Advanced Systems Lab (Fall'16) – Second Milestone

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

Grading

Section	**Points**
1	
2	
3	
Total	

1 Maximum Throughput

The goals of this section are, first, to find the minimum virtual clients/threads in pool combination that achieves the highest throughput in my system when it is connected to 5 servers with no replication, with a read-only workload configuration on the clients' side, and second, to provide a detailed breakdown of the time spent by read requests inside the middleware.

1.1 Hypotheses

My hypotheses, previous to running any experiments, are as follows:

1. For a fixed number of threads and varying number of clients, I expect to observe a throughput plot with a similar shape to the baseline throughput plot from Milestone 1. The saturation point in the baseline lies in the 64-70 clients range for 1 server. Since we have 5 servers here instead of 1, and since the throughput achieved in the stability trace is roughly 50% of that of the baseline, I believe the saturation point for this setup should be at around $0.5 \times 5 \times 70 = 175$ clients.

2. The throughput curve should shift up as we increase the number of threads, until it hits a ceiling. Given that memcached servers can process a large load of incoming requests and the fact that the middleware design is simple, I believe the ceiling will be due to limitations of the middleware.

1.2 Maximum throughput experiment and discussion

In order to investigate the goals of this section, I ran the following experiments. Differences between them are highlighted in the table with blue and red colors. The "blue" experiment has larger steps, and the "red" experiment is a finer-grained experiment used to pinpoint the desirable configuration more exactly.

Number of servers	5 (Basic A2), servers restarted for every run
Number of client machines	5 (Basic A2)
Middleware	Present (Basic A4)
Virtual clients / machine	**10 to 70** / **36 to 50 (steps of 2)**
Threads in pool	**8, 16, 24, 40** / **16, 24**
Workload	Key 16B, Value 128B, Writes 0%
Overwrite (`-o`)	0.9
Window size (`-w`)	1k
Replication factor	1 (no replication)
Runtime x repetitions	3m x 4

For each run of the experiment, I take into consideration the data pertaining to the 60s-120s period of each memaslap instance. In a 0% writes setup, memaslap instances first process all write requests before starting the experiment. With a window size of 1k, this means that each virtual client first sends 1000 write requests to the middleware. Thus, memaslap instances might take some time to start, and more importantly, different instances will start their runs at different times. Furthermore, one does not have much direct control over this. The decision to measure the 60s-120s period was made for the following reasons. First, it ensures that all memaslap instances are stable during that period. This is so because, from personal experience, no two memaslap instances start with more than 20 seconds gap between them, even for larger numbers of clients. Thus, the 60s-120s period for each memaslap instance will be such that all instances have already started sending read requests some time ago and are in stable state. Second, measuring for 1 minute in stable state gives a more complete picture of the behavior of the system under consideration. Third, since memaslap instances may start at different times,

the period measured may differ from instance to instance. Using 1 minute intervals (which are relatively large) for measurements is also an attempt to ensure that measured periods from different memaslap instances intersect as much as possible. Nevertheless, since I always measure in stable state, averages and standard deviations of important parameters should be very similar even when the measured periods do not intersect.

One must take special care when computing standard deviations for throughput over the period measured in this experiment. Since memaslap instances start at different times, their logs are not in sync. Thus, the straighforward way to compute standard deviation – getting aggregate throughput for each second of the 60 seconds period and then computing the average and the standard deviation from these values – yields a value which can be quite far away from the real standard deviation, as variations may get smoothed out. To get a better estimate, I compute the standard deviation s_i for each client i, and then compute $z = \sqrt{s_1^2 + \cdots + s_5^2}$ as the estimate for that particular repetition. The final estimate is computed by taking the average of the standard deviation over the period measured for all 4 repetitions. This yields a good estimate for the standard deviation of the aggregated throughput one expects to observe over the period measured in a particular run of the experiment. Standard deviation for memaslap response time is computed by averaging the standard deviation reported in the logs over the period measured for each memaslap client, then computing $\sqrt{s_1^2 + \cdots + s_3^2}$ and averaging over repetitions (like for throughput standard deviation). Marked differences between standard deviations of middleware times and of memaslap times can be explained by the fact that this method is just an estimate, while for middleware times I compute the exact standard deviation

In Figure 1, I have plotted average throughput over 4 repetitions for each combination of clients and threads measured in the experiment, along with the respective standard deviations. These values are included in the log folder `proc21-memaslap`. The raw memaslap log files can be found in the log folder `logs21-memaslap`.

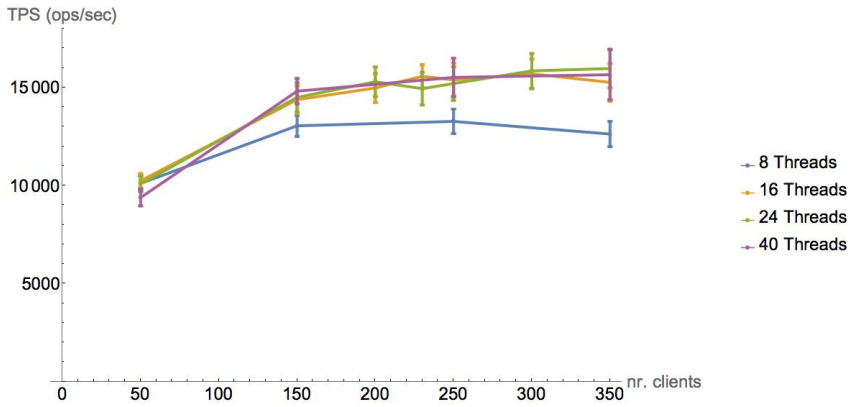

Figure 1: Average throughput for each combination of clients and threads with corresponding standard deviation over the measured period (60s-120s). Each curve represents how throughput varies with the number of clients for a fixed number of threads.

By direct observation of Figure 1, one notes that all throughput curves have the expected shape. Moreover, they are shifted up for larger numbers of threads, although the ceiling is hit quite soon: the curves for 16, 24, and 40 threads are higher than the curve for 8 threads with a difference of about 2000 ops/sec, and one notices almost no throughput gain when varying the number of threads from 16 to 40. Finally, note that the saturation point appears to be higher than the 175 clients estimate for setups with more than 8 threads. This discrepancy between Hypothesis 1 and observation can be explained as follows: for the estimate, I use the fact that throughput is reduced by 50% in the stability trace of Milestone 1 when compared to the baseline experiment of the same Milestone. This throughput reduction was observed for

a setup with 3 servers and full replication. Here, I deal with 5 servers and *no* replication, so using the 50% reduction in the previous calculations is too pessimistic. Thus, it is natural that the saturation point is a bit higher than the prediction. Note that, for all threads, throughput increases steeply from 50 to 150 clients. For 8 threads, it saturates at 150 clients. For 16, 24, and 40 threads, one still notices a smaller increase in throughput from 150 to the 200-230 clients range, after which they are mostly flat. Thus, I conclude that these threads saturate at around 200-230 clients. The fact that the throughput ceiling is hit so soon (at 16 threads) is due to limitations of the middleware. This is so because the distribution of time spent in the server for each request does not change much as one varies the number of clients, as memcached is prepared to handle very large loads, so it is not a limitation of memcached or the network. The ceiling exists because one has finite computing resources, so there comes a point when having more threads with less resources allocated per thread is worse than having less threads with more resources per thread. One design decision that affects how soon the ceiling is hit is the fact that write threads keep working at 100% even while they do not receive any set requests. This markedly decreases the resources that can be allocated to read threads. Thus, Hypothesis 2 holds.

The first goal is to find the minimum combination of clients and threads that achieves the maximum throughput. There is an important remark about this: system stability and resource minimality (connections and memory used) are important. Thus, one needs to find a sweet spot between achieving maximum throughput, being stable, and using as few resources as possible. Since 8 threads yield a relatively low throughput curve, I rule out configurations with 8 threads. Moreover, there are no significant differences in throughput curves for 16, 24 and 40 threads. Thus, I rule out configurations with 40 threads, as they use much more resources than ones with 16 or 24 threads with similar number of clients, and I observe no significant throughput gain. For 16 and 24 threads, it appears we can achieve close to maximum throughput in the 200-250 clients range. Note that the global maximum for throughput is achieved outside this range of clients and threads. Nevertheless, I choose to focus on this range because there are configurations achieving close to maximum throughput, while using less resources and being more stable. To analyze stability, I extracted standard deviation and approximate 50th, 90th, and 99th percentiles of response time from the memaslap logs.

