LINFO2241 - Architecture and performance of computer systems

Performance measurements of a client-server application

Corentin DENIS 58701700 corentin.g.denis@student.uclouvain.be
Merlin CAMBERLIN 09441700 merlin.camberlin@student.uclouvain.be

1 Description of the implementation

The whole implementation of the client-server application is divided into 11 classes. The main are: `Main.java`, `ServerMain.java`, `BruteForce.java`, `DumbBruteForce.java` and `SmarterBruteForce.java`.

Server The server is implemented in the class `ServerMain.java`. The server first creates the socket `serverSocket` and continuously listens for connections to be made to this socket. For each connection established, it creates a new `clientSocket` and runs a thread `RequestHandler`. This thread creates the required streams for the communication, processes the incoming request, reverses the hash and sends back the decrypted file. The network configuration is illustrated in the figure 1.

To process the incoming requests, the server uses a `threadPool` with 6 threads (matching with the number of cores on the physical server machine). Each thread will try to reverse the hash received. The server either performs a dumb brute-force (brute-force attack) approach or a smarter brute-force (brute-force attack combined with a dictionary attack) approach. The behavior of these strategies are respectively implemented in the files `DumbBruteForce.java` and `SmarterBruteForce.java`. The optimization performed on the smarter version is the subject of the section 2.

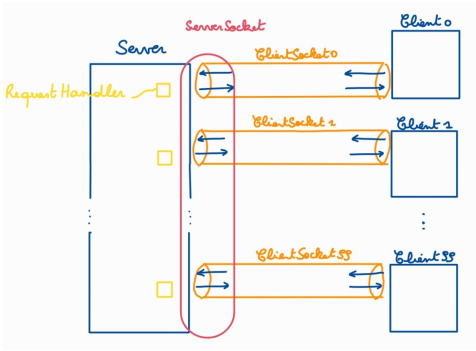


Figure 1: Client-Server network schema

Client The clients are implemented in the class `Main.java`. The client application works as follow. The client first encrypts the files located in the folder number given in the static variable `foldIdx`. A `folderIdx` of 0 represents the folder `files/Files-20KB`, a `folderIdx` of 1 represents the folder `files/Files-50kB` and so on in increasing size order. Once the encryption is performed, the client application stores the used passwords in a text file. This allows further client executions to skip the encryption and only load the passwords to compute the hash and length while creating the requests to send.

Once encryption is done or passwords loaded, the client application begins a loop to sequentially establish client connections. While the number of clients connections created does not reach the expected number `nClients`, the client application continues multiple clients connections. Each client has its own socket returned while connecting to the server and is identified by a unique number `iClient`. The inter-request time follows an exponential distribution with rate specified in the static variable `RATE`.

Once a `TCP` connection is established between a client and the server, a `ClientThread` is created and started. The request is then directly sent to the server using the derived streams from the `clientSocket`.

From the moment the request is sent until the server responses, the client waits. When the response is received, the client stored the decrypted file in the `tmp` directory. Once all the response have been received, the client application is finally turned off.

2 Server optimization

In order to optimize the client, the list of most commonly used passwords provided is used. Before listening for incoming connections, the server reads the file `10k-most-common_filtered.txt` and computes in advance the hash of each password. It stores the mapping between hash and password in the variable `dictionary`.

The smarter version of the server uses the dictionary created to perform a brute-force attack combined with a dictionary attack. The idea is the following. When the hash of a password is received, the server first checks

if the previously computed dictionary contains this hash. If so it returns the corresponding string which is the password to find.[1] Otherwise, a brute force approach is used.

We decided to make those optimizations to reduce the time the server will take to find the password corresponding to the hash, because lots of passwords used in real life are in the dictionary, so we just have to do a get in a dictionary instead of a whole brute-force. With that improved version, we expect the smarter version to be faster than the dumb version.

