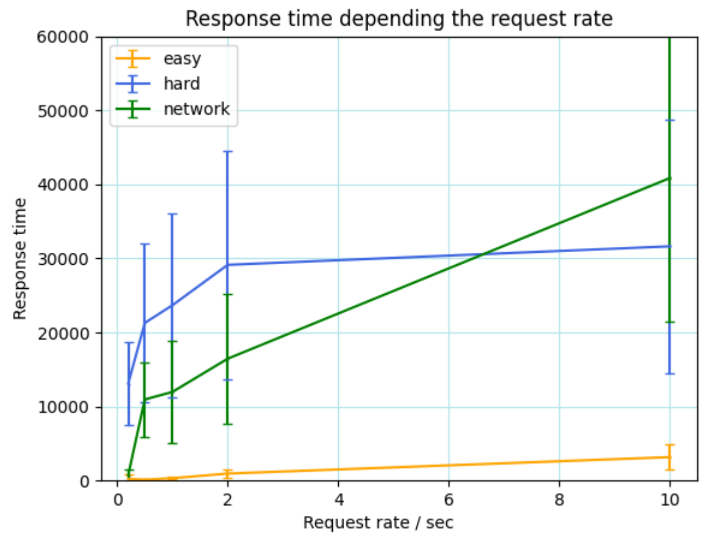
3 Measurements

Each set of data was measured on the optimised server with a number of threads equal to 8.

3.1 Request rate and difficulty impact



We can easily see on the plot above that the easy and network intensive requests are very quick to execute unlike hard which typically use more types and lots of match in the database.



Here we can quickly see that regardless of the type of request, if the rate of requests per second increases, the response time increases with it because the server can no longer keep up with the requests.

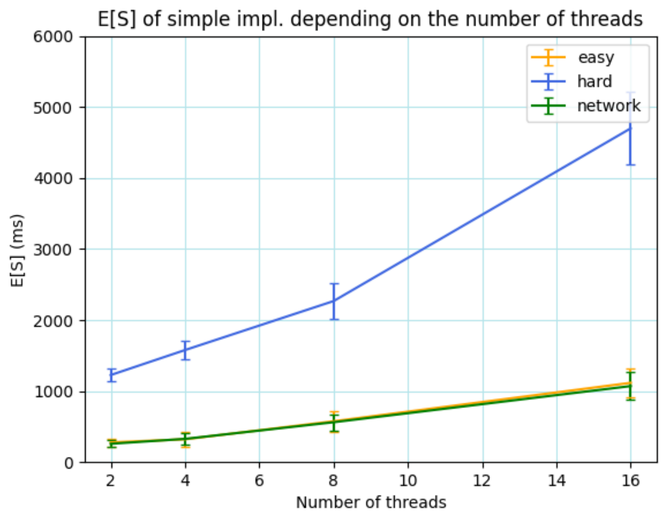
We can also note that we have large standard deviations especially for hard and network requests.

3.2 Number threads impact

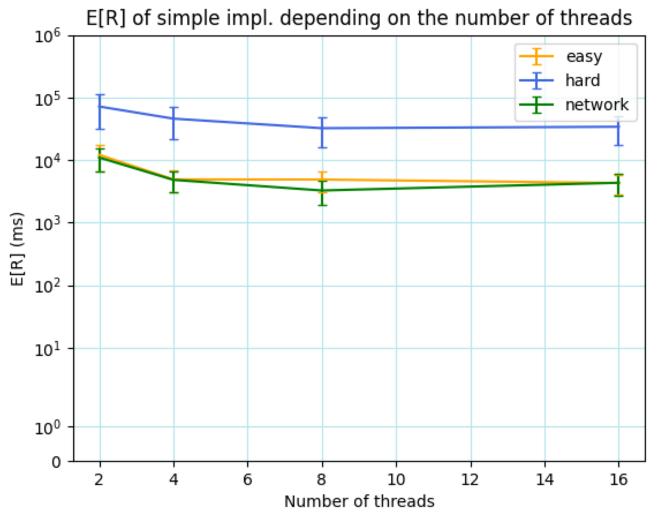
We can see in both version of our servers that the service time is growing, with a faster growth for the hard requests. The growth is due to the fact that for example the 16 threads in our graph is not really a 16 threads processing time, due to the time slicing of the execution, as for example in the round robin scheduling algorithm.

