Advanced Systems Lab (Fall'16) – Second Milestone

Name: ***Georgios Touloupas***
Legi number: ***16-932-550***

Grading

Section	Points
1	
2	
3	
Total	

1 Maximum Throughput

In this experiment we search for the highest throughput of our system for 5 memcached servers with no replication and a read-only workload configuration. We experiment with the number of virtual clients increasing their number from 20 to 400 in steps of 20. We use 5 memaslap clients with the same number of virtual clients (4 to 80 increasing in steps of 4). Simultaneously we experiment with the size of the read thread pools in the middleware corresponding to each memcached server. We repeat the experiments with 8 to 32 in steps of 8 threads per thread pool (for 5 thread pools).

Memaslap clients use the `smallvalue_readonly.cfg`[1] workload configuration (Key 16B, Value 128B, Writes 0%). When run with read-only workload, memaslap performs initially `win_size sets` per virtual client to populate the memcached table before performing any `gets`. In order to reduce the setup time, we reduce the `win_size` parameter to 1k (minimum value).

Each experiment for a specific number of virtual clients and thread pool threads runs for 150 seconds (aside from the initial setup time of the `sets`). As we demonstrated in milestone 1 the system is stable, therefore we can discard the first and last 30 seconds of the experiment as the warm up and cool down phases respectively and consider the 90 seconds left as 3 repetitions of 30 seconds each.

We use one Basic_A4 Azure machine for the middleware, 5 Basic_A2 Azure machines for the 5 memcached servers and 5 Basic_A2 Azure machines the 5 memaslap clients.

Number of servers	5
Number of client machines	5
Virtual clients / machine	seq(4, 80, 4)
Workload	Key: 16B, Value: 128B, Writes: 0%
Middleware	Replication: No, Threads: seq(8, 32, 8)
Runtime	150 sec
Log files	maxthroughput.zip

1.1 Choosing the Response Time Metric

Before exploring the experiment results for the maximum throughput configuration, we need to determine which metric and error should be used for the response time. Using the memaslap and middleware logs we plot the response time distributions for one of our experiments (260 clients, 24 threads) in Figure 1.

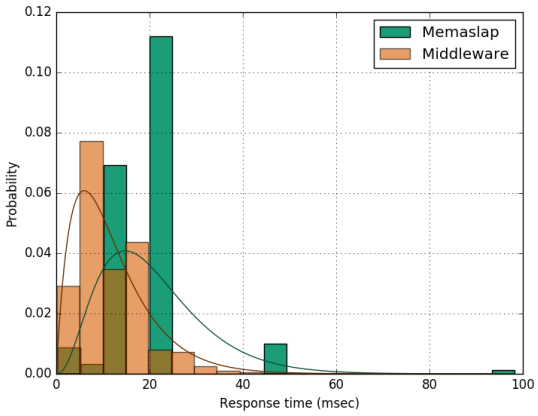

Figure 1: Memaslap and middleware response time distributions (260 clients, 24 threads)

[1] https://gitlab.inf.ethz.ch/gtouloup/asl-fall16-project/tree/master/workloads/smallvalue_readonly.cfg

As we can see, the response time reported from both memaslap clients and the middleware are skewed to the right (definitely not normally distributed) and can be approximated by the gamma distributions also plotted in Figure 1. For this reason **we will use percentiles instead of average and standard deviation as the metric for the response time (as well as queue time and server response time) throughout this report**.

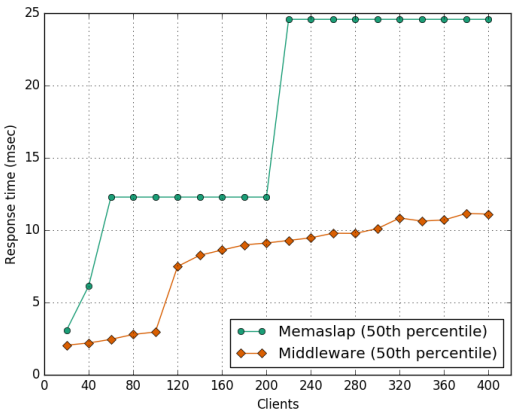

Figure 2: Memaslap vs middleware reported response time (24 threads)

In Figure 2 we observe that the response time metrics provided by memaslap are extremely coarse grained, since the bucket size for response time measurements used by memaslap is exponentially increasing (each bucket size is double the previous one), while the middleware reported response time (`tMW` metric) is free from this issue. Therefore, to obtain meaningful results **we will use the middleware reported instead of the memaslap reported response time throughout this report**.

1.2 Finding the Maximum Throughput

In order to determine the minimum number of threads and clients that together achieve the maximum throughput we look at the aggregate throughput plot for varying numbers of clients and threads in Figure 3.

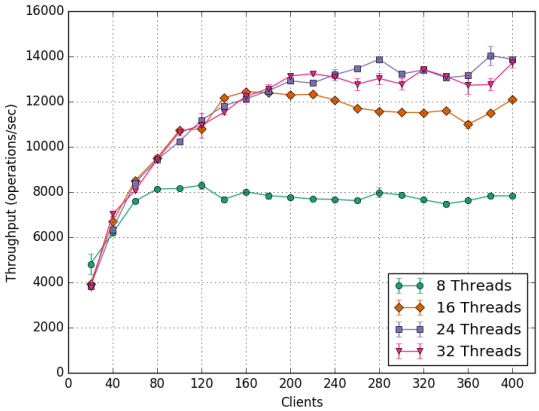

Figure 3: Aggregate throughput for different client and thread configurations

We see that, for any number of threads per read thread pool, by increasing the number of clients the aggregate throughput initially increases and eventually reaches a plateau, where

the system reaches saturation. For 24 and 32 threads the throughput plateau is around 13000-14000 operations/sec, while for 8 and 16 threads this plateau is lower (around 8000 and 12000 operations/sec respectively). This happens because the number of threads is no longer the bottleneck for over 24 threads.

Therefore our candidate thread configurations for the maximum throughput are 24 and 32 threads. We observe that for 24 threads the throughput rises for up to 280 clients to 13872 operations/sec and then fluctuates around the plateau. For 32 threads the throughput rises for up to 220 clients to 13235 operations/sec and then fluctuates around the plateau. In order to decide on the best configuration we also need to also look at the response time for 24 and 32 threads, presented in Figure 4.

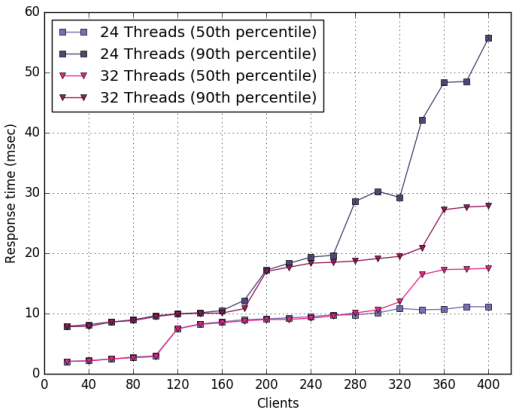

Figure 4: Response time percentiles for 24 and 32 threads

We can see that that for up to 260 clients the response time is roughly the same for 24 and 32 thread configurations, however at 280 clients 90$^{\text{th}}$ percentile for 24 threads rises from around 20 to 30 msec, denoting a substantial increase in the skewness of the response time. The reason for this increase is investigated in Subsection 1.3. Since we prefer the configuration with the less number of threads all other things equal, the choose the 24 threads for a configuration of under 280 clients because there is no significant difference in response time or throughput.

