Advanced Systems Lab (Fall'16) – Second Milestone

Name: *Lukas Burkhalter*
Legi number: *12-915-914*

Grading

Section	**Points**
1	
2	
3	
Total	

1 Maximum Throughput

In this section, we try to find the maximum throughput of our system for a certain thread configuration for the middleware with a read-only workload without replication. We first present, the experimental setup, continue with our prediction of the expected behavior. Then, we show the experimental results and, finally, discuss them based on our predictions.

1.1 Experimental Setup

Our experiment aims to find the highest throughput of our system with a minimal number of reader threads. Therefore, we vary the following parameters.

- The number of virtual clients
- The number reader threads per partition

We consider a read-only workload and perform the writes before the experiment starts. The experiment duration per configuration is 3 minutes. We use 5 load generating and 5 backend memchached servers on A2 basic machines. We set the window size of memaslap to 1000, and use a read-only workload configuration with small 128 value contents. The middleware runs on an A4 basic machine with fixed 2 backwriter threads. All memcached servers are running on 1 thread. We vary the number of reader threads in the middleware between 8,16,32,48 and 64 threads and the clients based on the observed results and interesting points. Each client machine runs always the same number of virtual clients.

We log on the client machines in a 1s seconds granularity and sample each 100th request in the middleware. Due to long setup cost of the experiments, we only repeat the points 3 times, which are close to the maximum throughput. The results may not be as accurate, because of the long setup costs the experimental environment in the cloud may change (i.e more traffic, hardware occupation etc.). From the 3 minute experiment, we subtract cooldown/warmup costs of 30 seconds on each end. The main parameters are summarized in the following table.

Number of servers	5
Number of client machines	5
Virtual clients / machine	0-100
Workload	Key 16B, Value 128B, Writes 0
Middleware Threads	8-64 reader threads, 2 write back threads
Runtime x repetitions	3min x (near max tp) 3
Log files	maxtpexp.zip

Metrics In the max throughput experiment, we are interested in two main metrics, the throughput in operations per second and a detailed breakdown of time spent in the middleware for a read operation type in milliseconds. The throughput data is extracted from the client machine side logs, whereas the middleware times are extracted from the middleware sampling log.

1.2 Expected Behavior

In milestone one, we already had a first impression of the throughput performance based on the stability experiment. However, the performance was poor compared to the baseline single memcached server. Therefore, we hope that by increasing the number of virtual clients, changing the reader threads configuration and adding two more memcached backend server, we see an improved maximum throughput, which is in the region of our baseline or even higher. Our system should be able to handle more clients and throughput, as our goal is to load balance the traffic among multiple memcached servers.

Middleware time breakdown	Description
mw time	The total time spend in the middleware, without the server costs included. $(T_{respsend} - T_{reqreceived}) - (T_{serverin} - T_{serverout})$
mc server time	The time spend in the network to the memcached server and memcached processing time. $T_{serverin} - T_{serverout}$
queue time	Part of the middleware time. The time spend in partition queue. $T_{queueout} - T_{queuein}$
processing time	Processing time in the middleware, also included in the middleware time. Since the observed write back queue time is mostly small, the processing time also includes the write back queue time. $mwtime - queuetime$

Table 1: A description of the different middleware time breakdowns used in the plots.

1.3 Experimental Results

In this Subsection, we present our results. An overview of the different throughputs for each configuration is presented in Figure 1. The x-axis shows the number of clients, whereas the y-axis shows the average number of operations per second. Note that not each client/reader-thread configuration has been evaluated, but only the interesting points. For each reader thread configuration, a line plot of the results is shown.

Additionally, we investigated a detailed time breakdown of the time spent in the middleware for an read operation type, which are depicted in Figure 2 and 3. A detailed description of each middleware time part is given in Table 1. The x-axis shows the number of clients and the y-axis the time in milliseconds. For 16,32,64 reader threads, a breakdown of time spent in middleware is provided with median and percentiles.

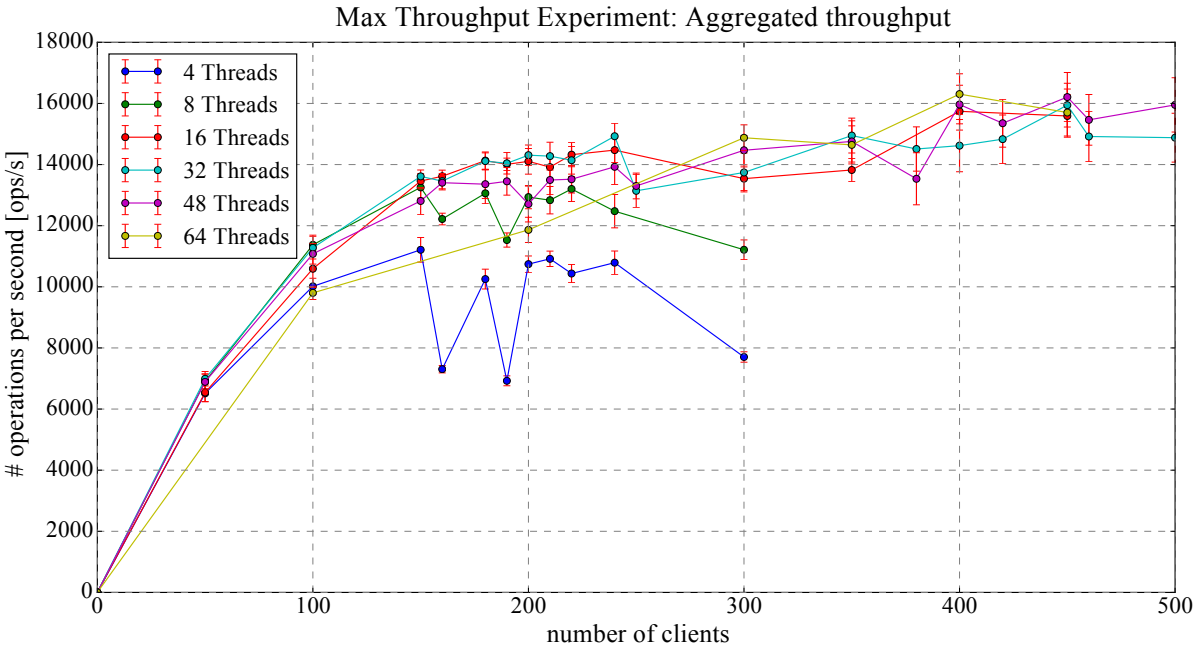

Figure 1: Depicts the hroughput for varying number of clients and different reader thread configurations

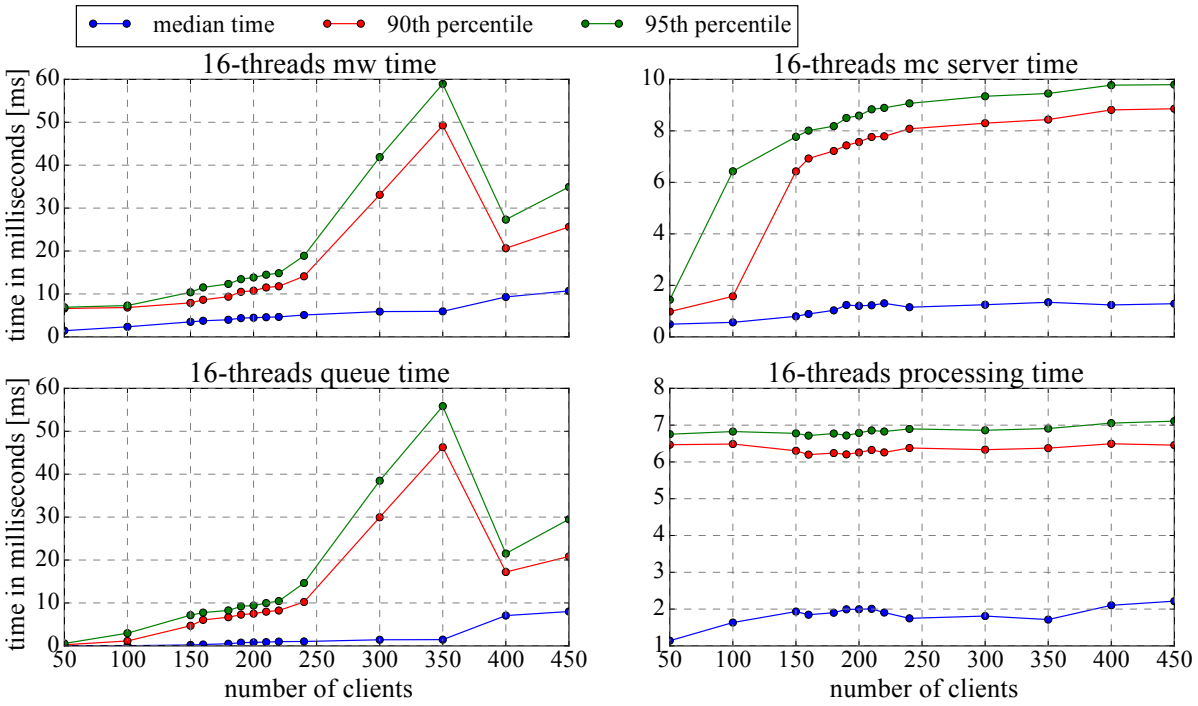

Figure 2: Depicts the middleware time breakdown for get requests in the max throughput experiment for a 16 reader thread configuration.

