Advanced Systems Lab (Fall'16) – Second Milestone

Name: *Antonios Karkatsoulis*
Legi number: *15-925-928*

Grading

Section	**Points**
1	
2	
3	
Total	

1 Maximum Throughput

For the first part of this milestone we carried out a number of experiments in order to find out the minimum number of threads per thread pool, as well as the minimum number of clients that achieve the highest throughput in the presence of our middleware.

1.1 Hypotheses

For the first milestone we carried out some baseline experiments in order to find out the the number of clients that saturate a memcached server. This number was found to be around 100 clients. Since, in our setup, as we will describe later, 5 servers will be used, the expected number of total clients that saturate the whole system is going to be around 500. Of course this assumption is quite simplistic, since:

- The clients are not equally distributed across the servers, although our hash function achieves a nearly uniform distribution for the requests.
- The baseline experiments were carried out without the presence of the middleware, which is a serious bottleneck for the system.

Taking into consideration the above constraints, we thus expect the system to saturate at less than 500 clients.

As far as the breakdown of the time spent in the middleware is concerned we can hypothesise the following:

- More threads per thread pool will generally lead to less time spent in the queue for each request, since requests will be able to be served more quickly. However, too many threads will result in a lot of contention for the CPU and will eventually not be able to serve the requests so quickly resulting in a drop in performance.
- More clients will also result in bigger queue times, since more requests are going to be piling up inside the queues, waiting to be served.
- As far as the time spent in the server is concerned, we expect that it will remain almost the same with an increasing number of clients, since memcached is designed to handle a very big number of requests efficiently.

1.2 Experimental Setup

The experimental setup is described in the table below. With respect to the client configuration the only things that we modified from the Milestone 1 experiments were:

- The `-o` value of memaslap was set to value 1, so that we produce as less different keys as possible. This was chosen in order to avoid overwhelming the memcached servers with entries, that could lead to possible evictions and therefore GET misses later on.
- The window size for memaslap was set to 1k objects, exactly for the same reason. The default window size that is 10k objects produces way too many SET requests at the warmup period.

Server Machines	5 X Basic A2 (2 cores, 3.5 GB memory)
Client Machines	5 X Basic A2 (2 cores, 3.5 GB memory)
Middleware Machine	Basic A4 (8 cores, 14 GB memory)
Virtual clients / machine	10 to 100 with steps of 10 / machine
Threads per pool	8, 16, 24, 32, 40
Workload	Key 16B, Value 128B, GETs only
Runtime x repetitions	30 sec x 3
Overwrite proportion	1
Replication Factor	1
Memaslap window size	1k
Log Files Folder	1_max_throughput

Table 1: Setup for the Maximum Throughput experiment

1.3 Experiment runs

The setup for this experiment is listed in Table 1. A read-only workload was used, which means that memaslap would first send a number of SETs and then try to retrieve the keys that it stored. By carrying out some small experiments we found out that the time for these SETs, which will be referred as "warm-up" period can take up to 40 seconds, depending on the number of clients. In order to not take this period into account, we ran each experiment for 3 minutes and started measuring from the first minute onwards. We then split the next 1.5 minute to 3 parts which we regard as separate experiments. Although carrying separate experiments might have been more accurate, it would also be very costly. With our approach we get a good enough approximation, since we care about the trend and not specific numbers. We also cut off the last 30 seconds, in order to avoid measuring in the cool-down period where some client machines might complete before the others, leading to an increase in the throughput of the clients that remain.

As far as the experiments themselves are concerned, we varied the number of threads in the thread pool from 8 up to 40 in steps of 8, and the clients per machine from 10 up to 100 in steps of 10. We noticed that after 100 clients per machine, meaning 500 clients in total, the throughput did not increase or sometimes dropped, so we decided not to increase the number of clients any further. In Figure 1 we present the average and standard deviation of the throughput measurement over the 3 30-second experiments that we took into account. The throughput was computed as the average over the runs of the average aggregated throughput of all the client machines. The standard deviation was computed as $\sqrt{(s1^2 + s2^2 + s3^2)}$ where $s1, s2, s3$ are the standard deviations of the runs. Since it is requested that we find the maximum throughput with a granularity of 10 clients, we then carried out some more experiments close to the maximum value increasing the number of clients per machine by 2 at a time so that we increase by 10 in total.

1.4 Results

From Figure 1 we can deduce that maximum throughput is achieved for 32 and 40 threads at about 250 clients. In order to choose the best value for the number of threads we also had to take into account the response time. Unfortunately, looking at the memaslap logs was not enough to decide, since the percentile values that we are interested in are the same for 32 and 40 threads (16.8 ms, 32.76 ms and 65.53 ms for median 95th and 99th percentile respectively). We thus looked into the middleware logs and calculated the average time a request spends in the middleware, as well as the standard deviation. For 32 threads the value was 11.4 ± 20 ms, while for 40 threads 8.5 ± 12.4 ms, which means that the system performs better for 40 threads. The time percentiles follow the same trend, although the 50th and 95th percentiles are very close. The 99th percentile for 32 threads is 72.67 ms, while for 40 threads 54.63 ms. Since the

