Advanced Systems Lab (Fall'16) – Third Milestone

Name: *Antonios Karkatsoulis*
Legi number: *15-925-298*

Grading

Section	**Points**
1	
2	
3	
4	
5	
Total	

1 System as One Unit

For this part of the Milestone we model our system as an M/M/1 queue and compare it with the experimental data. The way our real is system is mapped to the model is as this: We consider our whole system to be a *black-box*, i.e. everything except the clients is considered to be part of the system. Our **server** part includes the network between the middleware and the `memcached` servers, as well as the servers themselves. The remaining part of the system, i.e. the network between the clients and the middleware, the main receiving thread with the selector queue and the internal queues of the middleware are considered to be our single **queue**. The reason why we chose this mapping from our real system to the M/M/1 model will become clearly later, but basically it is because we can argue more easily about some parameters of our system (e.g. the estimated number of jobs inside the system).

We decided to re-run the stability trace from Milestone 1, since we deemed necessary to keep more accurate timestamps for the different parts of the system each request is in. The new `memaslap` logs, as well as the new middleware log can be found on Gitlab[1]. In order to model the queue, we have to calculate the arrival rate λ and the service rate μ.

- The arrival rate λ can easily be estimated from the reported **average throughput** of the memaslap clients. This is because our system is closed and memaslap clients will only send new requests after they have received responses for the previous ones. Therefore, the rate at which the memaslap clients send their requests, which is the reported throughput, is also the arrival rate for our system.

- The mean service rate μ can be estimated my the **maximum throughput** as measured by the memaslap clients. The maximum throughput is actually a lower bound on the number of requests that our system can process at each time. Given that this maximum throughput was obtained with an ideal configuration for our systems i.e. we wouldn't be able to change the number of clients or threads and achieve higher throughput, we can argue that this value is a good estimate for the mean service rate μ. Moreover, the point in time that the throughput gets maximized is actually the point when the latency due to the network is minimized. This is another reason to choose this value as our mean service rate.

From these 2 parameters we can model our queue and compute the estimated response times. Table 1 presents the values of these parameters. We can confirm that the utilization ρ is less that 1, which means that our system is stable.

λ	μ	ρ
16422 ops/sec	23520 ops/sec	0.698

Table 1: The basic model parameters for the M/M/1 queue.

Table 2 shows all the analysis for our system calculated by the above parameters, as well as the values measured through our experiments.

The **modelled** values are calculated as described in the bibliography[2].

We will now describe shortly how the **measured** values that we report in Table 2 were obtained.

For the mean *response time* we use the average response time, as reported by `memaslap`, since our system is considered as a black-box.

[1] `https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/experiments_m3/trace/trace_new.tar.gz`

[2] "The Art of Computer Systems Performance Analysis", Raj Jain, p. 525, Box 31.1

Parameter	Modelled value	Measured value
Mean response time $E[r]$	0.14 ms	11.83 ms
Variance of response time $var[r]$	0 ms	85.16 ms
50th percentile of response time	0.10 ms	-
95th percentile of response time	0.42 ms	-
99th percentile of response time	0.65 ms	-
Mean #of jobs in the system	2.31 jobs	192 jobs
Mean waiting time $E[w]$	0.10 ms	9.25 ms
Variance of waiting time $var[w]$	-	66.13 ms
50th percentile of response time	0.05 ms	-
95th percentile of response time	0.37 ms	-
99th percentile of response time	0.60 ms	-
Mean #of waiting jobs	1.62 jobs	157 jobs
Mean service time $E[s]$	0.04 ms	2.58 ms
Variance of service time $var[s]$	-	19.02 ms
Mean #of jobs in service	0.70 jobs	35 jobs

Table 2: Results of the M/M/1 model analysis.

The mean *service time* is exactly the average server time as measured by the middleware log, i.e. the time between sending a request to the server (or servers for replicated SETs) and receiving the response(s) back. This of course includes the queuing and processing inside the `memcached` server, as well as the network time between the middleware and the server, but there are not enough data to measure those. The mean *waiting time* is the remaining time, i.e. *mean response time - mean service time*. For the *variances*, since $var[r]$ is reported by `memaslap` and $var[s]$ can be calculated by the middleware logs, we can compute $var[w] = var[r] - var[s]$, assuming that the variables are independent. For the computation of the *number of jobs* inside our system: For the total mean number of jobs in the system, we can assume that it is equal to the number of memaslap clients while our system is in steady state (192 clients in our case) . This is due to the behaviour of the memaslap program, that ensures that for each client, a new request is sent only after the response for the previous one has been received. Moreover, since we take our system to include For the numbers of jobs in the servers and in the queue we do the following: We run exactly the same configuration for 10 minutes, but now we instrument our middleware code to log along with the request timestamps, the number of GET threads that are in the `RUNNABLE` state, i.e. the number of threads that are busy at that specific point in time. SETs comprise only 1 % of our total requests, so we can ignore them. So he average number of busy threads is found to be 35. The threads are mostly busy waiting for the server responses and so we can assume that approximately 35 jobs are being processed on average at any time by the servers. The average number of jobs waiting in the queue can then be assumed to be the remaining 192-35=157 requests. The modifications in our code[3] in order to instrument this experiment, as well as the respective logs[4] are available on GitLab.

