Université catholique de Louvain

LINGI2255: Architecture and performance of computer systems

Group Work 2 : Performance Evaluation Model

Jérôme Lemaire
Grégory Vander Schueren
December 24th, 2015

Introduction

For this second assignment, we were asked to design a simple client-server system and measure its performance. The goal was to identify how the various components contribute to the overall performance of the system and perform a model-based evaluation. This report starts by giving a brief overview of the overall program architecture. Next, we analyze how the response time of our server varies with the difficulty of an individual computation request. We then turn our attention to the behavior of our server under an increasing load. To this purpose, we simulate many independent clients and measure the average response time, the CPU load and the network load of the server. We compare these empirical results with the prediction of our queuing station model. We also suggest an improvement to better scale the server, implement it and measure its efficiency. Finally, we extend our server to perform parallel processing of the requests and take advantage of the multiples CPUs available on most modern computers. We extend our queuing model to reflect this new architecture and contrasts its performance with the former implementation.

1 Program Architecture

1.1 Overview

This section briefly describes the purpose of the main packages that compose our Java program :

Server : This package contains two classes : `Server` and `Computation`. The `Server` class implements our simple HTTP server that responds to the "/compute". In order to easily automate our simulations, the server parameters can be controlled remotely via the following requests :

"/start_recording" starts tracking the CPU load and performance of the cache.

"/stop_recording" stops recording the performance and responds with the various measures.

"/set_threads_count" the server restarts with the specified number of threads.

"/enable_caching" and "/disable_caching" starts and stops the cache feature.

Client : This package contains three classes : `Client`, `ComputationResult` and `RecordingResult`. The class `Client` is designed to offer an easy way to communicate with our server. It exposes multiple functions that map to the various requests supported by the server. It is used by the various plotters to simulate our multiple scenarios. `ComputationResult` encapsulates the results of a computation performed by the server (network time, computation time and parameters) and `RecordingResult` encapsulates the performance recording of the server over a period of time (CPUs usage and cache hit rate).

Computation : this package contains two classes that perform mathematical computations : `SquareMatrix` represents a square matrix that can be raised to a given exponent and `Factorizer` decomposes an integer into prime factors.

Plotters : this package contains Java classes responsible for drawing the multiple plots included in this paper. It automates the various simulations and allows to easily tune their parameters. If needed, any simulation can easily be reproduced. [1].

1.2 Server

For convenience, we chose to use HTTP as the application protocol. This allows to easily debug the application server from the browser but also to reuse existing libraries for the programming task. The request and response format follows the HTTP protocol.

The computation performed by our server simply raises an input square matrix to a given exponent. For the rest of this report, we will denote a computation request by the shorthand form $(N * N)^E$ where N is the size of the input matrix and E is the exponent it should be raised to. For example, $(4 * 4)^{10}$ denotes a request to raise a 4 by 4 matrix to the tenth power.

1. The reader can observe the various parameters used for each plot in the source file

1.3 Load generator

To perform our various analysis, we had to implement a load generator that simulates the behavior of many independent clients by sending requests with exponentially distributed inter-request times. The difficulty of the requests are also random and varying according to an exponential distribution. Our first task was thus to find a way to generate exponentially distributed random numbers. For this, we can use the inversion method that is detailed in figure 6 in appendix. [2] [3]

Our load generator is then used by the various classes from the `Plotters` modules. Its implementation is quite simple : for each simulation, we iterate over a range of request rates and sustain each request rate for a given number of requests. To ensure that we respect the request rate, the main thread sends each request on a new thread and sleeps for the time required between each request. To perform this, a `ThreadPool` is created to which we submit a `Callable` closure for each new request. When all the requests are sent, we collect the `Future` values of the multiple threads that were spawn before moving to the simulation for the next request rate.

2 Measurement 1 : Response time as a function of the difficulty

This first measurement observes how the response time for a single request varies as a function of the difficulty of the request (the difficulty can be the matrix size or the exponent). We start by performing a brief time complexity analysis of our computation algorithm. We then formulate various hypothesis about the expected behavior and we finally compare our hypotheses with our experimental results.

The time complexity of a $(N * N)^E$ operation is $O(N^3 * E)$. Indeed, to multiply a matrix by itself, we must first loop through each row i of its N rows, then through each column j of its N columns, and for each combination of i and j compute their dot-product (sum of N products). This operation must then by done E times since we need to multiply the matrix E times by itself to raise it to the E^{th} power.

Based on this time complexity analysis, we expect our computation time to be roughly a linear function of the exponent and a cubic function of the matrix size. Of course, there will also be an overhead for the network latency. When varying the exponent, this network overhead should be constant since the size of the request payload does not vary by much. However, when varying the matrix size, the network overhead should increase as the matrix size increases. To verify theses assumptions, we performed two simulations.

For the first simulation (results are depicted in figure 1) we vary the exponent for a matrix of fixed size (4 by 4) and report the average response time for a sample of 10 individual requests. As expected, the plot 1 indicates that the computation time seems a linear function of the exponent. We can also observe that the network adds a constant overhead of 72 ms on average. [4]

For the second experiment, we use the same setup but instead vary the matrix size and keep a constant exponent of 2. As seen on plot 11 (in appendix), the computation time seems indeed to be a cubic function of the matrix size. We can also observe that the network overhead increases as the matrix size increases.

