Advanced Systems Lab Report

Autumn Semester 2018

Name: Flavia Cavallaro
Legi: 16-907-503

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

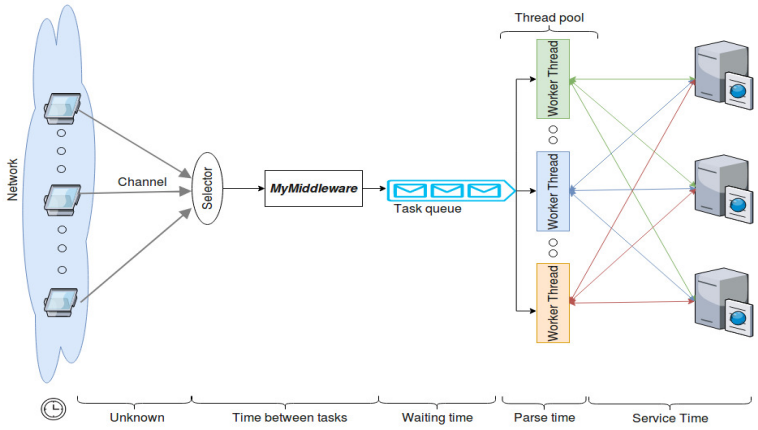


Figure 1: Implementation

1 System Overview (75 pts)

1.1 Implementation

My system is based on 5 classes.

- MyMiddleware
  This is the class that starts the system. Based on a single-thread architecture, it is the interface between the external network and my system, as we can see in figure 1, it uses a selector to switch between multiple connections in an efficient way. After having initialized the thread pool and the shut down hook, it will register the selector with its own server-socket-channel. The selector will then be used to select channels from the network.
  As soon as a channel is selected there are two possibilities:
  - The channel is "Acceptable"
    This happens the first time the middleware selects the channel. In this case my system will, first, accept the connection, second, register the new channel with the selector and third, associate the channel with a new Task object. Therefore, for every channel, a new Task object is created and assigned to the channel, this task object will then be reused for the different requests.
  - The channel is "Readable"
    After a channel is accepted it will start sending data. This data is read, without making distinctions between the different requests, and will be immediately put in to the queue. However, it may happen that the clients sends data separately therefore at the first read the requests is not completed. If this is the case, the channel will be put in a map, and the partial request will be associated to it. When the selector will iterate and select that channel again, it will try complete the request.
- Task
  This class is used to represent the request objects. Every client's channel is associated with a task that is reused for every request. A task can be of two types: "set" and "get". The multi-get is incorporated in the get request.
- ThreadPool
  The ThreadPool class is used to create all the worker threads and, most of all, to collect

the statistics. Statistics are kept in memory as maps. In fact, for every needed record(e.g. service time), the map collects for each worker thread a list of values, one value every second (in my case the window's size is one second). Once a value is collected it will be added to the above mentioned list. At the end, when the middleware is shut down, all the statistics are printed and averages are calculated. Since experiments are not too long, I believe is not too expensive to save in memory all the informations instead of logging it straightway.
Last but not least, ThreadPool provides a method to return the server that has to be queried. This method is called by the worker threads to equally distribute the load in the memcached servers.

- CollectStatistic
  CollectStatistis is a class used to instantiate the object used by the timer for collecting statistics.

- WorkerThread
  A given number of worker threads are created by the thread pool when the system is initialized. Each worker thread collects temporary statistics for the tasks processed in the one-second window. Each worker thread retrieves, if available, a task from the blocking queue, it will then call a method "build_task" to appropriately recognize the type of request. Based on the type it will process the request:

  - Set request
    A set request needs to be sent to every memcached server. Without any further processing, the request will be forwarded, and the thread will wait for the responses from all the servers.
  - Get request
    The differences in processing get requests relies only on the mode enabled, if it is "sharded" or "non sharded". In fact, in case of a single key, the non-sharded mode will be used. The non sharded method is adapted to correctly process both single-key and multiple-key requests.
    * Non sharded
      This mode will first ask the thread pool which server has to be queried, then it will send all the data to this server. Key-value pairs are collected in maps, that will then be used to send back the data to the client. The choice of the server to query is based on a simple round-robin implementation. The index of the servers is incremented each time a worker thread asks the pool which server has to be queried. When it exceeds the number of servers it is set back to zero.
    * Sharded
      After having queried all the servers, it will iterate on them to collect the queried data. These data will be saved in a map, that will then be used to retrieve all the values and send those in the same order as requested.

  In the end the Worker Thread will send back the response through the client's socket channel.

1.2 Instrumentation

All the statistics, except for the queue length, are collected in the ThreadPool as maps. For every needed measurement (e.g. service time, waiting time..) a map is used to collect data

for all the threads. For every map, the key is the thread's reference and the value is a list of measures, a measure is added to the list every window's time. A scheduled method calls several ThreadPool's methods, one for each measure, that collect the data from all the WorkerThreads. Every WorkerThread has an average attribute for every needed measure that keeps updating as it processes new tasks. When the middleware is shut down, all the statistics per-thread are printed. Therefore, the aggregated statistics are calculated and printed as well. As for the queue length, the only thing needed is a list of values that is updated every window's size.

1.2.1 Requested

- Average throughput
  The throughput is collected in each WorkerThread by simply counting the number of requests. The total number of requests always coincides with the number printed out by the memtier clients, although the average may be slightly different because I skipped the first element of the list when calculating it. At the end, the aggregated throughput per second is calculated by summing the throughput per second of all the worker threads.

- Average queue length
  The queue length is collected by the ThreadPool and the values are printed at the end in the aggregated statistics.

- Average waiting time
  The waiting time's start is set by the class "MyMiddleware" when the task is put into the queue. The end time is set by the worker thread when the task is dequeued. Each Worker thread keeps a local average of waiting times that is reset after being collected by the ThreadPool.

- Average service time
  The service time is measured in each worker thread using a map. The map is used to save the service time of each memcached server involved in the request. In fact, in case of set or multi-get requests, all the servers are queried. The WorkerThread will save the server's start time just before writing into the socket and it will then evaluate a service time (by subtracting the server's start time) once it reads a response. Every service time will then be used to update the local average of service times. This local average will be reset after being collected.
  This measure is, in my opinion, one of the less accurate. First of all, for the fact that is measured in the middleware, it is subject to uncertainty due to the network transmission latency that cannot really be evaluated. Secondly, some parsing time is considered as being part of the service time due to the implementation. For example, in case of set operations I first have to send data to all the servers, and then I can start reading for responses. This leads to the fact that the service time takes into account the time spent to write the request in every other socket and, in the same way, the time spent to read from every other socket. A similar behavior happens for multi-get requests.

- Sets, Gets, Multi-gets
  These measures are collected by the WorkerThread.

- Misses
  Measured in the worker thread, is a measure of the keys that were not retrieved.

1.2.2 My own measures

In order to have a better understanding of my system, and a more accurate prediction, I also measured other times that I thought could be reasonable.

- Parsing Time
  Measured in the worker thread is the time taken to parse the request, it does not take into account the service time.

1.3 Path of a task object

To sum up, I am gonna briefly describe the task's object path.

1. The channel will be selected by my middleware. The first line of the request will be read. Basing on the textline, the task's type will be set as "set" or "get" or "discard". In case of set and get the task will be put into the blocking queue.

2. The task waits in the queue until a worker thread pull it. Based on the type, and on the sharded/non-sharded mode enabled it will be processed.

3. The request will be sent to the server/servers.

4. The server/servers sends the response back to the worker thread. If it was the case of a multi-get, the responses are temporary saved in a map, then are ordered. Finally, the response is sent to the client by the worker thread.

2 Baseline without Middleware (75 pts)

2.1 One Server

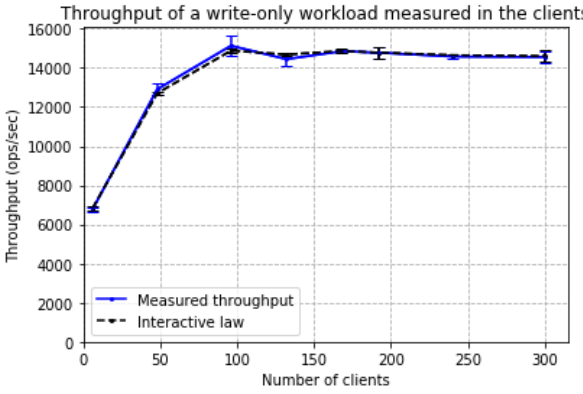
The configuration used in this experiment included all the clients ("Client 1", "Client 2" and "Client 3"), and the server denoted as "Server 1". From now on, all the experiments for all the sections were run using the private IPs. The number of virtual clients per threads used in this experiment is [1, 8, 16, 22, 28, 32, 40, 50].

2.1.1 Explanation: Write-only workload

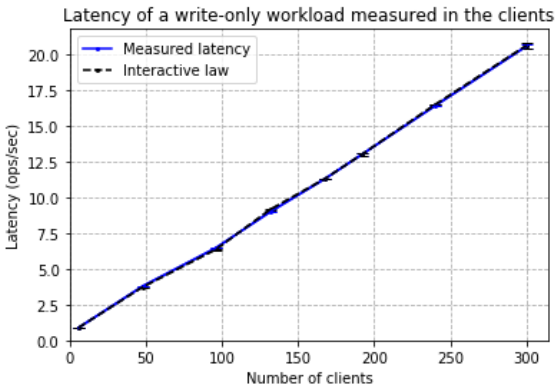
The graph in figure 2 shows the aggregated throughput and latencies measured in the clients. The graph starts with 6 client because each "Client Machine" (3 in total) executes one instance of memtier with two threads and 1 virtual client each.

