Advanced Systems Lab (Fall'16) – Third Milestone

Name: *Petar Ivanov*
Legi number: *16-931-636*

Grading

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

1 System as One Unit

1.1 Model

Modeling the whole middleware as a single M/M/1 queue can be seen as an abstraction of the actual architecture, as seen from the point of view of the memaslap clients. When a client sends a request it is unaware that the middleware can be communicating with more then one backend server, so the abstraction says that all requests are queued into a single queue and sent for handling to a single resource.

The first thing to do in the model analysis is calculate the arrival rate λ. Since this is done using the internal logs of the middleware (*RequestLog_stability_microbench.log* from Milestone 1), we should take into account the sampling rate. For the stability trace the rate was 100 for both get and set requests. Since the arrival rate is part of the set of operational quantities, we can calculate for a finite observation period using the following formula:

$$A\,\lambda = \frac{number\ of\ arrivals\ A}{time\ T}$$

For each of GETs and SETs, we calculate λ by multiplying the number of entries in the log files by a 100 and then add the two results. This gives a value of 11500, which is the same as the throughput, as seen from the memaslap logs in the microbench (*memaslap_stability_microbench.tar.gz*) and the experiment in Milestone 1 (*RequestLog.log*).

Next, we calculate the mean service rate μ, by first computing the mean service time $E[s]$ per get and set request. This is done using the internal logs of the middleware from the microbenchmark. Service time of a request is considered to be the time between taking off the request from the queue and enqueueing the server response for sending it back to the client. Thus, the service time for a get request is $T_{server} + T_{enqueue\_return}$ and for sets it is $T_{processtime} + T_{server} + T_{enqueue\_return}$. Similar to the results from Milestone 2 (Figure 10 an Figure 11), we get $E[s]get = 2.8ms$ and $E[s]set = 7.5ms$. We then apply the formula:

$$\mu = \frac{1}{E[s]}$$

to get respectively 355 and 133 for GETs and SETs. To get the total μ for the middleware we take into account the read-write ratio and apply the following formula: $0.99 * E[s]get + 0.01 * E[s]set = 352.64$. Finally, we need to verify the stability condition by computing the traffic intensity ρ.

$$\rho = \lambda/\mu$$

However, the result we get from this calculation is $\rho = 32.6$, which is impossible for a closed system. The conclusion, which can be reached is that an M/M/1 model of the entire middleware is not a suitable for this architecture. However, when calculating the traffic intensity, we can take into account that there are actually more than one queue and server. From *microbench_parsed_stability_metrics.log*, we see that all servers and get threads are being utilized almost equally, so we can change the way we calculate μ to $0.99 * threads * servers * E[s]get + 0.01 * E[s]set * servers$, which results in 16866.78. Thus, the true value of ρ is 0.68. This utilization may seem a little low, but as seen from Milestone 2, under this configuration (192 clients, 3 servers, full replication), the server is underloaded. Thus, these results can be used for further analysis of the M/M/1 model.

1.2 Model Results and Real System Behavior

The results of real-life actual behavior of the system are all stored and taken from *Request-Log_stability_microbench.log* and *microbench_parsed_stability_metrics.log.*

First of all, since GET threads are synchronous and the work is spread evenly among them, we can deduce that there are $threads * server = 48$ get **requests being handled at a certain time**. The parsed logs say that each of the SET threads has on average $0.97, 0.99, 0.97$ requests being served simultaneously. Thus, the middleware on average has 51 requests being served. Calculating the same metric using the equation:

$$E[n] = \rho/(1 - \rho)$$

gives 2.125. This result is far from the real value, because the model describes an architecture with a single queue and server.

Next, lets calculate the **mean number of jobs in the queue**. Each of the three *Queue-Managers* (one per server) has on average respectively $11.926, 12.074$ and 14.443 get requests divided among the threads responsible for each queue. That makes 38.44 GETs in all queues at a given time. Since the SET thread is asynchronous it takes requests off the queue much faster, which means there are a total of 0.009 requests in all of the queues. Thus, the middleware has on average 38.45 requests waiting in all queues combined. Calculating this using the module we get:

$$E[n_q] = \rho^2/(1 - \rho) = 1.445$$

requests waiting in the entire middleware. Again, because the abstraction it is a closed system with just one queue, the model underestimates the number of requests in the queues of the middleware.

