$M|\mathrm{G}|1$ *Queue Project*

Salame William 50691400 Sens Loan 71071700

January 2018

1 Implementation

I'll explain first the server implementation then the client. The server is composed of two thread :

- The first thread is the handshake thread and is responsible for accepting connections attempts from the clients and putting them in a queue for the second thread.
- The second thread computes the required matrix and then sends back the result to the client that previously sent the request before closing the connection.

The client is composed of two threads too :

- The first thread requests a connection to the server every 1 second and puts the request in a queue for the second thread.
- The second thread generates a random matrix and 'complexity' p, sends them to the server, waits a response then finally closes the connection.

Why use two thread for the clients ? Because we wanted to have a constant mean arrival rate, close the connection when we got an answer back from server and compute the response time of the request without halting the thread (To keep a steady arrival rate). If we waited for the request to be answered before actually sending a new one, the arrival rate would grow as the service rate grows and won't follow a Poisson process.

The distribution of the inter-arrival time (resp. the complexity) follow a uniform distribution with mean 1 second (resp. 5,17 and 40) and variance 200ms (resp. 2). And was implemented using random.nextGaussian :

```
randomValue = random.nextGaussian()*variance+mean;
```

The matrix were computed following a very simple $O(n^3)$ algorithm that means the total complexity of the matrix computation was $pO(n^3)$ were p is the 'complexity' (Power coefficient).

The request and response follows a simple text based protocol, the first number is the complexity chosen (the coefficient) of the matrix, the second is the size and then the rest are the matrix numbers row first. That means that 5 3 1.5 3.2 5.2 -2 0 23e1 3 0 6 gives us the matrix

$$\begin{bmatrix} 1,5 & 3,2 & 5,2 \\ -2 & 0 & 230 \\ 3 & 0 & 6 \end{bmatrix}^5$$

2 Measurement Setup

In terms of connection, the measurements were done using two Ethernet wired computers trough an Asus RT-AC68 router.For the computing power, the server was a I7 7700k OC@5Ghz, 8Gb 2400MHz DDR4 Ram. Both clients and server were launched and measured using IntelliJ IDEA Ultimate.

3 Measurement & Modeling

3.1 Measurements

We tested the response times of three different complexities.

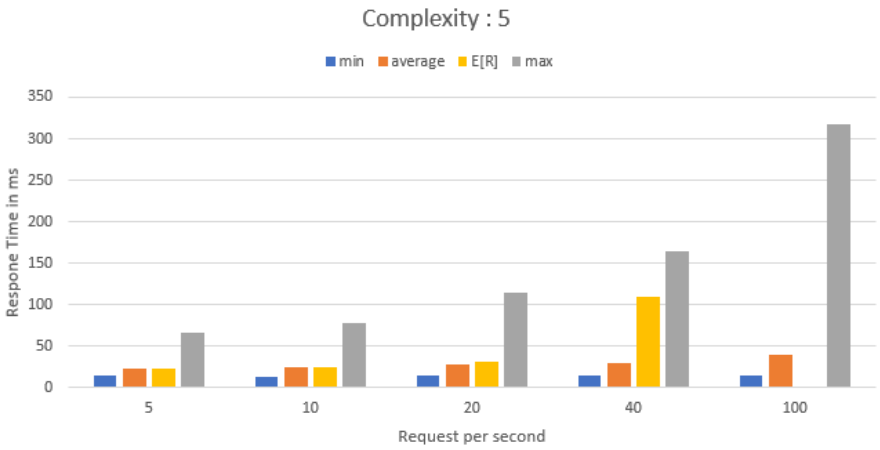

Figure 1: Data for matrix with mean complexity 5

We'll start by analyzing the results by saying that even when taking high coefficient matrix the load and service times measured were more impacted by latency than computation time since in average the computation with complexity 5, 17 and 40 were taking 2, 4 and 6 milliseconds server-side. All tests but the hundred requests per second ones (We'll see why later) were measured by

letting the program run for about a minute and computing the average response time, minimum and maximum trough all clients.

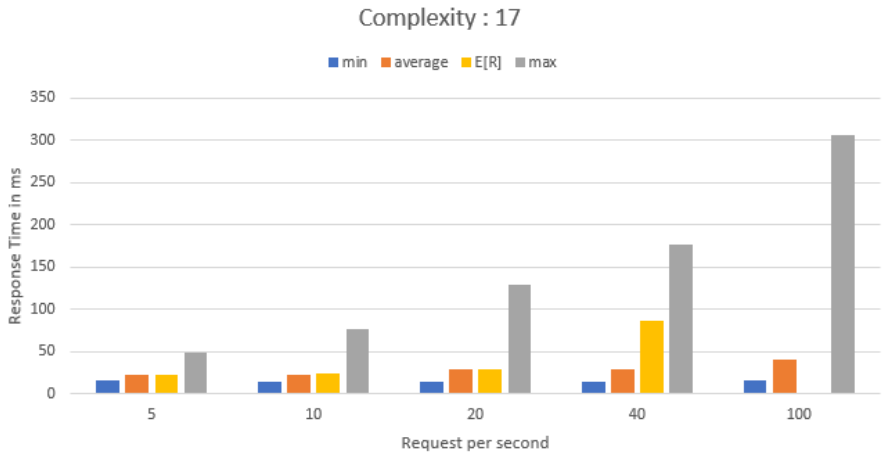

Figure 2: Data for matrix with mean complexity 17

The detailed results can be found in the appendices but we can see here that for every complexity, the results for 5, 10, 20 and 40 requests per seconds were quite close to each other ranging from 22,1 to 30 milliseconds. The hundred requests per second were a special case since running the program for longer time would just result in exponentially longer service times. We can already guess that the queue would be unstable at this arrival rate.