Discussion

From Table 2 we can clearly notice that the model does not fit with our experimental data. This is because our model is oversimplified and does not take into account the parallelism that is induced by the simultaneous processing of the middleware threads. As an example ti clarify this, say that we measure the average response time of a request to be 1ms and we have 10 busy threads processing the requests at any point in time. The inherent parallelism cannot be modeled with the M/M/1 model and so when the model sees this service rate it will think that each request takes only 0.1 ms on average. It can however predict the long-tail distribution of

[3] `https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/src/asl/GetThread.java`

[4] `https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/experiments_m3/busy_threads/threads.tar.gz`

the response times.

If our system had only one processing thread then the model would actually predict the response times a lot more accurately. For example if we multiply the mean number of jobs in the servers that we estimated before with the modelled mean service time $35 * 0.04 = 1.4ms$, we get a more accurate estimate (measured time was 2.58 ms). The same can be done with the mean waiting time. If we multiply the modelled waiting time with the number of threads per queue we get $0.10 * 16 = 1.6ms$, which also is a better estimate (real is 9.25 ms).

An interesting find is that, although the M/M/1 model cannot predict the absolute values for the response times, it can pretty accurately predict the proportions of waiting and service time with respect to the total response time. As a matter of fact the model predicts that 70 % of the total response time will be the waiting time and 30 % will be service time, while the measured percentages are 78 % and 22 % respectively.

2 Analysis of System Based on Scalability Data

For this part of the Milestone we analyse and model the system as M/M/m queues based on the data from the third part of Milestone 2 (**3 Effect of Writes**). We decided to choose this set of experiments, because they include the maximum number of parameters available and so we are able to get a better overview of the scalability of our system. The configurations we decided to use are the ones that have the extreme values for our 3 parameters, i.e. **no-full replication**, **3-7 servers** and **1-10 % write percentage**. We chose the extreme values because:

- We wanted to contain the number of models that we were going to build, so that it would be easier to draw conclusions from them. Taking into account all the experiments that we carried out for the third part of Milestone 2, for example, would result in 18 different M/M/m models.
- We know from Milestone 2 that these 3 parameters, affect the performance of our system monotonically, so there are not really many conclusions we can get from the middle values (5 servers and 5 % writes and half replication for all the configurations.)

So for each of the configurations we calculate the arrival rate λ and the service rate μ, exactly as in the first experiment, i.e. λ is the average throughput as reported from `memaslap` and μ the maximum throughput again as reported from `memaslap`, but this time divided by the number of servers m. More specifically, since we have 5 runs for each experiment, we take the average of the maximum value over the experiments for the above reported values. The basic parameters for each M/M/m model are summarized in Table 3.

Configuration (servers, replication, write percentage)	λ	μ	ρ
(3, 1, 1)	15426	6765	0.760
(3, 1, 10)	14960	7053	0.707
(3, 3, 1)	15516	7317	0.707
(3, 3, 10)	14416	6760	0.711
(7, 1, 1)	13145	2398	0.783
(7, 1, 10)	12678	2464	0.735
(7, 7, 1)	13097	2446	0.765
(7, 7, 10)	11194	2162	0.740

Table 3: Interarrival rates, service rates and utilizations for the chosen configurations.

We can notice that the values for the server utilization ρ don't vary much when the configuration is changing (if we also take into account the fact that μ is an estimate, so we can get some small variation from that). This can be explained as follows:

- When we increase the number of servers, there is less load for each server, but the effect of the network is bigger, since now we have a lot more network connections.
- When we increase the replication factor, a similar argument can be used: each server gets more load, because it also has to serve other SET requests not bound for it, but again we have more traffic over the network that keeps the utilization at almost the same levels.
- When one increases the percentage of SETs, it seems like the load in the servers would increase, but in fact the GETs are also affected, since they have to wait for the slow SETs to complete every time (due to the bahaviour of `memaslap`.)

We are now going to investigate the differences of the response times between the models and the real world. The **modelled** values for our analysis are again calculated according to the

book[5]. In Graphs 1 to 6 we present the mean response times for the different configurations, broken down to waiting and service time, as calculated from our model and as measured by our experiments. The way the times were collected from our system is exactly the same as in Part 1. Table 4 shows the mean number of jobs in the system, in the queue and in service as calculated by the M/M/m model for the different configurations, along with the estimated ones from the real system. The way we calculated the values for the real system was by carrying out the same experiment as in part 1, with the purpose of finding out the mean number of busy threads.

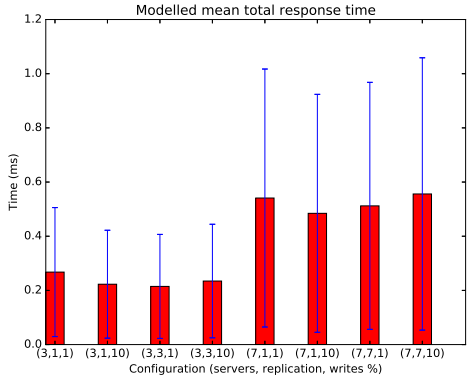

Figure 1: Mean **response time** as calculated by the **M/M/m** model for the chosen configurations.

