Advanced Systems Lab (Fall'16) – Second Milestone

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

Grading

Section	**Points**
1	
2	
3	
Total	

1 Maximum Throughput

1.1 Experiment design

The **goal** of this experiment is to discover what the maximum throughput of the middleware will be for a fixed number of 5 memcached servers. Thus, the main **performance metric** is throughput, which is an example of a *Higher-is-Better* metric.

There is a number of system parameters that will affect system performance, but only some of them will act as **factors** during this experiment. In particular, the parameter whose value will be varied is the number of threads in the thread pool for the GET request queue. This is of importance in this case, because of the **workload** that will be applied to the middleware. The ratio between GET and SET requests will be a fixed **workload parameter**, where the clients will perform read only operations. Finally, a workload parameter that will also act as a factor is the number of clients that use the middleware. The analysis done on the baseline experiment from Milestone 1, tell us that clients can generate more requests when they are spread over VMs. That is why this experiment will be carried out with 5 client VMs.

There will be two types of experiments to make finding the maximum throughput easier. First, a more coarse-grained approach will increment the number of virtual clients by 100 between experiment instances. The goal is to find a rough interval of where the maximum is. This will be a one-factor experiment where the number of threads is fixed to 32.

Number of client machines	5
Virtual clients / machine	100 to 700
Thread pool	Fixed - 32
Log files	tps_100_700.tar.gz

After discovering the interval, a finer-grained set of experiments will be run, where the number of clients on each VM is increased by 4, thus making a difference of 20 clients between experiments. More importantly this will be a 2k experiment where k = 5, as 5 values will be chosen for each factor. The reason for varying the number of clients by 20 is that my expectation is that a value of 10 will not result in substantial difference in the results. This is discussed more in the next subsection. The number of threads in the thread pool will also vary by doubling the amount each time, because I want to cover enough values and see when this parameter stops making a difference. For this I will use the following 5 values: 8, 16, 32, 64, 128.

An important point to make is the window parameter, which memaslap will consult when populating the servers with values. If the default value of 10k is used, then for a relatively big amount of virtual clients, memcached will start evicting keys, which will result in a lot of get misses recorded by memaslap. This will affect performance because although memcached will still have to do a lookup for a missing key, the size of the response will be smaller. That means less load on the network and overall less response time for failed requests, which will result in unstable results. To avoid affecting performance because of requests being missed, the experiment will set a fixed value of 2k for the a window workload parameter.

The final two things to mention are the runtime of each experiment and the replication (number of times the same configuration is run). The experiments in Milestone 1 show that the middleware reaches a stable state, at which it can run for at least an hour, and that the warm-up phase of 20 seconds, which was allocated previously, is enough for the middleware to start working on a stable level. That is why 2 minutes and 30 seconds runtime are assumed to be enough per experiment instance. Although this is enough to show how the system will behave, a replication factor of 3 will be used to average out the results and get rid of accidental anomalies.

Number of servers	5
Number of client machines	5
Virtual clients / machine	380 - 460
Workload	Key 16B, Value 128B, Read-only
Middleware	Replication = 1
Middleware	Threads in pool = 20 to 100
Runtime x repetitions	2.5 x 3
Sampling	Every 100th GET
Log files	tps_360_440.tar.gz, tps_360_440.log
	timestamps_borders.tar.gz, parsed_timestamps_borders.log

1.2 Hypothesis

As seen in the stability trace, for 192 clients and 16 threads in the thread pool, the middleware generates and average throughput of around 10k. My expectation is that these configurations were limiting the middleware and it is actually capable of handling more throughput. The reasons I expect to see higher TPS with some other values of the factors are:

- Clients will be spread across 5, instead of 3 VMs, which as seen from the Baseline experiment in Milestone 1 is supposed to generate more requests, due to less context switching between threads on a single multi-core processor.
- More virtual clients will be able to generate more requests.
- Get requests will be handled more efficiently for an increased number of threads in the pool.

To give a more concrete expectation, I expect the middleware to reach its limit at around 500 clients and 15k TPS. This is because, as seen from the baseline, a single VM running many clients, will start saturation the memcached servers. Since we are running 5 client machines and 5 servers, I see that a limit will be placed not only by the middleware performance, but by the servers as well.

As for the size of the thread pool, I expect that there will be a point at which adding more threads will not improve performance. This is because we have a closed system under test and thus a limit on the requests that we can have in the queue at a given time. Given this assumption, I think the size of the thread pool will not play a major role in the performance. However, considering that there will be only GET requests coming from the client, I expect that a pool with size of around 60 will give the best performance.

1.3 Experiment results and Analysis

The initial experiment can be considered successful because it achieved its purpose of showing roughly the configurations under which the middleware generated the most throughput. The results of the experiment are stored in *tps_100_700.tar.gz* and are visualized in Figure 1.

Thus, we reach the conclusion that the middleware shows maximum performance at around 12340 get operations per second, when there around 400 clients issuing requests and a fixed sized thread pool of 32. It is important to note that TPS measurements have been taken only after the server population phase has finished, so that only GET requests are taken into account. Another noteworthy observation is that the performance achieved here for 200 clients is higher, than the performance of the middleware under the stability trace experiment, where 192 clients were running. The main difference of the setup is the type of requests that are handled by the middleware, which is one confirmation that GETs take less time than SETs with full replication.

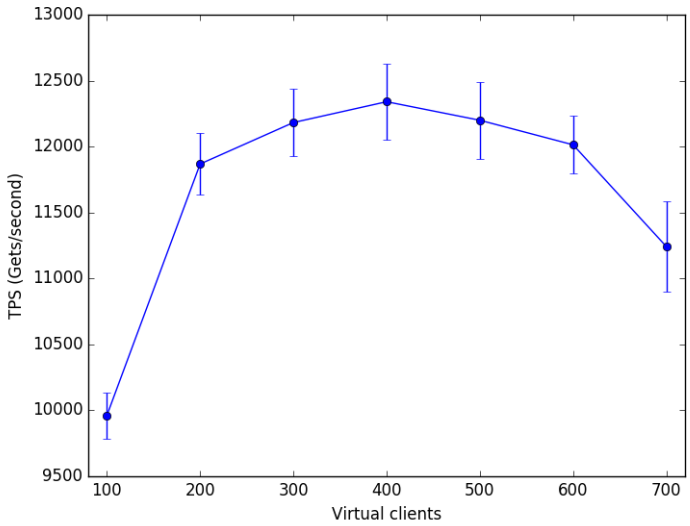

Figure 1: Coarse-grained experiment with fixed pool = 32 threads

The standard deviation is also included in the graph and for 400 clients it is equal to *289*. This can be used to calculate the confidence interval and make sure that the number of samples and runtime is enough for the experiment. In order to get a correct measurement, the samples include a warm-up and cool-down phase of 10 seconds. This ensures that we do not take into account measurements taken while less than all 5 VMs were sending requests. This may happen when one VM finishes its population phase a few seconds later than the others.

$$CI = \overline{x} \pm t \times \frac{s}{\sqrt{n}}$$

For a 95% Confidence interval we have the following equation:

$$CI = \overline{12340} \pm 2.01 \times \frac{289.3}{\sqrt{50}}$$

,

which results in the conclusion that in 95% of the cases the value would be between 12258 and 12422.