The mean response time for this configuration, as seen by the memaslap output for the stability trace, is 18.94 ms. This includes the both the waiting time and receiving service, as per the definition of response time. Using the M/M/1 model, the it can be computed using Little's Law, which states that Mean number of requests in the system equals the arrival rate times the mean response time, or:

$$E[r] = \frac{1/\mu}{1 - \rho} = 0.19ms$$

In order to compare how the model handles variation, we calculate the 90th percentile of response time using $2.3 \times E[r] = 10.58ms$. In reality, the 90th percentile is quite similar - $11.67ms$. Just like when predicting the mean number of jobs in service, the M/M/1 model is far off the true value. However, it gives a close approximation of the variation.

As a conclusion, an M/M/1 model of the middleware provides a very high level of abstraction, which gives incorrect results of the actual performance. This is the case even if the result for mean service rate per server are normalized, as done at the end of section 1.1 in order to reflect some of the internals of the middleware.

2 Analysis of System Based on Scalability Data

2.1 General Model

First we describe how a general M/M/m queue can be used in order to provide a model of the middleware architecture. Similarly, to the M/M/1 queue from the previous task, this an abstraction of the middleware that views it as a single queue. However, now the model acknowledges the fact that there are m servers which process requests from the queue. The general architecture can be seen in Figure 1.

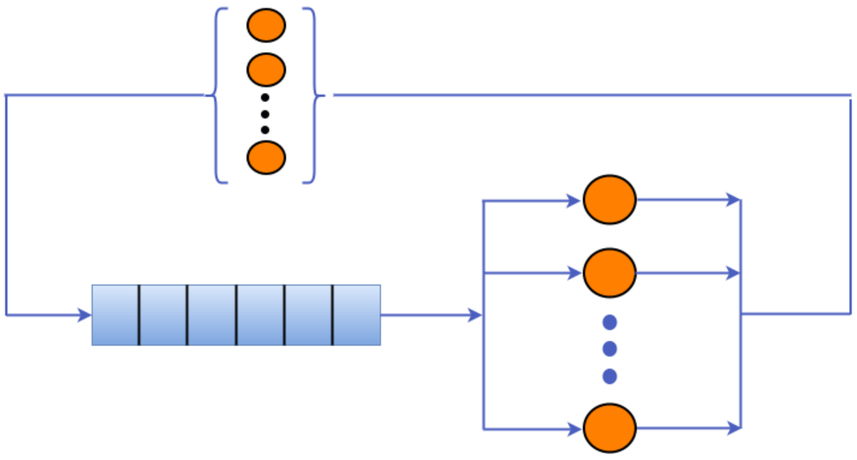

Figure 1: General M/M/m queue architecture

Unfortunately, it is not possible to build a more granular M/M/m model of the middleware because the internal queues do not meet the specification of an M/M/m queue. Even with the replication where each set request is sent to several servers, that is not the case since all backends process the same request, instead of each one working on a different request taken from the queue. Thus, each QueueManager will still be an M/M/1 queue.

In order to study the scalability of the middleware, we will compare the behavior of the model to that of the real world performance observed in Milestone 2. Here we consider the same factors that were used then, but will fix the value of replication to *full* since that is when the middleware exhibits more interesting behavior and will allow to study the effect of sets on the model in more detail. The levels for the servers will be 3, 5 and 7 since they correspond to three different specializations of the general architecture in Figure 1. Finally, the two extreme levels of the write ratio factor - 1% and 10% writes, will be used to study how the model will predict the scalability of the middleware. Thus, this section considers 6 different models.

2.2 Initial Model Results

The first thing to do for each model is to calculate the **traffic intensity** ρ. In order to find this, we have to again first compute the **mean arrival rate** λ and **mean service rate** per server μ.

λ can again be calculated from the memaslap logs collected, when running the experiment in Milestone 2 (*write_parsed_memaslap_All.log*) by finding the mean TPS for each configuration.

Calculating μ involves finding the mean service time $E[s]$ of a request, per configuration. Service time s is again assumed to be the time between taking off a request from a queue and enqueuing the response for sending back to the client. The results from the microbenchmark are stored in *write_timestamps_bench.tar.gz* and *write_parsed_timestamps_bench_All.log* and are used to find $T_{enqueue\_return}$ and add it to the request times logged in the original experiment. To find μ for each request type, we apply $\mu = 1/E[s]$, but before that we have to get a correct value for $E[s]$, which takes into account the read-write ratio and that for get requests, there are actually several queues where GETs are stored, and thus processed simultaneously.