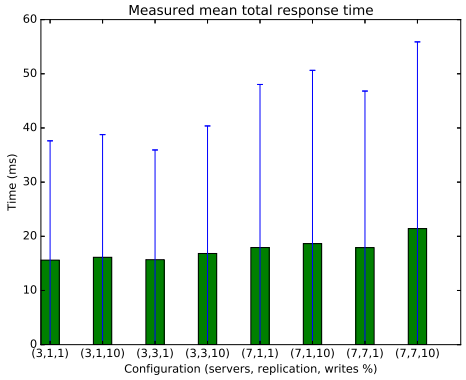

Figure 2: Measured mean **response time** for the chosen configurations.

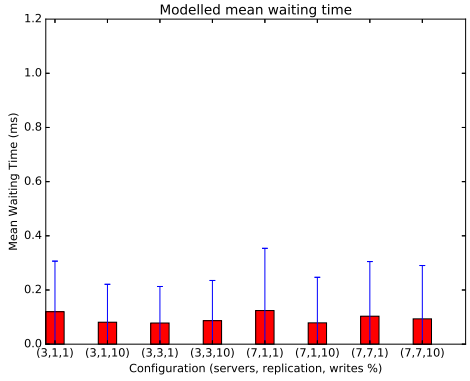

Figure 3: Mean **waiting time** as calculated by the **M/M/m** model for the chosen configurations.

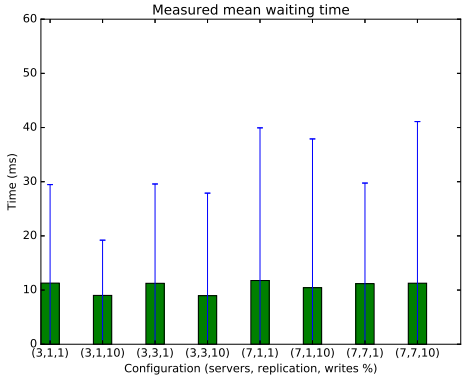

Figure 4: Measured mean **waiting time** for the chosen configurations.

[5]"The Art of Computer Systems Performance Analysis", Raj Jain, p. 528, Box 31.2

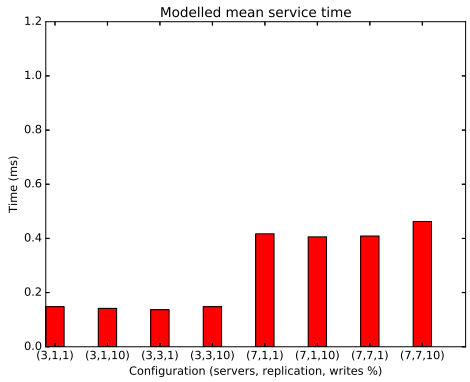

Figure 5: Mean **service time** as calculated by the **M/M/m** model for the chosen configurations.

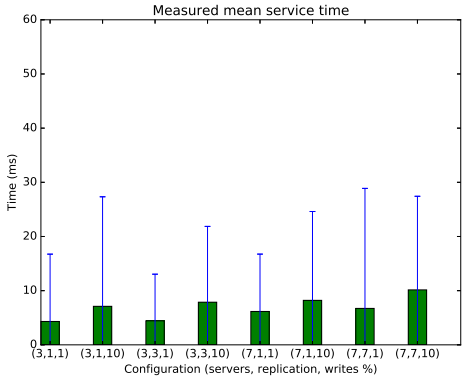

Figure 6: Measured mean **service time** for the chosen configurations.

Configuration (servers, replication, writes %)	Model $E[n]$ (jobs)	Real $E[n]$ (jobs)	Model $E[n_q]$ (jobs)	Real $E[n_q]$ (jobs)	Model $E[n_s]$ (jobs)	Real $E[n_s]$ (jobs)
(3, 1, 1)	4.13	246	1.85	187.52	2.28	58.48
(3, 1, 10)	3.33	246	1.21	191.36	2.12	54.64
(3, 3, 1)	3.33	246	1.21	188.82	2.12	57.18
(3, 3, 10)	3.38	246	1.25	192.8	2.13	53.20
(7, 1, 1)	7.11	246	1.63	156.95	5.48	89.05
(7, 1, 10)	6.14	246	1.00	163.53	5.14	82.47
(7, 7, 1)	6.70	246	1.35	154.11	5.36	91.89
(7, 7, 10)	6.22	246	1.05	167.41	5.18	78.59

Table 4: Number of jobs in the system, queue and service as predicted by the M/M/m models (**Model**), compared to the experimental values (**Real**).

Discussion

- The first thing that we notice is that the M/M/m models are still incapable of predicting absolute response time for the system, i.e. predicted response times are still almost an order of magnitude lower than the real ones (Figures 3 and 4). This happens because, although the real system maps a bit better to the model (which now takes into account the distinct servers), it still fails to take into account the parallelism of the real system, due to the thread pools. In order to check if the parallelism is actually the reason why the model doesn't fit to our experimental data, we scale the service times predicted by our model by the factor $\frac{\#\text{ busy threads}}{m}$ (because we want to get the number of busy threads per server). The results are shown in Table 5. What we notice here is that for the configurations with 1 % writes, the scaled model service times are comparable to the measured ones. For 10 % writes, there is a bigger mismatch, which can be attributed to the fact that our scaling factor only accounts for the busy `GET` threads, while now the `SET` requests account for a significant part. So we can conclude that the parallelism is the main reason that our model doesn't fit to our experimental data.

