Project : measurement & modeling
LINGI2241

Piotrowski Maciej (11411700)
maciej.piotrowski@student.uclouvain.be
Dubuisson Tom (70901700)
tom.dubuisson@student.uclouvain.be

1 Introduction

This document is a report for the project of the LINGI2241 course which consist in the implementation of a simple client/server application, the evaluation of his performance and his modeling. Hereafter you will find a description of our implementation including the implementation of the performance evaluation, an explication of the method we use to improve the performance of the protocol with a discussion of the result, a discussion of the experiments we made and last the modeling of our application. You will also find a user manual in annex.

2 Implementation description

In this section we will describe the implementation of the client/server application. We use *Java Socket* to implement the communication between the clients and the server.

2.1 Client

Our client implementation is done in 2 parts. Either you want to run it with random requests generation or you want to put them manually. If you want to run a client with random request generation, there is a lot of parameters that you need to give in arguments in order to do so (you can see the user manual in annex for more details). But the main running process is the same for both versions. The client will create a new socket connection on a TCP port and an associated incoming and outgoing stream. Then it will "listen" until some request is made. Next thing, it will launch a thread that will retrieve the request, send it to the server, wait for the response and finally save the operation time.

2.2 Multi server (with the basic protocol)

When the server starts, it will first store the "dbdata.txt" file in a structure (2D array) in the main memory. The 2D array consist of N (the number of lines in "dbdata.txt") columns and 2 rows, so that it can store the type and then the sentence. Next, it will open a server socket on a TCP port and with a limited backlog to 101 clients connections. It will listen to client connection and once any occurs, it will launch a new *MultiServerThread*. Our server also launch a thread while it's running to listen if "save" is typed in so that it can save the mean server service time since the start into a text file. Our server contains also a thread pool to be able to process several request at the same time.

2.3 Multi server thread

The *MultiServerThread* constructor take the socket from the server and the specified protocol. Each *MultiServerThread* represents one client connection. It creates an incoming and outgoing stream like the client. Once it receive a request (on the incoming stream) from the client, it checks if the pattern is correct "<type>;<regex>" and run the search (with the protocol) using a thread from the server's pool. If all thread are occupied, it will wait until one is free. Once the protocol process finished, it sends back the response by his outgoing stream.

2.4 Protocol

The protocol operates on a request received by the *MultiServerThread*. First, it splits the request into 2 parts: the types and the regex. It will then go over all the first row of the "dbdata.txt" 2D array to see if he can match one of the types provided in the request. If there is a match, it will check if the sentence part in the array contains the request regex. If it's the case, it will then check if it didn't already found the same sentence and add it (or not, if it was already found) to the response. After it iterates over the entire array, it returns the result containing unique sentences divided by "newlines".

