Advanced Systems Lab (Fall'16) – Second Milestone

Name: *Teymur Babayev*
Legi number: *15-926-231*

Grading

Section	Points
1	
2	
3	
Total	

Introduction

In this report first of all I'd like to mention changes done to the code:

- Formatting of instrumentation has been changed to ease parsing process - in first version I was using Java's SimpleFormatter, now I implemented CustomFormatter which writes middleware logs in .csv format. Change in code can be seen in MiddlewareServer class.
- Object loggerMap in MessageProcessor changed it's type from HashMap to ConcurrentHashmap. Very rare concurrency issues have been observed only when using more than 160 clients per memaslap machines (I used total of 3 memaslap machines).
- Method socketChannel.write() in AsynClient was performing partial writes (never happened with less than 176 clients / VM). Because of that, this method was put into while loop with condition byteBuffer.hasRemaining().
- Class Message got changed static fields to conform with new formatting strategy.

Since neither of these changes may break functionality, and in contrast it makes system behavior more robust, there was no need to rerun stability trace as in milestone 1 to ensure that system is capable of not breaking.

I'd also would like to clarify which names I use for timestamps for internal MW logs: total time spent in MW is T_{total}. Time spent in queue is T_{queue}. Time required to process request by server is T_{server}. Finally, I introduce last timestamp which is called T_{left} and it is calculated as T_{left} = T_{total} - T_{queue} - T_{server}. This T_{left} consists of 2 parts: hashing request and putting it into queue and time between receiving response from memcached and actually sending it back to memaslap. Since it is really important to know how T_{left} is split between these 2 operations, I conducted microbench experiment.

I changed code of my system to measure also time spent for hashing. Code was correct since I didn't introduce any new time measurements - hashing time was calculated as $T_{queue.add}$ - $T_{request.received}$ (following notation for timestamps from milestone 1). Experiment setup was totally same as setup in experiment 2 of this milestone (refer to table), apart from fact that I was not increasing number of replicas. I got following results: hashing time for 3 servers = 6.8 microseconds, for 5 servers = 9.9 microseconds and for 7 servers = 7.26 microseconds. Apparently, hashing is superfast and makes less than 0.1% of T_{left}. It means that sending reponse back to memaslap is slow, and this is explainable due to architecture of my system, where working threads of "clients" call MiddlewareServer's method send(), which puts response into local queue in MiddlewareServer class. Following is typical Java NIO architecture with iteration over selected keys and it all makes sending requests back slow. This information will be useful in following experiments of milestone 2, and we will often refer to it. Refer to logfile "hash_test" to observe the results of this microbench.

In the end of the introduction section I would like to mention inconsistency of Azure behavior. Experiment ran of different day or even different time of the day produce totally different results. It is not only about network lag, but the internal speed of VM's is different, despite **system code is same**. This leads to inconsistent results across experiments. In fact, experiment ran in a single batch always produces same patterns but scale of overall performance is different, that's why comparing performance across experiments is in general hardly possible.

1 Maximum Throughput

Before committing a maximum throughput experiment, the following hypothesis has been made:

Increasing a number of clients will lead to increase in throughput until a certain point, where middleware performance reaches a ceiling. Afterwards, increase in the number of clients won't cause any increase in TPS - since we are in the closed system context, we will constantly have some requests left in the queue, hence, response time will increase, but TPS will stay the same.

Increase in the number of threads will also lead to increase in TPS up to a certain point, but after that point we will probably observe a degrading in performance due to the overhead created by excessive amount of threads. Reaching that certain ”point” means adding more threads will not add any value, probably make things worse.

It means that we are looking for the number of clients, after which maximum TPS saturates, and number of threads, after which maximum TPS degrades.

Following setup has been used for maximum TPS experiment:

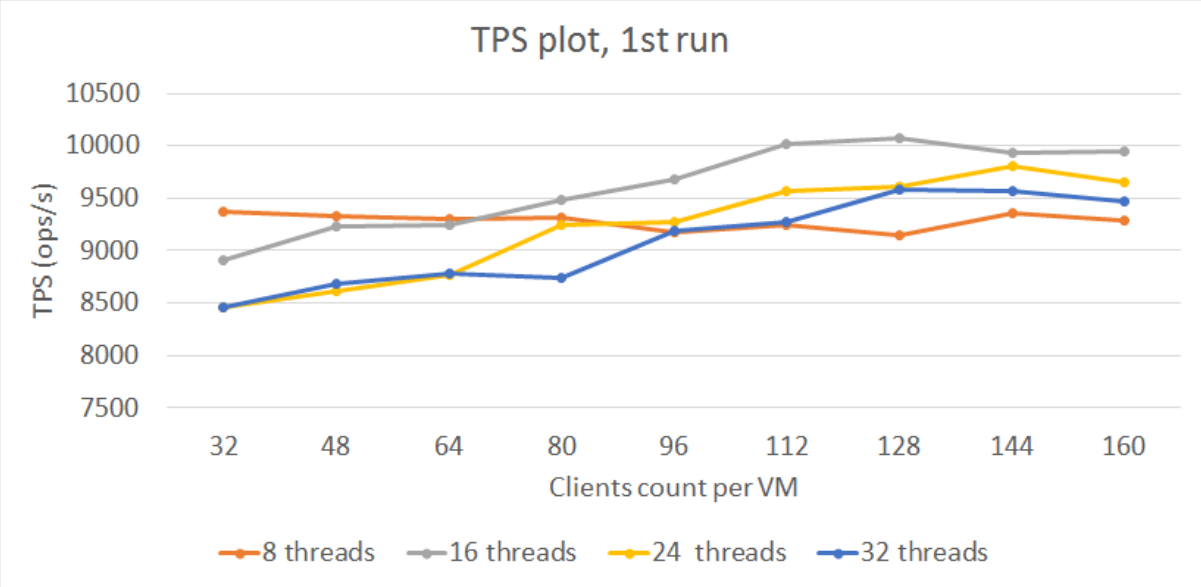
Memcached VM's	5
Memaslap VM's	3
Replication factor	1
Virtual clients / VM	32 to 176
Number of threads PTP	8 to 32
Workload	Key 16B, Value 128B, Writes 0%
Runtime x repetitions	90s x 4-5
Warm-up, cool-down period	20s
Sampling ratio	1 of 100 requests
Window size	1k
Memaslap output	once in 5s
Log files	experiment_1, exp_1_total

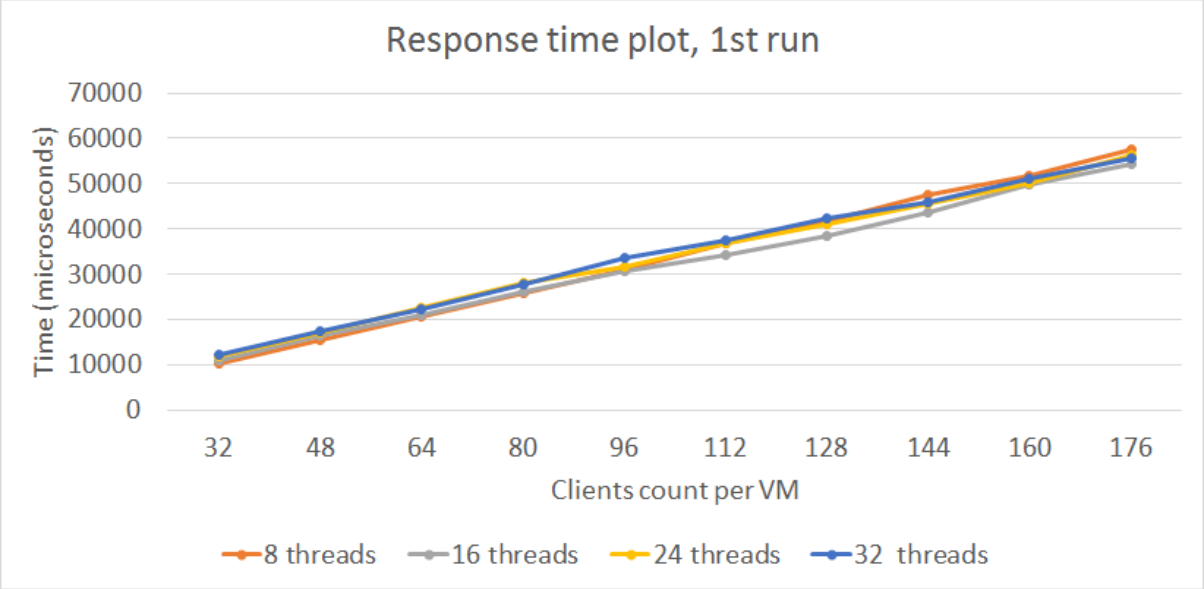
Despite 5 VM's were available as client machines for this experiment, decision has been made to use 3 VM's to stay consistent with the second and third experiment, were no more than 3 memaslap machines are available. To obtain some measure of variation for our data and get an estimate of expected value of TPS / response time, experiments were repeated 4 times. Decision to repeat experiments 4 times has been made to obtain 95% confidence intervals within 10% of the mean of TPS, which was achieved (see later). Experiment time was defined to be 90s, cutting of first and last 20s as warm-up and cool-down period. During the first milestone, warm-up / cool-down period never exceeded 10s, but since we are in the setup with higher loads, 20s period was considered to be a good choice. The left 50s are enough to get meaningful results from the experiment with memaslap output once in 5s.

Window size for memaslap was set to 1k, since at default value of 10k memcached servers got overloaded, started evicting keys and lead to *get_misses*, which would corrupt our experiment.