From the logs *write_parsed_metrics_bench.log*, we should notice two things for the behavior of GET threads. First of all, for all configurations, the average number of requests waiting in a queue is larger than 0. Secondly, all threads in the middleware have similar load throughout the execution of the program, i.e no thread is doing less work than others. Because of this, it is safe to make the assumption that the number of GET requests that are handled in parallel is equal to the number of GET threads operating in the middleware, thus *active_threads* is calculated as 16. This, of course, is an approximation since there are minor differences, but gives a good view of the model behavior. The same log also provides information about the average number of set requests executing on the server in parallel. Therefore μ gets calculated as $\mu = reads * \mu get + (1 - reads) * \mu set$, where $\mu get = E[s]get/active\_threads$ and $\mu set = E[s]set/sets\_in\_service$. The final formula for calculating the traffic intensity ρ for an M/M/m model is:

$$\rho = \frac{\lambda}{m\mu}$$

This is a more accurate representation of the middleware architecture, than the M/M/1 model as it considers the actual number of backends. The traffic intensity for each model is visualized in Table 1. It also shows the values of response time before fitting them to the M/M/m model using the parallelization data.

	3 servers 1% writes	3 servers 10% writes	5 servers 1% writes	5 servers 10% writes	7 servers 1% writes	7 servers 10% writes
E[s]get (ms)	2.816	3.141	2.966	3.816	3.234	4.355
E[s]set (ms)	5.854	8.882	6.44	9.657	7.346	9.818
sets in service	0.978	3.76	0.986	2.748	0.996	2.449
ρ	0.8486	0.9585	0.4962	0.6514	0.3698	0.5076

Table 1: Core M/M/m model data

There are a couple of important conclusions and notes on scalability that can be made from this Table, before even going into details of the model. First of all, we see that for a certain number of backends ρ increases with the increase in writes. The reason for this is that as seen in Milestone 2 (Figure 4 and 6), SET requests are more time consuming, and increasing their number, will result in an increase of the overall response time of the middleware. This is captured by the growth of the traffic intensity.

Second and more important is the fact that ρ decreases as we add more servers. This is counterintuitive, as we saw in Milestone 2 that the performance of the middleware drops as we move to configurations with more servers (Figure 4, 6, 7). However, lets revisit the way Service time is computed - it involves $T_{processtime} + T_{server} + T_{enqueue\_return}$ but does not include the part of the middleware which takes the most time - $T_{enqueue\_return}$. Therefore, the model tells us that adding more servers will result in better scalability from the point of view of traffic intensity. Another conclusion we can make, from the model behavior is that none of the 3 timestamps involved in the Service time are the bottleneck of the middleware. This, however, will be further examined in the Network of queues models.

2.3 Further Model Analysis

We now proceed with the analysis of several metrics which are produced by the M/M/m model, but for that we have to calculate two necessary probabilities for each model: Probability of zero jobs in the system p_o and Probability of queuing ϱ:

$$p_0 = \left[1 + \frac{(m\rho)^m}{m!(1-\rho)} + \sum_{n=1}^{m-1} \frac{(m\rho)^n}{n!}\right]^{-1}$$

$$\varrho = P(\geq \text{m jobs}) = \frac{(m\rho)^m}{m!(1-\rho)} p_0$$

After calculating those values for each model, can proceed to examine how the model will predict the scalability of several metrics and compare that to the real results obtained in Milestone 2. The calculations for all models considered here are shown in Table 2:

	3 servers 1% write	3 servers 10% write	5 servers 1% write	5 servers 10% write	7 servers 1% write	7 servers 10% write
p_0	0.04	0.01	0.081	0.035	0.075	0.028
ϱ	0.728	0.923	0.127	0.305	0.018	0.081
E[n] (model)	8.404654643	26.01446268	4.296797878	5.774800409	4.146300263	5.47476344
E[n] (actual)	118	109	116	122	140	145
E[n_q] (model)	5.858995848	23.13913466	1.815811133	2.517703978	1.557951522	1.921699784
E[n_q] (actual)	67	51	31	31	21	18
E[r] (model)	0.4932883438	1.93539426	0.209756053	0.3294169473	0.2186322996	0.327587804
E[r] (actual)	15.70152951	16.99245811	15.96124354	17.48060997	16.72780812	18.41151345
E[w] (model)	0.3037181761	1.7049747654	0.010071819	0.0490017438	0.00090725284	0.0075598296
E[w] (actual)	2.303893608	2.382295491	0.8996637125	1.11634673	0.7281979329	1.330341576

Table 2: Core M/M/m model data

We will first examine the scalability of the middleware in terms of **mean number of jobs in the system** and **mean number of jobs in the queue**. The actual values, observed during the experiment in Milestone 2 can be calculated from *write_parsed_metrics_bench.log*, which shows how many requests on average are in each queue and how many requests are in service for a given SET queue.