3 Measurement setup

Network In order to measure the performance of the client-server application, two different physical machines have been used. The server was located in *Incourt* (*Walloon Brabant, Belgium*) and the clients were located in *La Louvière* (*Hainaut, Belgium*). The network latency between the two was measured using the `ping` command. On average the latency between the 2 physical machine raised at 20 ms.

Hardware On the server side, an *Intel(R) Core(TM) i5-11400H* CPU was used with a clock frequency of 2.70GHz. This CPU has 6 cores and 12 logical processors. On the client side, an *Intel(R) Core(TM) i7-6500* CPU was used with a clock frequency of 2.50GHz. This CPU has 2 cores and 4 logical processors.

Software First of all, we modified the `Request` object to contain a new attribute `requestId` to uniquely identify a request. This attribute was used in the dictionary `startTimes` to keep in memory the moment when each request was sent. The server replies with the decrypted file, its length and the corresponding `requestId`. By substracting the sending time to current time at the reception, we could compute the response time for a specific request. Secondly, to evaluate the CPU and network load, we used the *Windows' task manager* on the server side.

3.1 Strategy

Requests Each of the 100 clients makes a request encrypting file of 20kB and then waits for the answer from the server. The inter-request time between clients is exponentially distributed with a rate specified in the variable `RATE`.

Encryption Around 70% of the 100 request over 20kB files are encrypted with a random lowercase-letters password of wanted size. The remaining files are encrypted with a password of the same size that we found in the given `10k-most-common_filtered.txt` file. We wanted to simulate a real-life example (where 30% of the passwords would be in the most common file). These percentages are probability, not ratios.

Scenarios We did 4 different scenarios. For each one, the studied measure is the average response time. The first two scenarios study the evolution of the response time depending on different requests rates[2] : [1, 20, 40, 60, 80] [requet/s]. The two other scenario study the evolution of the average response time according to the difficulty of the server's task : [3, 4, 5, 6] (password size). By increasing the password size, we expect the server to slow down due to the increased CPU work.

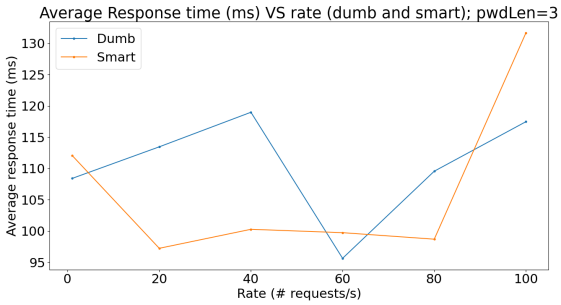
Version comparison For each scenario, we ran the dumb brute-force and the smarter brute-force with the same password used for encryption to compare the performance.

3.2 Results

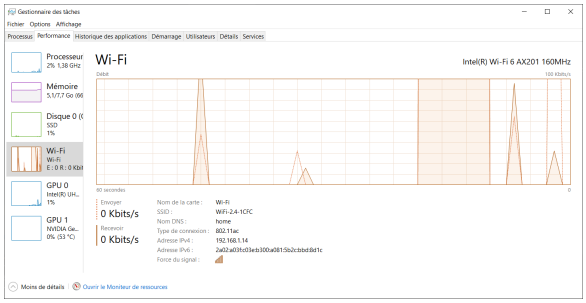
First scenario In this scenario, we fixed the password length of 3 and we varied the rate of the exponential distribution determining the inter-request time.

[1] Technically, SHA-1 collision can occur. This means that 2 different passwords could have the same hash. However those situations are so improbable that we did not even consider those cases.