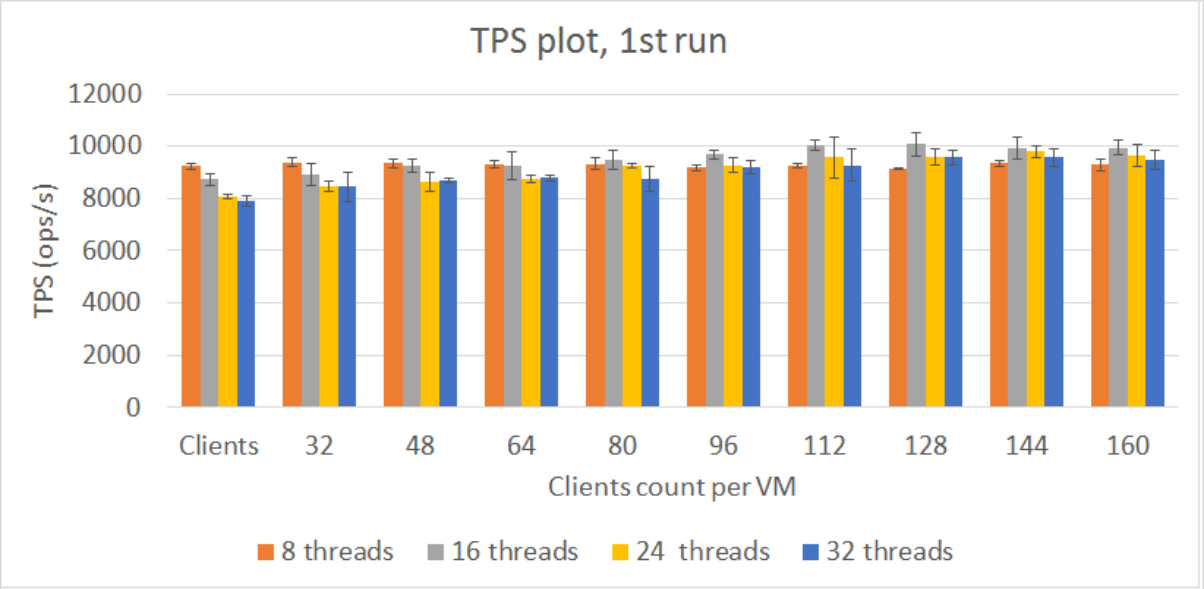
In the first run of the experiment, number of threads was increased from 8 to 32 with step size of 8. Such a step size seems reasonable choice since our MW runs on the 8-core VM. Number of virtual clients per machine was increased from 32 to 176 in step size of 16, meaning that we tested ~100 to ~530 virtual clients in total with step size of ~50.

The following 2 plots demonstrate TPS and response time for the first run of the experiment:





Since adding standard deviation bars would make these plots unreadable, following are TPS bar plot with std bars and table with memaslap log. dist. information, which gives general idea about percentiles of response time:



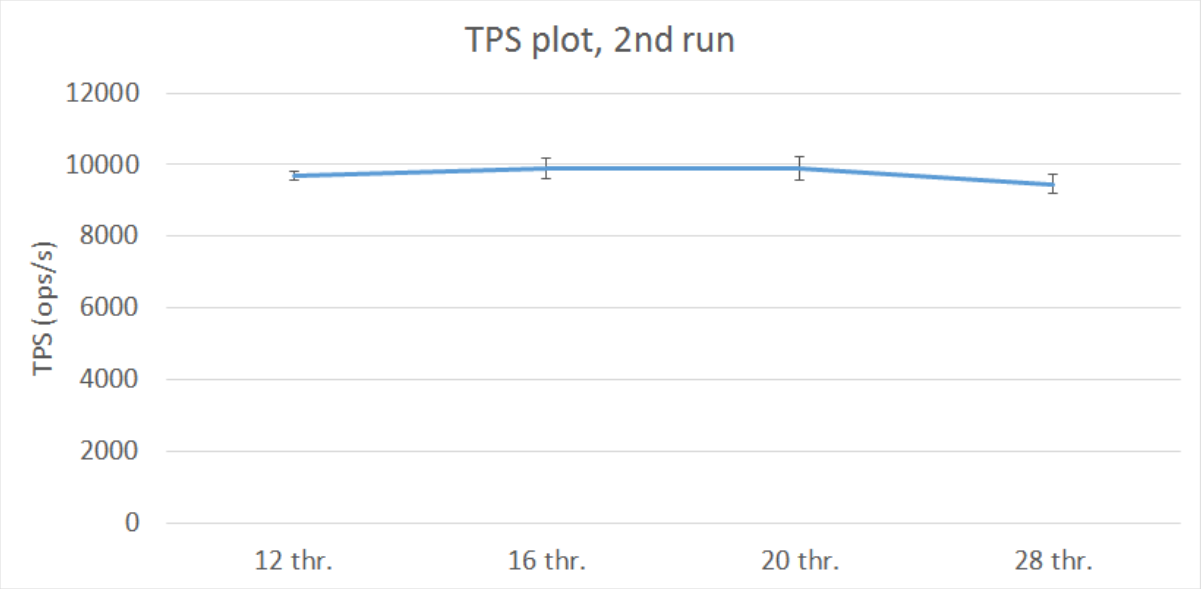
	Time(us)	Clients count per VM			
		96	112	128	144
8 threads	8192	4.80%	5.21%	3.71%	3.42%
	16384	23.31%	22.75%	18.41%	19.44%
	32768	66.12%	57.44%	49.86%	49.22%
	65536	94.31%	88.92%	84.70%	81.72%
16 threads	8192	2.25%	1.15%	0.96%	0.85%
	16384	19.25%	12.00%	11.24%	9.79%
	32768	68.94%	54.78%	48.17%	43.57%
	65536	97.13%	94.32%	91.84%	88.27%
24 threads	8192	2.36%	1.27%	0.71%	0.73%
	16384	20.95%	13.97%	9.65%	8.56%
	32768	69.00%	57.87%	48.51%	42.15%
	65536	96.05%	93.44%	90.69%	87.23%
32 threads	8192	2.09%	1.12%	0.92%	0.67%
	16384	18.30%	13.13%	11.62%	9.06%
	32768	65.48%	56.18%	50.95%	43.44%
	65536	94.56%	91.94%	89.65%	86.60%

Memaslap logdist data

As it is seen on the provided plot / table, behaviour of TPS and response time follows our hypothesis. It is well seen, that saturation point for 8 threads PTP is reached in the very beginning. 16 threads PTP results are showing the best performance, meaning that 16 is probably the best choice for the number of threads. It is well seen that the peak for maximum TPS is reached with 128 virtual clients per VM. One could argue that 144 virtual clients give better performance, but the reported TPS for 128 clients is 10014 ops/s and for 144 clients is 10069 ops/s, and this difference is insignificant. After 144 clients, it is clearly seen that saturation point is reached for 16 threads per thread pool.

24 and 32 threads PTP are showing worse results, than 16 threads PTP, meaning that these amounts of threads create an overhead in the MW, and definitely are not demonstrating the best achievable TPS. As well, in both cases saturation is clearly seen - with 24 threads no significant increase is seen after 128 clients per VM, and with 32 threads - after 144 clients per VM.

Results show us that combination of 16 threads PTP and 128 clients per VM gives us the best achievable TPS of ~10000 ops/sec. However, decision has been made to also test the behaviour with 12, again 16, 20 and 28 threads PTP to ensure that these numbers of threads do not give better results than 16 threads PTP. Following plot and table are showing TPS and response time for these amounts of threads with number of clients being fixed as 128 per VM:

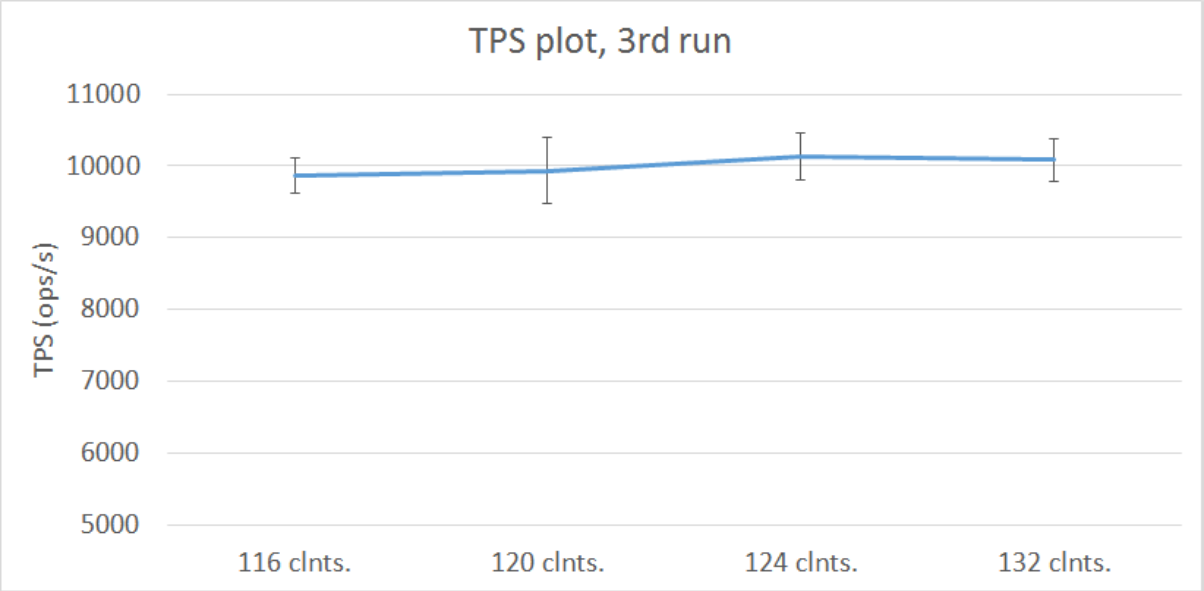


Time(us)	12 threads	16 threads	20 threads	28 threads
16384	10.72%	11.37%	10.27%	10.55%
32768	47.49%	50.34%	49.57%	49.40%
65536	90.89%	91.79%	91.26%	89.60%

Memaslap logdist data

Despite the result of 12 threads is pretty close to the one of 16 threads, there exists difference in 200 ops/sec - 9691 for 12 threads and 9881 for 16 threads. Also, neither of 4 repetitions of 12 threads test exceeded 10k TPS, while for 16 threads it reached more than 10k 2 times out of 4. It is worth to mention that difference in TPS for 16 threads PTP is totally due to the fluctuation in Azure network, since 16 thread run in this run followed absolutely same setup as in first run. So, final decision for the best number of threads has been made - 16 threads PTP give the best result for throughput.