1.4 Discussion

We expected to improve our performance significantly in terms of throughput compared to the stability experiment in milestone 1 by adding more memcached servers and more virtual clients. However, if we look at Figure 1, we still are far away from the baseline throughput in milestone 1. After around 200 clients all configurations start to flatten out and the maximum is observed around 450 clients with 48 threads with 16000 operations per second. However, compared to 16 threads there is only a small improvement in performance, considering three times as many threads. 64 threads also achieve the maximum throughput at around 400, but still do not improve significantly compared to the 48. Considering these observations, the main questions to ask are.

- Why does the throughput starts to flatten out for nearly all configurations and why we achieve the highest throughput around 450 clients.
- Why adding more threads only results in small or no performance increase after a certain amount of threads are used i.e 16.

For answering these question, we need to have a closer look at the middleware time breakdowns for the interesting configurations. We selected 16, 32 and 48 depicted in Figure 2 and 3. If we look at the middleware only time for 16 threads in Figure 2, we can see that the median time increases until about 200 clients, then it stays same to about 350 and then again starts increasing. We can observe the same pattern in the queue times. The higher percentiles a growing permanently as more clients are added. This behavior can be explained by looking at the queue time, first, the queues are empty and start filling up. The reader threads start working and peak the requests directly from the queue. At the certain load, the threads cannot keep up with the load and start competing on the queues, and the request has to wait longer in the queues. The queue time starts growing, because we use synchronous communication and the are all working on a request. Most of the work of the reader threads is waiting on the socket

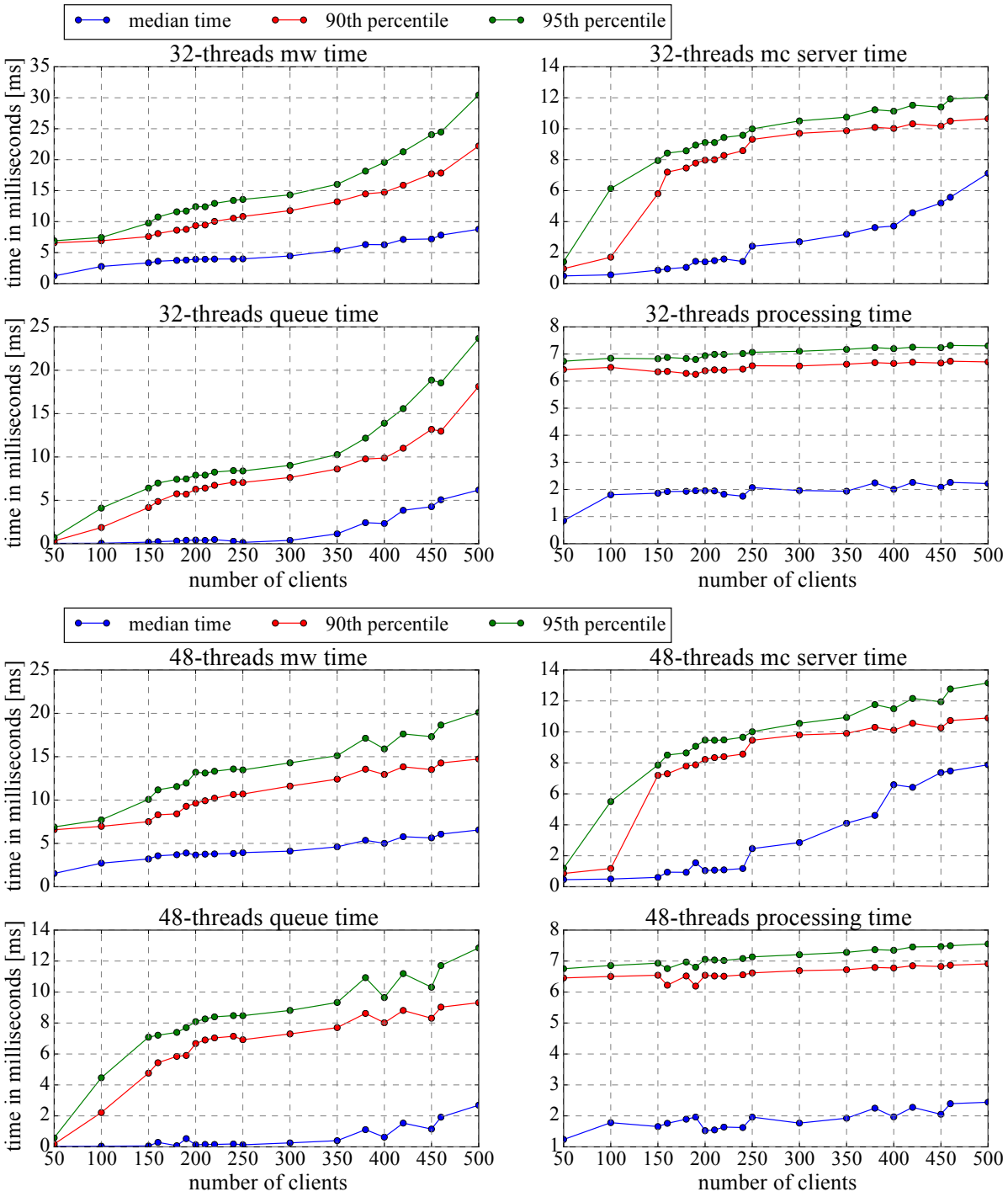

Figure 3: Depicts the middleware time breakdown for get requests in the max throughput experiment for a certain reader thread configuration.

for the requests and reading them. In this waiting time, they do not perform any work and cannot work on a new request, which results in waiting time in the queues.

This problem can be addressed by adding more reader threads, as most of the time they only sleep. If we look at the 32 thread configuration, we observe a much lower queue time until 350 clients compared to the 16 thread setting, since more threads can work on the queue and there work mostly consist of waiting. However, we can still observe an increase of the middleware time, despite having lower queue times. If we look at the server time, which includes the round

trip time, the memcached processing time and the time to read the request from the wire, we can observe an incremental increase. This is due to the fact the way more threads compete for CPU resources, requests in the queues and I/O time, which introduces synchronization and scheduling cost. Reading the request from the wire is the largest task a reader thread has to perform and requires CPU time. As the CPU times and I/O time for each thread are limited and new costs are introduced, we observe an increasing server time. The same holds for the 48 threads case.

Summarized, there is a trade-off between competing among CPU time and I/O time with more threads and the waiting time on the connection with fewer threads. If we add more threads the waiting time of the connection becomes less relevant, but new scheduling, synchronization and CPU time competition cost are added. Furthermore, we can observe more instability (more variations) when more threads are added, as the resource scheduling becomes more important, which introduces some random noise. The 48 threads configuration seems to trade off this costs in the best way, however also introduces some instability in the throughput with more clients. The longer middleware times directly influences the throughput of the system and causes the flattening of the curve. We also did not consider writes, which introduces new heavy CPU workloads on different threads, which might also have a big impact on this cost trade-off.

Summarized, we only achieved slightly higher throughput than in the stability experiment from the first milestone, which is disappointing. We expected a better performance with more threads and servers, but still are way below the throughputs achieved in the baseline. Our system, therefore, becomes obsolete.

2 Effect of Replication

In this section, we try to understand the effect of replication on the system behavior. The middleware can replicate a write-request to up to n memcached servers, where n is the total number of memcached servers. To understand the behavior of the middleware for different replication factors, we designed an experiment that we present in the following subsections. First, we look at the experimental setup and the systems configurations, then we proceed with the hypothesis for the experiment. The results are presented in subsection 2.3 and ,finally, we compare the system behavior to the expected behavior and explain the experimental observations.

