Group members :

— Manu Vaillant 56192100 manu.vaillant@student.uclouvain.be

— Simon Callens 38161600 simon.callens@student.uclouvain.be

Section 1 Implementation

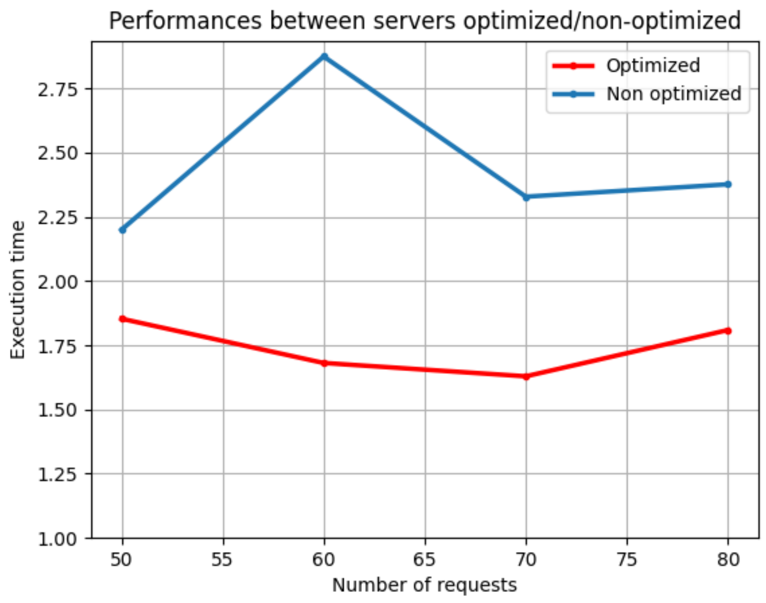
1.1 Basic server

To implement the server part, we have defined a thread pool executor containing 4 threads. Each thread can process a client query, so the server can process up to 4 different query in parallel. Each thread will brute force the client password until it finds a password with a hash similar to the one given by the client (in the query). Afterwards the thread will decrypt the client file with corresponding password and finally return it back to the client.

1.2 Server optimization

In order to optimize the brute force process of the server, we've used the "10k-most-common-filered.txt" file. As indicated by his name, this file contain the most frequently used passwords. When starting the server, we create an HashMap and fill it with all the most common used passwords. Each entry of the HashMap has as a key a password hash and as a value the corresponding password. Given that the majority of the code for both versions still the same, we kept only one code base which is configurable to be optimized or not through a simple Boolean variable.

This initialisation process allow all server threads to first check if a hash is already present in the HashMap before brute forcing it. If the hash is find in the Map, we have directly the corresponding password and we don't have to brute force it (which can take a long time). Here is a plot representing the performance of optimized and non optimized servers :



1.3 Client

Concerning the client, we also used a thread pool executor containing from 50 to 100 different clients. Each client will wait a certain amount of time (following an exponential distribution) before sending a request to the server. The client can also be easily configured for different difficulty (file size, password size and lambda for exponential distribution). We also chose to send an easy request (whose password came from "10k-most-common-filered.txt" file) to the server with a rate of 1/3. When the server returns the decrypted file for a client, it write the response time to a file (with a thread safe function to avoid threads conflicts when writing in the file).

Section 2

Measurements

2.1 Setup

To measure the performance of the server, we have used two local computers connected to the same internet router with the following hardware :

— **Server specifications** : Intel Core i7-8550U CPU 1.80GHz, 8GB RAM

— **Client specifications** : AMD Ryzen 5 5500U 2.10GHz, 16GB RAM

— **Network** : WiFi with average 20 MB/s upload speed and 80 MB/s download speed.