Configuration (servers, replication, writes %)	Model service time (ms)	Scaled model service time (ms)	Measured service time (ms)
(3, 1, 1)	0.148	2.89	4.31
(3, 1, 10)	0.142	2.59	7.11
(3, 3, 1)	0.137	2.61	4.44
(3, 3, 10)	0.148	2.62	7.87
(7, 1, 1)	0.417	5.30	6.16
(7, 1, 10)	0.406	4.78	8.21
(7, 7, 1)	0.409	5.37	6.73
(7, 7, 10)	0.463	5.20	10.15

Table 5: Comparison between scaled model service times and measured service times.

- We also notice that with an increasing number of servers, the model predicts that the service time also increases (Figure 5). This is basically due to the service rate going down, because for the same reasons that were explained in Milestone 2(CPU contention, more network effect). The experimental data also show a very slight increase as well (Figure 6). The important thing here is that, as the requests take more time to complete inside the servers, the servers become busier. So, although the load per server is now smaller, the utilization eventually stays pretty much the same, something that was confirmed in Table 3.

- As far as the waiting times are concerned, the model seems to predict the trend that they stay pretty much the same. The reason behind that is probably the variations in the maximum throughput for each configuration, a value that we chose as an estimate for μ. Normally we would expect a bit lower waiting times as the number of servers increases, because of a larger number of total threads. However, as we mentioned, unexpected variations in the max throughput (and consequently μ) probably drive the waiting times higher.

- From Table 3, we can also see that the M/M/m models fail to predict the number of jobs in the system. This follows naturally again from the fact that the model cannot take into account the parallelism in our system and so the predicted service rates are a lot lower than reality, as we explained in the first part of the Milestone. So, the model predicts that system processes jobs much faster than reality and that is why we get this discrepancy between the model and the experiments.

So as a conclusion, we could say that the M/M/m models can predict the trends of the response time as the configurations change, but fail to predict the absolute value. The main reason for this would be the fact that it can not account for the inherent parallelism in our system.

3 System as Network of Queues

For this part of the Milestone we model our system as a closed, multiclass queueing network using Mean Value Analysis (MVA). Our system is closed, because `memaslap` clients wait for responses from the requests they have sent, before issuing new requests. The 2 classes correspond to the `GET` and `SET` requests. The configuration that we decided to model is the one with 3 servers, full replication and 10 % write proportion. We chose that because we wanted to build a model that corresponds to a system with as many `SET`s as possible and the full effect of replication in place. However, we kept the number of servers to the minimum to avoid making it too complex in the end.

3.1 Description of the queueing network.

Figure 7 represents a schematic of the model that we used in order to predict the behaviour of our system. We will now describe the different parts of the queueing network and how they are mapped to the real system:

- The **network** between the clients and the middleware is modelled as an infinite queue (delay center) (Device 1). Its service time is set to the average network round-trip time between the clients and the middleware. In order to find this we subtract the time spent inside the middleware and the servers (from the logs) from the total response time as measured by `memaslap`. The reason why we use the round-trip time and not the latency(i.e. half of the round-trip time), is because with this infinite queue we model **both** the network part from the clients to the middleware, as well as the network part from the middleware back to the clients (response).

- The **selector** in the main thread of our middleware is modelled as an M/M/1 load-independent device (2), since its implementation is asynchronous. Its service time is set to the average time between receiving a request from a client and enqueuing it to one of the queues, which of course is really low.

- The **GET queues** and **threads** are modelled each by a load-dependent M/M/40 queue (Devices 3,6,9). The number of servers in the queue is set to the number of processing threads per server, which in our case is 40. This queue models not only the queueing and processing of a thread, but also the queueing and processing inside the `memcached` servers and this is the reason why the service time here is set to the mean time between dequeuing a request and receiving the response back from the servers (replicated case).

- The **SET queues** and **threads** are a bit more difficult to model and we expect our predictions here to be slightly worse. The best way we could find to model them, while keeping the model relatively simple was to pipe 2 queues together: one M/M/1 queue that is load-independent to model the asynchronous thread with service time set to the time of one "cycle" of our `SET` thread which includes: dequeueing a request, sending it to the server, checking for responses from the server and if a response is available, sending it back to the client (Devices 4,7,10), and a delay center (Devices 5,8,11) that models the processing of the `SET` requests in the `memcached` servers. The service rate of the delay center is set equal to the mean server time for the `SET` requests as measured by the middleware logs.

Table 6 lists all the devices of the queueing networks, as well as the variables used for their analysis.