2.1 Experimental Setup

To understand the effect of replication on our system, we vary the following configuration parameters and run each configuration for 3 minutes.

- The number of memcached servers N
- The replication factor S

The other parameters are fixed to a constant value. Based on the maximum throughput experiment, we decided to use 3 load generating machines with 64 clients each in combination with 16 reader threads per partition in the middleware. Despite 36/48 threads had a higher throughput in the first experiment, we decided to use 16 threads because we have 5 percent write load and the max throughput experimented was conducted with a read-only load combined with one replication configuration. In the maximum throughput experiment, we showed that the system starts to be saturated after 192 virtual clients with 16 reader threads and the throughout was relatively close to the max measured throughput. Furthermore, we observed a throughput in the stability experiment, which is close to measured maximum throughput, despite having only three memcached servers and also writes in the load, which indicates that the number of memcached servers doesn't play a big role. The stability experiment used the same configuration. Therefore, we decided to use the 16 reader threads configuration for a good load on all setups.

In order to compare the effect of the replication, we keep the load fixed for all configurations. The load/middleware setup is similar to the stability experiment in the first milestone. We set the the number of write back threads to 2 and set the o memaslap parameter to 0.9 as for all previous experiments.

The number of memcached backends is varied between 3, 5 and 7, and the replication either to one, half or all. For the client and mecached machines, basic A2 machines were used, whereas the middleware was deployed on a basic A4 machine in the azure cloud. Each configuration was run for 3 minutes. In order to consider warm-up/cool-down costs, we subtracted 30s on each end from the 3 minutes. The log granularity on the client side was set to 1s and the write log granularity in the middleware was reduced to 10 for having more fine grained data for the set operation analysis. The important parameters are summarized in the following table.

Number of servers	3, 5, 7
Number of client machines	3
Virtual clients / machine	64
Workload	Key 16B, Value 128B, Writes 5
Middleware	Replicate to one, half, all
Middleware Threads	16 reader threads, 2 write back threads
Runtime x repetitions	3min x 5
Log files	repexp.zip

Metrics In the replication experiment, we are interested in three main metrics, the throughput in operations per second, a detailed breakdown of time spent in the middleware for each operation type and the response time from the client side in milliseconds. The throughput and client side response time data is extracted from the client machine side logs, whereas the middleware times are extracted from the middleware sampling log.

2.2 Expected Behavior

We expect that the general performance of our system increases as we add more memcached servers, as we have more backend servers available to balance the load on. Therefore, we suspect that a request spends less time in the middleware as the number of backend server increases. As a result, we should also measure a slightly higher throughput with more servers. As outlined in the first milestone, we believe that write requests are more expensive than reads. This gap should get bigger as we increase the replication factor because write requests become more expensive as more data has to be written to the network. Summarized, our system should overcome the scalability issues from one single memcached server by load balancing over multiple memcached servers.

2.3 Experimental Results

In this, subsection we present our experimental results. In Figure 4, 5, 6, we depict a detailed breakdown of the time spent in the middleware for get and set instructions separately. Each Figure summarizes the results for a certain replication factor (i.e one, half, all). The detailed description of each depicted middleware time is summarized in Table 1 . The x-axis represents the number of memcached servers the middleware balances on, whereas the y-axis shows the time in milliseconds. We show the median and the 90/95 percentile of the times spend in the middleware.

In Figure 7, we present the throughput of our system for all the experiment configurations. The x-axis shows the number of memcached servers and the y-axis the number of operations per second. We plotted the average throughput with its corresponding standard deviation.

In Figure 8, we present the client side response time. We plotted the average response time with standard deviation, for varying number of servers. We did not plot the median and percentiles, as the client side log had not enough precision for computing precise percentiles to be used in the comparison.

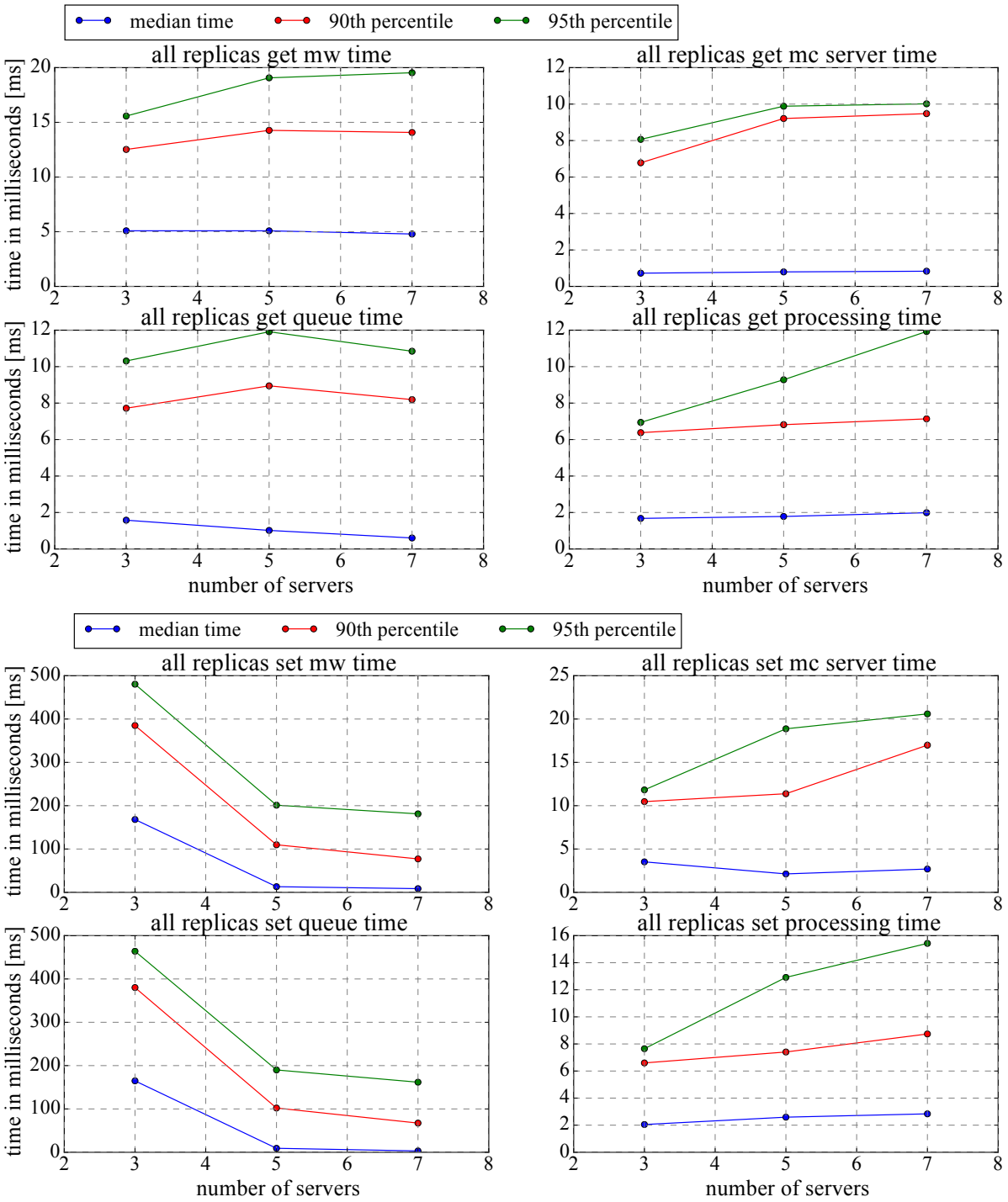

Figure 4: Depicts the middleware time breakdown for set/get requests with replication factor all.