I decided to consider 50th, 90th and 99th percentiles throughout this report for the reasons detailed next: from personal experience, changes in the behavior of time distributions from the middleware in the experiments of this report are mostly translated into changes for higher percentiles (especially 90th and 99th), while lower percentiles are not significantly affected. Furthermore, higher percentiles also give a nice picture of how the stability of a certain operation evolves, which is useful when studying behaviors of time distributions. Finally, in the case of memaslap, approximations are very rough and so, for lower percentiles, do not provide any useful information.

Standard deviation alone is not meaningful, as one deals with a distribution with a long tail, so it needs to be coupled with percentiles for a more complete picture of the system's stability. Percentiles are approximated by analyzing the Log2 distribution featured in the memaslap logs. Featured values for percentiles are upper bounds which are not necessarily very tight, but are still suitable to compare response time distributions. The Log2 distribution buckets also include requests from the warmup and cooldown phases. Nevertheless, given that approximations are rough and that these phases do not last for long (the system spends a great majority of the experiment time in a stable mode), these requests will not affect approximations significantly.

In Table 1, I compare the stability of the configuration achieving maximum throughput nearby with that of other farther away points achieving larger throughput.

(threads,clients)	TPS± STD (ops/sec)	50th p. (ms)	90th p. (ms)	99th p. (ms)	RT STD (ms)
(16,230)	15570 ± 592	33	66	131	5.5
(16,300)	15694 ± 754	66	66	262	7.2
(24,300)	15847 ± 884	66	66	262	7.5
(24,350)	15968 ± 984	66	66	262	8.6
(40,350)	15653 ± 1280	66	66	524	11.0

Table 1: Comparison of response time distribution statistics and throughput for some configurations. Percentiles and standard deviation in milliseconds correspond to response time reported by memaslap in experiment of Figure 1.

Note that, in Table 1, the (16,230) configuration seems to be much more stable than the others, due to lower percentiles and standard deviation. Furthermore, configurations nearby (16,230) have similar stability to it. Thus, I conclude that I can focus on configurations close to 16 threads and 200 clients to determine the configuration which achieves close to maximum throughput with minimal resources and decent stability.

I decided to run a finer-grained analysis of the 180-250 clients and 16-24 threads ranges, as I conclude in the previous experiment that the desirable configuration lies in these ranges. The resulting plot can be found in Figure 2. The raw memaslap logs can be found in `fine-memaslap` and the raw middleware logs can be found in `fine-mw`. The processed data can be found in `fine-proc-memaslap` and `fine-proc-mw`. Note that this experiment was conducted at a different time than the previous one, so the results are not comparable. The setup is the same as the first experiment, but now clients/machine range from 36 to 50 in steps of 2, and I only consider 16 and 24 threads.

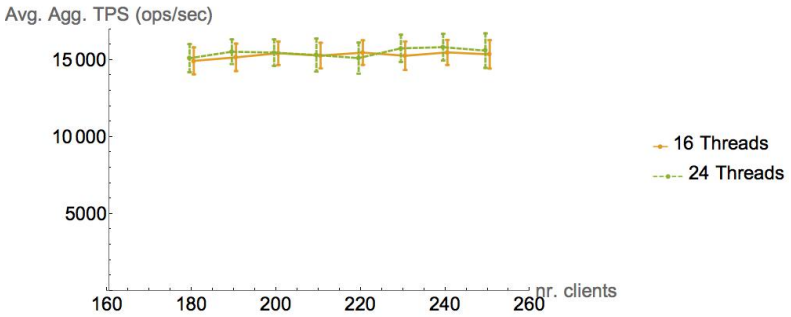

Figure 2: Fine-grained experiment between 180 and 250 clients, with 16 and 24 threads.

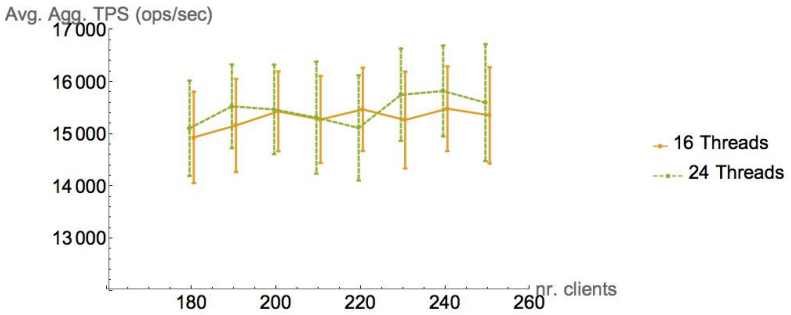

Figure 3: Zoom-in of fine-grained experiment between 180 and 250 clients, with 16 and 24 threads.

In Figure 2, the curve for 16 threads starts by increasing linearly from 180 to 200, and then stabilizes, oscillating periodically around an almost constant value. This can be more easily seen in Figure 3. Thus, I conclude that the saturation point is hit at 200 clients for 16 threads. The behavior for 24 threads is a bit more erratic, but it yields similar throughput to the configurations with 16 threads, and there seems to be a saturation point at around 190-200 clients as well.

Based on the two experiments conducted, I choose the configuration with 16 threads and 200 clients as the one achieving close to maximum throughput, while using minimal resources and having good stability. More justifications for this decision follow in the next paragraphs.

First, the (16,200) configuration is the first "peak" of the 16 threads curve in Figure 2. Also, one notices almost no throughput increase after the 200 clients mark, while it was increasing linearly before this. Second, while there are some configurations with 16 and 24 threads achieving more throughput, these configurations use more resources (more threads and/or clients) than (16,200), and are also generally less stable. These negative differences overweight the difference in throughput achieved. Refer to Table 2 for a comparison of the stability between (16,200), nearby configurations, and configurations achieving higher throughput.

(threads,clients)	TPS± STD (ops/sec)	50th p. (ms)	90th p. (ms)	99th p. (ms)	RT STD (ms)
(16,190)	15164 ± 891	33	66	131	5.9
(16,200)	15432 ± 764	33	66	131	5.2
(16,220)	15471 ± 797	33	66	131	6.1
(16,240)	15484 ± 813	33	66	262	6.4
(24,190)	15528 ± 801	33	66	131	5.2
(24,200)	15469 ± 858	33	66	131	6.2
(24,230)	15750 ± 884	33	66	262	6.4
(24,240)	15822 ± 869	33	66	262	6.4
(24,250)	15599 ± 1120	33	66	262	8.0

Table 2: Comparison of response time distribution statistics and throughput for some configurations. Percentiles and standard deviation in milliseconds correspond to response time reported by memaslap in the experiment of Figure 2.

Observing Table 2, one notices that (16,200) is one of the most stable configurations, as it features relatively small percentiles and response time standard deviation when compared to other configurations. Configurations with the largest difference in throughput when compared to (16,200) are (24,230) and (24.240). These configurations feature larger 99th percentile and standard deviation. Furthermore, they use $5 \times 8 = 40$ additional threads, and the gain in throughput is small. Thus, I conclude that (16,200) is a better choice. Configurations (16,220) and (16,240) achieve very similar throughput to (16,200), and are also similarly stable. Since they use more clients (and thus more connections with associated socket buffers), I prefer (16,200). Finally, the configuration (24,190) deserves a remark: it achieves similar throughput and stability to (16,200). One can take a closer look at the resources used by both configurations: (24,190) has 10 less clients, but 40 more threads in total than (16,200). It follows that (24,190) uses 30 extra connections, along with associated socket buffers and thread buffers. Thus, I prefer (16,200) to (24,190), as it achieves similar throughput, is similarly stable, and uses less resources. Note that to overcome this difference in resources consumed, one would have to decrease the number of clients in the 24 threads configuration to 150 or less. As evidenced in the first experiment (Figure 1), the throughput would be quite smaller for the 24 threads/150 clients configuration when compared to configurations near 24 threads and 200 clients, as it is still increasing in this range, so there would be no benefit in decreasing the number of clients, and 16 threads/200 clients would still be the superior choice.

In Table 3, I present throughput and response time distribution for a few more configuration near (16,200) that were not considered in Table 2.