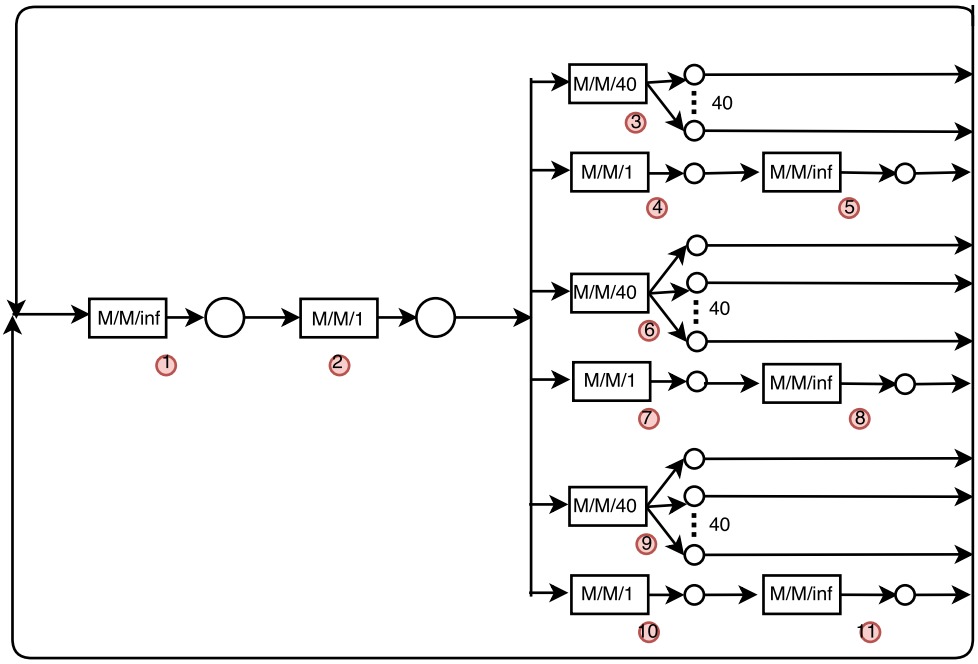

Figure 7: The closed, multiclass queueing network that models our system.

Device ID	Service time (μs)	# Visits
1	7827	1
2	24	1
3,6,9	7871[6]	1/3
4,7,10	24	1/3
5,8,11	10810	1/3

Table 6: Service time and average number of visits for the devices in the queueing network.

3.2 Analysis of the queueing network.

In order to analyse our queueing network we used the *Java JMT Tool*[7] and more specifically the `JMVA` package, that performs the Mean Value Analysis algorithm to a given queueing network. One problem that we encountered during our analysis with *JMT* was that, the program does not allow inserting load-dependent stations, when we use 2 classes of requests. This led us to the decision to carry out two separate analyses, one for the `GETs` and one for the `SETs` and combine the results in the end. We are free to do that because the processing of the two different types of requests takes place independently. For the parts that are the same (i.e. the network and the selector) for both classes, we add up the utilizations (which in this case are the number of requests in the devices), in order to get the final utilization for the device. Since for this configuration we used a total of 246 clients, meaning a total of 246 requests in steady state and 10 % of them are `SETs`, our input to the queuing network comprises of 221 `GET` requests and 25 `GET` requests. The results of the analysis can be found on Table 8.

[6]The device is load-dependent so the service rate actually depends on the number of clients and is $7871/min(\#clients, \#threadsperpool)$.

[7]`http://jmt.sourceforge.net/`

Device ID	Type	# requests	Throughput(ops/sec)	Utilization
1	Network	111.81	15004	111.81
2	Selector	0.35	15004	0.49
3,6,9	GET M/M/40	4555	37.85	1
4,7,10	SET M/M/1	0	446	0
5,8,11	SET M/M/inf	6.70	446	6.70

Table 7: Outputs of the queueing network analysis.

Finally on Table 8 we present the response times for the `GET` and `SET` requests separately, as predicted by the analysis of our queueing network, along with the experimental ones as measured by the `memaslap` clients.

Type of request	Model response time (μs)	Measured response (μs)	Model TPS (ops/sec)	Measured TPS (ops/sec)
`GET`	16173	16924	13665	13110
`SET`	21282	24286	1339	1456

Table 8: Comparison of response times and throughput between the model and the experimental values.

3.3 Discussion

From the utilizations in Table 8 we would normally conclude that the bottleneck of our system is actually the network between the clients and the middleware since it has the highest utilization value. However, it is known that an M/M/inf can never be the bottleneck in a system, so both the network (1) and the `SET` servers are excluded from the possible bottlenecks. This leads us to conclude that the bottlenecks are actually the `GET` devices (utilization 1). The problem here, is that since we have modelled our M/M/40 queues to include both the `GET` threads and the processing inside the servers , we cannot really distinguish which of the two is the actual bottleneck. We conclude however that **the bottleneck is at the `GET` threads** because:

- We already know from Milestone 2 (maximum throughput experiment), that when we increase the load, the server time stays the same, while the waiting time increases. This means that the `memcached` servers are actually keeping up with the increased number of requests, but the `GET` threads cannot.
- We know from experience that the `memcached` servers are heavily optimized to handle large numbers of requests, in systems a lot bigger than the one at hand. Therefore, it is not likely that they are the bottleneck here.

The predictions of the queueing network model seem to be pretty accurate for the `GET` requests as seen on Table 8. It was expected that this complicated queueing model would give more accurate predictions than the M/M/1 or the M/M/m. For the `SET` requests however, we notice that the predictions are not that accurate, although at the same order of magnitude and still a lot better than the simpler M/M/1 and M/M/m models. We notice though, that the difference between measured and predicted response time (3 ms) is almost the same as the queuing time for the `SET`s (2.6 ms). This leads us to the conclusion that our model fails to model the queue for the `SET`s. One possible explanation for this has to do with our implementation of the `SET` threads that cannot be modelled accurately. When a thread picks up a request from the queue, sends it to the servers and checks for responses, sometimes it finds them and has to forward them back to the client. During that time, the new requests that will arrive, will have to pile up inside the queue, thus increasing the queue time. Our model cannot account for that, and this is the reason why we get this discrepancy in the results for the `SET`s.