The data were aggregated as follow: For each "Client Machine", given the same number of virtual clients, three files were printed as output, one for each try. These files contained the measurements for throughput and latencies. In order to have the aggregated measurements, given each try and the number of virtual clients, the throughputs of the three different clients ("Client 1", "Client 2" and "Client 3") were sum together. These measures were then averaged upon the different tries. This produced the average/std total throughput for each number of virtual clients per threads(x-axis on the graphs). As for the latencies, a different approach was used. In fact, a weighted average was calculated based on the throughput of each client with respect to the total. In both figures the standard deviation is very small, this show a great stability of the system.

Moreover, the graph in figure 2 shows also the interactive law. The ideal throughput shown in 2a is calculated as the inverse of the measured latencies (in seconds) multiplied by the number



(a) Throughput experiment 2.1 write-only.



(b) Latencies experiment 2.1 write-only.

Figure 2: Write-only experiment 2.1 .

of clients. Indeed, the ideal latencies shown in 2b were calculated as the inverse of the measured throughput multiplied by the number of clients. Comparing the interactive laws with the measures we see that the two lines almost coincide, which means that the latencies/throughput measured by memtier were very accurate and that no errors were made.

What we can see from this graph is that the throughput tends to grow enormously until 150 clients, then it stabilizes.

Therefore, I would define the saturated phase of the server between 150 and 300 clients. After this range, the memcached is not able to successfully reply to the clients, therefore I stopped at 50 virtual clients per thread.

As we can see from the latencies graph, figure 2b, the latency grows linearly with the same coefficient, this is an obvious explanation from the fact that the throughput saturates, therefore, from the inverse of the interactive law, the response time is directly proportional to the number of clients.

As a general remark, I want the reader to notice that the error bars are plotted in the graphs as well, but they are just too small to be noticed. This indicates a good stability of the system.

2.1.2 Explanation: Read-only workload

The configuration used in this experiment is the same as before and the experiment was run in the same session.

The graphs in figure 3 show the aggregated measurements in all the clients for the "Read-only" workload.

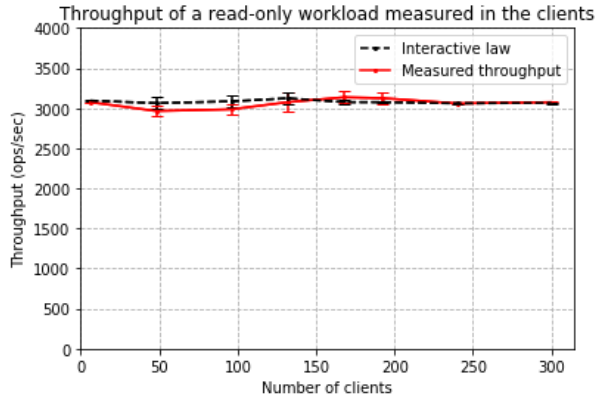
The data were aggregated in the same way as for the "Write-only" experiments.

The throughput graph in figure 3a shows an interesting and unexpected behavior. In fact, no matter the number of clients, the throughput measurements are are always around 3000, therefore the server saturates immediately.

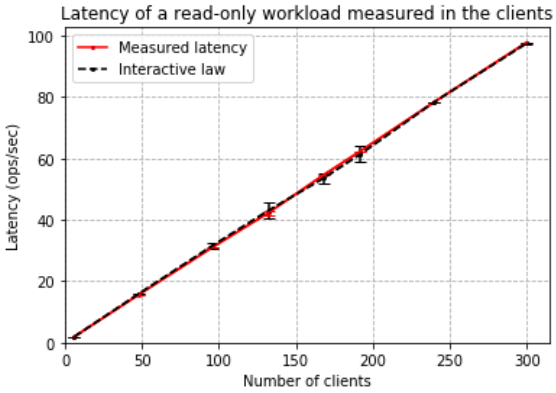
Even in this case, the interactive laws almost coincide with the measurements, which means that the measurements made by memtier were very accurate and that no errors were made. The error bar are very small in this case as well, which means good stability of the system.

2.1.3 Comparison of Write-only and Read-only workload

In figure 4 we can finally compare the two workloads. As we can see, the number of get requests executed per second is much smaller than the number of sets. And in fact, as we can see in



(a) Throughput experiment 2.1 read-only.

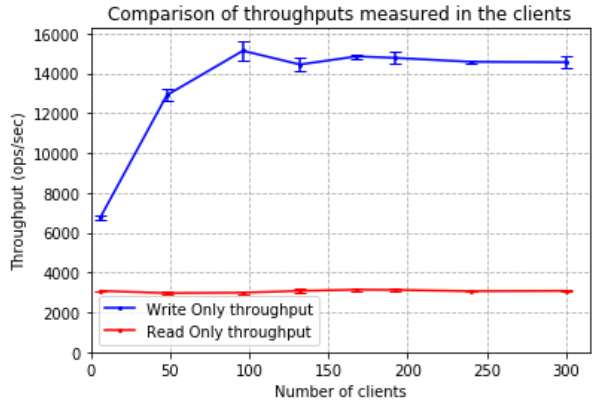


(b) Latencies experiment 2.1 read-only.

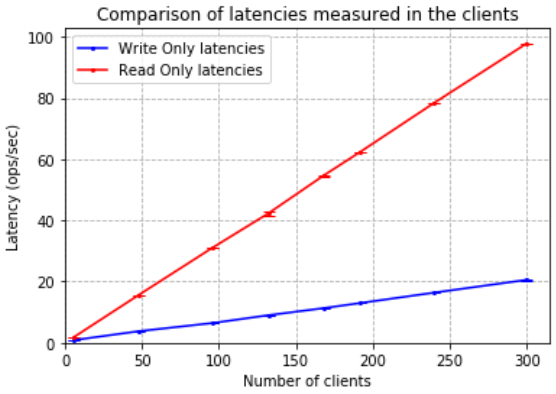
Figure 3: Read-only experiment 2.1 .

figure 4b, the measured response times for the write only workload are smaller than the one for the read only workload, given the same number of clients.

The reason behind these measurements will be given in the following summary, that will allow us to compare different configurations therefore to make stronger conclusions



(a) Throughput experiment 2.1.



(b) Latencies experiment 2.1.

Figure 4: Comparison workload of experiment 2.1 .

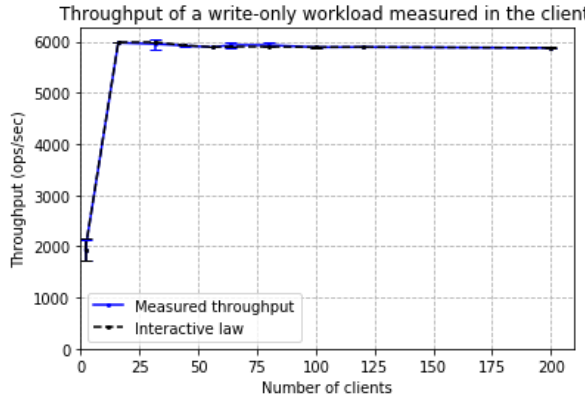
2.2 Two Servers

The configuration used in this experiment included "Client 1" and the servers denoted as "Server 1" and "Server 2". The number of virtual clients per threads used are [1, 8, 16, 22, 28, 32, 40, 50, 60, 100]. Which will be multiplied by 2 in the graphs because we only use one client machines with two instances of one thread.

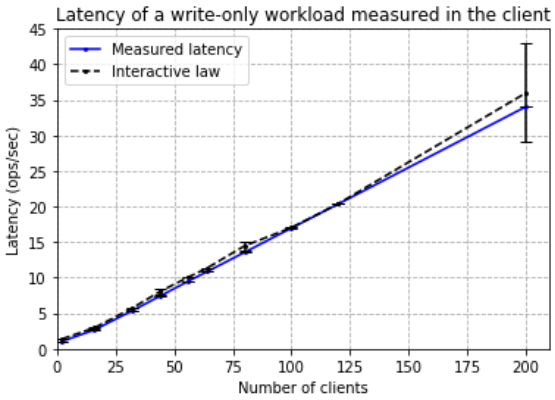
2.2.1 Explanation: Write-only workload

The graph in figure 5a shows the aggregated throughput measured in the two instances of memtier in the client machine.

The graph start with 2 client because "Client 1" executes two instances of memtier with one thread and one virtual client each.



(a) Throughput experiment 2.2 write-only.



(b) Latencies experiment 2.2 write-only.

Figure 5: Write-only experiment 2.1 .

The data were aggregated as follow: For each instance of "Client 1", given the same number of virtual clients (x-axis), one file of measurements was printed as output, one for each try. This file contains the measurements for the throughput and the ones for the latencies. In order to have the aggregated measurements, given each try and the number of virtual clients, the throughputs of the two different instances were sum together. These measures were then averaged upon the different tries. This produced the average/std total throughput of the only client, "Client 1". As for the latencies, the weighted average was calculated based on the throughput of each instance with respect to the total.

The graphs in figure 5 also shows the interactive laws. Even in this case the two lines perfectly match which means that the latencies measured by memtier were very accurate and that no errors were made.

What we can see from this graph is that the throughput saturates immediately, in fact, after 2 virtual clients (the first point in the graph), it reaches immediately the saturation phase. However, in contrast to what we found previously, where the throughput saturated around 15000 requests, here only 6000 requests per second are executed. Even in this case the error bars are very small because the system is very stable.

2.2.2 Explanation: Read-only workload

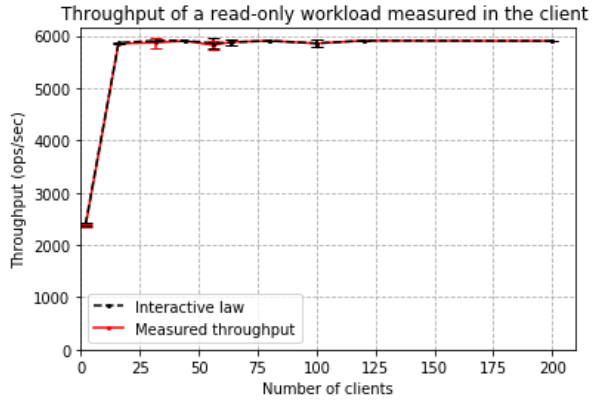
The configuration used in this experiment is the same as before, also the aggregated statistics were calculated in the same way.