(threads,clients)	TPS± STD (ops/sec)	50th p. (ms)	90th p. (ms)	99th p. (ms)	RT STD (ms)
(16,180)	14934 ± 873	33	66	131	5.6
(16,200)	15432 ± 764	33	66	131	5.2
(16,210)	15275 ± 834	33	66	131	5.8
(24,180)	15108 ± 913	33	66	131	6.1
(24,210)	15312 ± 1074	33	66	131	7.4

Table 3: Comparison of response time distribution statistics and throughput for configurations near (16,200). Percentiles and standard deviation in milliseconds correspond to response time reported by memaslap in the experiment of Figure 2.

Observing Table 3, one notices that the configurations have similar stability (i.e. similar response time distribution statistics). Moreover, configuration (16,180) achieves somewhat less throughput, as it is at a point where the throughput curve for 16 threads is still increasing (albeit less steeply than in the 50-150 clients range).

1.3 Breakdown of times in middleware

In this subsection, I present a detailed breakdown of the time a `get` request spends in the middleware when there are 16 threads in the read thread pool and 200 total clients. The breakdown can be found in Table 4 and Figures 4, 5 and 6, and corresponds to the fine-grained experiment of Figure 2. It includes average, standard deviation, and 50th, 90th and 99th percentiles of the time spent in the server (T_{server}), the time spent in queue (T_{queue}), and the total time spent in the middleware by a `get` request, (T_{total}). These parameters provide a detailed quantitative picture of the underlying distribution of each of these times. The values are computed from the logs produced by the middleware (with a logging ratio of 1 log every 100 `get` requests). To compute the values, I aggregate the logged requests of each repetition over the period measured (in this case, the middle third of the requests logged) and compute the relevant statistics. This ensures that all data considered was collected in stable state (it corresponds roughly to the period I measure in). Also, it provides enough requests to obtain meaningful values: one third of the requests logged corresponds to 60 seconds, so, if we assume an average throughput of 15000 ops/sec over this period, we consider $60 \times 15000/100 = 9000$ requests per repetition. The time a request spends outside the queue and server (i.e. when it is being hashed and enqueued, between dequeuing and writing, and between reading the response and forwarding it back) corresponds to $T_{\text{total}} - T_{\text{queue}} - T_{\text{server}}$. This time generally makes up less than 1.5% of the total time and so is insignificant. Thus, $T_{\text{total}} \approx T_{\text{queue}} + T_{\text{server}}$ and I consider only these times in the breakdown.

16 threads/200 clients	Avg. (ms)	50th p. (ms)	90th p. (ms)	99th p. (ms)	STD (ms)
T_{server}	3.4	1	8	15	7.3
T_{queue}	3.5	1	10	23	7.0
T_{total}	7.0	7	14	34	10.4

Table 4: Breakdown of time spent in the middleware for the 16 threads, 200 clients configuration. Data extracted from the middleware logs of the runs for this configuration in the fine-grained experiment of Figure 2.

Observing Table 4, one first notices that statistics for queue and server time are similar, giving more evidence that the configuration chosen is a saturation point. Furthermore, all distributions considered appear to be tail distributions. Thus, average and standard deviation alone are not significant, and must be coupled with other parameters, such as the percentiles, to get a nice picture of the respective distribution. Moreover, there is an offset between the response time statistics output by memaslap, and the statistics of T_{total}. Average response time

reported by memaslap for the fine-grained experiment with 16 threads and 200 clients was 13.2 ms, with a standard deviation of 11.6ms. The 6.2ms difference in average and 1.2 ms difference in standard deviation are due to the network between the clients and the middleware: the travel times of a request from the client to the middleware and of a response from the middleware to the client are not accounted for in T_{total}, while they are included in the response time observed by memaslap.

Figures 4, 5 and 6 are histograms which give a nice visual picture of the distributions of the times logged in the middleware. These histograms were computed by aggregating the middle third of requests logged of each repetition and allocating them to 1ms buckets. It is thus clear that, as believed from Table 4, all distributions are far from gaussian. Furthermore, one notices tails in the histograms, especially in the T_{total} and T_{queue} distributions. Interestingly, there are two bumps in all distributions. The bump in Figure 6 is due to variances in network and server behavior. In turn, the bump in Figure 5 depends on this one. A momentary spike in server time leads to a backed up queue. For a short time afterwards, requests experience larger queue times as the queue is emptied out. Since requests arrive in an almost constant flux and normally are processed right away, it follows that it should take as much time for the queue to return to normal as the time in the server during the spike. This explains why the queue time bump exists, and why the bumps are located in the same zone (at ≈10ms). Since most requests have very small server time and queue time, one expects most requests with larger server time to still have small queue time and vice-versa. This means that if there are x requests with queue time around 10ms and y requests with server time around 10ms, it is reasonable to expect roughly $x + y$ requests with total time around 10ms, which is what can be observed in Figure 4. Thus, the total time and queue time bumps are direct consequences of the server time bump, which is caused by variations in network and server behavior independent of the middleware.

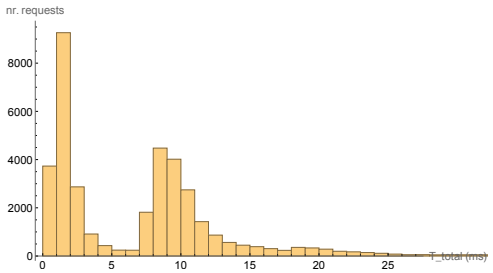

Figure 4: Histogram of T_{total} for 16 threads, 200 clients in fine-grained experiment.

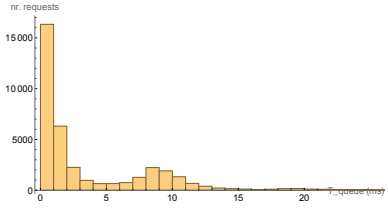

Figure 5: Histogram of T_{queue} for 16 threads, 200 clients in fine-grained experiment.

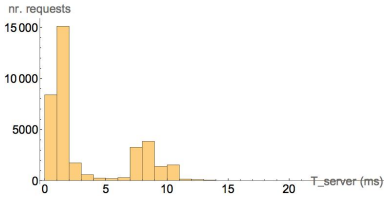

Figure 6: Histogram of T_{server} for 16 threads, 200 clients in fine-grained experiment.

2 Effect of Replication

The goal of this section is to explore how the system's behavior changes when the number of servers and the replication factor vary.

2.1 Hypotheses

My hypotheses, prior to running any experiments, are as follows:

1. For a fixed number of servers and increasing replication factor, `set` requests throughput and response time should be negatively impacted, and more so than `get` requests performance. This negative impact on `set` requests should be more noticeable for larger number of servers, as the replication factor has more room to increase.

2. For a fixed replication factor (none, half, or full), one expects throughput to increase as the number of servers increases. This is so because requests are uniformly distributed over the servers, and so each server receives less load.

3. The operation that becomes more expensive in the middleware should be `set` server time. This is because, as replication factor increases, it will depend on the performance of more servers and connections, and will equal the performance of the slowest server (including the connection).

4. The system should not scale like an ideal implementation. This is due to CPU contention and increased network latency for increasing number of servers and replication factor, as this implies more connections and threads in the system.

2.2 Experiment and discussion

In order to achieve the goal of this section and test my hypotheses, I ran the experiment detailed next. Note that I try to stay very close to the configuration selected in Section 1: 16 threads and 200 total clients.

Number of servers	3, 5, 7 (Basic A2), servers restarted every run
Number of client machines	3 (Basic A2)
Middleware	Present (Basic A4)
Virtual clients / machine	66 ($3 \times 66 = 198 \approx 200$ total clients)
Threads in pool	16
Workload	Key 16B, Value 128B, Writes 5%
Overwrite (`-o`)	0.9
Replication factor	none, half, full
Logging ratio (`get`/`set`)	every 100/every 5
Runtime x repetitions	105s x 5

The raw memaslap and middleware logs are in `logs22-memaslap-5` and `logs22-mw-5`, respectively. The processed data from the memaslap and middleware logs is in `proc22-memaslap-5` and `proc22-mw-5`. For each run of the experiment, I take into the consideration the 30s-90s period of each memaslap instance. From experience, the system achieves a stable state much before the 30 seconds mark. Moreover, the cooldown phase only occurs after the 90 seconds mark. Furthermore, taking a 1 minute interval gives a more complete picture of the system's behavior. For this experiment, I decided to set the logging ratio for `get` requests at 1 log per 100 requests, and at 1 log per 5 requests for `set` requests. These ratios were chosen since I wanted roughly the same number of `get` and `set` requests logged so that one gets more reliable statistics when comparing the performance of both types of requests. Since the write proportion is very small (only 5%), changing the logging ratio for `set` requests does not impact the performance of the system in any observable way. To confirm this, I ran the same experiment with a "every 100" logging ratio for `set` requests and 3 repetitions (all other parameters are set like in the table above) in the exact same Azure virtual machines of the experiment of the table above. The raw memaslap and middleware logs of this experiment can be found in `log22-memaslap-100` and `log22-mw-100`, respectively. The processed memaslap and middleware logs can be found in

`proc22-memaslap-100` and `proc22-mw-100`. Observing the logs, one notices extremely similar throughput, response time and middleware statistics in both of these experiments. The trends for each parameter of interest are the same for both experiments. This means that I can modify the logging ratio for `set` requests without secondary effects on the experiment. Finally, standard deviation for throughput and middleware time distribution statistics are computed like in Section 1.