In order to compare the actual middleware values, next we compute the results as given by the model, using the formula:

$$E[n] = m\rho + \rho^{\varrho}/1 - \rho$$

The predictions from the model, however, show a much lower number of jobs. This is because the model cannot predict correctly neither the jobs in the queue, nor the jobs in service. The reason for that is it considers only a single queue, where all requests are stored, whereas in reality there is more than one such queue. This is especially true for the GET queue, since we have seen from the logs that the asynchronous SET queues are almost always empty, even with the increase of the number of writes.

The table shows that in reality, the middleware has most of its jobs in service rather than waiting in a queue. That is mainly due to the parallelism of get requests in the actual architecture. The M/M/m model on the other hand, considers only some of the actual parallelism of the middleware. It acknowledges that we have multiple servers, but not the facts that there are several threads serving each GET queue and that the SET thread is asynchronous. These two factors are a significant cause of the larger number of jobs in the system as opposed to what is predicted by the M/M/m models.

Although this is a more accurate way of representing the middleware, than an M/M/1 queue, it still does not consider the most time consuming part of the middleware. This leads to making the model believe that requests will be served much quicker than in reality and thus have less requests waiting for service, since from its point of view they will be taken off quicker.

Next lets examine how our M/M/m model will handle the **Mean response time** $E[r]$ and **Mean waiting time** $E[w]$. First the $E[w]$ as observed in Milestone 2 is extracted from the logs. Important to note is that the response time depends on what we consider to be the boundary of the system. For the purpose of this analysis $E[r]$ will be the total time which the request spend in the middleware, which does not include the time it takes for memaslap to receive the result. Therefore, the actual results are taken from *write_parsed_timestamps_All.log.* For Waiting time we will consider not only T_{queue}, but the time to parse a request, T_{parse}, as well.

We can use the M/M/m model formulas to find $E[r]$, $E[w]$ as well as its 75th percentile, since this is the variation threshold that was used Milestone 2.

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

$$E[w] = E[n_q]/\lambda = \frac{\varrho}{m\mu(1-\rho)}$$

First of all, if we look at the predictions for $E[r]$ from the model, we will notice that it will suggest that the middleware will scale in the opposite way to what was observed in Milestone 2, when backends are increased. Again, the model does not consider that T_{return} which was shown in Milestone 2 to grow with the increase in servers and therefore the model deduces that adding more servers will spread the load and increase performance.

The actual values are also not close enough to the real times observed in Figure 14 from Milestone 2. The reason for this is a little more subtle. When we calculated the service rate, the result was taken to be the μ of the entire middleware. In other words the model assumes that its only queue is capable of serving that many requests per unit of time, while in reality the total service rate is achieved because of many smaller queues serving requests at some μ which is less than the total one. As a result, the model sees its queue as a very efficient in handling requests, which results in a predicted response time which is smaller than the actual one, by a factor of 50.

Finally, just as expected from the previous model results, $E[w]$ also decreases when adding more servers since this metric directly connected to the behavior of $E[r]$.

3 System as Network of Queues

3.1 General Model

Again lets first describe the network architecture, shown in Figure 2, which will be explored in this section. During Milestone 2, it became apparent that the Middleware consists of two major parts - processing requests and sending the responses back to clients. As explained in Milestone 1, each *QueueManager* manages one GET and one SET queue, in which client requests are stored. The GET queue is handled by n synchronous threads which take requests off the queue and send them to the servers. In order to consider this parallelism, we model each GET queue as an M/M/m queue, where m is the number of threads working on that queue. The SET queue achieves parallelism through the asynchronous thread which takes requests off the queue, without waiting to receive all responses for the previous one. Therefore, each SET queue is modeled a single M/M/1 queue. Replication does not need to be taken into account for the model, since from the point of view of the middleware, there will be an increase in T_{server} of SETs. Finally, the number of *QueueManagers* in a model, will be equal to the number of backends, since this is the core architecture of the middleware application.