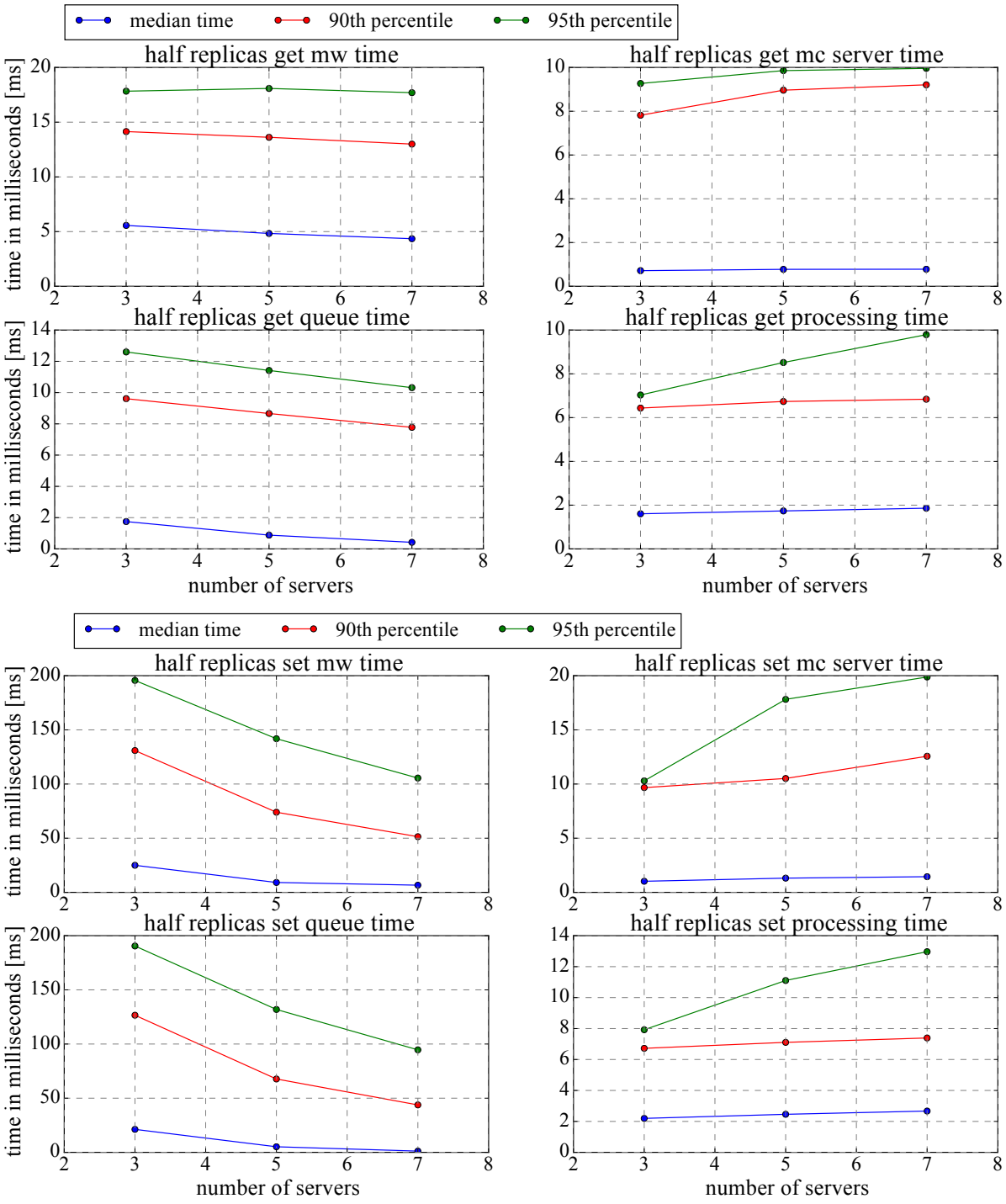

Figure 5: Depicts the middleware time breakdown for set/get request with replication factor half.

2.4 Comparison to Expected Behavior

In this experiment, we observe various non-expected behaviors . We expected to see a performance increase as we raise the number of memcached servers. However, the observation tells us a different story. If we look at the throughput plot in Figure 7, we do not see an increase in the throughput when the number of servers increases, instead we see a decrease. There is a small increase when we look at the half replication case, but only from 3 to 5 servers. Generally, we see a lower throughput if the replication factor is changed to a higher level.

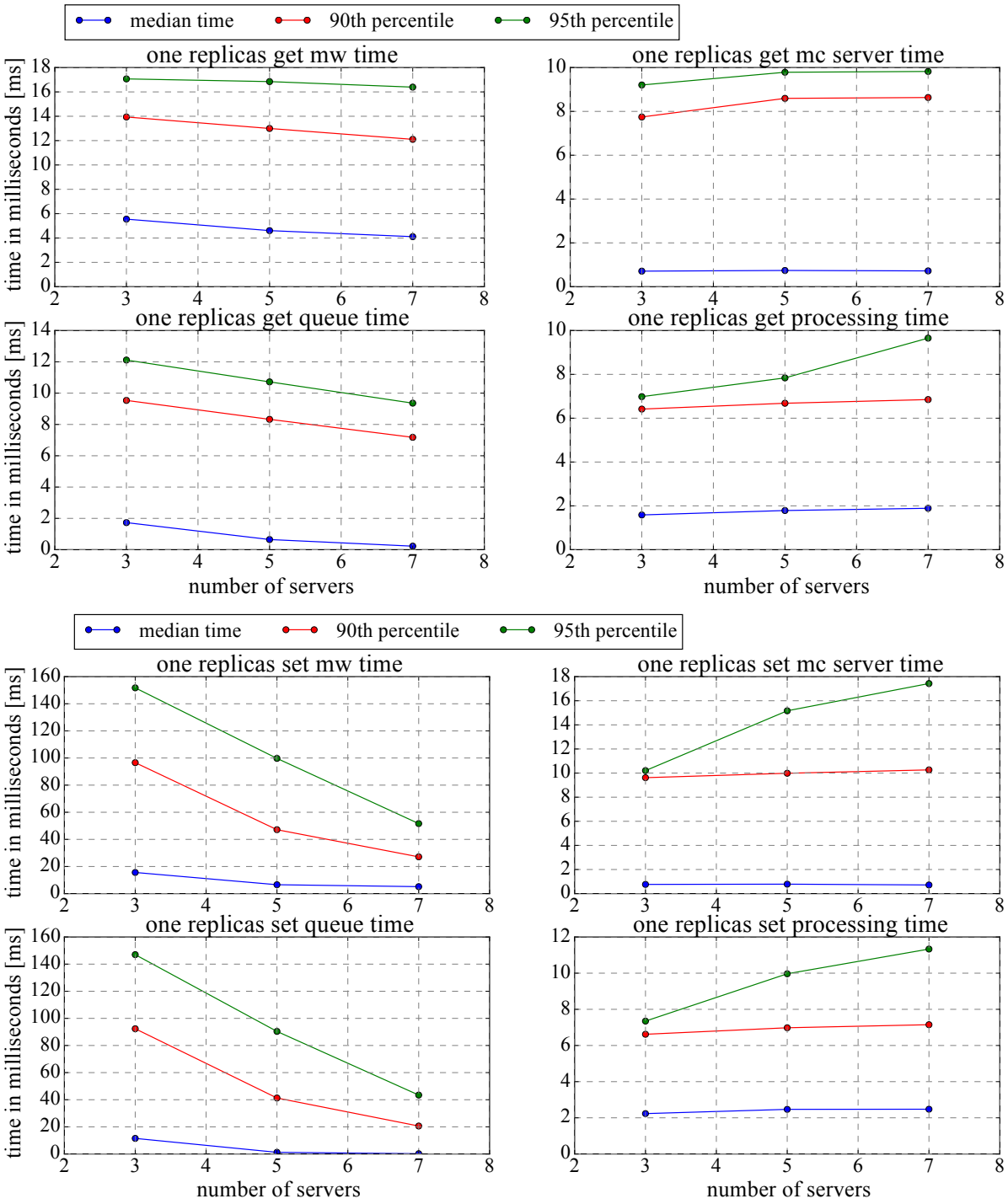

Figure 6: Depicts the middleware time breakdown for set/get request with no replication i.e one.

The time breakdown in the middleware also shows some interesting behavior. If we look at the Figures 4, 5, 6, we observe that the get requests middleware time is nearly not affected by the replication factor and the number of mechached servers, it's always around 5 ms, just the server processing time is slightly increasing. However, the set request behave differently, the operation gets more expensive as the replication factor increases but has a lower middleware time as the number of backends raises.