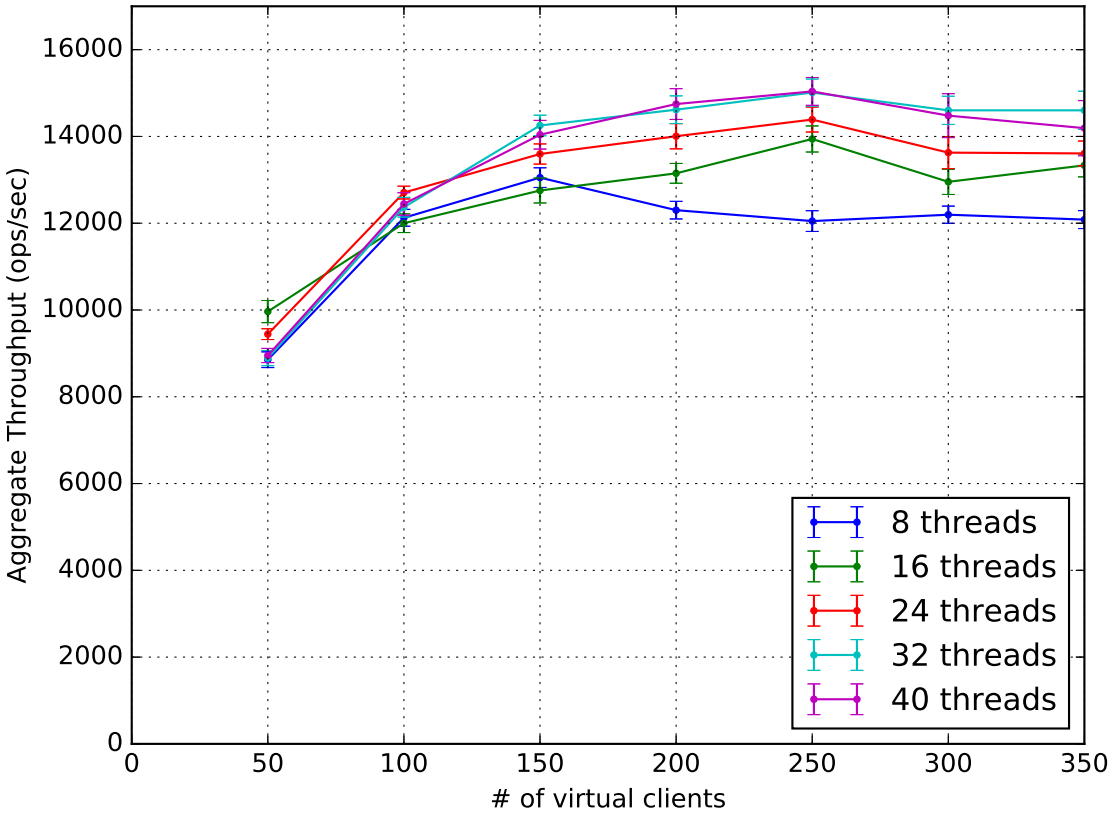

Figure 1: Aggregate throughput for varying number of clients and threads.

throughput already has reached its maximum for 32 threads and with 40 threads we managed to reduce the response time by about 25 %, we do not need to check for more threads that will consume more middleware resources.

What we also notice, is that for 8 and 16 threads the performance is really slow , due to the fact that there are not enough threads to consume requests fast enough from the queues.

The results of the finer grained search around 250 clients that specify the optimum number of clients are presented in Table 2. In order to obtain them, exactly the same experimental setup as before was used. While trying to choose the optimal configuration, we noticed that the throughput value is very similar for all the configurations and so we had to look into the times the requests spend in the middleware (T_{total}). We can see that for 250 clients, the throughput is the second highest and also with the smallest standard deviation and moreover the times spent in the middleware are quite low. We thus choose 250 clients at 40 threads as our optimal configuration.

Nr. Clients	Aggr. TPS (ops/sec)	TPS stdev	median $T_{total}(ms)$	95th percentile $T_{total}(ms)$	99th percentile $T_{total}(ms)$
230	14844	±447.3	7.43	18.47	55.81
240	14971	±333.0	7.37	18.59	52.81
250	14938	±308.2	7.40	18.51	54.63
260	14933	±369.4	7.40	18.41	53.42
270	14522	±333.9	7.55	20.14	61.99

Table 2: Fine-grained experiments near 250 clients that correspond to the maximum throughput (40 threads per thread pool).

We should note here that the numbers presented in Table 1 may be slightly different from the values in Figure 1, since the experiments were carried out at different times of the day, which means that the network overhead and probably the physical machines assigned were different.

Another remark is that, in order to make our decision we used the times for each request, as measured by the middleware and not from the memaslap logs. We made this decision, because the memaslap logs give us very coarse-grained estimates of the percentiles of the response times, which are not enough to decide on which configuration is better. As a proof of concept Table 3 presents the numbers for the response times for the fine-grained experiment as described before but taken from the memaslap logs. Of course, the times are larger than the middleware ones, as they include one more level of the network (from the clients to the middleware), but the important thing is that they are mostly the same.

Nr. Clients	median $T_{total}(ms)$	95th percentile $T_{total}(ms)$	99th percentile $T_{total}(ms)$
230	16.38	32.77	65.54
240	16.38	32.77	65.54
250	16.38	32.77	65.54
260	16.38	32.77	65.54
270	16.38	32.77	65.54

Table 3: Fine-grained experiments near 250 clients that correspond to the maximum throughput (40 threads per thread pool) taken from the memaslap logs (Approximation).

1.5 Breakdown of time per operation

In Table 4 we present a breakdown of the times the GET requests spend in different parts of the system, that is : total time spent in the middleware from the moment a request is received to the time the response is sent back to the client (T_{total}), time spent in the queue (T_{queue}) and time spent in the server (T_{server}). We get our results from the middleware logs and present the median , 95th percentile and 99th percentile values, as well as the average and standard deviation. The times presented correspond to the chosen configuration of 250 clients and 40 threads. We note here that we do not present the time for parsing the requests, decoding the operation and hashing, that corresponds to T_{total} - T_{server} - T_{queue}, since it is negligible. Since in our experiments we cut off 1 minute for warm-up and 30 seconds for cool-down period, we tried to do the same with the middleware logs, by cutting off the first third and the last sixth of each log, so that the numbers we present are as accurate as possible.

Time	Median (ms)	95th p. (ms)	99th p. (ms)	Average (ms)	Stdev (ms)
T_{total}	8.19	21.44	87.36	10.13	15.29
T_{queue}	1.24	8.43	18.80	3.03	6.98
T_{server}	6.89	16.19	67.31	7.01	13.34

Table 4: Breakdown of time spent in the middleware for the chosen configuration (250 clients, 40 threads)

We notice that GET requests spend in average almost 70 % of their time being processed by the server and about 30 % in the queue. However, we should note that the time distribution has a long tail, as can be seen by the percentiles, and so the average times are not really representative. The percentiles however confirm that most of the time is spent inside the server.

