INGI2241 Group Work 2

Performance analysis of client-server system

Crombé Henri (72580800)
Declercq Mallory (41661200)

December 2015

1 Introduction

For this project we were asked to implement a client-server system that allows a client to send requests of adjustable difficulty to a single server. Upon arrival of a client request, the server computes the answer and sends it back to the client. We implemented different client-server schemes in Java and measured the performance of each of them. The next sub-sections are dedicated to describe the server and the measurement setups used to collect our data.

1.1 Description of the server

In the context of our project, the server is responsible of computing the multiplication of two-dimensional matrix. The server receives requests which contains a matrix (M) and an exponent (P) and it returns the computation of M^P. In order to keep the protocol simple and analyzable, we chose to send matrix filled with real values and of fixed size (50 rows x 50 columns). Therefore we adjust the difficulty of a request by varying the exponent P while keeping a fixed size for the matrix. We made this choice because sending matrix with size bigger than 50x50 greatly increases the network transfer time and therefore, the time spent to measure response time.

To enable the communication between the clients and the server, the server uses a SocketServer class and clients connect to it by the mean of simple sockets. The clients and the server exchange serializable messages that we defined. When a client wants to send a request to the server, it has to create and send an object *Message*. The Message will contain the matrix and the exponent to compute and the server will send the answer by the mean of a Message containing the resulting matrix. The Message also stores information such as the sending time and the computation time (on the server) for the request.

1.2 Measurement setup

To perform our tests and measurements, we had the chance to run our implementations on a real server hosted by a friend. The server runs on Windows Server 2012 and is located in Belgium (+-50km from the client(s)). The server possesses two processors Intel Xeon running at 2.2 GHz giving a total of 16 virtual cores. The server also possesses 16 Go RAM and is connected to the Internet with an up-link of 6 Mbps and a down-link of 70 Mbps. All our tests have been made on this machine and were performed using the operating system Windows Server.

In order to get consistent measurements we used different methods :

Computation & Response time We used Java to compute these measurements. The measurements are stored into log files.

CPU load We used a tiny program called CPU load the get the average CPU load for a given process.

Network load The network load has been measured on the performance monitor of Windows Server. It allows to see the average network load of the last 60 seconds for a given process.

2 Task 2.1

2.1 Measurement #1

For the measurement #1, we were asked to measure the time our server needs to handle an individual client request. The graph below depicts the variation of the response time as a function of the request difficulty. The difficulty varies in function of P (P varies between 100 and 2000). We calculated the average response time for a given P by calculating the mean response time of 250 request. Therefore, for each P, we sent 250 requests one after the other and calculated the mean times values.

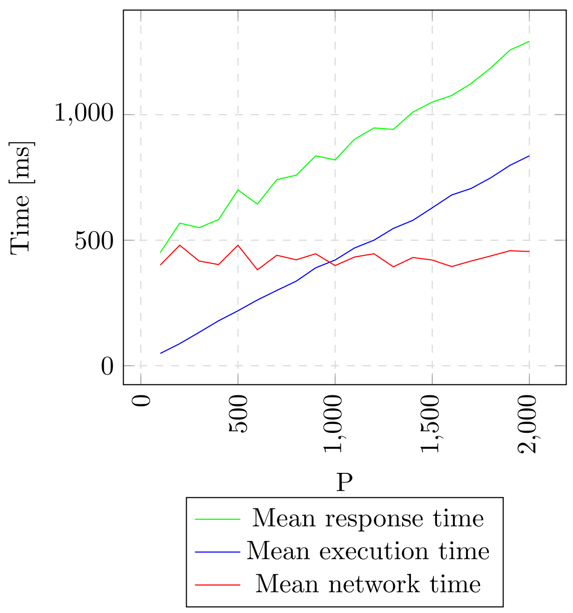


Figure 1: Impact of request difficulty on response time.

