Advanced Systems Lab

Milestone 2 Report

Karolos Antoniadis

This is the report for the second milestone of the "Advanced System Lab" project. We start this report by giving a brief introduction about this milestone's objectives. Next, in Setion 2 we delve into the interactive response time law and see how it applies on our system. In Section 3 we show our 2^k experimental design. We continue in Section 4 by describing where queueing occurs in our system and in Section 5 we state our assumptions about our queues in our models. Afterwards, in sections 6 and 7 we gradually describe our analytical queueing models for our system. We conclude our report in Section 8.

1 Introduction

Goal of this milestone was to develop an analytical queueing model for our system. After developing the model it was needed to derive the performance characteristics of our system and predict its performance which was going to be compared with our results from the first milestone. All the formulas used for this report were taken from [1].

2 Interactive Response Time Law

As was already stated in the report of the first milestone we have a **closed system**. Therefore the response time law should apply to our results. We have verified that all of our experimental results were successfully satisfying the response time law and specifically showed that the law holds in our trace in the previous report. In this section we are going to have a closer look of this law and how it applies in our "Increasing the Message Size" experiment from the first milestone. Before going on let us remind the reader of the interactive law. The interactive response time law states the following:

$$X = \frac{N}{R + Z}$$

where X is the throughput, N is the number of clients issuing requests, R is the response time of a request, and Z is the think time. Let us also remind the reader that for the "Increasing the Message Size" experiment the configuration was the following:

- One m3.large client instance with 50 clients
- One m3.large middleware with 20 middleware threads and 20 connections
- One m3.large database

In Figure 1 we can see the real calculated throughput of our system compared to the throughput calculated using the response time law. The throughput was calculated using the response time of all the types of requests, while the think time is zero. As can be seen in the figure the corresponding throughput lines match each other which clearly states that our system satisfies the aforementioned law.

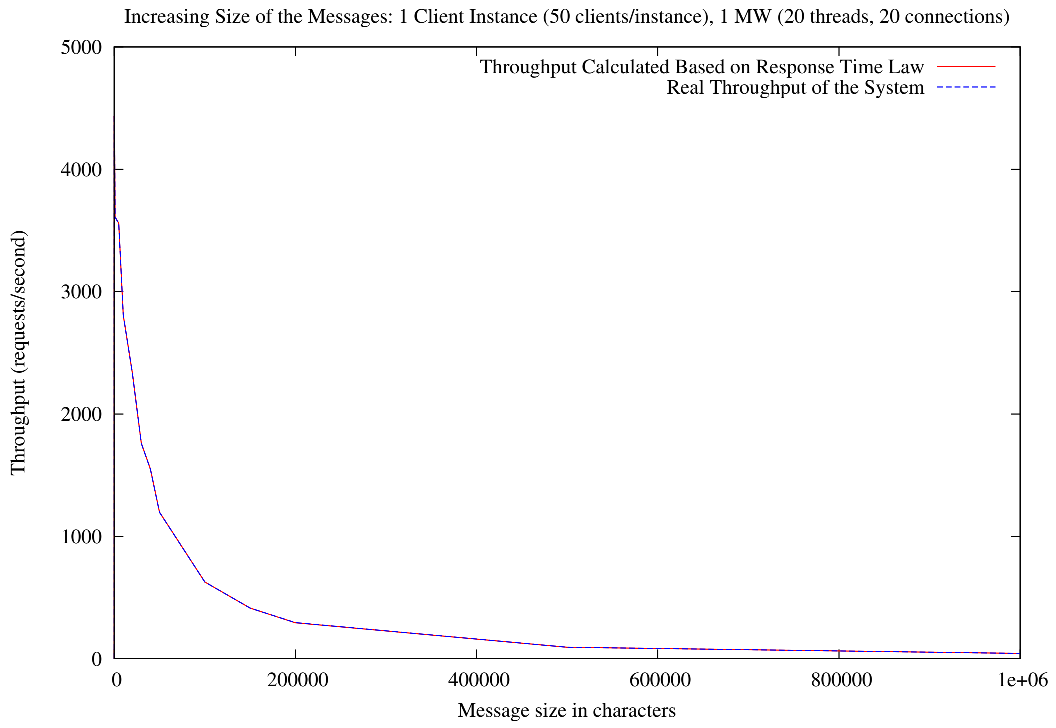

Fig. 1: System's Throughput Compared to Throughput Calculated with the Response Time Law