2 Effect of Replication

For the second part of this milestone we explored how the feature of replication affects the performance of our system. We varied the number of memcached servers and the replication factor and measured the throughput and response time of our system.

2.1 Hypotheses

Before we carry out the experiments for this part, we can hypothesise the following:

- SET operations are expected to be more expensive that GET operations, not only because they take longer in the memcached servers, but also because, although SET threads are asynchronous, we only use one thread per server to handle them.
- As more servers are added to the system, the throughput is expected to increase, since every server will get less load, through our load balancing mechanism. This however assumes an ideal situation, that will be discussed later on.
- GET and SET requests are not expected to be impacted the same way as the replication factor and the number of servers increase. First of all, GET requests are not expected to be impacted at all by the replication factor, since each GET request is only forwarded to one server. However, due to the operation of memaslap, as it will be explained later, this hypothesis is going to be disproved.
- As far as SET requests are concerned, a bigger replication factor should imply a lower throughput and higher response time, because every SET request now has to wait for the slowest of the respective replication servers to process it.

2.2 Comparison with ideal system

We now compare our real system with an ideal system. We define an ideal system as a system where the network has constant behaviour in the sense of latency, the servers can handle arbitrarily large loads of requests and respond back at a constant time. Finally, the middleware machine would have infinite CPU resources and the performance of each thread would be independent of the other threads.

With a constant number of clients, as is the case in our experiment, adding more servers to the system would result in higher throughput. More specifically for 5 % SET operations and 95 % GET operations and for a total throughput of T, S servers and full replication, the load per server would be $0.05 * T + 0.95 * T/S$. This means that an increasing number of servers

will lead to higher throughput. The response time in an ideal system would be constant for an increasing number of servers since the latency of the network would be constant and even with full replication all the ideal servers would reply at exactly the same time.

In the non-ideal system that we carry our experiments on, the latency of the network is an important factor of the response time of every request. Moreover, as replication factor and thus the number of requests towards the servers increases, this latency will be playing an ever-increasing role. Finally, our non-ideal middleware has limited CPU resources, so with more servers, more threads contend for CPU time, requests take more time to be dequeued, leading to a drop in throughput for both SETs and GETs.

2.3 Experiments

Table 5 describes the experimental setup for the replication experiment. As we noted above, we varied the number of memcached servers, as well as the replication factor, while keeping the number of threads per thread pool and the number of clients fixed to the configuration that we chose in the first part. In particular we run for 82 clients per machine, that leads to 246 clients in total and 40 threads per thread pool.

Server Machines	3,5 and 7 X Basic A2 (2 cores, 3.5 GB memory)
Client Machines	3 X Basic A2 (2 cores, 3.5 GB memory)
Middleware Machine	Basic A4 (8 cores, 14 GB memory)
Virtual clients / machine	82
Threads per pool	40
Workload	Key 16B, Value 128B, 5% writes
Runtime x repetitions	3 minutes x 5
Replication Factor	1, half servers, full
Overwrite proportion	1
Memaslap window size	10k
Log Folder	2_replication

Table 5: Setup for the Replication experiment.

2.4 Results

Below we present the aggregate throughput achieved, as the number of servers and the replication factor varies. The throughput measurements for the GET and SET requests are presented in Figures 2 and 3. The way the above statistic metrics were derived is: We first exclude the first minute of every experiment as warm-up period, as well as the last minute as cool-down, for reasons similar to the first experiment. Then we aggregate the throughput measurements over the client machines and average over the 5 runs of each experiment. For the standard deviations presented, we compute from memaslap the standard deviation within each experiment over every second that memaslap logs and finally we take the average of the standard deviations over all the runs of the experiment. We thus get a quite good approximation of the standard deviation.

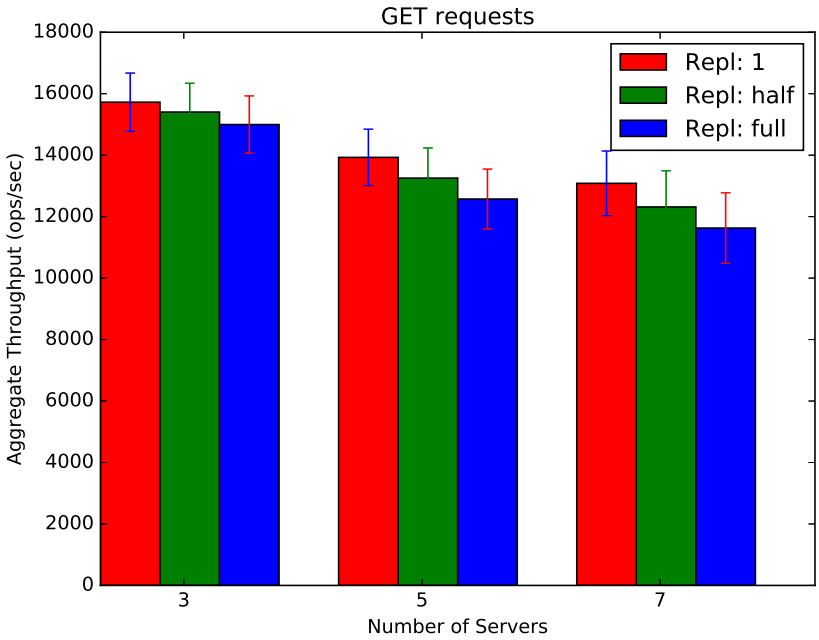

Figure 2: Aggregate throughput of 250 clients, for GET requests and for varying number of servers and replication factor.