4 Factorial Experiment

For this part of the Milestone we carry out a $2^3 5$ factorial experiment, so that we have $\mathbf{k = 3}$ factors and $\mathbf{r = 5}$ replication of the experiments. Our factorial experiment is based on the results of the third experiment of Milestone 2 (**3 Effect of Writes**), where we measured the throughput of our system with varying number of servers, replication factor and percentage of writes. Our 3 factors are the parameters of this experiment: the **number of servers**, the **replication factor** and the **percentage of writes** and the 2 levels are going to be the extreme values, meaning 3-7 servers, no-full replication and 1-10 % writes. The reason why we choose the extreme values of the factors as our levels is because we know the effect of each of the two factors is unidirectional, that is, the performance continuously decreases as the factor is increased from minimum to maximum.

4.1 Computation of Effects

As we mentioned before, we define our 3 variables x_A, x_B and x_C as:

$$x_A = \begin{cases} -1 & \text{if 3 servers} \\ 1 & \text{if 7 servers} \end{cases}$$

$$x_B = \begin{cases} -1 & \text{if no replication} \\ 1 & \text{if full replication} \end{cases}$$

$$x_C = \begin{cases} -1 & \text{if 1 \% writes} \\ 1 & \text{if 10 \% writes} \end{cases}$$

In order to analyze the $2^3 r$ design we use the typical sign table used in the factorial experiments, but we include the throughput results from all the repetitions of the experiment (vector y) and also calculate their mean value $\bar{y}$. We assume an *additive* model for our analysis since the ratio of the maximum to the minimum value throughout all the repetitions and the experiments is $y_{max}/y_{min} = 15964/10614 = 1.50$, which is small.

I	A	B	C	AB	AC	BC	ABC	y	Mean $\bar{y}$
1	-1	-1	-1	1	1	1	-1	(15157, 15707, 15375, 15437, 15451)	15425
1	-1	-1	1	1	-1	-1	1	(14838, 14748, 15225, 15134, 14854)	14960
1	-1	1	-1	-1	1	-1	1	(15963, 15642, 15554, 15964, 14727)	15516
1	-1	1	1	-1	-1	1	-1	(14822, 13874, 13967, 14811, 14606)	14416
1	1	-1	-1	-1	-1	1	1	(12900, 13401, 12945, 13285, 13196)	13145
1	1	-1	1	-1	1	-1	-1	(13002, 12747, 12434, 12443, 12761)	12677
1	1	1	-1	1	-1	-1	-1	(12779, 13225, 12844, 13277, 13360)	13097
1	1	1	1	1	1	1	1	(11493,11148, 11350, 11363, 10614)	11193

Table 9: Sign table for the calculation of the effects.

From the sign table we can compute the effect coefficients by summing up each respective column according to the signs and dividing by 8. The effects are given in Table 10.

q_0	q_A	q_B	q_C	q_{AB}	q_{AC}	q_{BC}	q_{ABC}
13804	-1275	-248	-492	-135	-101	-259	-100

Table 10: Computation of the effects q_i.

4.2 Estimation of experimental errors

The estimate of the throughput based on the linear regression model is given by:

$$\hat{y_i} = q_0 + q_A x_{Ai} + q_B x_{Bi} + q_C x_{Ci} + q_{AB} x_{Ai} x_{Bi} + q_{AC} x_{Ai} x_{Ci} + q_{BC} x_{Bi} x_{Ci} + q_{ABC} x_{Ai} x_{Bi} x_{Ci}$$

where $\hat{y_i}$ is the estimated throughput when the factors A, B and C are at levels x_{Ai}, x_{Bi} and x_{Ci} respectively.

We then compute the error between the estimated and the real value for every pair of levels and for each run.

i	I	A	B	C	AB	AC	BC	ABC	e_{i1}	e_{i2}	e_{i3}	e_{i4}	e_{i5}	$\sum_{j=1}^{5} e_{ij}$
1	1	-1	-1	-1	1	1	1	-1	-268.4	281.6	-50.4	11.6	25.6	0
2	1	-1	-1	1	1	-1	-1	1	-121.8	-211.8	265.2	174.2	-105.8	0
3	1	-1	1	-1	-1	1	-1	1	447	126	38	178	-789	0
4	1	-1	1	1	-1	-1	1	-1	406	-542	-449	395	190	0
5	1	1	-1	-1	-1	-1	1	1	-245.4	255.6	-200.4	139.6	50.6	0
6	1	1	-1	1	-1	1	-1	-1	324.6	69.6	-243.4	-234.4	83.6	0
7	1	1	1	-1	1	-1	-1	-1	-318	128	-253	180	263	0
8	1	1	1	1	1	1	1	1	299.4	-45.6	156.4	169.4	-579.6	0

Table 11: Computation of errors for each of each of the observations. (Rounded numbers)

The variance of the errors can be estimated by the sum of the squared errors calculated above and is:

$$SSE = \sum_{i=1}^{8} \sum_{j=1}^{5} e_{ij}^2 = 3232937.6$$

4.3 Allocation of Variation

In order to calculate which factor is more significant for the performance (throughput in our case) of the system we compute the variation due to each of them. If we also take into account the sum of squared errors SSE that we computed before, we can also compute the Total Sum of Squares (SST) as:

$$SST = SSA + SSB + SSC + SSAB + SSAC + SSBC + SSABC$$

where

$$SSj = 2^k r q_j^2, \quad \text{where } j = A, B, C, AB, AC, BC, ABC$$

In Table 12 we present all the SS values along with their contribution to the total SST:

Component	SSA	SSB	SSC	SSAB	SSAC	SSBC	SSABC	SSE	SST
Value	65073459	2463633	9687481	727651	405821	2677545	401001	3232938	84669528
% SST	**76.86 %**	2.91 %	**11.44** %	0.86 %	0.48 %	3.16 %	0.47 %	3.82 %	100 %

Table 12: Sum of Squares values for the factors and their contribution to the total variation.