Based on this observations, we can explain the behavior by looking at the implementation

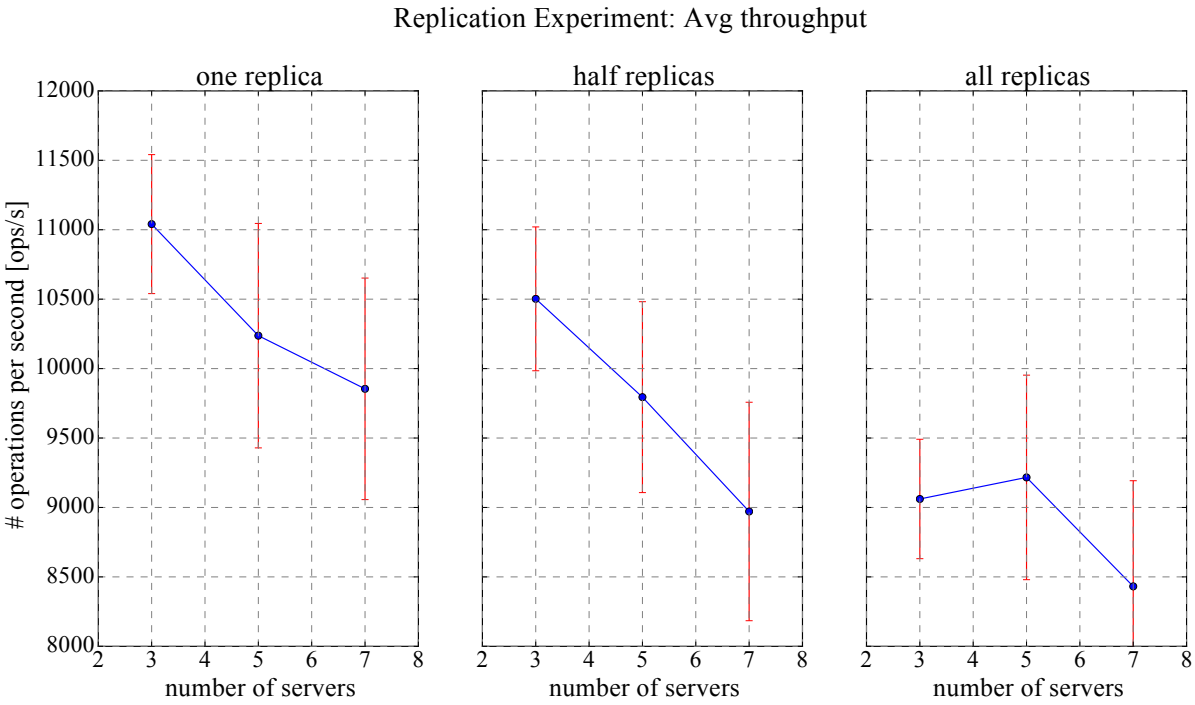

Figure 7: Depicts the middleware throughput per number of servers for different replication factors.

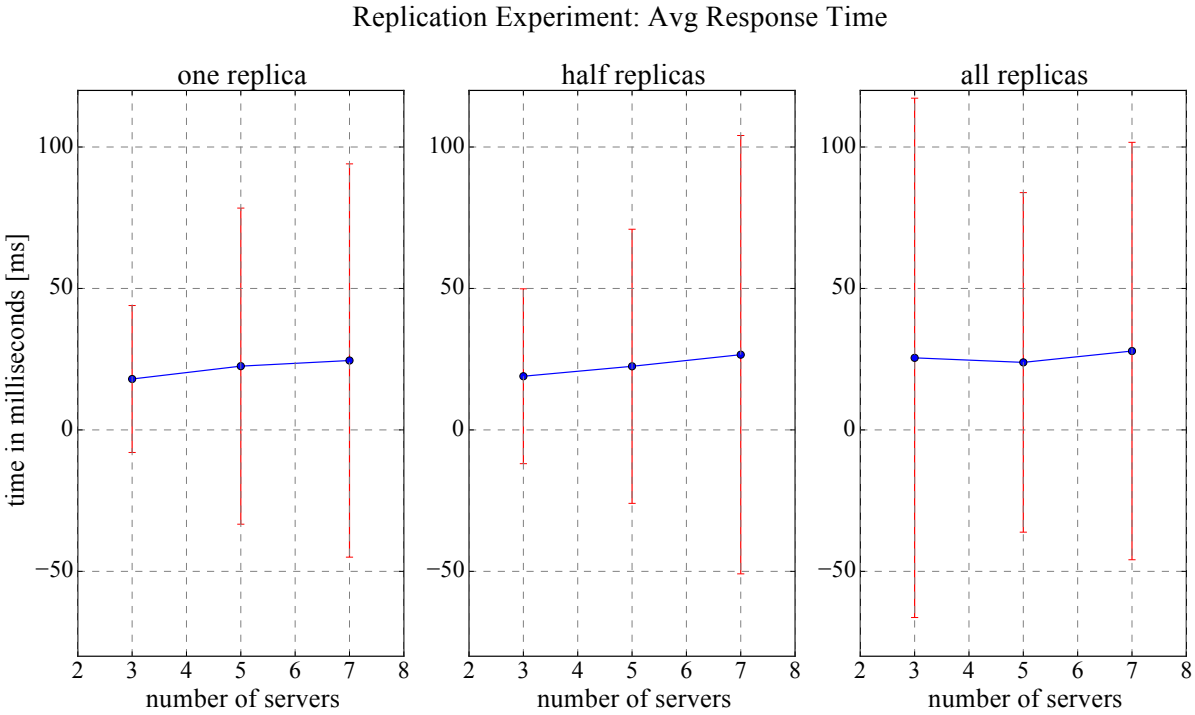

Figure 8: Depicts the response time from the client side per number of servers for different replication factors.

of the writer thread. The writer thread operates with asynchronous communication on NIO sockets. When the writer thread processes a request, he loops over the nio sockets and tries to send the data to one up to max all replication servers and receive the answers on them. Therefore, compared to a reader thread, a writer thread does not sleep on a socket and occupies

a lot of CPU time. This time logically increases when we replicate to more server as more data has to be written to the network. We see an exponential growth in middleware time as we increase the replication factor. This is due to the fact that we have to write to multiple connections, and also have to wait for multiple responses, while the thread has to compete with others for CPU and I/O time. When the number of servers grows, we see a big decrease in the middleware time for writer threads, because we also add more writer queues and threads. Since we have only one writer thread per partition, the number of writer threads equals the number of queues and partitions. As a result of more partitions, we reduce the queue time for each write request, which results in a decrease of middleware time.

In contrast, read request are nearly not affected by the replication factor. We do also see a reduced queue time if we raise the number of servers, but the server time and the processing time gets slightly more expensive. If we add more servers, we have a huge increase of reader threads in the system, and with more replications, also more CPU intensive worker threads are added. Since all the threads compete for resources like CPU time, I/O time, dequeue request, we also introduce costs for synchronization and scheduling. These costs compensate the fact that we have more reader threads, which add more costs for all threads in the system. Not only for the reader and writer threads, but also for the threads that read requests in and write requests back to the client.

The middleware time does not include the time for reading a request in from the client, but the request reading time is also affected by the scheduling and synchronization cost. The total response time on the client side (Figure 8), is slightly higher when these costs are added. Therefore, we see a decrease of throughput if we increase the number of partitions. The case for all replicas where it increases can be explained by the immense costs of writing to all partitions. The effect of reducing the queue time with more threads is higher weighted than the introduced cost for synchronization a scheduling from 3 to 5 memcached servers. Therefore, we see a low increase of throughput.

We also expected that our system will be more scalable than a single memcached server. But our system scales poorly. An ideal implementation would benefit from adding more server backends. If no synchronization, I/O, scheduling and hardware limitations are assumed, we should double the throughput when we double the number of memcached backends, since we doubled the memcached resources and could route half of the traffic to other servers. However, our implementation faces the presented limitations and scales poorly, which more or less makes it not usable, because one single memcached server performs much better.

3 Effect of Writes

In the writes experiment, we are interested in investigating the influences of writes on the system, as we previously experienced that writes cost more than read requests. We start off, with the experimental setup, followed by a discussion on the expected results. In Subsection 3.3, we present the results of the experiment and ,at the end, we discuss the results and compare them to our expected behavior.

3.1 Experimental Setup

To understand the effect of writes of our system, we vary the following parameters of our system.

- The number of memcached backends
- The percentage of writes in the workload
- The replication factor

Similar to the replication experiment, we use three load generating A2 basic machines with 64 virtual clients each. The middleware runs on an A4 basic machine with 16 reader threads and 2 writeback threads. Each configuration experiment was running for 3 minutes including warmup/cooldown phases of 30s. For experimental accuracy, we repeated each experiment 5 times.

We vary the write workload by setting the write load in the memaslap workload file to either 1, 5 or 10 percent. We set the o parameter to 0.9 and used a load of 128B per set request. The number of server backends was set to 3,5 or 7 and the replication factor was either one (i.e. no replication) or all. We used a logging granularity of 1s on the client side and reduced the log sampling rate for writes in the middleware to 10. In the following table, we summarized the most important parameters.

