Advanced Systems Lab (Fall'16) – Third Milestone

Name: *João Lourenço Ribeiro*
Legi number: *15-927-999*

Grading

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

1 System as One Unit

The goal of this section is to build an M/M/1 model of our system using the data from the stability trace. Since I was lacking the middleware logs from the stability trace of Milestone 1, I re-ran it. The raw memaslap and middleware logs, and the processed data are in `stab-trace`. In order to model the system, I consider the client-middleware network, middleware receiver thread and middleware read/write queues jointly as a black box which is mapped to the "queue" of the queuing model. This is valid because client think time is negligible, and so one can think of requests first traversing the whole client-middleware network (ignoring the clients in the middle) upon arrival to the system. Furthermore, due to the behavior of memaslap, arrival rate and distribution coincide at the beginning and end of the client-middleware network. Finally, the way of estimating service rate below is affected by the network and so it should be part of the system. The read/write threads, memcached servers and associated middleware-server networks are "merged" into a single server and correspond to the "server" of the queuing model. A request is deemed to have been serviced when the middleware sends the response back to the client, i.e. when it writes the response into the buffer of the corresponding client socket. Therefore, the response time in the model is mapped to the memaslap response time. The service time corresponds exactly to the time interval elapsed between the middleware dequeuing a request and sending its response to the client, $T_{\text{end}} - T_{\text{dequeue}}$. The waiting time in the model corresponds to the difference of these two times.

Due to the behavior of memaslap (a new request is sent only when the previous request finishes being processed), the number of requests that arrive in a given time period is exactly the number of requests completed in that period, i.e. the throughput. Thus, I estimate λ as the average throughput over the period measured (between 300 seconds and 3900 seconds) in the stability trace and obtain $\lambda = 16134$ requests/sec. In order to estimate μ I proceed as follows. If the system exhibits a given throughput over a period of 1 second at some point during the trace, then one can infer that the system has the capacity to process at least that many requests offered by memaslap over a decently sized time interval in a stable manner. Therefore, the maximum throughput achieved by the system in a 1 second period over the whole stability trace is a decent estimate of the service rate μ (due to the high number of available 1 second samples too), and also a lower bound for the real service rate. This means that the estimate for ρ is an upper bound on its real value. Therefore, if this estimate is less than 1 then so is the real ρ, and so one infers that the system is stable, as expected from the previous Milestones. The maximum throughput achieved over the 300s-3900s period of the trace was 18885 requests in a second, so I set $\mu = 18885$ requests/sec. This yields a traffic intensity $\rho = \lambda/\mu = 0.854$.

Tables 1, 2 and 3 showcase, for several system metrics, the comparison between the model predictions and the observed values. In order to accurately estimate the average number of jobs in the system, waiting, and being serviced, I proceed as follows. Due to the behavior of memaslap (a client sends a request as soon as he receives a response from the previous one) and since think time for clients is very small (see Table 11 in Section 5), I estimate the number of jobs in the system as $3 \times 64 = 192$ jobs, as there are 64 clients per memaslap instance. To estimate the number of jobs being serviced, I proceed as follows. When a request is dequeued by a thread from a queue in the middleware, the time it spends being moved from the queue to the server and back to the client is negligible compared to the server time. Thus, I can assume as a very good approximation that whenever a thread is busy, the corresponding request is in the server. Therefore, the average number of busy read threads in the system is a very accurate estimate of the average number of jobs being serviced (since the write proportion is very small). In order to estimate the number of busy threads in the trace, I decided to run an experiment with the same setup of the stability trace for 2 minutes and with 1 repetition, in which I also log the number of busy read threads every time a request is logged.

The raw memaslap and middleware logs of this experiment can be found in `busy-31`. I take the numbers of busy threads logged in the 30s-90s period of the experiment and compute its average, which is approximately 44.14 threads. Finally, since the number of jobs waiting in queue is simply the difference between the total number of jobs in the system and the number of jobs being serviced, I estimate its average as $192 - 44.14 = 147.86$ jobs. The model predictions are computed according to the formulas found in Box 31.1 of the book *The Art of Computer Systems Performance Analysis*, by Raj Jain.

	Avg. resp. time (ms)	50th perc. (ms)	90th perc. (ms)	STDev (ms)
M/M/1 Model	0.363	0.252	0.837	0.363
Real value	12.077	–	–	10.446

Table 1: Comparison of resp. time statistics between prediction and observation. Real percentiles cannot be computed from available data.

	Avg. waiting time (ms)	50th perc. (ms)	90th perc. (ms)	STDev (ms)
M/M/1 Model	0.310	0.195	0.780	0.359
Real value	9.437	–	–	9.366

Table 2: Comparison of waiting time statistics between M/M/1 prediction and observations. Real percentiles cannot be computed from available data.

	Avg. serv. time (ms)	STDev serv. time (ms)	Mean jobs in server	Mean jobs in queue	Mean jobs in system
M/M/1 Model	0.053	0.054	0.854	4.995	5.849
Real value	2.640	4.626	44.14	147.86	192

Table 3: Comparison of serv. time and mean number of jobs between M/M/1 prediction and observations.

Note that every value predicted by the M/M/1 model is much smaller than the corresponding observation in the system. This can be explained by the fact that there is a high amount of parallelism in our system as follows. The service rate is influenced by the fact that the system actually processes several requests at the same time which take longer than in the model. When the M/M/1 model is given the high service rate, it interprets it instead as being a consequence of the fact that every request actually spends much less time being processed. Furthermore, the model does not distinguish between `get` and `set` requests, and the latter are more expensive, especially since they are replicated, which is another feature that the model does not capture.

Nevertheless, there are certain things that the model can predict fairly well, or that can be translated to close predictions of observed values. First, the model does predict that response time and waiting time follow tail distributions, as evidenced by the predicted percentiles and standard deviations (these are close to their associated averages) in Tables 1 and 2. This happens because the model assumes exponential service times, which then imply that response and waiting times follow a tail distribution. Note also that the predicted standard deviation (0.363 ms) for response and waiting time is equal to its average. The real standard deviation (10.446 ms) is also close to the real average response time (12.077 ms). This is again due to the fact that the predicted time distributions have a tail. Second, the proportion of requests in the queue in relation to total number of requests in the system is $4.995/5.849 = \rho = 0.854$, which is close to the observed proportion of $147.86/192 \approx 0.77$. This also holds with a bit less accuracy for the proportion of requests in the server in relation to total number of requests in the system, which is $0.854/5.849 = 0.146$, against the observed ratio $44.14/192 = 0.230$. These predictions are close due to the fact that the exponential distribution assumption for service time holds remarkably well. The predictions for the proportion of time spent in the queue/server are also quite close due to this same reason. Third, the average service time predicted by the model is model resp. time − model wait. time $= 0.363 - 0.310 = 0.053$ ms. Since I have, on average, 44.14 busy read threads servicing requests instead of just 1, it is reasonable to expect that the observed server time in our system, real resp. time − real wait. time $= 12.077 - 9.437 = 2.640$ ms, is close to $44.14 \times 0.053 = 2.339$ ms, which holds as expected within an error of about 11%.