4.4 Confidence Intervals for Effects

We now compute the confidence intervals for the effects based on the 5 repetitions of the experiments we carried out. If we assume that errors are normally distributed with zero mean and variance σ_e^2, then it follows from our model that the measured values y_i are also normally distributed with the same variance σ_e^2. The variance of the errors is estimated by the SSE as:

$$\sigma_e^2 = \frac{SSE}{2^2(r-1)} = 101029.3$$

The estimated variance of each of the factors q_i is:

$$s_{q_0}^2 = s_{q_A}^2 = s_{q_B}^2 = s_{q_C}^2 = s_{q_{AB}}^2 = s_{q_{AC}}^2 = s_{q_{BC}}^2 = s_{q_{ABC}}^2 = \frac{s_e^2}{2^3 r} = 2525.7$$

Now we are ready to compute our confidence intervals for the effects. We have $2^3(r-1) = 32$ degrees of freedom and we choose 90 % confidence, thus our t-value is 1.697. The confidence intervals for the effects can be computed as:

$$q_i \mp t s_{q_i}$$

Table 13 presents the variance of each effect, as well as the respective confidence intervals.

Effect	q_0	q_A	q_B	q_C	q_{AB}	q_{AC}	q_{BC}	q_{ABC}
CI	[13719, 13889]	[-1361, -1190]	[-333, -162]	[-577, -407]	[220, -50]	[-186, -15]	[344, -173]	[-185, -15]

Table 13: Confidence Intervals for the effects (90 % confidence).

4.5 Discussion

- As we can see from Table 12, by far the most important factor (77 %) is the **number of servers**, something that we had already found out through our experiments in Milestone 2. There we saw a drop in throughput as we increased the number of servers and we attributed it to a number of factors, such as CPU contention and larger effect of the network.
- The second most important factor (11.5 %) is the proportion of writes in our workload. Moreover, the replication factor doesn't seem to be affecting system performance at all (3 % allocated variation). The reason why the proportion of writes is more important than the replication factor can be justified as follows: There is a bigger effect on performance when we turn `GET`s into `SET`s both for the replicated and the unreplicated case, than turning the same amount of `SET`s from unreplicated to replicated, because we know that `GET`s are more expensive than `SET`s. Another reason could be that an increasing replication factor has a significant effect only when the write proportion is large, while an increasing write proportion always has a significant effect regardless of the replication factor, simply because now 9 % of the requests turned from `GET`s to `SET`s (i.e. from cheap to expensive).
- All the interactions between the factors are not significant, except for the **interaction between the replication factor and the percentage of writes**. This was also expected as we know that the replication factor has a lot more impact on performance when the number of writes increases, or vice versa.
- From Table 13 we can see that none of the confidence intervals include 0, so with 90 % confidence all of the effects are significantly different than 0.
- The fact that the variation due to the error is very small (almost 4 % of the total variation), together with the low y_{max}/y_{min} ratio, leads us to the conclusion that additive model is a good fit for our experimental data.

5 Interactive Law Verification

For the last part of the milestone, we check the validity of our experiments from the second part (**2 Effect of Replication**) of Milestone 2, using the Interactive Law. If X is the reported average throughput from `memaslap` and N the load of our system, which in our case is the number of total virtual clients, then the expected response time R will be:

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

where Z is the think time of the `memaslap` clients. In order to calculate the average think time of the clients we carry out a small experiment: On the same Azure virtual machine, we instruct one virtual client only, to send requests to one `memcached` server through our middleware. The reason we chose to run the client and the middleware on the same machine was to avoid the overhead of the network while measuring think time. Of course the measurement is still not accurate, since even through the `localhost` there is still some overhead to send the requests, but in general it is a good approximation. We then measure the think time of the client for every request as the difference of the time when this request was sent and the time when the previous request was received. Table 14 shows the experimental setup for this experiment and Table 15 shows the statistics for the think times measured.

Server, Client, Middleware Machine	1 X Basic A2 (2 cores, 3.5 GB memory)
Virtual clients	1
Memcached servers	1
Threads per pool	16
Workload	Key 16B, Value 128B, 1% writes
Replication Factor	1
Overwrite proportion	1
Memaslap window size	10k
Log File	`mw_interactive.log`[8]

Table 14: Setup for the think time experiment.

Think time statistic	Value (ms)
Average	0.143
Standard Deviation	0.983
Median	0.018
90th Percentile	0.029
95th Percentile	0.042
99th Percentile	6.757

Table 15: Statistics for the think time distribution of the memaslap clients.

From Table 15, we can deduce that most of the values for the think time are negligible compared to the total response times, as seen in Table 16. Even taking into account that the standard deviation of the think time is quite high, because of the few requests that have a huge thinking time, the values are still negligible. Therefore we can safely **neglect the think time** for our calculations. From the 99th percentile, we can also notice that there are a few

[8] `https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/experiments_m3/interactive_law/mw_interactive.log`

requests that have unusually high think times. These can probably be attributed to the network stack, because as we said before, we cannot completely remove the network overhead, even when running at `localhost`. We should also note here that there existed a few measurements for the think time that were negative. This happens, because sometimes the GET threads take quite a long time to set up the timestamp, so it can happen that they set it after the main thread has set the starting time for the next request. We decided to drop those requests completely, as they would lead to a worse thinking time estimate.