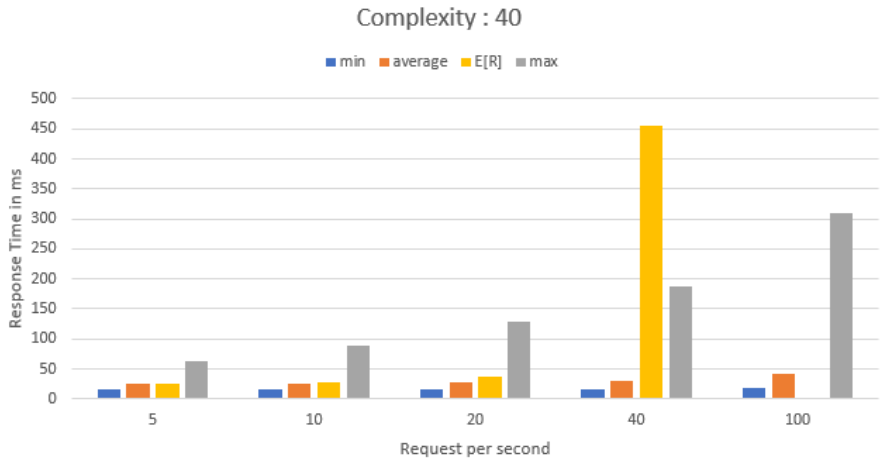

Figure 3: Data for matrix with mean complexity 40

3.2 Choosing our model

After testing the estimated services times, we had the tools to compute the mean response time of the model, but which model should we use ? The choice was actually quite simple, during our tests (apart from the hundred request per second ones) we didn't have any timeouts that meant that we didn't lose any customers and the model M|G|1 will have the same results as the M|G|1|m one since the effective arrival rate was actually the same. If our service time was so long that we'd actually timeout a part of our request then the M|G|1|m model would be better to use since we would 'lose' clients as the buffer was full.

3.3 The M|G|1

So using the M|G|1 model we calculated the parameters and compared them with the original times we found before. The service time measured and computed were quite close for the 5, 10 and 20 requests per second and the model was coherent with the results we found before. But the service times computed for 40r/s were quite higher than the ones we measured. Two things may have caused that :

- Since the network was the major bottleneck in the computation, the data measured were quite volatile and even while testing multiples times a single context, that meant that the second moment of measurements were fluctuating a lot.
- The second thing is that we may not have tested the context long enough, and it didn't reach his stable state and was still growing (Even though we tested some cases for ten minutes).

The second thing we calculated was ρ and the results were more in line with our first measurements.

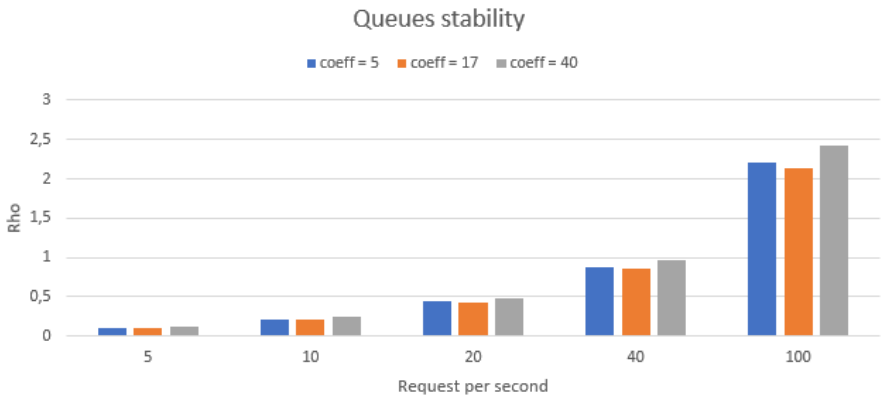

Figure 4: ρ obtained using M|G|1

We can see here that for every case but the hundred per seconds ones the queues were stable since $\rho < 1$. The ρ of the 100r/s were higher than one and that explains why the Response times kept growing as we extended testing times.

Tests lasted 40 seconds and correspond to the average values of all the clients combined								
Inter-arrival time has a mean of 1 second so arrival rate corresponds to the number of clients								
Spec : Server computer : I7 7700k@5Ghz,8Go DDR4 Ram@2400Mhz,Ethernet connection between the two computers								
Matrices were size 40 and content was randomized real between 0 and 250								
coeff = 5	5	10	20	40	100			
min	14	13	15	14	14	**E[S]**	**E[S²]**	
average	23,3	24,4	27,2	30,3	38,9			
max	66,42	77,84	114,75	164,27	318,28	0,022	0,000526	s
Rho	0,11	0,22	0,44	0,88	2,2			
E[R]	0,023477	0,02537	0,031387	0,109614	UNSTABLE			
coeff = 17	5	10	20	40	100			
min	16	15	15	15	16	**E[S]**	**E[S²]**	
average	22,1	22,8	29,1	28,9	40,3			
max	49,31	76,06	129,33	176,39	306,34	0,02137	0,00047	s
Rho	0,10685	0,2137	0,4274	0,8548	2,137			
E[R]	0,022687	0,024361	0,029585	0,086163	UNSTABLE			
coeff = 40	5	10	20	40	100			
min	17	17	17	16	18	**E[S]**	**E[S²]**	
average	25,0	26,3	27,8	30	40,9			
max	64	89	128,79	187	309,4	0,02424	0,000654	s
Rho	0,1212	0,2424	0,4848	0,9696	2,424			
E[R]	0,0261	0,028555	0,036932	0,454424	UNSTABLE			
E[R] = E[S] + Lambda*E[S²]/2*(1-Rho)								

Figure 5: Data measurements