2 Analysis of System Based on Scalability Data

The goal of this section is to build several M/M/m queuing models of the system in several different configurations considered in Milestone 2, compare them to the experimental data, and to analyze the real scalability of the system compared to the scalability of the models. In order to obtain more precise measurements of middleware times, I decided to re-run the experiments of Section 3 of Milestone 2. The raw memaslap and middleware logs, and the processed data for the experiment can be found in `m23-rerun`.

To map the system to the M/M/m model, I proceed as follows. The "queue" of the model corresponds to the client-middleware network, the main receiving thread and read/write queues of the middleware. This is valid because client think time is negligible, and so one can think of requests first traversing the whole client-middleware network (ignoring the clients in the middle) upon arrival to the system. Furthermore, due to the behavior of memaslap, arrival rate and distribution coincide at the beginning and end of the client-middleware network. Finally, the way of estimating service rate below is affected by the network and so it should be part of the system. Each of the "servers" of the model corresponds to a memcached server, along with its associated middleware-server network and read/write threads. Therefore, the parameter m corresponds exactly to the number of memcached servers in the system. A request is deemed to have been serviced when the relevant thread associated to the memcached server sends the response back to the client. Therefore, the response time in the model is mapped to the memaslap response time. The service time corresponds exactly to the time interval elapsed between the middleware dequeuing a request and sending its response to the client, $T_{\text{end}} - T_{\text{dequeue}}$. The waiting time in the model corresponds to the difference of these two times.

The reasons for choosing to model the system as in the previous paragraph are as follows. First, in the real system, each request is routed to one of m different servers, which have a single internal queue for processing requests inside. Thus, choosing m as the number of servers seems to be sensible and brings us one step closer to reality: now one is able to distinguish between servers. This model still markedly underestimates its predictions, mostly due to the fact that it still does not fully capture the parallelism of the system. Also, the model does not distinguish between `get` and `set` requests, nor can it handle replication. Nevertheless, since I am interested mainly in how the model scales compared to the real system (qualitative behavior), I find such differences acceptable as the model still showcases interesting trends when it scales, which can be explained based on both the model and system.

In order to analyze scalability, I vary number of servers, replication factor and write proportion. The reason is as follows: as evidenced in Milestone 2, these are the factors that have the greatest effect on the system's performance. The most interesting configurations to analyze and compare correspond to extreme cases of each of these factors: 3 and 7 servers, no replication and full replication, 1% and 10% writes, as these 8 configurations give the best picture of how the system scales, since they showcase the biggest differences in system performance possible. For every M/M/m model considered, I estimate λ as the average throughput of the system over all the repetitions for that particular configuration, for the reasons already explained in Section 1 (throughput tells us how many requests arrived in a given period). To estimate μ, I first compute the maximum throughput over all repetitions for the relevant configuration, again for reasons specified in Section 1 (it is an amount of requests that I know the system can process in a stable manner). Then, since the load in our system is distributed close to uniformly over the memcached servers, I set $\mu = \frac{\text{max. throughput}}{m}$, where m is the number of memcached servers. All predictions are computed based on the formulas of Box 31.2 of the book *The Art of Computer Systems Performance Analysis*, by Raj Jain. Table 4 summarizes the parameters of the M/M/m models associated to each configuration considered.

Looking at Table 4, one notices that utilization does not change much from configuration to configuration, even when going from 3 to 7 servers. In fact, this makes sense given the features

(serv.,rep., writes)	(3,1,1%)	(3,3,1%)	(7,1,1%)	(7,7,1%)	(3,1,10%)	(3,3,10%)	(7,1,10%)	(7,7,10%)
λ (req/sec)	16024	15973	13375	12953	15376	15174	13012	11121
μ (req/sec)	6515	6422	2284	2335	6354	6177	2399	2031
$\rho = \frac{\lambda}{m\mu}$	0.820	0.829	0.837	0.792	0.825	0.819	0.775	0.782

Table 4: Comparison of arrival rate (λ), service rate per thread (μ), and traffic intensity (ρ) for configurations under consideration.

of our system. As seen in Milestone 2, when the number of servers increases, one deals with more CPU contention and more connections over the network, which make it more unstable. Therefore, while each server gets a lighter load of requests, these requests take longer to be moved from the queue to the memcached server by the read/write threads, and also take much longer to traverse the network between the middleware and the memcached servers. Therefore, each request has a higher service time, and so in general the servers are close to as busy as for 3 servers. As further evidence for this, note that the service rate (per server) drops significantly from 3 to 7 servers, due to the fact that throughput drops in general (and so maximum throughput also drops significantly). This is because memcached server/network/thread pool combos are much less efficient in processing requests when there are more servers. That utilization does not change much when one fixes the number of servers but varies the other factors can be explained as follows. The service rate is always taken as the maximum throughput, which is prone to some variation (much more than the average, for example). Thus, although the other factors have an effect on the performance of the system, it may not be significant enough to not be somewhat affected by the variation of the maximum throughput. As can be seen in Figures 2, 4 and 6, changing these configurations yields only relatively small changes in average times in reality also. Figures 1 through 6 showcase both predicted and observed response time, waiting time and service time for each of the configurations considered.

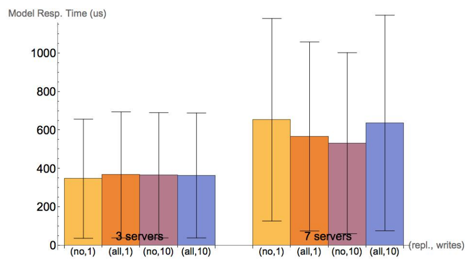

Figure 1: Response time predicted by the model.

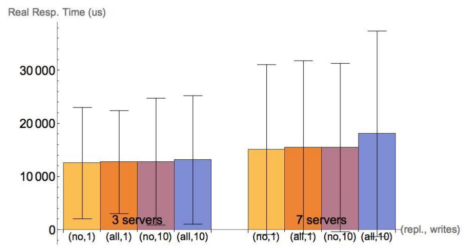

Figure 2: Observed response time.

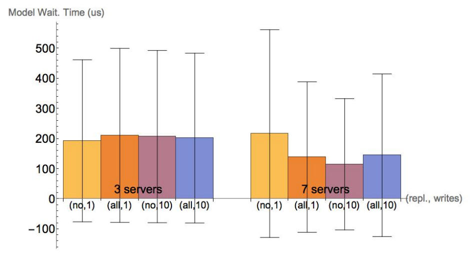

Figure 3: Waiting time predicted by the model.