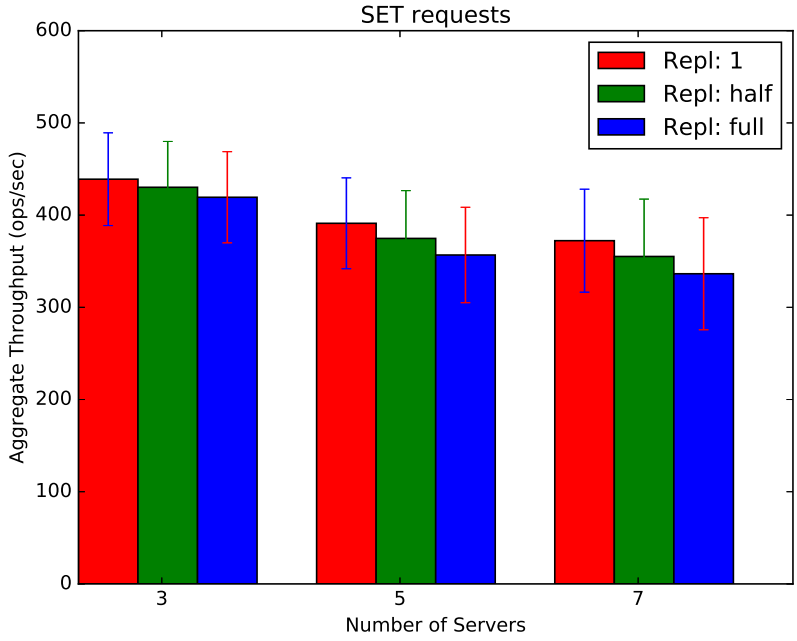

Figure 3: Aggregate throughput of 250 clients, for SET requests and for varying number of servers and replication factor.

In order to find out if GET and SET operations are impacted the same way as the configuration changes, we present the throughput results from Figures 2 and 3 in Table 6, along with the relative change of the throughput as the replication factor increases. We notice that the SET and GET requests are affected almost equally, a result that will be discussed later on.

(Servers, Replication)	GET Aggregate TPS (ops/sec)	GET TPS relative change	SET Aggregate TPS (ops/sec)	SET TPS relative change
(3,1)	15723 ±948	-	439 ±50	-
(3,2)	15402 ±939	-2.05 %	430±50	-2.05 %
(3,3)	14996 ±930	-2.56 %	419±49	-2.56 %
(5,1)	13928 ±918	-	391±49	-
(5,3)	13254 ±980	-4.84 %	375±51	-4.09 %
(5,5)	12573±974	-5.14 %	357 ±51	-4.8 %
(7,1)	13083 ±1051	-	372 ±56	-
(7,4)	12315 ±1175	-5.87 %	355 ±62	-5.57 %
(7,7)	11629 ±1145	-5.57 %	336±61	-5.35%

Table 6: Aggregate throughput for the GET and SET requests, along with the relative percentage changes as the replication factor grows for each number of servers.

For completeness reasons we also present the statistics for the response time from the memaslap logs. We can see that they follow the opposite trend from the throughput, something that also confirms the expected behaviour. We can also notice that as the number of servers and replication factor increases, the standard deviations of the response times get bigger, since more uncertainty is introduced by the network. Unfortunately, the statistics that memaslap outputs regarding the response time distributions are very coarse-grained and we cannot use the average response time in our arguments, since we know that the distribution has a long tail. Therefore, we have to resort to the throughput measurements for our discussion.

(Servers, Replication)	Median (ms)	95th p. (ms)	99th p. (ms)	Average (ms)	Average relative change	Stdev (ms)
(3,1)	32.77	65.54	131	14.88	-	1.97
(3,2)	32.77	65.54	131	15.17	-1.9 %	1.98
(3,3)	32.77	65.54	131	15.59	-2.8 %	2.07
(5,1)	32.77	65.54	262	16.91	-	2.59
(5,3)	32.77	65.54	262	17.84	-5.5 %	2.96
(5,5)	32.77	65.54	262	18.89	-5.9 %	3.88
(7,1)	32.77	65.54	262	18.30	-	4.33
(7,4)	32.77	65.54	262	19.69	-7.5 %	5.20
(7,7)	32.77	65.54	262	20.48	-4 %	6.34

Table 7: Percentile distributions, average and standard deviation of the response times for GET requests, as measured by memaslap.

(Servers, Replication)	Median (ms)	95th p. (ms)	99th p. (ms)	Average (ms)	Average relative change	Stdev (ms)
(3,1)	32.77	65.54	262	19.93	-	3.15
(3,2)	32.77	65.54	262	20.67	- 3.7 %	3.24
(3,3)	32.77	65.54	262	21.47	-3.9 %	3.35
(5,1)	32.77	65.54	262	21.30	-	3.34
(5,3)	32.77	65.54	262	22.90	-7.5 %	4.60
(5,5)	32.77	65.54	262	24.45	-6.7 %	5.96
(7,1)	32.77	65.54	262	22.66	-	7.18
(7,4)	32.77	65.54	262	25.84	-14 %	8.22
(7,7)	32.77	65.54	262	28.51	-10.3 %	6.86

Table 8: Percentile distributions, average and standard deviation of the response times for SET requests, as measured by memaslap.

Our main conclusions from the above analysis are:

- For a fixed replication factor, as the number of servers increases, the throughput decreases which contrasts our hypothesis. This happens because our middleware implementation is not ideal and so, more servers imply more processing threads that have to compete for the limited CPU resources, eventually negatively impacting the performance. Even worse, although GET threads sleep between serving the requests, we have implemented our SET threads so that they spin in a `while` loop serving requests making them very expensive. Moreover, more servers imply more connections to them and so the unstable network factor becomes increasingly more important.

- As the replication factor increases, the throughput for the SET requests decreases, since a lot more SET requests are generated towards the memcached servers and for every request, the middleware has to wait for the slowest response from all the replication servers to come back.

- Although the throughput of the GET operations (see Fig. 2 and Fig. 3) should normally not be affected by the replication factor, we notice a linear decrease, similar to that of the SET operations. This happens due to the way memaslap clients send their requests. Memaslap sends blocks of SET requests and only after the SETs have finished do GET requests get issued, before the next block of SETs starts again and on and on. This means that the performance of the GET requests is tightly coupled to that of the SET requests. In an experimental setup similar to the first experiment, where all the SET requests would be issued as a batch in the beginning and then only GET requests would be issued, the replication factor would not affect the performance of the GET requests and confirm our hypothesis.