Figures 7 and 8 show average throughput for varying number of servers and replication factor. First, notice that, as predicted in Hypothesis 1, throughput decreases for both `get` and `set` requests for fixed number of servers and increasing replication factor. The reason is as follows: with a larger replication factor, the middleware must wait for the response of more servers before finishing processing a `set` request. The server response time is thus the response time of the slowest server considered. With larger replication factor, the slowest response time tends to be larger, as we consider more servers. Thus, `set` throughput decreases. The increase in processing time for `set` requests then influences the throughput achieved for `get` requests. This is so because each virtual client only sends a request after the previous one has been completed. Therefore, if `set` requests become slower, a virtual client cannot send as many `get` requests per second as before.

Notice also that throughput decreases for fixed replication factor and increasing number of servers, contrary to Hypothesis 2. This discrepancy is due to the fact that I did not take into account that a CPU has a finite amount of resources that can be allocated to the existing threads. Increasing the number of servers from S to $S + 2$ leads to an extra 34 threads in the middleware (16×2 read threads and 2 write threads). Thus, CPU resources must be allocated over a larger number of threads, so the performance level of each thread decreases. Write threads are especially expensive, as they are always working at 100% by design choice, while read threads block when not needed. There comes a point after which increasing the number of servers actually decreases the overall performance of the system, as the decrease in threads' performance levels overweights the fact that less requests are allocated to each server per second.

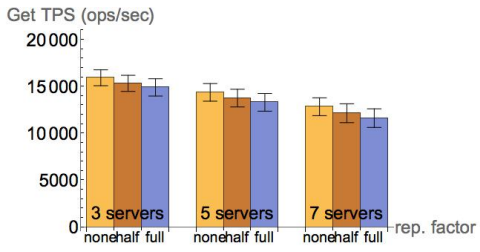

Figure 7: Average `get` throughput for varying replication factor.

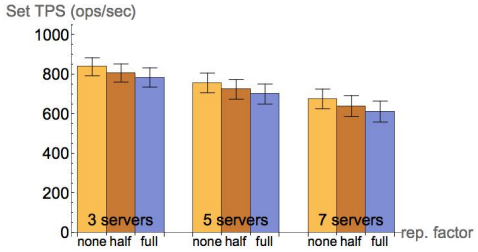

Figure 8: Average `set` throughput for varying replication factor.

Tables 5 and 6 show `get` and `set` response time statistics from memaslap. These are computed like in Section 1. Averages and standard deviations are extracted directly from memaslap logs, while percentiles are approximated from the Log2 distribution buckets. One notices that `set` response time distributions seem to be more unstable than `get` distributions for all configurations, in the sense that they feature larger averages, standard deviations and percentiles. Moreover, just like with throughput, response time distributions seem to become more unstable as one increases both number of servers and replication factor for both operation types.

(servers, rep. factor)	Avg. (ms)	50th p. (ms)	90th p. (ms)	99th p. (ms)	STD (ms)
(3,1)	11.8	33	66	131	5.8
(3,2)	12.2	33	66	131	6.4
(3,3)	12.6	33	66	131	6.8
(5,1)	13.1	33	66	131	7.1
(5,3)	13.6	33	66	262	7.5
(5,5)	14.1	33	66	262	8.3
(7,1)	14.7	33	66	262	9.1
(7,4)	15.6	33	66	262	9.7
(7,7)	16.3	33	66	262	10.9

Table 5: Memaslap `get` response time statistics for different configurations.

(servers, rep. factor)	Avg. (ms)	50th p. (ms)	90th p. (ms)	99th p. (ms)	STD (ms)
(3,1)	13.6	33	66	131	6.0
(3,2)	14.7	33	66	262	6.7
(3,3)	15.5	33	66	262	7.3
(5,1)	16.6	33	66	262	7.7
(5,3)	17.9	33	66	262	8.3
(5,5)	19.0	33	66	262	9.2
(7,1)	17.9	33	66	262	10.2
(7,4)	19.4	33	66	262	10.7
(7,7)	20.9	33	66	524	12.2

Table 6: Memaslap `set` response time statistics for different configurations.

The first goal of this section is to determine whether `get` and `set` requests are impacted in the same way by the different configurations tested. As a first observation, already mentioned above, note that both request types showcase the same trend: throughput and response time are both negatively impacted as number of servers and replication factor increase. In order to see whether one request type is impacted more negatively than other by the increase of servers and replication factor, I compute relative differences against the base cases in throughput as replication factor increases (and number of servers is constant) and vice-versa. In this case, the base case for varying replication factor for each number of servers S is the respective setup with S servers and no replication. For varying number of servers and fixed replication factor, the base case is the setup with 3 servers and the same replication factor. Relative differences are computed as follows: if the base case has value x and one observes value y for another setup, then the relative difference is $100 \times (y - x)/x$, i.e. it is the relative change (in percentage) in the value relative to the base case. If one request type has markedly larger relative decreases than another, I conclude that it is more negatively impacted. Tables 7 and 8 showcase relative decrease of throughput as replication factor increases for all numbers of servers and `get` and `set` requests. It is clear that relative decrease is the same for both request types, for increasing replication factor and fixed number of servers. Thus, I conclude that the throughput of both request types are impacted the same when increasing replication factor.

nr. servers	none to half	none to full
3	−3.8%	−6.5%
5	−4.2%	−7.4%
7	−5.4%	−9.5%

Table 7: Relative difference in throughput for `get` requests for increasing replication factor.

nr. servers	none to half	none to full
3	−3.8%	−6.4%
5	−4.2%	−7.4%
7	−5.3%	−9.5%

Table 8: Relative difference in throughput for `set` requests for increasing replication factor.

One can also investigate how both request types are impacted by increasing the number of servers, while keeping the replication factor fixed. Tables 9 and 10 showcase the relative difference in throughput for increasing number of servers and fixed replication factor, for both request types. Once again, both `get` and `set` throughputs are clearly impacted the same. This leads me to conclude that throughput of both request types is always impacted the same as one changes the setup. This contradicts part of Hypothesis 1. The explanation for this is that I did not predict the large degree of dependence between the request types. This is due to the behavior of memaslap: virtual clients do not send additional requests before receiving the response for the current one. This means that slow processing of `set` requests heavily affects the amount of `get` requests that are sent to the middleware per second.

rep. factor	3 to 5	3 to 7
none	−9.8%	−19.4%
half	−10.3%	−20.8%
full	−10.7%	−22.0%

Table 9: Relative difference in throughput for `get` requests for increasing number of servers.

rep. factor	3 to 5	3 to 7
none	−9.8%	−19.4%
half	−10.2%	−20.7%
full	−10.7%	−22.0%

Table 10: Relative difference in throughput for `set` requests for increasing number of servers.