Now that the maximum throughput interval has been decided, a set of more fine-grained 2k experiments with k = 5, can be run. The point is to examine the performance around the 400 client configuration from the previous experiment. Thus, 400 virtual clients have been set as a central point and since there is not a dramatic change of performance when number of clients change by a 100, $\delta clients$ will be equal to 20 instead of 10. Therefore, the other values are 360, 380, 420 and 440. The results of those experiments are stored in *tps_360_440.tar.gz* and the parsed data is in *tps_360_440.log*. The results are shown in Figure 2.

As it can be seen from Figure 2, the best performance is achieved with 380 clients and a thread pool size of 8. Good for showing the stability of the middleware is that the configuration with 400 clients and thread pool of 32 threads, from the coarse-grained experiment, again has TPS of around 12300. However, in order to ensure that an 8 thread configuration, will provide the most consistent performance, we should also examine the standard deviation between the two configurations giving best performance - 8 and 16 threads. This is depicted in Figure 3, where we can see that the trace with 8 threads provides better performance, while maintaining a very similar standard deviation as shown in detail in the table below. Therefore we can reach the conclusion that the middleware will provide its best performance of **13330 TPS** in the configuration where we have **380 virtual clients and 8 threads in the thread pool**.

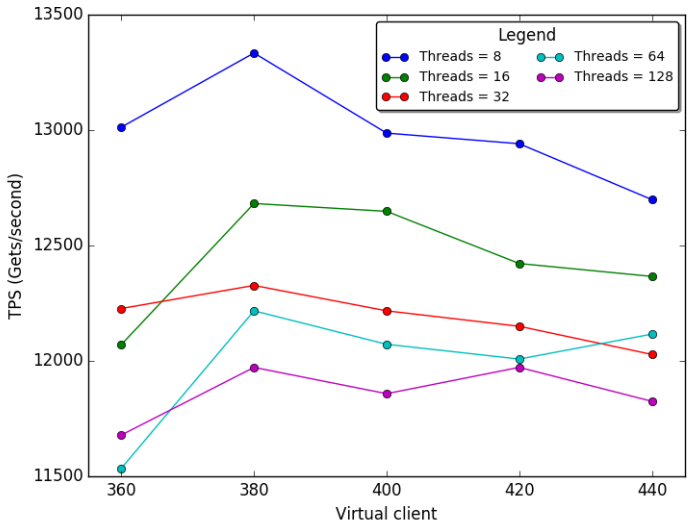

Figure 2: 2k experiment

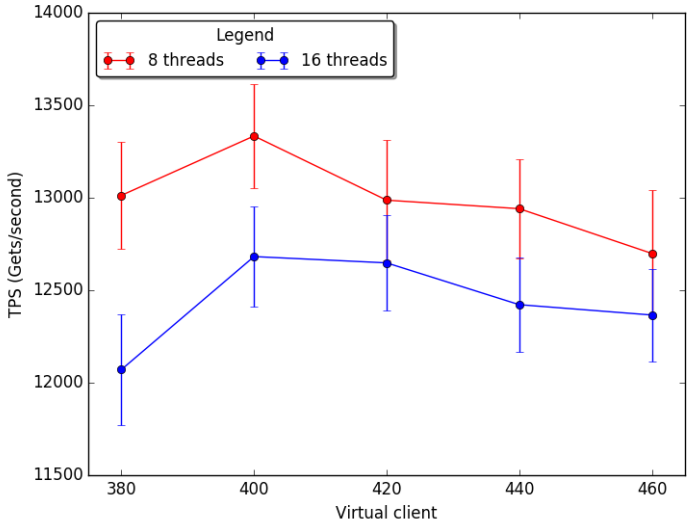

Figure 3: Server time difference between experiments

	360 clients	380 cl.	400 cl.	420 cl.	440 cl.
8 threads (std_dev)	287	281	328	268	343
16 threads (std_dev)	299	272	258	255	249

There are a couple of reasons why the middleware achieves the best TPS under that configuration. First of all, as suggested in the hypothesis, there is a point where the clients start saturating the middleware and it is no longer able to serve requests at the rate it was working at, when there were fewer clients. Thus, 380 clients is the point where the middleware starts getting overloaded. On the other hand, before that point the middleware is able to handle requests from clients with little delay and keep up with the rate at which they are sent. For example, having a 100 clients is an extreme case, because the system is hugely underloaded. Again, this was expected, because as seen from the Baseline experiment, there is a limit to the workload a certain number of clients can place on the middleware.

Similar argument can be made for the size of the thread pool. From the experiments we learn that the more threads there are, the bigger the contention at the GET queue is. Because of the way this is implemented, all threads are constantly polling the head of the queue, which creates some concurrency delay because of locking and synchronizations. Furthermore, running more threads than there are processor cores, will result in context switching, which does not help performance. It was again predicted in the hypothesis that, there will be a point after which adding more threads will not improve the performance. However, my expectation was that it would take more than 8 threads to do that. The conclusion for this is that 8 threads are enough to handle the requests in the queue, otherwise more of them will just be in contention.

The above provides and intuitive explanation of why the maximum throughput is achieved with these values of the factors. However, to get a more detailed and concrete explanation, we should examine how the time spend in the middleware changes at the border configurations. For that we take the timestamp logs, taken during the traces with 8 threads (*timestamps_borders.tar.gz* and *parsed_timestamps_borders.log*). The stats are presented in the below table.

	GETs 360	**GETs 380**	GETs 400
T_{parse} (ms)	0.01207660	0.01231402	0.01124946
T_{queue} (ms)	0.82582398	0.82464785	0.97104232
T_{server} (ms)	1.39008446	1.35251127	1.55420358
T_{return} (ms)	14.91118652	14.20334295	14.82913253
T_{total} (ms)	17.17681179	16.42170614	17.39080552

There are two things we learn from that table. First of all, as expected the time spent in each part increases when we move away from the highest throughput configuration. This is most noticeable in T_{return} - the time it takes between receiving the response from the server and sending it back to the client. Secondly, the same timestamp appears to be taking most of the time for request processing, which means we have found a potential bottleneck. This will be further examined in Milestone 3.

Finally, we give a table with statistics including standard deviation about the timestamps under the best configuration - 30 clients and 8 threads.

	GETs (mean)	GETs (std_dev)	SETs (mean)	SETs (std_dev)
T_{parse} (ms)	0.01231402	0.14457305	0.00967424	0.19036998
T_{queue} (ms)	0.82464785	2.23746803	0.68237895	5.97962474
T_{server} (ms)	1.35251127	2.6497771	3.39611938	10.44147153
T_{return} (ms)	14.20334295	10.81412428	11.65818910	8.22186546
T_{total} (ms)	16.42170614	9.41573511	15.78843058	10.48846277

2 Effect of Replication

2.1 Experiment design

The goal of this experiment is to explore how the middleware behavior changes when the clients are producing more SET requests and in particular analyze how these configurations affect the two types of requests. Unlike the first experiment, behavior is now defined in terms of both throughput and response time. However, in order to get a more detailed view of how different operations are affected, the response time will be of higher importance. Therefore for each configuration that we run the system with, the performance metrics will be throughput, response time and time spent by logged requests in each part of the middleware.