The graph in figure 6 shows both the throughput and latencies for the read-only workload and the corresponding interactive laws.

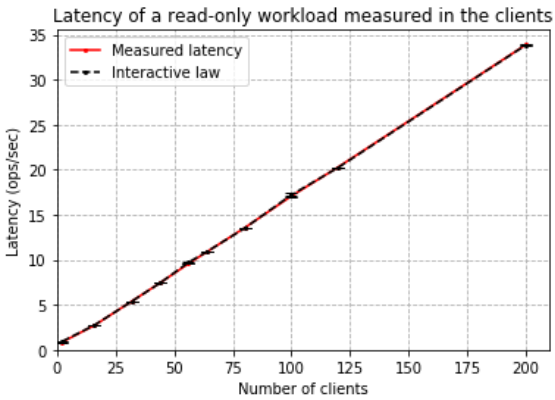
Even for the read-only workload, the saturation phase comes immediately after 2 clients. The saturation point is around 6000 requests. This can be easily explained considering that the previous experiment with one server had a saturation point of 3000 requests, therefore, it is clear why in this configuration (two servers) the throughput saturates around 6000 requests. No sign of over-saturation are shown.

2.2.3 Comparison of Write-only and Read-only workload

In figure 7 we can finally see the comparison between the two workloads. As we can see, the number of requests for the write-only workload is the same.

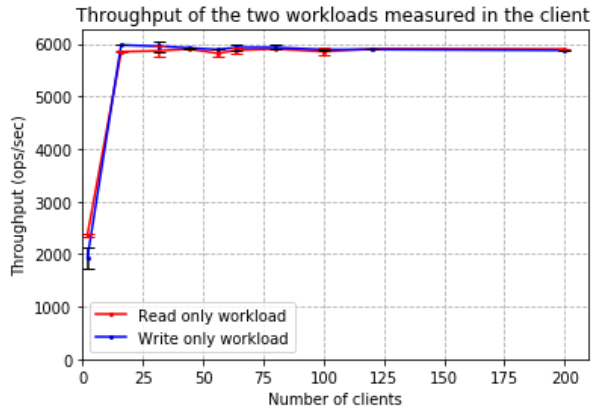


(a) Throughput experiment 2.2 write-only.

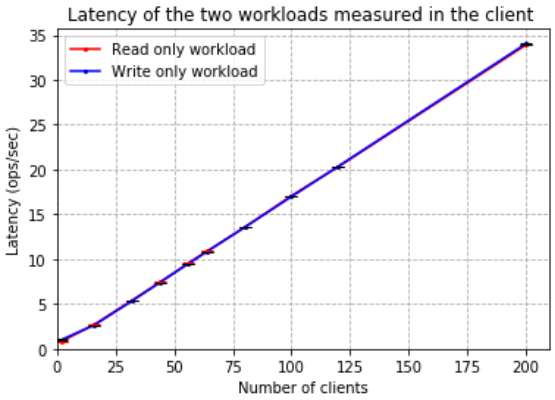


(b) Latencies experiment 2.2 write-only.

Figure 6: Write-only experiment 2.2 .



(a) Throughput experiment 2.2.



(b) Latencies experiment 2.2.

Figure 7: Comparison workload of experiment 2.2 .

2.3 Summary

Based on the experiments above, fill out the following table:

Maximum throughput of different VMs.

	Read-only workload	Write-only workload	Configuration gives max. throughput
One memcached server	3135.95	15133.52	Write-only
One load generating VM	5906.68	5979.05	Write-only

The above table was filled by taking the maximum values of the average throughput. For the Write-only workload, in the configuration with one server, the throughput associated with 16 virtual clients (96 in the x-axis of the graph 4) per threads was taken. For the Read-only workload, in the configuration with one server, the throughput associated with 28 virtual clients (168 in the x-axis of the graph 4) per threads was taken.

For the Write-only workload, in the configuration with one load generating VM, the throughput associated with 8 virtual clients (16 in the x-axis of the graph 7) per threads was taken. For the Read-only workload, in the configuration with with one load generating VM, the throughput associated with 60 virtual clients (120 in the x-axis of the graph 4) per threads was taken.

The graph 8 shows the throughput and latencies for the two modalities (one-server and two-servers) for the two workloads. Unfortunately the experiment 2.2 was run for a smaller number of clients in comparison to 2.1, because the memcached could not handle it, Therefore the graph, figure 8 was cropped to 600 clients.

First of all, comparing the write-only workload we see that two different behaviors are measured for the one and two servers configuration. In fact, in the one server configuration the throughput reaches 15000 requests, while for the two server configuration it reaches 6000 requests. This is due to the fact that in the one-server config. 3 clients machines are used, while in the two-servers config. 1 client machine is used. Therefore, what we can conclude is that in the experiment 2.1 the bottleneck is the server while in 2.2 is the client. In fact, from 2.1 we see that a server is able to reply to 15000 requests but in 2.2 it only replies to 6000 requests, this is because the client does not ask for more. At the same time, 2.1 reaches a maximum throughput of 15000 requests, however said that a client can send 6000 requests, we would expect to see 18000, but we do not measure this throughput, therefore the server is the bottleneck.

As for the read-only workload, we see that the one server configuration (exp 2.1) reaches a maximum throughput of 3000 requests, while the two servers configuration (exp 2.2) reaches 6000 requests. From exp 2.2 we can safely deduce that a client can receive for sure 6000 requests, however, in exp 2.1, 3 clients in total received only 3000 requests. Therefore the problem lies on the server that is not able to send more than 3000 requests. At the same time, when increasing the number of servers, exp 2.2, the throughput increases as well, but it is still bounded by the two servers, that now guarantee double the throughput.

To sum up, a single server is not able to reply to more than 15000 set requests per second and 3000 get requests per second.

(a) Throughput for experiments 2.1 and 2.2.

(b) Latencies for experiments 2.1 and 2.2.

Figure 8: Comparison workloads for experiments 2.1 and 2.2 .

Last but not least, if we compare the latencies of the write-only cases (the two-server write-only line coincides with the two-servers read-only line) we have a further proof of the fact that the two server configuration gives the same throughput, while the one-server is faster with the write-only workload (which means that produces a greater output).

3 Baseline with Middleware (90 pts)

3.1 One Middleware

This experiment was run using "Client 1", "Client 2", "Client 3" as clients, "Middleware1" was used for the middleware and "Server 1" was used for the server. In all the experiments, for each

memtier client, the number of connections per thread were: 1, 8, 16, 22, 28, 32. I tried to run experiments with a bigger number of connections per thread, however due to unknown reasons connection error were raised, therefore I stopped at 32 clients.

3.1.1 Explanation: Write-Only workload

The graph in figure 9 shows the measurements recorded in the middleware for the write only experiment, each line represents a different configuration of the middleware. The different configuration involves the number of worker threads used in the middleware, the worker threads are those threads that interact with the server.

Looking at figure 9a, two things can be said. First of all, the number of requests executed per second, when increasing the number of threads, increases as well. Second, it takes more time to reach the saturation point, as the number of threads increases. This is obviously due to the fact that more threads can handle more requests.

Looking more in detail at the behavior observed with 8 threads, we see that the saturation phase is reached immediately after 50 clients. Starting from that point the queue length (and consequently the waiting time) starts increasing faster. Consequently, the service time, figure 14 flats immediately. This is due to the fact that 8 threads are not able to execute too many requests, therefore, when the queue starts filling in, it means that the threads have reached their maximum capacity and the server is not exploited more than that.

The 16 threads configuration is similar to the previous one, however, since 16 threads can handle more requests the queue length is smaller and the server are exploited more (higher service times).

In the 32 threads configuration there are small differences, for example the saturation phase starts around 100 clients.

Finally, 64 threads, saturates at the very end of the graph, around 150 clients and is the configuration that guarantees the minimum response time.

Last but not least, as shown from the standard deviation, the configurations with more threads tend to become less stable and in fact bigger standard deviation are calculated.

Looking at the latencies indeed, figure 9b, we see that increasing the number of threads increases the throughput and decreases the response time. The latencies are calculated as the sum of waiting time, service time and parsing time. However, the latencies' graph is not enough to understand the true behavior of the system. In fact, we need to understand what composes the total latency time. It is shown in figures 10 that the response time is always composed by the waiting time and service time. In fact the queue length, figure9c is always different than zero, which means that, no matter the configuration, the threads are not enough to dispose all the requests.

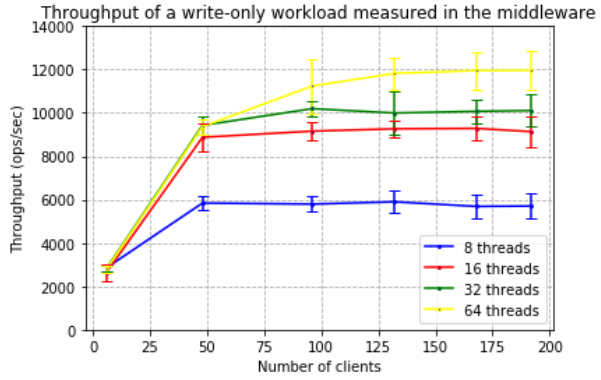
As for the service time, for all the threads configurations, it tends to saturate, this is easily interpretable. In fact, the number of clients that interact with the server is fixed, whether 8, 16, 32 or 64 threads.

Finally, if we compare this experiment with experiment 2.1, considering 64 clients for that experiment, we saw that the throughput reached with that configuration was around 14000. However, in this case, the reached throughput is lower, in fact it is around 12000 requests. This means that the middleware is the bottleneck of the system.

