Advanced Systems Lab (Fall'16) – Second Milestone

Name: ***Rishu Agrawal***
Legi number: ***15-945-355***

Grading

Section	Points
1	
2	
3	
Total	

1 Maximum Throughput

1.1 Experiment Setting and Design

Since I had a lot of leftover credits from the month of October that would expire by the beginning of November, I decided to utilise those and ran some experiments that I anticipated would be useful. The idea behind this was to check for the interaction between factors in the Middleware. I ran the following experiments by choosing the parameters based on factorial design:

Number of servers	Number of Virtual clients / machine	Number of client machines	Key, Value size	Middleware	Number of threads in pool	SET/GET ratio	Log files
5	32	5	16B, 128B	R=1	8	0/1	maxtpexp1
5	48	5	16B, 128B	R=1	16	0/1	maxtpexp2
5	48	5	16B, 128B	R=1	32	0/1	maxtpexp3
5	64	5	16B, 128B	R=1	48	0/1	maxtpexp4
5	64	5	16B, 128B	R=1	96	0/1	maxtpexp5
Runtime x repetitions = 720s x 5							

Before running the experiments, despite this being a preemptive run, I made the following hypothesis regarding the behaviour of the throughput:

- For the same number of virtual clients per machine, I expect the throughput to decrease when the number of thread increases due to the overhead introduced by context switching.
- From the first milestone, I expect the throughput to saturate at a point between 200-300 virtual clients, so I expect a lower throughput for 160 (32*5) virtual clients, compared to 240 (48*5) virtual clients.

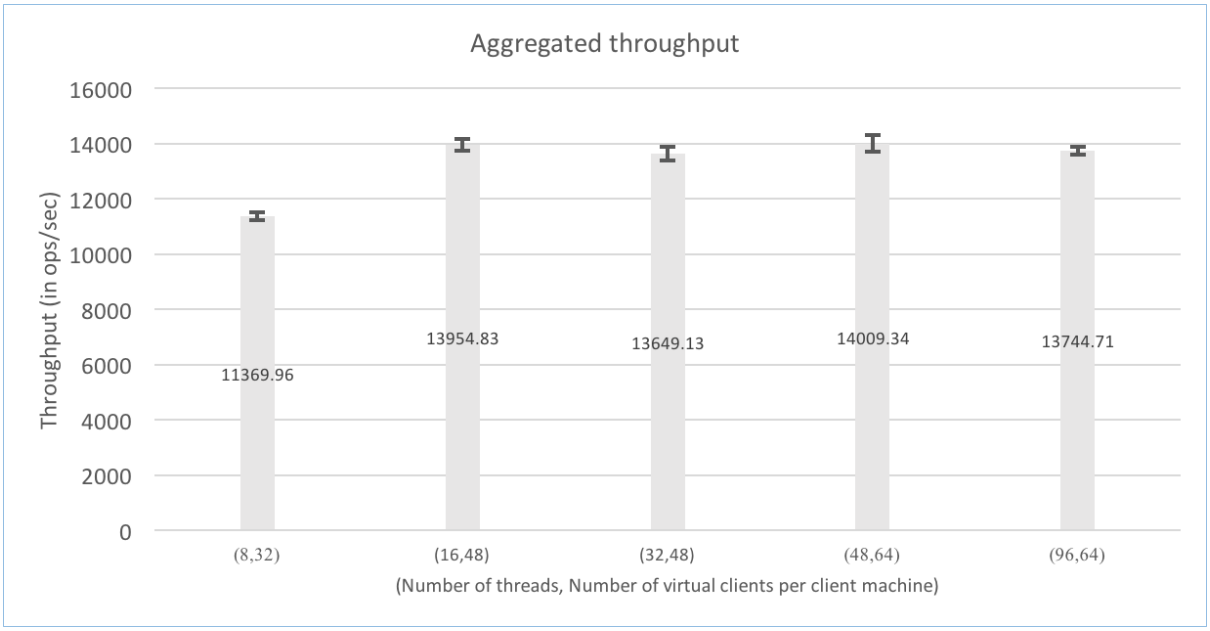


Figure 1: Aggregated Throughput for the 1st batch of experiments

Through this experiment, I made the following observations:

- As we can see from Figure 1, the first hypothesis is valid as between the configurations from 16 to 32 threads and 48 to 96 threads the throughput decreases for a fixed number of virtual clients.
- The second hypothesis can't be validated with certainty from this set of experiments as the number of parameters explored are too few but it does give me some estimation of the ranges to explore.

- Running the experiments for a long time is futile as the throughput reaches a stable point fairly quickly. After inspecting the memaslap output, it seems fair to discard the output of first and last 30 seconds as these constitute the warmup phase and cooldown phase.
- It is a better idea to instead increase the repetitions as it also helps in getting a stronger confidence bound.

Next, I decided to bifurcate the task of finding the minimum configuration that yields maximum throughput into two sections:

- Exploring the effects of varying the number of threads
- Exploring the effects of virtual clients for a thread configuration

For the first section, the main questions to answer are: Which is the minimum thread number that sees a substantial gain in performance? Which is the thread number that sees a consistently high throughput and a low response time?

For the second section, the main questions to answer are: At which range of virtual clients does the throughput saturate? How does it relate to the change in response time and why is this a good choice?

1.2 A note on Azure

There are several problems with the Azure platform that I observed during my brief experience with it for the project. I would like to highlight some of those:

- The Azure platform randomly allocates the public IP addresses for the VMs in the cloud. This often causes serious dips in performance as more often than not, the subnets allocated to the virtual machines are different and it adds a substantial network cost to the system's performance leading to inconsistent results across different times.
- It is extremely hard, if not impossible, to expect consistency in results from Azure even if all the VMs belong to the same subnet. I ran an experiment in the difference of a day for the same configuration and observed a change in throughput of about 2000 despite the VMs belonging to the same subnet. However, this can be explained by the traffic in the cloud as it is likely that the VMs allocated by Azure reside on a single system and the load on that system varies according to the time and day.

Thus, to maintain some form of consistency, I discarded the results from the experiments I performed across different days and ensured that for a section, I ran all the experiments on the same day.

1.3 Exploring the effects of varying the number of threads

The goal of these experiments is to determine the minimum number of threads for which a maximum throughput is obtained consistently. For this section, the following experiments were run with the following configurations:

Number of servers	5
Number of client machines	3
Virtual clients / machine	32, 64, 96, 112, 128, 144
Workload	Key 16B, Value 128B, Writes 0%, Reads 100%
Middleware	No Replication (R=1)
Number of threads in pool	2, 4, 8, 16, 24, 32
Runtime x repetitions	180s x 5
Log files	threadexp1...threadexp36

The first 30 seconds and the last 30 seconds of the measurements are discarded and not accounted for in the computation of aggregated throughput and average response time from the memaslap logs from 3 client machines and middleware instrumentation logs. However, in the case of plotting the cumulative distribution function from memaslap data, since the log-2 distribution is obtained only in the end and it accounts for all the observations in the log, I had no other option than to include the measurements from the first and last 30 seconds.

Since, each experiment is run for 5 iterations, I average the aggregated throughput and response time across these 5 iterations. In the case of the middleware instrumentation logs, since the logs are sampled at a rate of 1 every 100 requests, I aggregate them instead of averaging them in order to reduce the sampling factor by 5 and getting a more robust representation of data.

1.3.1 Hypothesis

Since the Middleware runs on a basic A4 VM on Azure, it has 8 cores to run on. I expect the throughput to increase with the number of threads up to a certain point but then expect to see some degradation in performance due to the overheads caused by context switching in threads. For the response time, I expect the response time to decrease with the increase in the number of threads upto a certain point after which the overheads caused by the switching of threads would increase the response time instead.

1.3.2 Throughput

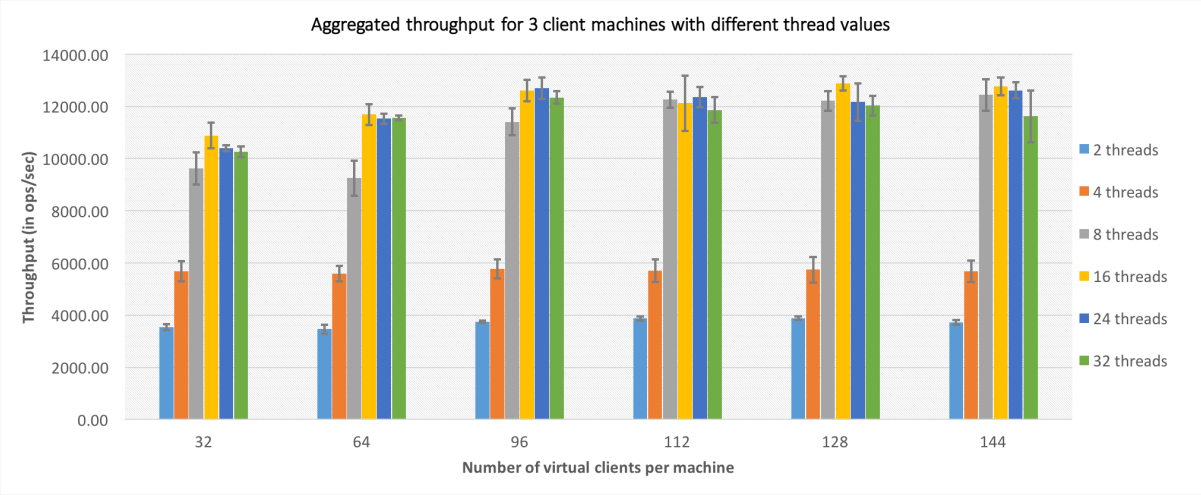


Figure 2: Aggregated Throughput for different number of clients across various number of thread values in the threadpool

As seen from **Figure 2**, the throughput is minimum across the various client configurations for 2 threads. As I increase the number of threads to 4, the throughput increases by 50-60%. This increase is even more significant when the number of threads are doubled from 4 to 8 as the aggregated throughput doubled as well. Things get interesting hereafter as the gains obtained by doubling the number of threads in the threadpool from 8 to 16 aren't as significant as earlier and vary by upto 20%. After 16 threads, however, I do not see gains in throughput and the throughput between 16 and 24 is almost the same, besides some points where the system with 16 threads performs better than 24 threads. When the number of threads is increased to 32, the throughput actually decreases a little (by upto 10% for an increase in the number of clients). This is in accordance with the hypothesis that I make earlier as for a fixed number of cores, having fewer threads would not fully utilize the CPU resource, and having more threads would cause threads fighting over the CPU resource. Threads also introduce overheads such as context switching and after a certain point, the benefits of having more threads for reads are undone by these overhead costs, thus, bringing down the throughput. Since we need a minimum configuration that achieves the maximum throughput, using **Figure 2**, the thread number that sees a consistently high throughput with substantial gains would be 16.