We would like to note that in all the checks we did with the response time law we used the response time and throughput for all the types of requests but we never checked for a specific type of request individually. Let us see what happens for a specific type of a request, for example let us assume we are looking at the experiment where we used messages of $2 \cdot 10^5$ characters. In this case from the experiment we get the results depicted in Figure 2. Note that the calculated throughput values are all pretty much the same. The explanation of this was given in the previous report and has to do with the way clients work: they issue almost the same amount of requests per given type. Based on those results let us try to apply the response time law to see if it applies for every specific type of request. For example for the "LIST_QUEUES" request we have the following throughput

$$X = \frac{N}{R+Z} = \frac{50}{75.2135+0} = 0.664req/ms = 664req/s$$

The throughput of $664req/s$ is quite different from $98req/s$ which is the "LIST_QUEUES" throughput. So what is going on? The problem in the above throughput calculation is that we assumed that the think time is 0. This is not the case for a specific request as was for all the types of requests. For example, in case of a "LIST_QUEUES" request there is actually some think time since by the way the client works is that after a "LIST_QUEUES" request a "RECEIVE_MESSAGE" request could take place and then a "SEND_MESSAGE" request as can be seen in Figure 3. Note here that the "RECEIVE_MESSAGE" request is issued only when there is a message to be received. Nevertheless "RECEIVE_MESSAGE" is called most of the times. So the think time is actually

$$Z = 304.199 + 130.552 = 434.751ms$$

By using this think time we can calculate the throughput again which now is:

$$X = \frac{50}{75.2135 + 434.751} = 0.098req/ms = 98req/s$$

which is also the throughput of our system. Similarly we can see that the throughput of "SEND_MESSAGE" is

$$X = \frac{50}{304.199 + (75.2135 + 130.552)} = 0.098 req/ms = 98 req/s$$

and of "RECEIVE_MESSAGE" is

$$X = \frac{50}{130.552 + (75.2135 + 304.199)} = 0.098 req/ms = 98 req/s$$

Although we have verified that the law holds in all of our previous experiments by having a closer look we saw that the think time is not zero when considering a specific type of request instead of all the types of requests together. We consider this to be an interesting observation of our system.

type of request	throughput (requests/sec)	response time (ms)
LIST_QUEUES	98.1166666667	75.2135
SEND_MESSAGE	98.130952381	304.199
RECEIVE_MESSAGE	98.0880952381	130.552

Fig. 2: Throughput and Response Time Calculated for Every Type of Request when Message Size is $2 \cdot 10^5$ Characters

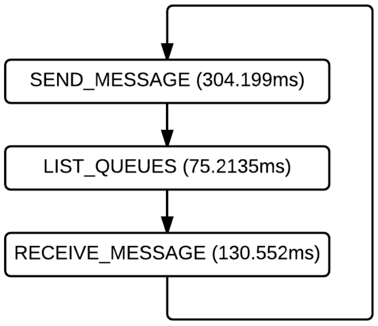

Fig. 3: Flow of a Client Issuing Requests with their Corresponding Response Times

3 The 2^k Experiment

Since we did not have a 2^k experimental design in our first milestone we created one for this milestone. The factors we used for this design were the number of connections (A) to the database, the number of middleware threads (B) in the thread pool and the Amazon EC2 instance type (C) of our database. We chose those factors in order to check their impact on the performance of our system. The levels for every factor can be seen in Figure 4.

The following configuration was used for the experiments:

- One m3.large client instance with 100 clients
- One m3.large middleware with A connections and B middleware threads
- One database of C instance type

Hypothesis: We expect the major increase on our system's performance to happen when changing the instance type of the database from m3.large to m3.xlarge. This is because we showed in the first milestone that the database is our bottleneck. Similarly based on our previous "Increasing the Number of Threads" and "Increasing the Number of Connections" experiments we do not except a great impact when changing those factors. In spite of that we used those three factors because we considered it interesting to see how they all interact with each other.

We ran each of the 2^3 experiments for 10 minutes and removed two minutes from the beginning of our logs corresponding to the warm-up phase, as well as one minute from the end corresponding to the cool-down phase. In Figure 4 the throughput for every experiment is depicted.

A (#connections)	B (#threads)	C (db instance type)	throughput (requests/sec)
20	20	m3.large	4810
20	40	m3.large	4695
40	20	m3.large	4791
40	40	m3.large	4609
20	20	m3.xlarge	7511
20	40	m3.xlarge	6687
40	20	m3.xlarge	7443
40	40	m3.xlarge	7684

Fig. 4: Results of 2^3 Experiment

In Figure 5 we can see the **sign table** for calculating the effects of our design. Our results can be interpreted as follows. The mean performance is 6028 requests/sec, the effect of changing the instance type was the greater one with 1302 requests/sec. Increasing the number of connections has an effect of 103 requests/sec. Positive throughput also have the combinations of AB, AC and ABC. Checking Figure 5 we can also see that there are two cases with negative effect. The first one is when increasing the number of threads, while the second one when increasing the number of threads as well as changing database instance type. This implies that it is probably worthless to just increase the number of threads. And as we can see in AB it does not make sense to increase the number of threads without also increasing the number of connections.