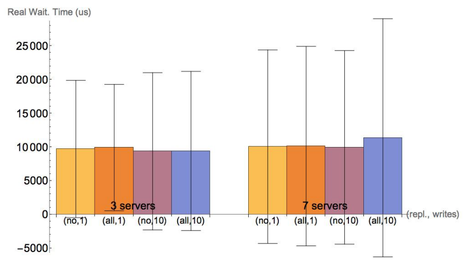

Figure 4: Observed waiting time.

Looking at Figures 1 through 6, one sees that, as already mentioned, the model severely underestimates the various times. This is again due to the fact that it does not model all the parallelism in the system (namely the threads in each pool separately). Thus, the model interprets a given service rate not as having several "slower" requests being processed at the same time (which is the case in the real system), but rather as being able to process just m requests at the same time, but much faster on average.

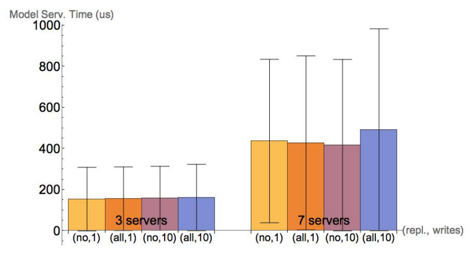

Figure 5: Service time predicted by the model.

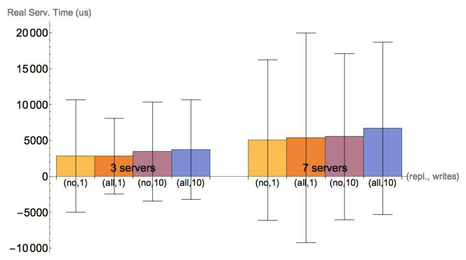

Figure 6: Observed service time.

In the following paragraphs, I identify and explain interesting differences/similarities in trends for predictions and observations. Note that the trends of both predicted and observed response/waiting/service time for 3 servers seem to match quite well, both for the average and standard deviation of each configuration. This is due to the fact that, for 3 servers, increasing replication factor and write proportion only affects the performance of the system slightly (markedly less than for 7 servers). In the real system, this means that performance drops only slightly, which can be seen by a small monotonic increase in the averages in Figure 2. This slight drop in performance means also that the estimates for service rate are also only slightly affected, and in a similar way to average throughput (which is the arrival rate). This explains why the predictions for 3 servers in 1, 3 and 5 are relatively constant and so match the observed trends quite well. The predictions for response and service time in Figures 1 and 5 do not follow a slight monotonically increasing trend like the real values simply because of unavoidable variations in the maximum throughput, as mentioned before. Standard deviations in both prediction and reality are close to their respective averages. This is so because the model assumes service time follows an exponential distribution, and so, in general, times follow certain tail distributions, which match quite well with the observed time distributions.

When going from 3 to 7 servers in response time (Figures 1 and 2), there is an increase in the average in prediction and observation, although the relative increase is sharper in the prediction. This increase is as expected, since system performance drops significantly from 3 to 7 servers, and thus so do service rate and arrival rate. The explanation for why the increase is sharper in the model is as follows. For 7 servers, the system is more unstable, mostly due to increased CPU contention and more connections over the network. The model, when faced with a significant (and similar) drop in arrival and service rates, considers a response time distribution with the same shape as before, but different parameters: thus, standard deviation stays relatively the same when compared to the corresponding average and so the average must grow. On the other hand, while this predicted distribution fits well for 3 servers, it fits worse for 7 servers, due to the fact that the network is much more unstable in this case and distorts the time distributions. This is evidenced by the fact that, while the average does not increase as much in the real system, the standard deviation increases more in relation to the average, so now the predicted standard deviation/average ratio underestimates the real standard deviation/average ratio.

Regarding waiting time (Figures 3 and 4), note that the average and standard deviation seem to decrease significantly from 3 to 7 servers, while the average increases slightly in reality (about 0.6ms increase), and standard deviation also increases. For 7 servers, real waiting time increases with replication factor and with write proportion when there is full replication. This discrepancy can be explained as follows. From 3 to 7 servers, service rate drops significantly, so service time increases sharply in the model (see Figure 5). Nevertheless, since there are more servers serving the single queue in the M/M/7 model, each request waits less time, even if each server is slower. This is not the case in the real system, since waiting time includes time spent in the network between middleware and clients, which becomes increasingly unstable for larger number of servers, and one has to deal with CPU contention everywhere. This is not modelled by the M/M/m models. This counteracts the fact that there are less requests per queue, coupled with the slower service time (due mainly to increased CPU contention and instability in the middleware-server network), so real waiting time increases slightly as expected.

For service time (Figures 5 and 6), both prediction and observation feature a sharp relative increase in service time from 3 to 7 servers, and the relative increase is sharper in the prediction. The ratio of standard deviation in relation to the average is also underestimated by the model. This holds because for 7 servers the network is more unstable overall and there is more CPU contention, and the model does not account for this: it assumes all servers work at 100% all the time, so each server must be much slower to account for the drop in service rate, while in reality less servers can work at the same time but have a higher service rate. That the predicted service time does not follow the same monotonically increasing trend as the observed one can be explained by slight variations in the maximum throughput in the experiments.

Table 5 showcases expected number of jobs in the servers/queue/system for both the model and the real system. I estimate the values in the table just like in Section 1 (I ignore the `set` requests, which behave a bit differently, for simplicity: the estimates are still relatively accurate since the model does not distinguish between them and the write proportion is not too large). To get the average number of busy read threads, I run a small benchmark for each configuration with its setup for 2 minutes and 1 repetition. Memaslap and middleware logs are in `busy-32`.

(serv.,rep., writes)	(3,1,1%)	(3,3,1%)	(7,1,1%)	(7,7,1%)	(3,1,10%)	(3,3,10%)	(7,1,10%)	(7,7,10%)
Model $E[n]$	5.55	5.86	8.75	7.34	5.74	5.52	6.92	7.09
Model $E[n_q]$	3.09	3.37	2.90	1.80	3.26	3.06	1.50	1.61
Model $E[n_s]$	2.46	2.49	5.85	5.54	2.48	2.46	5.42	5.48
Real $E[n]$	198	198	198	198	198	198	198	198
Real $E[n_q]$	154.99	154.98	142.28	139.90	159.00	157.08	144.85	145.43
Real $E[n_s]$	43.01	43.02	55.72	58.10	39.00	40.92	53.15	52.57
Real serv. time (ms)	2.857	2.834	5.077	5.393	3.479	3.750	5.542	6.710
$\frac{\text{\# busy threads}}{m}$ × pred. serv. time	2.208	2.237	3.479	3.552	2.054	2.210	3.166	3.695

Table 5: Comparison of expected jobs in system (n), in queue (n_q) and in server (n_s) for configurations under consideration. Comparison between real service time and "$\frac{\text{\# busy threads}}{m}$ × predicted service time" estimate.