As the graph depicts, the mean network time is quite steady and doesn't influence that much the mean response time. What affects the mean response time is the execution time of the computation. As the execution time increases linearly and slowly, the mean response time follow the same dynamic. Therefore, the main contributor to the response time is clearly the variable P since the matrix size is fixed. If the matrix size is not fixed, the variation in the network time would be more unpredictable and would make the analysis far more difficult.

2.2 Measurement #2

For measurements #2, we implemented a load generator that simulates the behaviour of several clients. To do so, the load generator creates and runs client threads which are delayed by an exponentially distributed inter-request times. Each client thread is in charge of connecting to the server (socket streams) and to send only one request. The waiting delay between the creation of two client threads is computed by this formula : $\frac{ln(1-u)}{-coef}$, where u is a random number between 0 and 1 and $coef$ is a fixed chosen number used to influence the arrival/request rate (i.e. λ). This way of doing allows us to simulate the behavior of many independent clients that send requests with exponentially distributed inter-request times.

By making some tests and by seeing the potential results, we decided to parametrize our load generator as follow. First, we decided that the load generator varies the difficulty of the requests with a P randomly chosen between 500 and 1000. By analysing the measurements #1, we know that the average response time for this interval of difficulty will oscillate around $\approx$ 700 milliseconds (execution + network time for one request). This means that the average service time for the requests will oscillate around $\approx$ 700 milliseconds. Therefore, our service rate (μ) for the server will oscillate around $\approx \frac{1}{0.7} \approx$ 1,4 requests per second. By choosing P in this interval, we ensure that the computation will not take too much time and, inversely, not to few time. The goal here is to have requests of varying difficulty, but we also want the computation time to be contained in realistic, determinable interval (measurements are really time consuming).

Then, knowing that the server will have an average service rate of 1,4 req/s, we decided to vary the arrival rate λ between 0.2 and 4 request(s) per second. So that, at first, the arrival rate won't overload the server ($\lambda < \mu$) and we will see what happen when λ becomes greater than μ.

Thirdly, since we need to measure the average response time in function of the request rate, we decided to send 200 requests a variable difficulty for each request rate we tried. This means that for a given request rate, we calculate the mean response time of 200 requests sent with exponentially distributed inter-request times.

We used the load generator described above to make the graph of figure 2. The goal is to evaluate how the response time is influenced by the request rate. The request rate λ varies between 0.2 and 4 req/s and the average response time for a given λ is computed over response time of 200 different requests.

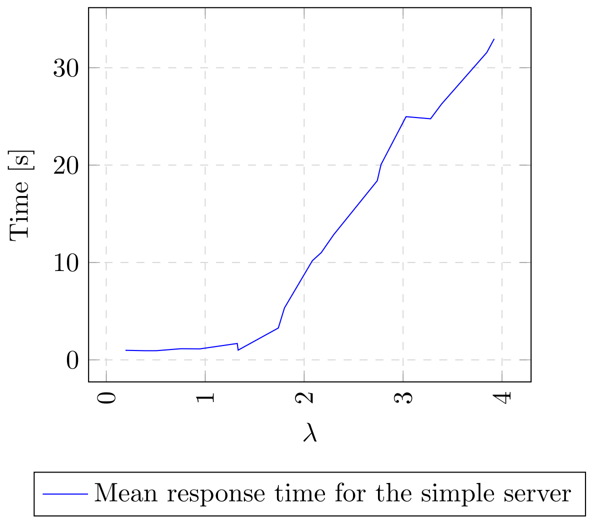
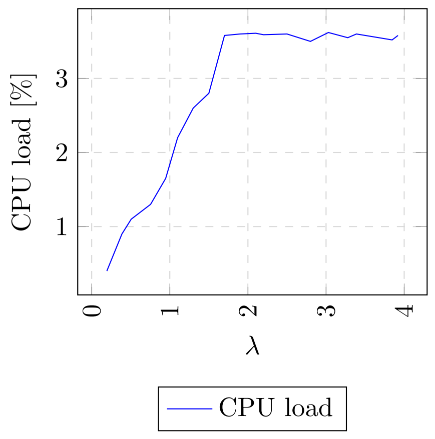


Figure 2: Impact of request rate on response time.