I	A	B	C	AB	AC	BC	ABC	y
1	-1	-1	-1	1	1	1	-1	4810
1	1	-1	-1	-1	-1	1	1	4791
1	-1	1	-1	-1	1	-1	1	4695
1	1	1	-1	1	-1	-1	-1	4609
1	-1	-1	1	1	-1	-1	1	7511
1	1	-1	1	-1	1	-1	-1	7443
1	-1	1	1	-1	-1	1	-1	6687
1	1	1	1	1	1	1	1	7684
48230	824	-880	10420	998	1034	-286	1132	Total
6028	103	-110	1302	124	129	-35	141	Total/8

Fig. 5: Sign Table for Calculating Effects

This 2^3 experimental design verified our previous knowledge about the factors from our already done experiments. But it also showed us that some combinations of factors and levels are useless, something we did not know from our previous results.

4 Queueing in the System

Before starting with modelling our system as a network of queues we discuss on the queueing that occurs in our system. Queueing can occur in many places. The ones that we thought of are the following.

- Middleware Threads: There is a fixed number of middleware threads working on receiving the requests from the clients. If more clients try to communicate with the middleware than the number of threads then queueing occurs. A client is served when a middleware thread picks the client's socket as was explained in the previous report. This queue is a First In, First Out (FIFO) queue, also known as First Come, First Served (FCFS). There is no queueing in case the number of clients is less than the number of threads. But implicit queuing could exist due to the number of threads our CPU can handle as will be explained below.

- Connections to the Database: We also have a fixed amount of database connections. Middleware threads are the ones that use connections so if the number of threads is the same as the number of connections then there is no queueing for getting connections. This was the case in most of our previous experiments, except the "Increasing the number of Threads" (with more than 20 threads) and "Increasing the Number of Connections" (with less than 20 connections) experiment. Therefore in most of our experiments there was no queueing in this part of the system. Note although that there is queueing happening in the database as will be explained below.

- Database: Since all of the requests work on the same table there is going to be queueing in the database, since the DBMS cannot operate fully parallel. Furthermore by the way we implemented the *receive_ message* stored function using the "FOR UPDATE" clause (implicit locking), there is definitely queueing at this part of the database.

- Network: The network is used when a request is sent from the client to the middleware and vice versa, and when the middleware is communicating with the database. Queueing is possible to happen while TCP packets are being transmitted to their destination through intermediary routers. From our previous experiments although we did not calculate network times explicitly we did not find any specific delays on the network. Using "iperf"[1] as well showed really huge bandwidth (1Gbps+) between our EC2 instances. By using Wireshark and our usual message sizes, messages of 20 characters[2], we found that the transmitted message with the greatest size had a size of only 709 bytes. If we compare this message size with our bandwidth speed we can see that network is not actually delaying our system, probably by just a small constant time. Because of this and since there is no queueing actually happening in the network we are not going to consider the network in our models.

- CPU: We could argue that using a high number of middleware threads or concurrent database connections in our system would increase performance. This in some degree is true but there is a limit. Since using a high number of threads in our system not all of them can be executed at the same time due to hardware constraints like number of processors etc. In some sense this can be considered as queueing since a processor has to context-switch between executed operations. We are going to consider this when designing our database's service rate $\mu(n)$ function later on.

- Serialization: Finally we would like to note that *synchronized* blocks in our code could queue operations. In our system we only have one *synchronized* block that exists in the *ConnectionPool* class. This block is only used when not all of the given connections have been created, so only at the beginning (warm up) of our system and therefore is not being taken into consideration.

[1] https://iperf.fr/

[2] We used message sizes of 20 characters in all of our previous experiments except the "Increasing the Message Size" experiment.

5 Queues Assumptions

In the following models we simplified our analysis by considering infinite buffer capacity and infinite population size, as well as an FCFS service discipline. We are also assuming exponentially distributed arrivals as well as service times. Finally we are going to use the Kendall notation for our queues. Therefore our queues are going to be of the form "M/M/m/$+\infty$/$+\infty$/FCFS" which can be abbreviated to "M/M/m" where m is the number of servers.

About the infinite population size assumption we would like to note the following. **None** of our experiments has infinite population size. In every experimental data point we used a specific number of clients (population), even in our "Increasing Number of Clients" experiment if we consider every data point per se we had a fixed number of clients. This implies that our system is always stable since we know from [1] that "In the finite population systems, the queue length is always finite; the system can never become unstable.". Nevertheless we assume infinite population size to simplify the analysis of our models. Finally, note that in some models based on the stability condition that states $\lambda < m\mu$ where λ is the arrival rate, m is the number of servers and μ is the mean service rate per sever, our system could happen to be considered unstable. This does not really mean that our system is unstable, but merely that the specific model is incapable of successfully describing our system. This is going to be explained later on as well.