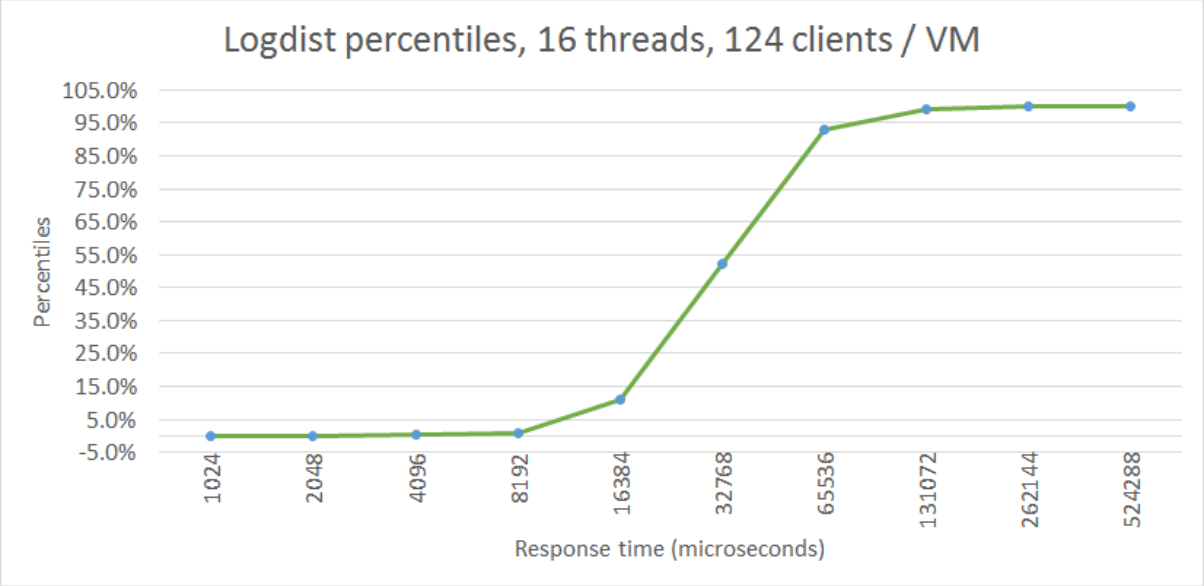
Third run has been also conducted. Since in this question we are required to find the optimal number of virtual clients, rounded to 10, and in the first run step size of 16 per machine (~50 in total) was used, third run was done to determine the exact number of virtual clients to achieve the best TPS. After fixing the amount of threads per thread pool to 16, 116, 120, 124, 132 virtual clients per VM were tested. Also, 5 repetitions instead of 4 were used to increase accuracy of output. Following are plot and table demonstrating TPS and response time as a function of virtual clients:



Time (us)	Clients count per VM			
	116	120	124	132
16384	12.10%	11.75%	10.94%	10.44%
32768	55.30%	53.12%	52.11%	48.71%
65536	94.34%	93.35%	92.93%	91.16%

Memaslap logdist data

It is clearly seen from the plots, that the actual peek is reached at 124 clients per VM. The result for 132 clients per VM is worse, meaning, that at 124 clients the reached the ”hill top”. Eventually, highest throughput of my system is achieved with total number of virtual clients equal to ~370 and size of the thread pool equal to 16. Following is a percentile plot for response time of our best configuration, obtained from memaslap logarithmic distribution data:



It is observed that median is actually lower than 32ms and we obtain 90th percentile to be below 100ms, which proves that this configuration is a good choice.

The difference in rate of growth of response time is the main factor which defines where the saturation / "hill top" point will be reached. According to the Interactive Response Time Law, if our response time grows as fast as the number of clients, we will have stable throughput (neither increasing, nor decreasing). The reason why we have increase in TPS at first when increasing the number of clients (for our case of having 16 threads PTP) is that at first our response time increases slower than number of clients. However, at some point it starts to grow faster. Exactly at this point the saturation of TPS starts, which can be observable on the TPS and response time plots / tables. To find a clear reason why response time starts to grow faster we have to look up into the instrumentation logs. Analyzing only one log (16 threads, 124 clients) will yield no meaningful results, that's why it is important to check the logs at points around this configuration. One should mention that all following instrumentation timestamps in context of maximum TPS experiment are for *get* requests. Following table provides mean timestamps from the instrumentation logs for 112, 124, 128, 144 clients per VM and their offset (difference in response time) from memaslap:

	Op. type	T (total)	T (queue)	T (server)	T (left)	Offset from MSL
112 clnts.	get	23348.53	7951.73	2328.70	13068.10	10828.31
	set	16371.32	2189.97	6665.45	7515.90	
124 clnts.	get	24900.44	9455.53	2486.61	12958.30	12389.81
	set	16753.54	2140.22	7160.61	7452.71	
128 clnts.	get	25358.57	9590.12	2340.81	13427.64	13285.47
	set	18268.58	2314.21	6608.37	9346.01	
144 clnts.	get	28452.13	12160.03	2411.55	13880.55	15176.10
	set	19375.41	2308.06	6783.36	10283.99	

Mean timestamps

It is seen from the table above that there is more increase in T_{queue} from 124/128 to 144, than from 112 to 124/128. It means that queue almost never gets empty and always has some number of requests in it. Increasing number of clients we also increase number of these requests, creating additional overhead in the system. Also, offset from memaslap adds some value to the increase in response time. The growth in offset is linear, which comes along with adding more load to network with linear increase in clients number. Biggest component of response time, however, is T_{left}. As stated at the beginning of report, negligible part of this time is spent on hashing, and almost all of it is spent on sending response back to memaslap. It is logical to observe increase in this time with adding more clients since iteration over selected keys in MiddlewareServer class will take more time - our MiddlewareServer employs only 1 thread. T_{server} is pretty constant and is the minor component of response time.

While mean T_{queue} might not be the best metric to support our thoughts, following is the table of 25th, 50th, 75th and 95th percentile for T_{queue}:

T (queue)	Op. type	25%	50%	75%	95%
112 clnts.	get	170.75	3970.00	10820.25	28108.13
	set	2.50	337.00	2154.75	9312.25
124 clnts.	get	609.25	6074.63	12499.56	31791.69
	set	2.00	306.38	2413.81	9285.40
128 clnts.	get	428.00	5916.25	12695.69	32964.56
	set	3.00	383.50	2554.50	9463.45
144 clnts.	get	759.44	7335.13	16266.38	41683.48
	set	2.50	382.38	2528.00	9529.55

T_{queue} percentiles

These numbers follow trend of the discussions above and support main ideas.

Possible solution to increase in T_{queue} would be increasing the number of threads PTP. However, provided plots do not prove that adding more threads yields better results. Following is the table with mean times from instrumentation logs for thread pool size 12, 16, 20 and 24, while number of clients/machine is 128. No experiments have been run with thread pool sizes 12, 20, 24 for 124 clients (they have not been run in the same time, and running them in different time would give different numbers because of constant fluctuations in Azure performance), however, difference in 12 clients should make no significant impact for the purpose of this analysis.

	Op. type	T (total)	T (queue)	T (server)	T (left)	Offset from MSL
12 thrds.	get	25965.95	12025.86	2306.38	11633.71	14085.75
	set	17441.70	2197.98	6750.59	8493.14	
16 thrds.	get	24619.86	9532.76	2461.41	12625.69	14851.11
	set	17412.99	2261.26	6896.10	8255.63	
20 thrds.	get	24878.77	8345.50	2590.64	13942.64	14589.33
	set	17272.63	2248.00	6981.47	8043.16	
24 thrds.	get	25383.79	7227.31	2867.11	15289.37	15573.85
	set	17954.12	2241.76	6812.91	8899.45	

Mean timestamps

It's seen from the above numbers that increasing thread pool size has a positive impact on T_{queue}. However, adding resources to one specific part of the system creates overhead in the performance of other components. All the operations where threads in thread pool do not cooperate (and the only place where they cooperate is while taking element from the queue) are slower, including memcached response time. This can be explained due to the fact that in our architecture each thread in thread pool uses different socketchannel to memcached - increasing TP size we increase # of connections to memcached. Also, introducing more threads, we increase the cost of threads' "context switch". These 2 facts explain an increase in T_{server}. Altogether, context switch is the main reason why almost everything in the MW becomes slower. Offset from memaslap output is stable with 12, 16, 28 threads and a bit higher in 24 threads - it is because 24 threads results are from the first run of experiment (3rd run was done 6 hours later than 1st). Increase in T_{left} (sending back) is approx. same as decrease in T_{queue} - we can see this trade-off. Sending responses back seems to be the slowest part of the MW - if it would take less time, the impact of 20 and 24 threads would be insufficient to produce response time higher than one in 16 threads setup, and the maximum TPS would be higher with saturation achieved with higher number of clients than what we have now.

Percentile table as before follows and supports above ideas:

T (queue)	Op. type	25%	50%	75%	95%
12 thrds.	get	644.38	7696.88	16607.81	42994.58
	set	2.00	296.25	2463.50	9307.25
16 thrds.	get	612.31	5910.50	12593.94	31849.91
	set	2.00	315.38	2426.06	9398.30
20 thrds.	get	460.25	4771.13	10645.63	26809.21
	set	2.00	333.75	2488.88	9411.65
24 thrds.	get	257.69	3744.00	9579.00	23719.59
	set	2.50	371.75	2346.25	9409.10

T_{queue} percentiles

Before concluding the discussion of the experiment, one should mention that confidence intervals supporting precision of 10% around mean for TPS were calculated and can be found in **exp_1_total** logfile.

Last addition to the 1st experiment is sanity check using IRT Law. Considering that RT should be equal to *Total number of clients / TPS*, we put numbers for the best configuration and get *372 / 10128 = 36.7ms*. This is close to result we got from memaslap output which is equal to 37.2ms: 0.5ms difference is insignificant. In fact, IRTL holds for all values in this experiment.

Lognames convention: memaslap log names follow next scheme: trace_log_*clientspervm_repetitionid*

Instrumentation log names follow next scheme: instrum_*threadpoolsize_clientspervm_repetitionid*.csv

Directory structure is intuitive.

2 Effect of Replication

Hypothesis:

The effect of servers number increase should balance the load across the servers, fill queues slower and thus, increase overall TPS and decrease response time both for *get* and *set* requests.

Replication factor increase should have no affect on *get* requests, since replicating accounts only for write op's, not read op's. The *set* requests, however should see an increase in response time and overall TPS and response time should decrease with higher number of replications.