As depicted in the graph, the average response time is quite low and steady when the arrival rate is below 1,4 req/s, this is due to the fact that the server isn't overloaded and can process the requests faster than they arrive. When the arrival reaches 1,4 and grows up, the average response time grows dramatically. This happens when the server cannot process the request fast enough. Then, the requests are placed in the OS queue buffer and waits for their turn. This adds delays to the response times and therefore to the average response time. The more messages the loader send for a given $\lambda > \mu$, the greater the average response time will be.

Regarding the CPU and network load, we can see that their load increases sharply until a fixed value ($\approx 3.5\%$ for the CPU and ≈ 100 KB/s for the network). This fact is easy to explain because when the request rate is low, the requests arrive slowly to the server and then the processor spends most of its time waiting for request (idle state). When the request rate increases, the processor has more requests to process for the same amount of time, and then, the time it spends waiting for request becomes smaller and its processing time becomes larger (CPU load increases). Increasing the request rate will increase the CPU load until a fixed value that corresponds to the average processing load of a request by the CPU when there are so much request arriving to the server that it has always at least one request to process (no idle states). The above explanation is also true for the network load except that the fixed value corresponds to the average network load when the server always has a request to process.



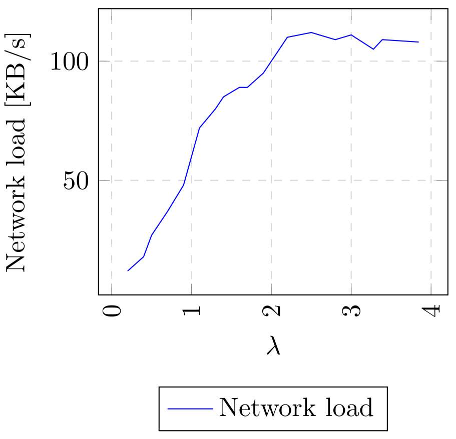


Figure 3: Impact of request rate on CPU/network load.

2.3 Modeling #1

In order to compare the response times of measurement #2, we use a M/M/1 queueing station model. Our queueing station model is parametrized as follow in order to be relevant with our client-server implementation:

- Single station because we use a single server to process the requests.
- The arrival rate of requests (λ) is determined by a Poisson distribution.
- The Buffer is infinite because we have configured the server with a backlog larger than the total number of requests sent.
- FCFS scheduling policy in order to match the behaviour of the built-in queue used by the server.

Based on this queueing model, we computed the mean response time with this formula: $E[R] = \frac{1}{\mu-\lambda}$ where μ (service rate) is equal to ≈ 1.4 req/s (cf. measurement #2) and λ (arrival rate) varies from 0.2 to 1.39.

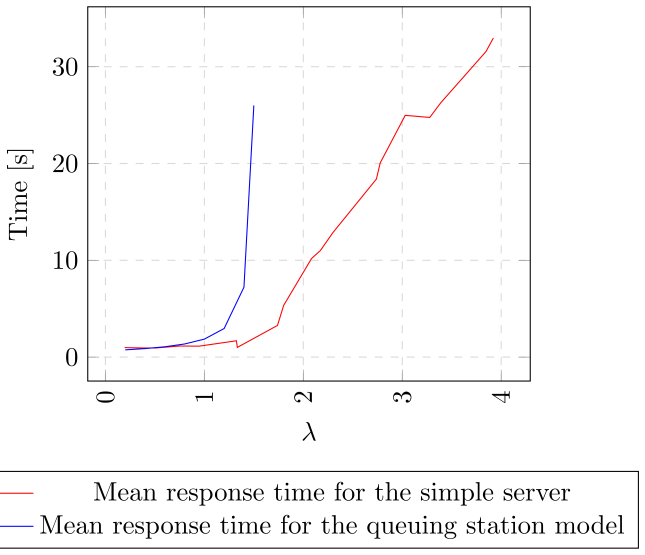


Figure 4: Impact of request rate on response time.