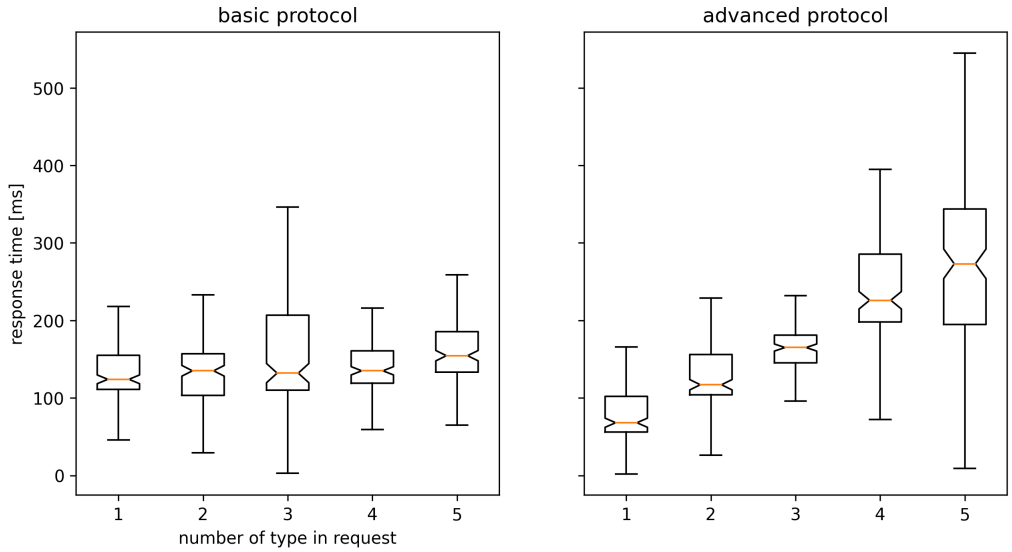
3 Optimization method

3.1 How we optimized the server?

When a client make a request, he's not always asking to look for all types but despite that, the server in the basic version is looking through all the array to find the matching responses. So the first improvement that we've made is to store the "dbdata.txt" file in an *HashMap* so that the server will now only iterate over sentences with types matching those in the request. The second improvement it's basically checking if there isn't any duplicate (same regex and type) while storing the file in the memory instead just before sending back each response. Since, theoretically, there can be the same sentence but with a different type, we need to still check the response before sending but it's a lot shorter than before. It takes just a bit longer to start the server but it isn't noticeable.

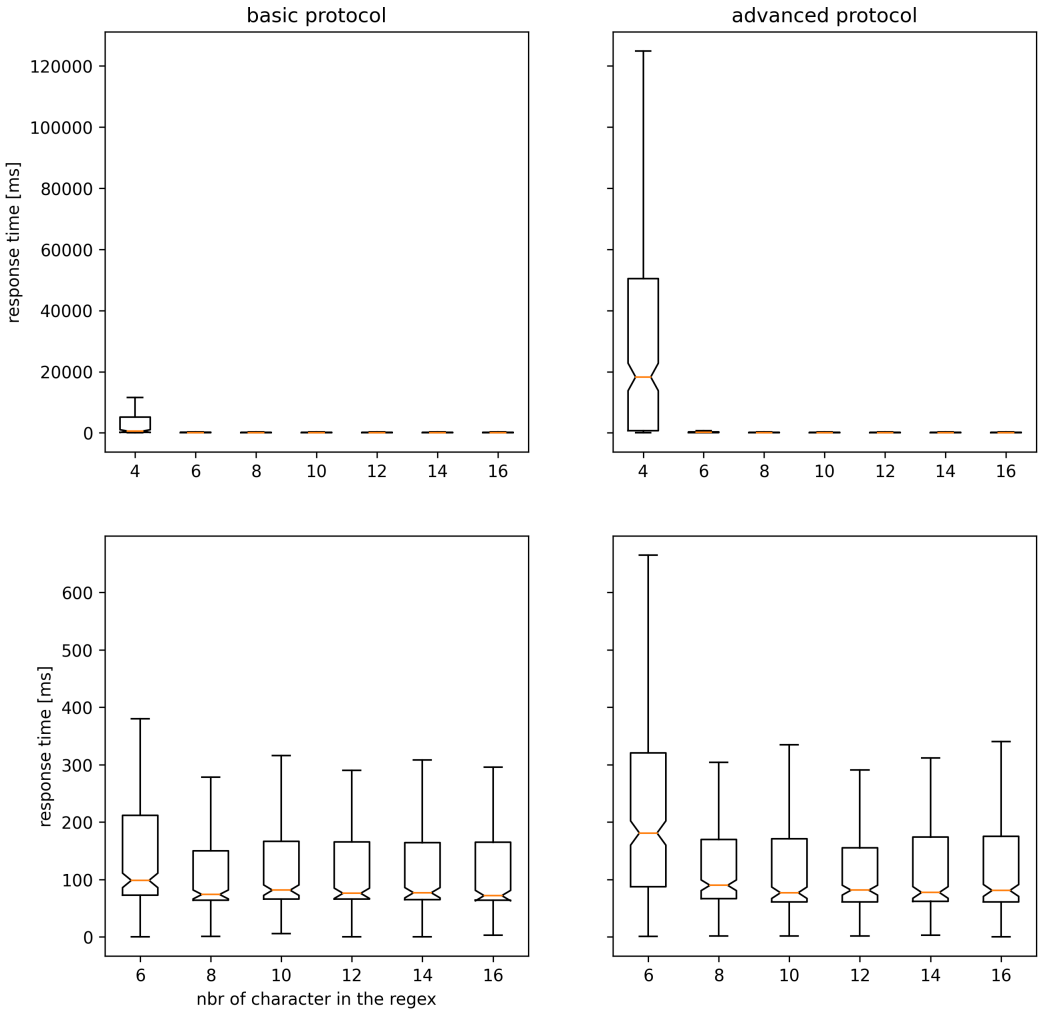
3.2 Impact of the optimization process

First we tested our basic and advanced version with various type complexity, with 10 repetitions of each experiment :



As expected, we can observe that our advanced version is quite faster when there is 1 or 2 types. With 3 types in the request the mean time is quite similar in both versions. The problem is, that if the request contains 4 or more request types, our advanced version becomes slower than the basic one.