Following setup has been used for replication effect experiment:

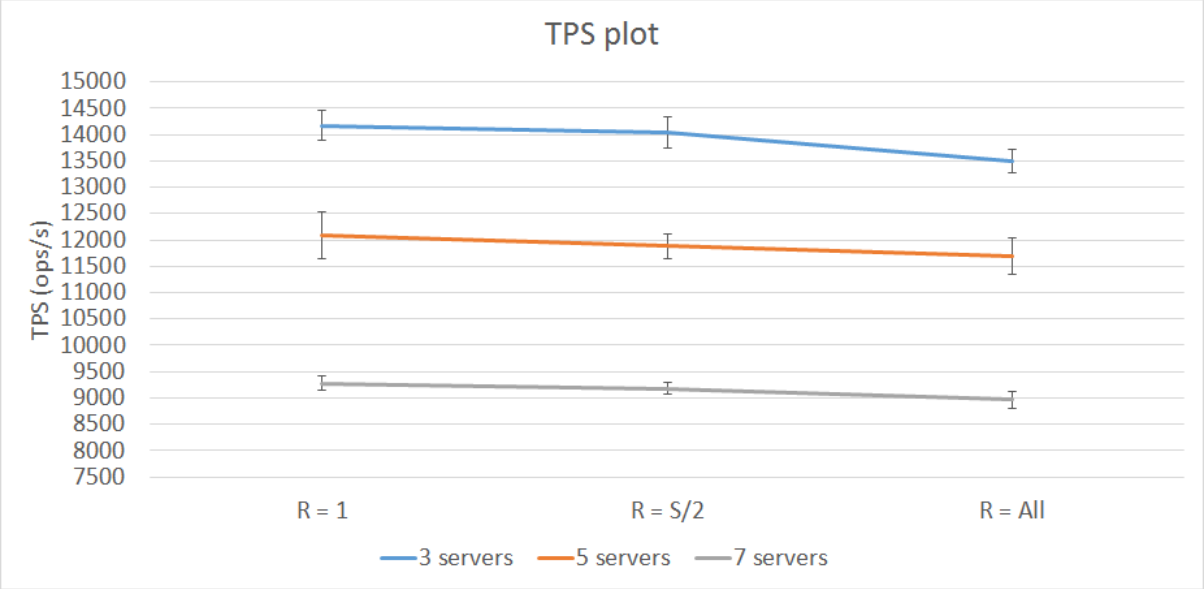
Memcached VM's	3, 5, 7
Memaslap VM's	3
Replication factor	1, $\frac{S}{2}$, all
Virtual clients / VM	124
Number of threads PTP	16
Workload	Key 16B, Value 128B, Writes 5%
Runtime x repetitions	90s x 4
Warm-up, cool-down period	20s
Sampling ratio	1 of 100 requests
Window size	1k
Memaslap output	once in 5s
Log files	experiment_2, exp_2_total

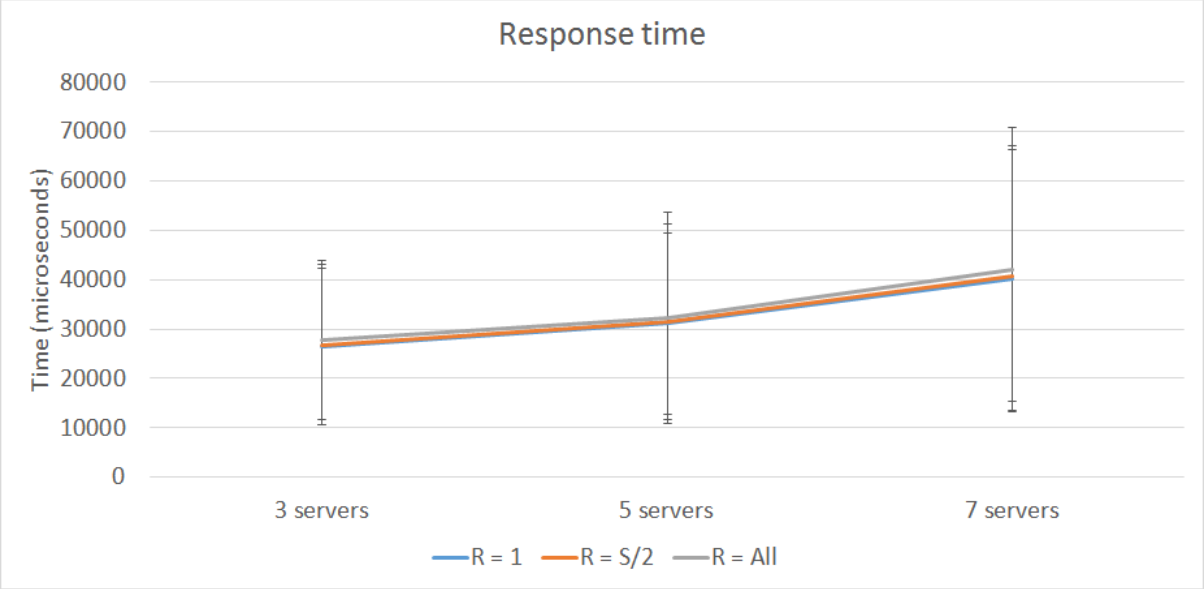
Since combination of thread pool size of 16 and 124 clients per VM yielded best TPS result and optimal usage of system resources in first experiment, decision has been made to use same combination for second experiment.

Runtime, # of repetitions, warm-up / cool-down period, sampling ratio, memaslap output frequency are same as in maximum TPS experiment since choice of these parameters provided enough information to analyze information. Window size was left to be 1k to stay consistent with first experiment.

All other parameters in the table above are just following the hard requirements of this experiment setup. The replication factor $\frac{S}{2}$ was ceiled, so for S = 3 it equals 2, for S = 5 it equals 3, for S = 7 it equals 4.

The following 2 plots and table demonstrate TPS and response time obtained for this experiment:



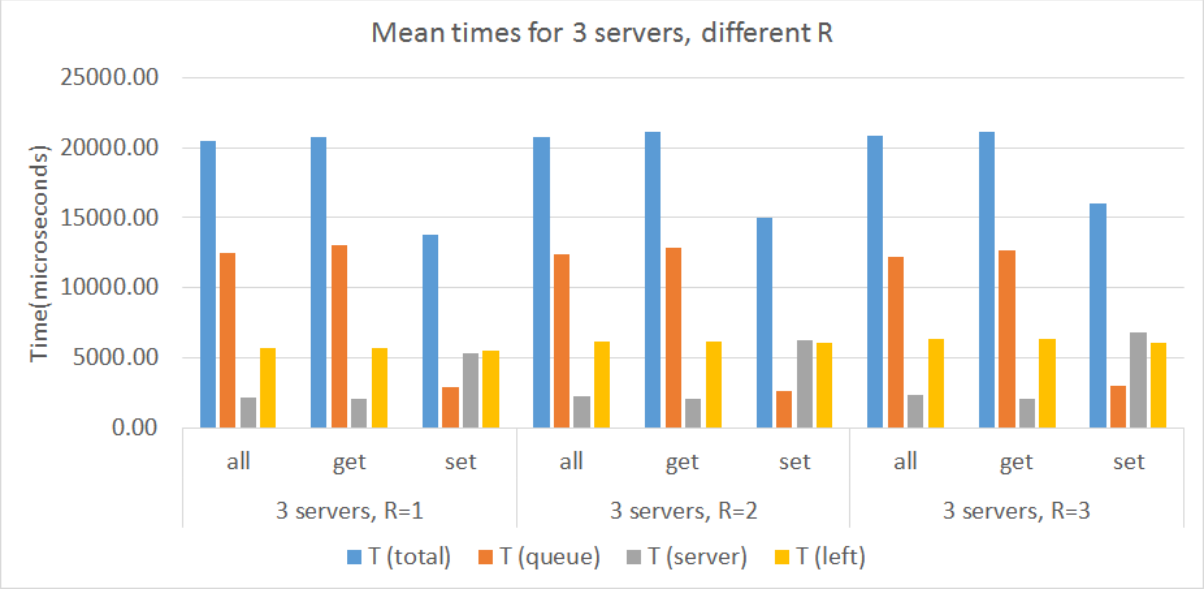


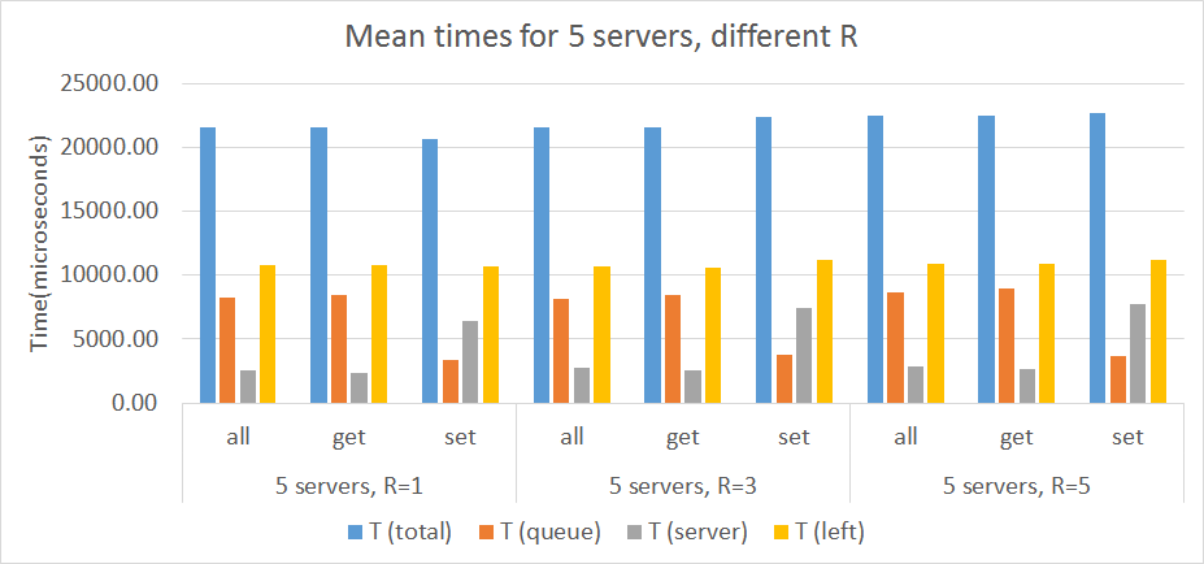
	3 servers			5 servers			7 servers		
Time(us)	R = 1	R = S/2	R = all	R = 1	R = S/2	R = all	R = 1	R = S/2	R = all
16384	24.0%	21.7%	21.0%	15.3%	15.1%	13.8%	9.5%	8.3%	8.2%
32768	74.2%	72.6%	72.4%	63.6%	63.4%	61.0%	46.6%	45.0%	44.3%