Again, there are several factors and fixed parameters to that experiment. The first factor is the **number of backend servers** which the middleware will store/fetch requests to/from. This parameter is relevant to the performance of the whole system, because it determines the level of load balancing we can achieve. The values for that factor will be 3, 5 and 7 servers. The second factor is a middleware parameter and it determines what is the **replication** that the middleware will do when it receives a SET request. This is important to what we want to find out about the system, because the execution sequence of the middleware depends on it. First of all it determines how many requests are going to be made per SET and secondly - how many responses the middleware should wait for before it is able to send a final answer to the client. The levels of replication will be 1, half and all backend servers.

There are also a few important parameters that will have a fixed value throughout this experiment. First, in order to improve the chances of seeing more drastic changes in performance, the clients will place a **5% write workload** on the middleware. Secondly, as we have seen multiple times, the **number of client VMs** also affects the performance of the whole system. Since there will be experiments, where 7 machines will be used as servers, this leaves us with 3 VMs to act as clients. This is fixed so to ensure consistency even between experiments that use less than 7 VMs as servers. From the maximum throughput experiment we have seen that the best performance is achieved when using 5 VMs with 76 clients as each. However, we also know that a single VM has a point beyond which having more clients is not efficient. With this in mind, each VM will have 120 clients, making it **360 clients** in total. Finally, the **size of the thread pool** will be set to 8, based on the previous results.

The logging of requests at the middleware will also be changed. In order to ensure we capture an equal number of GETs and SETs, the rate at which each operation type is logged, will change. Based on the ratio set by the workload configuration file, we will log each 500th GET and each 25th SET. Since writing to a file is one of the most time consuming operations in the middleware, this change will affect the whole performance and it will not be reasonable to compare the results with thos from previous experiment. This, however, does not interfere because this is a standalone experiment.

Finally, as already seen, each experiment instance can be run for 2 minutes to get enough samples during the stable phase, while leaving enough warm-up and cool-down time. Again, to get reliable measurements for all configurations, there will be an experiment replication factor of 3.

Number of servers	3, 5, 7
Number of client machines	3
Virtual clients / machine	360
Workload	Key 16B, Value 128B, 5% write
Middleware	Replication = 1, $\lceil \frac{S}{2} \rceil$ and all
Middleware	Threads in pool = 8
Runtime x repetitions	2 x 3
Sampling	Every 500th GET and every 25th SET
Log files	repl_memaslap.tar.gz, repl_timestamps.tar.gz
	repl_parsed_memaslap.log, repl_parsed_timestamps.log

2.2 Hypothesis

First of all, as mentioned, because of the change in the logging rate, the throughput achieved here cannot be accurately compared to the one from the previous experiment. However, I expect in this scenario to generally achieve less throughput because of the fewer clients overall, the presence of more SETs which as established are more expensive to handle and the logging rate. As for the configurations that will be applied, I expect TPS for get requests to not be impacted by the replication factor and the number of backend servers. However, the middleware will probably handle less set requests as the replication factors increases.

The variation in response time between configuration will depend on how get and sets are impacted by the factors. Since get requests are only sent to one server, the replication factor should not make any difference. However, if the number of server backends is smaller, then each server will have more entries, which in theory may increase the lookup time. However, the lookup should be a very quick process, because of the hashing used by memcached. Thus, the response time should not be affected by get requests, since they themselves should not be affected by a larger replication factor. In the case of, SETs however, the bigger the replication factor is, the more requests the middleware will have to send. This also means that, it will wait for all servers to respond before sending its final response to the client. Therefore, I expect to see an increase in the overall response time for SETs, as the number of servers and replication increases. Since there are also more set requests issued by clients, I expect this to cause an increase in the overall response time of the middleware.

The part of the request lifecycle that is directly related to replication is T_{server}. This includes the time from sending the SET to the first server until, we hear all responses. Thus as the replication factor and number of servers increases, I expect to see the biggest impact on this timestamp. This, itself may also have a secondary impact on other times such as T_{queue}, but I do not expect to see much difference in other timestamps for the set requests. The reason is that the system under test is closed and will not add any more requests to the queue unless it has seen a response for the previous ones.

Finally in order to be able to accurately compare the scale up of the middleware to that of an ideal implementation, we first have to define what an ideal scale up is. For such a systems, the replication factor will not have an effect on the performance of the system. In other words, the response time as the replication factor grows, will remain the same as for the case where there is not replication. However, in real life there is an important factor that will prevent the system from behaving ideally. In case one of the replication servers takes longer than the other to respond, this will drop the performance, both in terms of reducing throughput and increasing response time. This is exactly what I expect to happen to the middleware.

2.3 Experiment results and Analysis

2.3.1 Configuration impact on GETs and SETs

Let's begin the analysis by first examining how get requests are impacted by the various configurations. The memaslap log files, which are referred to are *repl_memaslap.tar.gz and repl_parsed_memaslap.log*. Figure 4 shows how the response time for get requests changes using the median plus 75th and 25th percentile. In the hypothesis it was suggested that the replication factor will not play a role here and indeed it can be seen that changes in the levels do not cause the middleware to increase or decrease the response time (the lines remain relatively horizontal). However, it turns out that the number of backend servers significantly affects performance. In the hypothesis it was assumed that some overhead may be introduced by the lookup table at the memcached server, but this is not the case, because response time is highest when there are more servers and respectively less entries in each of them. Thus, the size of the cache does not affect performance. To answer why we get such results, we have to analyze the middleware timestamps in more detail, which is done in the next subsection.

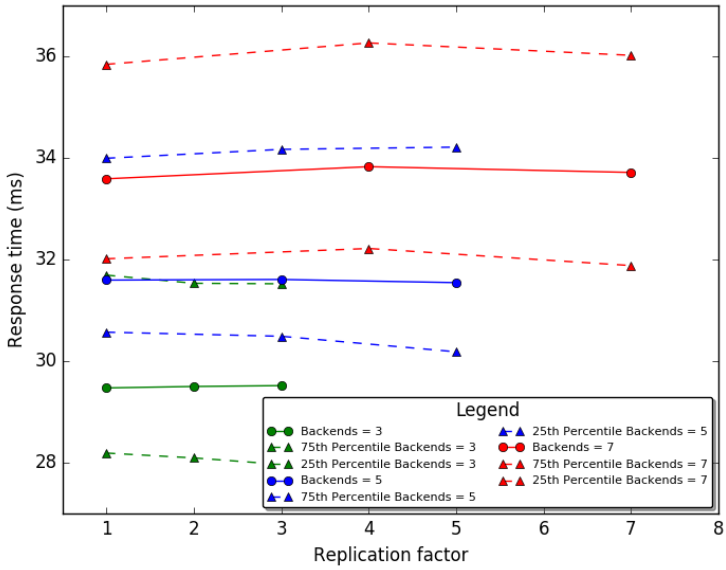

Figure 4: Overall Response Time for GET requests per number of backends

The same pattern, which was observed for the response time, can be seen for the other performance metric - throughput, in Figure 5. TPS remains stable as the replication factor increases, which again proves that GETs are not correlated to it. Naturally, the smallest response time for the 3 servers, leads to best throughput.

