LINFO2241
Task 1-2

Volders Maxime - *maxime.volders@student.uclouvain.be* - 26521700
Gauthier Arnold - *gauthier.arnold@student.uclouvain.be* - 56571700

January 8, 2022

1 Description of the implementation

1.1 Server

For the server side, we decided to create a new class to help the support of threads, the `ClientProcessor` class. This class is called from `ServerMain.java`, in which we choose the number of threads. Every time the **Server** receives a sending request from the **Client**, a new `ClientProcessor` was created. In this second class, we mostly took the same code that was given to us as a starting point. The thing we modified, as it was asked, was the way to find the password used by the **Client**.

For the non-optimized version, we used brute force. We first generate a string made of a's that is the size of the password to find. Then, we check to see if it's the correct password to start the loop. If it isn't correct, the last char of the string will be incremented, and it loops until a hashed password matches the received hashed password. After that, we try to decrypt the file. If the file is correctly decrypted, we send the decrypted file and we end the connection. If the decryption sends an error, we catch it and we start the loop all over again with the next password, until we find the right password.

For the optimized version, we took to non-optimized version of the **Server**, to which we added our optimization before the brute force decryption. We added a similar loop, but instead of checking every word of a certain length, we check the words of the right length in the `10k-most-common_filered.txt` file. We can see the impact the version of the server has on our performances on Figure 2 in section 2.3.2.

1.2 Client

For the **Client**, we took exactly the same client as the one that was given to us. The only thing we modified, as explained in section 2.2, is that we randomly generate a password or we take a random word of the `10k-most-common_filered.txt` file.

2 Tests

2.1 Measurement setup

To measure the average response time of our **Server**, we did several tests, changing each time one parameter. The hardware used was the following

- We used our local kot network, where we opened port 3333 to execute our tests.
- The **Clients** and **Server** were run on two different computers.
- The **Server** was executed on a 16 Gb RAM computer with 6 cores.

- The **Clients** was executed on a 8 Gb RAM computer with 2 cores.

For the **Server**, we used 12 threads, since the computer used has 6 cores.

2.2 Workload used

To run the **clients**, we created a bash script `TestClients.sh`. This script executes the test for 2, 5, 10, 25, 50, 75 then 100 **Clients**. It waits a random exponential time

$$t = \frac{ln(1 - random)}{-\lambda} \tag{1}$$

to launch each **Client**, so we can vary the workload of our network (with $\lambda = 2$).
To perform our tests, we also implemented a random password generator that works in two ways:
We can either generate a random password of a given length, with only lower case letters, or we can select a random line in the `10k-most-common_filered.txt` file. This allowed us to test efficiently every aspect of our server, with easy and difficult passwords, of different lengths.

2.3 Test results

2.3.1 Password difficulty

First, you can see on Figure 1 that we compared the time of the optimized server regarding the difficulty of the password. In yellow, you can see a password coming from the list `10k-most-common_filered.txt`, which can vary between 4 and 8 characters. Then, we can see the time for a generated password of 2, 3 and 4 characters in blue, red and green. We can see that when a password is coming from the list, it takes as much time to decrypt as a random generated password of 4 characters, even though it is longer.
Of course, these are all random generated passwords, that's why the 3 characters line is below the 2 characters line ("aaa" takes less time to decrypt than "zz").