2.2 Workflow

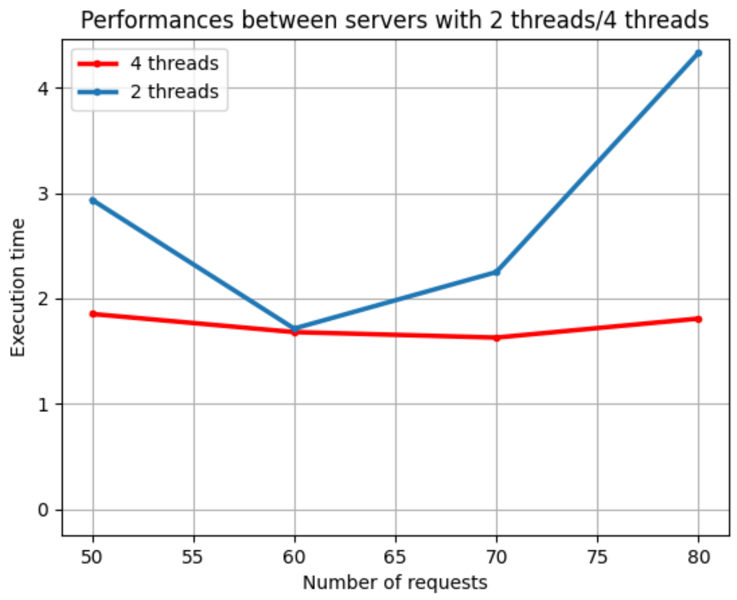
We don't use any specific software to measure the performance, we only launch client and server Java applications on corresponding computer. As explained in *1.3 Client*, each client write its corresponding response time in a shared CSV file that we've used to plot the results with Python.

Clients threads generate a request based on current configuration (password size, network file size, inter request rate) and send it to the client through a TCP connection.

2.3 Results

2.3.1 Different number of threads

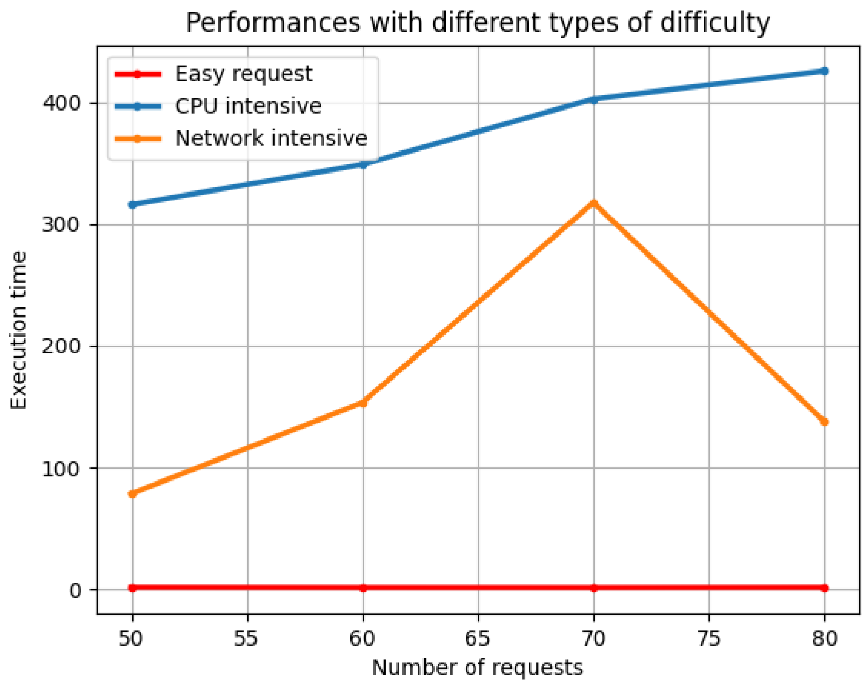
Firstly we've measure the impact of the number of threads in the queuing station. On the graph below we can see the importance of having a multi-threaded server. Server with 4 threads have a constant execution time depending the execution time. However, the 2-threaded server has a deteriorating execution time. Both measurements were done with easy requests (5KB file size, 5 char password).



2.3.2 Different request difficulties

After that, we have measure the impact of various request difficulties :

- **Easy request** : 5 char password, with 50KB network file size.
- **CPU intensive** : 6 char password with 50KB network file size.
- **Network intensive** : 5 char password with 5MB network file size.



Analysis of the results :

- We clearly observe that the difficulties of client query have a huge impact on the response time of the server. Response time for difficult requests increase a lot due to the increase of average service time. Because the average service time is higher, the queue become gradually overloaded and unstable, so clients have to wait in the queue before being served.
- The performance of CPU intensive requests is constantly degrades. By increasing the size of all clients passwords, we exponentially increase the number of password to brute force which cause the poor performances.
- About the network intensive requests, we can see that we had a peak of execution time around 70 requests and then the execution time comes back about 100 s (execution time). This peak was due to the network connection. Unlike CPU intensive requests, network intensive queries strongly depend on the network. We could imagine that the network was slower when we make these measurements.