Number of servers	3, 5, 7
Number of client machines	3
Virtual clients / machine	64
Workload	Key 16B, Value 128B, Writes 1,5,10
Middleware	Replicate to one, all
Middleware Threads	16 reader threads, 2 write back threads
Runtime x repetitions	3min x 5
Log files	writesexp.zip

Metrics For quantifying the performance of our system, we use the throughput in operations per second and the response time in milliseconds from the client side log. Furthermore, we considered the middleware response time and a detailed breakdown of time spent in middleware by using the middleware log. We were not able to compute the median of the response time from the client side log, as the log does not provide enough precision for seeing differences between the experiment points.

3.2 Expected Behavior

Based on the observations from the previous experiments, we expect that writes, in general, are expensive for our system. Therefore, we assume that when the write workload is increased, we should observe a reduction in performance, because the number of expensive operations was increased.

3.3 Experimental Results

We provide several plots for illustrating the results of this experiment. In Figure 9, we show the average throughput of our system based on the different write loads and number of servers, measured from the client side. We also provide the measured response times from the client side operating with the different configurations (Figure 10). We show the average response time with standard deviation because the client log did not provide not enough detail to compute the median and percentiles. To contrast the client side response time, we show the middleware response time for read and write requests separately in Figure 11. The middleware response time is defined as the time a request enters the middleware until the request leaves the middleware ($T_ressend - T_reqreceived$). Finally, we give some comparison plots providing a detailed breakdown of time spent in middleware for some interesting configurations (Figure 12, 13, 14).

3.4 Discussion

In general, we expected to see a performance decrease, when a greater amount of expensive write operations are thrown at our system. If we look at the observations in the throughput and

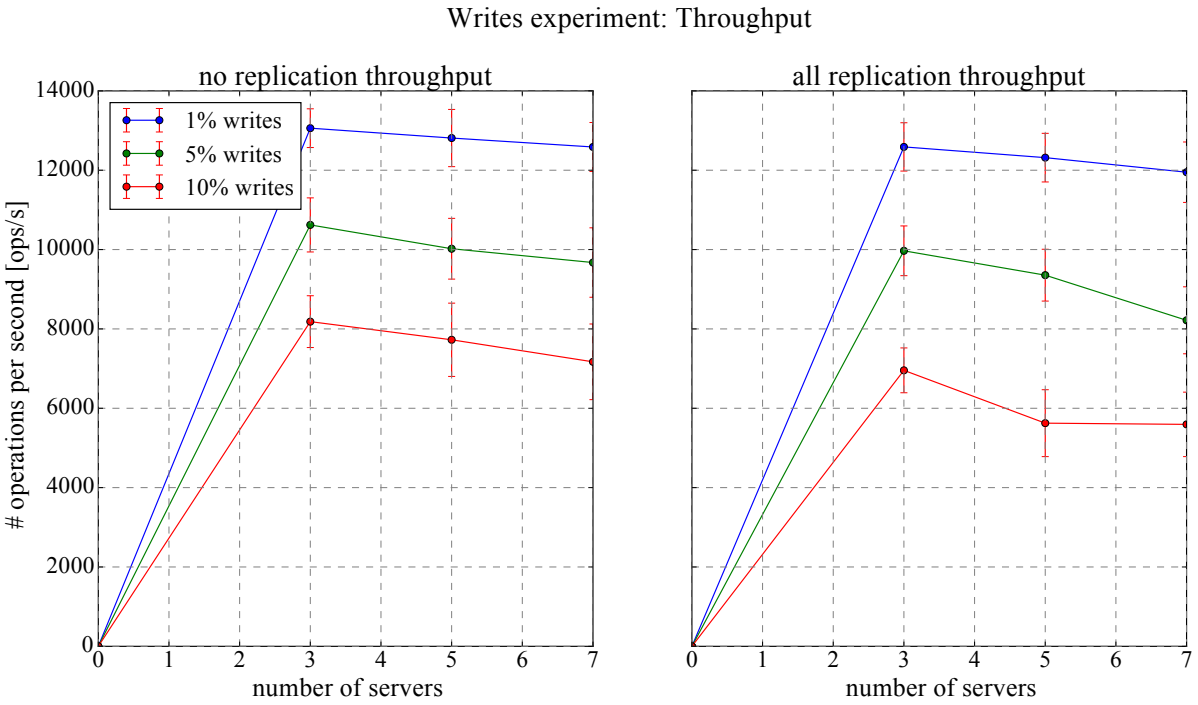

Figure 9: Depicts the middleware throughput per number of servers for different replication factors and different write workloads.

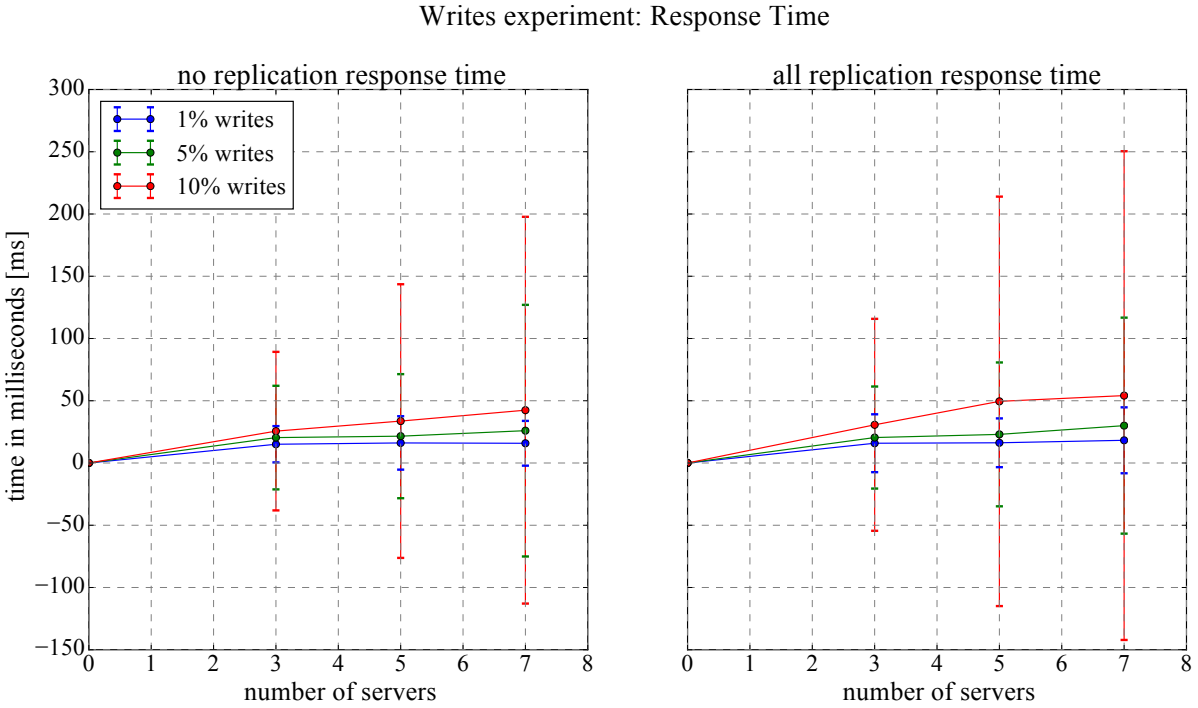

Figure 10: Depicts the response time measured from the client side for different write workloads and varying number of memcached servers.

response time plots from the client side, we exactly see this behavior (see Figure 9 and 10). We observe a lower throughput and a higher response time for a higher write workload. This holds for the non replicating as for the replicating case. For the all replication case, the corresponding throughputs are lower, because the write operations get more expensive, which is also observable