3 Measurement 2 : Load generator

For the second measurement, we were asked to extend our client to a load generator that would send requests of varying difficulties to the server. We decided to keep the matrix size constant (4) and make the exponent vary. This choice should limit the influence of the network on the performance analysis since it implies HTTP requests of constant size.

Before diving into the simulation of independent clients, we first wanted to validate a few hypothesis to be sure that our server behaves as expected. From the results of measurement 1, we know that it takes about 92ms on average to compute a $(4 * 4)^{100.000}$ request and that network contribution to the response time is constant at about 72ms.

2. http ://www.columbia.edu/ ks20/4404-Sigman/4404-Notes-ITM.pdf

3. As it will be explained later, we later also needed to generate a normal distribution for another simulation. We then realized that the Apache Commons Math Library provides many convenient classes to easily generate such distributions such as `ExponentialDistribution` and `NormalDistribution`. We thus replaced our custom generated exponential distribution by those provided by this Apache Commons Library.

4. The tests were performed between a server located in Louvain-La-Neuve and a client in Braine l'Alleud. A traceroute indicates that the packets start from Braine l'Alleud and transit via Brussels then Wavre before reaching Louvain-La-Neuve.

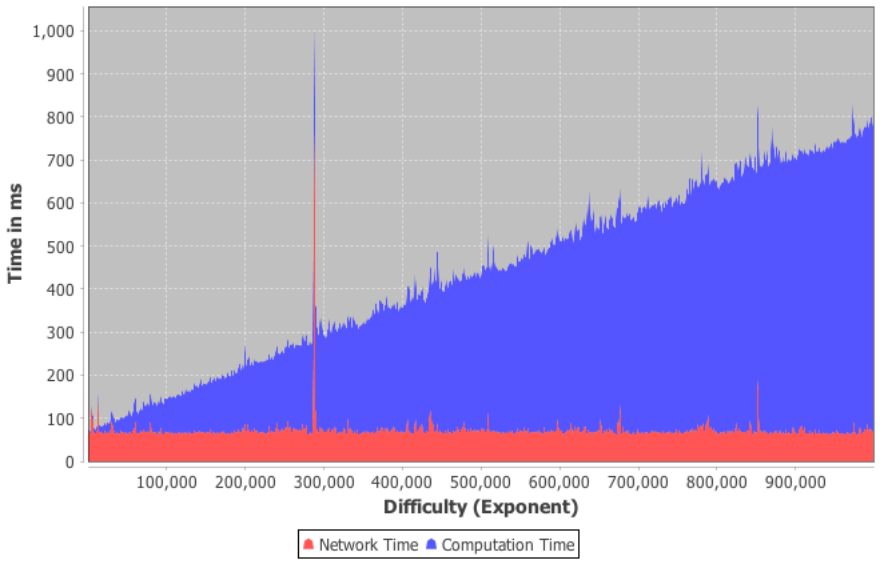

FIGURE 1 – Task 2.1 : We vary the exponent for a constant matrix size of 4. The reported time is an average from a sample of 10 observations. The computation time is a linear function of the exponent.

We would thus expect our server to be able to process a request every 92 ms without any additional delay in the response time (using a single thread) or about 11 requests per second (RPS). To test this hypothesis, we will simulate a linearly increasing request rate using a fixed inter-request time of $\frac{1}{request\_rate}$ seconds. We expect the average response time to be about 167ms when the request rate is lower than 11 RPS. If we increase the load to more than 11 RPS we should see the response time increase as requests start to arrive faster than our server can process. Requests would then have to be queued on the server before being processed. Also, the queue and the response time should continuously grow as we maintain a request rate above 11.

To perform our CPU analysis, we used the Javasysmon library [5]. This library is designed to provide an OS-independent way to manage OS processes and get live system performance information such as CPU and memory usage. It is written in Java and C code that is called via the Java Native Interface (JNI). It works by getting a CPU snapshot which contains the total amount of time the CPU has spent in user mode, kernel mode, and idle. Given two snapshots, it allows us to calculate the CPU usage during that time. The server we are using has an Intel i5 processor, with 2 physical cores and 4 logical threads via Intel Hyper-Threading Technology. Our program reports the CPU load as a percentage of the total capacity across the 4 logical threads. We would thus expect the CPU load to average slightly above 25% once we reach about 11 RPS. Indeed, there should be a unique logical thread running at full capacity with additional CPU capacity used by the OS and the other programs.

Figures 12 and 13 verify our various hypothesis. We can observe that the response time stays constant until 13 RPS and seems to grow indefinitely as the request rate increases. In parallel, we can observe on figure 13 that the CPU load increases as the single thread processes more and more requests. In accordance to our hypothesis, the CPU averages at about 27% once we reach 13 RPS. While the CPU load is increasing, we can see that the response time stays flat since our server is able to absorb the additional load. Once the CPU load starts to flatten, we can clearly observe that the response time start to increase. The single thread is running at full capacity and requests continue to arrive faster that it can process. Using `iftop`, we were also able to observe that the network load stays relatively low and is clearly not a performance bottleneck at the request rate we are operating. With a request rate of 13 RPS, the network load on the server was about 120 kb/sec received and 220 kb/sec sent.