Section 3

Modeling

3.1 Model

To attempt to model our client-server application, we choose to use a $M\|M\|m$ model. We choose this one for different reasons :

— The arrival time of our code follows an exponential process (as mentioned before)

— The capacity of the queue is infinite (no customer is rejected)

— FCFS (First come first serve) service policy

— We have m different services stations

These different reasons let us two possibilities for modeling : $M\|M\|m$ or $M\|G\|m$. Since the first one is easier to calculate and more documented in the course, we made the assumption that the service time is also exponentially distributed.

3.2 Parameters

We modeled a $M\|M\|4$ with an arrival rate λ and a service rate µ. As mentioned in the project statement, the service time is equivalent to the time that our system needs to process a single request without any queuing delay. Since our optimized server can be very fast with a password which is in the dictionary or less fast with a password beginning by z, we decide to evaluate the service time by taking the mean time of different request with a low arrival rate (without queuing delay).

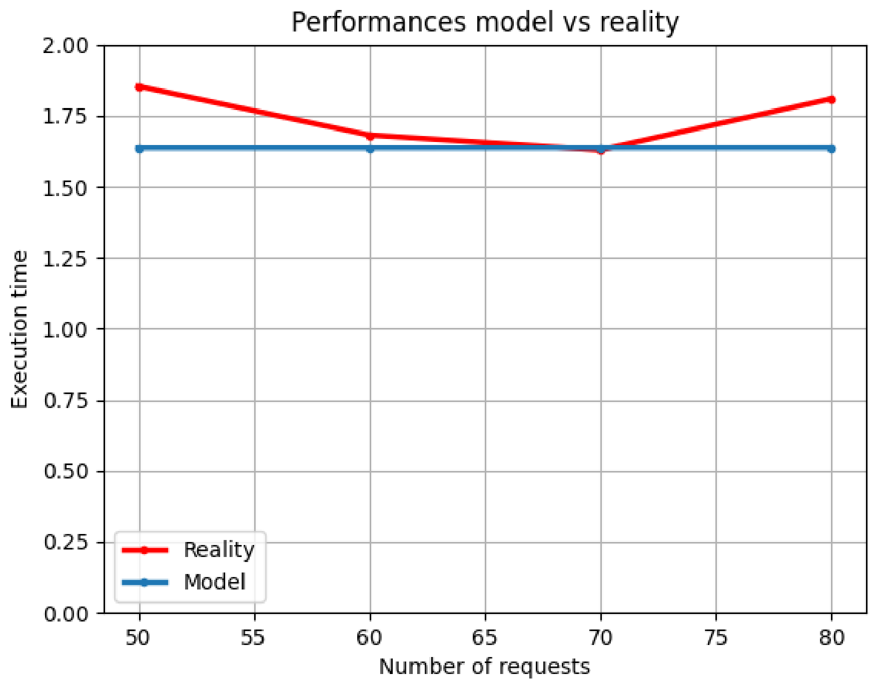
We applied then the following formulas in order to have the theoretical mean response time :

$$\pi_0 = \left(\sum_{i=0}^{m-1} \frac{a^i}{i!} + \frac{a^m}{m!\,(1-\chi)} \right)^{-1} \text{ with } a = \frac{\lambda}{\mu}, \chi = \frac{\lambda}{m\mu}$$

$$E[R] = \frac{1}{\lambda}\left(a + \frac{\chi a^m}{(1-\chi)^2 m!} \pi_0 \right)$$

3.3 Comparison with reality

For the first experiment (optimized server, 4 threads, 5char, 50KB files), the mean response time of our model varies slightly from 1.62 to 1.64 seconds. This is due to the fact that the queue is often empty and that the requests are processed almost instantaneously. In reality, the time is a bit longer because of the varying quality of the network and the fact that some requests stay a bit longer in the queue if the previous passwords were more difficult to crack. We can see that the modeling evaluation suits pretty well the actual real performances of the system.



Unfortunately, the modelling doesn't fit reality for other styles of request (Network and CPU intensive). We assume this is because the queuing station become overloaded (and unstable) with hard requests.