The previous discussion allowed us to conclude that changing the setup in the experiment leads to the same impact on both `set` and `get` throughput. Nevertheless, this does not tell us how response times for requests of both request types are impacted: it could be that `set` requests take longer, but `get` requests take the same time. In this case, one could still notice the same impact in throughput for both request types since larger response times for `set` requests greatly affect the number of `get` requests created by the memaslap virtual clients. I compute relative differences in average response time from the values of Tables5 and 6. The base cases considered are the same as for the relative differences for throughput considered before. Tables 11 and 12 show that relative differences of average response time for varying replication factor are clearly larger for `set` requests. The situation for varying number of servers is also clear, as showcased in Tables 13 and 14: relative differences are significantly larger for `set` requests, especially when increasing from 3 to 5 servers. To complement this analysis, one can also look at the 99th percentile approximations and standard deviation in Tables 5 and 6: they are consistently larger for `set` requests, which means that the associated response time distribution is considerably more unstable than for `get` requests. Thus, I conclude that, in general, `set` requests are more impacted than `get` requests by the setup changes, especially when one varies the replication factor. The explanation for why `set` response time is more impacted than `get` response time when replication factor varies is as follows: for larger replication factor, the response time of a `set` request depends on the performance of more servers and connections. Since the server response time corresponds exactly to the response time of the slowest server, it increases and becomes more unstable for larger replication factor. On the other hand, increasing the replication factor does not affect `get` requests directly: they are still sent to only one server. Nevertheless, all servers experience an increase on their `set` load as replication factor increases. This implies a slight performance drop for the servers, as `set` requests are expensive, which explains the increase in response time for `get` requests. That `set` response time is more impacted than `get` response time for increasing number of servers is due to server contention and to the fact that there are much fewer threads dedicated to `set` requests. From this, the remaining part of Hypothesis 1 holds.

nr. servers	none to half	none to full
3	3.4%	6.8%
5	3.8%	7.6%
7	6.1%	10.9%

Table 11: Relative difference in average response time for `get` requests for increasing replication factor.

op. type	none to half	none to full
3	8.1%	14.0%
5	7.8%	14.5%
7	8.4%	16.8%

Table 12: Relative difference in average response time for `set` requests for increasing replication factor.

rep. factor	3 to 5	3 to 7
none	11.0%	24.6%
half	7.9%	27.9%
full	11.9%	29.4%

Table 13: Relative difference in average response time for `get` requests for increasing number of servers.

rep. factor	3 to 5	3 to 7
none	22.1%	31.6%
half	21.8%	32.0%
full	22.6%	34.8%

Table 14: Relative difference in average response time for `set` requests for increasing number of servers.

In order to see which operation becomes more expensive in the middleware as the setup changes, I compute statistics for the distributions of T_{queue} and T_{server} for all setups considered. Figures 9, 10, 13 and 14 showcase statistics for T_{server}, and Figures 11, 12, 15 and 16 showcase statistics for T_{queue} over various setups.

First, increasing the replication factor for a fixed number of servers does not have a significant effect on the server time of `get` requests (see Figures 9 and 13) as they are always sent to a single memcached server. The small increases in 99th percentile (which then affect the average and standard deviation) can be explained as follows: the servers receive more `set` requests for larger replication factors, so they become slightly more unstable. Nevertheless, their performance is not significantly affected as memcached is designed to handle large loads. Notice also that queue time distribution for `get` requests is not significantly affected by varying the replication factor. This can be explained as follows: since the load imposed on the system is constant over all the replication factors and since the configuration chosen is near a saturation point, `get` queue time over these setups behaves like the server time distribution. This is so because read threads are busy most of the time with no significant dead time, and the time taken to process a `get` request usually corresponds approximately to its server time, as read threads are blocking. Thus, queue time distribution is also not affected by changes in the replication factor. The small differences between different factors occur almost exclusively at the 99th percentile, and are due to a slight increase in memcached server load (which makes it slightly more unstable) and variations in the experiments.

One observes that, when varying number of servers for a fixed replication factor, `get` queue time decreases slightly from 3 to 5 servers, and then remains practically constant from 5 to 7 (see Figures 11 and 15). This can be explained by the fact that I chose the optimal configuration for 5 servers in Section 1. Thus, this configuration has enough threads to steadily process the incoming load with 5 or more thread pools. With 3 thread pools, the number of threads is thus lower than what it should be to ensure a steady dequeuing of requests. It is also clear that the `get` server time distribution seems to become slightly larger and more unstable, as 99th percentile increases for varying number of servers (see Figures 9 and 13). This is due to the increase in the number of middleware-server connections as the number of servers becomes larger: the network becomes more unstable, and server time distributions for both request types follows suit.

Note that the `set` queue time distribution is not affected by increasing the replication factor. This is so because write threads are asynchronous, and thus the dequeuing process is not influenced by the server time of `set` requests. On the other hand, `set` server time is

markedly more affected by the replication factor, mostly at the 99th percentile (see Figures 10 and 14). The explanation is natural: for larger replication factor, `set` server time depends on more servers and connections, and equals the time of the slowest server. Thus, the probability that one of the servers or connections experiences a drop in performance or quality increases significantly with the replication factor. This implies that `set` server time tends to be larger (as evidenced by Figure 14) and becomes more prone to bigger spikes for increasing replication factor (as evidenced by the 99th percentile of Figure 10). Furthermore, `set` server time is considerably and consistently more expensive and unstable than other operations for all setups, as evidenced by its statistics.

When one increases the number of servers but fixes the replication factor, `set` operations are slightly affected, albeit much less than in the case where replication factor increases. First, `set` queue time increases slightly and becomes more unstable for 7 servers, as evidenced by the 99th percentiles and standard deviation (see Figures 12 and 16). This is due to CPU contention and to there being few write threads. Second, `set` server time is also affected, mostly at the 90th and 99th percentiles (see Figures 10 and 14). This is due to the fact that, as the number of server increases, each replication factor implies writing a `set` request to more servers in absolute terms. This is also why the difference is slightly more noticeable for larger replication factor.

Putting together all the data, I conclude that server time for `set` requests is the operation which becomes more expensive in the middleware as the replication factor as the setup changes. All remaining operations are not significantly affected. Thus, Hypothesis 3 holds.

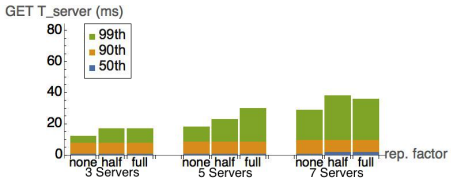

Figure 9: Percentiles of T_{server} of `get` requests for varying replication factor and number of servers.

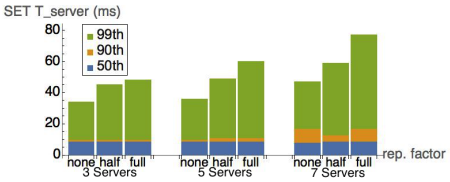

Figure 10: Percentiles of T_{server} of `set` requests for varying replication factor and number of servers.

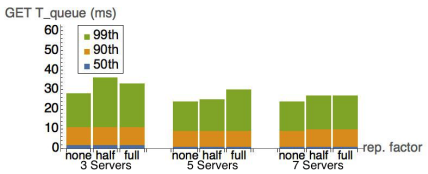

Figure 11: Percentiles of T_{queue} of `get` requests for varying replication factor and number of servers.

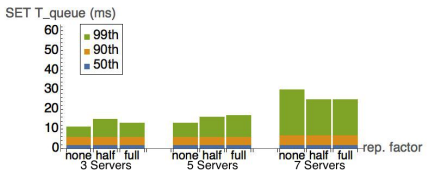

Figure 12: Percentiles of T_{queue} of `set` requests for varying replication factor and number of servers.

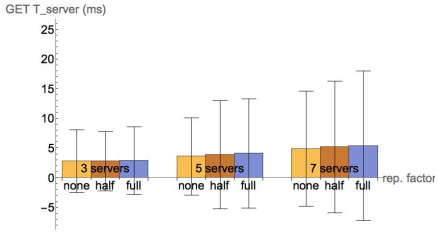

Figure 13: Average and standard deviation of T_{server} of `get` requests for varying replication factor and number of servers.

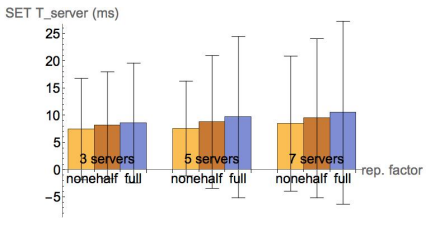

Figure 14: Average and standard deviation of T_{server} of `set` requests for varying replication factor and number of servers.

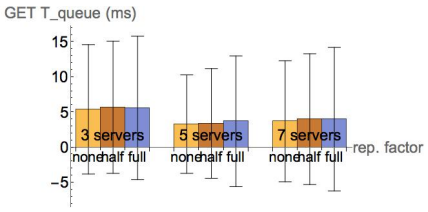

Figure 15: Average and standard deviation of T_{queue} of `get` requests for varying replication factor and number of servers.