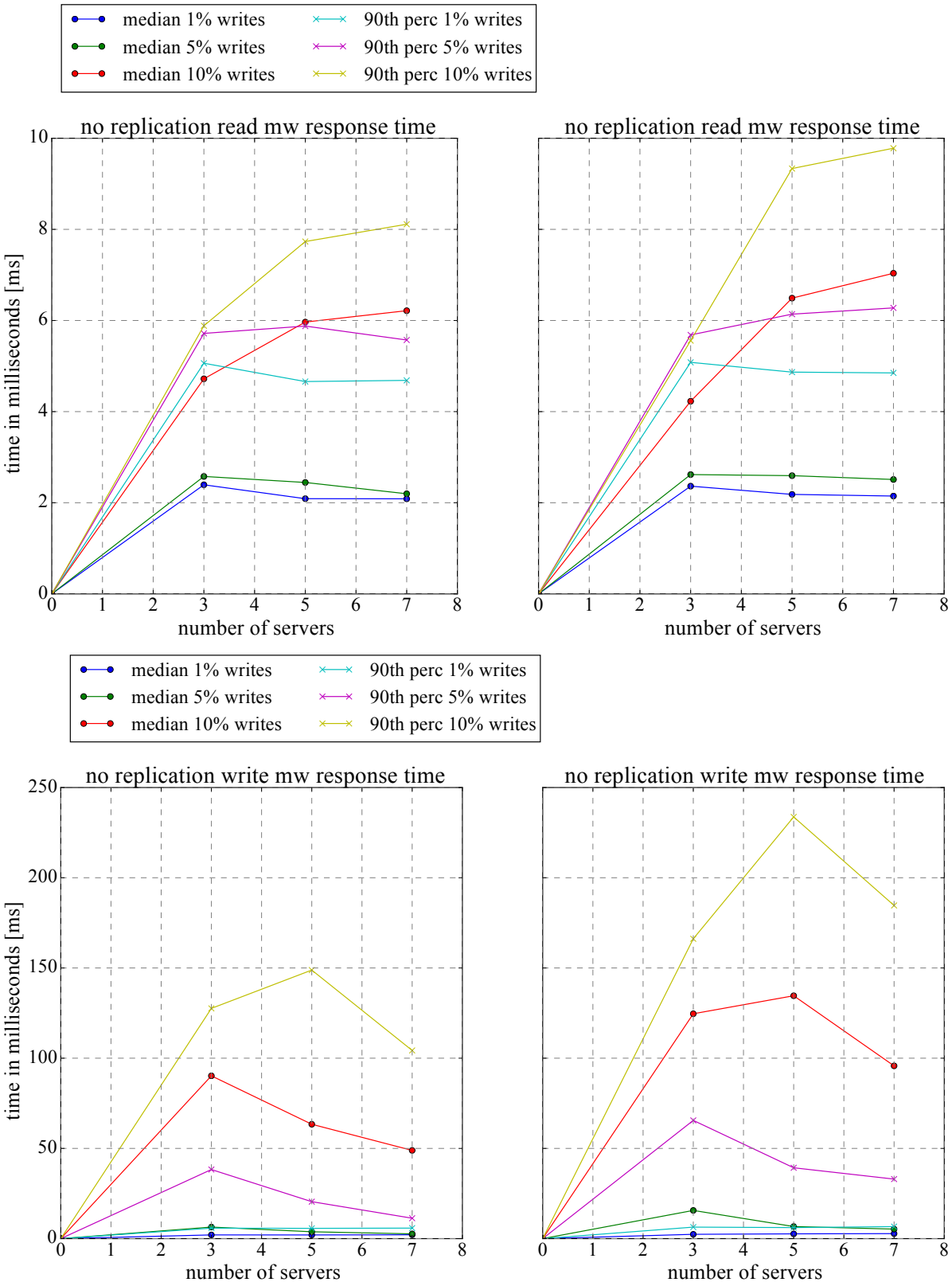

Figure 11: Depicts the middleware response time with no replication and to all replication.

in the response time plot. Similar to the previous experiment, the throughput decreases when the number of server increases and the response time grows.

However, more interesting behaviors can be observed in the middleware response time plots

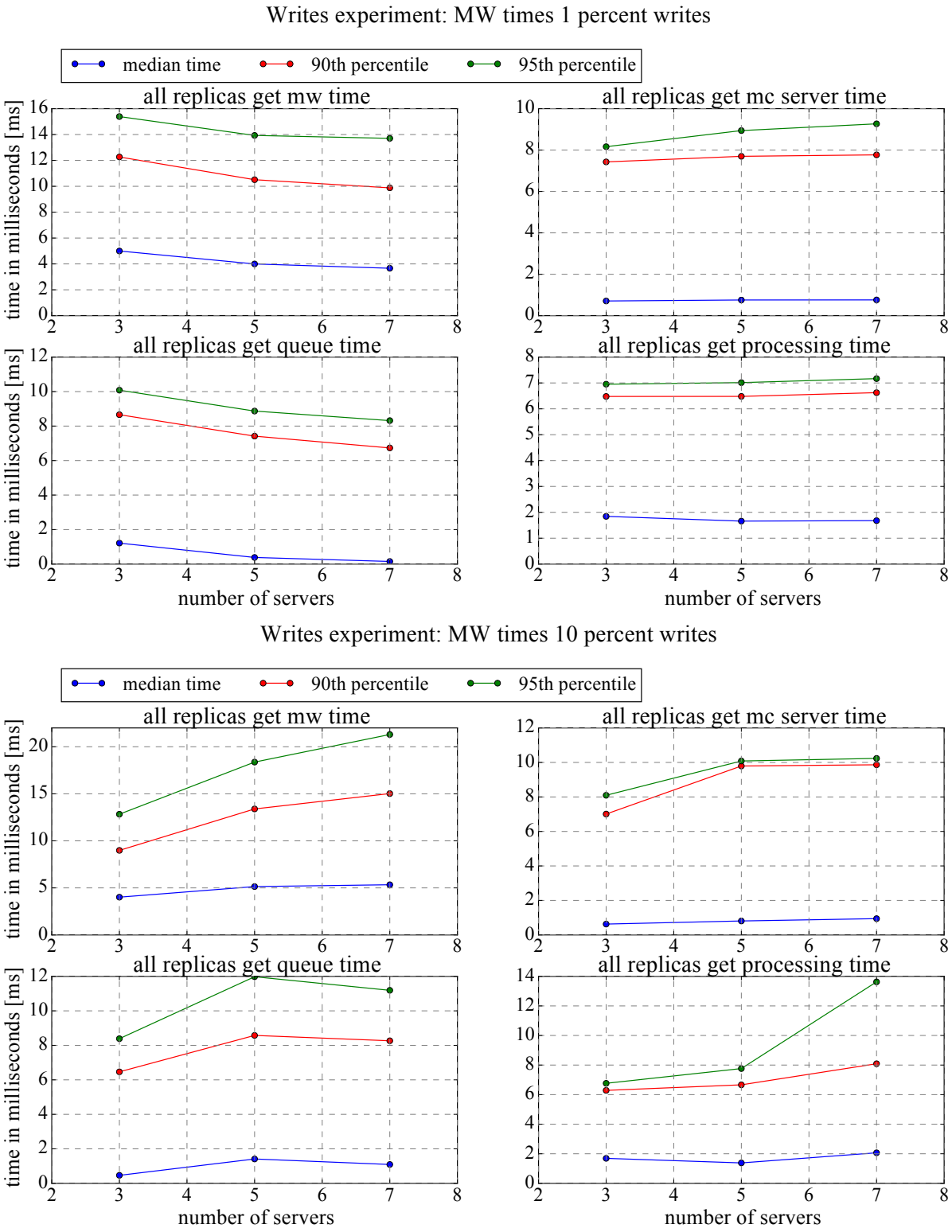

Figure 12: Depicts the middleware get time breakdown for a 1 and 10 percent workload and replication factor all.

(see Figure 11). If we look separately at read operations, we see that for the 1 and 5 percent workload, the middleware response time is more or less constant when the number of memcached server increases, but grows in the 10 percent workload case. This holds for the replicated and non replicated case. In the detailed middleware time breakdown plot in Figure 12, we can see