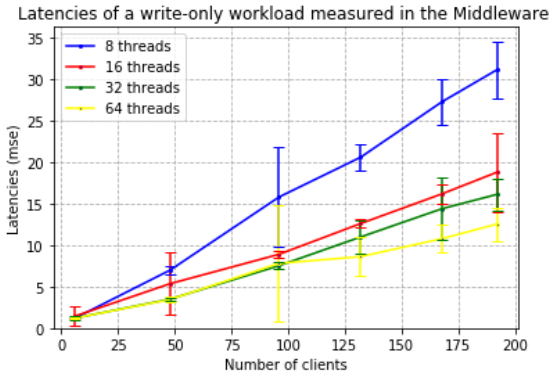
In table **??** the interactive laws are shown. We see that they slightly differ from the true throughput. This is due both to the non-measured additive latencies and from the fact that the std are not irrelevant. From now on, in all the sections, the interactive laws are calculated with the response time, to whom is added the average ping latency of that experiment.

VC	8 Threads		16 Threads		32 Threads		64 Threads	
WO	**Meas.**	**Int. law**	**Meas.**	**Int law.**	**Meas.**	**Int law**	**Meas.**	**Int law**
1	2855.69	2408.92	2650.74	2189.59	2827.38	2402.59	2791.57	2401.82
8	5842.30	5788.92	8875.98	7212.87	9426.52	10012.02	9367.37	10113.21
16	5793.48	5612.83	9159.35	9414.28	10195.12	10875.14	11236.05	10537.29
22	5898.31	6026.22	9266.99	9490.75	9995.72	10740.55	11822.04	13313.43
28	5687.7	5870.53	9286.5	9598.8	10072.65	10696.00	11953.39	13887.35
32	5705.1	5919..42	9134.59	9559.21	10109.50	11022.59	11965.09	13878.78
RO	**Meas.**	**Int. law**	**Meas.**	**Int. law**	**Meas.**	**Int law.**	**Meas.**	**Int law.**
1	2779.68	4034.683	2737.37	3896.49	2764.07	3995.69	2705.26	4853.45
8	2934.34	3229.32	2936.95	3280.15	2935.28	3309.65	2934.10	3314.48
16	2924.86	3110.78	2929.4	3138.30	2924.55	3159.33	2927.30	3156.96
22	2919.98	3073.51	2924.06	3098.17	2923.32	3133.01	2945.09	3157.79
28	2917.23	3147.71	2922.14	3090.68	2913.17	3160.63	2922.1	3183.21
32	2921.83	3114.74	2925.34	3168.38	2910.45	3181.47	2934.77	3193.06

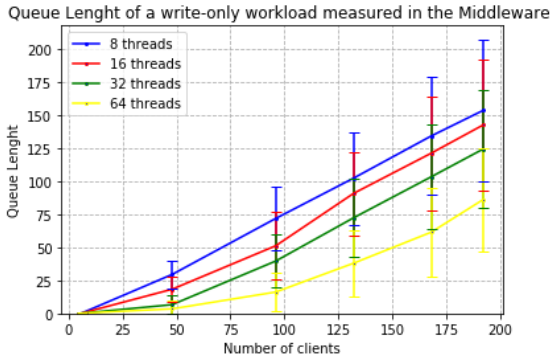
Table 1: Interactive laws for Write-only and Read-only workload of experiment 3.1.



(a) Throughput of write-only workload for experiment 3.1.

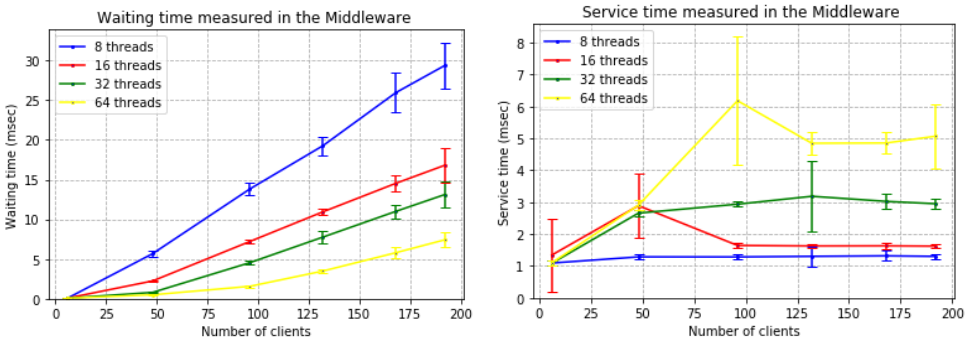


(b) Latencies of write-only workload for experiment 3.1.



(c) Queue lengths of write-only workload for experiment 3.1.

Figure 9: Throughput, latencies and queue lenght measured in the Middleware.



(a) Waiting time of write-only workload for experiment 3.1.

(b) Service time of write-only workload for experiment 3.1.

Figure 10: Waiting time and service time measured in the Middleware.

3.1.2 Explanation: Read-Only workload

Looking now at the read-only graphs, figure **??**, we see that the throughput, figure 11a, is the same no matter the configuration used. Moreover, we see that it reaches almost 3000 request which is the same number we found in experiment 2.1 when using one server. The measured latencies are the same, while the queue length decreases, this is due to the fact that the service time increases and therefore balances the decreasing waiting time. The interactive laws of the read-only are shown in table **??**. Same conclusion as before can be made.

In this case we see that the throughput converges toward 3000 requests, which is exactly what we observed in experiment 2.1, therefore, from a first analysis we can conclude that in this case the bottleneck is the server.

3.1.3 Comparison of Write-only and Read-only workload

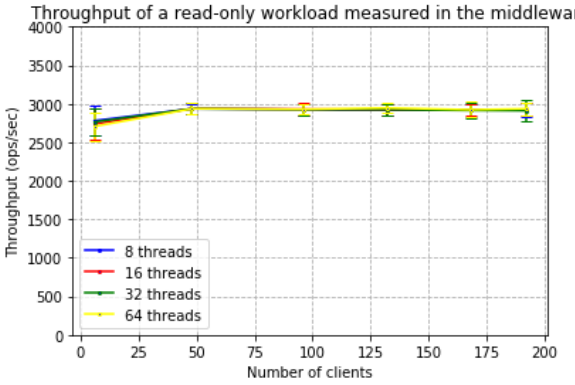
Comparing the throughput graphs of the two workloads, 9a and 11a, we see that, as we expected, the two throughputs are very different. As a proof, we see that the latencies associated with it are smaller in the case of the write-only case. The service time is around 3 times higher in the case of the read-only workload. The error bars are unfortunately not too small, for what I have seen, this was due to the fact that I used a warming up time too small, in the following experiments I increased it and I measured smaller std.

3.2 Two Middlewares

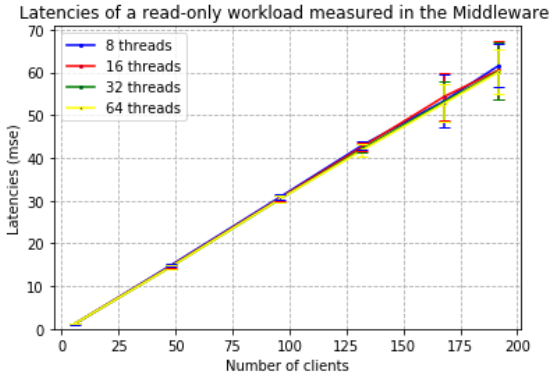
This experiment was run using all the Clients machines, both Middlewares and "Server 1" as server. It was run with a different allocation respect to the previous experiment, however small changes in the latencies were measured. The private IPs were used.

3.2.1 Explanation: Write-Only workload

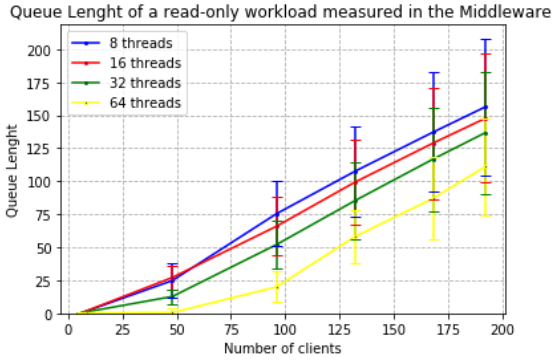
First of all, in order to understand the data in figure **??** we have to understand how these data were calculated. For the throughput and the queue length, the data inside each middleware was aggregated as explained in the previous section. After that, fixed the number of clients used in the experiment, both the average and the std measures of the two middlewares were sum up together. While for the latencies, a weighted average was used starting from the aggregated latencies of each middleware.



(a) Throughput of read-only workload for experiment 3.1.

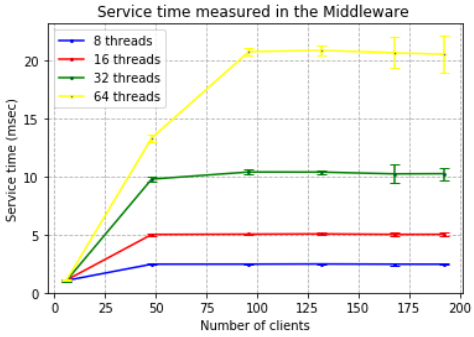


(b) Latencies of read-only workload for experiment 3.1.



(c) Queue lengths of read-only workload for experiment 3.1.

(a) Waiting time of read-only workload for experiment 3.1.



(b) Service time of read-only workload for experiment 3.1.