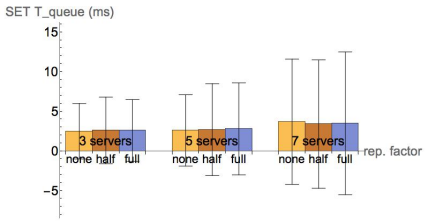

Figure 16: Average and standard deviation of T_{queue} of `set` requests for varying replication factor and number of servers.

Finally, the last objective is to compare the scaling of our system to that of an ideal implementation. In this experiment, as one increases the number of servers for a fixed replication factor (none, half, or full), the total request load assigned to each server decreases. This is straightforward to see for the "no replication" case. For half and full replication, one needs to be a bit more careful. Suppose there are S servers, T total requests, and half replication. Suppose also that `set` requests are 2 times more expensive than `get` requests, which is a pessimistic upper bound. Recall that `set` requests make up 5% of all requests. Each server receives approximately the same quantity of `get` requests and a $\lceil S/2 \rceil / S$ fraction of `set` requests. The request load per server is then

$$L_{\text{half}} = \frac{T(2 \times 0.05 \times \lceil S/2 \rceil + 0.95)}{S}.$$

Note that L_{half} decreases as S increases, i.e. server load decreases as the number of servers increases for half replication.

For full replication, the reasoning is analogous: `get` requests are uniformly distributed over servers and each server receives all `set` requests. The load per server in this case, L_{full}, is thus

$$L_{\text{full}} = 2 \times 0.05 \times T + \frac{0.95 \times T}{S},$$

which also clearly decreases for increasing S.

Thus, an ideal implementation would achieve higher total throughput for fixed replication (none, half, or full) and increasing number of servers, provided the load imposed on the system (i.e. virtual clients in memaslap) is constant throughout the experiments. In my case, though, the implementation showcases decreasing `get` and `set` throughput for increasing number of servers and fixed replication factor, as can be evidenced by Tables 7 and 8. As discussed previously, the relative decrease is very similar for both request types, so the relative decrease in total throughput is also similar: one experiences roughly a 10% decrease in throughput every time the number of servers is increased, for all replication factors. I conclude then that my system behaves quite differently from an ideal implementation. This is due to the fact that, in a real implementation, increasing the number of servers also sharply increases the number of connections between the middleware and the servers. Thus, there is a marked increase in network latency, which affects performance. Furthermore, in reality there are finite resources that can be allocated to the middleware. Thus, the system has to deal with CPU contention: as the number of servers increases so does the number of threads, and thus each thread is allocated less resources, which slows down its performance.

Furthermore, in an ideal implementation, servers would behave in a consistent manner. This means that server time would be constant over time for both request types. Thus, increasing the replication factor would not have any influence in the server time of `set` requests. Clearly, this does not hold in my system. Looking at Figure 10, one notices a marked increase in the 99th percentile of the server time distribution for `set` requests, for all numbers of servers. This means that the distribution becomes more unstable and tends to feature larger values. Thus, Hypothesis 4 holds.

3 Effect of Writes

The goal of this section is to study the changes in the system's performance as the proportion of `set` requests increases.

3.1 Hypotheses

My hypotheses, prior to running any experiments, are as follows:

1. As the write proportion increases for fixed number of servers and replication factor (none or full), there should be a decrease in total throughput and increase in response time achieved by the system. This is so because `set` requests are more expensive than `get` requests, as there are fewer threads dedicated to processing them (1 write thread per server) and they take longer in the server.

2. Increasing the write proportion should impact configurations with more servers and full replication more severely, since this entails writing each `set` request to more servers at once.

3. The major cause of the drop in performance when increasing write proportion should be `set` server time. This is so because the number of clients is fixed, write proportion is always relatively small and write threads are asynchronous. Thus, queue time should be unaffected by increasing write proportion. On the other hand, for full replication, increasing write proportion leads to more stress on the servers, as well as more requests whose processing times depend heavily on the performance of several servers and connections at once.

3.2 Experiment and discussion

In order to investigate the goals of this section, I ran the following experiment.

Number of servers	3, 5, 7 (Basic A2), servers restarted every run
Number of client machines	3 (Basic A2)
Middleware	Present (Basic A4)
Virtual clients / machine	66 ($3 \times 66 = 198 \approx 200$ total clients)
Threads in pool	16
Workload	Key 16B, Value 128B, Writes 1%, 5%, 10%
Overwrite (`-o`)	0.9
Replication factor	none, full
Logging ratio (`get`/`set`)	every 100/every 100
Runtime x repetitions	105s x 4

The raw memaslap and middleware logs are in `logs23-memaslap` and `logs23-mw`, respectively. The processed data is in `proc23-memaslap` and `proc23-mw`. I measure the 30s-90s period, like in Section 2. From personal experience, the logging ratio used is enough to provide meaningful and representative statistics of the relevant distributions. All averages, standard deviations, relative differences and statistics are computed like in Sections 1 and 2. Figures 17 through 22 showcase total throughput and average response time trends for the configurations considered in the experiment. Note that, similarly to Section 2, throughput decreases as one increases the number of servers and the replication factor. Interestingly, total throughput and response time are not affected by increasing the write proportion in the "no replication" case, which contradicts part of Hypothesis 1. This is explained as follows: when there is no replication, `set` requests are processed exactly like `get` requests, except that they must be directed to write threads. For each fixed number of servers, and since I use 16 read threads in each pool, the number of write threads is always a $1/16 \approx 0.0625$ fraction of the number of read threads.

This means that there are not much fewer write threads per `set` request than read threads per `get` request. Furthermore, write threads, unlike read threads, are asynchronous. Thus, even though `set` requests are more expensive on the server side than `get` requests (e.g. see Figures 27 and 29), write threads can keep sending `set` requests while waiting for responses. Combining these remarks, I conclude that write threads can process `set` requests at the same rate as read threads can process `get` requests when there is no replication involved and the write proportion is not much larger than the proportion between read threads and write threads.

In the full replication case, increasing the write proportion implies a decrease in throughput and an increase in average response time, as predicted in Hypothesis 1: a larger replication factor implies that `set` tend to be much more expensive at the server/network side, as their performance now depends on many more servers and connections over the network. I claim that configurations with 7 servers are the ones more severely impacted by variations in the write proportion when compared to the base cases. I define the base cases as follows: for each number of servers (3, 5, 7) there are two base cases, namely the setups with 1% write proportion. I then study performance reductions caused by increasing the write proportion against the respective base case: for example, for a configuration with S servers, full (no) replication and 5%/10% writes, I compare the relative difference in performance in relation to the base case with S servers, 1% writes and full (no) replication. Since write proportions do not affect "no replication" setups, it suffices to look at full replication. Specifically, I compare relative differences in throughput and response time when increasing the write proportion from 1 to 5 and from 1 to 10 for 3, 5 and 7 servers with full replication. Tables 15 and 16 showcase these relative differences. Given that relative differences are significantly greater for 7 servers setups in both tables, I conclude that such configurations are the ones more severely impacted by increasing the write proportion. Thus, Hypothesis 2 holds.

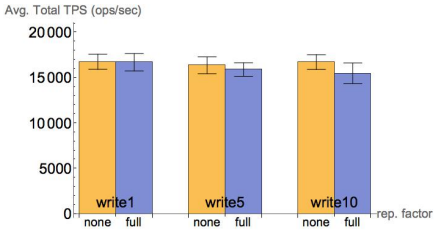

Figure 17: Average total throughput for varying write proportion and 3 servers.

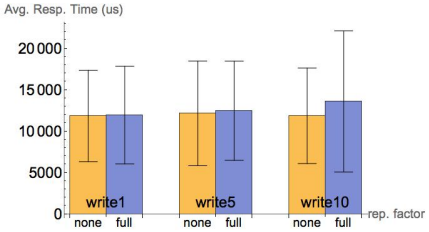

Figure 18: Average response time for varying write proportion and 3 servers.

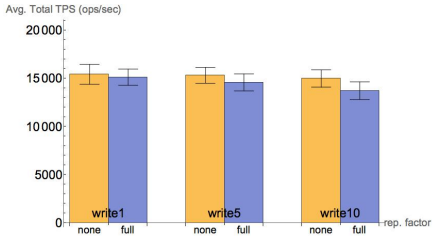

Figure 19: Average total throughput for varying write proportion and 5 servers.

