Architecture and performance of computer systems – Measurements and Modeling

Robin Paquet (02192001 - robin.paquet@student.uclouvain.be)
Luc Gabrys (13481800 - luc.gabrys@student.uclouvain.be)

January 9, 2022

1 Introduction

In this project, we will analyze the performance of a simple client-server application that we implemented in Java. We will describe the implementation of our application in the next section. We have implemented two versions of the server, a simple version and an optimized one. The way the server was optimized was left to us. The goal of this work is to measure the performance of the server in a systematic way and to show how the optimizations we have chosen impact the performance.

2 Implementation

2.1 Client

Our implementation of the client is basically based on the code given in the base archive in terms of requests sending to the server. The changes that have been done in the client implementation concern the number of requests, the sending time of requests and the length of the password sent to the server. Also the pattern of the password are generated randomly to perform measurements under different conditions. We have implemented two solutions for sending requests :

- The first allows you to send a request with a fixed password size.
- The second solution allows you to enter a request pool to send to the server.

In the second case when we send a fixed thread pool to the server, we use the exponential distribution formula (which is below) to send each request at a different interval.

$$Math.log(1 - rand.nextDouble())/(-\lambda)$$

The size of the file and the password length sent remains the same for each request from the pool.

2.2 Server

For the implementation of the server, the code of the given archive helped us to receive request from the clients. Some change have been made for it to work properly. The server opens a socket and waits for incoming requests from the client. It can receive requests from different clients at the same time. When it receive a request, the server call a bruteforcing class for each thread. This method is implemented so as to generate a word combination by crossing the alphabet from a to z and stop when the matching password is found. At this point, the server send the decrypted file to the concerned client and try to decrypt another request sent by an other client.

3 Optimization

3.1 Server

The optimized server is basically the same, we have changed some code in order to start bruteforcing in a different way. Rather than starting from the letter a to the letter z for the simple server, the optimized server creates a second thread which will, in the opposite direction, try to find the password by starting with the letter z. So, if the password begins by a letter which is at the end of the alphabet, we will reduce the time needed to find the password.