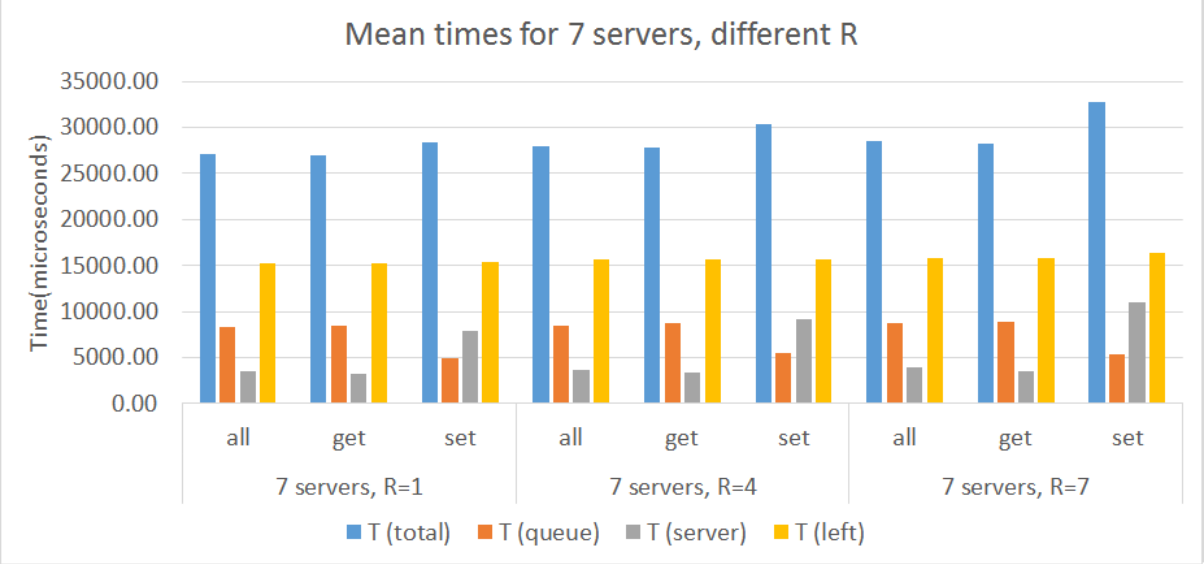
Memaslap logdist data

It is clearly observed from the plots above that part of our hypothesis regarding replication effect is correct - indeed, increasing a replication factor has a negative impact on performance of the system. However, other part of our hypothesis regarding changing the amount of servers failed. To discover why we've got such results and to answer main questions in this experiment we will deal with information from instrumentation logs.

First the analysis in change of behavior when increasing the number of servers will be done. In order to discover what exactly creates a decrease in TPS and increase in response time, following plots showing mean time measures for different number of servers with different replication factors:



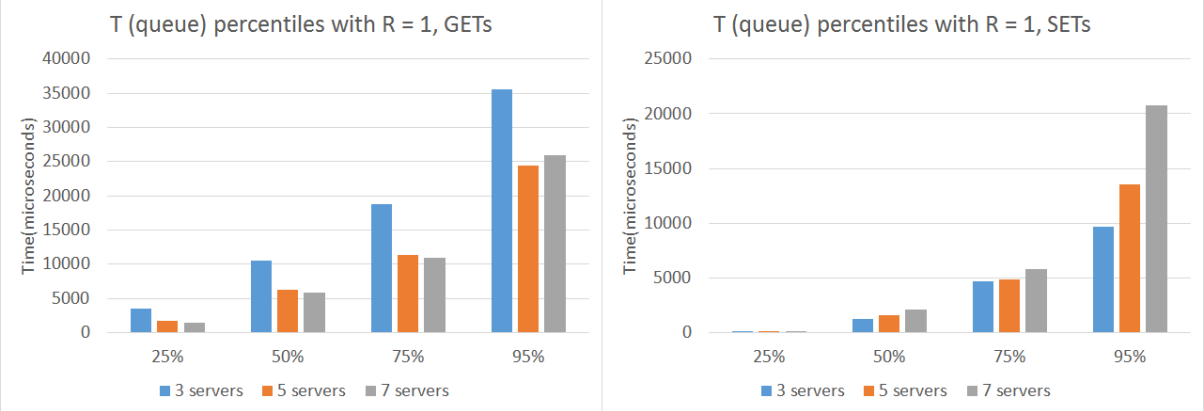




The main expectation was that we should observe a decrease in response time due to decrease in queue time. However, the main factor that significantly increases over the increase in server count is T_{left}. As was stated in

the beginning of report, very minor portion of T_{left} is spent on hashing the request key and putting it into the queue and more than 99% of that time is spent on sending response back to memaslap. T_{left} is same for *get* and *set* requests and it increases drastically when adding more servers. The only possible explanation to this fact is the following: with each new added server we add 16 threads, 16 socketchannels (since each thread uses its own socketchannel) to new memcached server and 1 new queue. While adding queue and additional socketchannels can't affect T_{left} so severely, the only option left is overhead created by added threads. This basically makes sense because sending responses back is done by a single thread in queue in MiddlewareServer.java, and adding more threads to one specific part of the system in general slows down the performance of threads in other parts of the system. Apart from allocating CPU time and memory to new threads, we also have to pay a price of threads context switch. This is main reason why we such a strong increase in T_{left}.

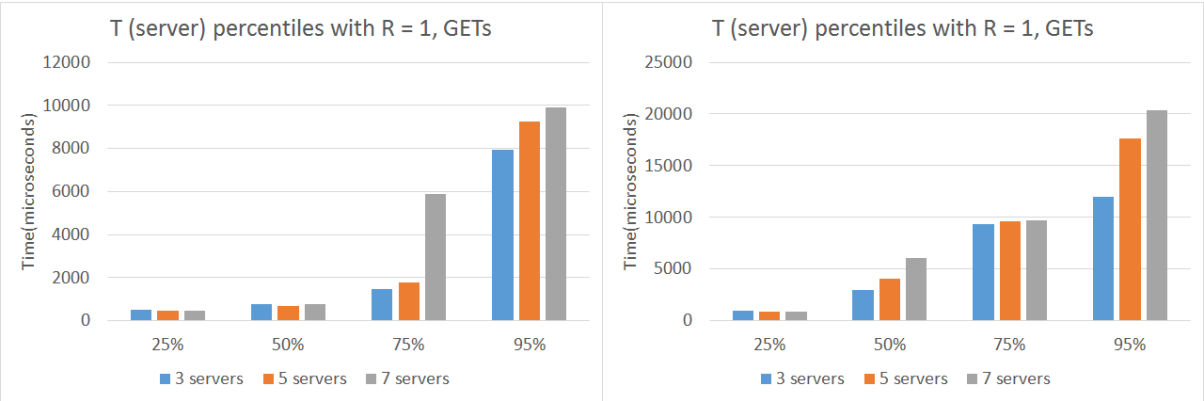
T_{queue} also demonstrates interesting behavior. Following are two plots with 25th, 50th, 75th and 95th percentiles for T_{queue} for *sets* and *gets* correspondingly:



It is seen from plots that for *gets* we do observe significant decrease in time when switching to 5 servers and less significant decrease when switching to 7 servers. This can be explained by the fact that adding new servers decreases T_{queue} only if queue never gets empty. In 3 servers context T_{queue} is significantly bigger than T_{left} (refer to table above), and in our closed system context we receive new requests after response is sent back and it happens faster than the queue gets empty. In 5 servers setup T_{queue} almost reaches T_{left} - so the queue almost gets empty (but not fully) pretty often. In 7 servers setup T_{left} is significantly higher than T_{queue} - system now manages to get the queue empty before receiving new requests, and adding more servers at this points will not increase performance since we are able to empty the queue. One should mention that 95th percentile is higher for 7 servers than 5, but 95th percentile contains outliers which cause such a behavior.

For *sets* things are different: in all cases T_{queue} is lower than T_{left} (refer to table above), so queue in all cases gets emptied. This is due to the fact that we process *sets* in asynchronous fashion, which makes it much faster, and we have only 5% of *sets* - load is much lower than with *gets*. Adding more servers nowhere is capable of decreasing T_{queue}. Moreover, T_{queue} gets more expensive with adding new servers - again, the only possible explanation for that is overhead created by adding more threads, as in case with T_{left}.

Things are different with T_{server}. From the above table one can see an increase in server time with more servers. Different reason could be reason for such a behavior. To account for the appropriate one we should observe the percentile plot for *sets* and *gets* correspondingly:



Percentile plot for *gets* explains the main reason of more expensive T_{server}. It is well seen that 25th, 50th and 75th percentiles are pretty much same for 3 and 5 servers. However we see a huge increase in 75th percentile for 7 servers and there is only one reason for that: the added memcached VM's are located in a different subnet, physically farther from middleware VM. This is a thing which may happen in Azure since we have no control over how and where VM's get initiated. If added 2 VM's are located much further it explains why 50th percentile stays stable - it counts the smallest timestamps and they will come from closest servers to middleware, and 75th percentile already includes VM's located farther and changes the T_{server} drastically. It makes sense since adding more VM's increases probability of working with VM which will have higher network latency between itself and middleware. 95th percentile follows this logic, but may also contain some outliers so no exact assumption can be made based on it.

T_{server} for *sets* requires explanation on how it is logged: it is less accurate than T_{server} for *gets* since here we are dealing with asynchronous setting. In the best case we will get same measurement as in synchronous, but pretty often we will have not as exact metric since we are not waiting for response to be received - at the moment when it is ready we might be doing something different - taking element from queue or writing to memcached. This may artificially increase measured time and compensate cost of more distant server, which is observed on the percentile plot. Despite medians show growth with more servers, 75th percentile shows flat results. Sadly, there exists no more precise method for logging T_{server} in asynchronous mode, but if 95th percentile is to be analyzed, it follows logic about more distant servers. The mentioned difference in architecture of synchronous and asynchronous clients and their logging style is main reason why T_{server} for *sets* is always higher than for *gets*.

Read and write requests are indeed impacted differently in this analysis of changing number of servers. It is seen from main table above that in 3 servers case, *sets* are much faster than *gets*. Adding more servers increases T_{left}, which slowly becomes dominant among other times. Read requests are able to resist that to some extent due to decrease in T_{queue}, while write requests only see increase in T_{queue}. This causes almost same response time for both operations in 5 servers case. When switching to 7 servers case, *sets* become slower than *gets* mainly due to increase in T_{queue} again.

Increase in replication number has different effect on the performance. When discussing changes in times now, I'll be referring to changes while **increasing** replication factor, with number of servers stated in the sentence. From the mean time plots above it is clearly seen, that fixing amount of servers and increasing replication factor has a negative impact on response time. For all number of servers, total response time slightly increases, causing a degrade in TPS. In case of 3 servers, T_{left} increases for both *sets* and *gets*, however the increase is around 0.2ms and is insignificant. For 5 servers, T_{left} decreases when using R = 3, and then increases to initial value. For 7 servers fluctuation is around 0.5ms, which is still insignificant. Since no clear pattern is seen regarding T_{left} and considering that it should not be affected by replications, fluctuations in it can be considered to be random noise for both operation types.