The same log files are used to examine the behavior of SETs. As proposed, set requests are affected by both factors, as it can be seen from the increase in response time in Figure 6. Firstly, when the replication factor is higher, the middleware will send requests to more servers, but more importantly, it will have to wait for responses from all servers, before returning a result to the client. This means that T_{server} is proportional to the time it takes for the slowest server to respond. It can be the case that all servers respond at almost the same time, but it is also possible that one server takes more time than the rest. This is the case where a higher replication factor will result in a higher T_{server}. This is again examined using the middleware timestamps.

Similarly to the GETs, set requests are also affected by the number of backend server. This is again examined later, using the timestamp data.

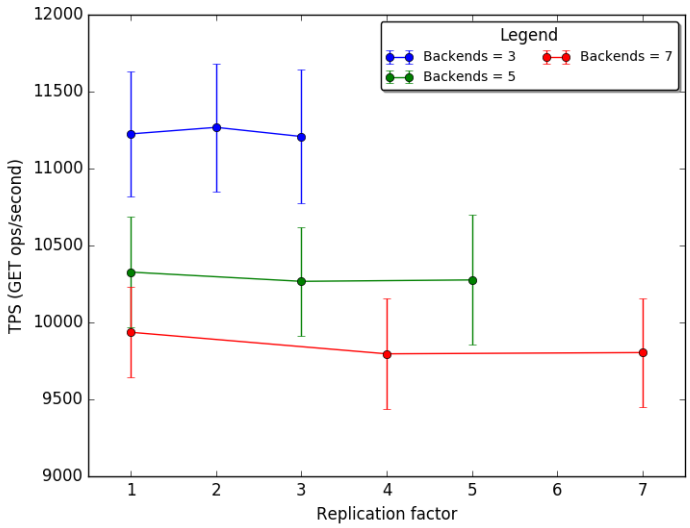

Figure 5: Throughput for GET requests per number of backends

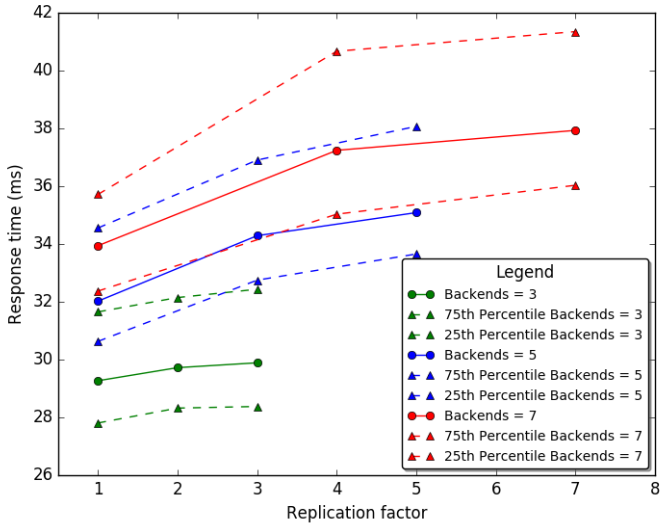

Figure 6: Overall Response Time for SET requests per number of backends

Finally, Figure 7 just confirms, what was already expected. The throughput for SETs is impacted by both factors , the same way the response time is. Performance for this metric, reduces as the replication factor and the number of servers increases.

2.3.2 Impact on operations inside middleware

After examining the results coming from memaslap, lets answer the question of how and where the configurations change the behavior of requests inside the middleware. First of all there are three parts of handling a request that are affected by the change of backend servers and replication factor. These are the queue time of a request, the server time and the return time. To get a clear understanding of what is happening in the middleware, these timestamps will be studied for both GETs and SETs. The graphs that are presented in this subsection show the response time for each configuration in term of the median time and its 75th and 25th

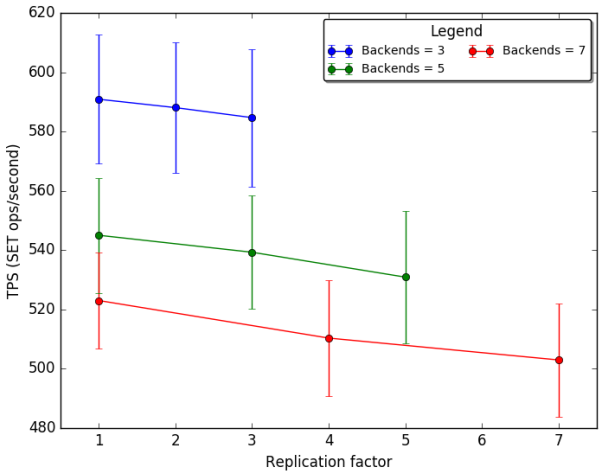

Figure 7: Throughput for SET requests per number of backends

percentile, in order to present enough data about the variation. The log files which provide the results for this subsection are *repl_timestamps.tar.gz and repl_parsed_timestamps.log.*

Lets first examine the effect of the configurations on T_{queue} for GETs from Figure 8. Most notable is the fact that, unlike the overall response time for GETs, shown in the previous section, this time the fewer backend servers there are, the longer a request spends in the queue. This is related to a claim made in the hypothesis. Earlier it was suggested that less backends means more entries in each, thus a higher lookup time. However, something that was missed when making the hypothesis, is that fewer servers, also mean proportionally fewer GET queues. Despite the even load balancing described in Milestone 1, the size of the queues is going to be bigger than the size, for the same workload and more servers. As a result the threads in the thread pool are not able to handle the incoming request as efficiently, as if they were spread across more queues.

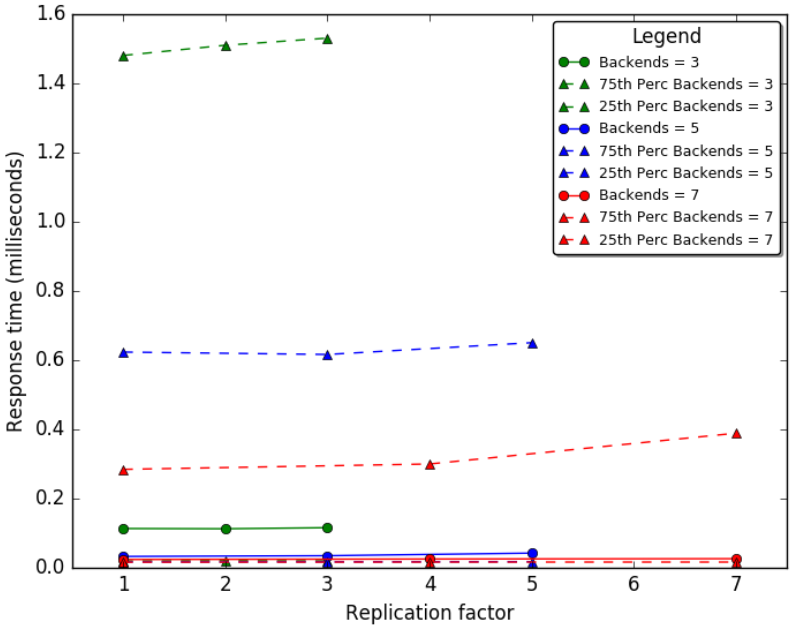

Figure 8: Queue time for GET requests per number of backends

One last thing to note is that T_{queue} is actually small compared to T_{server} and T_{return}, since 75% of the request spend less than 1.5 milliseconds in the queue. Although in these scenarios the queues become a bottleneck, they have a small effect on the overall response time observed in section 2.3.1.