Then we tested our basic and advanced version with various regex complexity (the complexity is higher when the number of characters in the regex is small), with again 10 repetitions of each experiment :



The graph show us an unexpected behavior in our advanced version with 4 as the regex complexity. Indeed, our advanced version is a lot slower when the regex contain only 4 characters. We don't know the reason for this but a possible explanation is that our algorithm making random requests made some more difficult 4 characters requests for our advanced version than the basic one. For example, the request sentence "etos" gives only a few response lines but "that" is a lot more frequent. Thus the client requesting "that" will wait a lot longer thus it need to transfer all the data trough the network.

For the other regex complexity we observe a certain similar curve between the 2 versions and it's pretty constant with a mean response time of more or less 100ms.

After seeing those results, it seems that the response time with our advanced protocol depend linearly with the number of type in the request and is faster than the basic one when there is one or two types in the request. But it provides no gain when the regex is more difficult to match.

4 Measurement setup

- ***Hardware :***
  - *Server side :* Acer Swift 3 SF314 with an Intel® Core™ i5-7200U CPU @ 2.50GHz × 4 ;
  - *Client side :* MacBook Air (M1, 2020) with an Apple M1 core ;

- ***Network :*** Network provided by Telenet, ping of 13ms, download speed of 76mbps and upload speed of 29mbps (results given by speedtest.net) ;
- ***Operating System :***
  - *Server side :* Ubuntu 18.04.5 LTS ;
  - *Client side :* macOS Big Sur Version 11.1.

5 Workload used in experiments

Hereafter, you will find a detailed description of our request generation system and of our evaluation process, but before those we will discuss the default parameters used during the experiments.

The default parameters ware those :

- a regex of **40** characters ;
- **1** type per request ;
- **6** requests per client, all different ;
- a mean inter-request time of **500** ms ;
- **5** repetitions of each experiments ;
- **5** simultaneous clients ;
- **8** service threads ;
- a backlog of **101**.

About the request generation system, we have implemented a class called *Sequence* which generates sequence of requests. They are several parameters that allow to custom those sequence : the number of character in the regex, the number of type in the requests, the number of request in a sequence and the number of different request in the sequence (we do not used this one in our experiment). Each sequence is generated with a *seed* so we can get the exact same sequence several time, because the all process of generating sequence is random. The request regex is a sub-part of a random sentence found in the "dbdata.txt" file and the type of the request is at least the one of the sentence.

The evaluation process is implemented in a class called *Evaluation* that measure the response time for each request send and save those times in a file. In addition to the parameters for the request sequence generation, there is specific parameters for the evaluation process : the mean inter-request time, the start value of the variant, the end value of the variant, the value to add to the variant at each step, the number of repetition of each experiment, the number of simultaneous client and obviously the variant (the parameter on which to vary). The process have three imbricate *for loop* the first one to increment the variant, the second to repeat one experiment (a state where each parameter do not vary) and wait that each client have receive all their response, and the third one to launch the clients. At the end of the process the result his save into a text file to be plot using a python script.

On the server side, it is possible to save the mean service time manually by typing "save" into the server console, the mean is compute over each service process perform between two save or between the launch of the server and a save. It is not so useful when using a variant in the client evaluation side.

We also used some tools to measure the CPU and network load on Linux. We used "top" for the CPU and "iftop" for the network. We tested it with a number of clients variation and the results were as expected. The bigger is the number of client, the bigger is the size of the data exchanged via the network and bigger is the CPU use.