After confirming our various hypothesis, we ran the same experiment again this time using random difficulties and inter-request times, both following an exponential distribution. Figures 14 and 15 illustrate our results and show results that are very similar to plots 12 and 13. We can observe a slightly higher volatility in CPU load and response times due to the random difficulties and inter-request times but all the previous observations still apply.

5. `https://github.com/jezhumble/javasysmon`

4 Modeling 1 : Queuing station model

We were also asked to make a queuing station model of our system. Since our server uses a single thread and both the inter-request times and the difficulties follow an exponential distribution, we can model our system as a M|M|1 queue (we assumed that the queue was of infinite size since it is much larger than the number of requests we send).

The goal of our model is to predict the expected response time E[R] as a function of the request rate λ. We know that E[R]=E[N]/λ where E[N] is the average number of requests in the system. By using Continuous Time Markov Chains (CTMC) where states represent the number of requests in the system, we can obtain the probability π_i to have the system in state i when it is stationary. We can then get E[N] by computing the expected value of this probability function as in Figure 7

From this equation we can get the value of the expected mean response time. Indeed, we fix the value of λ which is the request rate (or the inverse of the mean value of the exponential distribution of the inter-request times). And we can easily know μ (the inverse of the service time) for a given difficulty from measurement 1. From this formula, we can also observe that the model becomes unstable if $\lambda > \mu$.

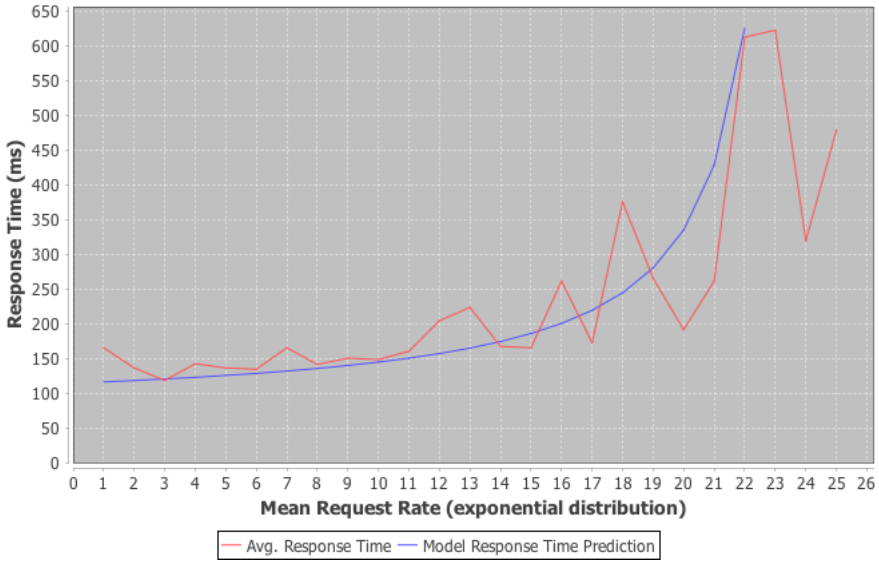

FIGURE 2 – Comparison between M|M|1 model prediction and experimental results. The exponent follows an exponential distribution of mean 50.000 while the mean request rate increases. The service rate is $\mu = \frac{1}{42*10^3} = 23,81$.

Figure 2 compares on the same plot the prediction from the model and the mean response time observed when we vary the request rate for a fixed difficulty of $(4 * 4)^{50000}$. We can observe that our experiment fits the model very well.

5 Measurement 3 : Adding a simple cache

We were also asked to implement an improvement to the performance of the system and show its impacts on the response time. We decided to add a simple cache on the server that would cache entire responses. Since we needed to control the hit rate, we also had to control the distribution of the difficulties (to make sure that the some difficulties happen to be requested several times). We thus decided to switch from an exponential distribution for the difficulty to a normal distribution so that we could fix the standard deviation to produce an arbitrary hit rate of about 20%.

Figure 16 depicts on the same plot how the mean response time evolves with and without the cache. As expected, the server with cache can maintain a stable response rate for a higher request rate. Whereas the regular server starts to struggle at about 20 RPS, the server with cache only starts to struggle at about 25 RPS. On the interval [4, 17], we can also clearly see that the mean response time for the server with cache is slightly lower (some requests are answered almost instantaneously and drive the mean down).

6 Measurement 4 : A multi-threaded server

Finally, we extended our server to process multiple requests in parallel using multiple threads. In this section, we repeat the experiment from Measurement 2 and measure how the response times change with the request rate and the number of server threads. We also build a new model to represent our system with multiple threads.

Since server uses multiple threads and both the inter-request time and the difficulty follow an exponential distribution, we can model our system as a M|M|m queue (again, we assumed that the queue was of infinite size since it is much larger than the number of requests we send). We follow the same reasoning as previously with the single thread model. In order to find E[R], the expected response rate, we need to know the average number of requests in the system E[N]. After solving the CTMC stationary equation of the system, where the states represent the number of requests in the system, we present the derivation of the theoretical performance in Figure 10.