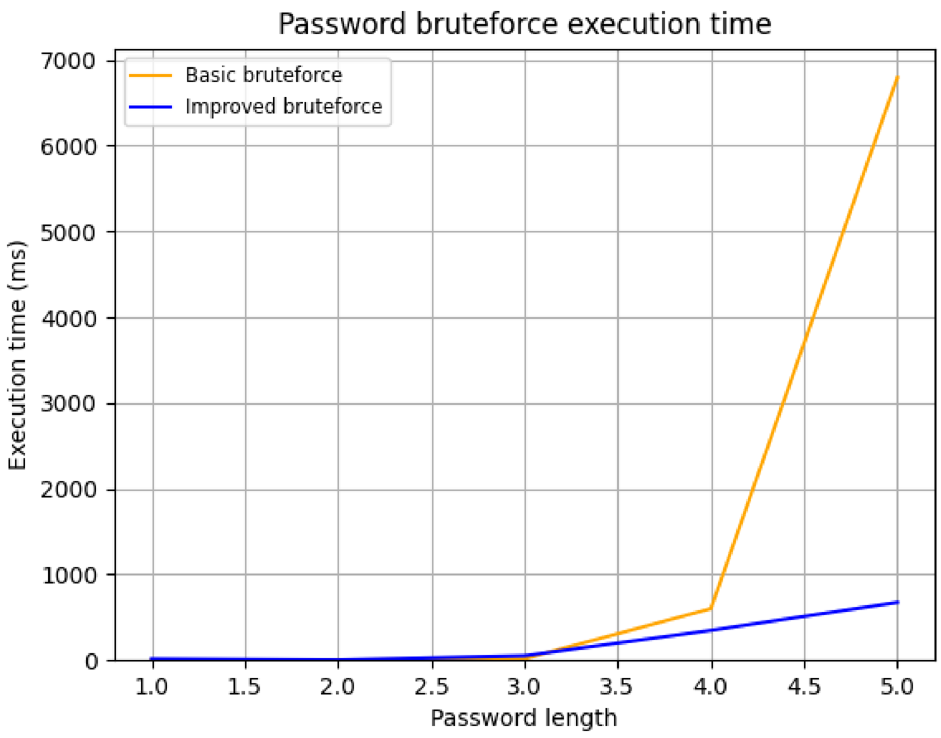

Figure 1: Performance of the regular and optimized bruteforce servers

4 Measurement setup

We tested our implementations using different setups. Firstly, we wanted to understand how our algorithms worked for passwords of increasing length. We tested this part by sending one request at a time and measuring the computation time locally.
We then wanted to know how the time was lost when transferring files over the network. We connected two computers using socket connections, and measured multiple times (total elapsed time and computation time), keeping the password size at 4 and using only one thread as well.
Finally, we checked the response time of the server when it receives up to 100 requests (1 to 10 by increase of 1 and 10 to 70 by increase of 10) at one time which is our limit case. We used a password size of 2 and a fixed file. The parameter λ is equal to the number of clients we wanted to use Both for the simple and for the optimized, we have configured the servers in such a way that it can receive a maximum number of 10 clients which can wait for a connection to be accepted at the same time.

5 Measures

For the first part of our measurement, we saw that the increase on time needed to compute the bruteforcing of the password by our algorithms is exponential, as theory made us expect (Exponential of base 26, maximum time divided by 2 for the improved version). The second one is far more efficient in the graph shown in Figure 1, but that difference should quite decrease replicating the measures as the passwords are randomly generated.
The results for the second part were that, as it was expected due to the proximity of the machines and the quality of the connection, the time needed to send the request through the network increases linearly in term of the size of the file sent. This can be seen on Figure 2.
Finally, our last measures showed that the time needed indeed growth with the number of requests (Figure 3. The decrease in time when there is a lot of clients can be explained by the fact that the server rejects some of the client's requests when the waiting queue is full.

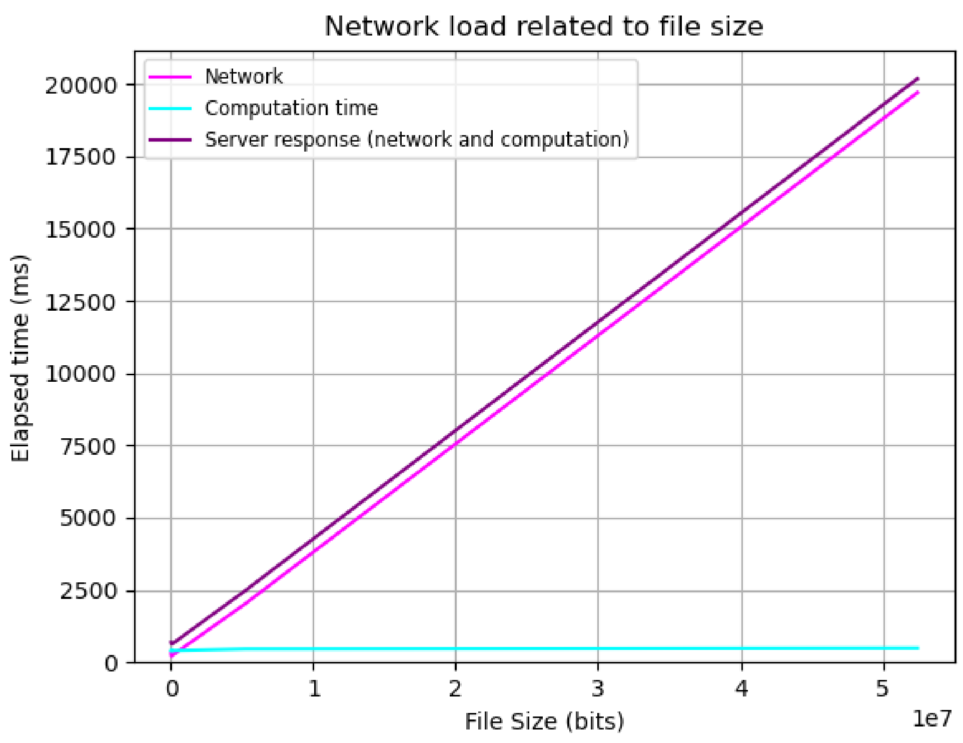

Figure 2: Time elapsed for different file sizes (1 thread, same password size (4))