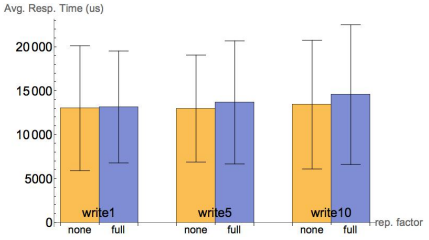

Figure 20: Average response time for varying write proportion and 5 servers.

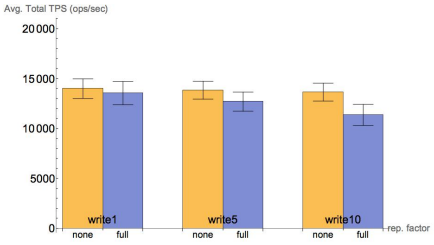

Figure 21: Average total throughput for varying write proportion and 7 servers.

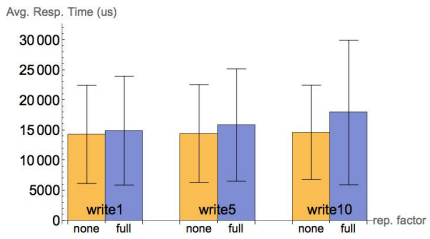

Figure 22: Average response time for varying write proportion and 7 servers.

nr. servers	1% to 5%	1% to 10%
3	−4.8%	−7.3%
5	−3.6%	−9.3%
7	−6.3%	−16.1%

Table 15: Relative differences in throughput for full replication and varying write proportion.

nr. servers	1% to 5%	1% to 10%
3	4.4%	13.9%
5	3.9%	10.6%
7	6.2%	20.2%

Table 16: Relative differences in average response time for full replication and varying write proportion.

In order to investigate the main reason behind the reduced performance observed before and to continue the detailed explanation of the behavior of the system, I turn to the middleware logs. More specifically, I compute percentiles for T_{queue} and T_{server} for both `get` and `set` requests for all configurations. As a first remark, note that every number of servers showcases similar trends for all time distributions studied, albeit at different scales, for varying write proportion and replication factor. For most parameters of interest, either they are not significantly affected by increasing the number of servers, or one notes more marked increases when varying the write proportion for larger numbers of servers, especially for full replication setups. This is because, in absolute terms, full replication implies writing to many more servers for larger numbers of servers.

One notices that the queue time distribution for `set` requests is not significantly affected by the change in write proportion nor by the change in replication factor, for a fixed number of servers. Also, these distributions seem to be very similar to queue time distributions of `get` requests. Differences observed in Figures 25 and 26 are all small and almost totally occur in the 99th percentile. Thus, differences between setups are explained by small variations between experiments. This confirms my hypothesis that queue time is not negatively affected by setup changes. As previously mentioned, this is due to the fact that write proportion is always small and also due to write threads being asynchronous. Therefore, a single write thread can keep dequeueing `set` requests while waiting for responses. The small write proportion means that one write thread suffices to dequeue requests fast enough so that the queue does not get backed up.

Regarding the `set` server time distribution in Figures 29 and 30, note that it is not significantly affected by varying the write proportion in the "no replication" case (Figure 29), although there are some slight increases in 99th percentile, and in 90th percentile for 7 servers. This is so because, even though memcached servers are under a heavier `set` load for larger write proportion, this load is still small and memcached servers are designed to handle large loads of requests with no significant drops in performance. The slight increases already mentioned are due to the server becoming slightly more unstable with an increased load of `set` requests. On the other hand, there is a noticeable increase of the 99th percentile for the full replication case (Figure 30), while 90th percentile increases slightly and 50th percentile stays constant. Also, for 5 and 7 servers, the full replication server time distribution features markedly larger 99th percentiles than the "no replication" server time distribution, which also means it is much more

unstable and susceptible to spikes. Marked increases for varying replication factor are observable for larger numbers of servers because these are the cases where going from no replication to full replication is a more severe change. The explanation is as follows: in a full replication setup, in contrast to the "no replication" case, the server time of each `set` request depends on the performance of all servers instead of only 1. If all servers behave normally, this does not make a big difference. Nevertheless, the probability that one of the servers and connections experiences some drop in performance/quality is much larger than for just 1 server and connection, especially for larger numbers of servers. Thus, spikes in server time become much more common in the full replication case, which leads to larger 99th percentiles compared to the "no replication" case. The increase of 99th percentile in Figure 30 happens because a much heavier `set` load is imposed on all servers and the connections between them when one increases the write proportion. For example, if one increases the write proportion from 1 to 10, this means that 10% of all requests are now written to all the servers, instead of just 1%. Note that when write proportion (and `set` load) increases, `get` load decreases. Nevertheless, `set` requests are more expensive than `get` requests, and so there is an overall increase in the load of the system (especially of the servers). This increase in load leads to spikes in server time being more common, as connections and servers are relatively more overwhelmed by requests. Thus, one experiences larger 99th percentiles as one increases the write proportion. This last argument cannot be applied to the "no replication" case (Figure 29) because, while `set` request load does increase, each request depends on only 1 server instead of all, and the load of each server is also much smaller than the corresponding load in the full replication case.

Server and queue time distributions for `get` requests are not significantly affected by varying replication factor for fixed write proportion and number of servers. This behavior was already explained in Section 2 for a 5% write proportion, and the argument carries over to the other proportions considered: basically, `get` requests still depend on the performance of only 1 server even when the replication factor increases. The mostly small differences between the "no replication" and full replication cases occur at the 99th percentile, and are due to the fact that the server is slightly more unstable with the added load of `set` requests. For the "no replication" case, as write proportion increases for a fixed number of servers, queue and server time are also practically unaffected, with some small increases at the 99th percentile for server time (due to increased `set` load on servers). Queue time is unaffected because, for `get` requests, it is heavily dependent on server time, as explained in Section 2. Since server time is not significantly affected, then the queue time distribution for `get` requests is similar over write proportions. In the full replication case, `get` server time observes some increases at the 99th percentile. This is so because, for increasing write proportion, the load imposed on each server grows considerably, as `set` requests are expensive and written to every server. Thus, servers become more unstable, and there occur more performance slowdowns, which translates into more and bigger server time spikes for `get` requests and larger server times in general. That the 99th percentile increases for the `get` queue time distribution for varying write proportion can be explained by variations of the network between experiments and by the increased unstability of the `get` server time distribution (as mentioned before, these two distributions are heavily related for `get` requests).

Notice that the server time for `set` requests (Figures 29 and 30) appears to dominate over the remaining time distributions considered. This domination is also markedly stronger in the full replication case. Furthermore, the times considered in all figures amount to virtually the total time spent in the middleware by `get` and `set` requests, as seen in Section 1. Thus, I conclude that the main reason for the reduction in performance observed in the full replication case as one increases the write proportion is the increase in server time, in particular for `set` requests, which confirms Hypothesis 3. This conclusion matches with the changes in total throughput and memaslap response time studied before. In fact, as one increases the write proportion in the full replication case, the system has to handle larger proportion of `set` requests, which are much more expensive than `get` requests, as they feature similar queue time distribution but

markedly more unstable and much larger server times, features which also increase with write proportion. Thus, the system is not able to process as many requests per second as before, and so throughput drops. Regarding response time, since the proportion of `set` requests increases and they are markedly more expensive, it leads to an increase in average response time. Standard deviation also tends to increase, as the response time distribution becomes more unstable, since `set` requests feature a significantly more unstable server time distribution.

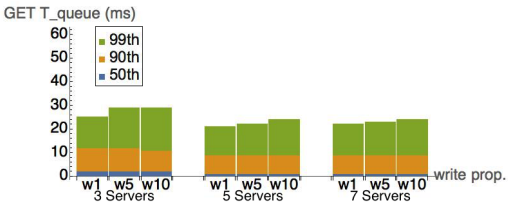

Figure 23: Percentiles of T_{queue} of `get` requests for varying write proportion and number of servers with no replication.

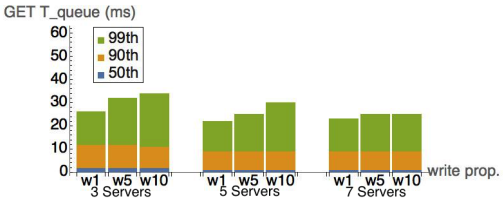

Figure 24: Percentiles of T_{queue} of `get` requests for varying write proportion and number of servers with full replication.