3.2.1 Simple version

In our simple version, the E[R] of the different executions depending of the number of threads is slightly decreasing, because the server will handle more clients at the same time, so in average they will wait a bit less time. But if we take the E[R] of a simple client in a 16 threads execution, his E[R] will be greater than a single E[R] of a well placed client in a 2 thread execution. Overall, we can see that increasing the number of thread is worth on the long way, and if the number of real cores on the computer is bigger, the E[R] will be even better.



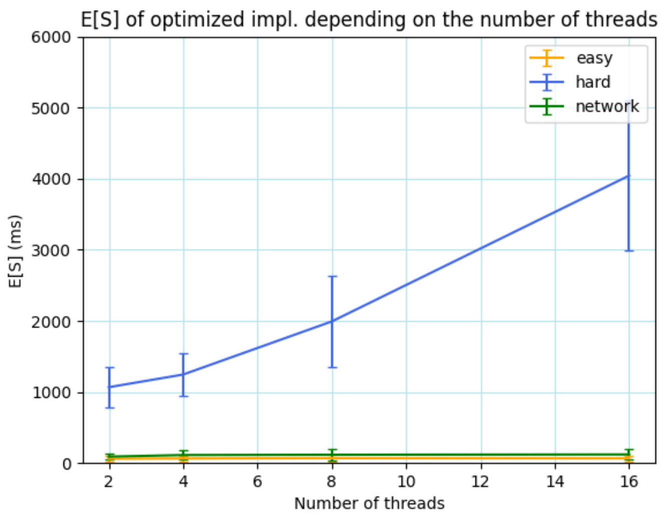
[b]0.4

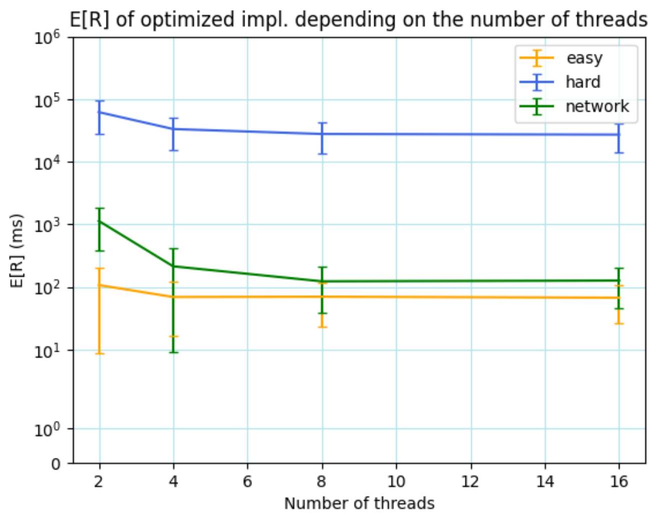


[b]0.4

3.2.2 Optimized version

In our optimized version, the service time is clearly lower for the easier requests, with a service time close from 100 in average. That is due to multiple facts, the cache that we added, and the modification to the data structure that we made, as explained in the implementation part.





4 Modeling - Conclusion

We can see that the queuing model is a M|M|m queue, this is shown by the fact that our server is multi-threaded, we have multiples clients running in parallel and they decide to send requests following an exponential inter-sending time. Our multiple threads are working as multiple service stations, so the m service stations tend to be more efficient in this type of implementations because we tested it with a lot of clients running in parallel, and this reduces the average E[R] of the server.

The measurements of the server shown that the queuing model is quite a good tool to analyze and plan a new server, because the different results lead us to the same results as in the mathematical model. Especially in the measurements where we compare the response time of the server with the different rates of arrivals of the clients.