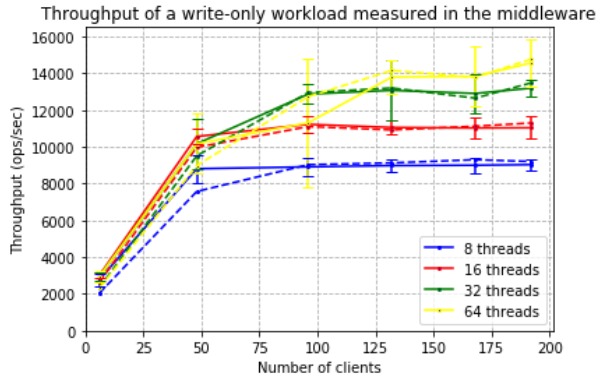
Figure 12: Waiting time and service time measured in the Middleware.

Looking at figure 12, we have a confirm of the fact that increasing the number of threads helps increasing the throughput and decreasing queue lengths and latencies.

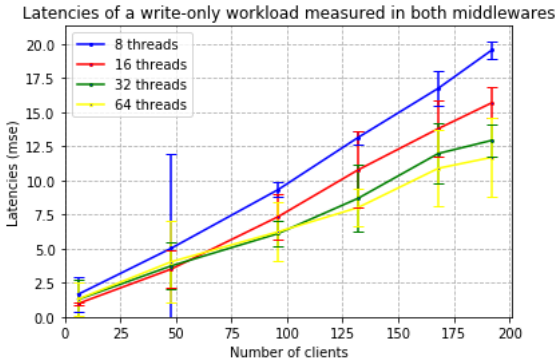
Looking at the behavior observed with 8 threads, we see that the saturation phase is reached immediately after 50 clients. Starting from that point the queue length (and consequently the waiting time) starts increasing faster. Consequently, the service time, figure 14 flats immediately, this is due to the fact that 8 threads are not able to execute too many requests therefore when the queue starts filling, around 50 clients, it means that the threads have reached their maximum capacity therefore the server is not exploited more than that. Considering now 16 threads, the

saturation phase is reached between 50 and 75 clients, as we would expect, 16 threads can handle more requests therefore the throughput is bigger. Same things can be said for 32 threads, where the saturation point is reached around 100 clients. The behavior for 64 threads was uncertain. I tried to run more experiments but error of "connection reset" were raised therefore I couldn't try more. In order to have better conclusions, I estimated the utilization ratios for 64 WTs (and 64 Virtual Clients) and I found out that the maximum value reached is 0.98 which means that the WTs are exploited at the maximum therefore also in this case, the bottleneck is the middleware.

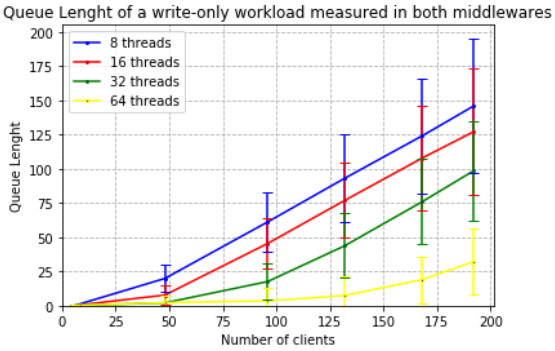
In this case, the interactive laws are shown as dashed lines in the throughput graph, since they do not create too much confusion I decided to plot them in the graph instead of the table.



(a) Throughput of write-only workload for experiment 3.2.



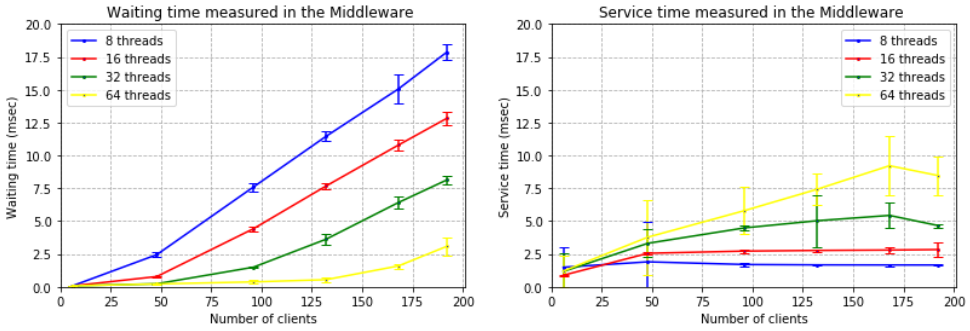
(b) Latencies of write-only workload for experiment 3.2.



(c) Queue lengths of write-only workload for experiment 3.2.

3.2.2 Explanation: Read-Only workload

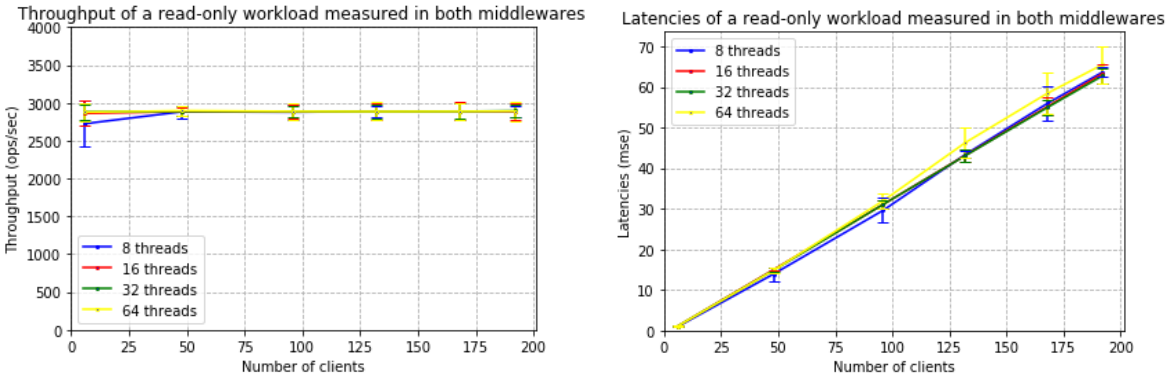
The same aggregation was done for the read-only workload. What we see is that the throughput converges towards 3000 requests, as we would expect. In this case, increasing the number of middleware has no effect in terms of throughput of response time. The only thing that changes is that now the response time is composed more by the service time rather than waiting time, as observed in figures 12 and 16. This is easily explained observing the fact that the total requests are now split between two middlewares and that the server is therefore exploited more (two times the number of threads of each middleware). As a consequence it increase the service time because now double the threads interact with it.



(a) Waiting time of write-only workload for experiment 3.2.

(b) Service time of write-only workload for experiment 3.1.

Figure 14: Waiting time and service time measured in the Middleware.



(a) Throughput of read-only workload for experiment 3.2.

(b) Latencies of read-only workload for experiment 3.2.

(c) Queue lengths of read-only workload for experiment 3.2.

(a) Waiting time of read-only workload for experiment 3.2.

(b) Service time of read-only workload for experiment 3.2.

Figure 16: Waiting time and service time measured in the Middleware.

3.3 Summary

Maximum throughput for one middleware.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware	2934.10	14.48	0.71	0.02
Reads: Measured on clients	2977.33	16.08	n/a	0.02
Writes: Measured on middleware	11953.39	10.79	5.79	n/a
Writes: Measured on clients	12236.33	13.67	n/a	n/a

Maximum throughput for two middlewares.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware	2896.24	14.82	0.36	0.0
Reads: Measured on clients	2925.16	16.29	n/a	0.0
Writes: Measured on middleware	13800.9	8.01	0.54	n/a
Writes: Measured on clients	14028.87	9.38	n/a	n/a

The previous tables were filled up as follow. Considering the table with one middleware, I used 64 worker threads for both configuration, and 28 Virtual Clients for the write-only, while 8 Virtual Clients for the read-only. Considering now the table with two middlewares, I used 64 worker threads for both configuration, and 22 Virtual Clients for the write-only, while 8 Virtual Clients for the read-only. The values where chosen by taking the values corresponding to the first saturation point. Which guarantee a lower response time.

First of all, in both tables, the throughput measured on clients and middleware differ of small values. This is due to some implementation choices regarding my middleware, in fact, I prefer not to consider some values of throughput at the beginning and start of the experiment. Second, as we can see, also the response times measured on the clients are different than the one measured on the middleware, this is due to the additive latencies that cannot be measured like the time spent before being selected by my net-thread and the latencies due to the network.

Comparing now the two tables, we see that almost no difference there is for the read-only case. However, there is a relevant difference when considering the write-only case. In fact, two middlewares have the effect not only of increasing the maximum throughput from 12000 to

14000, but also to decrease the response time corresponding to that values. Of course, we are considering different configurations for the virtual clients, but what we can do is have a look at the two latencies graph 9b and 13b, here we see that the two middlewares configuration has values of latencies similar to that of one middleware, however, as we found out, it guarantees a bigger throughput.

In both experiments we see that the bottleneck for the read only is the server, which cannot guarantee more than 3000 requests. As for the write-only workload, as we saw in experiment 2.1, one server is able to guarantee around 15000 requests, but here my middleware never reaches that value, therefore it is the bottleneck.

4 Throughput for Writes (90 pts)

4.1 Full System

The experiment 4.1 was run with all the machines, unfortunately with a different allocation than the previous experiments however no significant difference in the ping latencies was measured. The private IPs were used. The virtual clients used are [1, 8, 16, 22, 28, 32].

4.1.1 Explanation: Write-Only workload

The data were aggregated in the same way of experiment 3.2. Even in this case the legend represents the number of worker threads per Middleware.

As we can immediately see in figure 17a, the throughput grows with the number of threads. The saturation point is clearly reached and it is around respectively 6000, 9000, 11000 and 13000 requests for respectively 8, 16, 32, 64 worker threads. What we notice is that changing the number of threads influence the number of clients needed to reach the saturation point. The configuration with 8 threads reaches immediately the saturation point, after 50 clients and in fact we see that the queue length starts increasing more rapidly as well as the waiting time while the service time remains still, figure 18. In this case the total response time is mainly composed by the waiting time.