Regarding Table 5, note first that, when one changes from 3 to 7 servers, there is a general decrease in the expected number of jobs in the queue ($E[n_q]$), and a general increase in the expected number of jobs in the server ($E[n_s]$) for both the model and the real system. This is natural, since this change implies that there are more servers both in the model and in the real system, and so there is less load per server. The changes are sharper for the model: in fact, the real system goes from about 44 busy threads out of 48 total, to about 53 busy threads out of 112 total, so the ratio of busy threads decreases. This is due to increased network instability and CPU contention: the other requests are not in the read/write queues so often (otherwise they would have been serviced already, since there are many free threads), but actually have to spend much more time to traverse the client-middleware network and to be moved by the main receiving thread. Also, in practice only a limited number of threads can actually run in parallel. In general, $E[n_q]$ has the same behavior as one changes replication factor and write proportion for both the model and the real system, for reasons already mentioned. This actually explains why $E[n_s]$ does not have the same behavior in the model and in the real system as follows. The real system is closed, while the model is open. Since the total number of requests is thus fixed in reality, this implies that, for a fixed number of servers, the behavior of $E[n_s]$ is exactly the opposite as that of $E[n_q]$, and thus also of the predicted behavior for $E[n_s]$. Finally, note that, just like it was explained in Section 1, one expects the real service time to be somewhat close to $\frac{\text{\# busy threads}}{m}$ × predicted service time, which holds especially well for configurations with small write proportion, while it has larger error for 10% writes. This increase in the error is mostly due to the fact that I do not take into account the `set` requests which are also in the server in my estimates for the number of requests being serviced at the same time: they are relevant especially when the write proportion is larger and there are more servers, which is a good explanation why the error increases in general as servers/write proportion increase.

3 System as Network of Queues

The goal of this section is to build a network of queues to model the system based on the modeling efforts of the previous Sections, compare it to experimental data, analyze its behavior, and identify potential bottlenecks. A diagram of the proposed network of queues can be found in Figure 7.

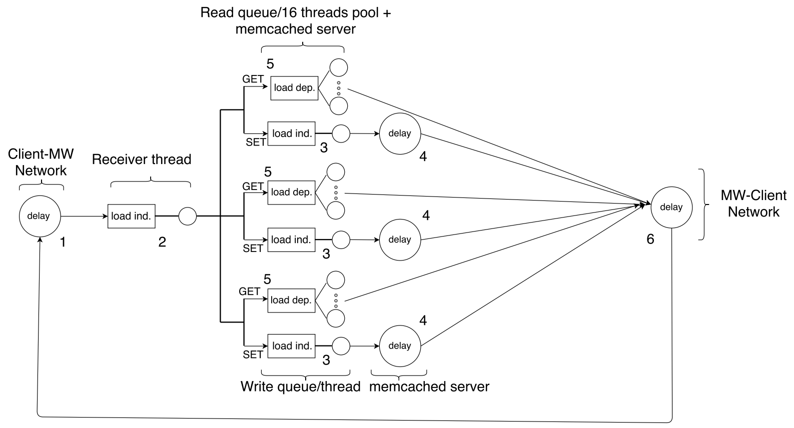

Figure 7: Diagram of proposed queuing network for configurations with 3 memcached servers. The extension to more servers follows easily by adding more branches with devices 3, 4 and 5. Devices with the same number have the same parameters and model the same part of the system, but are associated to different memcached servers. Arrows indexed by GET or SET mean that only requests of the indexed type arrive at the associated device. Non-indexed arrows mean that both request types may traverse them, when applicable. At each point where an arrow splits into several arrows, load is distributed uniformly for each request type between the available options.

The explanation for why I decided to model the system like in the diagram is as follows. First, I model the system as a closed network since that is a closer approximation to how the system works in practice: every virtual client has exactly one request in the system at all times (I include the client-middleware network in the system as well, and think time is negligible), so one can think that each request loops in the system. In the real system, each `get` or `set` request may switch class when it returns to the client to be sent again. For simplicity, in the model I consider that requests do not switch class. This should not affect the model much: the write proportion in the model is constant, while memaslap also uses class switching as a way to enforce that the write proportion is always very close to the fixed one.

A request starts by being sent from a client to the middleware via the client-middleware network. Since there is a separate connection over the network for each request in the system, the client-middleware network can be modeled as a delay center, i.e. with an infinite number of servers (device 1). Its service time can be estimated by taking the offset between memaslap reported average response time and the average total time spent in the middleware for a given class of requests. Since this offset concerns two traversals of the client-middleware network, I set the service time of device 1 for a given request class as half of this offset. This request is received in the middleware by the receiver thread, which routes it to the relevant read/write queue. Thus, the receiver thread can be naturally modeled as a load-independent device (device 2), and its service time is exactly the average time it takes the receiver thread from receiving the request until it is enqueued, i.e. $T_{\text{dequeue}} - T_{\text{start}}$. The request is then moved to a read/write queue depending on whether it is a `get`/`set` request. Each read queue has, in my case, 16 threads in a pool dequeuing from it. Therefore, the read queue and associated pool are modeled as a load-dependent device (device 5) with service rate function $\mu(j) = \frac{\min(16,j)}{s}$, where j is the number of requests in the device and s is the service time for one request in the system, which represents a device with a single queue and 16 servers dequeuing from it (see first page of chapter 36 of *The Art of Computer Systems Performance Analysis*, by Raj Jain). In the model, I also integrate the associated memcached server in device 5, in the sense that the service time for 1 request is exactly the average time taken by a read thread between dequeuing a `get` request and sending the response from the memcached server back to the client, $T_{\text{end}} - T_{\text{dequeue}}$, and so it includes the server time for `get` requests. This was done with the goal of being able to model the fact that read threads are synchronous. In order to capture the fact that write threads are asynchronous, the write queue, write thread, and server combination is modeled as a load-independent device (device 3) followed by a delay center (device 4). Since the write thread does not block after sending a `set` request to the server but instead spins, I set the service time of device 3 to the time a write thread takes to spin when it does not have to process any responses from the server, i.e. $T_{\text{send}} - T_{\text{dequeue}}$ of `set` requests. Device 4 models the memcached server, and its service time is set to the difference in time between the write thread sending a `set` request to the server, and then sending its response to the client, $T_{\text{end}} - T_{\text{send}}$. Finally, the request is sent back to the client through the middleware-client network (device 6), which is modeled exactly like the client-middleware network (device 1) and has the same service time.