1.3.3 Response time

Since the CDF plots for 2 threads is similar to 4 and 16 is similar to 24, I omit these plots in order to avoid redundancy. As we can see from **Figure 3**, the CDF plot for 4 threads looks different than the CDF plots for other threads. Between a range 16384 to 32768 microseconds, all the configurations here complete 80% of the total requests. At first glance, this may seem contrary to the hypothesis made earlier as at this time bucket, all the other thread configurations complete a lesser percentage of requests. However, this notion would not be correct. I inspected the memaslap logs and the reason for this disparity is the fact that for the system with 4 threads, the configuration with different number of clients only completes about 1 millions requests in total. However, when the number of threads increases from 8 onwards, for more than 96 virtual clients per machine, the total number of requests completed

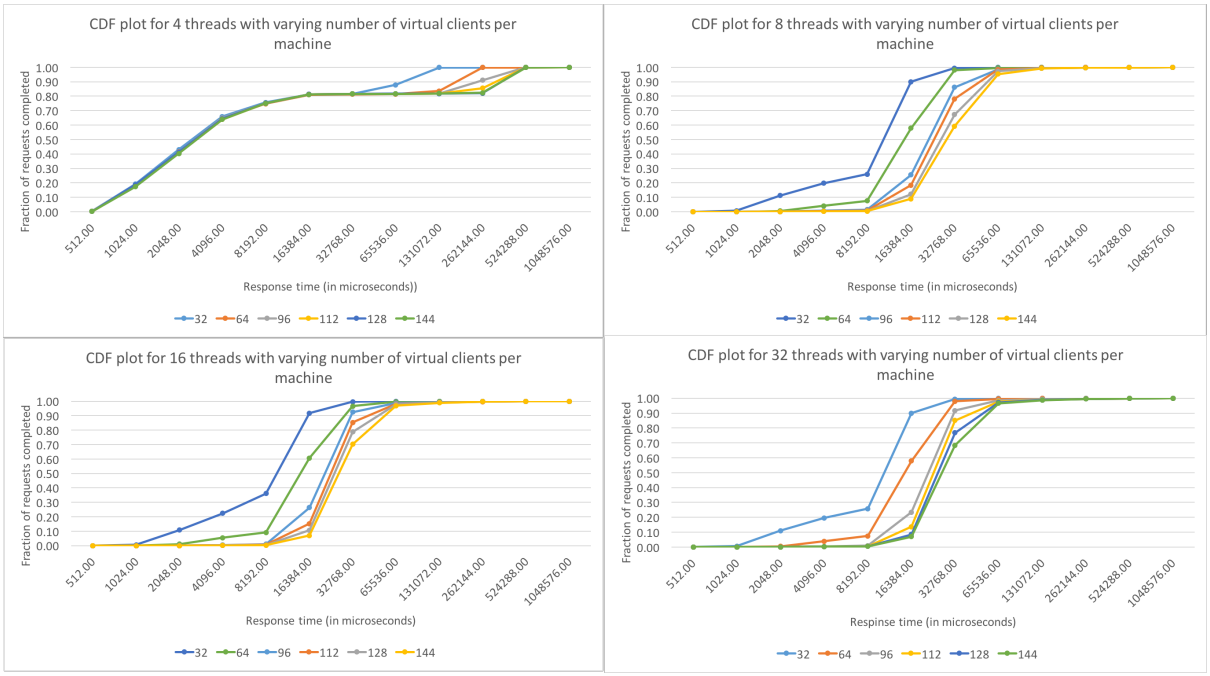


Figure 3: CDF plots for different number of clients across various number of thread values in the threadpool

increases almost by more than 2 folds. Therefore, for systems with higher number of threads in the threadpool, even a lower percentage can constitute a larger number of requests being satisfied.

Another point that I thought was noteworthy was the fact that for 4 threads, the last 20% of data takes more than 131072 microseconds whereas this is not the case for systems with a higher thread count in the threadpool. This is largely due to the fact that these requests spend a lot of time in the queue. From the graph in Figure 3, I also get an estimate that the number of clients at which the throughput saturates is somewhere between 96 to 112. This can be corroborated by the fact that the CDF plot for these points are very similar and almost overlapping. At these configurations, the system also has almost an equal number of total requests (approx. 2.2-2.4 million). As seen from the throughput plot, the configurations with 32 and 64 virtual clients aren't desirable due to their low throughput values and similar behaviour is observed even in this case. The total number of requests completed for these configurations is about 1.2-1.4 million.

Now, when we look at the differences between the plots for 8, 16 and 32 threads, and 96 and 112 clients, we can see that a higher percentage of requests is satisfied between 16384 and 32768 microseconds for 16 threads. I am compelled to look at this value for my justification because the plot is actually an approximation based on the log-2 distribution from memaslap logs. We don't exactly know what the median value is. Hence, this is the most apropos measurement to make comparisons. Thus, keeping this in mind, I choose 16 threads as the minimum number of threads for my middleware to obtain maximum throughput.

1.4 Exploring the effects of virtual clients for a thread configuration

The goal of these experiments is to determine the minimum number of virtual clients per machine for which a maximum throughput is obtained consistently. For this section, the following experiments were run with the following configurations:

Number of servers	5
Number of client machines	3
Virtual clients / machine	32, 64, 80, 96, 112, 128, 144, 192
Workload	Key 16B, Value 128B, Writes 0%, Reads 100%
Middleware	No Replication (R=1)
Number of threads in pool	16
Runtime x repetitions	300s x 10
Log files	vclientexp1...vclientexp8

The first 30 seconds and the last 30 seconds of the measurements are discarded and not accounted for in the computation of aggregated throughput and average response time from the memaslap logs from 3 client machines and middleware instrumentation logs. However, in the case of plotting the cumulative distribution function from memaslap data, since the log-2 distribution is obtained only in the end and it accounts for all the observations in the log, I had no other option than to include the measurements from the first and last 30 seconds.

In these experiments, I chose to run 10 iterations in order to obtain a minimum variation and a better estimate of average by getting a stronger confidence bound. Since, each experiment is run for 10 iterations, I average the aggregated throughput and response time across these 10 iterations. In the case of the middleware instrumentation logs, since the logs are sampled at a rate of 1 every 100 requests, I aggregate them instead of averaging them in order to reduce the sampling factor by 10 and getting a more robust representation of data.

1.4.1 Hypothesis

From the previous experiment, I have a rough estimation that my throughput should saturate somewhere between 96 to 112 virtual clients per machine. So, I expect the throughput to sharply increase before this configuration and then remain constant around this range. After a peak point, I expect to see the throughput dipping as the large number of clients will increase the waiting time and affect the overall response time, thus causing the throughput to decrease.

1.4.2 Throughput

	Number of Virtual Clients per machine							
	32	**64**	**80**	**96**	**112**	**128**	**144**	**192**
Margin of error with 95% confidence (Ops/sec)	146	244	87	78	353	193	380	185
5% of the mean (Ops/Sec)	608	626	670	709	696	729	714	701

Based on the estimated standard deviation, I calculate a 95% confidence interval for each virtual client configuration and compare it to the 5% interval around the associated mean. The table below compares the margin of error with 95% confidence to the value of 5% of the mean, for each virtual client configuration. It can be seen from the table above that the margin of error with 95% confidence is always smaller than 5% of the mean. This indicates that I can be at least 95% confident that the throughput (Ops/Second) values I observe will lie within the 5% margin around the mean.

The throughput for the given set of experiments is visualised in **Figure 4**. Looking at the plot, the maximum throughput occurs for 128 virtual clients per machine and is approximately 14,600 ops/sec. Starting from 32 virtual clients, the throughput first increases at a swift rate from 12,000 ops/sec to 14,000 ops/sec for 96 virtual clients. It then dips a little at 112 clients but has a high standard deviation at the point, so it is safe to assume that this was probably due to a bad measurement in some of the repetition and the throughput essentially saturates in this region. It peaks for 128 virtual clients and after this