Taking all the aforementioned observations into consideration, we choose the **260 clients** and **24 threads** as the maximum throughput configuration, with an **aggregate throughput of 13467 operations/sec** and a **median response time of 9.789 msec** (19.676 msec 90$^{\text{th}}$ percentile).

To sum up, for up to 260 clients there is a notable increase in throughput, while above 260 clients there is a significant increase in the skewness of the response time. For under 24 threads the maximum throughput is lower, while more than 24 threads do not offer any performance advantage considering maximum throughput and response time at that configuration.

1.3 Middleware Internals Behaviour

To examine how the time spent in the middleware, presented in Figure 4, is broken down for 24 threads, we take a look at the time spent in the read queue (`tQueue` metric) and the time waiting for the memcached server response (`tServer` metric), illustrated in Figures 5 and 6 respectively.

In Figure 6 we see that the time waiting for the memcached server response is stable around 260 clients, whereas in Figure 5 there is an increase for the 90$^{\text{th}}$ percentile in the time spent in the read queue from 260 to 280 clients. Therefore that the increase we noticed in the total response time in Figure 4 is exclusively due to the increased time spent in the queue.

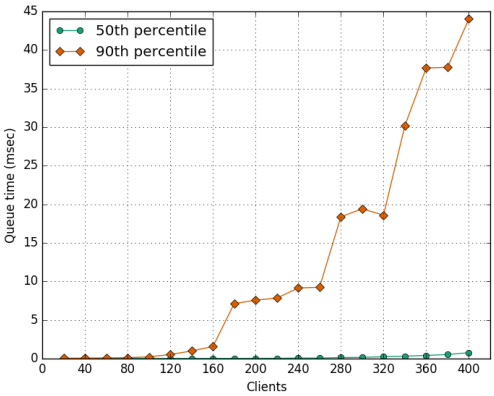

Figure 5: Queue time percentiles for 24 threads

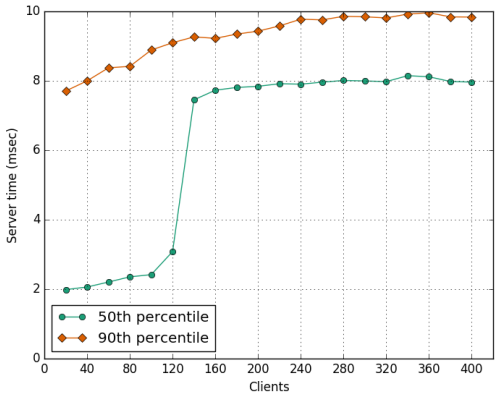

Figure 6: Server response time percentiles for 24 threads

By increasing the number of clients, the number of the requests in the middleware also increases, which means that the middleware will accumulate more requests in the queues, especially when reaching saturation. This is evident in Figure 5 where there is a continuous increase of queue time for the 90$^{\text{th}}$ percentile as we increase the number of clients.

Server time for reads should be approximately stable since it includes two times the network latency to the memcached server plus the time memcached processes the request. The sharp increase in median server time at 140 clients observed in Figure 6 could therefore be either due to the increase in memcached processes time or due to network congestion.

For the maximum throughput configuration with 260 clients and 24 threads we have a median response time of **9.789 msec**, a median time spent in the read queue of **0.065 msec** and a median time of waiting for the memcached server response of **7.961 msec**. Note that we cannot use the medians to extract any information for the rest of the operations in the middleware (hashing, forwarding the response to the client etc), since medians cannot be added or subtracted. The detailed breakdown of the time spend in the middleware is presented in Table 1.

Metric	**50$^{\text{th}}$ percentile**	**90$^{\text{th}}$ percentile**
`tMW`	9.789	19.676
`tQueue`	0.065	9.258
`tServer`	7.961	9.755

Table 1: Breakdown of the time spend in the middleware (measurements in msec)

2 Effect of Replication

In this experiment we explore how the behavior of our system changes for a 5% writes workload with 3, 5 and 7 memcached servers and the following three replication factors:

- Write to 1 (no replication)
- Write to $\lceil \frac{S}{2} \rceil$ (half replication)
- Write to all (full replication)

We use 3 memaslap clients with 87 virtual clients each (261 virtual clients in total) and 24 threads per read thread pool in the middleware. This configuration is chosen to be similar to the maximum throughput configuration we found in Section 1.

Memaslap clients use the `smallvalue_5.cfg`[2] workload configuration (Key 16B, Value 128B, Writes 5%).

Similarly to Section 1 each experiment runs for 150 seconds, we discard the first and last 30 seconds of the experiment as the warm up and cool down phases respectively and consider the 90 seconds left as 3 repetitions of 30 seconds each.

We use one Basic_A4 Azure machine for the middleware, 3 to 7 Basic_A2 Azure machines for the 3 to 7 memcached servers and 3 Basic_A2 Azure machines the 3 memaslap clients.

Number of servers	(3, 5, 7)
Number of client machines	3
Virtual clients / machine	87
Workload	Key: 16B, Value: 128B, Writes: 5%
Middleware	Replication: (No, Half, Full) , Threads: 24
Runtime	150 sec
Log files	replication.zip

We expect that an increase of the replication factor will negatively impact the performance of the system, since every `set` request will be sent to more memcached servers and all of the responses for a `set` request must be received before replying to the memaslap client.

2.1 Effect of Replication on Performance

Figures 9 and 10 show the aggregate throughput for `set` and `get` requests respectively for the replication experiments.

We note that the aggregate throughput has a similar behavior for `set` and `get` requests as the configuration changes. Throughput decreases when increasing the number of memcached servers. Throughput also decreases when increasing the replication factor (except for the configurations with 3 memcached servers).

To extract meaningful conclusions on the effect of replication on performance we also need to examine the response time, presented in Figures 9 and 10.

We see that the median of response time of `get` requests remains relatively stable for all configurations, while for the `set` requests it increases with the number of servers and with the replication factor. Therefore **`get` and `set` requests are not impacted the same way by different configurations**.