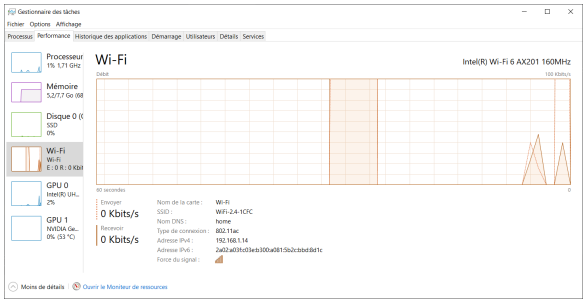
[2] Actually the rate λ is not a constant rate, the inter-request time is a random variable following an exponential distribution of rate λ. So the mean inter-request time is $1/\lambda$



(a) Dumb VS. Smart, pwdLength=3



(b) Network load (dumb, rate=20)

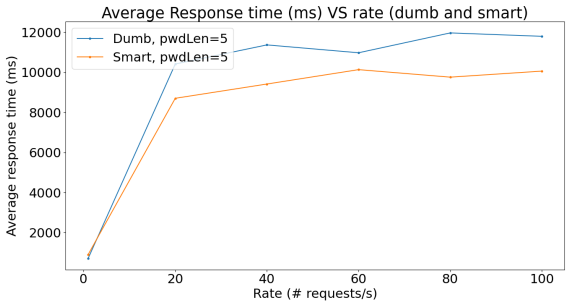


(c) Network load (dumb, rate=80)

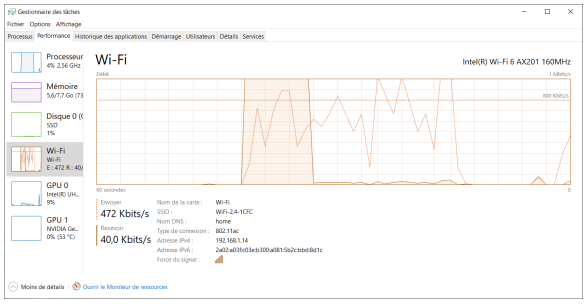
Figure 2: Evolution of the average response time as a function of rate for passwords of size 3

Discussion Related to the figure 2a, the first thing we can observe is that the smarter version of the server is not always better than the dumb version. This can be explained by the fact that the password length is only of 3. The required CPU work for password of such length is not really intensive. The gain brought by the dictionary search is also negligible. Related to the figures 2b and 2c, we can observe that the time during which the network load stays at 100 kbit/s is longer for the situation with request rate of 20. This corresponds with what we expected. For a higher request rate, by definition the number of requests sent each second is higher and therefore more requests are sent in the same given period of time.

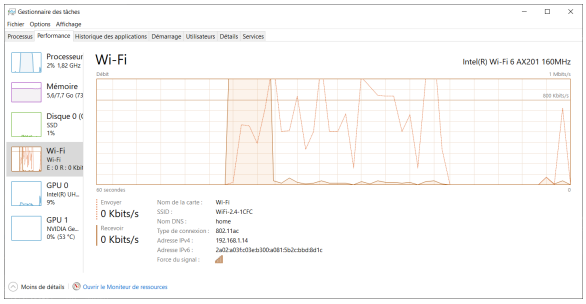
Second scenario In this scenario, we fixed the password length of 5 and we varied the rate of the exponential distribution determining the inter-request time. We also recorded the network load for rate of 20 and 80.



(a) Dumb VS. Smart, pwdLength=5



(b) Network load (dumb, rate=20)



(c) Network load (dumb, rate=80)

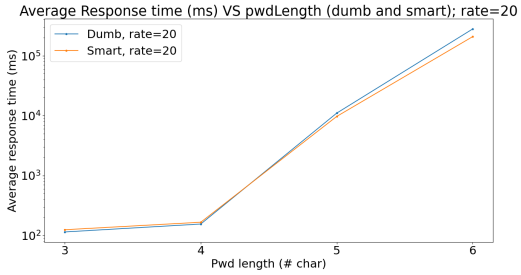
Figure 3: Network utilization

Discussion This scenario is more faithful to our expectations. We can see on the figure 3a that the response time globally increases with the rate: especially from rate 1 to 20. We think that with a rate of 1, the server is still able to treat the requests without being overloaded, the `threadPool`'s queue does not (or barely not) exceeds 6 (which is the server's number of threads) the queue is stable. With a rate higher than 20, it is no more the case. Indeed, with a password length of 5, the service time is no more negligible anymore. A second thing to note is the efficiency of the smarter version. As the figure 3a shows, the average response time is greater in the dumb version. The smarter version has respectively a [14.7%, 18.4%, 7.7%, 17.2%] smaller response time than the dumb version for rates [40, 60, 80, 100]. That is what we expected.