Figure 3 depicts on the same plot our experimental results and the prediction of our model. As before, the model works very well for a single thread. For two threads, the model is still quite good even though it underestimates slightly the actual performance of the system. However, adding a third thread does not seem to improve the performance of our system. After some investigation, we figured out its was probably due to the way the hyper-threading mechanism works. Indeed, logical cores share resources with other logical cores operating on the same physical core. The Intel hyper-threading mechanism works by having one logical core operate while the other logical core is waiting and has nothing to do, for example waiting for a cache or memory access. An additional logical core will thus not provide the same performance increase as a new physical core. In our case, the server is equipped with an i5 processor with only 2 physical cores and 4 logical cores. Figure 17 in appendix plots the CPU load for the same simulation up to the service rate.

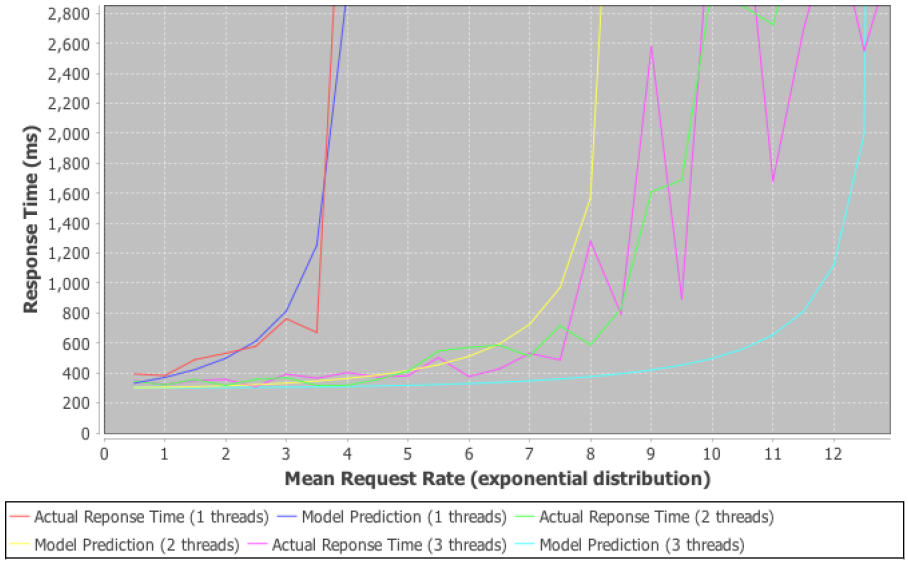

FIGURE 3 – Comparison between M|M|m model prediction and experimental results for up to 3 threads. The exponent follows an exponential distribution of mean 300.000 while the mean request rate increases. The service rate is $\mu = \frac{1}{230*10^3} = 4,3$

Conclusion

For this second assignment, we had to measure the performance of a client-server system and perform a model-based evaluation. Our first model, the M|M|1 queue, was surprisingly good. We were really surprised to see how well the model was fitting the observed result. The second model, the M|M|m queue, also fits pretty well our experimental data up to 2 threads. Since our server only has 2 physical cores and is unable to scale up to 3 real threads, the model with 3 threads does not fit the experimental data.

7 Appendix

7.1 Mathematics

FIGURE 4 – Model M|M|1

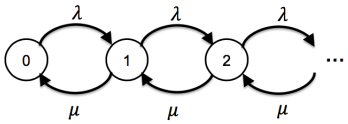

FIGURE 5 – CMTC model M|M|1

If $F_x(x) = 1 - e^{-\lambda x}$

Then $F_x(x) = U$

$1 - e^{-\lambda x} = u$

$e^{-\lambda x} = 1 - u$

$-\lambda x = ln(1 - u)$

$x = \dfrac{-ln(1-u)}{\lambda}$

$x = \dfrac{-ln(u)}{\lambda}$ because u is an uniform distribution

FIGURE 6 – Exponential distribution with inverse method

$P(N = i) = \pi_i = \left(\dfrac{\lambda}{\mu}\right)^i * \pi_0 = \left(\dfrac{\lambda}{\mu}\right)^i * \left(1 - \dfrac{\lambda}{\mu}\right) = \rho^i(1-\rho)$ where ρ represents the utilization

$E[N] = \sum_{i=0}^{\infty} i * P(N = i) = \sum_{i=0}^{\infty} i * \rho^i(1-\rho) = \dfrac{\rho}{1-\rho}$

Therefore,

$E[R] = \dfrac{1}{\lambda}\dfrac{\rho}{1-\rho} = \dfrac{1}{(1-\frac{\lambda}{\mu})\mu} = \dfrac{1}{\mu - \lambda}$