Increasing the number of memcached servers also increases the total number of read threads (72, 120 and 168 read threads in total for 3, 5 and 7 threads respectively). The overhead of managing a larger number of threads has an impact on the performance of the middleware

[2] https://gitlab.inf.ethz.ch/gtouloup/asl-fall16-project/tree/master/workloads/smallvalue_5.cfg

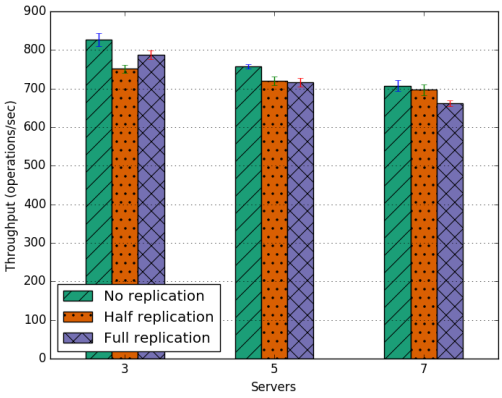

Figure 7: Aggregate throughput of sets

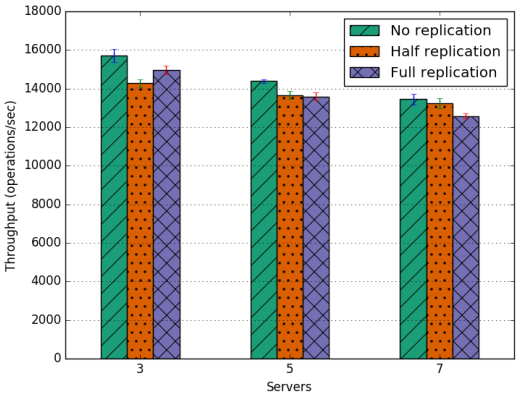

Figure 8: Aggregate throughput of gets

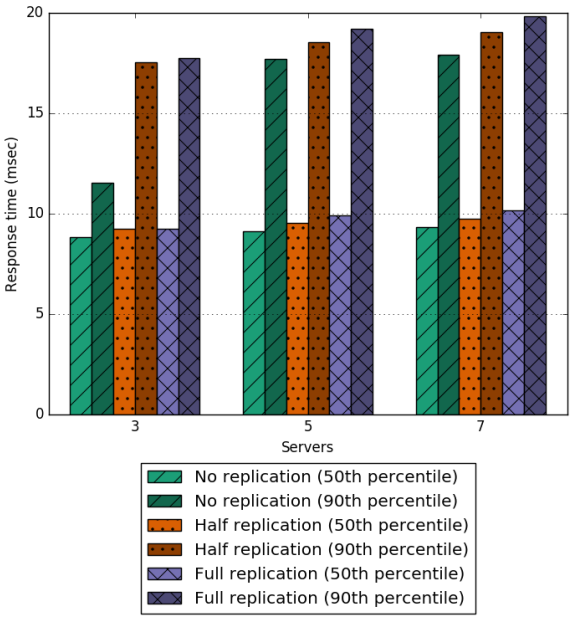

Figure 9: Response time percentiles for sets

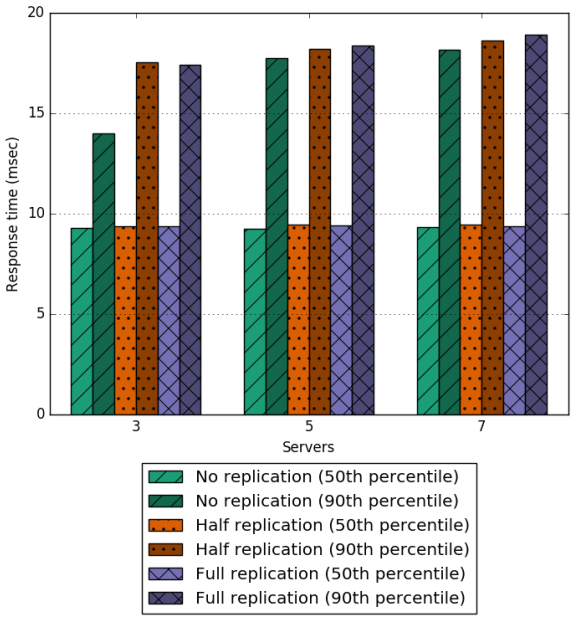

Figure 10: Response time percentiles for gets

machine as a whole, therefore a drop in the aggregate throughput for both `get` and `set` requests is to be expected when increasing the number of servers as we see in Figures 7 and 8.

Increasing the replication factor leads to increased server time, as we will see in Subsection 2.2, because every `set` request is sent to more memcached servers and all of the responses for a `set` request must be received before replying to the memaslap client. Increased server time leads to increased response time for `set` requests as we see in Figure 9, which causes the drop in the `set` request aggregate throughput for increased replication factor in Figure 7, because we have a closed system. Since the clients use a fixed 5% writes configuration and the throughput for `set` requests decreases, the throughput for the `get` requests also decreases by the same percentage, which can also be seen in Figure 8. This indicates that the bottleneck of our middleware should be in the execution path of the `set` requests requests. Increasing the number of servers on top of half or full replication accentuates this issue, decreasing further the aggregate throughput for both `set` and `get` requests.

2.2 Middleware Internals Behaviour

To examine how the time spent in the middleware is broken down for `set` and `get` requests, we take a look at the time spent in the read queue and the time waiting for the memcached server response.

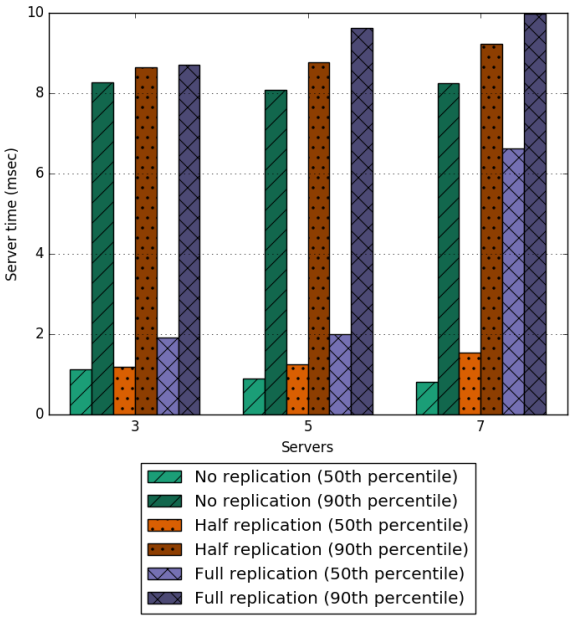

Figure 11: Server response time percentiles for sets

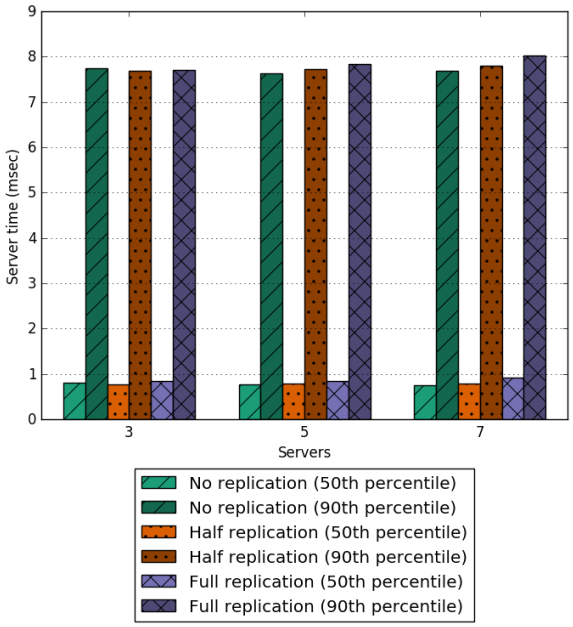

Figure 12: Server response time percentiles for gets

In Figure 11 we confirm that as the replication factor increases the median server time for `set` requests greatly increases. As we already mentioned in Subsection 2.1 every `set` request is sent to more memcached servers and all of the responses for a `set` request must be received before replying to the memaslap client. The median server time for `get` requests remains roughly stable throughout all experiments as we see in Figure 12.