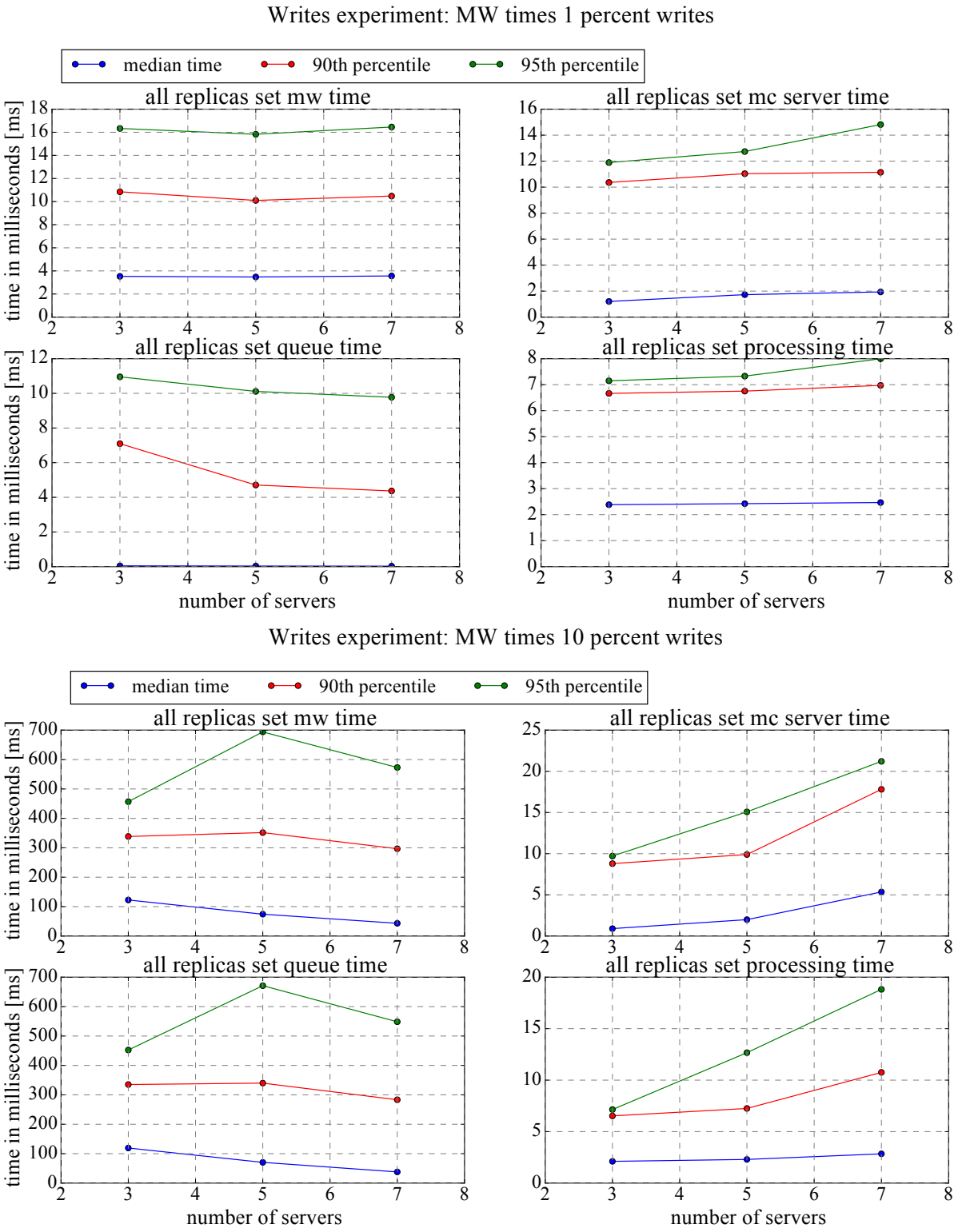

Figure 13: Depicts the middleware set time breakdown for a 1 and 10 percent workload and replication factor all.

that the sever time, processing time , queue time slightly increases as more servers are added in the 10 percent workload scenario. This strengthens the theory from the previous experiment that writer and reader threads have to compete agianst each other for resources. In the the 10 percent workload, we have more load on the resource intensive writer threads, which, therefore,

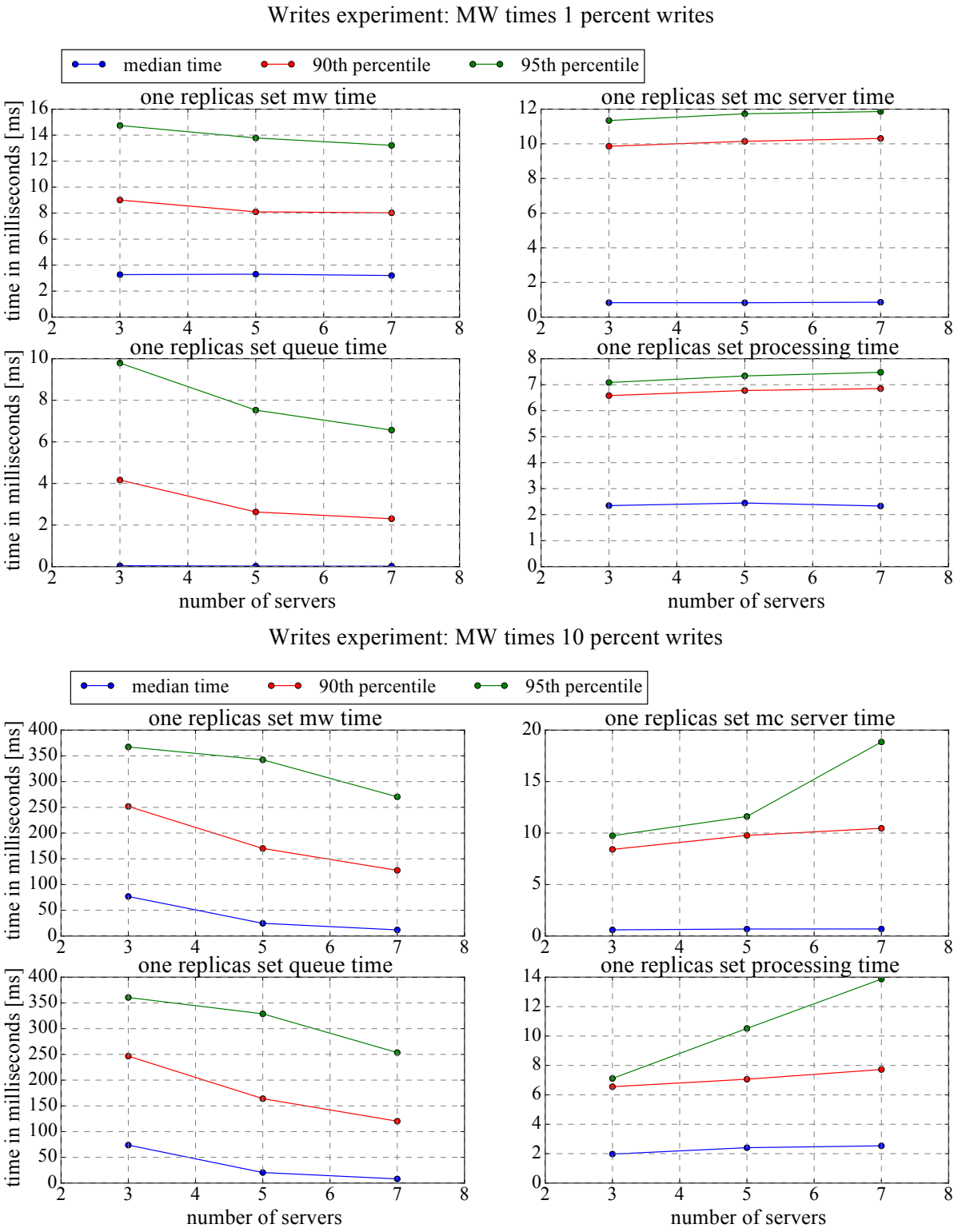

Figure 14: Depicts the middleware set time breakdown for a 1 and 10 percent workload and replication factor all.

use more resources. As a result, the reader threads have less, and their time grows. Therefore, reader and writer thread disturb each other.

In the middleware response time plot for set requests (see Figure 11), we can observe that the higher the write load the higher the mw response time for a write request. Furthermore, we

observe a similar behavior as in the replication experiment that the mw response time decreases as the number of servers grows. This holds for the replicated and non replicated case. If we look at the detailed breakdowns in Figure 13 and 14, which depicts the middleware breakdown times for 1 and 10 percent workloads in terms of set, we can observe that with a higher write load the write queues start to be more populated. As a result, the queue times grow for each write request. Therefore, we see an increase of the middlware response time as the write percentage workload increases. The queue time in the no replica 10 percent case goes up to 80ms. Since one more writer queue and thread are added when one memcached server is added, we also see a decrease in queue time for a set request. In the 10 percent case, the time is reduced for 100ms with 3 servers to 40ms with 7 servers, which is a huge improvement and results in a lower middlware response time.

However, for the all replica 10 percent write case the improvement from 3 to 5 servers is not enough for compensating the induced synchronization and scheduling cost and the middleware response time increases for this specific case.

Summarized we saw in the experiment what we expected to happen. The more expensive writes in the system the more the performance suffer. This becomes even more relevant when the writes cost is increased by replicating to multiple servers.

Logfile listing

Short name	**Location** `https://gitlab.inf.ethz.ch`
maxtpexp.zip	`/burlukas/asl-fall16-project/blob/master/data/maxtpexp.zip`
repexp.zip	`/burlukas/asl-fall16-project/blob/master/data/repexp.zip`
writesexp.zip	`/burlukas/asl-fall16-project/blob/master/data/writesexp.zip`