- From Tables 7 and 8 we can see that the response times naturally follow the opposite trend from the throughput, increasing as the number of servers and the replication factor increases. Of course the SET requests take on average more time to complete. Moreover, from the average response time measurements we can conclude that SET response times get impacted more as the replication factor increases. This is natural, since for larger replication factors, each SET request has to wait for more servers to reply. We also notice a slight increase in the GET response times, which can be attributed to the fact that as

the replication factor grows, SETs impose a bigger load on the servers, implicitly affecting also the performance for the GETs.

We will now investigate which operations become more expensive inside the middleware, as the replication factor and the number of server varies. For this reason we compute and plot the median, 95th and 99th percentile for the server time (T_{server}) and queue time (T_{queue}) for the different combinations of server and replication factor.

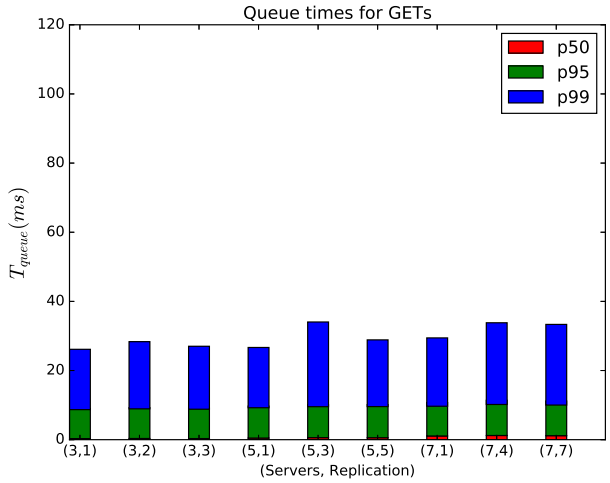

Figure 4: Percentiles of the time spent in the queue for GET requests for varying servers and replication factor.

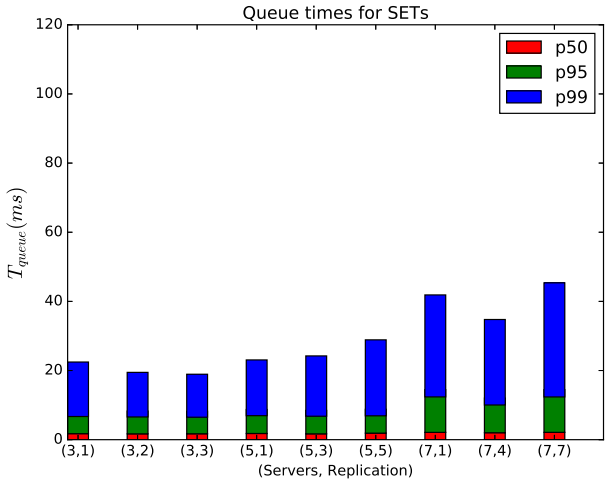

Figure 5: Percentiles of the time spent in the queue for GET requests for varying servers and replication factor.

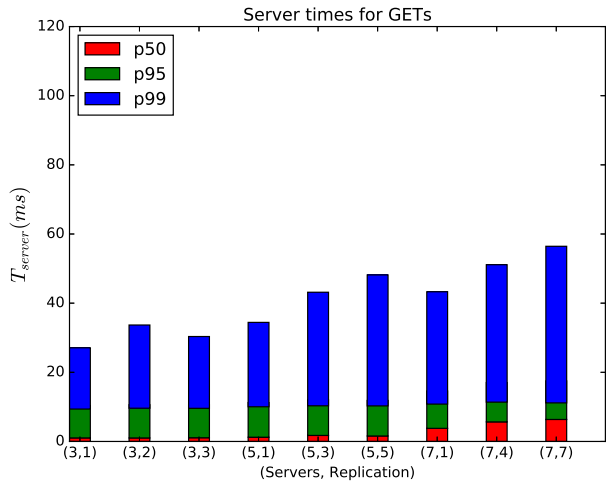

Figure 6: Percentiles of the time spent in the server for GET requests for varying servers and replication factor.

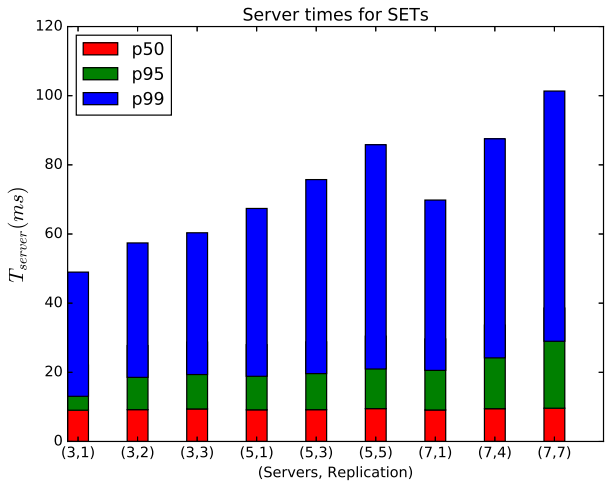

Figure 7: Percentiles of the time spent in the server for SET requests for varying servers and replication factor.

From Figures 4 to 7, we can conclude that it is the time requests spend in the server that makes the requests more expensive as the replication factor and the number of servers changes. Some explanations about the graphs:

- In Figure 4 we notice that the queue time for the GETs is not affected neither by the number of servers nor by the replication factor. Even if the replication factor increases each GET request is still sent to one memcached server so the GET requests continue being served at the same rate.
- From Figure 5 we can conclude that the queue time for the SETs is also not affected much, as the replication factor or the number of servers increases. Only for 7 servers we can notice a small increase in the queue time that can be attributed to the many more SET requests that are generated from the replication and which cannot be handled as fast by the single write thread, although it is asynchronous.
- Server times for the GET requests are affected as the replication factor increases (Fig. 6) but not by much. This can be explained like this: Higher replication factor may not be affecting GETs explicitly, but it imposes a higher load on the servers due to the increased number of SETs, eventually affecting the performance of the GETs as well.
- Server times for the SET requests are affected a lot as the replication factor increases (Fig. 7) . This is expected, since with higher replication factor, each SET operation generates a lot more SET requests towards the memcached servers and we have to wait for the slowest one in order to respond to the client. The more the servers, the higher the slowest response time will be with high probability.