In table 16 we present the reported throughput, the measured response time, the response time calculated by the Interactive Law, neglecting Z, as well as the percentage error between them.

(Servers, Replication)	Measured Average TPS (ops/sec)	Calculated Average Response Time (ms)	Measured Average Response Time (ms)	Percentage Error
(3,1)	15723	14.88	15.65	4.9 %
(3,2)	15402	15.17	15.97	5.02 %
(3,3)	14996	15.59	16.40	4.96 %
(5,1)	13928	16.91	17.66	4.26 %
(5,3)	13254	17.84	18.56	3.88 %
(5,5)	12573	18.89	19.57	3.45 %
(7,1)	13083	18.30	18.80	2.68 %
(7,4)	12315	19.69	19.98	1.43 %
(7,7)	11629	20.48	21.15	3.19 %

Table 16: Verification of the Interactive Law for the different configurations

The deviation of the Interactive Law calculated values, from the measured ones is between 1 and 5 % for the different configurations, that can be regarded as small. We also notice that the measured average response times are always larger than the ones calculated by the Interactive Law. This happens because the interactive law cannot model some random high network latencies that can be induced in the requests. We know that some requests are going to experience these large network latencies and we also know that the request times are distributed in a long tail distribution. So, every request that experiences these network latencies contributes to the increase of the measured average response time more.

Appendix A

In this appendix we include the stability trace for the experiment that was re-ran for the first part (**1 System as One Unit**). Although the only change we made to the middleware was to instrument it to log all the timestamps, we include the throughput and response time graphs again over the period of the experiment, as a proof that our system still behaves as expected.

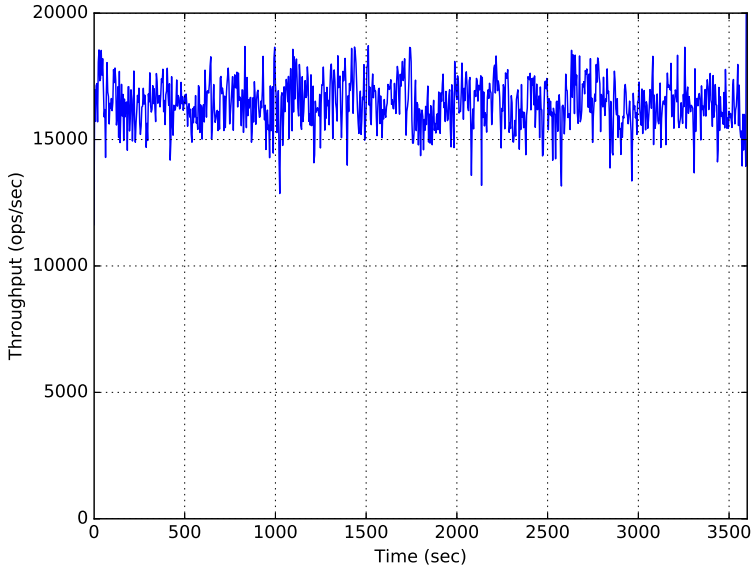

Figure 8: Aggregate throughput for the 3 client machines over 1 hour. (Sampling every 4 seconds)

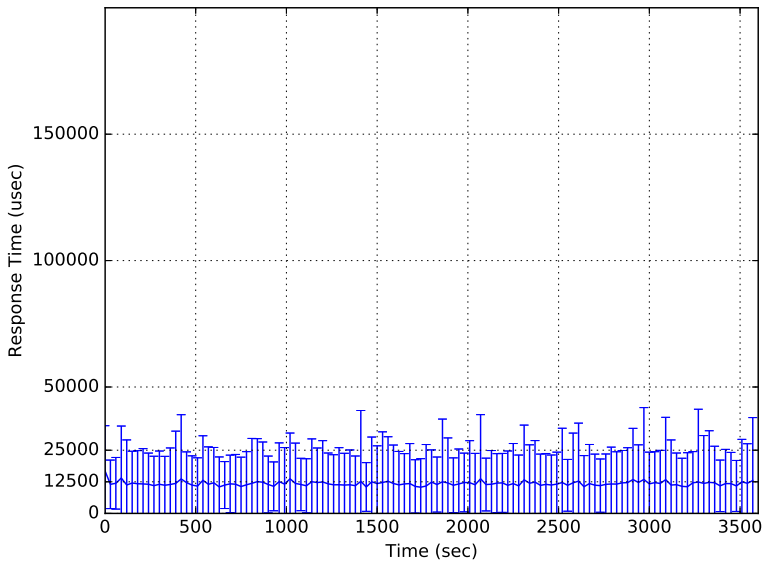

Figure 9: Average response time per request over 1 hour. (Sampling every 30 seconds)

Logfile Listing

Short name	Location
trace_new	https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/experiments_m3/trace/trace_new.tar.gz
mw_interactive.log	https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/experiments_m3/interactive_law/mw_interactive.log
busy_threads	https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/experiments_m3/busy_threads/threads.tar.gz