T_{queue} for SETs is again highest when there are the fewest number of backends. This is the case although the thread is asynchronous. However, there is a significant difference caused by the asynchronous operation. Set requests spend less time in the queue than get requests. The 75th percentile for 3 backends shows 1.6 ms, while for SETs it is around 0.16 ms.

However, unlike with GETs, T_{queue} grows with the replication factor. The reason is again an implementation one. After taking a request off the queue, the thread has to send it to all replication servers, before going back to handling the queue. Thus, the more servers it has to send to, the more the next request has to wait before leaving the queue.

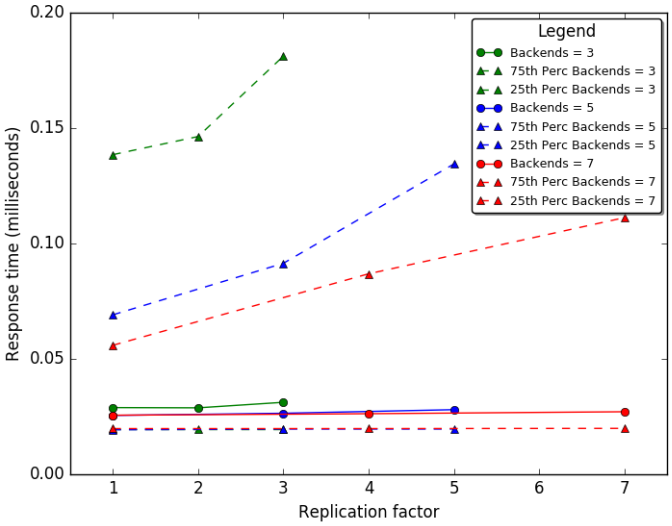

Figure 9: Queue time for SET requests per number of backends

The difference between the Server time for GETs and SETs is sufficiently high so that it can be visualized on a single graph, shown in Figure 10. It confirms that, as expected, T_{server} for GETs is affected neither by the replication factor, nor by the number of servers, thus invalidating the hypothesis that a bigger cache can influence response time. We see that there are small differences between the median response time, but important here is the equality of the percentile lines.

In the hypothesis it was suggested that the replication will have the biggest effect on T_{server}. Indeed from Figure 10 we see that it can grow up to 5 times when the replication is increased from none to full (7 for example). As already explained this it due to the fact that the middleware has to wait for the response of more servers. This then increases the chance that at least 1 server will take longer time, which will influence the total server time of a request.

Finally, using the data $T_{returnt}$ we can answer an important question which was imposed in section 2.3.1 - why is response time for both SETs and GETs growing with the increase of backend servers? This seems even stranger now that we have established that the middleware behaves in the opposite way for T_{queue}. However, Figure 11 shows two relevant facts:

- $T_{returnt}$ is several times higher than the other timestamps for both operation types and therefore has more impact on the overall response time.
- $T_{returnt}$ shows the behavior we witness for the overall time - increase in response time with the increase of memcached instances.

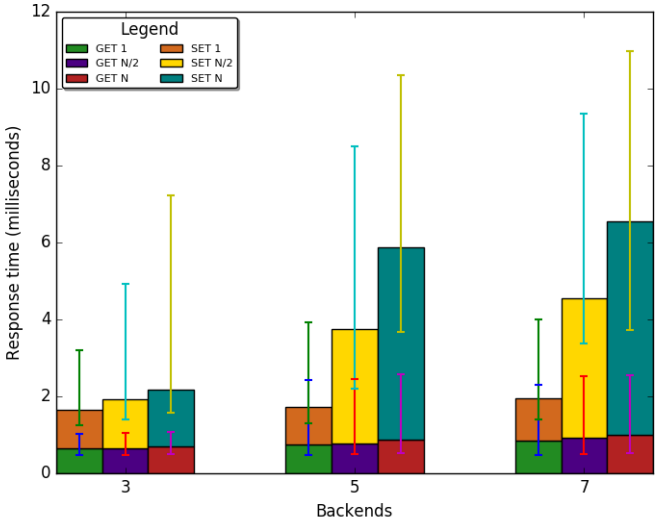

Figure 10: Server time for GETs and SETs

This behavior is due to an implementation decision of the middleware. When either of the GET or SET threads need to send a response, they first send it back to the same thread that receives requests from the clients. Thus both types of requests end up in the same pool(implemented as a queue) before finally being sent back. This creates an unnecessary contention point that could have been avoided, had the implementation included a separate component for sending responses. Moreover, when there are more servers, there are more responses being sent concurrently into that pool. As a result, there are no strong guarantees how long it will take for a response to actually be sent to the client, after arriving in the pool. An interleaving that occurs in this case is that a response is received from memcached, but several other responses from other servers get stored in the queue before it, so it will have to wait its turn. On the other hand, for less servers there is less chance that a response will be outrun by another response, simply because there are fewer of them trying to access the resource at the same time.

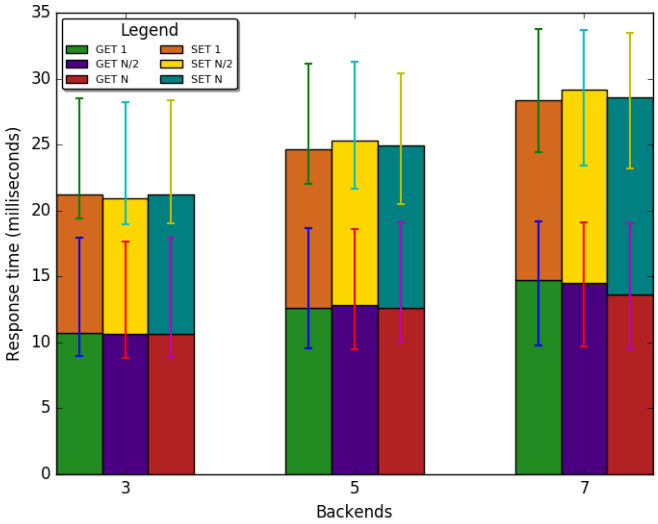

Figure 11: Return time for GETs and SETs

2.3.3 Scale-up against ideal implementation

Now that we have seen how the system behaves as the configuration changes, we can compare the performance to that of an ideal implementation. When talking about ideal implementation, the important factor to consider is replication. As explained in the hypothesis, in the ideal solution all servers would respond to a SET request simultaneously and therefore T_{server} would remain the same as if there was no replication.

A mathematical formula that can be given for representing the response time is:

$$T_{server} = x + \delta x$$

where δx is the time difference between how long it took for the slowest server to respond and the fastest. Thus, in an ideal world δx would be 0, but because that is not the case lets examine how the δx actually changes as the configurations change.

Scale up is the ability of a system to deal with larger loads when adding resources. In the context of the middleware a larger load is considered to be a higher replication factor and the resources are the number of backends. With that in mind, lets visualize how δx changes in Figure 12. The figure shows how the server time behavior, described in 2.3.2 differs from that of the ideal system.