Finally, we have E[R*] the total mean response time,

$E[R^*] = E[R] + Network\_time$

FIGURE 7 – Mathematical reasoning for E [R] in M|M|1 model

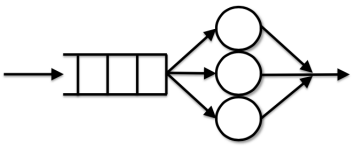

FIGURE 8 – Modele M|M|m

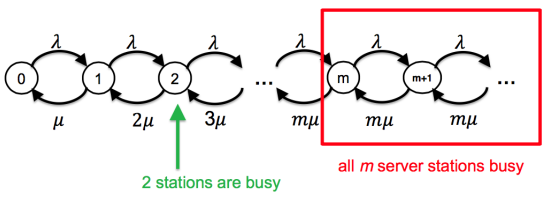

FIGURE 9 – CMTC modele M|M|m

We define utilization of each individual server, $\chi = \frac{\lambda}{m * \mu}$,

$$\pi_0 = \left(\sum_{i=0}^{m-1} \frac{a^i}{i!} + \frac{a^m}{m! * (1-\chi)} \right)^{-1} \quad where \quad a = \frac{\lambda}{\mu}$$

$$\pi_i = \begin{cases} \frac{a^i}{i!} * \pi_0 & \text{for} \quad 0 \leq i \leq m-1 \\ \frac{a^m}{i * m^{i-m}} & \text{for} \quad i \geq m \end{cases}$$

since,

$P(N = i) = \pi_i$

$$E[N] = \sum_{i=0}^{\infty} i * P(N = i)$$

therefore

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

$$= \frac{1}{\mu} + \frac{a^m}{m * \mu * (1-\lambda)^2 * m!} * \left(\sum_{i=0}^{m-1} \frac{a^i}{i!} + \frac{a^m}{m! * (1-\chi)} \right)^{-1}$$

Finally, we have E[R*] the total mean response time,

$E[R^*] = E[R] + Network\_time$

FIGURE 10 – Mathematical reasoning for E [R] in M|M|m model

7.2 Figures

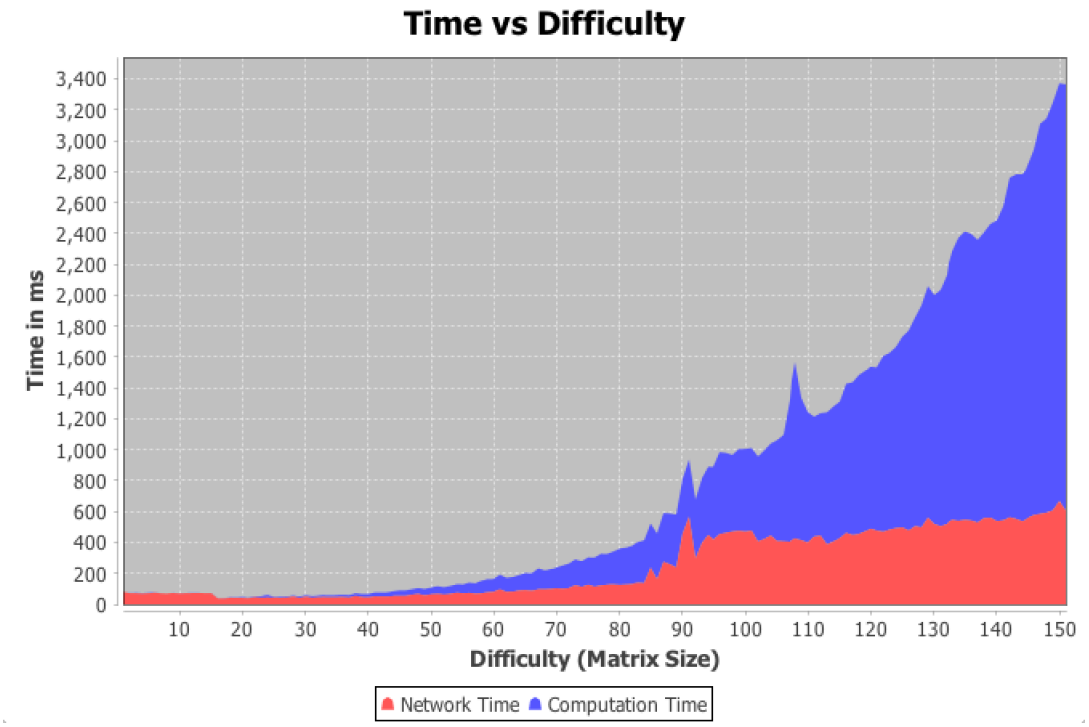

FIGURE 11 – Task 2.1 : We vary the matrix size for a constant matrix size of 2. The reported time is an average from a sample of 10 observations. The computation time is a cubic function of the exponent.