As depicted in the plot, the model confirms the observations made during measurement #2. When the arrival rate (λ) exceeds 1.4 req/s, the server is not able to process the requests as quickly as they arrive and then the mean response time increases drastically.

2.4 Measurements #3

We have improved the previous model by adding a cache at the server side. Our cache runs with a trivial replacement policy (FIFO) and its size is fixed to 200 entries. For the measurement, we take as hypothesis that the server always receives the same matrix to compute, only the exponent varies (from 500 to 1000). We have implemented the cache using a Hashmap where the key represents the difficulty of the matrix computation (exponent P) and the value is the result returns by the server (M^P). Here, the idea is to avoid the re-computation of M^P by looking inside the cache if the result is already known. This way of doing should improve the performance of the server. By looking at the plot below, we can effectively see that the response time is improved thanks to this modification. In our measurements, the mean response time of the server with a cache is much better than the mean response time of the basic model because the size of our cache is quite big and then the probability to have a hit is high. With a cache of smaller size, we expect better results than the basic model but less good than those we have at the moment.

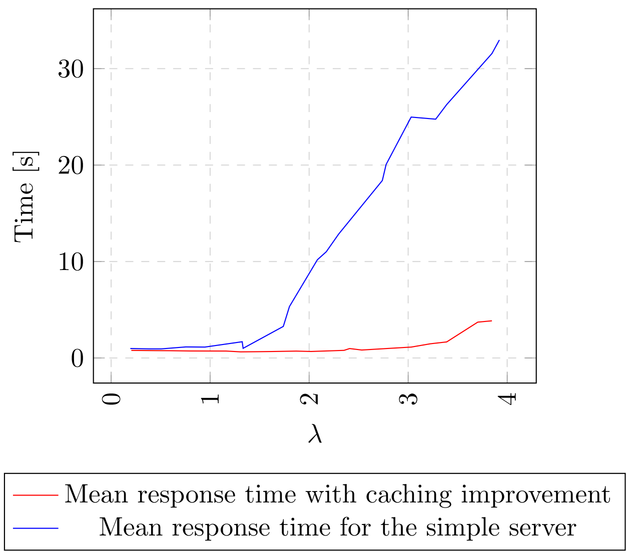


Figure 5: Impact of request rate on response time.

3 Task 2.2

For second part of the project, we extended our single-threaded server to process several requests in parallel. To achieve this, we implemented a class ServerHandler that extends the class Thread which receives a socket as argument and is in charge of receiving the client request and to compute the result. The main function of our multithreaded server is in charge of synchronizing the different threads. Upon arrival of a client on the SocketServer, and if there is free thread remaining in the pool, a ServerHandler thread is launched and this new thread computes the answer and sends back to the client. We computed the mean response time in function of the request rate for a server running 2,4,8 and 16 threads. We also analyzed the mean CPU and network load in function of the request rate with different number of threads. The figure 6 depicts our measurements of the mean response time with multi-threading while figure 7 depicts the CPU and network measurements.

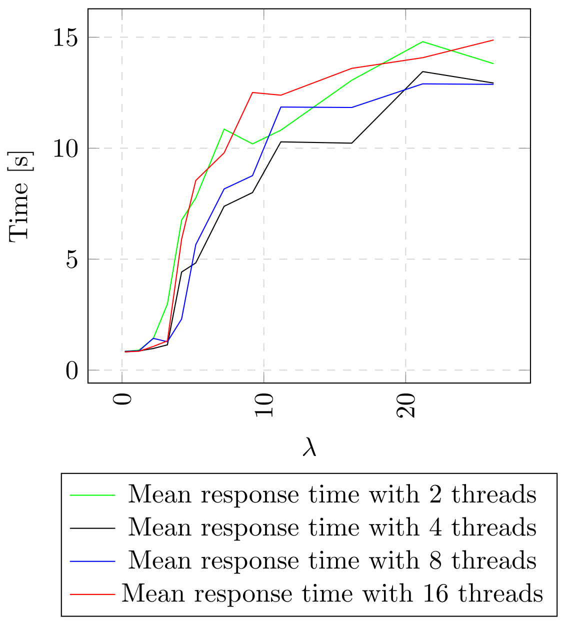
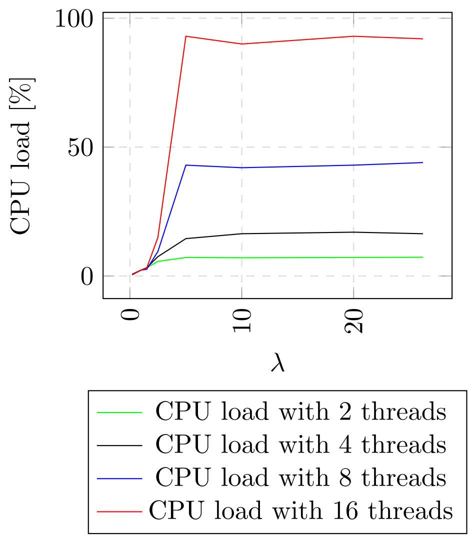