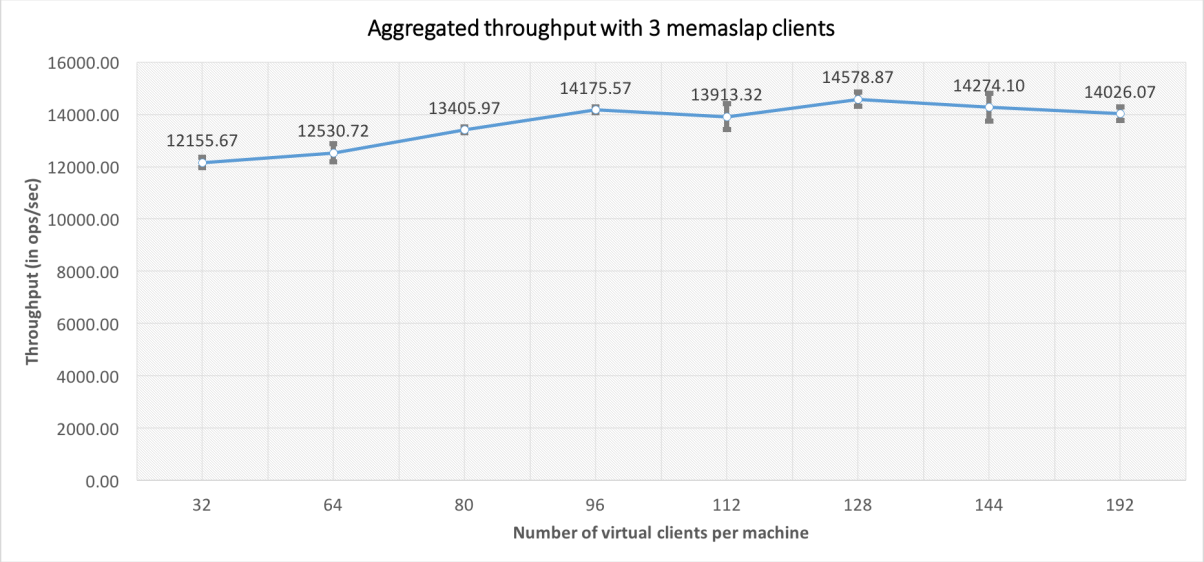


Figure 4: Aggregated Throughput for different number of clients across 16 threads in the threadpool

point, for 144 and 192 virtual clients, the throughput seems to decrease gradually. This is in line with the hypothesis made above as for lesser number of clients, the system does not reach it's optimum performance and is under-loaded leading to an underperforming system. At the saturation phase, the system works at it's maximum efficiency and as the number of clients are increased, the throughout increases gradually. However, after this point, increasing the number of virtual clients overloads the system, and it starts to have performance issues. Since the scope of our experiment is to find the minimum configuration for which the maximum throughput is attained, by just looking at the throughput, I am inclined to pick the configuration with 96 virtual clients as my maximum.

1.4.3 Response time

For this experiment, I also wanted to look at the plots for the average response times and identify the problems associated with using it. I plot the average response time across the different client configurations in **Figure 5**. In a first glance itself, one problem is glaringly clear- the presence of large standard deviations. Based on this alone, it is pretty evident that the average isn't a good metric to use as a reference for making conclusions about the behaviour of the system. But looking closer, the average response time does validate the conclusions made by me after looking at the throughput. The range of saturation lies between 96 to 128 and the points before and after this range are either underloaded or overloaded. I make this deduction based on the steep change in slope after 128 virtual clients. In the saturated phase, the slope change is fairly mild, whereas for the underloaded phase, the slope is fairly horizontal.

Taking this into account, I also plot the CDF plot for the different client configurations as seen in **Figure 6**. Since the CDF plots based on memaslap logs is an approximation as it is based on the log2 distribution and does not tell us the exact distribution of the response time, we are interested in looking at the behaviour of the system within a range. Looking at **Figure 6**, it is reasonably clear that the range to be inspected is between 16384 to 32786 microseconds. Since I already established that 32 is under-performing, the CDF plot is a good tool to verify that claim. The CDF plot for 32 virtual clients is very far from the plots for other configuration and has a low response time for more than 95% of the requests. Things aren't that obvious for 64 virtual clients. At closer inspection, between a range of 16384 to 32768 microseconds, I see that the plots for 64, 80 and 96 clients almost cluster together and at 32768 microseconds, all these configurations have completed greater than 95% of the the total requests.

The configurations with 112, 128 and 144 virtual clients cluster together and complete about 75-85% of the total requests at this time. The configuration with 192 clients, meanwhile, only completes 40% of the total requests. As explained in the previous section, the percentage itself isn't a very accurate metric of comparison as it is possible that this configuration satisfies a much larger number of requests

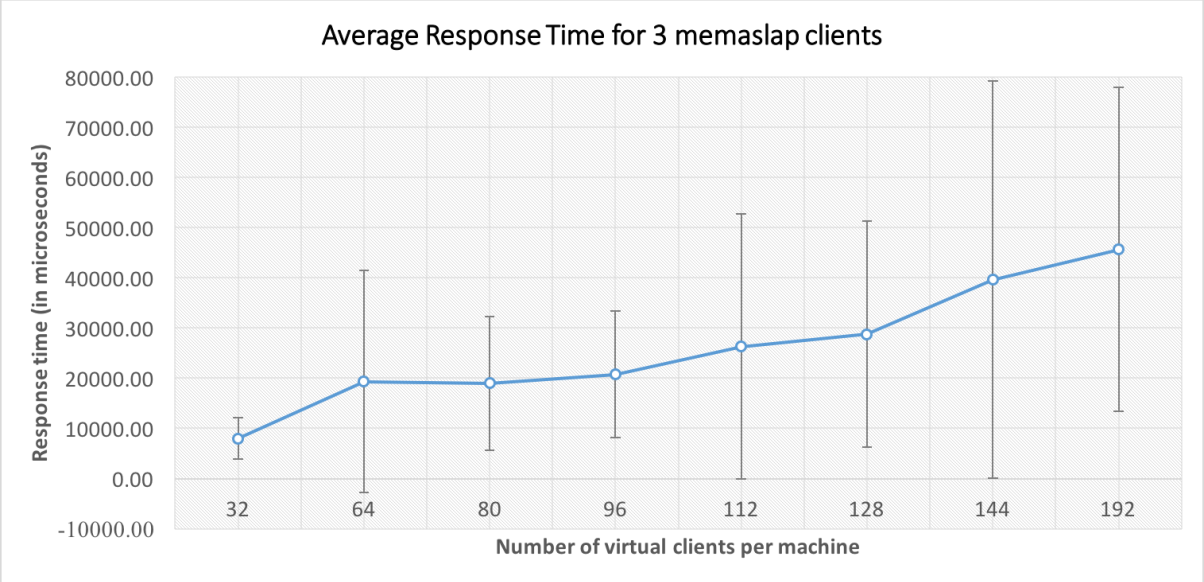


Figure 5: Average response time for different number of clients across 16 threads in the thread-pool

in total despite having a lower percentage. However, after looking at the total requests satisfied during this period for different configurations, I come to the conclusion that the saturation occurs for either 96 or 112 virtual clients which also fits the hypothesis stated above.

But since the goal is to choose the minimum possible configuration, I choose 96 virtual clients per machine as the configuration that yields the maximum throughput for my middleware. In conclusion, my middleware reaches the maximum throughput of **14,175 ops/sec** for **290 virtual clients**(96*3, rounded to the nearest 10) and 16 threads in the threadpool. I also looked at performance variations around this configuration and explained the behaviour for these configurations in the sections above.

1.5 Breakdown of time spent in middleware for the configuration with maximum throughput

Following suit from the previous sections, I come to the conclusion that the maximum throughput for my system occurs at a configuration of **16 threads** and **290 virtual clients** (96 virtual clients per machine). The following table highlights the breakdown of the time spent in the middleware for the read (GET) operation type. All measurements are in microseconds.

Read operations			
Percentile	T_{queue}	T_{server}	T_{mw}
25	14	5	27
50	53	6	71
75	695	9	725
95	7991	17	8028

We can see that the most expensive operation is the T_{queue} for READ operations.

2 Effect of Replication

2.1 Experiment Setting and Design

For this section, I run the experiments with the specifications mentioned in the following table. From the previous section, I know that the maximum throughput is at 96 virtual clients per machine for 16 threads for 5 servers with 100% reads and no replication, so, I run at about 80% of that number, i.e. 80 virtual clients per machine, to account for unforeseen consequences caused by the change in the number of servers. As before, I aggregate the outputs from the memaslap clients and disregard the first and

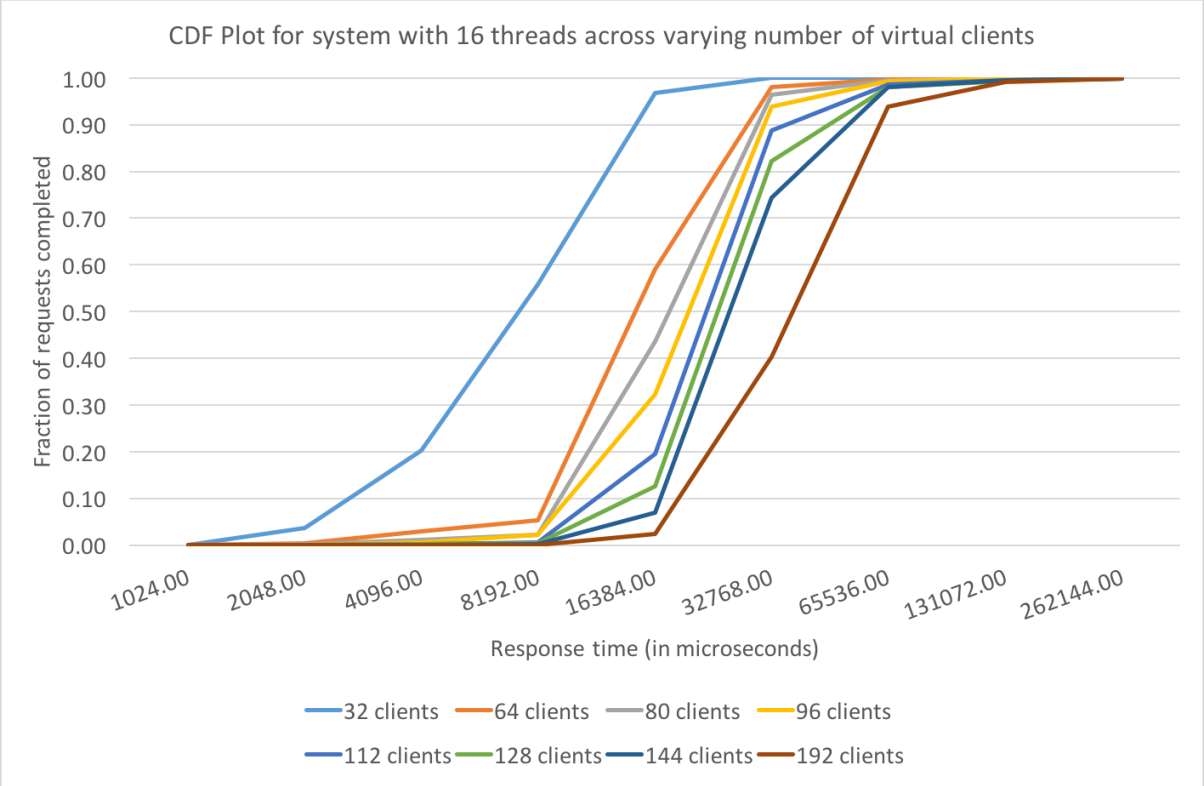


Figure 6: CDF plot of system with 16 threads for 3 memaslap clients and 5 memcached servers with no replication

last 30 seconds of data to account for warmup and cooldown phases. Since, I run the experiments for 5 iterations, I average the results for the memaslap data and aggregate it for the middleware logs.