T_{queue} demonstrates interesting behavior as well. The mean table shows no clear pattern for read operation. T_{queue} decreases when increasing rep. factor for 3 servers. In 5 servers, it first decreases and then increases again. In 7 servers setup we might observe an increase in T_{queue}. Considering that replication factor has nothing to do with T_{queue} for *gets* and sporadic changes in it, we may conclude that increasing replication factor has no effect on T_{queue}, apart from some noise happening in the system and not caused by replications. Following percentile table for T_{queue} for read op's proves our assumptions.

T (queue), GETs					
Percentiles		25%	50%	75%	95%
3 servers	get	3433.50	10473.75	18738.25	35560.88
3 servers	get	3566.06	10669.38	18804.94	32754.11
3 servers	get	3507.13	10558.00	18483.94	32452.23
5 servers	get	1658.56	6249.38	11401.81	24418.55
5 servers	get	1685.75	6180.00	11354.13	24338.34
5 servers	get	1755.75	6736.88	11825.94	26335.31
7 servers	get	1497.06	5823.00	10950.25	25988.65
7 servers	get	1500.88	6161.13	11276.19	27268.88
7 servers	get	1597.31	6073.13	11432.13	27527.14

T_{queue} percentiles for read op's

For write operation we may expect an increase in T_{queue}, since more operations are done in between of taking elements from queue with an increase in replication factor. Mean table indeed shows evidence mostly stable increase. Following are percentile plots for T_{queue} for write operations:

T (queue), SETs					
Percentiles		25%	50%	75%	95%
3 servers	R = 1	3.00	1274.13	4699.31	9700.23
3 servers	R = S/2	4.25	979.13	4371.38	9244.78
3 servers	R = all	6.56	1253.25	4792.63	9565.05
5 servers	R = 1	4.50	1540.75	4834.81	13502.80
5 servers	R = S/2	9.00	1716.88	5022.13	14294.98
5 servers	R = all	22.00	1751.25	5201.06	12690.25
7 servers	R = 1	43.25	2103.75	5779.25	20801.98
7 servers	R = S/2	114.25	2318.50	6026.63	22592.15
7 servers	R = all	162.75	2342.38	6036.38	20837.46

T_{queue} percentiles for write op's

It is seen from the plots that for 3 servers, T_{queue} first decreases when increasing R, then increases back close to original value. In 5 servers case, T_{queue} does increase for median and more significantly for 75th percentile. For 7 servers setup median keeps increasing but 75th percentile stays flat when going from R = half to R = all. 95th percentile is not considered since it is assumed to have outliers.

In fact, we do see clear increase in T_{queue} when going from R = 1 to R = all, however results for R = S/2 are somewhat sporadic for S = 3. Probably this could be explained by the fact that going from 1 replication to 2 is not so significant, but going higher becomes more expensive, what is observed in the table. In general though, increase in T_{queue} makes sense since increasing replication factor adds additional operations in between of looking up into queue. These are operations are pretty fast since they are in non-blocking mode, but still, impact should be observed. This means that queue operation is one of op's which become more expensive with increasing R.

T_{server} shows different behavior. Following are percentile plots for T_{server} for read op's:

T (server), GETs					
Percentiles		25%	50%	75%	95%
3 servers	R = 1	500.00	753.00	1481.38	7940.83
3 servers	R = S/2	477.00	715.50	1361.00	7945.00
3 servers	R = all	480.75	728.00	1451.69	8023.14
5 servers	R = 1	445.25	686.38	1770.50	9256.05
5 servers	R = S/2	450.00	701.13	1881.75	9442.89
5 servers	R = all	459.69	718.50	2179.50	9462.21
7 servers	R = 1	453.75	761.50	5892.81	9901.16
7 servers	R = S/2	455.63	790.50	6305.56	9957.59
7 servers	R = all	464.50	818.25	6868.94	9976.73

T_{server} percentiles for read op's

Fluctuations in T_{server} are negligible at all, since most of them is within 0.1 - 0.2ms. Plots above prove our expectation that replication factor has nothing to do with T_{server} for read op's. In fact, we see more than 0.2ms increase for 7 servers setup, however, as stated in server adding effect, 7 servers setup contains several servers located more distant from middleware VM, and it may cause more intense fluctuations. It is proved by the fact that despite out 95th percentile may contain outliers, it stays really stable when increasing rep. factor.

For *sets* we expect stable increase in T_{server} with increasing replications. This increase is observed on the mean times table. To strengthen our assumptions, plots with percentiles are following:

T (server), SETs					
Percentiles		25%	50%	75%	95%
3 servers	R = 1	996.06	2966.25	9358.69	12006.03
3 servers	R = S/2	1185.50	3794.00	9463.50	14277.75
3 servers	R = all	1488.56	7607.38	9738.06	13803.98
5 servers	R = 1	879.13	4039.25	9637.38	17620.56
5 servers	R = S/2	1321.88	7729.63	9865.69	18287.23
5 servers	R = all	1639.31	8125.50	9967.25	18612.44
7 servers	R = 1	871.75	6099.00	9718.50	20399.48
7 servers	R = S/2	1564.44	8657.88	10095.44	25850.93
7 servers	R = all	2057.94	9136.13	10846.94	30232.35

T_{server} percentiles for write op's

Stable and significant increase in T_{server} is observed in any configuration. Such an increase was expected since we start measuring T_{server} when first replication of request to server is sent, and we finalize measurement when last response from memcached is received. Increasing R will definitely lead to increase in T_{server}, which is observed above. T_{server} together with T_{queue} are timestamps which become more expensive for fixed # of servers and increasing replication factor.

Read and write requests are indeed impacted differently in this analysis of changing number of replications. For different number of servers we have different ratio of T_{total} for *gets* to T_{total} for *sets*, as was stated in analysis of adding more servers impact. However, within each configuration of servers, increasing replication factor doesn't affect read operations (apart from some randomly generated noise, as stated above) but it does significantly affect write operations mainly due to stable increase in T_{server} and T_{queue}, and it is the reason why write op's become slower and affect overall degrade in performance, causing less TPS and higher response time.

To conclude this sections, comparison with theoretically ideal system is to be made. From an ideal system one would expect a capability to scale across more servers: there would be no reason to add more servers if it doesn't improve the performance. Load balancing would add up to decrease in overall T_{queue} and T_{server} (in case when memcached itself gets overloaded). These two parameters would add value to overall decrease in T_{total} and thus, increase in TPS. Increase in replication factor would have no effect as well since it may be done in parallel. Unfortunately, my system is not coming along with ideal system in terms of scalability, since there exists a huge bottleneck while handling responses back to virtual clients. Due to this bottleneck it is hard to see increase in T_{queue} after some point and benefits obtained by adding more servers are not seen at all. Instead, adding more servers creates internal overhead for the MW and time spent in the bottleneck explodes, causing degrade in performance. Fortunately, replication seems to be behaving in some sense logical since precossing it in single threaded fashion was one of the requirements for this project.

Total statistics of middleware and memaslap logs is available and can be found in **exp_2_total** logfile.

Lognames convention: memaslap log names follow next scheme: trace_log_*replicationto_repetitionid*

Instrumentation log names follow next scheme: instrum_*numservers_replicationto_repetitionid*.csv

Directory structure is intuitive.

3 Effect of Writes

Hypothesis:

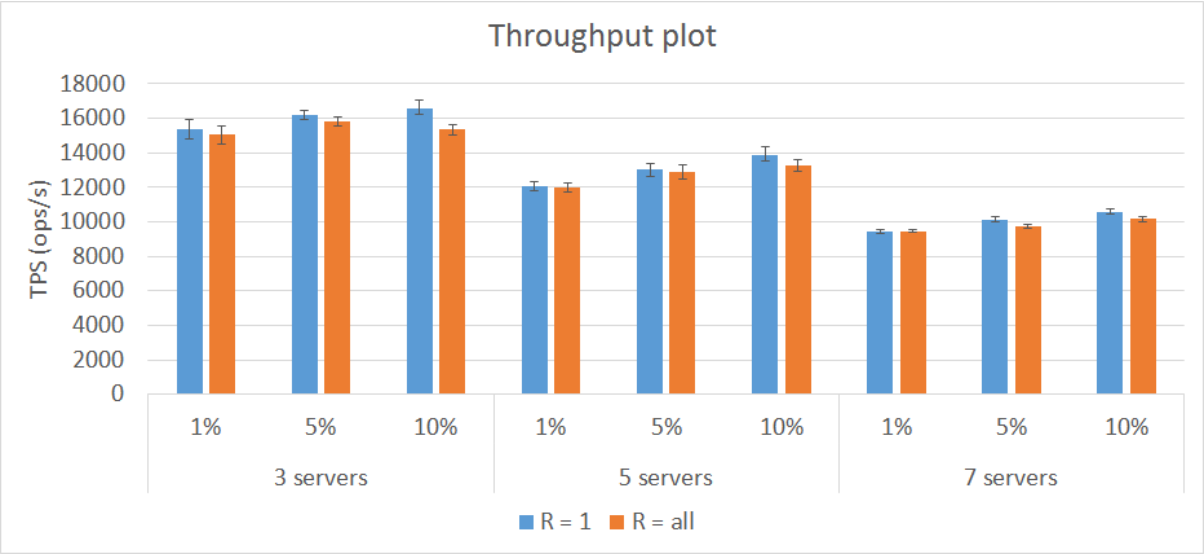
The effect of replications and adding servers will follow up results of 2nd experiment - adding more servers decreases performance, increasing replication factor does same.

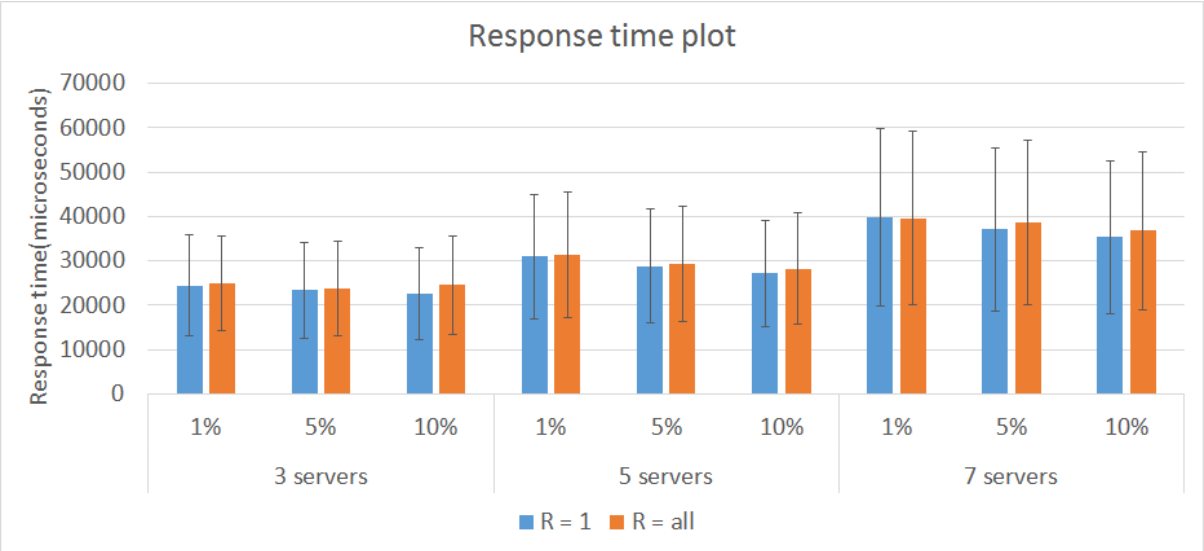
Output of 2nd experiment showed that write op's are faster than read for 3 servers and almost same for 5 and 7 servers. Considering that, I assume that increase in write load will improve performance for 3 servers setup and will have no effect on 5 and 7 servers setup.