Figure 6: Impact of request rate on response time with a varying number of threads.

As the figure 6 illustrates well, the use of multi-threading in the context of our project brings quite an improvement. First of all, our multi-threaded server improves the request rate the server can handle. While the single-threaded server could only serve 1,4 clients per second before being overloaded, the multi-threaded server seems able to handle up to 3,2 req/s before experiencing real delays. Then, the service rate of the server has been multiplied by a factor of 2. However, the service rate of the multi-thread server doesn't seem to grow with the number of threads. Indeed, we can see on the graph that the maximum service rate for 2 threads is quite the same as for 4, 8 and 16 threads. Then, using the multi-threading increases the performance of the server but using more than 2 threads is not useful in term of response time and service rate. We suspect that this is due to some limitation induced by the network. As the CPU load graph shows, the CPU consumption increases with the number of threads while the average response time doesn't decreases. This means that the network is the bottleneck in our case. The server cannot send results as fast as it computes them leading to a limited service rate.

As regards to the CPU load when using multi-threading, we clearly see that with more threads, there is more CPU consumption. Note that the scale for the CPU is in percent of the total of the 16 cores in the server. In the context of our project and our protocol, we should say that using more than 2 threads would be a waste of CPU consumption since the benefit in term of response time is almost null. Regarding the network load, the graph shows that network load increases with multi-threading until a certain point. As the server is limited in upload, it cannot sends the results back as it computes them leading to a limited service rate. We evaluated that the service rate should have raised up to $16{*}\mu$ (≈ 22) req/s with 16 threads and we are limited to $\approx$ 3,2.



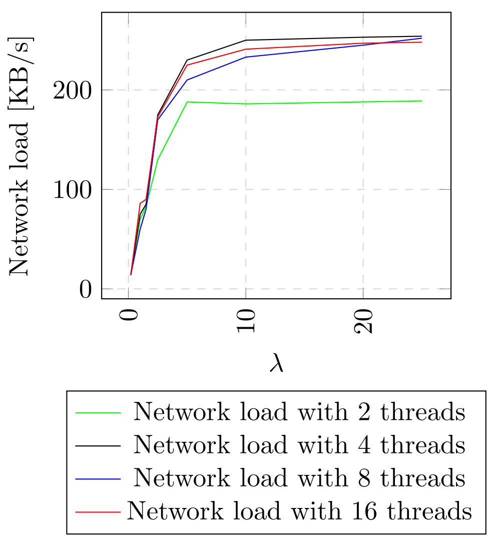


Figure 7: Impact of request rate on CPU/network load with a varying number of threads.

4 Implementation memo

Our project have been made using Eclipse Java and is divided in three different package :

Task_1_1 This package includes the implementation of our simple server and the client used to generate the measurement #1.

Task_1_2 This package includes the implementation of our simple server (Server.java) and the implementation of our improved server with caching (ServerWithCache.java) and our load generator for this package (Client.java).

Task_2_1 This package includes the code for the multi-threaded server (ServerMultithread.java) and the corresponding client (Client.java) used for the measurements of figure 6 and 7.