In order to test the model against experimental data, I consider the experiment from Section 3 of Milestone 2, in particular the setup with 7 servers, full replication, and 10% writes. I chose this configuration because it showcases extreme behavior of the system, as it has to deal with many servers (and thus heavy CPU contention), full replication (encoded in the service time of device 4) and a large write proportion. The data is taken from the re-run of this experiment from Section 2 of this Milestone. The memaslap and middleware logs can be found in `m23-rerun`. The relevant service times for all devices for both `get` and `set` requests (where applicable) can be found in Table 6. This table also includes the visit ratios (expected number of visits to the device by a request during a traversal of the network) of each request type for every device in the network: these are computed assuming uniform load-balancing, which is a very good approximation since in my system load-balancing is very close to uniform.

Device number	1	2	3	4	5	6
`get` service time (ms)	3.732	0.005	–	–	6.179	3.732
`set` service time (ms)	3.783	0.010	0.001	11.499	–	3.783
`get` visit ratio	1	1	0	0	1/3	1
`set` visit ratio	1	1	1/3	1/3	0	1

Table 6: Service times and visit ratios of both classes (where applicable) for different devices (see Figure 7 for the device numbering). Service times and visit ratios are the same for all devices with the same number.

I apply mean value analysis (MVA) to the network, with 198 requests in total in the system: 178 `get` requests and 20 `set` requests. This is accomplished by using the MVA algorithm implemented in the JMT application[1]. Note that the MVA algorithm in JMT does not support multi-class networks when they contain load-dependent devices. Nevertheless, one can still obtain extremely close approximations of the multi-class network analysis through MVA by applying the MVA algorithm separately to the `get` requests, then to the `set` requests, and combining the predictions in a straightforward manner. This is so because both request types only interact in the receiver thread (device 2), which has an *extremely* small service time. Therefore, `set` and `get` behaviors are practically independent (in the network model), and so their networks can be analyzed separately without losing any observable accuracy in the predictions. The models input into JMT can be found in `jmt`. Table 7 showcases the comparison between predicted and observed times for both `get` and `set` requests, along with the predicted utilization and expected number of requests for each device in particular.

Observing Table 7, there are several interesting conclusions to be made. First, the predicted `get` total response time is a bit off from the real one with a relative error of about 21%. As a direct consequence, predicted `get` TPS is also a bit off from the real `get` TPS with a relative error of about 29%. The model predicts a lower `get` response time than the one observed for two reasons. First, the seven devices 5 assume an exponential service time distribution which most

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

Device number	Total	1	2	3	4	5	6
Observed `get` time (ms)	17.526	3.732	0.005	–	–	9.995	3.732
Model `get` time (ms)	13.801	3.732	0.005	–	–	6.328	3.732
Observed `set` time (ms)	23.085	3.783	0.010	3.972	11.499	–	3.783
Model `set` time (ms)	19.076	3.783	0.010	0.000	11.499	–	3.783
Observed `get` TPS (ops/sec)	10015.9	10015.9	10015.9	0	0	1430.8	10015.9
Model `get` TPS (ops/sec)	12897.8	12897.8	12897.8	0	0	1842.5	12897.8
Observed `set` TPS (ops/sec)	1112.9	1112.9	1112.9	159.0	159.0	0	1112.9
Model `set` TPS (ops/sec)	1048.4	1048.4	1048.4	149.8	149.8	0	1048.4
Model # requests	198	52.101	0.084	0.0002	1.722	11.665	52.101
Utilization	–	52.101	0.080	0.0002	1.722	0.9999	52.101

Table 7: Comparison between real and predicted values for throughput, time spent in device, utilization and expected number of jobs for each device labeled with a given number (see Figure 7 for the device numbering). The utilization of a delay center is the expected number of requests in that device.

likely features a lighter tail than the real service time distribution. Second, the model assumes that all 16 servers are always working at 100% in parallel, which does not hold in reality, as one deals with CPU contention and a finite number of CPU cores, which limit how fast and how many read threads can run in parallel, especially when there are 112 total read threads. This explains the optimistic prediction. In fact, as strong evidence for this, the difference happens almost entirely in the queue time prediction for the read queues in device 5. The predicted queue time is $6.328 - 6.179 = 0.149$ ms, while the observed value is $9.995 - 6.179 = 3.816$ ms. Thus, the model significantly underestimates queue time for the reasons above. As more evidence, note that the predictions are much better in general when one reduces the number of servers, and thus CPU contention: e.g. for 3 servers, full replication and 10% writes, the predicted `get` total response time is 12.013ms, versus 12.835 ms observed, which gives a relative error of about 6%. The `set` response time prediction is also a bit off, giving a relative error of about 17%. This error can be explained by the fact that the write queues are not accurately modeled by the queues of devices 3 with service time $T_{\text{send}} - T_{\text{dequeue}}$. This is so because I designed my asynchronous write thread to follow a (dequeue request – send request to server – process responses (if any) – dequeue new request) loop. Therefore, the service time of device 3 does not accurately represent the actual time a write thread spends between successive dequeuings in general: it might spend some extra time processing responses. As evidence for this, note that the observed average queue time for `set` requests is 3.972 ms. On the other hand, the difference between observed and predicted `set` total response time is $23.085 - 19.076 = 4.009$ ms. This means that this is the only part that the model does not capture well for `set` requests.

In order to identify the bottleneck of the system, I look at the utilizations reported for all devices in the network. As is known, the bottleneck is exactly the device, excluding delay centers, which has the highest utilization (which implies the highest demand). Thus, the bottleneck of the system corresponds to the several instances of device 5, which represents the read thread pool and associated memcached server. It remains to decide which of the two (read thread pools or memcached servers) is the bottleneck of the system. To reach a conclusion, one can look closely at how the system behaves as its load is increased. Looking at the maximum throughput experiment of Section 1 of Milestone 2 (logs can be found in `max-tps`), one notes that, for 16 threads, `get` queue time goes up significantly while server time increases only slightly when load increases (e.g. coarse-grained exp., from 200 to 300 clients, queue time goes from 4.174 ms to 7.067 ms, while server time goes from 3.812 ms to 3.955 ms). This means that the load received by the memcached servers does not increase much, although the load imposed on the overall system does. This implies thus that read threads are not able to process the increased load faster, mostly due to the fact that most read threads are already busy for smaller loads, that read threads are synchronous, and that one has to deal with CPU contention and limited cores which limit how fast and how many read threads can be run in parallel. Therefore, the read thread pools are the bottlenecks of the system, and more specifically because the Azure VM has limited CPU resources and cores available, and due to their synchronous behavior.