Following setup has been used for replication effect experiment:

Memcached VM's	3, 5, 7
Memaslap VM's	3
Replication factor	1, all
Virtual clients / VM	124
Number of threads PTP	16
Workload	Key 16B, Value 128B, Writes: 1%, 5%, 10%
Runtime x repetitions	90s x 4
Warm-up, cool-down period	20s
Sampling ratio	1 of 100 requests
Window size	1k
Memaslap output	once in 5s
Log files	experiment_3, exp_3_total

Explanation on why this setup is used totally follows similar explanation for 1st and 2nd experiment. The following 2 plots demonstrate TPS and response time obtained for this experiment:





Following table provides log. dist. data from memaslap for response times:

		R = 1			R = all		
	Time	1% writes	5% writes	10% writes	1% writes	5% writes	10% writes
3 servers	16384	25.84%	25.83%	27.39%	21.33%	24.51%	24.18%
	32768	78.00%	80.41%	82.03%	78.29%	79.44%	77.95%
5 servers	16384	12.68%	14.68%	16.97%	11.91%	13.80%	15.41%
	32768	60.97%	65.60%	70.96%	59.22%	64.64%	68.29%
7 servers	16384	7.48%	8.65%	9.76%	7.20%	8.27%	8.71%
	32768	41.60%	46.30%	49.55%	41.77%	44.21%	47.51%

Memaslap logdist data

It is very well observed from TPS plot than our hypothesis for write percentage didn't hold. We observe the following: increase in write percentage lifts performance of the system in stable fashion for all combinations of servers / replication factors, and server count / replications by themselves affect performance in same way as in experiment 2. However, 2 points where this statement doesn't hold are observed on the plots: 1st, for 3 servers, performance is lower for 10% writes than for 5%, and it doesn't come along with other outputs. 2nd, for 7 servers and 1% writes replication to all servers provides higher performance than no replication case.

1st one is actually due to a network lag, since my internal logs tell that response time for 10% writes is lower than for 5% - 17792 microseconds for 5% and 17683 microseconds for 10%. Indeed, checking offset from memaslap revealed that for 5% writes offset became 5.9ms against stable 6.7ms for 1% and 10% case.

Performance metrics for 2nd one are in fact really close to each other, so that for TPS, e.g., both of measurements are within standard deviation (which generally is lower than confidence interval) of each other. This allows us to explain 2nd observation as a consequence of variation in the data.

Answering the question regarding biggest impact on performance relative to the base case, I'd like to define base case as a default setting of 3 servers, no replication and 1% of writes. If we take TPS as a measure of performance and assume that base case performance equals to "1", following table shows performances of other different setups:

	1%	5%	10%	
3 servers	1	1.05	1.08	R = 1
	0.98	1.03	1.00	R = all
5 servers	0.79	0.85	0.90	R = 1
	0.78	0.84	0.86	R = all
7 servers	0.61	0.66	0.69	R = 1
	0.62	0.63	0.66	R = all

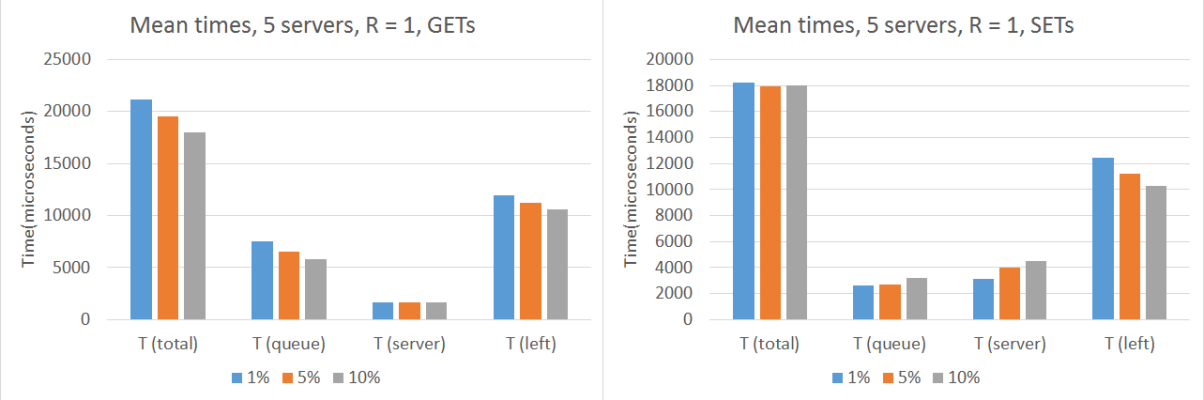
Performance measurement

It is clearly seen from the above table as well as from the plots, that biggest impact is at configuration of 7 servers, 1% writes, no replication. However, considering that R = 1 and R = all are mostly same due to explanations above and due to general trend of 2nd experiment showing that replication causes degrade in performance, theoretically I would choose same configuration with R = all. In contrast, configuration of 3 servers, R = 1 and 10% writes shows best possible performance achieved in this experiment.

According to the above table, it's clear that biggest degrading in performance is due to increase in number of servers, since other two factors do not cause such a severe drop. Increase in write percentage causes no damage to performance and only increases it, but change in absolute value is still bigger than replication factor's impact. Replications cause smallest impact to overall performance.

To analyze the exact behaviour of system we need to check internal logs. However, since this section is focused on the effect of writes, and considering that internal logs' results for changing the number of servers and replication factor do follow exact same trend as in 2nd experiment and are explained in detail there, now we will focus on how write percentage affects behavior of the system. Total statistics of middleware and memaslap logs is still available and can be found in **exp_3_total** logfile.

First we analyze the middle configuration - 5 servers, no replications. Following two plots demonstrate mean timestamps from middleware logs for *gets* and *sets* correspondingly:



It is seen from these plots than read operations become faster, since we see a linear decrease in T_{total}. This decrease is up to 2 main factors: first, we see decrease in T_{queue}, and second, decrease in T_{left}. Decrease in T_{queue} makes total sense, since while increasing percentage of writes, we decrease percentage of reads in the system. Hence, filling of read queue becomes slower and we see increase in T_{queue}. Decrease in T_{left} will be discussed in context of write op's.

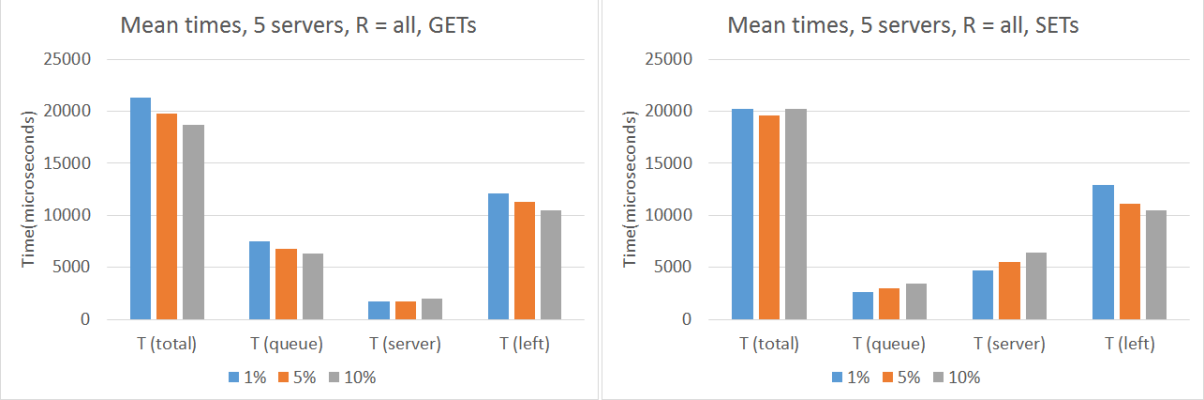
Write operations demonstrate different behavior. We can observe a V-shape in T_{total}, however, T_{queue} and T_{server} become more expensive with increased writes load. Increase in T_{queue} follows the logic of read op's - it becomes higher due to load increase. Since queue gets more requests, AsynClient has more work to do which affects increase in T_{server}, since we have to write to servers more often and thus, postpone reading from servers. This is the main connection between T_{queue} and T_{server}. Following 2 plots with percentiles prove increase in T_{server} and T_{queue} with increasing writing percentage (for *sets*).

Interesting that we do not see linear increase in T_{total} despite we have one in queue and server times - as mentioned above, we see V-shape. This is due to high T_{left} in case of 1% writes. Decrease in T_{left} with increase in write percentage first allows T_{total} to decrease in 5% setup, but later decrease cannot compensate increase in T_{server} and T_{queue}. This is main reason why we see this V-shape for total time for write operations.