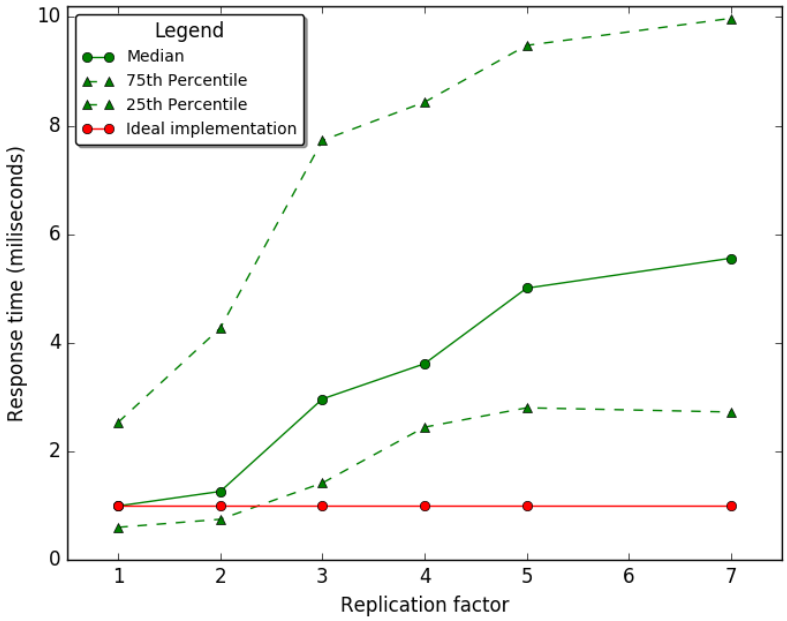

Figure 12: Server time scale up

From the Graph we see that the response time grows almost linearly as the replication increases. As already explained, the larger load increases the probability that one server will either take a longer time to respond or that the thread will be sending new requests before checking for incoming responses.

3 Effect of Writes

3.1 Experiment design

The goal of this experiment is to analyze how the behavior of the whole middleware changes with respect to the read-write ratio as opposed to examining the effects on each operation type in detail. However, the timestamps inside the middleware will still be used to back up the claims and results made for the middleware performance. Therefore, the main performance metrics, which will be used are throughput and response time.

Unlike the previous two experiments, this time we have a 3k factorial design. The first factor is **number of backend servers** and again the levels, which will be used for it are **3, 5 and 7 servers**. Similarly as before, this causes the number of client VMs to become a fixed parameter. Further to that, each VM will have 120 clients, since this number proved to be suitable for carrying out the experiments. The second factor, which we have already used, is the **replication factor**. However, since this time we are more interested in the consequences from increasing the number of writes, rather than examining the replication effect in much detail, we will now use only two values - **no replication of full replication**. Finally, we will also vary the **read-write ratio**. The values which will be used are **1%, 5% and 10%**.

Another fixed parameter, that is of interest here, is the sampling ratio used in the middleware. This cannot be changed between experiments to reflect the ratio used by the workload configurations, because different file access frequency will cause the results to be incomparable. To ensure, that a similar number of get and set requests will be logged, between experiments, we will write every 25th SET and every 500th GET.

Finally, each experiment instance will be run for 2 minutes in order to allow for enough samples to be collected, while also having a sufficiently large warm-up and cool-down phases of 10 seconds each. To get more accurate result, every experiment instance will be run 3 times.

Number of servers	3, 5, 7
Number of client machines	3
Virtual clients	360
Workload	Key 16B, Value 128B
Read-Write ratio	1%, 5%, 10%
Middleware Replication factor	None and All
Middleware Threads in pool	8
Runtime x repetitions	2 x 3
Sampling	Every 500th GET and every 25th SET
Log files	write_memaslap.tar.gz, write_parsed_memaslap_All.log
	write_parsed_memaslap_None.log, write_parsed_timestamps_All.log
	write_parsed_timestamps_None.log, write_timestamps.tar.gz

3.2 Hypothesis

Unlike the previous two experiments, where the hypothesis was made based on my understanding of the middleware implementation, this time a more informed prediction can be made. This is because a lot of relevant information was revealed during the previous experiment, whose purpose was to study the effect of replication on the two operation types.

As we have already seen, with the increase in the replication factor, the response time for set operation increases. This also corresponds to a decreased throughput by the middleware. Since SETs become more costly, increasing their number can only result in less throughput achievable by the the middleware. However, I expect this to happen simply because of the higher number of expensive requests and not because they will become more expensive.

With that in mind I expect that when there is no replication, increasing the number of backend servers will not increase the overall time for processing a SET. However, I expect a difference between the cases where there is no replication for X number of servers and full replication for the same backend number. Therefore, I expect that when we add a high SET write to that configuration, the middleware will be most overloaded.

In general, performance will remain steady as we add more SETs for no replication, but as soon as the middleware has to send requests to all backends, there will be a drop in performance

3.3 Experiment results and Analysis

3.3.1 Overall Middleware performance

In order to correctly analyze the behavior of the middleware we first have to define the base configuration. This will be the case where there are 3 backend servers and no replication and 1% writes.

The next thing to do is examine how the overall behavior of the middleware changes with respect to all different configurations. The data used to visualize the behavior shown in Figures 13 and 14 is stored in *write_memaslap.tar.gz*, *write_parsed_memaslap_None.log* and *write_parsed_memaslap_All.log*.

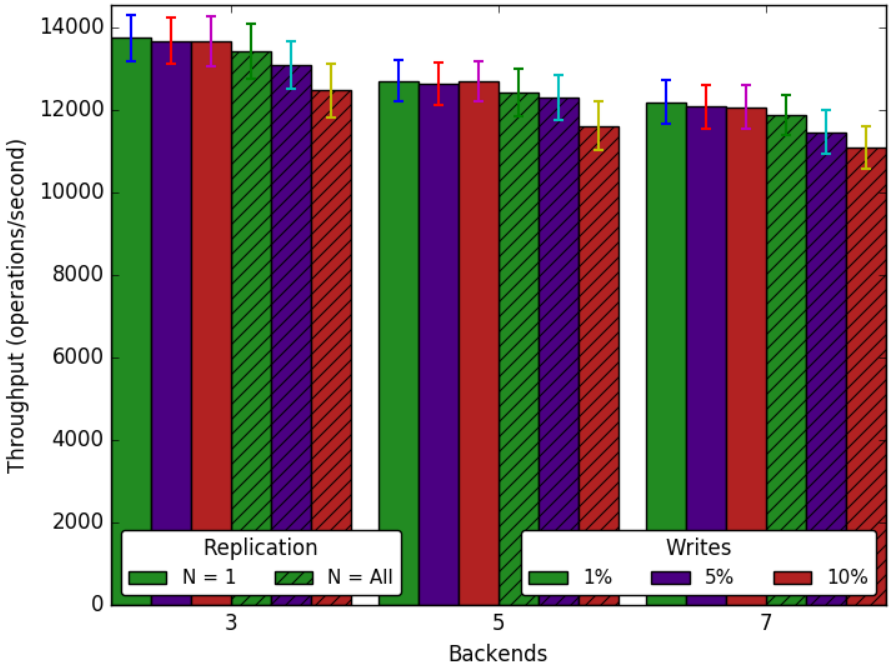

Figure 13: Overall Throughput of Middleware

In the previous experiment we reached the conclusion that set operations are costlier than GETs when there is replication and therefore in the hypothesis it was suggested that the more SETs there are, the less throughput will be achieved by the middleware. Indeed, with the increased number of SETs and backend servers, in the case where there is full replication, the throughput of the middleware decreases because it has to send and wait for more responses, as suggested in the hypothesis. Furthermore, another proof that replication causes set requests to become more expensive to handle, is the fact that the TPS is consistently less in the cases where there is replication, as opposed to the same experiment configuration, where there is no

replication. Thus, as far as throughput is concerned, the read-write ratio has the biggest impact on a system configuration with 7 backends and full replication.