	Servers		
Replication	**3**	**5**	**7**
No	0.018	0.020	0.020
Half	0.020	0.020	0.021
Full	0.020	0.020	0.023

Table 2: Queue time medians in msec for sets

	Servers		
Replication	**3**	**5**	**7**
No	0.231	0.042	0.030
Half	0.151	0.043	0.030
Full	0.278	0.048	0.036

Table 3: Queue time medians in msec for gets

Concerning the time spent in the queues, since there is a difference in the order of magnitude of the measurements for the 50th and the 90th percentiles as seen in Figures 13 and 14, we also consider the Tables 2 and 3 containing only the medians of the queue time measurements for `set` and `get` requests respectively.

For the `set` requests we only see an increase of the 90th percentile in Figure 13, the skewness of the queue time, but not the median in Table 2 when increasing both the replication factor and the number of servers, since `set` requests are performed asynchronously. The increase of the 90th percentile of the queue time with the replication factor can be explained if we take into consideration the implementation of the `WriteHandler`[3] thread in the middleware. The thread

[3] https://gitlab.inf.ethz.ch/gtouloup/asl-fall16-project/blob/master/src/ch/ethz/gtouloup/WriteHandler.java

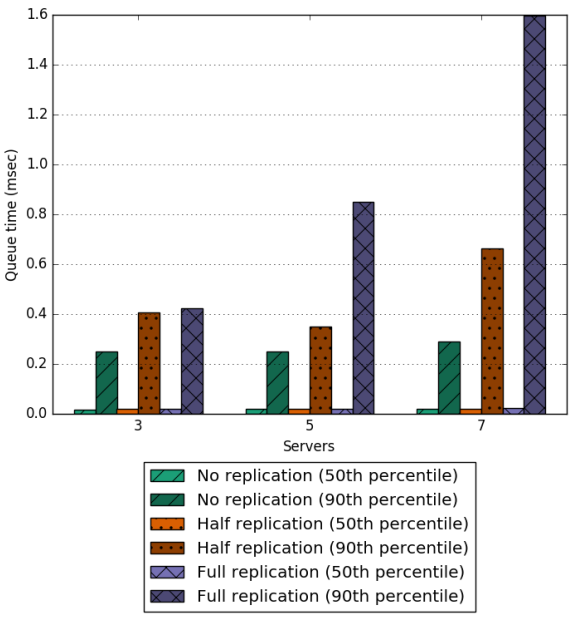

Figure 13: Queue time percentiles for sets

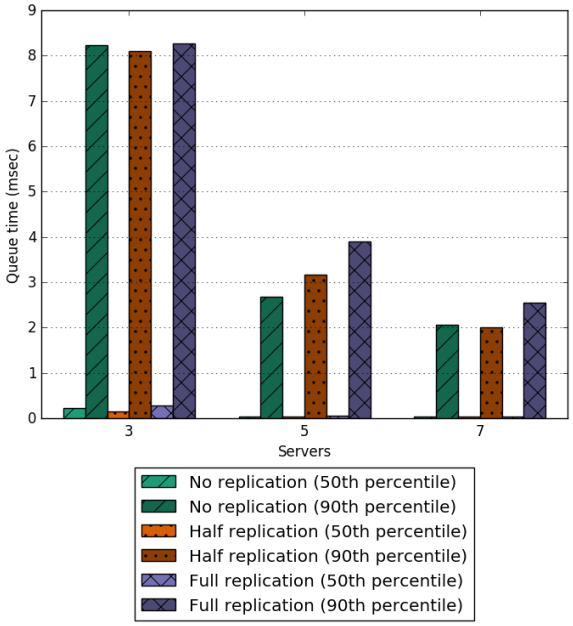

Figure 14: Queue time percentiles for gets

blocks on a selector that awaits for server responses. When a new write request is enqueued for the thread, the thread is notified via the `notifyWriteHandler` method that wakes the selector, allowing the thread to dequeue the request. For an increased replication factor or number of servers the threads spends more time processing the server replies than blocking on the selector, thus the increase in the queue time for the 90^{th} percentile is justified.

In Figure 14 and Table 3 we see that the queue time for the `get` requests remains relatively the same when varying the replication factor, while there is a big decrease when increasing the number of servers from 3 to 5. This happens due to the fact that the bottleneck of our middleware lies in the execution path of the `set` requests, as we mentioned in Subsection 2.1. The volume of `gets` that the middleware processes is constrained by the volume `sets` processes, as the clients force a fixed 5% writes workload. The `gets` are performed under capacity and the queue time decreases as we increase the servers, because the read queues and read thread pools increase proportionally at the same time.

To sum up, **as the replication factor and the number of servers increase, the server time for `set` requests greatly increases**.

2.3 Scalability with the Number of Servers

To compare the scalability of our system compare to that of an ideal implementation, we first have to define what an ideal implementation would be.

An ideal implementation of the middleware would not be the bottleneck of the system. All of its parts could be parallelized and there would not be any contention in the queues. In addition to that, all of the memcached servers would ideally be equally fast in processing the requests and the latency between the middleware and them would be constantly the same.

In this ideal implementation, the aggregate throughput would increase proportionally with the number of memcached servers for both `set` and `get` requests, since all the parts of the middleware would be parallelized and there would no contention in the queues. Moreover, the replication factor would have no effect on throughput, since all memcached servers respond simultaneously for every `set` request.

On the contrast to the ideal implementation, **our system exhibits a decreased aggregate throughput for an increased number of memcached servers for both `set` and**

`get` requests. As we explained in Subsection 2.1 this is due the overhead of managing a larger number of threads which an impact on the performance of the middleware machine as a whole and makes the middleware the bottleneck of our system. Moreover, an increase of the replication factor decreases the aggregate throughput for both `set` and `get` requests, since time window of the memcached servers responses for every `set` request increases with the number of servers. As we explained in Subsection 2.1, increased server time leads to increased response time for `set` requests, which causes the decrease of the `set` request aggregate throughput for increased replication. Since the clients use a fixed 5% writes configuration and the throughput for `set` requests decreases the throughput for the `get` requests also decreases by the same percentage, while increasing the number of servers on top of half or full replication accentuates this issue.

3 Effect of Writes

In this experiment we study the changes in throughput and response time of our system as the percentage of write operations increases. We experiment with the memaslap configuration increasing the writes workload number from 1% to 5% to 10%. At the same time we experiment with the number of memcached servers increasing their number from 3 to 5 to 7. We repeat the experiments for no replication and full replication.