6 Results and Modeling

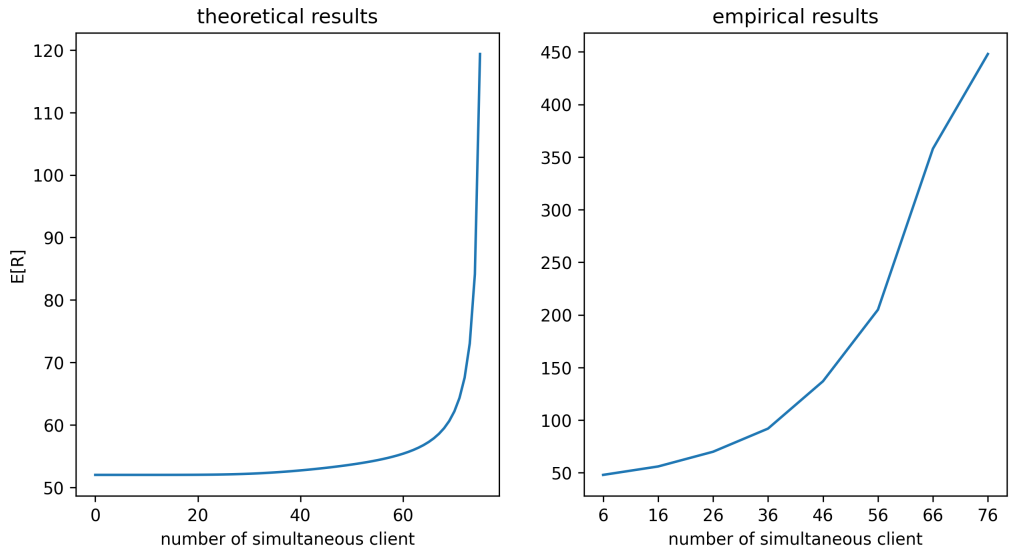
6.1 Modeling

Our implementation represents a M|M|c|k model service station. We set up the c to 8 (number of service threads) and the k to 101 (number of client's connection i.e. the backlog). Since we are never using more than 101 clients, we can consider our implementation as a M|M|m queue. The mean service time is 52 ms, the arrival rate is equal to n/500 (where n is the number of simultaneous client). With that we calculated χ and we can evaluate the queue stability ;the queue is stable if,

$$\chi \leq 1 \iff \frac{n}{m} \leq \frac{500}{52}$$

That means that with $m = 8$ our queue is instable when $n \geq 77$. In the experiments it works because we limited the number of requests. So if the clients were running non-stop, the requests would arrive faster than the service station sending back the response and that would overload the queue. To avoid that, we should decrease our arrival rate.

Concerning the estimation of the mean response time $E[R]$, we do not have a perfect matching between the theoretical and the empirical results but the tendency seems exponential in empirical result as in the theoretical results. Maybe this difference due to other running process on the server side.



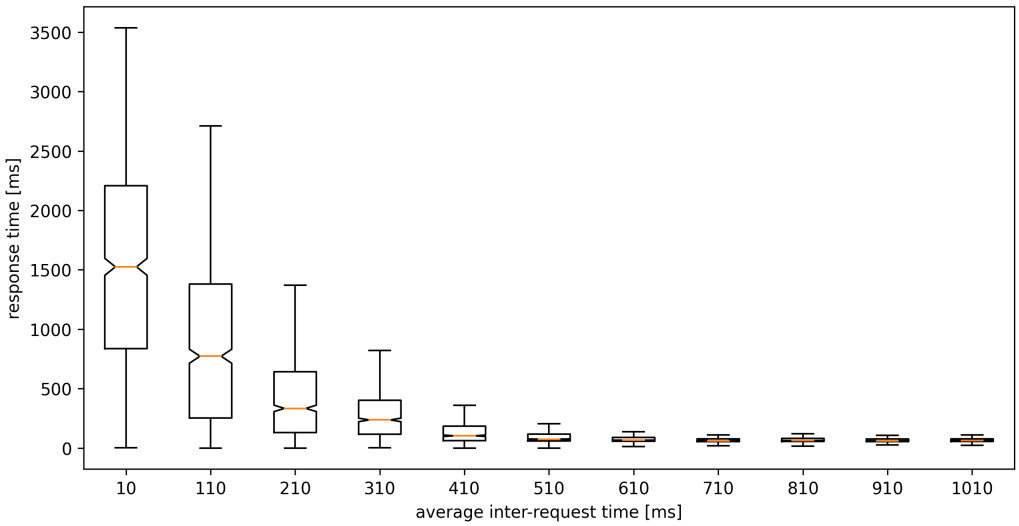
In this experiment, each client had only one request to send so the number of client match the number of request process by the server ; we do it that way because it seems to us that the model does not take into account that each client can make several request.