As for 16 threads, the saturation point is also reached around 50 clients, in fact, starting from that point, the queue length increases rapidly and consequently the waiting time. Even in this case, the total time is composed for the majority by the waiting time. However the waiting time is smaller than the previous case while the service time bigger.

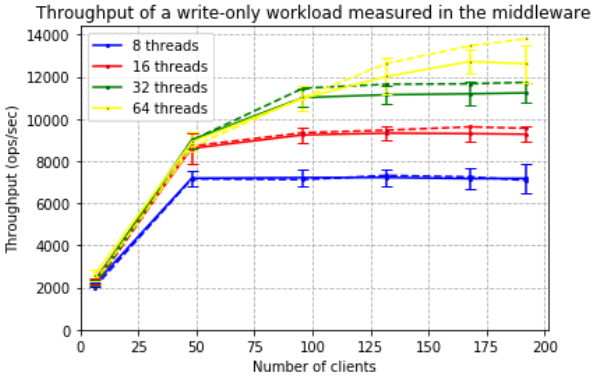
A different saturation point is observed with 32 threads, in fact it saturates after 100 clients. Same things as before can be noticed for waiting and service time.

The most important configuration, 64 threads, which reaches the highest throughput, is able to almost set to zero the waiting time and exploit the most the servers, in fact the service time is now big and higher than the waiting time.

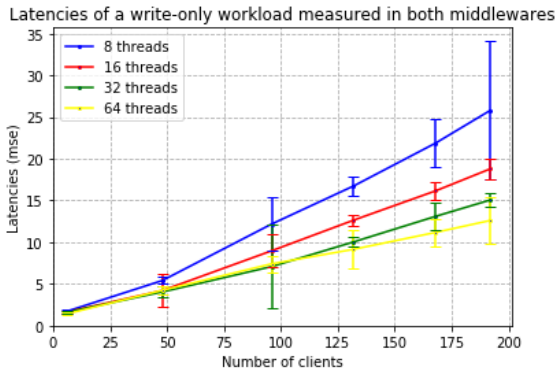
To sum up, increasing the worker threads has a positive effect in increasing the throughput. It also helps diminishing the total response time.

In graph 17a, the dashed lines are the interactive laws for the corresponding throughput (same color). These laws were calculated by adding the average ping latencies to the response time. We can see that these lines almost coincide with the measurements therefore no errors were made.

Last, the std are smaller, probably due to the fact that I let the middlewares warm up for more time.



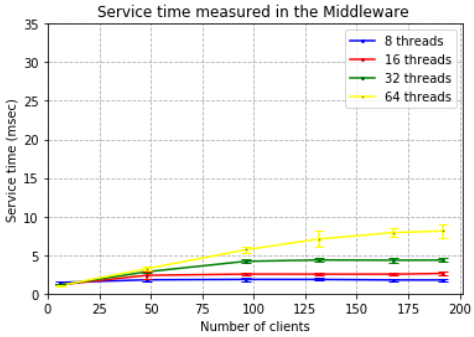
(a) Throughput of write-only workload for experiment 4.1.



(b) Latencies of write-only workload for experiment 4.1.

(c) Queue lengths of write-only workload for experiment 4.1.

(a) Waiting time of read-only workload for experiment 4.1.



(b) Service time of read-only workload for experiment 4.1.

Figure 18: Waiting time and service time measured in the Middleware.

4.2 Summary

Maximum throughput for the full system

	WT=8	WT=16	WT=32	WT=64
Throughput (Middleware)	7219.87	9322.17	11232.37	12725.80
Throughput (Derived from MW response time)	7882.69	10436.97	12748.34	15032.87
Throughput (Client)	7332.87	9522.83	11532.4	12870.36
Average time in queue	14.59	9.27	9.52	2.02
Average length of queue	94.31	76.47	98.00	23.76
Average time waiting for memcached	1.83	2.52	4.34	7.91

The maximum throughput table was filled by taking the maximum value observed. First of all, for the throughput measured in the middleware:

- 8 worker threads: point of the saturation phase measured with 22 virtual clients per threads, which means 132 total clients. The corresponding response time measured in the middleware was 16.74 msec.
- 16 worker threads: point of the saturation phase measured with 22 virtual clients per threads, which means 132 total clients. The corresponding response time measured in the middleware was 12.64 msec.
- 32 worker threads: point of the saturation phase measured with 32 virtual clients per threads, which means 192 total clients. The corresponding response time measured in the middleware was 15.06 msec.
- 64 worker threads: point of the saturation phase measured with 28 virtual clients per threads, which means 168 total clients. The corresponding response time measured in the middleware was 11.17 msec.

The throughput derived from the middleware was calculated as (number of clients / response time calculated in the middleware). The throughput measured in the client was calculated by first summing the throughputs of the two instances for each client, then summing the th. of all the clients and then averaging for all the tries.

The queue length, service time and queue time was taken, in each case (8/16/32/64 threads) for the corresponding number of clients associated to the maximum throughput.

For all the threads we see that the three throughputs (respectively from MW, Clients and RT) are similar but not the same. This is explained by the fact that the second throughput is calculated with a response time that however misses some elements like think time or network latencies. While the difference between the one measured in the client and the one in the middleware is due to some implementation choices, like discarding the first and last few seconds statistics. Nevertheless we notice that for all the threads this difference is usually very small as we expect. While the difference between first and second row are higher due to the fact that a small change in latency strongly affect the throughput. Therefore it is more prone to errors.

What we see in this table is that the 64 threads configuration guarantees the maximum throughput across all the configurations, moreover it guarantees smallest queue time. As for the time in queue, and the corresponding waiting time, we see that increasing the threads has the positive effect of decreasing these values, because more workers take part in the process. However, the server, as the threads grow tends to be slower, and in fact the waiting time increases with the number of worker threads.

A last important comparison needs now to be done. Comparing this experiment with 3.2, we see that this throughput is slightly smaller, figures 13a and 17a. In exp 3.2, the maximum throughput reached was around 14000 requests, while now it is around 13000.

In order to understand why, a reminder needs to be done. In this experiment, each middleware has to send the set request to all the (three) servers. This makes the middleware loose more time, therefore the middleware processes less requests.

Even the queue lengths are very similar, figures 13c and 17c, however the 4.1 experiment has a queue length slightly bigger. The waiting time, as a consequence of the bigger queue lengths, is bigger, as well as the service time.

5 Gets and Multi-gets (90 pts)

The experiment was run using all the clients, middlewares and servers, however, it was not run on the same session of the previous experiment, however the ping latencies do not differ too much. The configuration used in this experiment is the one with 64 threads, since the read-only case had always the same throughput, I based my decision on the measurements obtained in section 4.1, therefore I choose the 64 threads configuration which is the one that gave maximum throughput.

The percentiles graphs used in this section were aggregated as follow. Fixing a size of multi-get, for each client, I first summed the throughput of the two instances and calculated the weighted average of the latencies. This was done for the 60 seconds values collected in the clients. After that, the weighted average of the previously found latencies was calculated based on the previously found throughputs. This led to weighted latencies for each client with associated throughput. In order to aggregate all the clients, another weighted average was calculated on all the clients.

Analogous strategy was used for the middlewares. In fact, a weighted average of the latencies of the two middlewares was computed.

5.1 Sharded Case

5.1.1 Explanation

In figure 19 we can find the plot of the measurements made in the clients. First of all, as we can see in figure 19a, the response time grows as the size of the multi-get grows. This is obvious to explain, in fact, as the number of keys grow the middleware has to retrieve more keys, which means that each server is loaded more. This can be confirmed by the fact that the service times grow as the multi-key size grows, as can be seen in table 2. Moreover, what can be seen as a good property of the system is the fact that, even if the number of keys to be retrieved triplicates first, the response time does not. In fact, when the key-size is 6, the response time is not the double of the of the one associated with 3 keys and so on. Therefore, it is better to retrieve more keys at the same time. Finally, when calculating the throughput values as: key-size*(num of multi get) + (num of sets), I found out that the highest throughput is given when retrieving 6 keys.

The percentiles figure 19b has in the x-axis the percentiles and in the y-axis the associated response times. The black dot, represents the mean associated with the multi-key size.

Looking now more in detail we see that the response time usually does not grow too much between the 25th and 80th percentile, this means that almost all the requests have a predictable and not too high latency, indeed after that percentile, the response time does grow (it needs to be noticed that the scale does not let us appreciate too much the changes in the 6 and 9 keys configuration however the response times grow as well but less significantly than in the case of 1 and 3 keys). In all the configurations, the mean corresponds to a low-medium percentile, this means that a significant amount of requests have a higher latency, therefore the mean does not really give a truthful overview of the system.

Multi-key size	Response Time avg (std)	Service Time avg (std)	Waiting time avg (std)	Queue length avg (std)	Throughput avg (std)
1	1.45 (0.103)	1.178 (0.105)	0.074 (0.01)	0.256 (0.12)	4921.67 (353.14)
3	1.85 (0.17)	1.278 (0.132)	0.081 (0.01)	0.267(0.02)	4140.85 (420.8)
6	3.03 (0.08)	1.680 (0.093)	0.083 (0.01)	0.213 (0.01)	2833.55 (95.26)
9	4.95 (0.08)	2.175 (0.136)	0.087 (0.01)	0.200 (0.05)	1925.21 (78.34)

Table 2: Table of aggregated measurements of the middlewares for experiment 5.1.