6 Modelling with One Queue

The simplest model that we can have is a model with just one queue. We consider such a model in this section. First we use an M/M/1 queue and then use an M/M/m queue. So in our first queue we have just one server while in the second one we have m servers.

One Server

For our first model we are going to consider the one depicted in Figure 6. As we can see in the figure we have one queue and the clients which issue the requests (jobs) that are our terminals. For this queue we are going to consider the service of the queue as the the whole operation of what the middleware does as a whole, i.e. so the one that contains both the middleware and the database operations.

In order to check how well our model behaves we are going to use our trace results given in the previous report. We remind the reader that the configuration of our trace was the following:

- Two t2.small client instances with 50 clients each
- One t2.small middleware instance with 20 middleware threads and 20 connections
- One t2.medium database

As service time we are going to use the calculated time of $3.33ms$, which corresponds to the whole time taken inside a middleware thread, so $\mu = \frac{1}{3.33}$. Since our system is a closed system, the number of job arrivals is equal to the number of job completions, the **job flow balance** assumption holds. (The job flow balance holds in all of our experimental results and is going to be used in the following models as well.) The assumption holds since the number of arrivals is equal to the number of completions. This means we can use the arrival rate (λ) and throughput interchangeably, therefore $\lambda = 5.897 req/ms$. By using this as input we can calculate the traffic intensity of our system's queue which is

$$\rho = \frac{\lambda}{\mu} = 5.897 \cdot 3.33 = 19.63$$

But $\rho = 19.63 > 1$ meaning our system is not stable! As we already explained in the previous section, this merely implies that this model is not very good for describing our system. Specifically that traffic intensity is greater than one can be explained for the simple reason that we used 20 middleware threads and 20 connections so the general assumption of having just one server is not valid. To prove this is the case we created a new experiment where we just ran 50 clients using 1 thread and 1 connection. Specifically the configuration of our experiment was the following:

- One m3.large client instance with 50 clients
- One m3.large middleware with 1 middleware thread and 1 connection
- One m3.large database

Running this experiment for 10 minutes and removing 2 minutes as warm up and 1 minute as cool down we got arrival rate of $\lambda = 1.096509 req/ms$ and mean service rate of our one server $\mu = \frac{1}{0.906036}$. So in this case the traffic intensity is

$$\rho = \frac{\lambda}{\mu} = 1.096509 \cdot 0.906036 \simeq 0.9934 < 1$$

In this case we found out traffic intensity less than one, which we believe shows that the number of threads or connections should be taken into account when thinking about our model.

Finally for this latter experiment we found that the mean response time for a request was $45.5888ms$ with a standard deviation of 6.28519. Using the traffic intensity and the mean service rate we can compute the expected response time using the following formula:

$$E[r] = \frac{\frac{1}{\mu}}{1-\rho} = \frac{0.906036}{1-0.9934} \simeq 137.27ms$$

But the response time of $137.27ms$ is far away from the real response time of $45.5888ms$. We conclude that using just an M/M/1 queue to model our whole system is not enough. We need to refine our model.

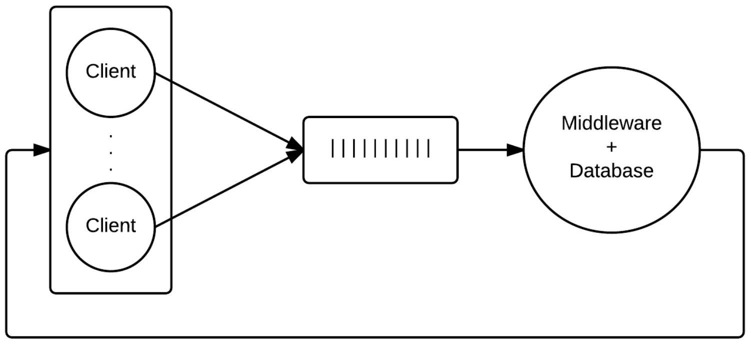

Fig. 6: Model with an M/M/1 queue

m Servers

For refining the model we are going to consider again a model with one queue but this time with m servers. As a server we are going to consider a middleware thread. So each server picks up a request from the client, calls the underlying database operation and returns the response to the client. This model is depicted in Figure 7.