6.2 Results

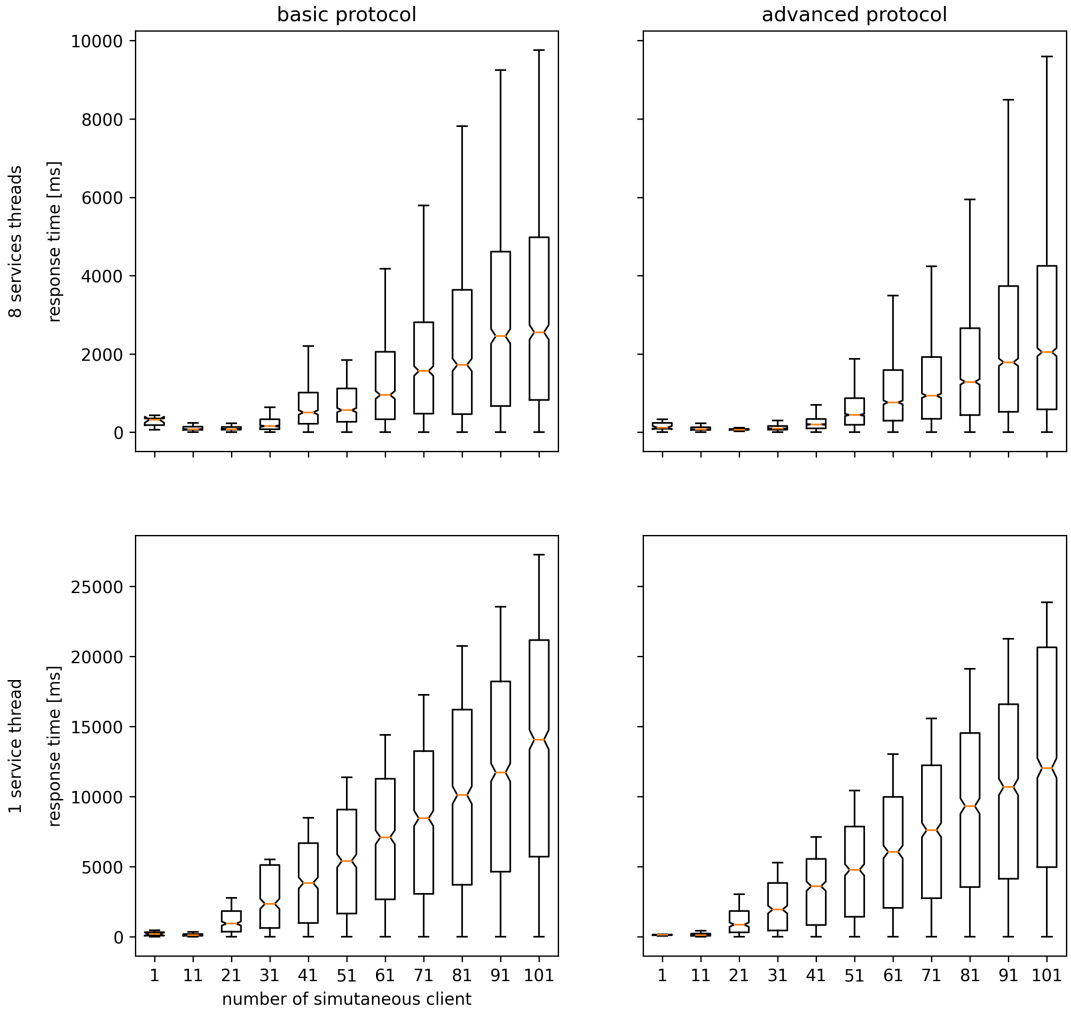
The first experiment that we made was to study the impact of the mean inter-request time on the response time. This experiment was conducted with 30 simultaneous client with the advanced protocol, results are below.