From inspecting the case with no replication we see that TPS remains stable when increasing the number of SETs, but drops with the increase of backend servers. This is a confirmation of how $T_{returnt}$ affects performance, as explained in the last part of experiment 2.

Lets now look at how the response time changes in Figure 14. It shows exactly what was to be expected after examining the TPS results. The response time gradually grows as the write percentage is increased. This can again be explained by the results from experiment 2 which state that the response time for set requests increases with the replication factor and the number of backends. Thus, it is normal that the more SETs there are, the bigger that time will be. Again, when there is no replication, the time grows only for the number of backends, because of $T_{returnt}$. This again confirms that configuration with the biggest effect relative to the base case is when the system has 7 backends, 10% writes and full replication. This discovery complements the hypothesis by proving that the number of backends contributes to the impact, in addition to the read-write ratio.

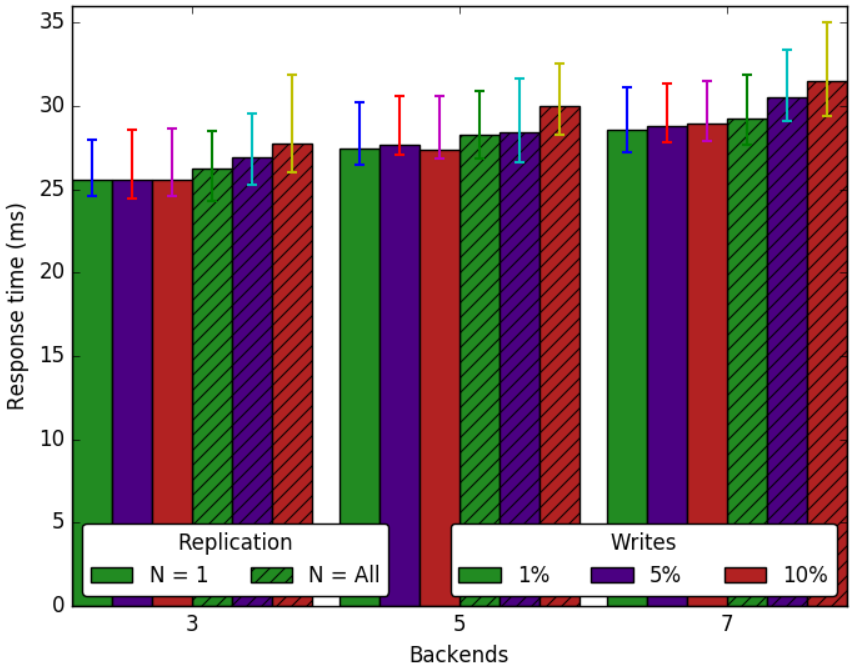

Figure 14: Overall Response time of Middleware

3.3.2 Timestamps

In order to see how the write ratio changes the way sets are handled, the timestamps from the middleware have to be explored. As discovered from experiment 2, set requests are the ones that are impacted to a much greater extent by the factors of the experiment. Even the new factor - read-write ratio, makes the overall middleware behavior more dependent on SETs. That is why when comparing the difference between the base case and the configuration with the biggest impact, will consist mainly of examining how the behavior of SETs within the middleware changes.

The timestamps which are going to be examined are again T_{queue}, T_{server} and $T_{returnt}$ and the files which contain the data are *write_timestamps.tar.gz*, *write_parsed_timestamps_None.log*

and *write_parsed_timestamps_All.log*. Let's first begin in T_{queue}, whose behavior is visualized in Figure 15.

In the cases where there is no replication, the queue time is highest for the maximum number of writes, but decreases as the number of backends grows. The later point is the same behavior that was seen from experiment 2 and this is caused because of the load balancing done on queues. The former point is also intuitive to explain, because having more SETs means there are more entries in the queue. Even though the thread is asynchronous it still has to take off requests, listen for responses and then go back to polling the queue. As the write ratio grows, the thread will have more responses to listen to and its work is going to be divided between sending and receiving.

Now, to compare the base case with the most affected configuration, the full replication cases are examined. Expectedly, full replication causes the queue time to grow even when the number of backends is the same. The most interesting thing worth noting here is that even with full replication, the queue time decreases as more backends are used, instead of increasing. This means that although there are more SETs, increasing the number of backend still aids performance. Moreover, T_{queue} still remains small compared to other timestamps. Therefore, the conclusion is that the write ratio has some effect to on the impact of servers as opposed to the base case, but is not the most important factor.

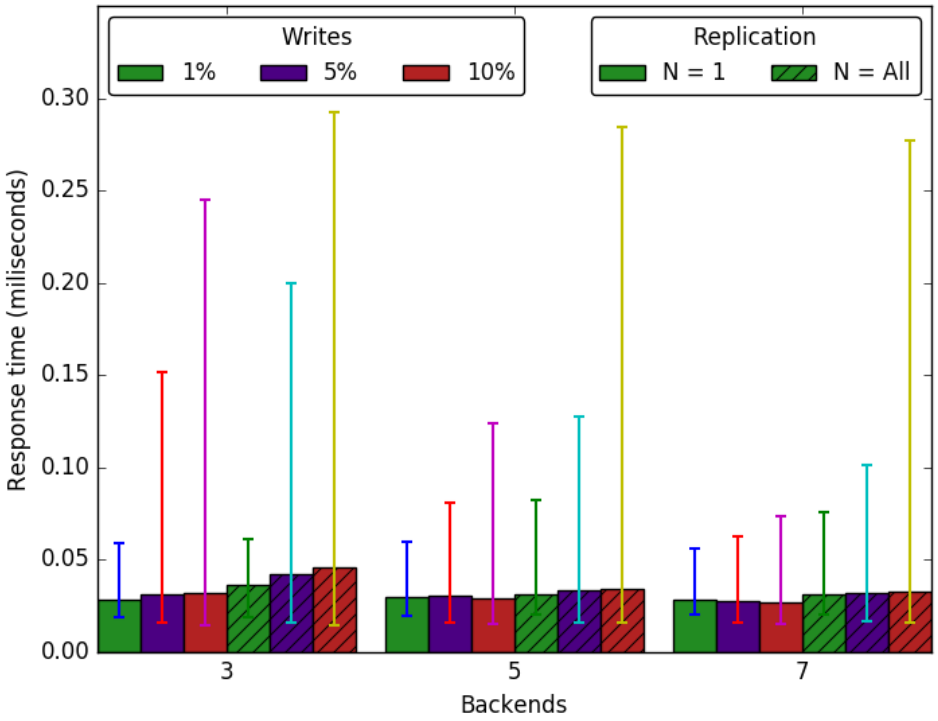

Figure 15: SET request Queue time

T_{server} shows some similar behavior for the case when there is no replication as seen from Figure 16. The bigger number of writes causes the response time to go up when the number of backends is the same, but T_{server} goes down when backends are increased. This is again due to load balancing.