6 Queuing station model

The queuing station model which is the best approximation of our server is the M|M|1|m queue because the requests are send in an aleatory manner, and because there is only one service station which needs an exponential time in term of the password length to force the password. The maximum queue length for incoming connection indications (a request to connect) is set to the backlog parameter in constructor. A M|M|1|m queue limitts the buffer capacity (including the place in service station). Every time the server receive too many requests from the client, the queue will be full and some request will be rejected. (eg: if we send more than 20 request at the same time from client to server, the server will refuse some request and the clients will not receive a response for every request they have sent). As we can see, the number of rejected requests is exponential, which matches quite well the theorical graph shown in the slide 7 of the course on M|M|1|m queues.

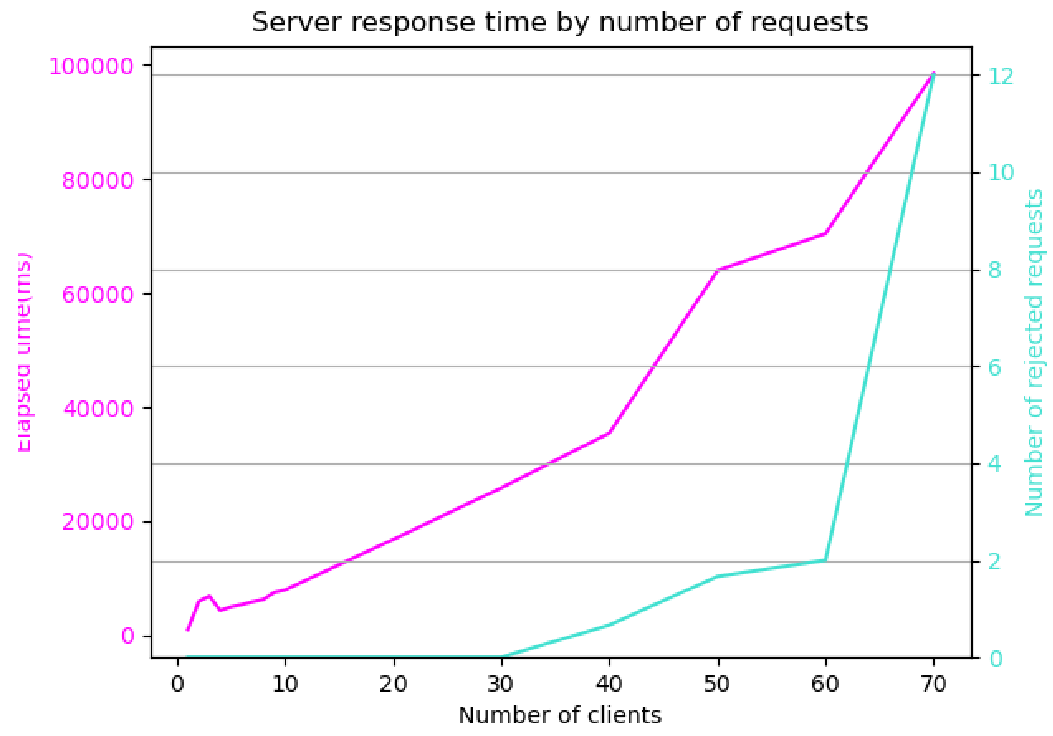

Figure 3: Response time of the server and number of refused requests

7 Conclusion

At the end of this project, we can see that when we perform a bruteforce the time measurement increases exponentially depending on the length of the password received in the request. We can do some improvements but the time is not radically changing. This proves that password hashing is an optimal solution to secure the encryption of a file.