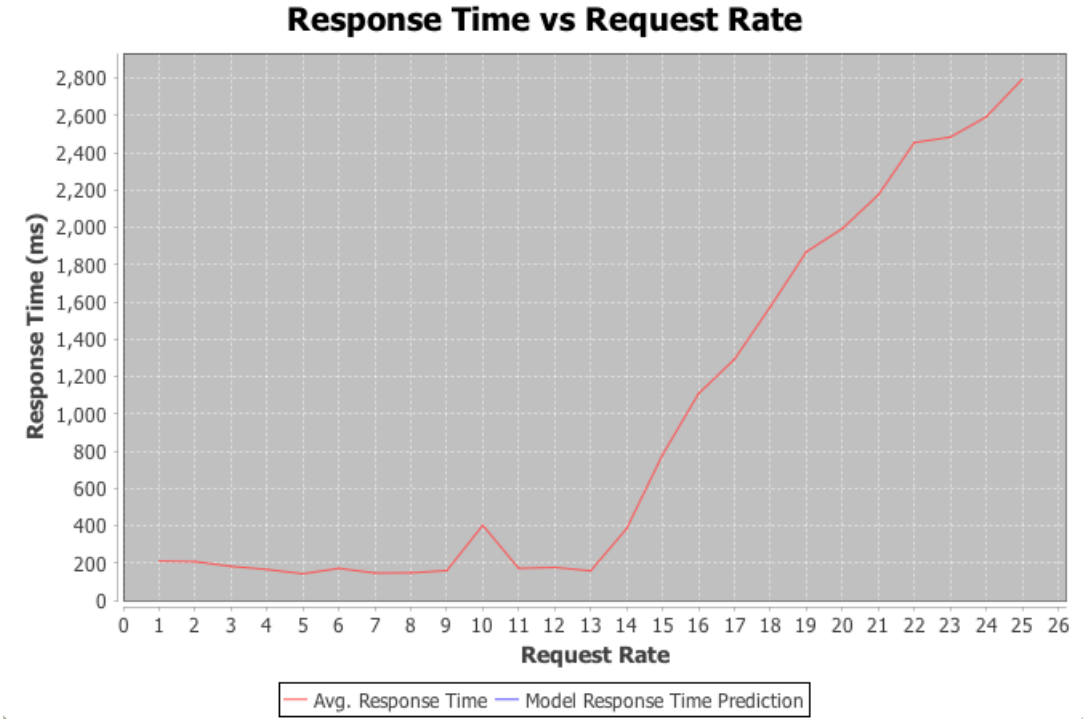

FIGURE 12 – Response Time (fixed non-random difficulty, 1 thread)

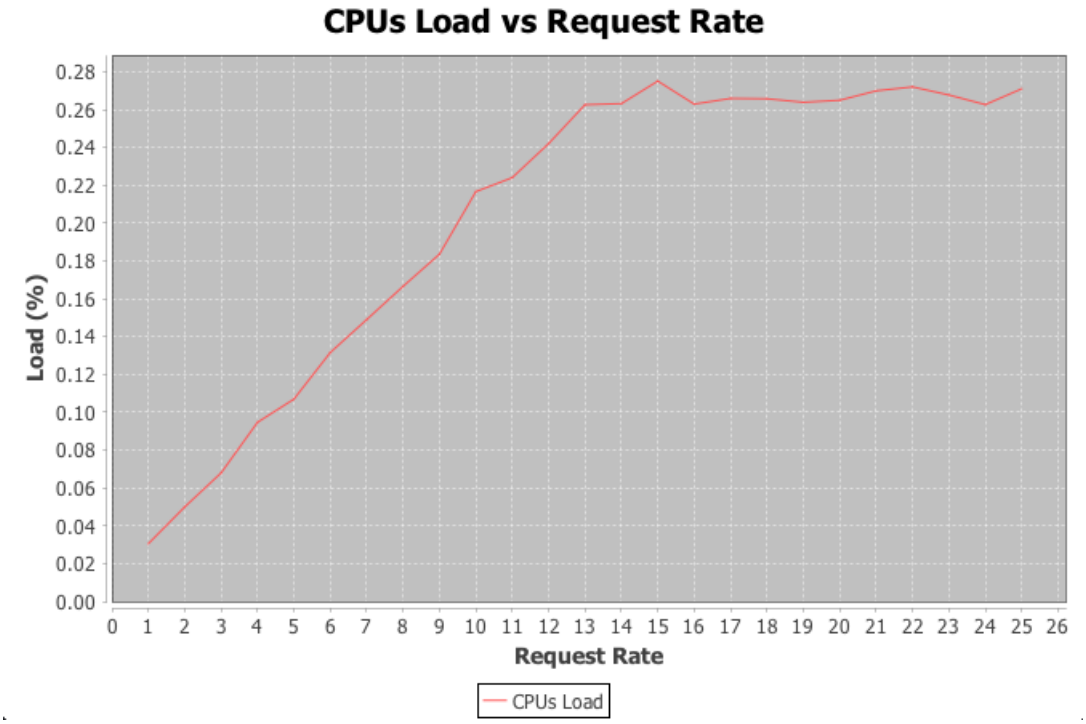

FIGURE 13 – CPU Load (fixed non-random difficulty, 1 thread)