We use 3 memaslap clients with 87 virtual clients each (261 virtual clients in total) and 24 threads per read thread pool in the middleware. This configuration is similar to the maximum throughput configuration we found in Section 1.

Memaslap clients use the `smallvalue_1.cfg`[4], `smallvalue_5.cfg`[5] and `smallvalue_10.cfg`[6] workload configurations (Key 16B, Value 128B, Writes 1%/5%/10% respectively).

Similarly to Section 1 each experiment runs for 150 seconds, we discard the first and last 30 seconds of the experiment as the warm up and cool down phases respectively and consider the 90 seconds left as 3 repetitions of 30 seconds each.

We use one Basic_A4 Azure machine for the middleware, 3 to 7 Basic_A2 Azure machines for the 5 memcached servers and 3 Basic_A2 Azure machines the 3 memaslap clients.

Number of servers	(3, 5, 7)
Number of client machines	3
Virtual clients / machine	87
Workload	Key: 16B, Value: 128B, Writes: (1%, 5%, 10%)
Middleware	Replication: (No, Full) , Threads: 24
Runtime	150 sec
Log files	writes.zip

We expect that an increase of the percentage of writes will negatively impact the performance of the system, since the bottleneck of our middleware lies in the execution path of the `set` requests, as we mentioned in Subsection 2.1.

3.1 Effect of Writes on Performance

Figures 15, 16 and 17 show the aggregate throughput for total, `set` and `get` requests respectively for the write percentage experiments.

We confirm that in Figure 16 the aggregate throughput for `set` requests increases when increasing the precentage of writes, which is expected. For the same reasons that we explained in Subsection 2.1, the aggregate throughput for `set` requests decreases when increasing the number of memcached servers or the replication factor.

In Figure 17 we see that the aggregate throughput for `get` requests decreases when increasing the precentage of writes, which is expected since the percentage of reads decreases accordingly. For the same reasons that we explained in Subsection 2.1, the aggregate throughput for `get` requests increases when increasing the number of memcached servers or the replication factor.

The total aggregate throughput displayed in figure 15 has the same behavior with the `get` requests throughput, since `gets` account for the majority of the requests (90%, 95%, 99%) for the experiment configurations.

[4] https://gitlab.inf.ethz.ch/gtouloup/asl-fall16-project/tree/master/workloads/smallvalue_1.cfg
[5] https://gitlab.inf.ethz.ch/gtouloup/asl-fall16-project/tree/master/workloads/smallvalue_5.cfg
[6] https://gitlab.inf.ethz.ch/gtouloup/asl-fall16-project/tree/master/workloads/smallvalue_10.cfg

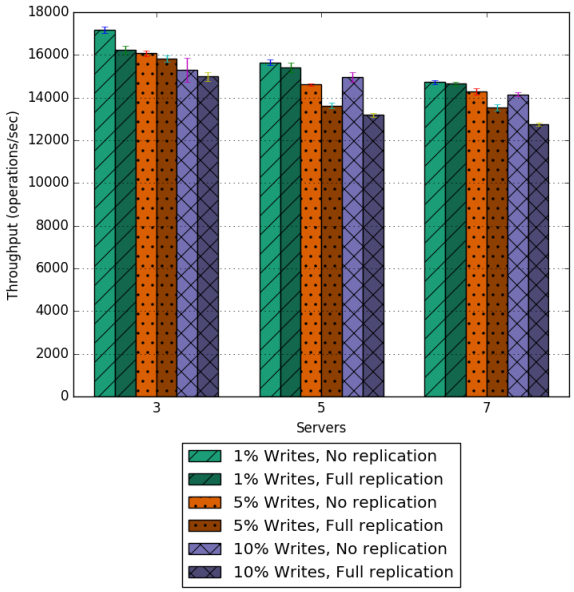

Figure 15: Total aggregate throughput

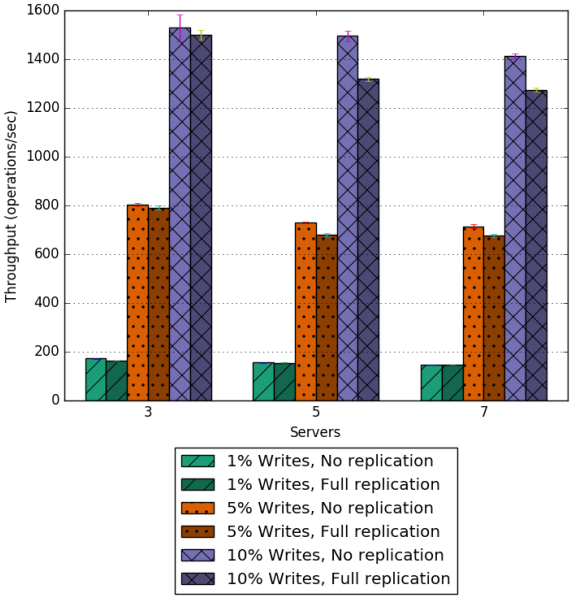

Figure 16: Aggregate throughput of sets

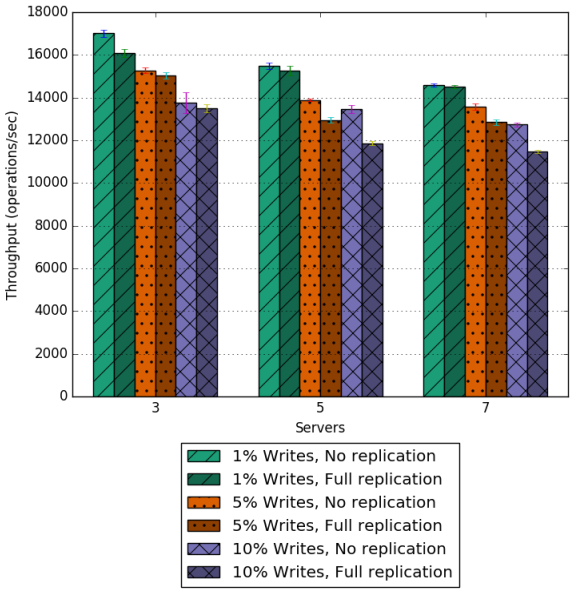

Figure 17: Aggregate throughput of gets

To extract meaningful conclusions on the effect of the write percentage on performance we also need to examine the response time, presented in Figures 18, 19, 20, 21, 22 and 23.