On figure 3b and 3c the time period dedicated to reception is smaller with a rate of 80. The time interval during which the server responds (in dotted lines) is much bigger in both cases due to the large service time on the server side. From these two scenario we can deduce that **a better CPU results in a shorter average response time**. Indeed, sending or receiving the requests faster will not improve, or very slightly, the performance in those two scenarios.

Remark If we decided to send files of 50MB instead of 20kB, the application would become more network intensive, especially for high rates. This would highlight the impact of the network on the server's performances.

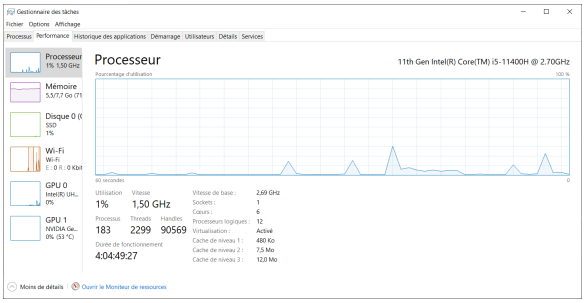
Third scenario In this scenario, we fixed the rate of 20 [request/s] and we varied the length of the password (from 3 to 6) used to encrypt the files. We also recorded the CPU load for password length of 3 and 5.



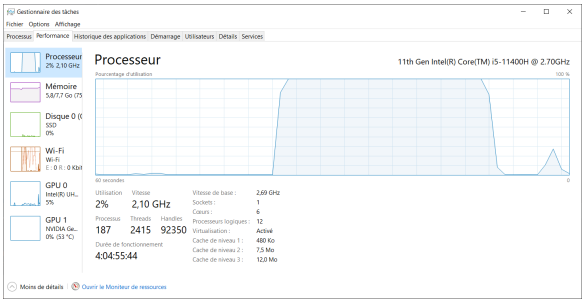
(a) Dumb VS. Smart, rate=20

Password length	Average response time [ms]
3	114
4	155
5	11 018
6	279 653

(b) Average response time with password length evolution for the dumb version



(c) CPU load (dumb, pwdLen=3)



(d) CPU load (dumb, pwdLen=5)

Figure 4: CPU utilization during third scenario

Discussion Related to figure 4a the average response time slightly increases with the password length from 3 to 5 and then significantly increases from 5 to 6. Indeed each additional letter in the password multiplies the number of possibilities to test during the hash reversing by 26. The time complexity of the brute-force is thus $O(26^n)$ where n is the number of letters. For 6 letters, it makes 308.915.776 operations in the worst case. With a rate of 20 and 5 or 6-letters password, the queue is again not stable and the CPU will again be the most important factor. The figure 4d shows that the CPU becomes overloaded for password of length 5. We can notice that the smarter version improves the response time as some requests will be resolved almost instantly (with just a get in a dictionary).

Fourth scenario This scenario is almost the same as the third one expect that we fixed the rate at 80 [request/s].

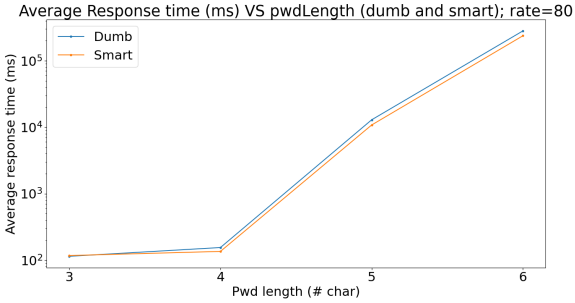


Figure 5: Dumb VS Smart, rate=80

Discussion We can see here that this scenario is very similar to the third one. For 5 and 6-length password, it is a totally CPU-dependant problem.