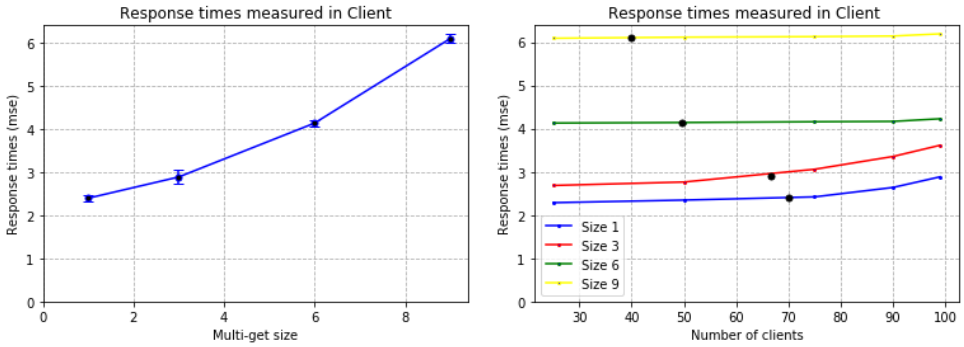
Multi-key size	Response Time avg (std) measured in MW	Response Time avg (std) measured in Client	Throughput avg (std)	Interactive law Thr = #Clients / RespTime(sec) #Clients = 2*2*3 RT in Middle
1	1.45 msec (0.103)	2.39 (0.08)	4921.67 ops/sec (353.14)	5000.75
3	1.85 msec (0.17)	2.89 (0.16)	4140.85 ops/sec (420.8)	4148.51
6	3.03 msec (0.08)	4.14 (0.07)	2833.55 ops/sec (95.26)	2896.21
9	4.95 msec (0.08)	6.1 (0.11)	1925.21 ops/sec (78.34)	1964.15

Table 3: Table of measurements of experiment 5.1.

Looking at table 2, we can conclude that the increment in response time (measured with the increment of the key-size) is due to the service time and no significant changes of waiting time are measured.

If we now look at table 3, we can compare the response times measured in the middleware and the one measured in the clients, and we see that there is a difference of 1 msec on average. This can probably be associated with the additive latencies of the network.

Finally, the interactive law in table 3 confirms the measured data.



(a) Response times for different multi-get sizes (1, 3, 6, 9).

(b) Response times associated with 25th, 50th, 75th, 90th and 99th percentile.

Figure 19: Response times for experiment 5.1. Aggregated for all the clients.

5.2 Non-sharded Case

Run multi-gets with 1, 3, 6 and 9 keys (memtier configuration) with sharding disabled. Plot average response time as measured on the client, as well as the 25th, 50th, 75th, 90th and 99th percentiles.

5.2.1 Explanation

The experiment was run using the same configuration as experiment 5.1.

Multi-key size	Response Time avg (std)	Service Time avg (std)	Waiting time avg (std)	Queue length avg (std)	Throughput avg (std)
1	1.42 (0.10)	1.12 (0.04)	0.076 (0.01)	0.44 (0.02)	4959.06 (293.14)
3	1.65 (0.11)	1.16 (0.09)	0.075 (0.01)	0.43(0.02)	4451.42 (320.8)
6	3.01 (0.06)	1.67 (0.06)	0.075 (0.01)	*0.14 (0.01)*	2865.85 (93.26)
9	4.94 (0.07)	3.75 (0.25)	0.074 (0.01)	0.14 (0.05)	1940.75 (68.34)

Table 4: Table of aggregated measurements of the middlewares for experiment 5.2.

Multi-key size	Response Time avg (std) measured in MW	Response Time avg (std) measured in Client	Throughput avg (std)	Interactive law Thr = #Clients / RespTime(sec) #Clients = 2*2*3 RT in Middlew
1	1.42 msec (0.08)	2.37 (0.08)	4959.06 ops/sec (293.14)	5060.38
3	1.65 msec (0.17)	2.64 (0.09)	4451.42 ops/sec (320.8)	4537.83
6	3.01 msec (0.06)	4.09 (0.07)	2865.85 ops/sec (93.26)	2933.04
9	4.94 msec (0.25)	6.05 (0.11)	1940.75 ops/sec (68.34)	1981.65

Table 5: Table of measurements of experiment 5.2.

In figure 20 we can find the plot of the aggregated measurements made in the clients. First of all, as we can see in figure 20a, even in this case the response time grows as the size of the multi-get grows. This can be confirmed by the fact that the service times grow as the multi-key size grows, as can be seen in table 4. Even in this case, if the number of keys to be retrieved triplicates, the response time does not and so on. Therefore, it is better to retrieve more keys at the same time.

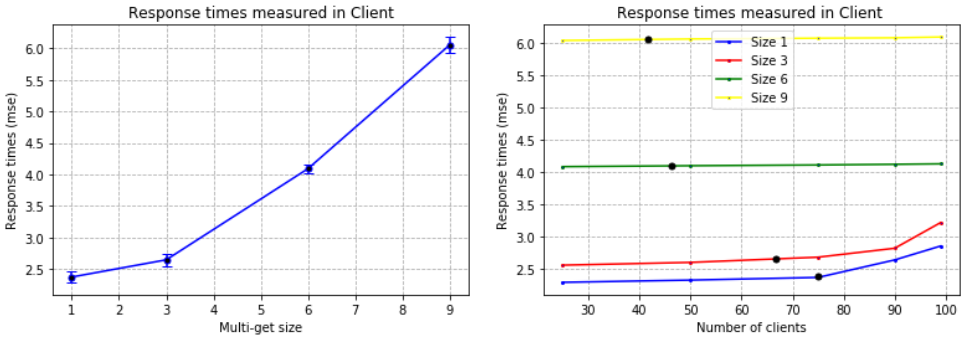
The percentiles figure 20b was drawn as before.

The response times usually do not grow too much between the 25th and 80th percentile, this means that almost all the requests have a predictable and not too high latency, indeed after that percentile, the response time does grow (it needs to be noticed that the scale does not let us appreciate too much the changes in the 6 and 9 keys configuration however the response times grow as well but less significantly than in the case of 1 and 3 keys). Analogous conclusion, to the one in experiment 5.1, can be made regarding the mean. In fact, it corresponds to a low-medium percentile, this means that a significant amount of requests have a higher latency, therefore the mean does not really give a truthful overview of the system.

Even in this case there is a difference of approximately 1 msec between response time measured on the client and on the middleware, figure 4, this can be associated to the additive latencies of the network.

5.3 Histogram

Figures 32 show the histograms of the response times. The vertical red line represents the value of the average. What we can see is that, the bell curve that we see in the clients measurements in experiment 5.1, 21a is more smooth in the Middlewares measurements 21b, this is probably due to the fact that in the middleware I measure smaller times, more prone to noise and, even more important, due to implementation reasons, the measurements made in the middleware are not as accurate as the one obtained by the clients (The middleware has to first send data to all the servers and then it starts listening to responses). Indeed, when looking at the histograms of experiment 5.2 we see that the client's curve is more smooth, figure 21c however it needs to be noticed that the interval that spreads these latencies is smaller than the previous one. Therefore, the latencies are more uniform distributed however they span a smaller interval. Looking at the latencies in the middleware, 21d, we see that these are very similar to the latencies found in



(a) Response times for different multi-get sizes (1, 3, 6, 9).

(b) Response times associated with 25th, 50th, 75th, 90th and 99th percentile.

Figure 20: Response times for experiment 5.2. Aggregated for all the clients.

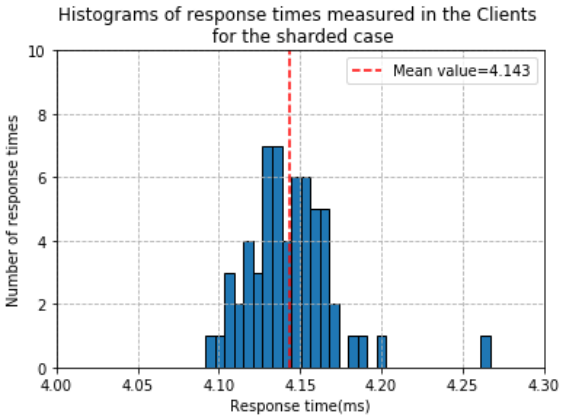
experiment 5.1. This confirms what can be seen in the previous throughput's tables, 3 and 5, that the throughput of the two configurations (sharded and non-sharded) is almost the same. The shift by one of the response time can be observed in the histograms as well. Explanation was given in the previos sections. Finally, by looking at the value of the mean, we see that the mean is a poor representation of what really happens in the middleware. In fact, many responses are slower than the mean value. To conclude, another clarification needs to be done, sometimes unexpected latencies were measured (probably due to network errors), I therefore decided to clean the data by removing such errors.

5.4 Summary

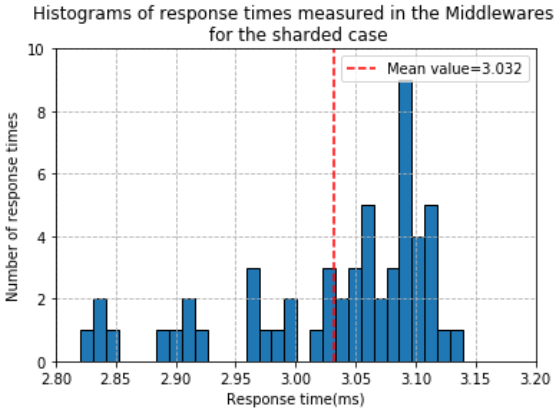
First of all, comparing the sharded and non sharded mode, we saw that the response times associated to the different key sizes, figures 19a and 20a, are almost the same. And in fact, the throughputs are similar. Therefore there is no preferred option. The cause can be found by looking at the parsing times and service times, table 6, in fact we saw that when the size is 9, the service time between the two configurations is different however the total time is not, this is due to the parsing time which is greater in the sharded configuration. This can be easily explained, in fact it takes more time to split and send 9 requests rather than just sending one. As a result, it takes more time for the server to reply to a multi-get of 9 gets rather than to one of 3 (9/3 servers) gets. What stated above holds when the key size is 9, for 1, 3 and 6, no differences between service times and parsing times of the two configurations were measured.