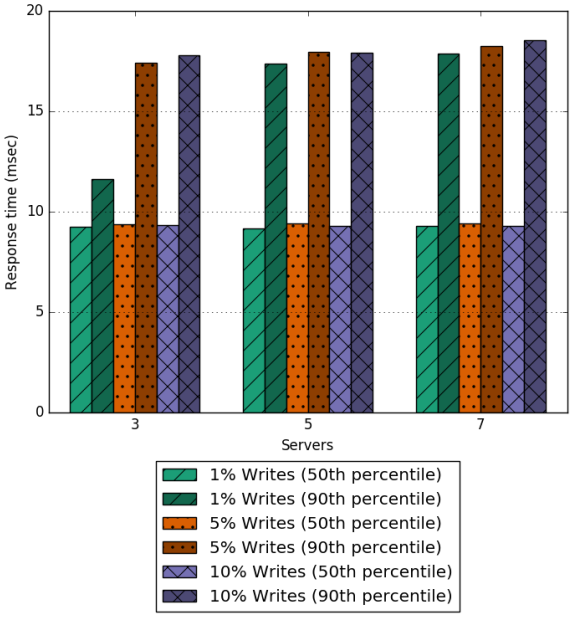

Figure 18: Response time percentiles for all requests (no replication)

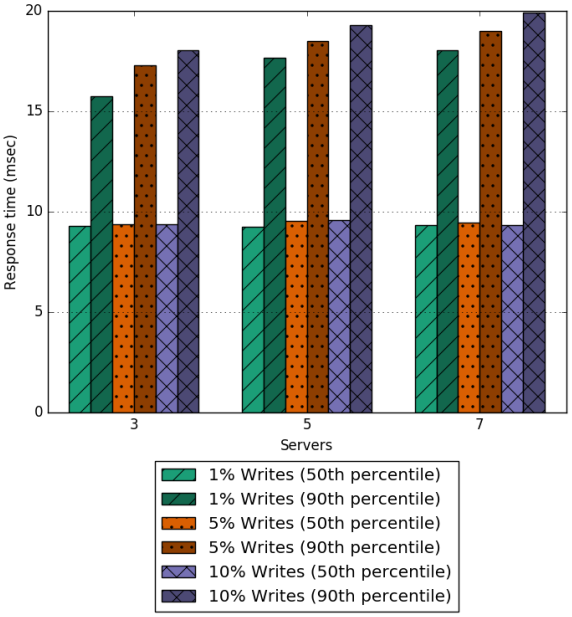

Figure 19: Response time percentiles for all requests (full replication)

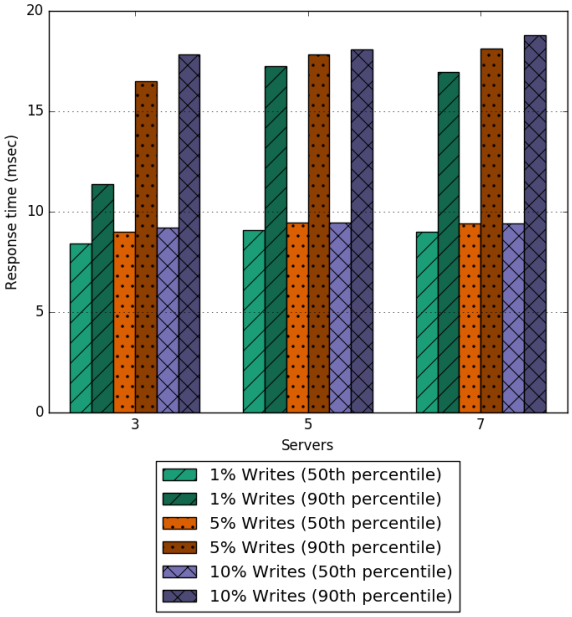

Figure 20: Response time percentiles for sets (no replication)

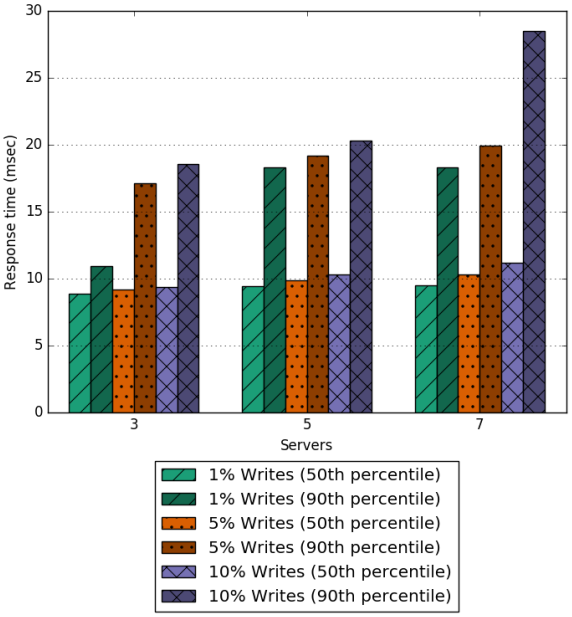

Figure 21: Response time percentiles for sets (full replication)

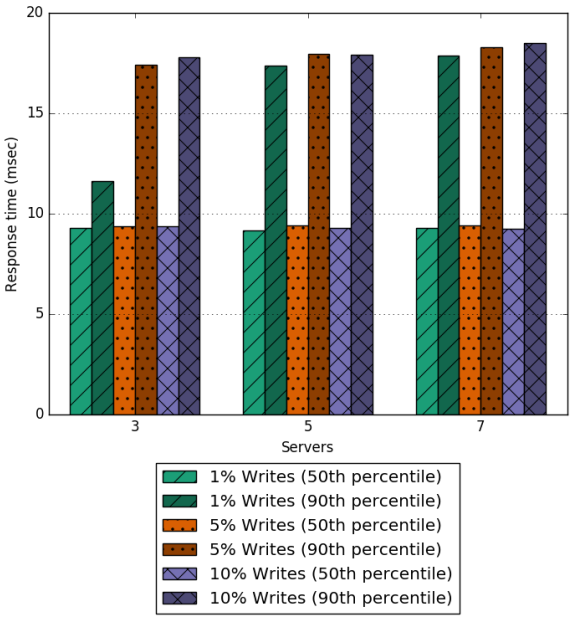

Figure 22: Response time percentiles for gets (no replication)

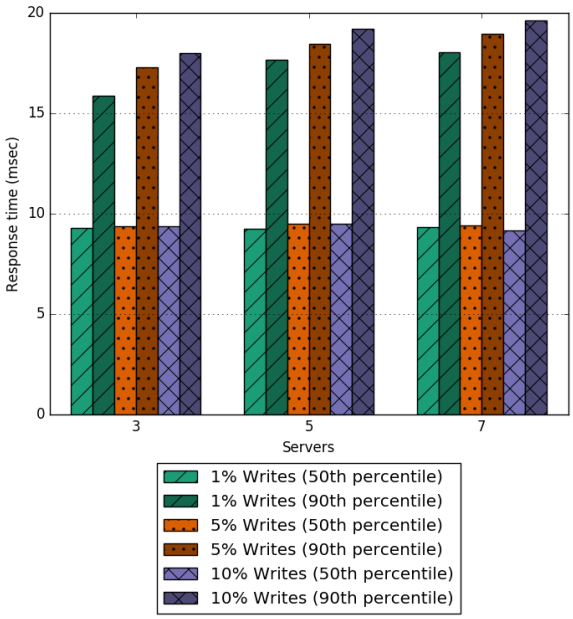

Figure 23: Response time percentiles for gets (full replication)

The response time for the total and the `get` requests remains roughly the same for all configurations as we can see in Figures 18, 19, 22 and 23 so we will focus on the response time for the `set` requests.