Unlike the previous two models, this time we acknowledge the existence of the queue which stores request responses until they are sent back to the client. This is modeled as a single queue which receives all types of requests from all processing threads that are ready to send a response to a client. Thus, we model it as a single M/M/1 queue, since there is only one thread which takes off responses and handles them sequentially.

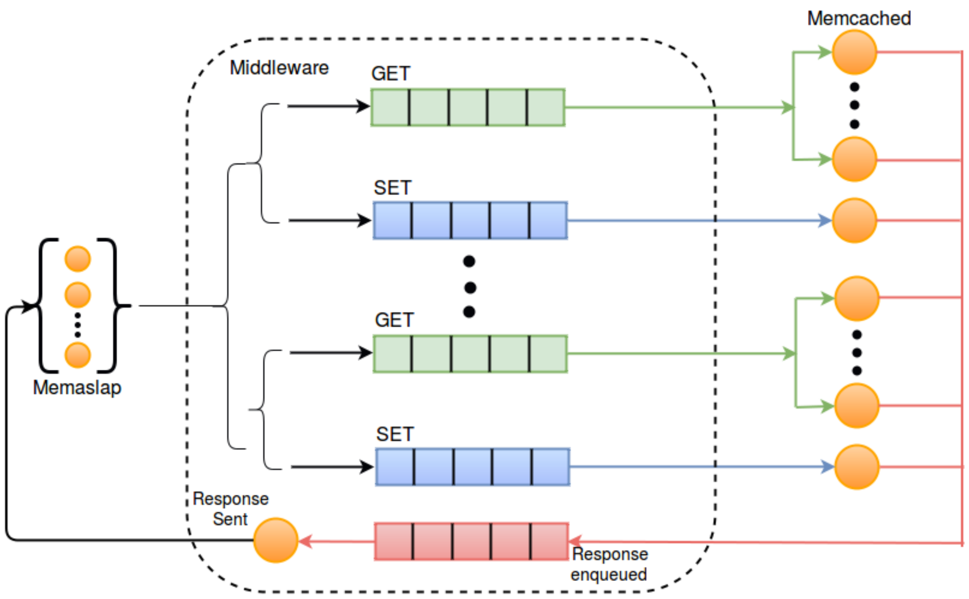

Figure 2: Queue Network Architecture

In this study we will fix the replication to full and write percentage to 10%. The varied parameter will be the number of backends, with levels of 3 and 7 servers since the 2k factorial design tells us that this is the factor with biggest impact on performance. These two values will also allow to examine the edge cases and compare the network of queues to the M/M/m queue from the previous section.

3.2 Model Analysis

Unlike the previous two models that we built, this time we have a queue hierarchy. In such queuing networks it is wrong to assume that the arrival rate for each queue is a Poisson process.

Because of this we cannot apply the formulas, we used to analyze the models from the previous sections. However, we can still analyze this model, by breaking it up into separate queues and using the Operational laws to calculate the necessary metrics.

For modeling purposes, we can see each queue as a device. The **GET queues**, which are presented here as n M/M/m queues, where n is the number of backends and m - the number of threads serving each queue, can be considered as *load-dependent service centers*. This is because the service rate depends on the number of jobs in the device. In particular, for M/M/m queues, the service rate should increase as we add more servers. The **SET** and **Response queues** are both modeled as M/M/1 queues, which makes them *load-independent service centers*.

For the overall analysis of the system, we will use Mean Value Analysis. This is done by first implementing the MVA algorithm (*mva.py*) and applying it separately for each type of device that we have in the queuing network. The inputs to the algorithm are the number of clients, think time, number of devices, service time per visit to the i_{th} device and visit ratio at the i_{th} device. Think time is calculated using the Interactive Response Time Law in the same way as in the next section. Si is computed the same way as in the previous section - dividing times by active threads (gets) and requests in service (sets). The original values before normalization are the same as in Table 1. Normalization is again done in order to take the parallelism into consideration. Finally, V_i is computed under the assumption that requests have uniform distribution over the same type of queue. The numbers come from the same log files as in the previous section. The final inputs of the new metrics are presented in the table below:

	N	Z	Vi gets	Vi sets	Vi resp_queue
3 servers	360	1.132 (ms)	0.297	0.04	1
7 servers	360	0.925 (ms)	0.1415	0.01	1

Table 3: Input to MVA algorithm

The three main operational laws that are used as part of the MVA algorithm are **Little's Law**, which allows us to relate the mean number of jobs in any part of the system with time spent at that part with the formula $Q_i = \lambda * R_i$, where Q_i is the total number of request in the device and not only in the queue. The other important law is **General Response Time Law**, which we use to estimate the total time jobs spend at a server, by multiplying the time per visit and visits to the server:

$$R = \sum_{i=1}^{M} R_i V_i$$

Finally, the Utilization of a device is computed using a combination of the **Forced Flow Law** and **Utilization Law** to give us the utilization of the i_{th} device: $U_i = XV_iS_i$.