Number of servers	3,5,7
Number of client machines	3
Virtual clients / machine	80
Workload	Key 16B, Value 128B, Writes 5%, Reads 95%
Middleware(R=)	No Replication, Half-Replication and Full-Replication
Number of threads in pool	16
Runtime x repetitions	180s x 5
Log files	repexp

2.2 Hypothesis

As per the reference document provided, I attempt to answer all the questions asked for the System Under Test (SUT). I would like to start first by addressing the change in behaviour due to the addition of servers. Since we are adding more servers, based on the assumption that my consistent hashing algorithm provides a uniform distribution of load across servers, I expect the throughput to increase upto a certain point. This is largely due to the fact that we have 95% read requests which are synchronous. And as mentioned in Milestone 1, for read requests, the most expensive operation is T_{queue} and because of the introduction of more servers, the T_{queue} decreases, thereby decreasing the response time for read requests. Moreover, I also don't expect the T_{queue} to change because of replication. The replication should not have an effect. Similarly, I don't expect to see any changes for the T_{server} for GET requests across the various configurations as it is independent of replication or the addition of servers.

In case of write or SET requests, I don't expect to see a change in T_{queue} due to addition of servers or changes in replication factors are the writes are asynchronous. Since the most expensive operation for SET requests is T_{server}, when the number of server increases, I expect the T_{server} to decrease a little for the same replication factor as there are lesser requests per memcached server and the number of responses

received at a particular instance will also be lesser. However, as the replication factor increases, for a fixed server, I expect the T_{server} to increase, with R = no replication being the minimum and R = full replication being the maximum.

In an ideal system, I would believe that the increase of servers would lead to an increase in throughput and decrease in response time as more servers would ensure a better load distribution and higher performance. Moreover, for a constant number of servers, I would expect the throughput to decrease and the response time to increase as the replication factor increases.

2.3 Throughput

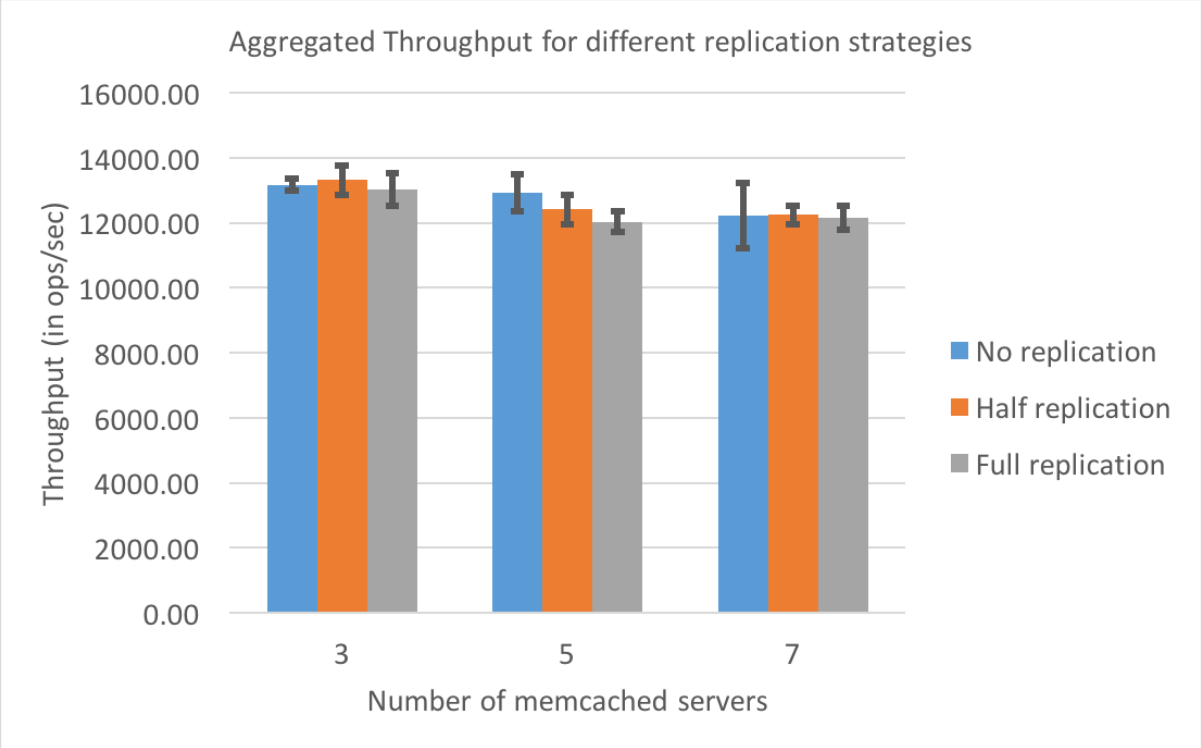


Figure 7: Aggregated throughput for the different configurations with varying number of servers and replication factors across 3 memaslap clients

As seen from **Figure 6**, the plot for the aggregated throughput is quite on the contrary to my hypothesis. For most part, the throughput seems to be almost similar across all configurations. If anything, it actually decreases a little with an increase in the number of servers as seen from the plot between 3 and 7. For 3 servers, the effect of replication is very negligible as the throughput is the nearly the same throughout. For 5 and 7 servers though, there is a small decrease as the replication factor increases. This may partly also be because of the fact that the % of writes is only 5. I suspect this decrease in throughput is due to the limitation in availability of CPU resources in the middleware because as the number of server increases, we also increase the number of threads in the system. When going from 3 to 5 to 7 servers, the number of active threads increases from 52 to 86 to 120.

2.4 Response time

For this section, I plot the CDF plots separately for the SET and GET requests to get a better idea of how each request behaves with respect to the change in configurations in the system. Although the CDF plot obtained from memaslap logs is an approximation of the the distribution of response times since we do not know the actual distribution of response times between crucial time ranges such as 16384 to 32768, it is informative enough to give me an insight of the behaviour of the response time.

First, looking at the CDF plots for the SET requests as shown in **Figure 8**, the response time does not behave as expected as the median of the response time increases with the increase in servers for a

fixed replication factor. However, for a fixed number of servers, the median of response time does increase with the increase in replication factors, which fits well with the hypothesis. For a higher percentile such as 90, the difference in the response time for different configurations becomes very little.

Next, looking at the CDF plots for the GET requests as shown in **Figure 9**, the differences become very arduous to point out and the plots for different configurations almost seem to overlap over each other. However, the behaviour of the response time is in alignment with the results obtained from the throughput.

To examine the behaviour more meticulously and to validate the divergence of the actual results from the hypothesis, I decided to analyse the results from the Middleware logs. This would also help in observing how the SET and GET requests are impacted by different setups and which operations become more expensive corresponding to these changes.

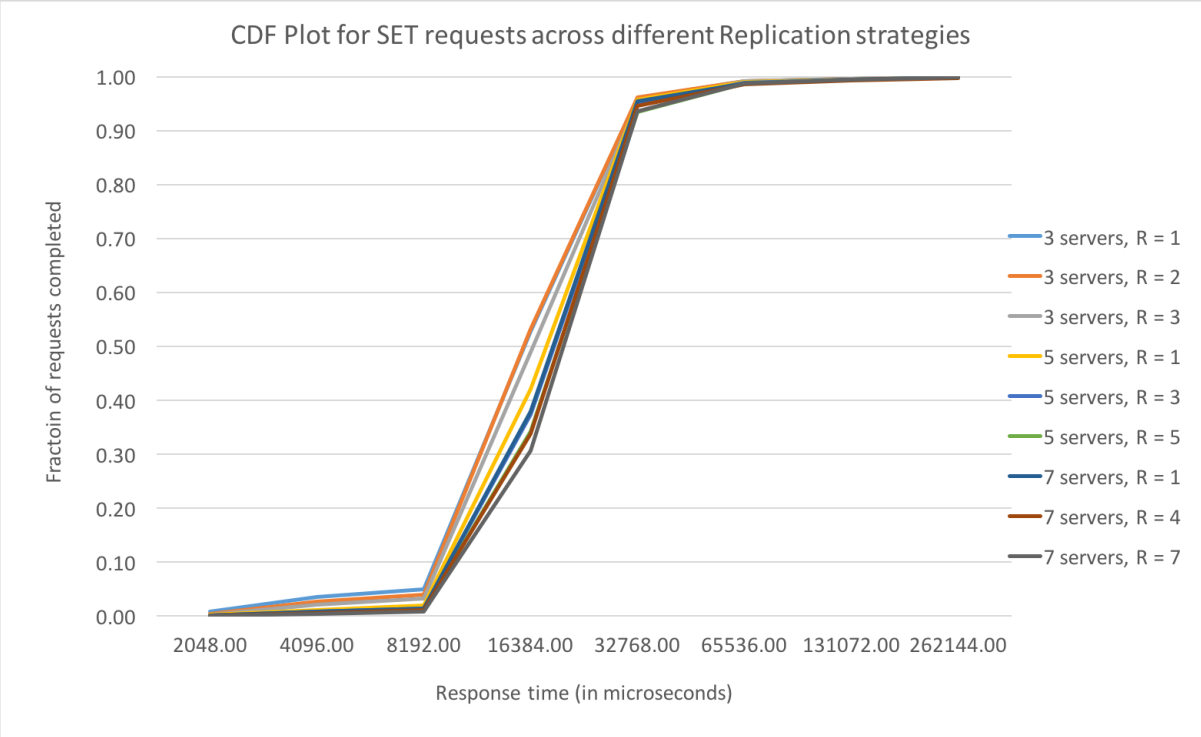


Figure 8: CDF plot for SET requests across the different configurations

2.4.1 Analysis of Middleware Logs

First, I start with the analysis of GET requests. The results here are in accordance with the hypothesis made earlier. As T_{queue} is the most expensive operation in the middleware for GET requests, we can see from **Figure 10** that the T_{queue} values decrease significantly with the increase in servers. They are also unaffected by the change in replication factors as the T_queue values are largely similar for no replication, half replication and full replication in the 25th, 50th and 75th percentile for a fixed number of memcached servers. The results for the 95th percentile differ a little across replication factors but even there, the difference doesn't exhibit a contradictory pattern.