In Figures 20 and 21 we note that response time for `set` requests increases when increasing the write percentage, which is expected since the of our middleware lies in the execution path of the `set` requests. For the same reasons that we explained in Subsection 2.1, the response time for `set` requests increases when increasing the number of memcached servers or the replication factor.

Considering the experiment with 1% writes and no replication for 3 servers as the base case, Table 4 shows the change relative to the aggregate throughput for total requests of the base case as a percentage when varying the writes percentage.

		Writes		
Servers	**Replication**	**1%**	**5%**	**7%**
3	**No**	0	-6.4	-10.9
	Full	-5.4	-7.9	-12.7
5	**No**	-8.9	-14.9	-12.9
	Full	-10.3	-20.7	-23.3
7	**No**	-14.2	-16.8	-17.6
	Full	-14.7	-21.2	**-25.8**

Table 4: Throughput change percentage relative to base case

Table 4 shows that **the biggest impact on performance relative to the base case is observed for 7 servers**. The two rows concerning the 7 server configurations present bigger changes relative to the base case in comparison to the respective rows for 3 and 5 servers.

We also see in Table 4 that the biggest change relative to the base (-25.8%) case occurs for the maximum of simultaneously the percentage of writes and number of servers as well the replication factor. Taking also into consideration the previous observations of this Subsection we conclude that **the main reason for the reduced performance is a combination of increased write percentage and increased number of servers along with an increased**

replication factor.

3.2 Middleware Internals Behaviour

To investigate the behavior of the system when varying the writes percentage we take a look at the time spent in the read queue and the time waiting for the memcached server response.

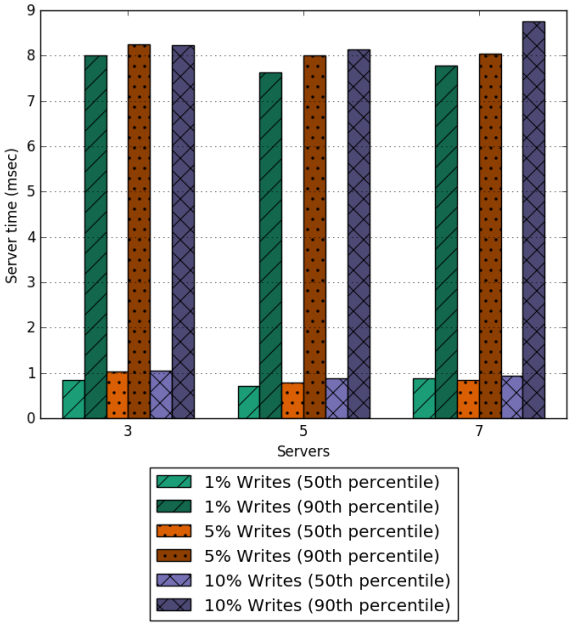

Figure 24: Server response time percentiles for sets (no replication)

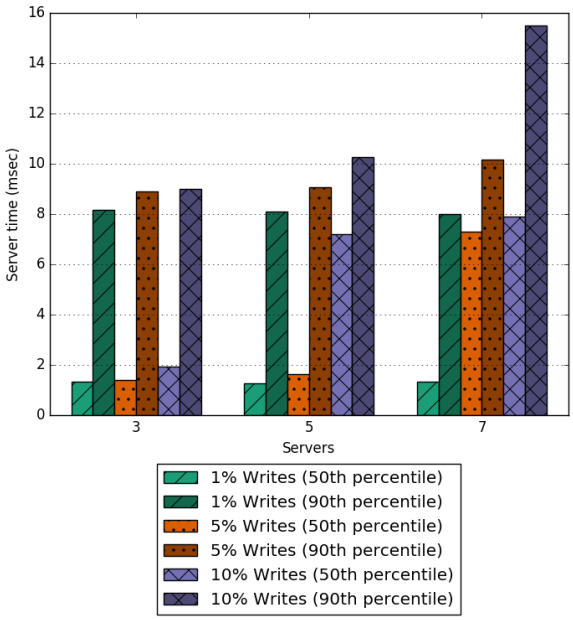

Figure 25: Server response time percentiles for sets (full replication)

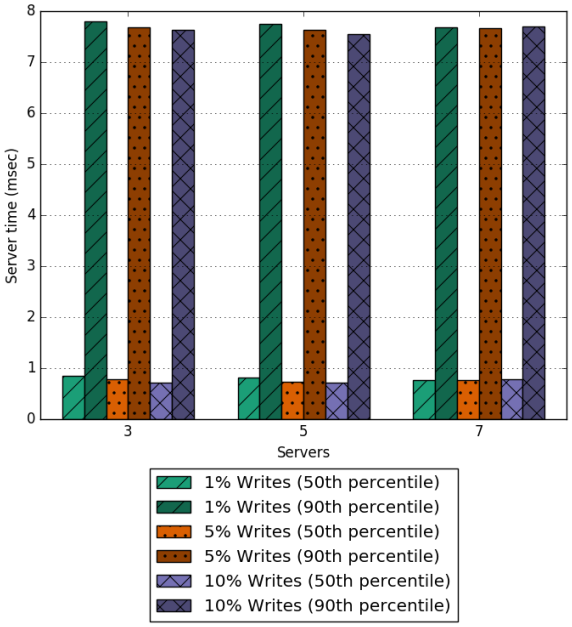

Figure 26: Server response time percentiles for gets (no replication)

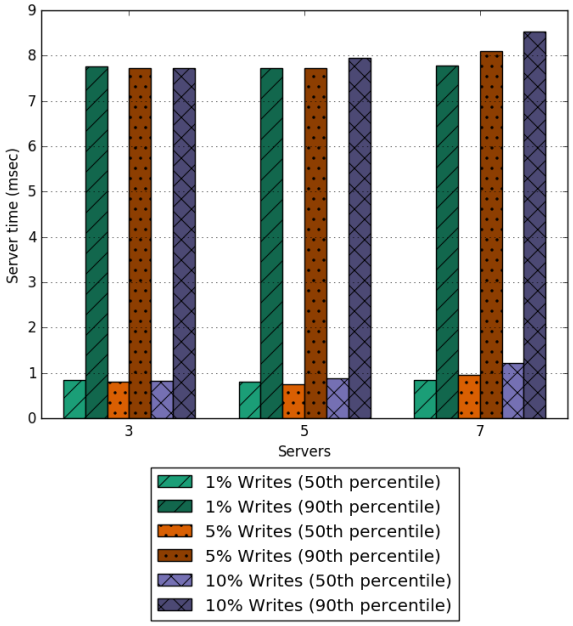

Figure 27: Server response time percentiles for gets (full replication)

In Figures 24 and 25 we see that the median server time for `set` requests remains relatively stable for the experiments without replication, however increases greatly for full replication

when the writes percentage is increased. This happens either due to the increase in memcached processes time or due to network congestion, since more requests are sent to the memcached servers the percentage of writes is increased.

The median server time for `get` requests remains roughly stable throughout all experiments as we see in Figures 26 and 27. The slight increase of server time for the experiments with full replication and 7 servers is either due to the increase in memcached processes time or due to network congestion.