In the case of full replication, the read-write ratio again has a noticeable effect on the response time. The increase is caused by the fact that the SET thread will have to deal with a bigger queue. Because of the way it is implemented, the thread will poll the queue, send the requests to the server and then poll again before checking for responses from the server. If

polling the queue and checking for responses were alternating events, then the bigger number of writes would have had less impact, when the number of backends remains the same. Thus, the hypothesis that SET operations will not get more expensive is invalidated for T_{server}.

Unlike with T_{queue}, increasing the number of servers, i.e. doing more load balancing does not stop the response time from increasing. The reason for that is simply because, more servers mean bigger replication. Thus, there are more requests being sent which makes us come back to how T_{server} is calculated. Just like in experiment 2, the more servers we are waiting for a response from, the higher the likelihood that one of them will take more time.

The conclusion for this timestamp is that the read-write ratio has an effect on it, such that T_{server} exhibits the behavior seen in the overall response time. Thus, T_{server} is partly responsible for the difference between the base case and the most affected configuration.

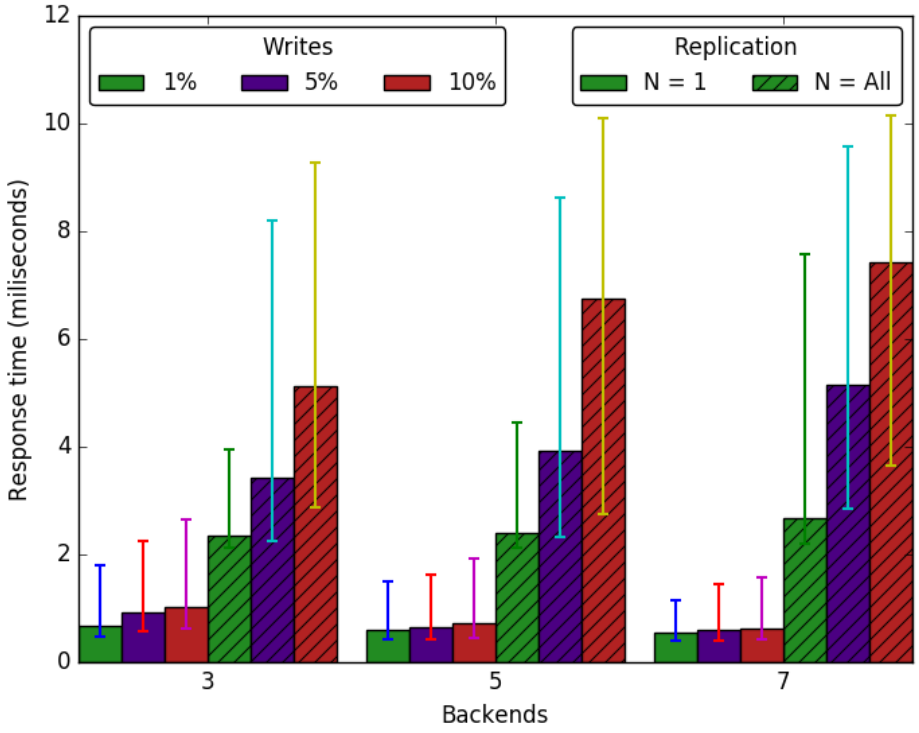

Figure 16: SET request Server time

Finally, lets examine $T_{returnt}$ since from the previous experiment we found that it is the timestamp that takes the most time.

First of all, the response time for all configurations with an equal number of backends remains similar. From experiment 2 we have seen that the replication factor does not affect $T_{returnt}$ because there is always just one response being sent all the way back to the client. However, it turns out that the read-write ratio is also irrelevant to $T_{returnt}$. Although this may seem counter-intuitive at first, there are two reason both of which have to do with the implementation. First of all, the pool that contains all responses, which are ready to be sent back to the client, contains both get and set requests. This means that although the number of SETs is increased, the size of the pool will remain the same, because the number of GETs is reduced. Thus, the sending thread is not put under a bigger load. The other reason is that all responses from a particular SetThread, will always come in the correct order. Thus, all requests from one server will always take similar time, until they get sent back.

Finally, just like in the previous experiment, $T_{returnt}$ increases with the number of backends as already described. Therefore, the read-write ratio does not affect the return time of a

request and is not responsible for the difference between the base case and the most affected configuration.

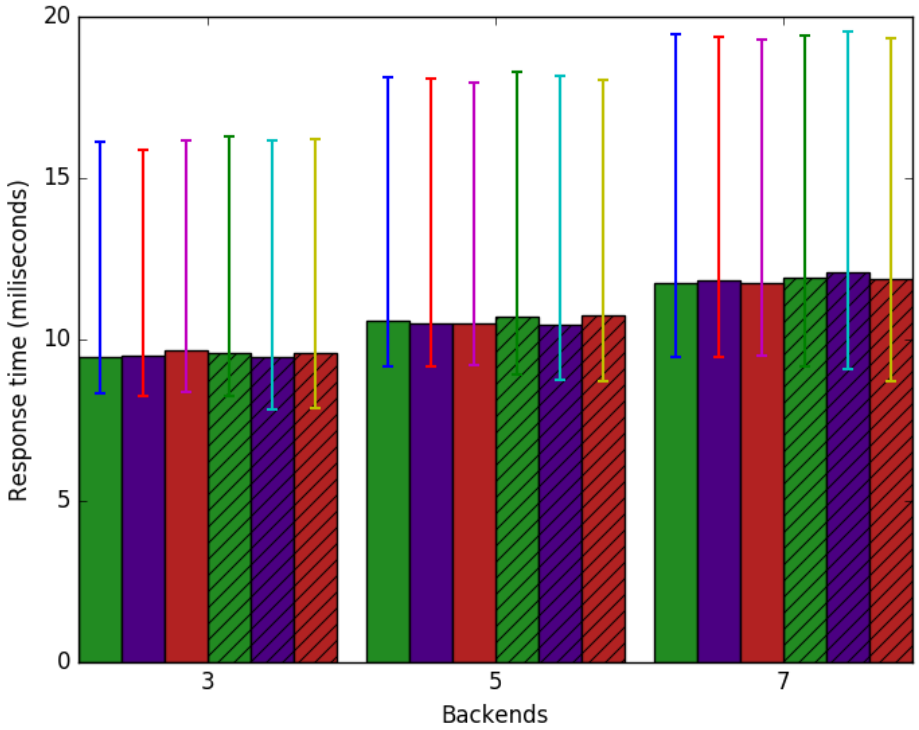

Figure 17: SET request Return time

The final conclusion is that T_{server} and $T_{returnt}$ both contribute towards the impact on the base case since they exhibit the same behavior as the overall middleware performance and are the two most time consuming operations as well. However, if only the read-write ratio factor is considered, then T_{server} is the main reason for the impact in performance, relative to the base case.

*For this milestone a minor bug in the code was fixed, that caused a small number SET requests to get dropped under a heavy load.