4 Factorial Experiment

The goal of this section is to design a $2^k r$ factorial experiment and then analyze it. I decided to choose an experiment where I vary three factors at the same time, i.e. $k = 3$. The reason for choosing $k = 3$ is to keep a tradeoff between a simple regression model, where one does not have to deal with many interactions which may make it harder to conclude which effects are the most important for the system, while being able to showcase interesting and complete behavior of the system at the same time. Thus, I want to choose three factors that have a great effect in the behavior of the system. Given what I observed and learned in Milestone 2, I decided to choose number of servers,replication factor and write proportion, as they have shown to be important parameters of the system which are related. In order to get a more complete picture of the effects of these parameters on the system, I choose extreme values for their two levels: 3 and 7 servers, no and full replication, and 1% and 10% writes. From the experience gained in Milestone 2, I know that average total throughput is a parameter of the system that is heavily influenced by the variation of the three factors chosen. Thus, I decided to analyze and understand how exactly number of servers, replication factor and write proportion affect average total throughput in the $2^k r$ factorial experiment. The data used in this factorial design is taken from the experiments of the re-run of the write proportion experiment (Section 3 of Milestone 2) already analyzed in Section 2 of this Milestone. The raw memaslap and middleware logs, and processed data can be found in `m23-rerun`.

Since I ran 4 repetitions and have 3 factors, I design and analyze a $2^3 4$ factorial experiment for total throughput. I use an additive regression model,

$$y_{ij} = q_0 + q_A x_{Ai} + q_B x_{Bi} + q_C x_{Ci} + q_{AB} x_{Ai} x_{Bi} + q_{AC} x_{Ai} x_{Ci} + q_{BC} x_{Bi} x_{Ci} + q_{ABC} x_{Ai} x_{Bi} x_{Ci} + e_{ij},$$

where A corresponds to the replication factor, B corresponds to the number of servers and C corresponds to the write proportion. Also, q_0 is the mean throughput, q_z for $z \in \{A, B, C, AB, AC, BC, ABC\}$ are the effects of each factor and of their interactions, and e_{ij} is the experimental error and y_{ij} is the total throughput of a given repetition j of a certain combination i of factors A, B and C. The variables x_A, x_B and x_C are defined as follows

$$x_A = \begin{cases} 1 & \text{if full replication} \\ -1 & \text{if no replication} \end{cases}, \quad x_B = \begin{cases} 1 & \text{if 7 servers} \\ -1 & \text{if 3 servers} \end{cases}, \quad x_C = \begin{cases} 1 & \text{if 10\% writes} \\ -1 & \text{if 1\% writes} \end{cases}. \quad (1)$$

The choice of using an additive model (and not a multiplicative one) is due to three facts: first, the ratio between the maximum and minimum throughput observed over all experiments is small (≈ 1.50), so the multiplicative model with a log transformation would yield similar results to an additive one, as the log function behaves close to linearly between extremes when their ratio is small. Second, the additive model is already a good fit, with a very small fraction of unexplained variation. Finally, the experimental errors do not showcase any sort of discernible pattern, and they are of the same magnitude for all combinations of levels.

4.1 Computation of effects and estimation of experimental errors

Table 8 showcases the data used for the factorial experiment and the resulting effects, which are computed with the sign table method using the values of the 10th column (pertaining to the mean $\overline{y}_i$ of each combination i considered over its repetitions).

One can now compute the experimental error e_{ij} associated to y_{ij}, the average total throughput of repetition j of combination i, as $e_{ij} = y_{ij} - \hat{y}_i$, where $\hat{y}_i$ is the predicted throughput of the model for configuration i in the factorial experiment, and which is also the average total throughput $\overline{y}_i$ over the 4 repetitions of that configuration. Table 9 showcases the estimates of the experimental errors for each repetition of each combination of factors.

One can now compute the value of SSE $= \sum_{i=1}^{8} \sum_{j=1}^{4} e_{ij}^2$, which will be useful in the next subsection. I obtained SSE $= 1714854$.

I	A	B	C	AB	AC	BC	ABC	(y_1, y_2, y_3, y_4) (ops/sec)	$\overline{y}$ (ops/sec)
1	-1	-1	-1	1	1	1	-1	(15957.9, 16019.6, 15772.2, 16214.9)	15991.2
1	1	-1	-1	-1	-1	1	1	(15731.8, 16146.6, 16022.3, 15827.2)	15932.0
1	-1	1	-1	-1	1	-1	1	(13473.7, 12843.5, 13444.8, 13648.9)	13352.7
1	1	1	-1	1	-1	-1	-1	(13379.6, 12949.1, 12644.2, 12817.1)	12947.5
1	-1	-1	1	1	-1	-1	1	(15448.7, 15982.3, 15460.5, 15984.5)	15719.0
1	1	-1	1	1	1	-1	-1	(15011, 15117.1, 15398.8, 15189.1)	15179.0
1	-1	1	1	1	-1	1	-1	(12523.5, 13155.3, 13213.9, 13201.8)	13023.6
1	1	1	1	1	1	1	1	(11348.2, 11146.7, 11199.3, 10820.9)	11128.8
14159.2	−362.4	−1546.1	−396.6	−212.6	−246.3	−140.3	−126.1		

Table 8: $2^3 4$ design analysis. The value y_j corresponds to the average total throughput of repetition j of a given configuration, for $j = 1, \ldots, 4$. The last row showcases the effects associated to each factor/interaction, computed using the sign table method.

	(no, 3, 1%)	(full, 3, 1%)	(no, 7, 1%)	(full, 7, 1%)	(no, 3, 10%)	(full, 3, 10%)	(no, 7, 10%)	(full, 7, 10%)
rep. 1	-33.3	-200.2	121	432.1	-270.3	-168	-500.1	219.4
rep. 2	28.4	214.6	-509.2	1.6	263.3	-61.9	131.7	17.9
rep. 3	-219	90.3	92.1	-303.3	-258.5	219.8	190.3	70.5
rep. 4	223.7	-104.8	296.2	-130.4	265.5	10.1	178.2	-307.9

Table 9: Error table for design analysis. Entry (i, j) corresponds to e_{ji} in ops/sec.

4.2 Allocation of variation

In this subsection, I compute the proportion of the variation due to each factor, to their interactions, and due to experimental errors. In a $2^3 4$ factorial experiment, SST = SSA + SSB + SSC + SSAB + SSAC + SSBC + SSABC + SSE, where SST $= \sum_{i=1}^{8} \sum_{j=1}^{4} (y_{ij} - q_0)^2$, SSz $= 2^3 4 q_z^2 = 32 q_z^2$ for $z \in \{A, B, C, AB, AC, BC, ABC\}$, and SSE was computed in the previous subsection. Table 10 showcases the sums of squares and the proportion of variation allocated to each effect, as well as unexplained variation due to experimental error.