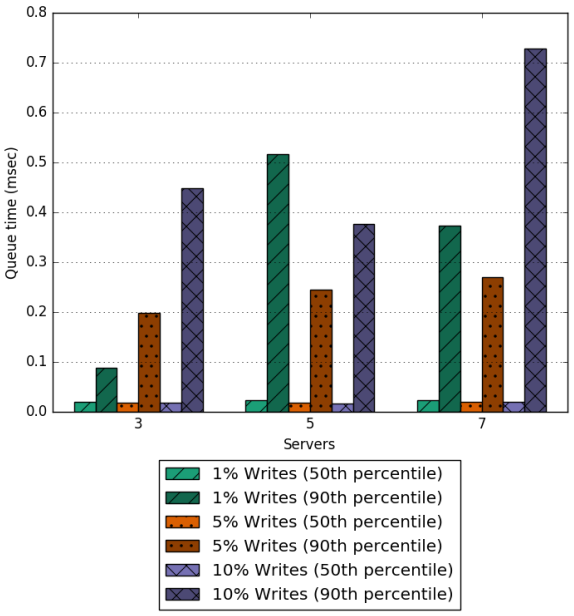

Figure 28: Queue time percentiles for sets (no replication)

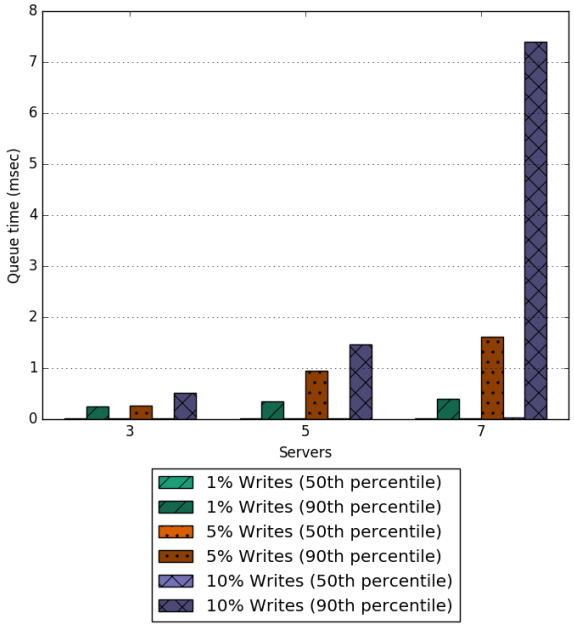

Figure 29: Queue time percentiles for sets (full replication)

	Writes		
Servers	**1%**	**5%**	**10%**
3	0.020	0.018	0.018
5	0.024	0.019	0.017
7	0.023	0.020	0.020

Table 5: Queue time medians in msecs for sets (no replication)

	Writes		
Servers	**1%**	**5%**	**10%**
3	0.024	0.019	0.018
5	0.024	0.021	0.021
7	0.025	0.023	0.028

Table 6: Queue time medians in msecs for sets (full replication)

Concerning the time spent in the queues for the `set` requests, since there is a difference in the order of magnitude of the measurements for the 50^{th} and the 90^{th} percentiles as seen in Figures 28 and 29, we also consider the Tables 5 and 6 containing only the medians of the queue time measurements for `set` requests.

For the `set` requests we note that the median of the queue time is stable in Tables 5 and 6 throughout all the experiments, since `set` requests are performed asynchronously. We only see a trend in the increase of the 90^{th} percentile in Figure 29, the skewness of the queue time, when increasing the percentage of writes. This can be explained if we take into consideration the implementation of the `WriteHandler` described in Subsection 2.2. For an increased write percentage the threads spends more time processing the server replies than blocking on the selector, thus the increase in the queue time for the 90^{th} percentile is justified.

In Figures 30 and 31 we see that the queue time for the `get` requests does not follow a clear trend when varying the percentage of writes. This happens due to the fact that the bottleneck

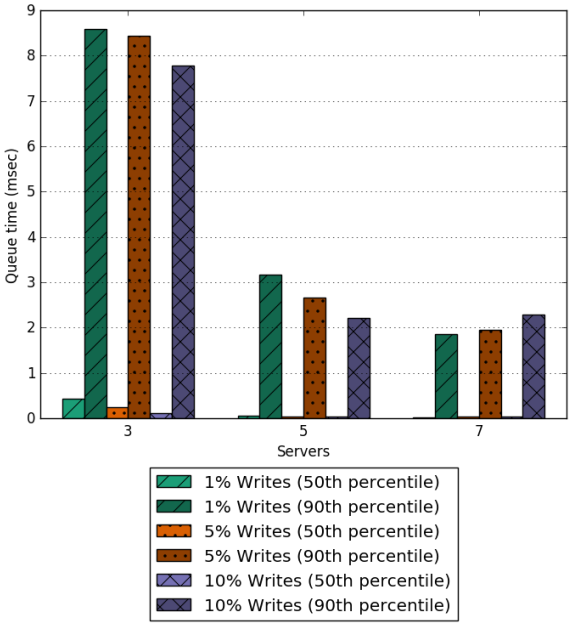

Figure 30: Queue time percentiles for gets (no replication)

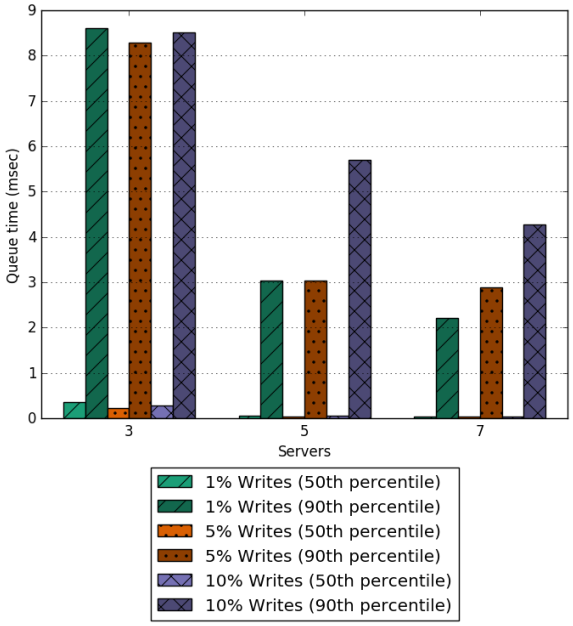

Figure 31: Queue time percentiles for gets (full replication)

of our middleware lies in the execution path of the `set` requests, as we mentioned in Subsection 2.1 and the `gets` are performed under capacity.

When increasing the replication factor or the number of servers both queue and server times for all experiments follow the trends observed and explained in Subsection 2.2

Logfile listing

Short name	Location
maxthroughput.zip	https://gitlab.inf.ethz.ch/gtouloup/asl-fall16-project/blob/master/logfiles/maxthroughput.zip
replication.zip	https://gitlab.inf.ethz.ch/gtouloup/asl-fall16-project/blob/master/logfiles/replication.zip
writes.zip	https://gitlab.inf.ethz.ch/gtouloup/asl-fall16-project/blob/master/logfiles/writes.zip