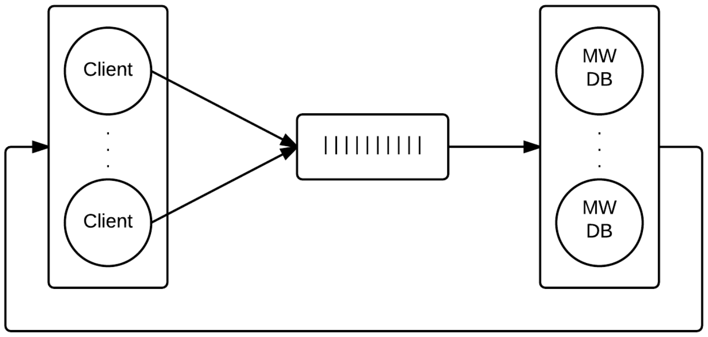

Fig. 7: Model with an M/M/m queue

Let us revisit our trace experimental results. We remind that the arrival rate is $\lambda = 5.897 req/ms$ and the mean service rate per server $\mu = \frac{1}{3.33} = 0.3003$. The traffic intensity is now defined as $\rho = \frac{\lambda}{m \cdot \mu}$ where m is the number of servers. m in our case is 20 since for our trace experiment we used 20 middleware threads. Using our trace results we can find the traffic intensity of

$$\rho = \frac{\lambda}{m \cdot \mu} = \frac{5.897 \cdot 3.33}{20} = \frac{19.63}{20} = 0.981 < 1$$

As we can see using m servers seems to give us a better model compared to our previous one. The mean response time can be calculated as:

$$E[r] = \frac{1}{\mu}(1 + \frac{\varrho}{m(1-\rho)})$$

In order to calculate this we have to calculate the probability of queueing ϱ which is

$$\varrho = P(\geq mjobs) = \frac{(m\rho)^m}{m!(1-\rho)} P_0$$

where P_0 corresponds to the probability of having zero jobs in our system and can be computed using the $m = 20$ and $\rho = 0.981$ as follows:

$$P_0 = (1 + \frac{(m\rho)^m}{m!(1-\rho)} + \sum_{n=1}^{m-1} \frac{(m\rho)^n}{n!})^{-1} = 9.34^{-10}$$

Using the computed P_0 we get $\varrho = 0.86$. Therefore the expected response time can be calculated and is:

$$E[r] = \frac{1}{\mu}(1 + \frac{\varrho}{m(1-\rho)}) = 8.57ms$$

This is actually less than our mean response time of $16.668ms$[3] and considering the standard deviation of our response time which about 5, the response time of $8.57ms$ is still far off. Next we consider the mean waiting time $E[w]$ which can be calculated as

$$E[w] = \frac{\varrho}{m\mu(1-\rho)} = 7.88ms$$

The real waiting time is around $13ms$, 13.3207 to be exact with a 4.17 standard deviation. Using "Little's Law" we can calculate the mean number of jobs in our system which should be $E[n] =$

[3] Again this value was taken from the first report from the "Stability" (trace) part.

$\lambda \cdot E[w] = 5.897 \cdot 7.88 = 46.46$. This number is also far off compared to the number of jobs we are expecting to have which should be around 100 since we have 100 clients. Using "Little's Law" with our real waiting time we get $E[n] = \lambda \cdot E[w] = 5.897 \cdot 13.3207 = 78.55$ which is still far off but much closer to the actual number of jobs in our system.

As we can see using an M/M/m queue gives us better results in comparison to a M/M/1 queue but still our model cannot predict the real performance of our system. To show this more blatantly we are going to use this model to see how it responds to our "Increasing Number of Clients" experiment. In what follows we computed the expected mean response time based on the arrival rate and mean service rate.

n (number of clients)	μ(1request /ms)	λ (requests/ms)	$\rho = \lambda/(m\mu)$	$E[r]$ (ms)
2	1 / 1.18	1.41919	0.08373221	1.18
5	1 / 1.76	2.47349	0.217667	1.76
10	1 / 2.95	3.09045	0.45584	2.95
15	1 / 3.91	3.59149	0.70213	3.97
20	1 / 5.30	3.360391	0.8905	6.54
25	1 / 4.98	4.0087	0.9987	139.45
30	1 / 4.79	4.198323	$1.0063 > 1$	-

As we can see after 30 clients traffic intensity is greater than one in this scenario and therefore we cannot calculate the response time. We can see in Figure 8 that in the first 20 clients the model can predict quite well the expected response time but after 20 clients it increases dramatically due to the increase in the traffic intensity. We believed this has to do with not taking into consideration the database explicitly in our model. And since we already have shown that the database is the bottleneck, by introducing more clients to the system there is contention starting to happening in the database which is not appropriately perceived by the model. To solve this we created a queueing network that is shown in the next section.