3 Effect of Writes

For the last part of the milestone, we study the changes in throughput and response time of our system as the percentage of write operations increases.

3.1 Hypotheses

- For a fixed number of servers and replication factor, as the percentage of write operation increases, the throughput for the SET operations is expected to increase. Suppose we have a total throughput of x for a fixed number of servers, replication factor and 1 % writes. Then the throughput for the SET operations should be almost $0.01 * x$. Now if we increase the replication factor to, say 10 %, even if the total throughput falls a bit, the throughput for the SET operations would still be about $0.10 * x$, almost 10 times bigger. Since the total throughput is expected to fall however, because SET operations are costlier as we have shown, the throughput for the SETs will increase less than 10 times.

- For the GET operations, the throughput for fixed number of servers, fixed replication factor and increasing percentage of write operations is expected to fall, since the number of GET operations will be smaller. Of course, we expect it to fall by a little, since GET operations will still occupy the biggest proportion.

- As the number of servers or the replication factor increases the throughput for both GET and SET operations is expected to fall for the same reasons that we analysed in the previous chapter.

- For full replication and a fixed write proportion, adding more servers will have a bigger impact on the performance. This can be explained as follows: Full replication means that, with S servers, every server has to process all the SETs issued and almost the number of GETs issued divided by S, because of the load balancing. This means that increasing S, will result in the expensive SETs being a bigger percentage of the total load per server, thus impacting the performance more.

- For the no replication case, we do not expect the impact in performance to be that big, since SET requests now are not multiplied and even at 10 % they are not too many in number.

- The reason for the reduction in performance as the write proportion increases, is expected to be the server time, for the SET requests and with full replication, a bigger write proportion means more SET operations and therefore a lot more SET requests toward the servers. Performance for the GETs is not expected to be impacted since not only GETs don't matter in the replication case, but also they decrease by a little as the write proportion increases.

3.2 Experimental Setup

We vary the number of servers between 3, 5 and 7, the percentage of SET operations between 1 %, 5% and 10 % and the replication factor between 1 and all. In table 9, we present all the details of the experiments we carried out. The workload files for memaslap can be found in the folder `workloads/`[1].

[1] `https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/tree/master/workloads`

Server Machines	3,5 and 7 X Basic A2 (2 cores, 3.5 GB memory)
Client Machines	3 X Basic A2 (2 cores, 3.5 GB memory)
Middleware Machine	Basic A4 (8 cores, 14 GB memory)
Virtual clients / machine	82
Threads per pool	40
Workload	Key 16B, Value 128B, 1 %, 5%, 10% writes
Runtime x repetitions	2 minutes x 5
Replication Factor	1, full
Overwrite proportion	1
Memaslap window size	10k
Log Folder	3_writes

Table 9: Setup for the Effect of Writes experiment.

3.3 Results

Below we present the aggregate throughput measurements for the GET and SET operations, with no and full replication and as the number of servers and the proportion of write operations change.

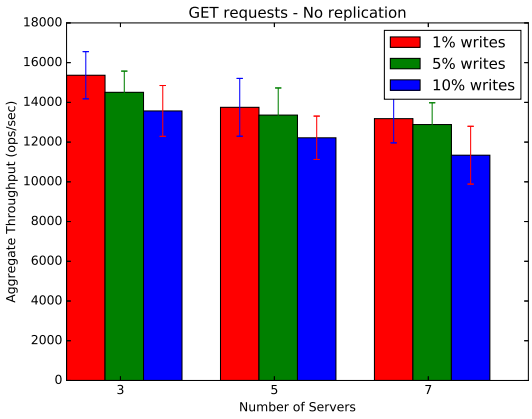

Figure 8: Effect of write proportion for GET requests and no replication.

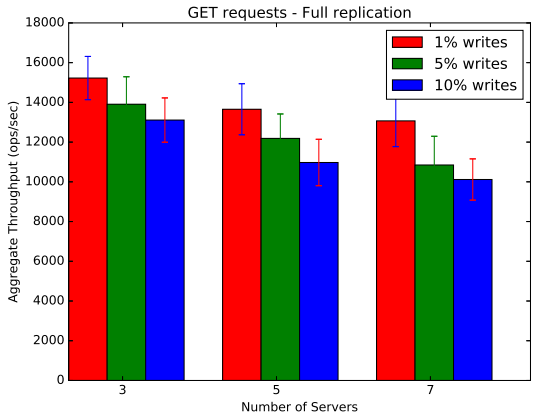

Figure 9: Effect of write proportion for GET requests and full replication.

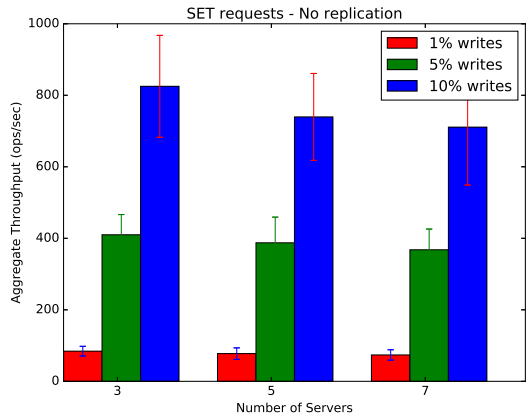

Figure 10: Effect of write proportion for SET requests and no replication.

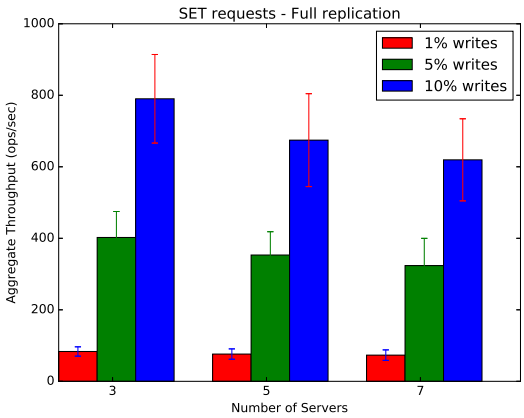

Figure 11: Effect of write proportion for SET requests and full replication.