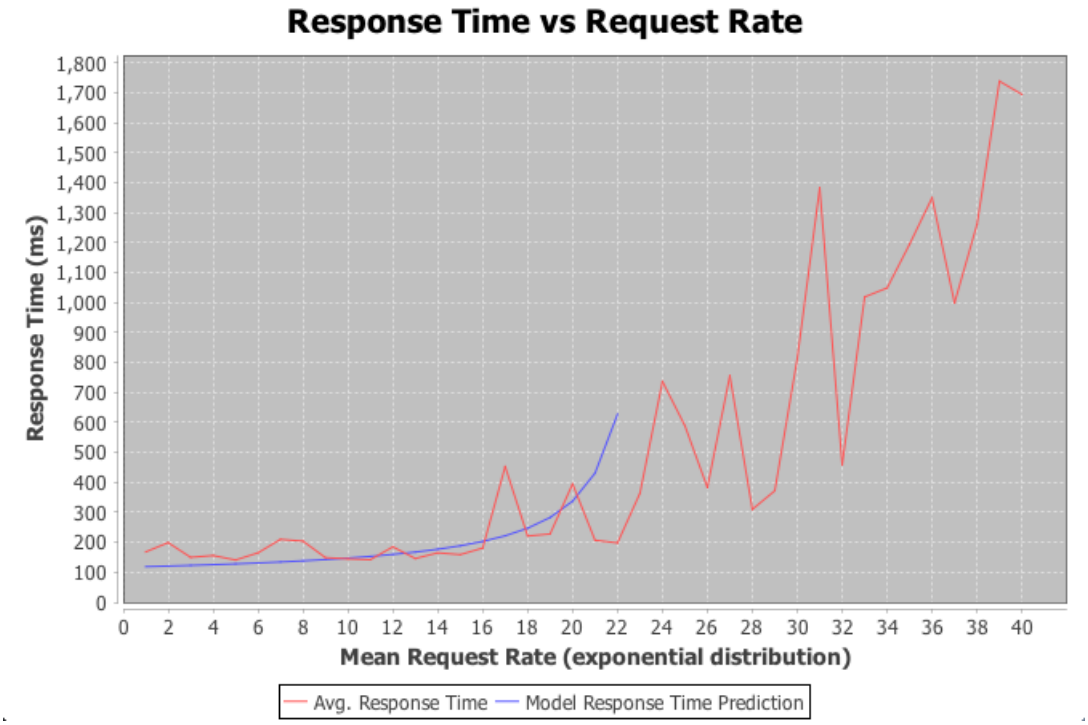

FIGURE 14 – Response Time (exponential distribution of difficulty of mean 300.000, 1 thread)

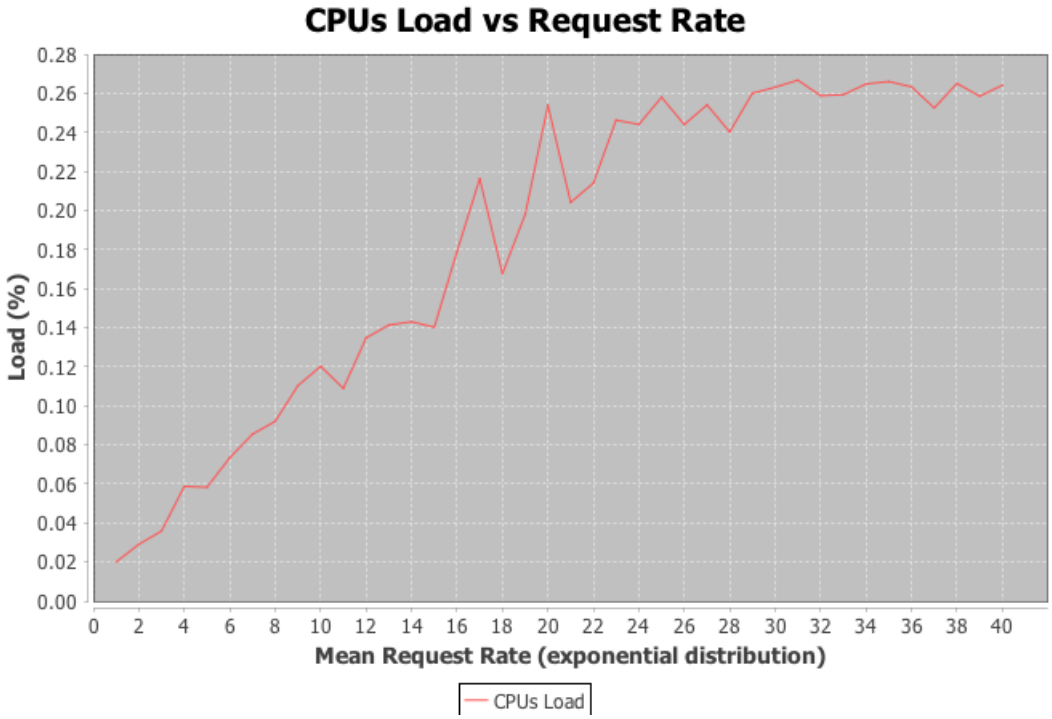

FIGURE 15 – Response Time (exponential distribution of difficulty of mean 300.000, 1 thread)