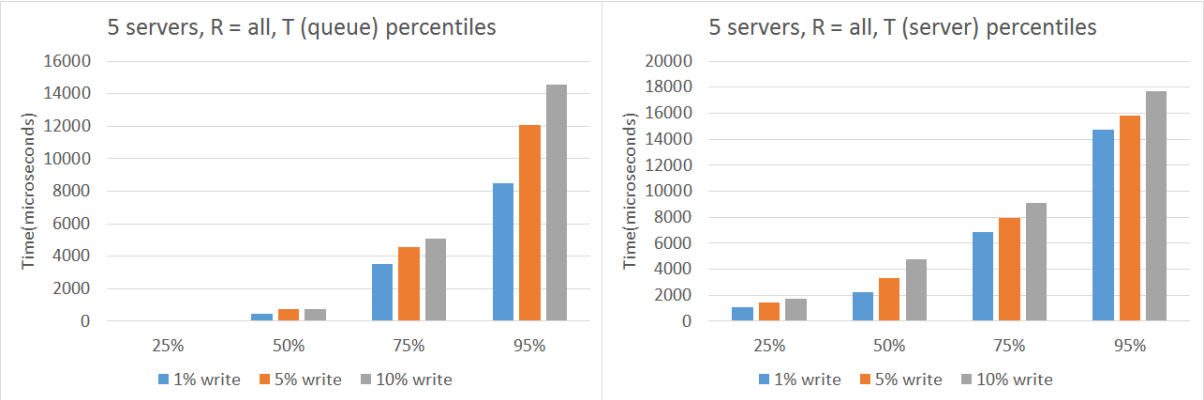
Apparently, T_{left} changes due to change in write percentage. It was discussed earlier that T_{left} is executed in single thread and increases when we add more threads to the system due to multithreading costs. In fact, in context of this experiment we see that threads for *gets* and *sets* cause different impact on T_{left}. This can be explained by fact that *gets* thread work in blocking fashion - they spend significant amount of time being idle and waiting, cause less impact on other threads in system. Increase in their count actually affects the system (slows down), but amount of work in single iteration of run() method for these threads is constant. For *sets* our threads are never blocked, they are spinning even if the queue is empty, since we may still want to read something from memcached. In contrast with *gets* threads, amount of work for these threads is changing when we change writes percentage or replication factor. Logical assumption is that spinning threads cause more impact to system performance than threads that may be blocked. Interesting is that lower loads for these threads cause more overhead for, e.g., T_{left}, due to ineffective usage of resources. When the load for them increases, threads do less "empty" work and overall it has a positive impact on other system components. It is impossible to state exact reason for that since this is totally due to hardware level and JVM routines, but clearly T_{left} is mainly affected by writer threads and decreases when workload for writer threads gets higher.

So, eventually, V-shape in *sets* time in general leaves T_{total} more or less constant. What adds value to increase in performance is decrease in cost for *gets* due to lower load.

Next, to prove that replication strategy behaves as described in experiment 2, I provide middleware mean times plots for 5 servers and full replication:

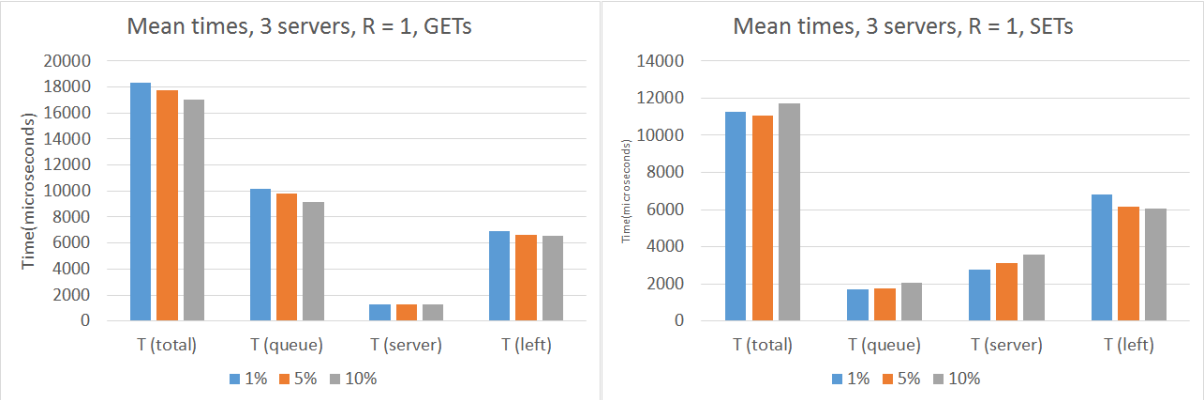


It is clearly seen from these plots that read op's are not affected by replication strategy, and the total trend for write op's exactly follows no replication case, with difference in values for T_{server} and T_{queue}. This was exact case in our second experiment. For full picture I provide percentile plots for T_{server} and T_{queue} (write op's):



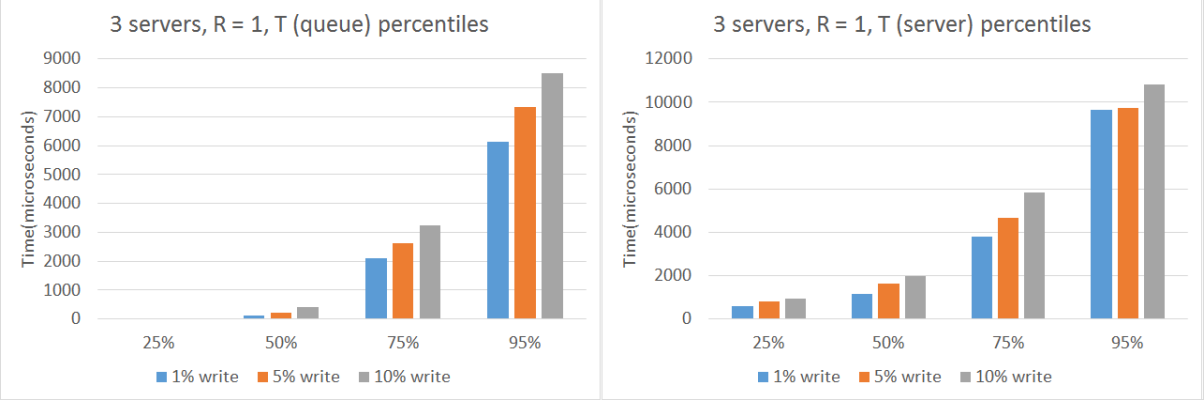
This information proves that replication strategy acts exactly as stated in experiment 2, hence, I will not discuss replication effects further in context of this experiment since it would be redundant.

Next plots shows mean timestamps from middleware logs for 3 servers, R = 1:



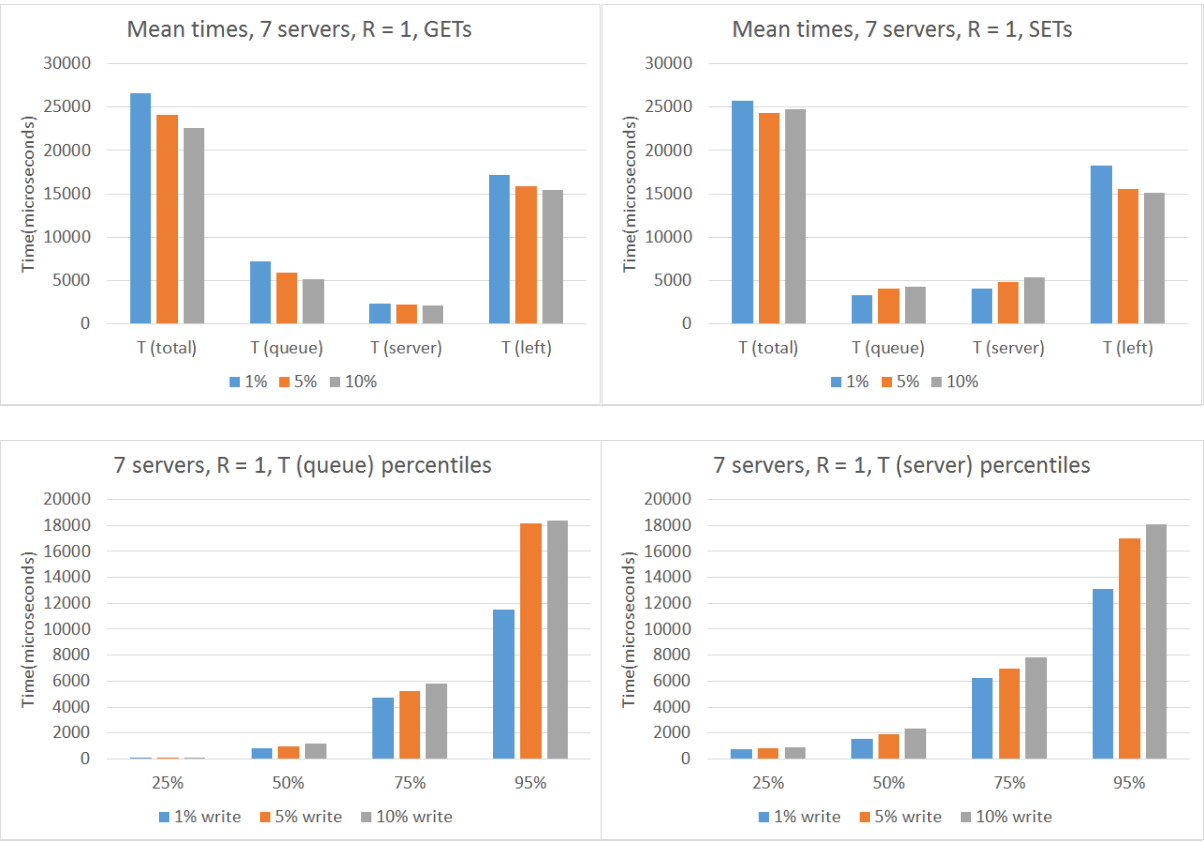
All the discussions above can be applied to these plots. We can see that general trend is saved, just the scale is different. Discussions from experiment 2 regarding change in performance due to change in server number fully applies here: dominant factor in decrease in response time is T_{left}. Other timestamps also follow same trends.

Percentile plots for T_{server} and T_{queue} (write op's):



Nothing can be added to discuss above plots apart from statements made above.

To conclude this experiment, following 4 plots show mean timestamps and percentiles for T_{server} and T_{queue} (write op's) with setup of 7 servers, R = 1:



As before, these plots follow trends mentioned in this section and conclude this experiment.

Logfile **exp_3_total** contains full information with timestamps both from MW and memaslap in table form.

Lognames convention: memaslap log names follow next scheme: trace_log_*replicationto_workload_repetitionid*

Instrumentation log names follow next scheme: instrum_*numservers_replicationto_workload_repetitionid*.csv

Logfile listing

Short name	Location
hash_test	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/milestone_2_logs/hashing_microbench/hashing_microbench.7z
experiment_1	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/milestone_2_logs/experiment_1/experiment_1.7z
exp_1_total	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/milestone_2_logs/experiment_1/exp_1_total.csv
experiment_2	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/milestone_2_logs/experiment_2/experiment_2.7z
exp_2_total	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/milestone_2_logs/experiment_2/exp_2_total.csv
experiment_3	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/milestone_2_logs/experiment_3/experiment_3.7z
exp_3_total	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/milestone_2_logs/experiment_3/exp_3_total.csv