In this case the queue is stable when the inter-request time is 195 ms.

We have choose 500 ms as default inter-request time after the seeing this plot, after that value the response time seems constant.



A second experiment that we made was to vary the number of client (within the backlog limit), the number of service threads and the protocol used by the services. Hereafter are the results :



Firstly, we can see that the advanced protocol is a bit faster than the basic but they have they follow the same tendencies. Secondly, the response time is almost constant for 1 to 51 simultaneous client with 8 service threads and between 1 and 11 simultaneous client with 1 service threads ; it is probably due to the stability of the queue.

7 Conclusion

To conclude, we are very proud of our project, it was for us a pleasant project to work on, with a workload well balanced and just enough challenges. It also allowed us to implement our own client-server application, test it and compare it to the theory we saw during the lectures. The testing and the obtained results helped us to understand more how a server works and the meaning of different values in theoretical calculations. We also enjoyed trying to upgrade it and we implemented a really nice random request generation. Despite that, there are still several ways to upgrade our application. We could improve our advanced version more, implement the automatic save of the service time (we only save the mean service time when manually requested right now) or change the client implementation so that each client would have only 1 thread to send the requests and 1 thread to receive the response instead of running a thread at each request made.

A Annex - user manual

This annex explain how to run our different executable.

First thing, a *MultiServer* need to be launch before launching a *Client* or an *Evaluation*.

A.1 Client - user manual

Here is a description of the different argument you can use when launching a *Client* :

- *[mandatory]* **-h** followed by the hostname of the server, the name of the computer it is launching on ;
- *[mandatory]* **-p** followed by the port number of the server ;
- **-f** followed the path to the "dbdata.txt" file, only required if the *Client* is in mode test ;
- **-t** launch the *Client in mode test* ;
- **-rc** followed by the number of character that will compose the regex of the request generated, only required if the *Client* is in mode test ;
- **-tc** followed by the number of type that will be in the generated request, only required if the *Client* is in mode test ;
- **-l** followed by the number of request the *Client* will send, only required if the *Client* is in mode test ;
- **-rv** followed by the number of different request that will be send by the *Client*, only required if the *Client* is in mode test ;
- **-pa** followed by the mean inter-request time in millisecond, only required if the *Client* is in mode test.

If you launch it in test mode, the *Client* will send automatically the request corresponding to the characteristic you choose ; otherwise it will wait that you type in a request in the correct format, you do not need to wait the response to type the next request.

A.2 Estimation - user manual

Here is a description of the different argument you can use when launching an *Estimation* :

- *[mandatory]* **-h** followed by the hostname of the server, the name of the computer it is launching on ;
- *[mandatory]* **-p** followed by the port number of the server ;
- *[mandatory]* **-f** followed the path to the "dbdata.txt" file ;
- *[mandatory]* **-rc** followed by the number of character that will compose the regex of the request generated ;
- *[mandatory]* **-tc** followed by the number of type that will be in the generated request ;
- *[mandatory]* **-l** followed by the number of request the *Client* will send ;
- *[mandatory]* **-rv** followed by the number of different request that will be send by the *Client* ;
- **-pa** followed by the mean inter-request time in millisecond, 1000 by default ;
- **-s** followed by the value at which the variant start ;
- **-e** followed by the value at which the variant end ;
- **-st** followed by the increment the value of the variant is increase at each step, 1 by default ;
- **-r** followed by the number of repetition for each experiment, 1 by default ;

- **-nc** followed by the number of simultaneous client in each experiment, 1 by default ;
- **-v** followed by the variable to vary on. It is supposed to be "regex_complexity", "type_complexity", "sequence_length", "request_variance", "pause" or "nbr_client". It requires *-s* and *-e*.

The result of the experiment will be save in the folder "experiments" with a file name corresponding to the parameter used, the format in which the result are saved is readable by our python plotter script.

A.3 MultiServer - user manual

Here is a description of the different argument you can use when launching a *MultiServer* :

- the first argument need to be the port number it will listen on ;
- the second one need to be the path to the "dbdatafile.txt" ;
- the third one need to be the name of the protocol, either "basic" or "advanced".

You can also save the mean service time by typing "save" in the command line, it will save it in file "experiences/Server_experience.txt" with the time at which the save was made.