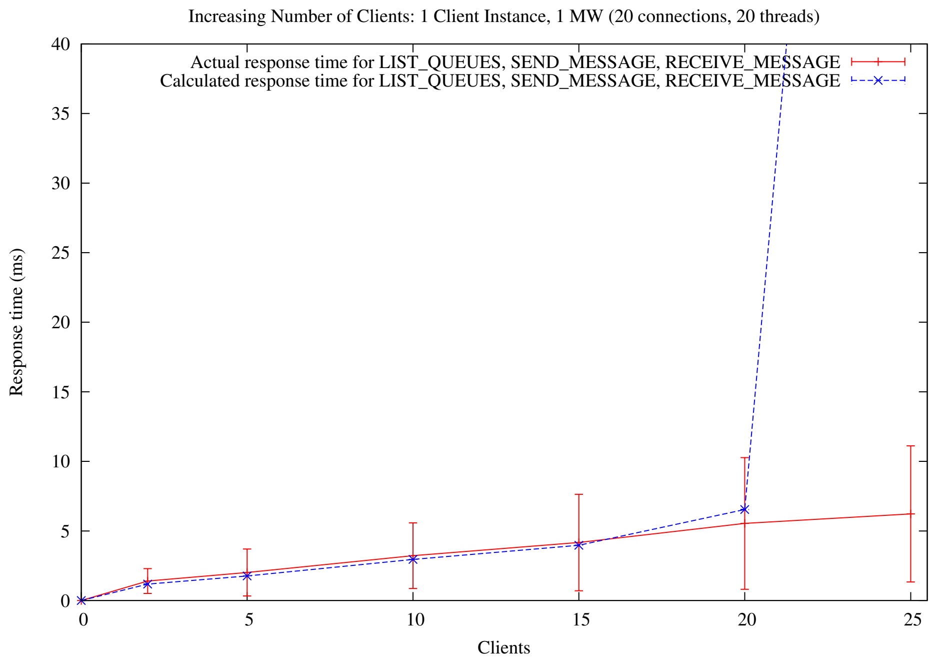

Fig. 8: Increasing Number of Clients Predicted Response Time in Contrast with Real Response Time

7 Queueing Network

In this section we consider a queueing network, which is a refinement of our previous model. Since we have a closed system and requests (jobs) in our system move from one queue to the next one without having externals arrivals or departures, our queuing network is a **closed queueing network**. This section's model is depicted in Figure 9, as can be seen there are two queues one corresponding to the middleware threads while the other corresponds to the database. The database "servers" correspond to the database connections in the sense on how many operations our database can actually execute at the same time.

We consider the middleware and the database as load-dependent service centers since their service rate depends on the number of jobs in the device. We wanted to compare this model with our "Increasing Number of Clients" experimental results. Since for this experiment we had 20 middleware threads and 20 database connections we considered both as M/M/20 queues.

In order to predict the response time based on our model we used the Mean Value Analysis (MVA) algorithm that includes load-dependent centers. The code of the MVA algorithms was implemented in Java using *BigDecimal* objects and can be found in the *MVA* class (`code/MVA.java` file). In order to provide as input the service rate of our middleware threads and our database we ran one more experiment with the following configuration:

- One m3.large client instance with 2 clients
- One m3.large middleware with 2 middleware threads and 2 connections
- One m3.large database

We chose two clients since we cannot have a running system with just one client by they way we have designed our clients. And we chose to have the same number of middleware threads and connections so there is no queueing in our system. Therefore the calculated values would correspond to the real service times. We concluded that the service time of the middleware device is $0.000534249ms$ while for the database device it is $1.2686ms$. MVA also takes the input think time, which in our case, as usually, is 0. Number of visits also needed for MVA was given and was one per device. The service rate was given using the following service rate $\mu(n)$ function where m is the number of servers:

$$\mu(n) = \begin{cases} \frac{n}{S}, & n = 1, 2, \ldots, m-1 \\ \frac{m}{S}, & n = m, m+1, \ldots, \infty \end{cases}$$

For the service rate function of the database, and for the reasons explained in Section 4, specifically the "Database" queueing part we consider the following changed function. This was done to take into account the queueing and inherent sequential execution that is happening in our database.

$$\mu(n) = \begin{cases} 0.5 \cdot \frac{n}{S}, & n \leq 5 \\ 0.3 \cdot \frac{n}{S}, & n \leq 20 \\ 0.3 \cdot \frac{20}{S}, & n = 21, 22, \ldots, +\infty \end{cases}$$

Our database is an enormous and complex system so it would be really hard and out of the scope of this report to actually try and model the internals of the database using queueing theory.