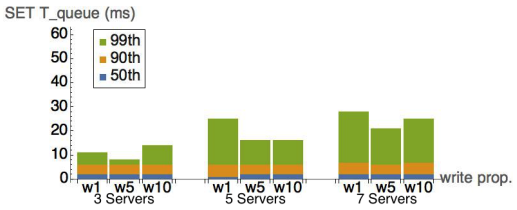

Figure 25: Percentiles of T_{queue} of `set` requests for varying write proportion and number of servers with no replication.

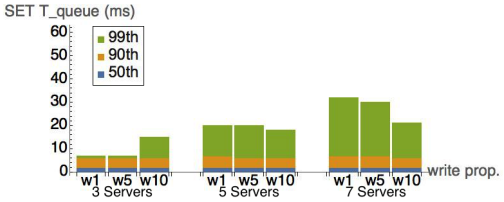

Figure 26: Percentiles of T_{queue} of `set` requests for varying write proportion and number of servers with full replication.

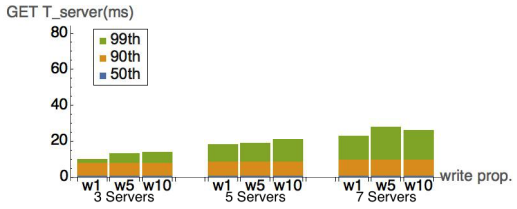

Figure 27: Percentiles of T_{server} of `get` requests for varying write proportion and number of servers with no replication.

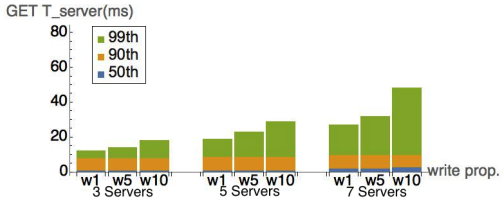

Figure 28: Percentiles of T_{server} of `get` requests for varying write proportion and number of servers with full replication.

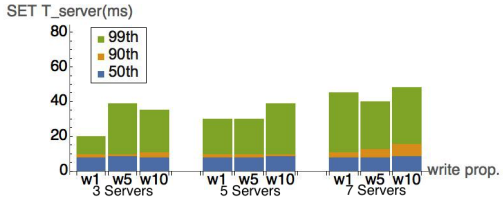

Figure 29: Percentiles of T_{server} of `set` requests for varying write proportion and number of servers with no replication.

Figure 30: Percentiles of T_{server} of `set` requests for varying write proportion and number of servers with full replication.

To finalize the detailed description of the behavior of the system, I present percentiles of T_{total} for both request types over all the setups considered (see Figures 31 through 34), and

briefly discuss the observed values. First, note that `get` total time is not significantly affected in the "no replication" case (Figure 31), as expected. The increases seen at the 99th percentile are due to the increase in server time (Figure 27). In the full replication case, one sees a more marked increase of the 99th percentile, due to the bigger increase in server time for `get` requests in this case (Figure 28). That total time for `set` requests increases in the "no replication" case (Figure 29) has a natural explanation: the write proportion increases, while write threads and number of servers do not. Thus, one gets a larger load of `set` requests per write thread and server, as evidenced by Figures 25 and 29. In the full replication case, one observes small increases at the 50th and 99th percentile level when compared to the "no replication" case. The explanation for this is that server time for `set` requests goes up as replication increases (compare Figures 29 and 30). Finally, the increase in total time as the write proportion gets larger is due to the increase in both server time and queue time for `set` requests (see Figures 30 and 26).

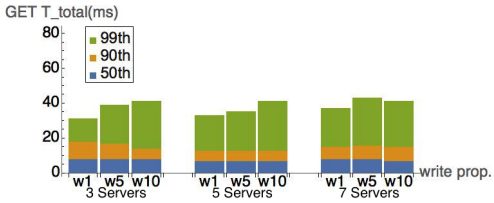

Figure 31: Percentiles of T_{total} of `get` requests for varying write proportion and number of servers with no replication.

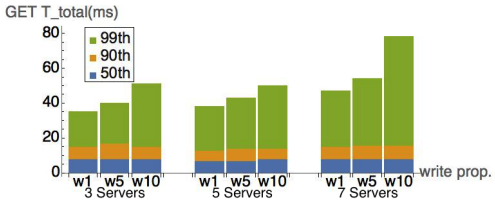

Figure 32: Percentiles of T_{total} of `get` requests for varying write proportion and number of servers with full replication.

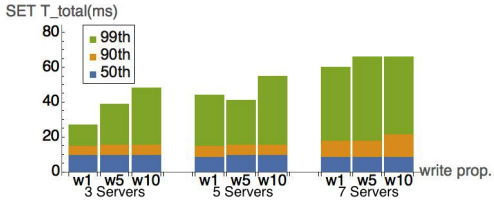

Figure 33: Percentiles of T_{total} of `set` requests for varying write proportion and number of servers with no replication.

Figure 34: Percentiles of T_{total} of `set` requests for varying write proportion and number of servers with full replication.

Logfile listing

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

Short name	Location
re-trace	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
proc21-memaslap	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
proc21-mw	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
logs21-memaslap	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/tree/master/M2
logs21-mw	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/tree/master/M2
fine-memaslap	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/tree/master/M2
fine-mw	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/tree/master/M2
fine-proc-memaslap	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
fine-proc-mw	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
proc22-memaslap-5	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
proc22-mw-5	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
logs22-memaslap-5	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
logs22-mw-5	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
proc22-memaslap-100	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
proc22-mw-100	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
logs22-memaslap-100	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
logs22-mw-100	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
proc23-memaslap	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
proc23-mw	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
logs23-memaslap	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2
logs23-mw	https://gitlab.inf.ethz.ch/ljoao/asl-fall16-project/blob/master/M2

Appendix: Code change

In this appendix, I detail the change in the code that I made for this milestone.

In order to withstand the large amount of write requests sent to the middleware at the start of experiments in Section 1 featuring large numbers of clients without memcached errors and partial reads, I felt the need to change the size of the receive and send buffers of the sockets connecting write threads to their respective servers, as well as the size of the buffer I was using to read from such sockets. There are no modifications in the code of the middleware besides this (in particular, no design choice was modified). The relevant code can be found in gitlab[1]. I took the liberty of re-running the 1 hour stability trace of Milestone 1 with the modified code and 20 second samples in memaslap. I keep this code for the all sections of this milestone. Figures 35 and 36 showcase throughput and response time in the stability trace. It is clear that the system is stable and that performance was not reduced compared to Milestone 1. The raw and processed logs for this stability trace can be found in `re-trace`.

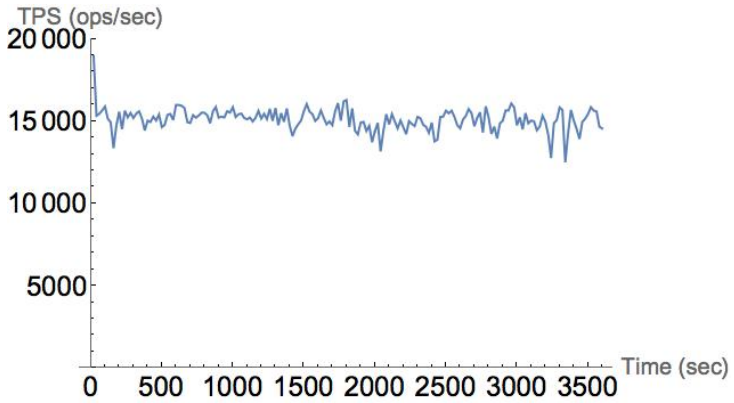

Figure 35: Throughput of stability trace re-run.

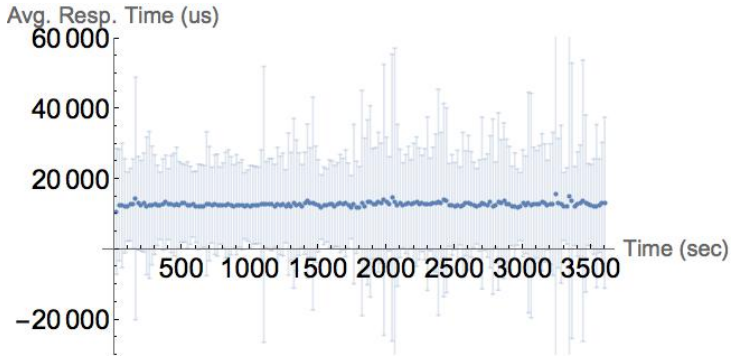

Figure 36: Average memaslap response time and standard deviation of stability trace re-run.

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