In the case of T_{server} cost for SET requests, as seen from **Figure 11**, the cost is fairly minimal (less than 20 microseconds for 95th percentile) but the interesting thing to note is the fact that the graphs look glaringly similar in all the configurations, thus, validating the earlier hypothesis that T_{server} is not impacted by replication or the increase in servers.

The T_{mw} plot is expectedly similar to the T_{queue} plot as it is the most dominant factor in the middleware costs for GET requests.

Next, I analyse the SET requests. As may be, the results here are also in accordance with the hypothesis made earlier. Since the writes are asynchronous, the requests do not spend a lot of time in the queue and the T_{queue} is not expensive. As seen in **Figure 12** The values for the 25th percentile, 50th percentile

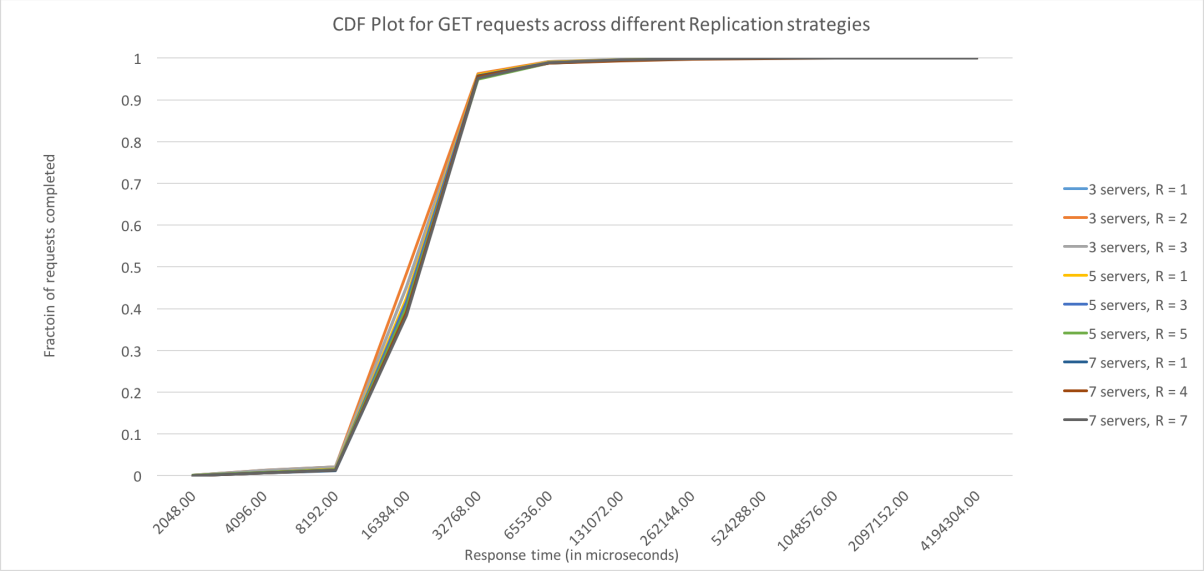


Figure 9: CDF plot for GET requests across the different configurations

and the 75th percentile are nearly identical in all the configurations. However, the results for the 95th percentile do differ. As the number of servers increase, they decrease slightly. However, they do not exhibit a fixed pattern across change in replication factors as the T_{queue} is the least for half-replication in 3 and 5 servers and is almost equal for no replication and full replication, whereas it displays a somewhat linear behaviour as the replication factor increases for 7 memcached servers. I would say this anomaly is incidental and not a problem of the middleware.

The most expensive operation in the middleware for SET requests is the T_{server}. As we can see from **Figure 13**, the T_{server} does follow the initial hypothesis that as replication factor increases, the number of requests to be sent and analysed also increase. In case of a full replication, the asynchronous worker thread has to wait until it has parsed all the responses from the replicated servers and only then can it send back a response. This effect is very dominant in the case of 3 servers as for the 75th percentile, the response time increases by a factor of 3 between half and full replication. Although this could also possibly be because of the fact that since there are only 3 servers, the load per server is high which contributes to additional processing cost as each worker thread has to satisfy more requests and parse more responses. Another thing to note is the difference in the absolute values when going from the 75th to the 95th percentile. The differences in T_{server} reduces to a significantly low factor across various replications at this point.

An interesting thing to note though, is the fact that the response time for 3 servers is consistently higher than 5 and 7 servers. This does imply that adding more servers has indeed reduced the response time from the middleware. However, these gains are offset by the less % of writes and the limitation in availability of CPU resources in the middleware machine, causing the anomaly that is seen in the throughput and memaslap response plots.

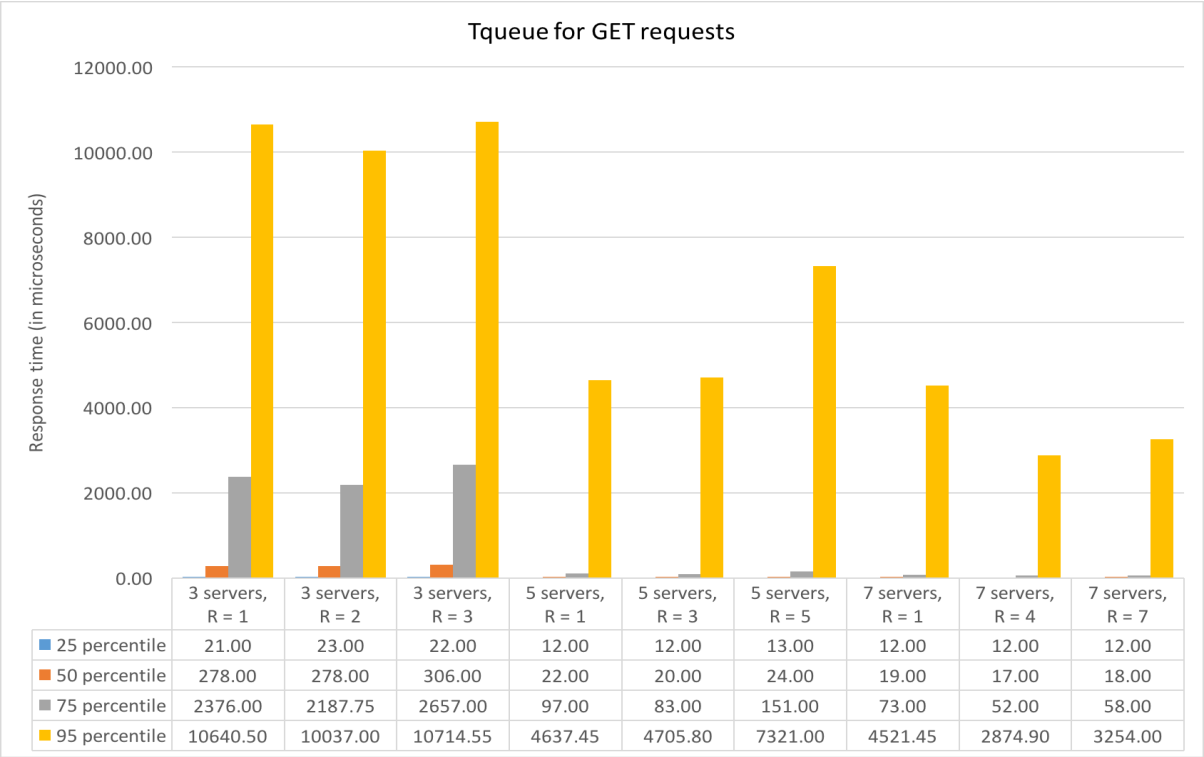
The T_{mw} plot is expectedly similar to the T_{server} plot as it is the most dominant factor in the middleware costs for SET requests.

Hence, as seen from the plots above, the SET and GET requests are impacted differently in terms of the costs of the main operations. The SET requests are mainly affected by the replication factors and the GET requests are mainly affected by the increase in servers.

2.5 Scalability of System

I assume that, in an ideal implementation of my Middleware, different threads dont compete for the CPU time. Therefore, adding more resources, such as increasing the number of Memcached servers, which consequently also increases the total number of Read and Write threads, would result in the increase in the performance (which translates to an increase in the throughput and decrease in the response time).

However, as seen from the experiments above, despite the middleware logs following the expected



	3 servers, R = 1	3 servers, R = 2	3 servers, R = 3	5 servers, R = 1	5 servers, R = 3	5 servers, R = 5	7 servers, R = 1	7 servers, R = 4	7 servers, R = 7
25 percentile	21.00	23.00	22.00	12.00	12.00	13.00	12.00	12.00	12.00
50 percentile	278.00	278.00	306.00	22.00	20.00	24.00	19.00	17.00	18.00
75 percentile	2376.00	2187.75	2657.00	97.00	83.00	151.00	73.00	52.00	58.00
95 percentile	10640.50	10037.00	10714.55	4637.45	4705.80	7321.00	4521.45	2874.90	3254.00

Figure 10: Time spent in the queue for GET requests for replication experiments

trend to a major extent, the middleware does not follow this trend as different threads do compete for the CPU resources and adding more resources consequently degrades the performance instead of improving it. I try to quantify this using a speed-up table where I define speed-up as the ratio of the average response time for 3 servers (for different replication factors) and the average response time of N servers (for different replication factors).