	Total	Errors	A	B	C	AB	AC	BC	ABC
SSz	91968300	1714854	4202680	76493600	5033330	1446360	1941240	629890	508838
Variation prop.	100%	1.86%	4.57%	83.17%	5.47%	1.57%	2.11%	0.68%	0.55%

Table 10: Sums of squares and corresponding proportions of variation allocated due to experimental error, factors, and their interactions. Variation proportion is computed as SSz/SST × 100%.

4.3 Confidence intervals for effects

In this subsection, I compute 95% confidence intervals for the effects of the $2^3 4$ factorial experiment. I assume errors to be i.i.d distributed according to a normal distribution with mean zero, and that all errors for the values of the predictors have the same variance. It then follows that the 95% confidence interval of effect q_z is of the form $q_z \pm t_{[0.975, 2^2 \cdot 3]} s_{q_z}$, for $z \in \{0, A, B, C, AB, AC, BC, ABC\}$, where $t_{[0.975, 2^3 \cdot 3]} = 2.064$ is the critical value of the t-distribution for these parameters, and $s_{q_z} = \frac{s_e}{\sqrt{2^3 \cdot 4}}$ for $s_e = \sqrt{\frac{\text{SSE}}{2^3 \cdot 3}} \approx 267.306$. Thus, $s_{q_z} \approx 47.253$, and one obtains 95% confidence intervals bounded by $q_z \pm t_{[0.975, 2^2 \cdot 3]} \cdot 47.253 = q_z \pm 97.531$ for each effect $z \in \{0, A, B, C, AB, AC, BC, ABC\}$.

4.4 Discussion of results

Looking at the results of the factorial experiment, one can point out several interesting outcomes. Every effect is significant at a 95% confidence level, since no confidence interval computed contains 0 (in fact, 97.531 is smaller than every effect computed). That the effect of each factor is significant is as expected, since it can be seen in Section 3 of Milestone 2 that the factors chosen markedly affect total throughput as they vary. The effects due to interaction are also significant. The significance of the effect of the interaction between number of servers (A)

and replication factor (B) can be explained as follows: increasing both number of servers and replication factor at the same time leads to more CPU contention and a more unstable network, and a higher `set` load per server, as the requests are replicated more. Thus, processing `set` requests becomes even more expensive due to the interaction than if one of the factors was fixed: not only do they have to be replicated to several servers, but also the write thread handling the request and the network it has to traverse are much slower. The significance of the effect of the interaction between number of servers (A) and write proportion (C) can be explained as follows: CPU contention and network instability increase, which makes every request slower. Also, `set` requests are always more expensive than `get` requests, and their proportion increases significantly. Thus, not only is every request more expensive, but a markedly larger fraction of them are significantly more expensive than before. The significance of the effect of the interaction between replication factor (B) and write proportion (C) can be explained by the fact that increasing replication factor significantly widens the difference in processing times between `set` and `get` requests, while increasing write proportion means that now one has a larger proportion of these even more expensive requests. Finally, the significance of the interaction between all factors can be explained largely by what has been said above: increasing both number of servers and replication factor lead to much more expensive `set` requests and widen their difference to `get` requests, while increasing write proportion means that the system has to deal with a markedly larger proportion of these very expensive requests.

Note that the variation unexplained due to experimental errors amounts only to 1.86% of total variation. Therefore, the additive regression model was a good choice, the experiment provides reliable data on which to base our regression model, and the behavior of the system showcases interesting variations. A major part of variation (84.09%) was allocated to the number of servers, while much smaller proportions have been allocated to replication factor (4.57%), write proportion (5.47%) and their interactions. These differences in variation allocated can be explained as follows: increasing the number of servers leads to significantly more CPU contention and more connections over the network (which make the network more unstable), which overweights the smaller request load per server. This worsening in performance affects both read and write threads, and so both `get` and `set` requests. On the other hand, increasing the replication factor markedly increases the processing time of `set` requests, while `get` requests are only very slightly affected due to higher `set` load on the servers, which are more expensive than `get` requests. Especially for 1% writes, this means that changes in replication factor significantly affect only an extremely small fraction of requests. Furthermore, the increase in write proportion is only from 1% to 10%, and so in both cases the fraction of `get` severely dominates over the `get` requests. Thus, both the increases in replication factor and write proportion affect approximately the same fraction of requests (they affect all `set` requests and also `get` requests slightly), which is always a small fraction of the requests affected by increasing the number of servers, which has a significant effect on system performance. Due to this, these two factors are also naturally allocated a similar fraction of variation. The fact that write proportion is allocated slightly more variation than replication factor can be explained as follows: increasing replication factor has a significant effect mostly when the write proportion is large (10%). On the other hand, increasing write proportion always has a significant effect on throughput because the difference in response time between a `get` request and a `set` request is always significant, even if the `set` request is not replicated.

5 Interactive Law Verification

The goal of this section is to verify the validity of all experiments from one of the sections in Milestone 2 using the interactive law. I decided to focus on the experiments of Section 2 of Milestone 2. Since I re-ran the experiments of Sections 2 of Milestone 2 to be used in Section 2 (but ended up not showing them in that section), I use the data from these new experiments. The raw memaslap and middleware logs, and the processed data can be found in `m22-rerun`.

The interactive law states that the throughput X and response time R of an ideal system are related by

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

where N is the load imposed on the system and Z is the think time of the clients. In our case, the load N is the number of clients connected to the system, due to the behavior of memaslap. Thus, in my case, $N = 198$.

It remains to estimate the think time Z of the memaslap clients. In order to do this, I conducted the following experiment:

Number of servers	1 (Basic A2)
Number of client machines	1 (Basic A2)
Middleware	Present (Basic A2)
Virtual clients / machine	1
Threads in pool	16
Workload	Key 16B, Value 128B, Writes 5%
Overwrite (`-o`)	0.9
Logging	every request
Replication factor	1 (no replication)
Runtime x repetitions	2m x 1

The goal of the experiment is to compute statistics of the memaslap think time. The middleware log from which think time statistics are computed is in `think-time`. This is accomplished by measuring the difference between the T_{end} and T_{start} timestamps of two consecutive requests sent by memaslap, which yields an approximation (upper bound) of the think time. Nevertheless, it includes also network latency and the time spent setting the T_{start} timestamp of the second request. In this experiment, the server, client and middleware are all in the same Azure VM, in order to greatly decrease the effect of the network latency between the client and the middleware. Table 11 showcases statistics for the memaslap think time. It is clear that think time follows a tail distribution.

Avg. (ms)	25th p. (ms)	50th p. (ms)	75th p. (ms)	90th p. (ms)	STD (ms)
0.2175	0.0235	0.0263	0.0412	0.0922	1.0168

Table 11: Think time statistics.