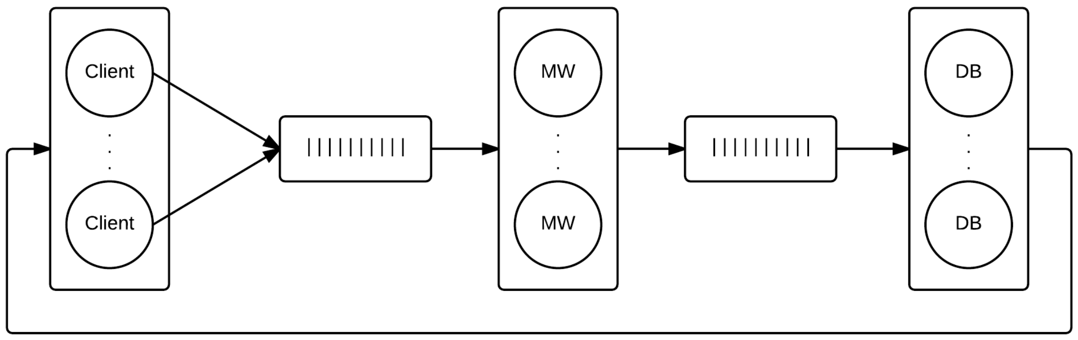

Fig. 9: Queueing Network

Using MVA we calculated the response time and the respective throughput. The comparison of the calculated results and the real ones can be seen at Figures 10 and 11. As we can see in Figure 10 response time was calculated pretty well since it is always inside the error bar of our actual response time. Throughput on the other hand in Figure 11 stabilizes much quicker than it does in our real system and stays stable afterwards.

Although our analysis results do not fit perfectly our actual experimental results they show the trends of our system. So this model can be used to actually predict the general performance of our system.