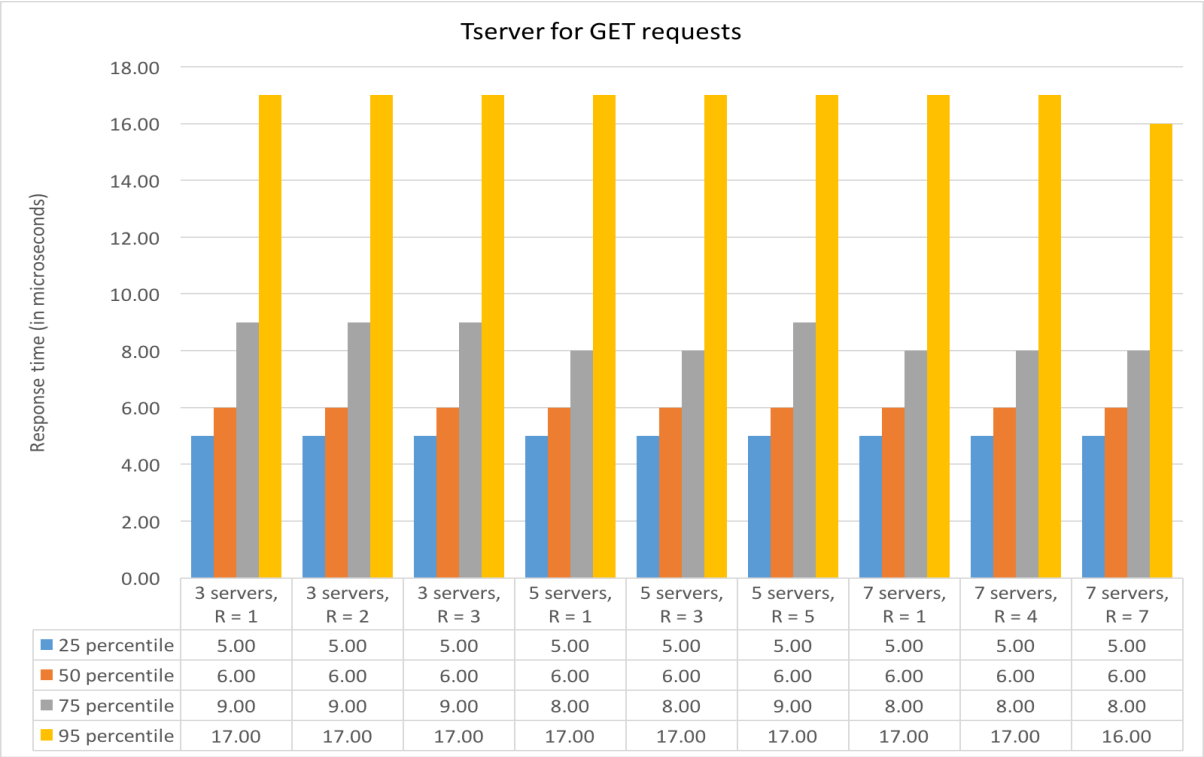
	3 servers	5 servers	7 servers
No replication	1	0.93	0.85
Half replication	1	0.82	0.87
Full replication	1	0.79	0.79

In an ideal system, I would have expected the speed-up to be greater than 1. But in my case, the speed-ups constantly deteriorate with increase in resources. The effect is worse with increase in replication factor.

3 Effect of Writes

3.1 Experiment Setting and Design

For this section, I run the experiments with the specifications mentioned in the following table. I use 80 virtual clients per machine and like before, I aggregate the outputs from the memaslap clients and disregard the first and last 30 seconds of data to account for warmup and cooldown phases. Since, I run the experiments for 5 iterations, I average the results for the memaslap data and aggregate it for the middleware logs.



	3 servers, R = 1	3 servers, R = 2	3 servers, R = 3	5 servers, R = 1	5 servers, R = 3	5 servers, R = 5	7 servers, R = 1	7 servers, R = 4	7 servers, R = 7
25 percentile	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00
50 percentile	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00
75 percentile	9.00	9.00	9.00	8.00	8.00	9.00	8.00	8.00	8.00
95 percentile	17.00	17.00	17.00	17.00	17.00	17.00	17.00	17.00	16.00

Figure 11: Time spent to process the GET request by the server for replication experiments

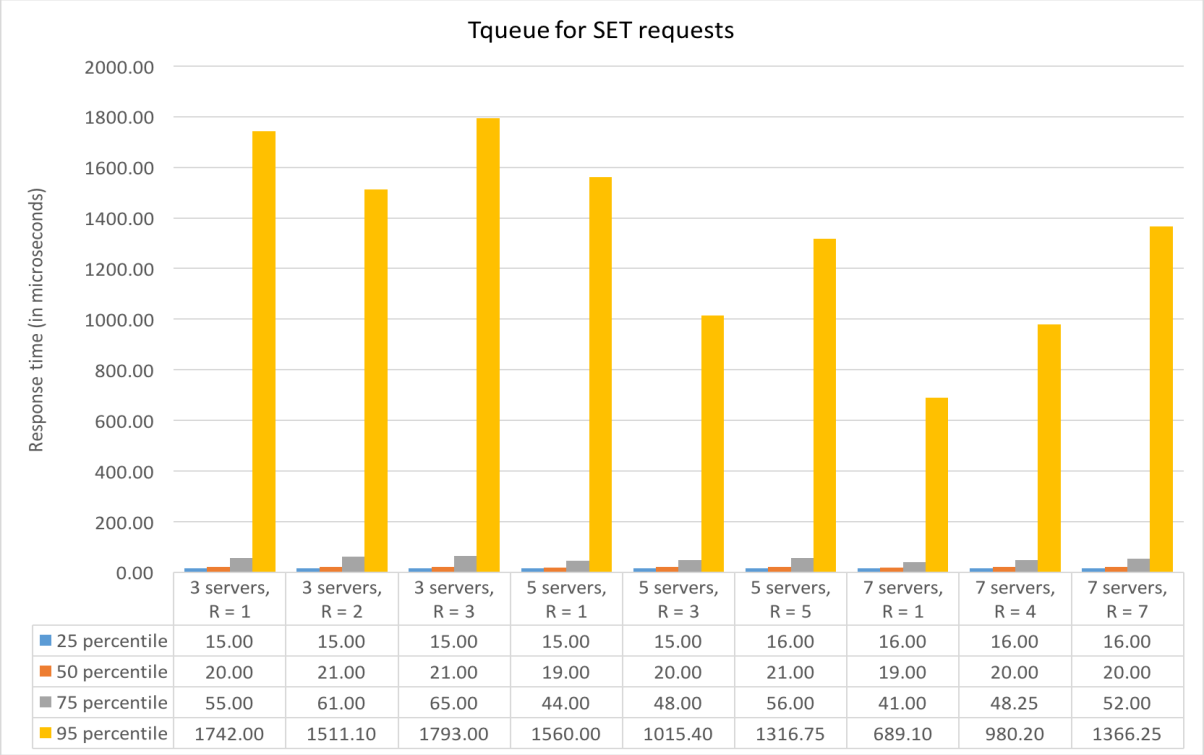
Number of servers	3,5,7
Number of client machines	3
Virtual clients / machine	80
Workload	Key 16B, Value 128B, Writes 1%, 5% and 10%
Middleware(R=)	No Replication and Full-Replication
Number of threads in pool	16
Runtime x repetitions	180s x 5
Log files	writeexp

3.2 Hypothesis

As per the reference document provided, I attempt to answer all the questions asked for the System Under Test (SUT). Since we have already analysed the effect of replication and the increase in servers in the section above, I analyse the system by looking at the variations for write percentages across two cases:

- when there is no replication
- when there is full replication

For a fixed number of servers, in the case of no replication, I expect to see no change in the response time and throughput across the variation in write percentages from 1 to 10. My logic behind this notion is that when there is no replication, increasing the write percentages have no impact on the throughput and the response time as the SET operations are asynchronous and dont add an overhead.
For a fixed number of servers, in the case of full replication, I expect to see an increase in the response time as the time spent in the middleware to process the request from the server - T_{server} - will increase as the write operation waits to receive all the responses from the secondary servers and only then sends back the response to the client. Therefore, as the write ratio is increased from 1 to 10% I also expect the throughput to decrease. But owing to the observations made in the previous section, I am a little skeptical about the margin of this decrement.



	3 servers, R = 1	3 servers, R = 2	3 servers, R = 3	5 servers, R = 1	5 servers, R = 3	5 servers, R = 5	7 servers, R = 1	7 servers, R = 4	7 servers, R = 7
25 percentile	15.00	15.00	15.00	15.00	15.00	16.00	16.00	16.00	16.00
50 percentile	20.00	21.00	21.00	19.00	20.00	21.00	19.00	20.00	20.00
75 percentile	55.00	61.00	65.00	44.00	48.00	56.00	41.00	48.25	52.00
95 percentile	1742.00	1511.10	1793.00	1560.00	1015.40	1316.75	689.10	980.20	1366.25

Figure 12: Time spent in the queue for SET requests for replication experiments

Finally, I expect the biggest impact on performance to occur for 7 servers with full replication and a 10% write ratio compared to a system with 3 servers and a 1% write ratio with no replication.