Our obvious hypotheses are confirmed. The throughput for the GET operations drops as the number of servers and the proportion of writes increases, but not by much, since GET operations still make up the majority of the total operations. The throughput for the SET operations is impacted a lot more as the proportion of them increases, because SETs still make up a small proportion of the total operations.

In order to get a better overview of the effect of the proportion of writes we present the total throughput for both GETs and SETs , both for the replicated and the non-replicated case.

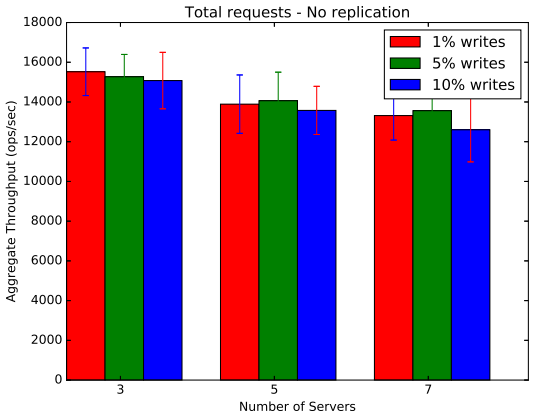

Figure 12: Effect of write proportion for all the requests and for no replication.

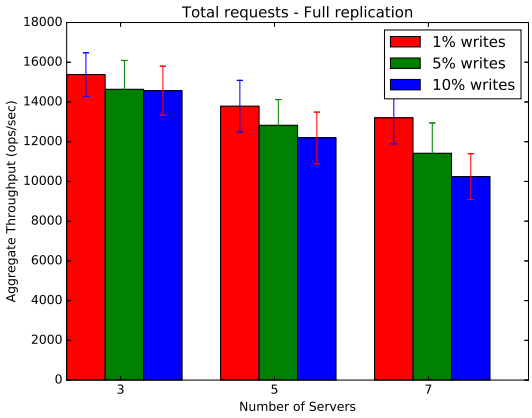

Figure 13: Effect of write proportion for all the requests and for full replication.

Looking at Figures 12 and 13 we can infer that, in general, replication has a heavier impact on the performance of the system, as the proportion of write operations increases. Table 10 shows the percentage differences in throughput as the proportion of writes increases for the different number of servers, for no and for full replication. We define the base case for our comparisons as the 1 % write proportion for 3 servers. We can conclude that for 7 servers and full replication we see the biggest impact in performance, confirming our hypothesis. As we mentioned before, more servers with full replication leads to an increased total load per server. We also notice that for no replication, the impact as the write proportion grows, is not that important. This can be attributed to the fact that without replication, not so many SET requests are generated, even

at 10 % write proportion.

# Servers	Replication	Write Proportion	TPS Percentage Difference
3	1	1 %	0 %
3	1	5 %	-1.62 %
3	1	10 %	- 2.86 %
3	3	1 %	-0.92 %
3	3	5 %	-5.68 %
3	3	10 %	-6.16 %
5	1	1 %	-10.51%
5	1	5 %	-9.39 %
5	1	10 %	-12.54 %
5	5	1 %	-11.15%
5	5	5 %	-17.35%
5	5	10 %	-21.45 %
7	1	1 %	-14.22 %
7	1	5 %	-12.62 %
7	1	10 %	-18.82 %
7	7	1 %	-14.95 %
7	7	5 %	-26.43 %
7	7	10 %	-34.02 %

Table 10: Throughput percentage differences from the base case (3,1%) for varying number of servers write proportion and replication.

In order to investigate the reason for this reduced performance we look into the logs from our middleware and plot the times they spent in the queue and in the server. We only take into account the configuration with 7 servers as this is the number of servers were the operations get impacted more.

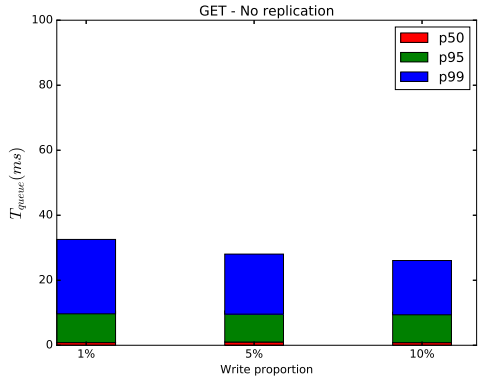

Figure 14: Percentiles for T_{queue} for GET requests, 7 servers, increasing write proportion and no replication.

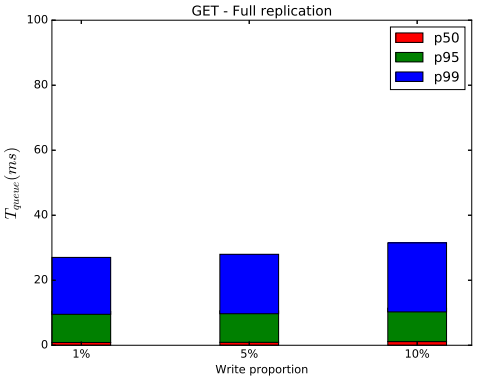

Figure 15: Percentiles for T_{queue} for GET requests, 7 servers, increasing write proportion and full replication.

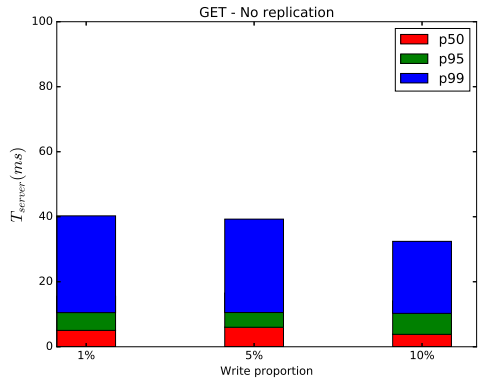

Figure 16: Percentiles for T_{server} for GET requests, 7 servers, increasing write proportion and no replication.