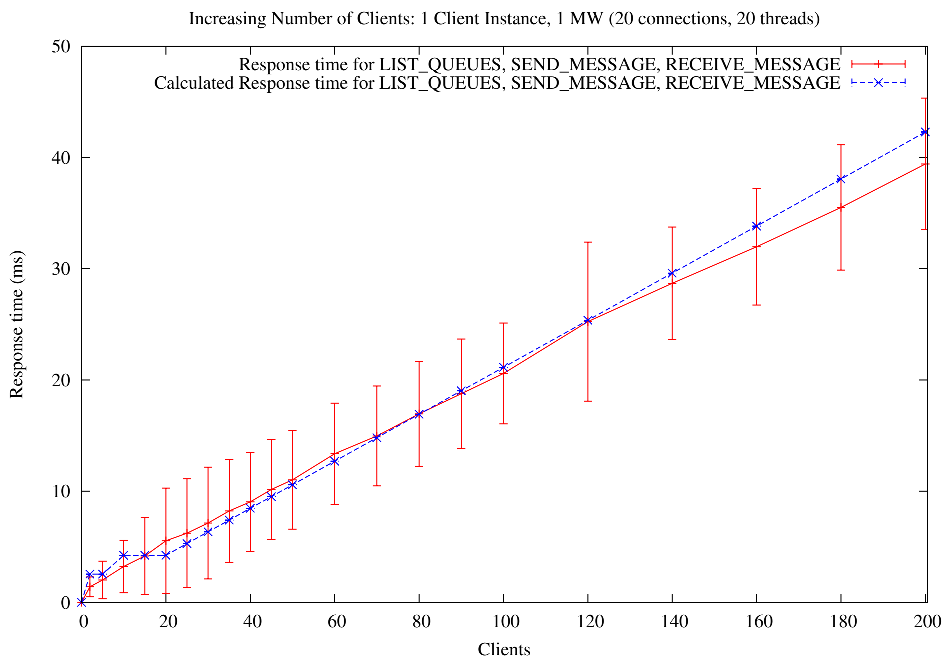

Fig. 10: MVA's Calculated Response Time Compared to Actual Response Time

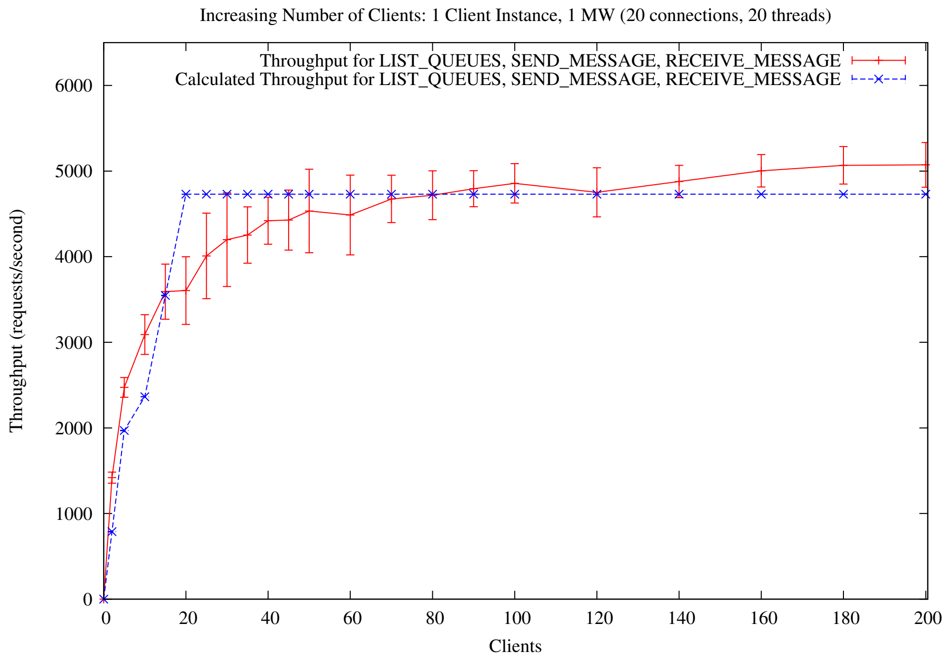

Fig. 11: MVA's Calculated Throughput Compared to Actual Throughput

Visit Ratio

Before going on we would like to describe another possible model shown in Figure 12. We remind the reader that by the design of our system, a socket is being read many times by a middleware thread and is returned back to its sockets queue if it does not contain any data. In this sense the visit ratio of the middleware could be much higher. In spite of that we did not consider this model since it would complicate our analysis. Now we just have a client issuing a request and then the request is immediately queued up which simplifies our analysis.

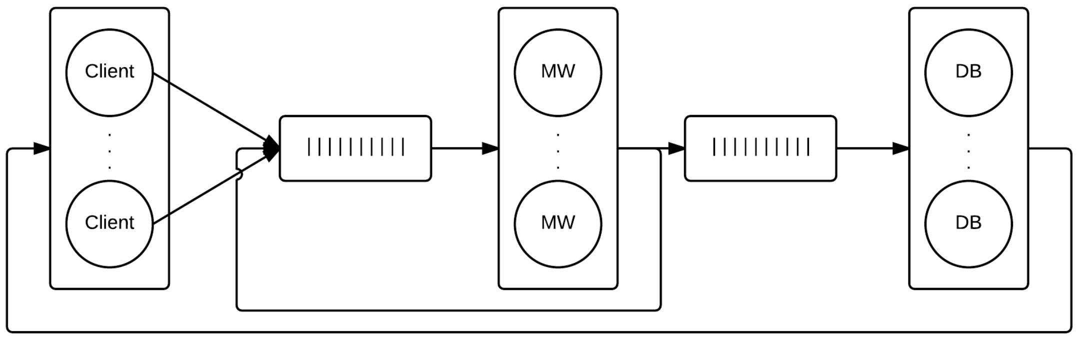

Fig. 12: Queueing Network with Different Visit Ratio

Finding Bottleneck

In the previous report we showed that the database is the bottleneck. In this subsection we are going to show that the database is the bottleneck using operational laws. Based on the queueing network of Figure 9 we know that both the middleware and the database are visited once. We know that the device with the highest demand and hence the highest utilization is the bottleneck device. The demand of a device i can be calculated as follows: $D_i = V_i S_i$ where V_i is the visit ratio of the device and S_i is the service time of the device. Based on our 2 clients experiment described previously we have:

Device	V_i	S_i	D_i
Middleware	1	$0.000534249ms$	0.000534249
Database	1	$1.35318ms$	1.35318

We can therefore conclude that the database is the bottleneck device which was excepted. Note that similar results were given by our MVA's algorithm execution as well. Since the returned utilization per device were approximately 0.01 and 0.99 for the middleware and the database respectively.

8 Conclusion

In this report we explained in detail the interactive response time law and how it applies to our system. We specifically showed that the law holds for every type of request considering different think times. Then we presented our 2^3 experimental design and showed how the number of middleware threads, number of database connections and database instance type interact with each other.

We continued by giving the parts of our system where queuing is happening. Afterwards we made some assumptions and simplifications of our system. We continued modelling our system using one queue, first an M/M/1 queue that proved to be insufficient for describing our system. We continued with an M/M/m queue, one with m servers which proved to have better results but was still far away from our real system.

Then by creating a network of queues and using the MVA algorithm we found results that show the general trend of our actual system.

At the end we would like to note that through this milestone we got accustomed to queueing theory and how it can be used to model a real system.

References

[1] R. Jain. The art of computer systems performance analysis. 1991.