The results for the models we examine in this section are presented in Table 4:

	E[n] (real)	E[n] (model)	E[r] (real)	E[r] (model)	Xi	Utilization
GET 3 servers	98.58	6.463849327	3.14130003	1.1061462	2496.89	82%
SET 3 servers	10.69	1.226432656	8.8823256	1.3831021	5.87	56%
Response q 3 servers	816.32	359.6	10.96568	3.944126	162.26	64%
GET 7 serv	129.85	8.766321445	4.35514192	2.9125881	1116.15	46%
SET 7 serv	15.11	1.584657332	9.81818348	3.0215388	420.16	13%
Response q 7 serv	902.43	392.4	13.2384103	5.6974712	156.21	89%

Table 4: MVA Results

3.3 Result Analysis

The first thing to notice is that the queuing network model built here predicts metrics a little better than the M/M/m, but is still off from the true values. Possible causes of this are discussed at the end of this subsection. Lets first examine what the model results do tell us.

First for all, this model actually differentiates between the two types of operations in the middleware, by distinguishing between devices that handle them. That is why we can see separate metrics (Jobs in device and Response time of device) for each of GETs and SETs. The model is successful in showing that there are many more GET request at a certain time than SETs. This is mainly due to the fact that he writes account for only 10% of all requests in the system. The response time per device is not predicted accurately, either, but it still captures the fact that SETs take longer to get processed, especially since we are modeling full replication.

What I consider to be the biggest strength of this model, is the fact that it captures the behavior of the response queue. We can see that the results correctly predict that the larger number of requests at any given time, in the whole middleware, are actually responses that are waiting to be sent back to the client. This finally justifies the conclusions that were made in Milestone 2. However, to expand on that, we need to examine each device's utilization. This is relevant because the device with the highest utilization is the middleware's bottleneck.

In the case of 3 servers we notice something that was initially not expected. The GET queues are the ones with highest utilization and not the response queue. However, the reason for this is that in this model, there are only 3 queues that have to handle all get requests. As we have seen from the logs as well, this is also when the number of requests waiting in the queue is highest. Therefore, it is actually normal that when there is a small number of backends, the bottleneck will be the device that has to deal with most of the requests.

However, when we increase the number of backends, the middleware will do more load balancing, which is also the main purpose of the program. This means that because of the consistent hashing algorithm, each GET queue will handle less load and potentially have more resources free. Therefore, we witness the behavior that was suggested in Milestone 2 - that the bottleneck device becomes the response queue. This makes sense since this is a simple M/M/1 queue, through which each request that enters the middleware goes. Although its service time is low since it only has to sent the response to the client, jobs are still handled in a sequential manner. The conclusion from this study is that if the middleware is to be implemented again or upgraded, the first thing to do to improve performance is have each GET or SET thread send responses to client directly, rather than going through a single queue. This can have an effect on the rate at which requests are taken off the queue, but will certainly aid performance.

Finally to close off this chapter we discuss, why the model results are not entirely correct. First of all, modeling a GET queue as a M/M/m queue, means that there will be m servers that handle requests. However, in reality the middleware has 1 server per queue, which handles requests that threads send, sequentially. This is a reason why the model underestimates the response time of GETs. Furthermore, throughout the whole analysis we have assumed that all threads work at full capacity, during the entire time. However, in reality that is not true, since there are context switches and threads may not get equal CPU time or one thread may handle less requests if it happens that it often has to wait longer for a response from memcached. Secondly, the results show that simply dividing $E[s]$ by the average number of requests in service is not the best way to model an asynchronous thread. Finally, this model ignores all places where a request has to go over the network. We have already seen that performance of the middleware also depends on the network itself, since it is one of the places that introduces most overhead. Therefore, the network between clients and middleware and middleware and servers can potentially also be modeled as a queue. The conclusion of Section 1, 2 and 3 is that modeling and applying queuing theory is a crucial part of performance analysis, but doing that is not trivial and must be implemented carefully, in order to produce practical information about the system under test.