Taking this experiment into account, I set $Z = 0.2175$ ms. Note that this is a very pessimistic estimate of the average think time. In fact, think time should not experience significant variation throughout the experiment for both `get` and `set` requests, and so percentiles should provide a better estimate of the real think time, which appear to be negligible. Variations showcased by the statistics of Table 11 are probably mostly due to the local network latency and CPU resource allocation.

Table 12 showcases the comparison between real average response time and the values predicted by the interactive law for the different configurations considered in the experiment. The predictions are computed as $198/X - 0.2175$, where X is the corresponding throughput. The relative error between predicted and real values is computed as follows: if x is the real value and $\hat{x}$ is the prediction, then the error is $100 \times (x - \hat{x})/x$ (in %). Note that, even when using a very pessimistic estimate for the think time Z, the relative errors are all between 2.5% and 3.5%. Furthermore, predicted and real response time follow the same trend as one moves down the table (since throughput decreases as response time increases in the experiment). Given that the percentiles showcased in Table 11 are very small compared to the observed response times, another option for verifying the interactive law would be to set $Z = 0$, as it appears to be negligible, save for spikes which are likely caused by external factors. Errors for this case are showcased in Table 13. Note that all relative errors become even smaller than in Table 12. Thus, the exact predictions for response time (which would use the exact average think time) and the relative errors are bounded from below by the values of Table 13, and from above by the values of Table 12.

Combining the observations of the previous paragraph shows that my system follows the interactive law with good accuracy in the experiments considered.

(servers, rep. factor)	Avg. TPS (ops/sec)	Real Avg. RT (ms)	Predicted Avg. RT (ms)	Rel. error
(3,1)	16281	12.255	11.944	2.538%
(3,2)	16069	12.500	12.104	3.168%
(3,3)	15470	12.934	12.582	2.722%
(5,1)	15157	13.206	12.846	2.726%
(5,3)	14406	14.017	13.527	3.496%
(5,5)	14149	14.118	13.776	2.422%
(7,1)	13696	14.655	14.239	2.839%
(7,4)	12920	15.564	15.108	2.930%
(7,7)	12637	15.854	15.451	2.542%

Table 12: Interactive law verification with $N = 198$ and $Z = 0.2175$ ms.

(servers, rep. factor)	Avg. TPS (ops/sec)	Real Avg. RT (ms)	Predicted Avg. RT (ms)	Rel. error
(3,1)	16281	12.255	12.161	0.767%
(3,2)	16069	12.500	12.322	1.424%
(3,3)	15470	12.934	12.799	1.044%
(5,1)	15157	13.206	13.063	1.083%
(5,3)	14406	14.017	13.744	1.948%
(5,5)	14149	14.118	13.994	0.878%
(7,1)	13696	14.655	14.457	1.351%
(7,4)	12920	15.564	15.325	1.536%
(7,7)	12637	15.854	15.669	1.167%

Table 13: Interactive law verification with $N = 198$ and $Z = 0$ ms.

The behavior of the errors between the predictions and the observed values can be explained as follows. First, note that our system is far from ideal (as already seen in Section 2 of Milestone 2), and that throughput and response time are both random variables and do not stay constant throughout an experiment. Second, the response time follows a distribution with a long right-handed tail. This means that the average (which I am using in the interactive law verification) lacks some accuracy in predicting the behavior of response time. These reasons explain the relative errors between prediction and observation. Note also that all relative errors are positive, i.e. predicted response times are always lower than the observed values. Again, this can be explained by the fact that response time follows a right-handed tail distribution: to see this, observe Figure 4 of Section 1 of Milestone 2, which shows that the total time spent in the middleware by a request follows a tail distribution. Since the client-middleware network follows also a tail distribution, memaslap response time distribution also has a tail. Thus, spikes in response time occur more often and are larger to the right (increase in response time) than to the left (decrease in response time). These spikes increase the average response time, although they last only for a very short amount of time. Thus, average throughput is not affected as much as average response time by these spikes, which explains why observed average response time is always larger than the predictions.

Logfile listing

Below you find a list of folders containing log files. Each folder contains a README file.

Short name	Location
stab-trace	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/tree/master/M3_1
busy-31	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M3_1/threads_
m23-rerun	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/tree/master/M3_2/M2_3
busy-32	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M3_2/busy-32.
jmt	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/tree/master/M3_3
max-tps	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/tree/master/M2_1
m22-rerun	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/tree/master/M3_2/M2_2
think-time	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M3_5/think_ti

Appendix: Minor code changes and re-running the stability trace

Code changes consisted only in modifications in the logging functionality of the middleware: timestamps (in nanoseconds) were added besides the intervals logged before, and a synchronized variable shared between read threads which details how many busy read threads there are in the system was also added.[2][3][4]

I plot throughput for the re-run of the stability trace (Milestone 1), which was used in Section 1 of this Milestone, to show that it is stable. The plot can be found in Figure 8. The time slice analyzed corresponds to the 300s-3900s period in the plot.

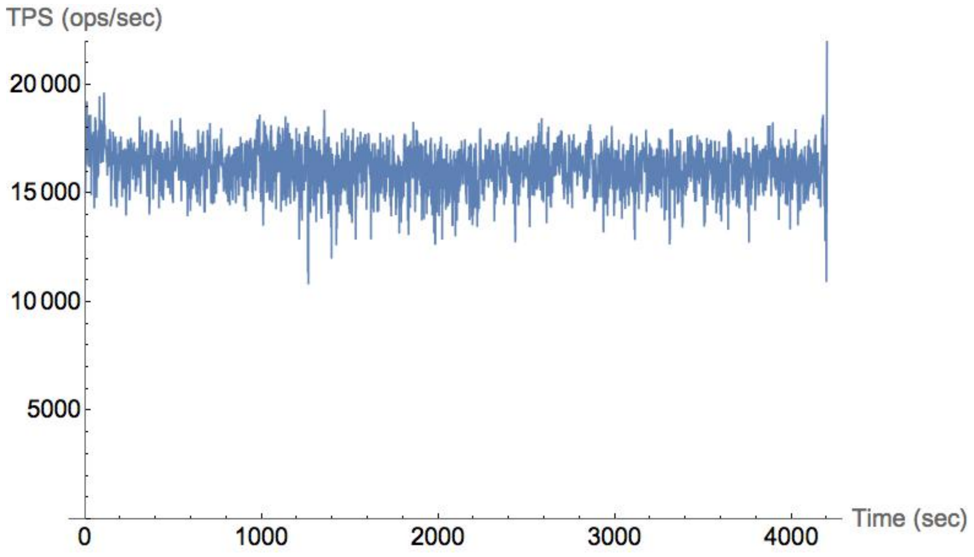

Figure 8: Throughput plot of stability trace re-run.

[2]https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/src/joao/asl/Middleware.java

[3]https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/src/joao/asl/ReadThread.java

[4]https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/src/joao/asl/WriteThread.java