3.3 Throughput

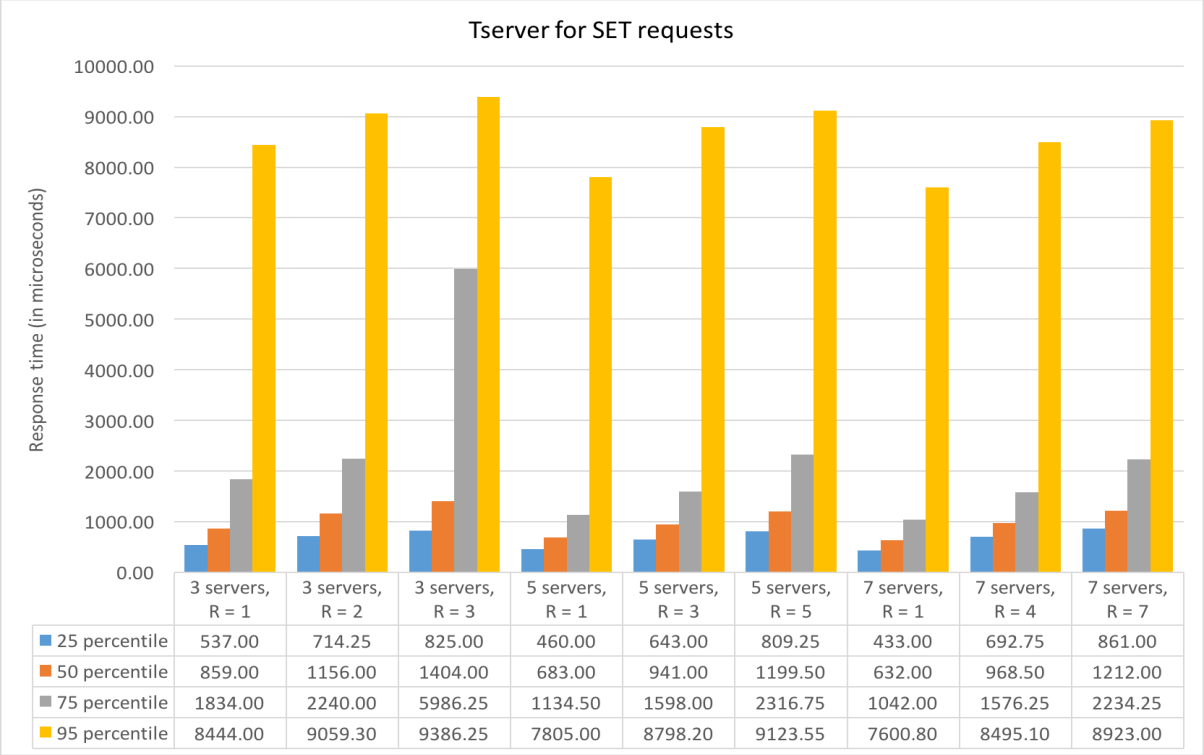
As we can see from the plot for the no replication case in **Figure 14**, the values for 1, 5, and 10% write ratios are similar across the same number of memcached servers. This can also be corroborated using the fact that the measurements vary by at most 450 ops/sec which is fairly low when factored with the range of throughput values (12-13k ops/sec). Even this difference occurs in the case of 3 servers and 10% writes, whose standard deviation is the highest across other measurements. This is in alignment with the hypothesis that for no replication, the variation in write percentages has no impact on the throughput as the asynchronous writes have no overhead.

As we can see from the plot for the full replication case in **Figure 15**, the values for 1, 5, and 10% write ratios are noticeably different across the same number of memcached servers. In fact, as the number of servers increases, the difference in throughput becomes more pronounced. Across 3, 5 and 7 memcached servers, 7 servers with 10% writes has a throughput difference of about 2000 ops/sec as compared to 1%writes. For 5 servers, this difference is a modest 1000 ops/sec and for 3 servers, even lesser. This behaviour is expected, since replication incurs an overhead in terms of waiting for individual responses from the replication servers. Thus, this observation falls in accordance to the hypothesis provided above.

3.4 Response Time

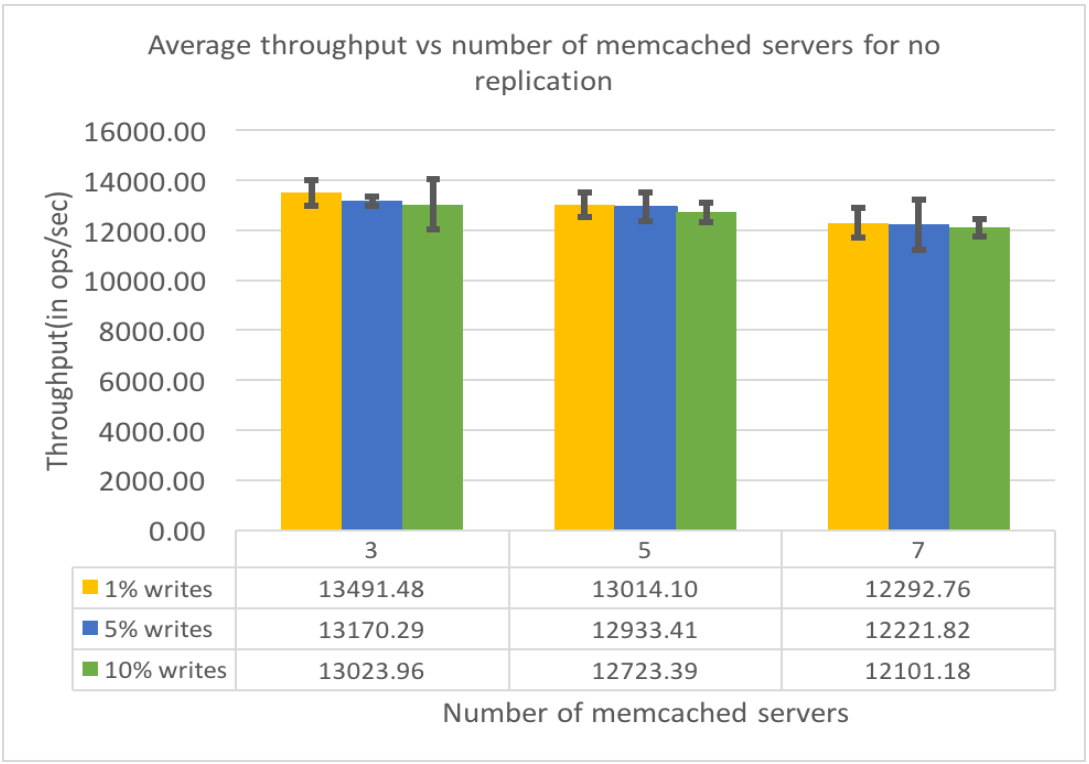
Since we are varying the percentage of write operations, I think it is only justifiable to look at the CDF plots for SET requests as the GET requests are essentially not affected at all by replication. I also verified this by looking at the CDF plots from the memaslap data for the GET requests and the observations essentially overlap, thus, allowing me to imply that the percentage of writes don't affect the GET requests.

Also, since most of the crucial points in the memaslap log distribution exist between the time range of 16384 to 32768 microseconds, in order to reduce redundancy and be more concise, I use a table to



	3 servers, R = 1	3 servers, R = 2	3 servers, R = 3	5 servers, R = 1	5 servers, R = 3	5 servers, R = 5	7 servers, R = 1	7 servers, R = 4	7 servers, R = 7
25 percentile	537.00	714.25	825.00	460.00	643.00	809.25	433.00	692.75	861.00
50 percentile	859.00	1156.00	1404.00	683.00	941.00	1199.50	632.00	968.50	1212.00
75 percentile	1834.00	2240.00	5986.25	1134.50	1598.00	2316.75	1042.00	1576.25	2234.25
95 percentile	8444.00	9059.30	9386.25	7805.00	8798.20	9123.55	7600.80	8495.10	8923.00

Figure 13: Time spent to process the SET request by the server for replication experiments



	3	5	7
1% writes	13491.48	13014.10	12292.76
5% writes	13170.29	12933.41	12221.82
10% writes	13023.96	12723.39	12101.18

Figure 14: Plot showing the variation of throughput for different memcached servers with different percentages of writes and no replication

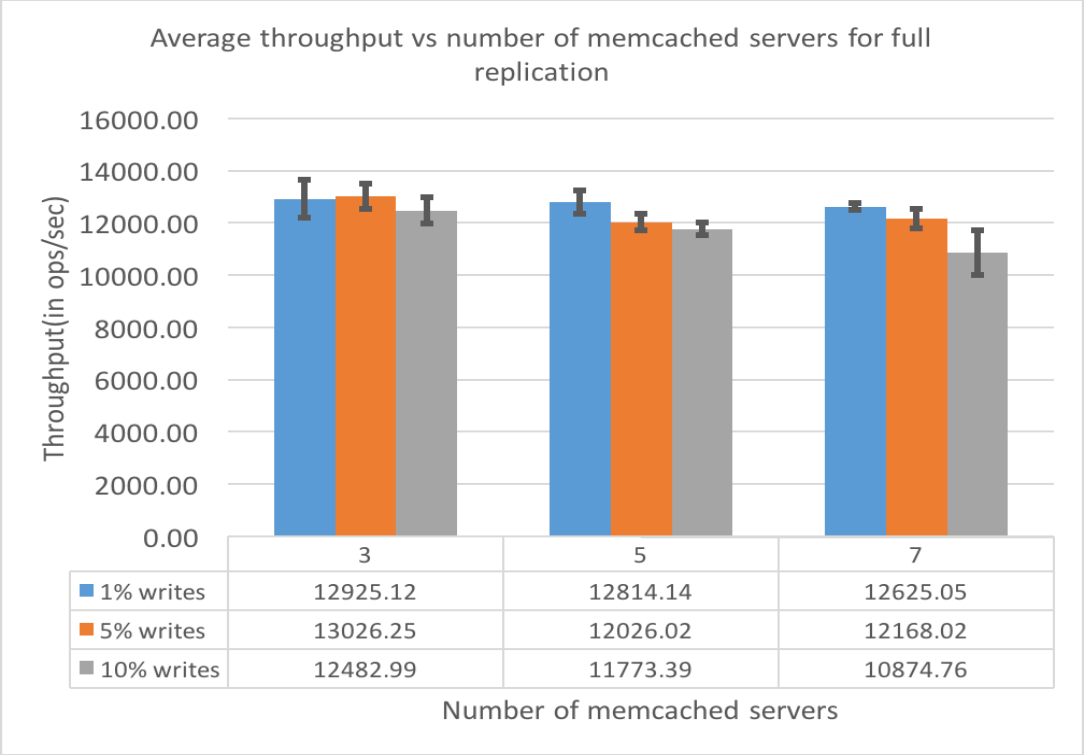


Figure 15: Plot showing the variation of throughput for different memcached servers with different percentages of writes and full replication

showcase the % of requests that are completed at these time ranges for different configurations instead of the CDF plots.