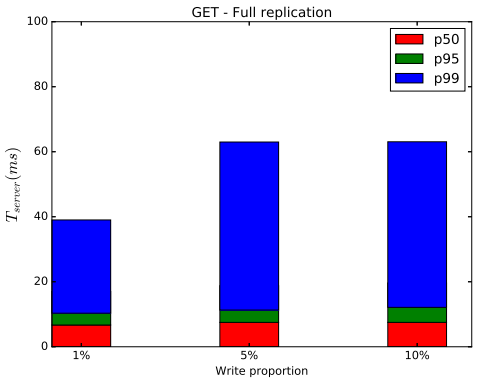

Figure 17: Percentiles for T_{server} for GET requests, 7 servers, increasing write proportion and full replication.

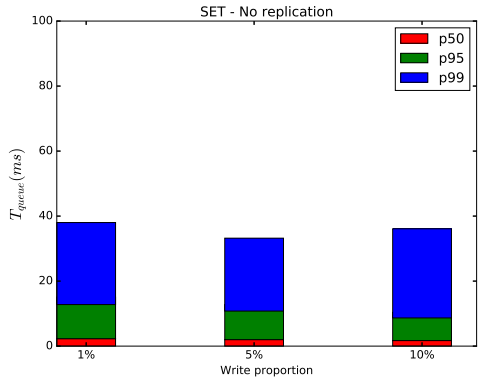

Figure 18: Percentiles for T_{queue} for SET requests, 7 servers, increasing write proportion and no replication.

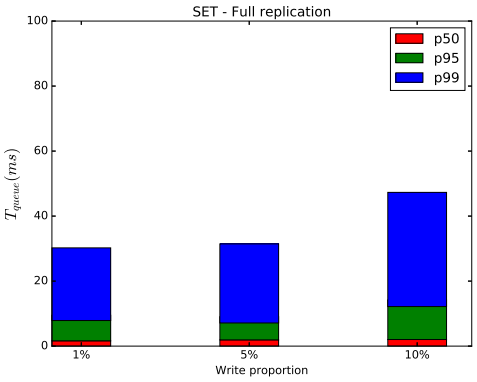

Figure 19: Percentiles for T_{queue} for SET requests, 7 servers, increasing write proportion and full replication.

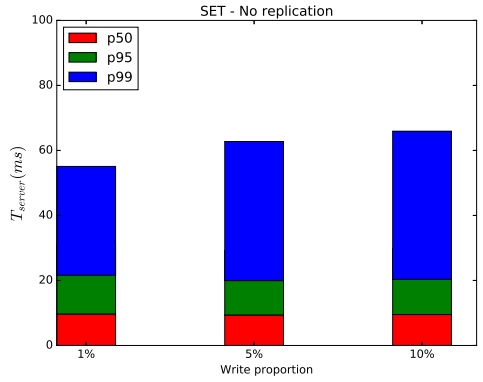

Figure 20: Percentiles for T_{server} for SET requests, 7 servers, increasing write proportion and no replication.

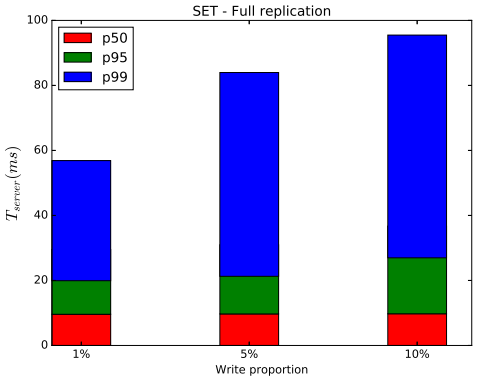

Figure 21: Percentiles for T_{server} for SET requests, 7 servers, increasing write proportion and full replication.

The time that the requests spend in the queues is almost constant for the SET operations (Figures 18 and 19) and similar to that of the GET operations (Figures 14 and 15). This happens because the proportion of writes, even at 10 % is still quite small, so the asynchronous write threads are still able to serve the SET requests at a rate where the load on the queues is kept at reasonable levels, even for the full replication case.

As far as the time spent in the servers is concerned, we notice from Figure 20 that with no replication the time spent in the server for the SET requests is not affected for increasing write proportion. This is expected, because each SET operation goes to one memcached server, that is optimized to process it very quickly (the total number of SET requests is still small). Moreover SET threads are asynchronous and able to handle the SET requests quite fast, despite them being more expensive inside the servers. However, for the replicated case (Figure 21) we see that the write proportion has a big impact on the server time. This happens because with 7 servers that we have now, one SET request has to wait for responses for 7 servers instead of 1, so there is a higher probability that one of them will take quite long to respond. So, as the proportion of writes increases, more requests have to be sent over the network and the load on every server is increased respective to the write proportion, leading to higher server times.

As far as the GET requests are concerned, we can see that the server and queue times are not much impacted by the increasing write proportion or replication factor. This can be explained as follows: each GET request is only sent to one server even for full replication, so both the queue and the server loads are not expected to rise much. Moreover, the write proportion increases, but only up to 10 %, which means that the proportion of the GET operations does not vary much.

Therefore, we come to the conclusion that the main reason for the reduced performance as the write proportion increases is the time requests and especially SETs spend being processed in the server, as we hypothesized.

Logfile listing

Short name	Location
1_max_throughput	https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/experiments_m2/1_max_throughput/memaslap_logs.tar.gz
2_replication	https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/experiments_m2/2_replication/replication_memaslap_logs.tar.gz
3_writes	https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/experiments_m2/3_writes/writes_memaslap_logs.tar.gz

Naming conventions

- 1_max_throughput : experiment_vc_[virtual clients]_t_[threads]
- 2_replication : experiment_sc_[number of servers]_r_[replication]
- 3_writes : experiment_sc_[number of servers]_r_[replication]_wr_[write proportion]

The middleware log of each experiment is stored in the respective folder as : `mw.log`.