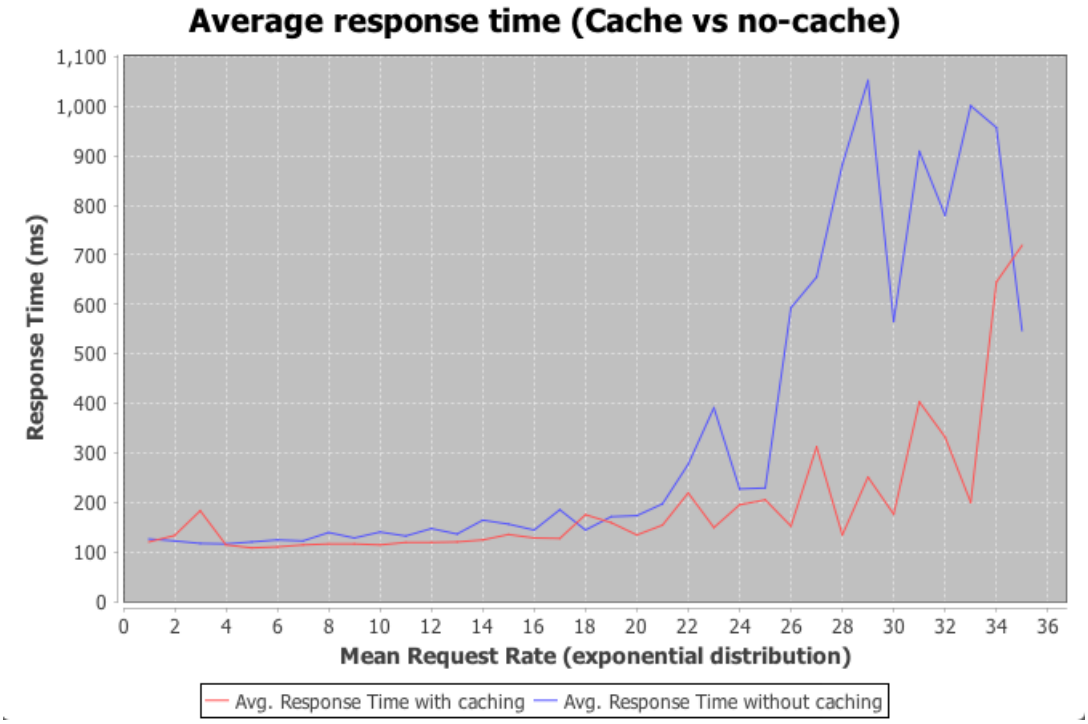

FIGURE 16 – Comparison between server with cache and without cache. The exponent follows a normal distribution of mean 50.000 and standard deviation of 100. The cache hit rate was about 20% on average.

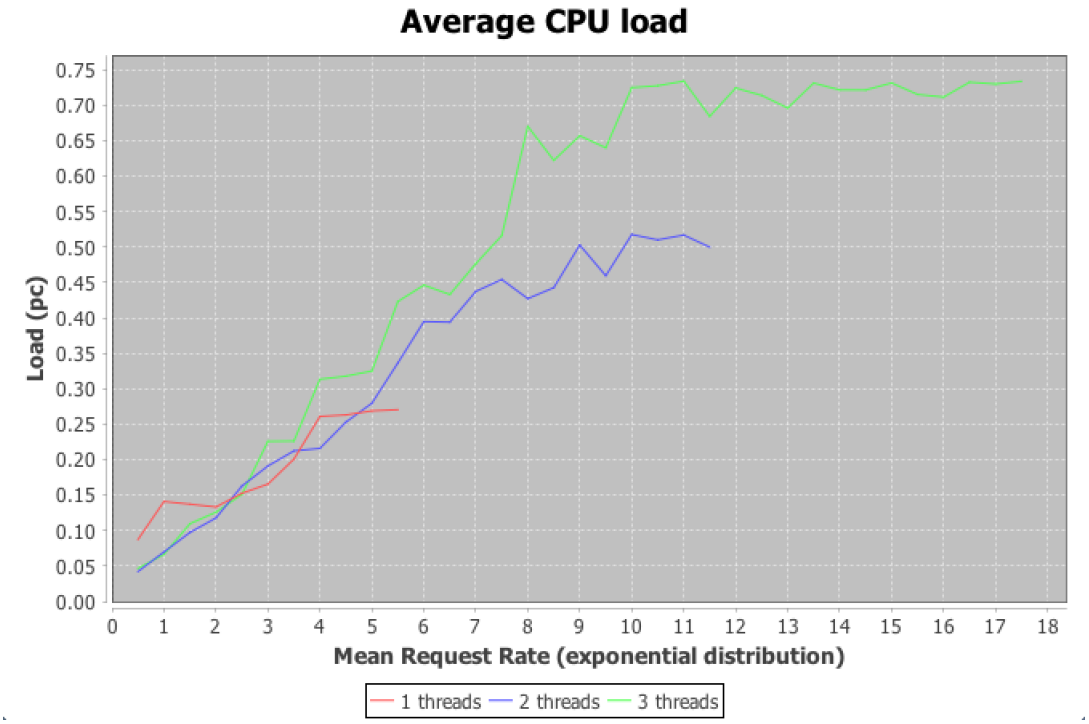

FIGURE 17 – CPU Load (exponential distribution of difficulty of mean 300.000, varying number of threads)