No replication						
	Within 16384 microseconds			Within 32768 microseconds		
Memcached servers	1% writes	5% writes	10% writes	1% writes	5% writes	10% writes
3	51	49	48	96	96	95
5	41	40	38	96	96	95
7	40	38	36	95	95	95

Full replication						
	Within 16384 microseconds			Within 32768 microseconds		
Memcached servers	1% writes	5% writes	10% writes	1% writes	5% writes	10% writes
3	57	49	42	96	95	93
5	42	34	29	95	94	91
7	37	31	25	94	94	89

Referencing the values from the table above, we see that in the case of no replication, the percentile values at 1%, 5% and 10% writes for 3, 5 and 7 are almost equal and don't vary by more than a percentile of 2. However, in the case of full replication, this is not the case. In the bucket of 16384 microseconds, the variation in percentile is much larger (6-8) and it can be seen that as the percentage of writes increases, the percentile of requests whose response time is less than that time range also decrease suggesting that the response time has increased. The largest drop in the percentage of operations completed is for the 10% write ratio. The write operations during replication are more expensive, thus the largest ratio of 10% results in the least operations completed. Although these measurements do not suggest any conclusion on their own, but they also do not exhibit a behaviour that would contradict the hypothesis made earlier.

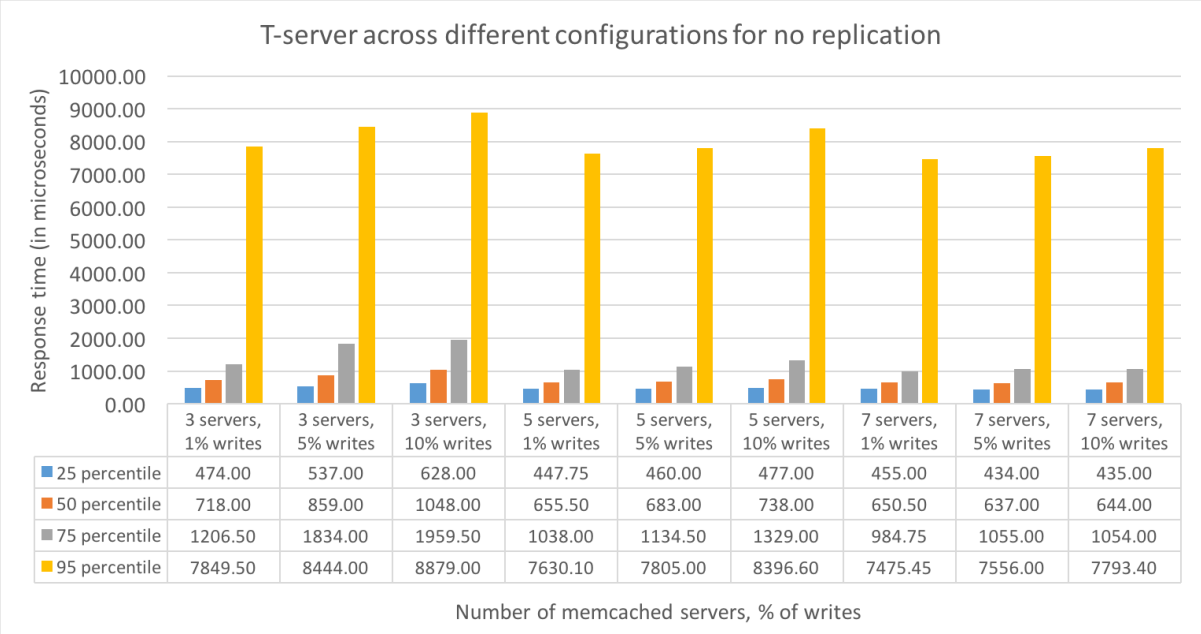
Thus, to get a more complete picture, I also look at the middleware logs. The logs that are of interest are the T_{server} logs as they are the one that see a major difference in cost across the write percentages.

As we can see from **Figure 16**, in the case of no replication, barring the increase from 1% to 5% in the 75th percentile for 3 servers, the percentile graphs look the same across the different write percentage configurations, thus supporting the hypothesis. In the case of 3 servers, it is likely that due to the large number of requests from 1% to 5%, the load per memcached server increases causing the server time to increase by a factor of 0.5.

Moreover, from **Figure 17**, in the case of full replication, we see a steady increase with the increase in write percentages for a fixed number of servers. This pattern is especially noticeable in the case of the 75th percentile where the increase factor is almost 3 going from 1% writes to 5% writes. As mentioned above, in the case of 3 servers, it is likely that due to the large number of requests from 1% to 5%, the load per memcached server increases causing the server time to increase at 5% itself. This increase in T_{server} for full replication seems reasonable as it is the most expensive operation in a write request and since we increase the number of writes, we also accumulate these expensive operations.

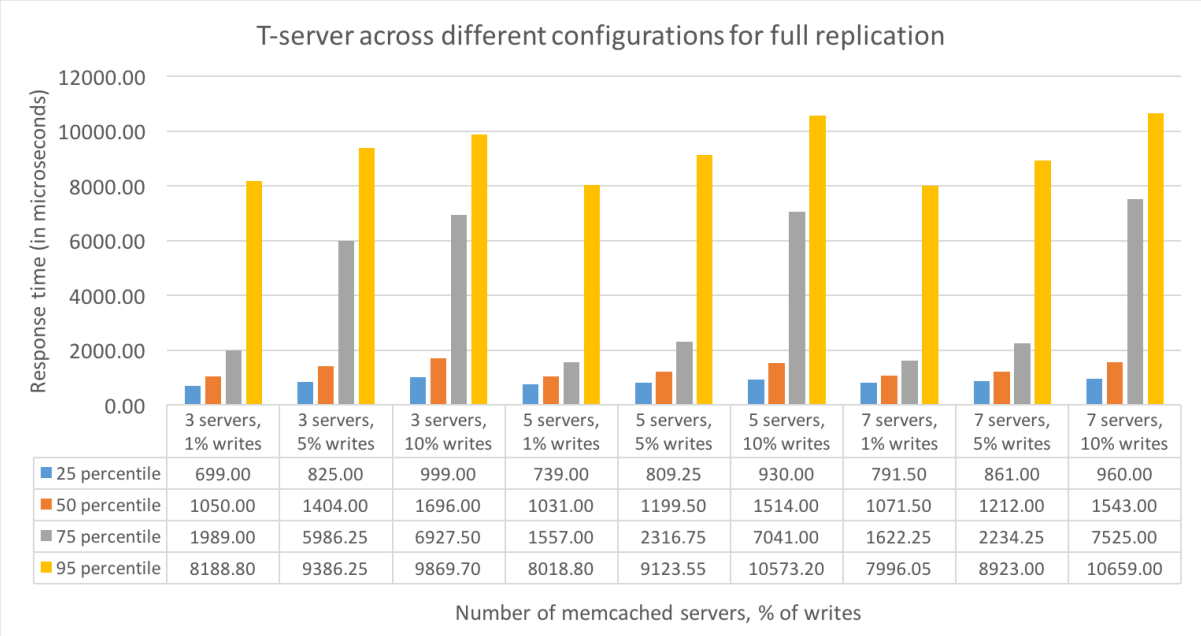
Therefore, as mentioned in the hypothesis, considering the base case to be 3 servers with 1% writes and no replication, the biggest impact on performance occurs for a system with 7 memcached servers and full replication with 10% writes. The reasons for this can be summarised as:

- Full replication writes are expensive than no replication as for each request, the worker thread also has to wait and parse the responses from all the secondary servers.
- The greater the number of servers, the costlier the write operation per request for full replication.
- Greater % of writes also mean lesser % of reads. It is effectively replacing cheaper read operations with more expensive write operations.



	3 servers, 1% writes	3 servers, 5% writes	3 servers, 10% writes	5 servers, 1% writes	5 servers, 5% writes	5 servers, 10% writes	7 servers, 1% writes	7 servers, 5% writes	7 servers, 10% writes
25 percentile	474.00	537.00	628.00	447.75	460.00	477.00	455.00	434.00	435.00
50 percentile	718.00	859.00	1048.00	655.50	683.00	738.00	650.50	637.00	644.00
75 percentile	1206.50	1834.00	1959.50	1038.00	1134.50	1329.00	984.75	1055.00	1054.00
95 percentile	7849.50	8444.00	8879.00	7630.10	7805.00	8396.60	7475.45	7556.00	7793.40

Figure 16: Plot showing the variation of T_{server} for different server configurations, write percentages and no replication



	3 servers, 1% writes	3 servers, 5% writes	3 servers, 10% writes	5 servers, 1% writes	5 servers, 5% writes	5 servers, 10% writes	7 servers, 1% writes	7 servers, 5% writes	7 servers, 10% writes
25 percentile	699.00	825.00	999.00	739.00	809.25	930.00	791.50	861.00	960.00
50 percentile	1050.00	1404.00	1696.00	1031.00	1199.50	1514.00	1071.50	1212.00	1543.00
75 percentile	1989.00	5986.25	6927.50	1557.00	2316.75	7041.00	1622.25	2234.25	7525.00
95 percentile	8188.80	9386.25	9869.70	8018.80	9123.55	10573.20	7996.05	8923.00	10659.00

Figure 17: Plot showing the variation of T_{server} for different server configurations, write percentages and full replication

Logfile listing

Each logfile has a README.txt file inside which maps the corresponding experiment number to the parameters used to run that experiment.

Short name	**Location**
maxtpexp	https://gitlab.inf.ethz.ch/ragrawal/.../maxtpexp.zip
threadexp	https://gitlab.inf.ethz.ch/ragrawal/.../threadexp.zip
vclientexp	https://gitlab.inf.ethz.ch/ragrawal/.../vclientexp.zip
repexp	https://gitlab.inf.ethz.ch/ragrawal/.../repexp.zip
writeexp	https://gitlab.inf.ethz.ch/ragrawal/.../writeexp.zip