4 Factorial Experiment

4.1 Setup

The analysis for this part will be done on the experiment, which explored the write effect on performance (Section 3, Milestone 2). This is originally a 3-factor experiment, but for the purpose of this section we will fix the replication factor. The value which it will take is "full replication", because we have already seen that we can make more in depth analysis of the middleware, rather than if it uses no replication.

The two factor considered in this analysis are the number of servers and the write ratio. In order to cover the whole Milestone 2 experiment, the levels which will be chosen for both are the minimum and maximum - **3** servers vs **7** servers and **1%** writes vs **10%** writes. This is also made possible because both factors are unidirectional as seen from Milestone 2. The log file, which contains the data that is used here is *write_parsed_memaslap_All.log.* From it we compute the mean TPS from all level combinations:

Writes	Servers 3	Servers 7
1%	13428	11888
10%	12478	11102

The final thing to mention before beginning the analysis is that the purpose of the 2k factorial design is to determine the effect, which the k factors have on some predefined performance metric. In our case, we will examine how number of servers and write ratio affect the Throughput of the middleware. Here we take that notion further and will do a 2k factorial design with replication in order to estimate the experimental errors and isolate their effect on the weight of the factors. For this we use the fact that all experiments in Milestone 2 were done with replication = 3. The analysis is done using additive model, since the multiplicative is not necessary, because the differences of factor levels are not very big.

4.2 Computation of Effects

The first step of the analysis is to compute the effects of the factors via the sign table method. To do this, we define two variables X_A and X_B, where

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

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

With this in mind we compute the following sign table:

I	**A**	**B**	**AB**	**y**	**Mean y**
1	-1	-1	1	(13584, 13140, 13560)	13428
1	1	-1	-1	(11860, 11644, 12160)	11888
1	-1	1	-1	(12742, 12622, 12070)	12478
1	1	1	1	(10986, 11296, 11024)	11102
48896	-2916	-1736	164		Total
12224	-729	-434	41		Total/4

where y is the TPS from all 3 replications and is taken from *write_memaslap.tar.gz* from Milestone 2. The computed results give us the following nonlinear regression model for the 2kr design:

$$y = 12224 - 729x_A - 434x_B + 41x_Ax_B + e$$

which is interpreted as follows: the mean performance is 12224 TPS, the effect of servers is -729 TPS, the effect of writes is -434 TPS; and the interaction between servers and writes is responsible for 41 TPS.

4.3 Experimental Errors

Because of the 2kr design we are able to approximate the experimental errors. This is done so that, we get more precise numbers for the factor weights, by ignoring the estimated errors. The goal of this subsection is to find the sum of the squared errors (SSE) and use it later to estimate the variance of the errors.

The response for any factor values is computed with the following formula:

$$\hat{y}_i = q_0 + q_Ax_{Ai} + q_Bx_{Bi} + q_{AB}x_{Ai}x_{Bi}$$

which is then used to find the experimental error by subtracting the estimated response from the measured value y_{ij}:

$$e_{ij} = y_{ij} - \hat{y}_i$$

The findings are presented in the table below:

	Effect				Estimated Response	Measured Response			Errors		
	I	**A**	**B**	**AB**							
i	12224	-729	-434	41	$\hat{y}_i$	y_{i1}	y_{i2}	y_{i3}	e_{i1}	y_{e2}	y_{e3}
1	1	-1	-1	1	13428	13584	13140	13560	156	-288	132
2	1	1	-1	-1	11888	11860	11644	12160	-28	-244	272
3	1	-1	1	-1	12478	12742	12622	12070	264	144	-408
4	1	1	1	1	11102	10986	11296	11024	-116	194	-78

The sum of all errors is 0, which is compulsory for correctness so we can now find SSE using the following formula,:

$$SSE = \sum_{i=1}^{2^2}\sum_{j=1}^{r} e_{ij}^2 = 573080$$

4.4 Variation and Factor Weights

Now, when deciding the impact of each factor we can isolate the experimental errors from the calculations. The Total Sum of Squares can now be divided into four parts - one for the variation of each factor, one for the variation of their interaction and one for the errors:

$$\sum_{i,j}(y_{ij} - \overline{y}_{..})^2 = 2^2rq_A^2 + 2^2rq_B^2 + 2^2rq_{AB}^2 + \sum_{i,j} e_{ij}^2$$
$$SST = SSA + SSB + SSAB + SSE$$