4 Modelling

4.1 Queuing station model chosen

Given the implementation of our server which uses multiple `TCP` connections and a `ThreadPool` to handle the incoming requests, the most appropriate model is the M|M|m queue model. In fact, our application fulfilled all its requirements.

- The `ThreadPool` disposes of a single queue. Given the number of requests sent to the server and the fact that none are discarded, the queue behaves as it has unlimited space in the queue.
- The `ThreadPool` has 6 threads which makes the server behaves as they were $m = 6$ service stations.
- Requests are proceeded as they come in which correspond with the *First Come First Served* service discipline of the M|M|m queue model.
- We designed the clients such that they send their requests with an inter-arrival time following an exponential distribution with rate λ.
- Given the bruteforce method used, the job service times are exponentially distributed with rate μ.

4.2 Theoretical average response time

We decided to compare the model with the experimentation of the third scenario (password length evolution with a rate $\lambda = 20$). The number of services station m = 6 (number of threads). And we are trying to find $\mathbb{E}[R]$ the theoretical mean response time.

In order to compute the expectation of the service time, we change our implementation such that a new request is only sent when the answer of the previous one is received. This removed any queuing delay. We computed the mean service time over 100 requests for each password length (3, 4, 5, 6). Given the expectation of the service time, we derived the service rate $\mu = 1/\mathbb{E}[S]$, the utilization $\chi = \frac{\lambda}{(m \cdot \mu)}$, the probability to have 0 customer in the system $\pi_0 = (\sum_{i=0}^{m-1} \frac{a^i}{i!} + \frac{a^m}{m!(1-\chi)})^{-1}$ and finally the theoretical average response time $\mathbb{E}_{th}[R] = \frac{1}{\lambda}(a + \frac{\chi a^m}{(1-\chi)^2 m!}\pi_0)$ with $a = \frac{\lambda}{\mu}$.

Password length	$\mathbb{E}[S]$ [s]	μ [services/s]	χ	$\mathbb{E}_{th}[R]$ [s]
3	0.238	42.052	0.079	0.048
4	44.82	22.311	0.149	1.618
5	613.43	1.630	2.045	x
6	15506.06	0.064	51.687	x

The theoretical average response time could not be computed for password length of 5 and 6 due to the utilization χ which is bigger than 1 in those cases. In such situations, the arrival rate λ is higher than m times the service rate of a single service station μ. In other words, the system can not served the incoming request as fast as they come in. There is accumulation inside the queue and the queue is no more stable. Thus, the *Continuous-Time Marchov Chain* stationary distribution cannot be found. To obtain consistent theoretical average response times for password of those lengths, we should reduce the arrival rate λ. This way, we would have $\chi < 1$ and the queue would be stable.

The figure 6 plots the theoretical average response time in comparison with the experimental average response time. As we can see, the theoretical and experimental values are consistent for password length of size 3 and 4. The order of magnitude is the same. The small difference could be explained by the fact that different passwords have been used to encrypt files during the calculation of the mean service time and during the measurements of the mean response time.

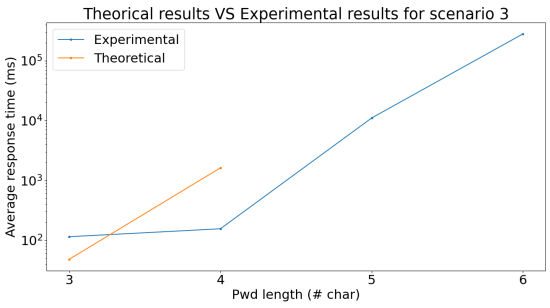


Figure 6: Theoretical results VS experimental ones for scenario 3