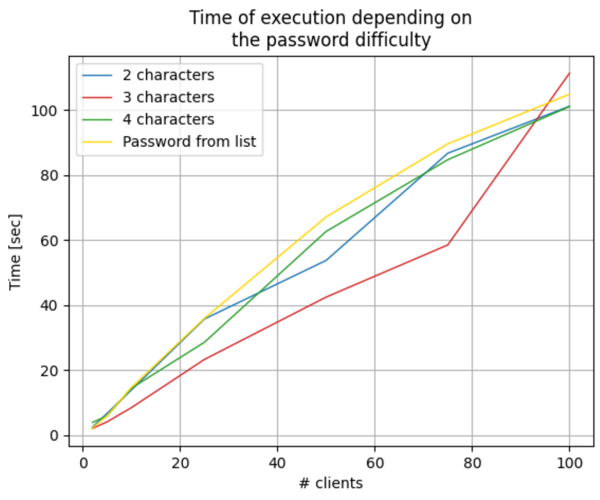
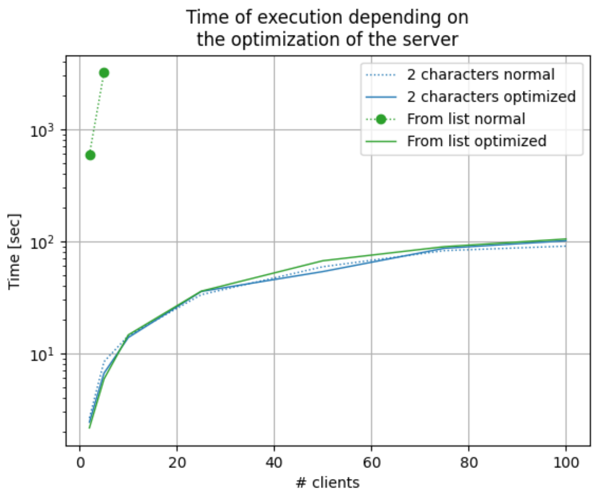


Figure 1: Time of execution depending on the password difficulty

2.3.2 Server parameters

To test the efficiency of our server, we tested two parameters. First, on Figure 2 left, we compared the normal version of our server versus the optimized version of our server. We can see, in blue, that for a random generated password, the two versions take the same time.
But for a password taken from the `10k-most-common_filered.txt` file, which is between 4 and 8 characters long, we see that our optimized server takes 10^2 less time. We stopped the test at 5 clients, since that

already took us nearly 2 hours, which means that for 50 clients, with 12 threads, we would have been at approximately 10 hours.



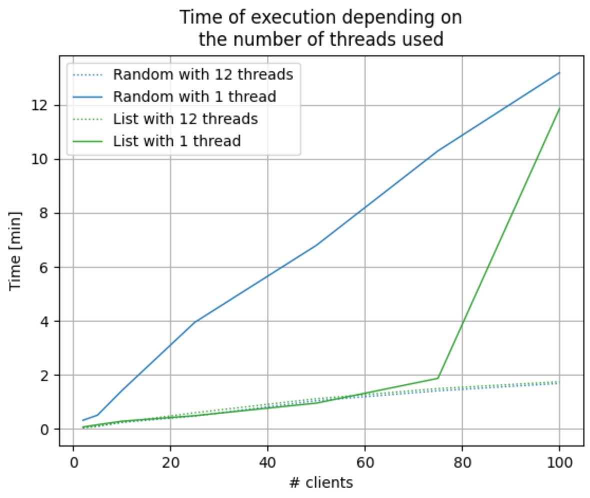


Figure 2: Time of execution depending on the server parameters, with $\lambda = 2$

Secondly, on Figure 2 right, we compared the impact of the number of threads we choose. We tested this with our optimized server. We can immediately see that, the more threads we have, the faster it goes (which is normal, since more threads means more data processing concurrently). The random stands for a random generated password, and list stands for a password from the `10k-most-common_filered.txt` file.

2.3.3 Network load

To test how the network load would influence the execution time, we changed the value of λ in Eq 1. As λ becomes smaller, the waiting time between each clients becomes greater. That way, the clients wait less time in the queue of the server, and thus the execution time is smaller.
We can see these measures on Figure 3. These tests were executed on the optimized server, using passwords from the `10k-most-common_filered.txt` file.

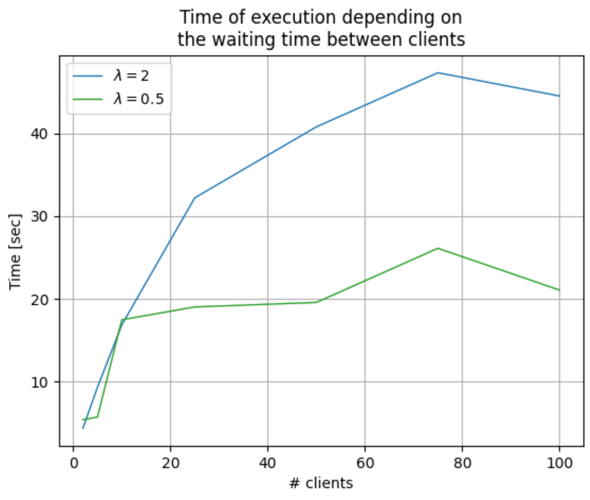


Figure 3: Time of execution depending on the waiting time between clients

3 Modelling

We decided to generate a request using an exponential rate given by the following formula:

$$\lambda = \left(\frac{ln(1-p)}{-k}\right)^{-1}$$

with p a random number between 0 and 1, and k a constant chosen by us. The bigger the k, the faster the requests will be sent. In our case, we decided to take k = 0.5 to limit the number of waiting requests.
With this exponential rate we saw that we were in the case of a M|M|m queue, with m the number of service stations. m is 12 in our case since we used a computer with 6 cores and 12 threads. We measured the average service time E[S] = 1.8382s.
To calculate the response time, we used to following formula :

$$E[R] = \frac{1}{\lambda}\left(a + \frac{\chi a^m}{(1-\chi)^2 m!}\pi_0\right) = 1.838200003$$

With :

- $\lambda = 0.7213$
- $\mu = 0.544/s$
- $\chi = \frac{\lambda}{m\mu} = 0.1105$
- $a = \frac{\lambda}{\mu} = 1.326$
- $\pi_0 = (\sum_{i=0}^{m-1} \frac{a^i}{i!} + \frac{a^m}{m!(1-\chi)})^{(-1)} = 0.265542$

We can see that we have a theoretical waiting time of almost 0. This is explained by the low k that we have chosen earlier. You can see on the following graph that as we increase our k, our average execution time gets bigger.

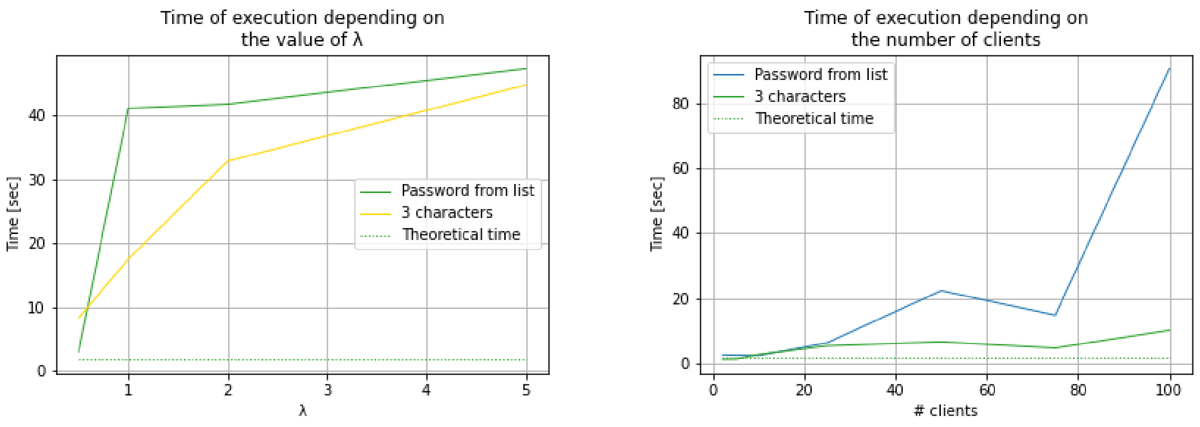


Figure 4: Comparison between theoretical and practical execution

But if we compare the theoretical E[R] with the results we have from the execution, we can see some big differences. The formula doesn't take in mind the number of requests that are made. The more requests we send, the bigger the waiting time will be. We can see that clearly on Figure 4: Figure 4 left shows the time of execution with a varying λ with the optimized `Server` and password from the `10k-most-common_filered.txt` file and 3 characters passwords, and Figure 4 right depends on the number of clients with k = 0.5. We tried the same for a k = 2, and the gap between the theoretical E[R] and the results we had was even bigger. We can clearly see that the formula doesn't take the length of the waiting queue into account.

In conclusion, we can say that the M|M|m formulas give an approximation of the total response time, but since it doesn't take the size of the waiting queue into account, the difference with the reality can be big.