We now compute the variation of each part:

$$SSA = 2^2 r q_A^2 = 6377292$$
$$SSB = 2^2 r q_B^2 = 2260272$$
$$SSAB = 2^2 r q_{AB}^2 = 20172$$
$$SSE = 573080$$
$$SST = 9230816$$

And finally we compute how much of the variation is explained by each factor, by dividing it by SST. The results we get are:

$$SSA/SST = 69.09\%$$
$$SSB/SST = 24.49\%$$
$$SSAB/SST = 0.22\%$$
$$SSE/SST = 6.2\%$$

The conclusion is that the number of backends has the biggest effect on the Throughput of the middleware. The detailed description of why and how the number of backends affect the middleware performance has been discussed in the last section of Milestone 2. The main points are that first of all, having more backends results in a higher number of requests being sent over the network, in the case of full replication. For example having 7 servers, will cause a thread to wait for 7 responses, rather than just 3 when, we have 3 backends. Secondly, the number of servers has an impact on T_{return}, which is the most time-consuming part of the middleware. Thus, increasing the servers, will have an even bigger effect on the middleware performance, since that will increase the time requests spend in its slowest part.

Now it is confirmed that the write ratio also has a noticeable effect and therefore it should always be considered when analyzing the performance of the middleware. This was already seen in the relevant section of Milestone 2, but now we know that this parameter alone accounts for 1/4 of the TPS variation. The reason for its effect is obvious. With full replication, SET requests take more time, and thus increasing their number will only increase the overall response time of the middleware.

The interaction between the two factors, however, has a negligible effect and can be considered unimportant in practice.

5 Interactive Law Verification

The experiment which will be checked against the Interactive Response Time Law is the one, which studied the effect of writes. As a reminder, it considered 3 factors - number of server, replication and write ratio, with 3 levels each. In this study we will fix the replication to full, which will leave us with 9 configurations to explore. The law states that the Response time is equal to the number of users divided by the throughput minus the think time of each client. Thus the formula, which we will use to compute the thinking time of a client is:

$$Z = (N/X) - R$$

where the thinking time Z is the time it takes for a client to send a new request after receiving a response for the last one. The number of clients that was used during this experiment was $N = 360$. The results are presented in Table 5

Servers	Write ratio	R (ms)	N	X	Z (ms)
3	1%	26.2615	360	13428.5833333	**0.546986871975**
3	5%	26.942	360	13088.7	**0.562641408238**
3	10%	27.717	360	12478.6666667	**1.13223602941**
5	1%	28.261	360	12424.55	**0.713892450833**
5	5%	28.393	360	12313.47	**0.843275395969**
5	10%	29.9635	360	11615.2633333	**1.0302011043**
7	1%	29.217	360	11888.1533333	**1.06524736903**
7	5%	30.5355	360	11471.2633333	**0.847268361263**
7	10%	31.5005	360	11102.3533333	**0.925057824775**

Table 5: Interactive Response Time Law Results

As we have seen from Milestone 2, the throughput of the middleware goes down with the increase in servers and number of writes - data sources can be found in *write_parsed_memaslap_All.log* from Milestone 2. Naturally, the response time also increases because of the larger number of SET requests which are more computationally expensive when the middleware is operating with full replication.

The same pattern in the behavior is observed for the thinking time Z of a client. Z increases with the increase of backends and it also increases as the write percentages become bigger.

Although the results for Z seem reasonable, we have to keep in mind that they may not be 100% correct. The reason is that the calculations depend on measurements taken by memaslap and these may have some statistical error involved. Remember that each memaslap instance runs on one machine which simulates 120 clients running at the same time. Each second memaslap takes measurements for each of these clients and averages out the result, to give data for the given period. Because of that heavy load, there may be some inaccuracies in the timings. Moreover, at some point taking the average may not be the best option because of the variances in network load. However, as mentioned, the results presented above validate the Interactive Response Time Law, but is still good be wary of such events.

A Benchmarks and Timestamps

In order to get more precise data for the analysis done in this report a few benchmarks were done. One of the reasons for that was to get more information about the state of different queues - number of requests waiting in a queue, requests in service and total number of requests handled. More importantly, they provided more precise data on one of the timestamps introduced in Milestone 1 - T_{return}. It was realized that this action is not atomic and there are actually two parts to it - time until request is enqueued and time the request spends in the return queue. These are referred to as $T_{enqueue\_return}$ and T_{return_queue} respectively throughout this report.