A comparison between the latencies measured in clients and middlewares can be done by looking at the values of the histograms. In fact, given the same experiment (sharded or non sharded), we see that the latencies measured on the clients are usually shifted on the right of around 1 msec. As mentioned before, this can be attributed to the network latencies.

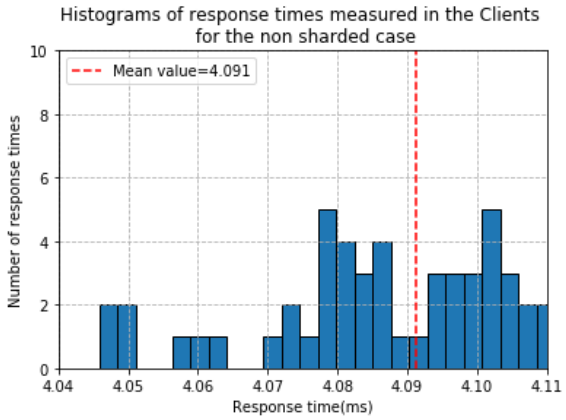
Considering now the throughput, table 7 of the two configurations some important conclusions can be drawn. The table elements are calculated as follow. The second and third column are the number of sets + number of multi-gets. The forth and fifth columns are calculated when considering the number of multi-gets as (key-size)*(num multi-get). These results show the conclusion that we reached in experiment 2, each server cannot reply to more than 3000 requests, in this case, since we have 3 servers, it would be 9000 and in fact the throughput (minus the set requests) is around 9000.



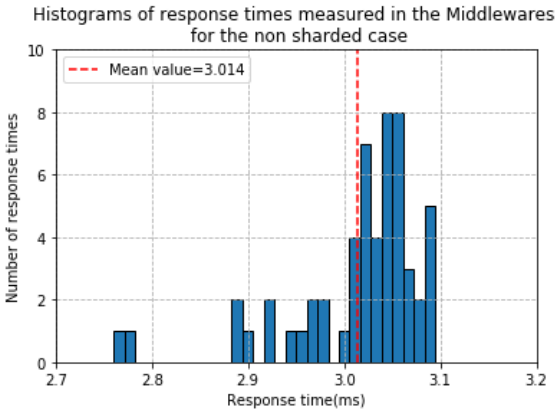
(a) Histogram of the latencies aggregated from all the clients for experiment 5.1.



(b) Histogram of the latencies aggregated from all the middlewares for experiment 5.1.



(c) Histogram of the latencies aggregated from all the clients for experiment 5.2.



(d) Histogram of the latencies aggregated from all the middlewares for experiment 5.2.

Figure 21: Histograms of experiments 5.1 and 5.2 (sharded and non-sharded case).

Multi-key size	**Parsing Time avg (std) sharded case**	**Parsing Time avg (std) non sharded case**	**Service Time avg (std) sharded case**	**Service Time avg (std) non sharded case**
1	0.20 (0.01)	0.217 (0.03)	1.17 (0.10)	1.128 (0.08)
3	0.49 (0.05)	0.41 (0.02)	1.278 (0.13)	1.160 (0.17)
6	1.26 (0.3)	1.26 (0.04)	1.68 (0.08)	1.673 (0.06)
9	2.69 (0.6)	1.11 (0.04)	2.17 (0.08)	3.75 (0.25)

Table 6: Table of parsing times for sharded and non sharded configuration.

Multi-key size	**Throughput sharded**	**Throughput non sharded**	**Throughput * key Size sharded**	**Throughput * key Size non sharded**
1	4921.67	4959.06	4921.67	4959.06
3	4140.85	4451.42	8281.71	8902.84
6	2833.55	2865.85	9917.42	10030.486
9	1925.21	1940.75	9626.06	9703.75

Table 7: Table of Throughput for sharded and non sharded configuration.

6 2K Analysis (90 pts)

For the following sections I am gonna use X_a to refer to the servers, X_b for the Middlewares, X_c for the worker threads.

$$X_a = \begin{cases} -1, & \text{if 2 servers} \\ 1, & \text{if 3 servers} \end{cases}, X_b = \begin{cases} -1, & \text{if 1 Middleware} \\ 1, & \text{if 2 Middlewares} \end{cases}, X_c = \begin{cases} -1, & \text{if 8 Worker Threads} \\ 1, & \text{if 32 Worker Threads} \end{cases}$$

The following additive model is used in my analysis:

$$y = q_0 + q_a X_a + q_b X_b + q_c X_c + q_{ab} X_a X_b + q_{ac} X_a X_c + q_{bc} X_b X_c + q_{abc} X_a X_b X_c + error$$

6.1 Write-only

6.1.1 2k analysis of throughput

First of all, the 2^k analysis was run with repetition, therefore instead of 2^k analysis we consider a 2^kr analysis. Where k is 3, because we change the number of middlewares, servers and worker threads, and r is 3 because we consider 3 repetitions. The experiments configurations are as follow:

- 1: 1 Middleware, 1 Server, 8 Threads.
- 2: 1 Middleware, 1 Server, 32 Threads.
- 3: 1 Middleware, 3 Server, 8 Threads.
- 4: 1 Middleware, 3 Server, 32 Threads.
- 5: 2 Middleware, 1 Server, 8 Threads.
- 6: 2 Middleware, 1 Server, 32 Threads.
- 7: 2 Middleware, 3 Server, 8 Threads.
- 8: 2 Middleware, 3 Server, 32 Threads.

First of all, the throughputs of the different tries were averaged as shown in table 8 (column 2). This value was then used in table 9 (column 10), to calculate the "Total/8" for each effect. Then, as stated in the above mentioned equation, the estimated response was calculated using all the previously found effects. In order to calculate the SSE, I had to calculate squared errors and sum all of them together. The errors were calculated by subtracting from each try's throughput the mean. After that, the effects found in "Total/8" were squared and multiplied by 24, these results are shown in table 10. Finally, these values were used to first calculate SST = SS0+SSA+...+SSE, and then to calculate the Variation as Vi=SSi/SST.

The values of the variation in table 11 show that the main cause of the throughput change is q_c which has a variation of 44%. Since we said that Xc represents the number of worker threads, this means that changing the worker threads has a good influence on the throughput. This confirms what we saw in all the previous experiments with a write-only workload, where changing the number of threads helped the throughput increasing, figures 9a or 13a and 17a. The other two factors that influence the throughput, however less significantly than the worker threads, are q_a and q_b which are number of server and middlewares respectively. In fact, we saw that going from exp 3.1 to exp 3.2, which means increasing the number of middlewares from 1 to 2, has a positive effect on the maximum value of throughput measured. While, for

Write-Only	Throughput avg	Measured responses		
Exp	**ymean**	**y1**	**y2**	**y3**
1	7687.25	7730.26	7557.61	7773.89
2	10733.46	10852.88	10368.46	10979.03
3	4680.34	4671.74	4725.05	4644.23
4	7167.81	7630.62	6678.76	7194.05
5	9065.82	9011.80	8981.35	9204.31
6	12984.88	13201.22	12695.75	13057.67
7	7219.17	6678.17	7053.97	7925.38
8	11407.70	11243.87	11480.86	11498.36

Table 8: Table shows different experiments throughput.

Write-Only	Effect								Throughput	Errors		
Exp	**q0**	**qa**	**qb**	**qc**	**qaqb**	**qaqc**	**qbqc**	**qaqbqc**	**ymean**	**e1**	**e2**	**e3**
1	1	-1	-1	-1	1	1	1	-1	7687.25	43.00	-129.64	86.63
2	1	-1	-1	1	1	-1	-1	1	10733.46	119.42	-364.99	245.57
3	1	1	-1	-1	-1	-1	1	1	4680.34	-8.63	44.71	-36.11
4	1	1	-1	1	-1	1	-1	-1	7167.81	462.81	-489.05	26.24
5	1	-1	1	-1	-1	1	-1	1	9065.82	-54.02	-84.47	138.49
6	1	-1	1	1	-1	-1	1	-1	12984.88	216.34	-289.13	72.79
7	1	1	1	-1	1	-1	-1	-1	7219.17	-541.00	-165.20	706.20
8	1	1	1	1	1	1	1	1	11407.70	-163.82	73.16	90.66
Total	70946.43	-9996.39	10408.71	13641.25	3148.73	-289.27	2573.91	828.196				
Total / 8	8868.30	-1249.54	1301.08	1705.15	393.59	-36.15	321.73	103.52		SSE = 1715428.90		

Table 9: Table used to measure effects and calculate errors.

Write-Only	Effect^2
SSO	1887523473.65
SSA	37472929.89
SSB	40627966.45
SSC	69781456.29
SSAB	3717937.73
SSAC	31378.92
SSBC	2484379.76
SSABC	257216.14

Table 10: Table show the sum of squares of the effects.

Write-Only	Mean Estimate	Variation Explained
qo	8868.30375	——
qa	-1249.54	24.01%
qb	1301.08	26.03%
qc	1705.15	44.71%
qab	393.59	2.38%
qac	-36.15	0.02%
qbc	321.73	1.59%
qabc	103.52	0.16%

Table 11: Table shows the mean estimate of the parameter and the related importance (Variation Explained).

Write-Only	Response Times avg	Measured responses		
Exp	**ymean**	**y1**	**y2**	**y3**
1	23.41	23.62	23.66	22.96
2	15.25	15.18	15.18	15.39
3	38.39	38.18	37.69	39.31
4	20.29	18.98	20.60	21.30
5	19.82	19.72	20.28	19.46
6	12.98	12.90	13.06	12.98
7	26.18	31.10	24.75	22.69
8	15.19	15.39	15.05	15.14

Table 12: Table shows different experiments response times.

the number of servers, we saw in experiment 4.1 that increasing the number of servers had the effect of decreasing the throughput of a factor similar to the increase measured in experiment 3.2, and in fact the percentage of the variation is almost the same, 24% vs 26%. However the sign of q_a and q_b are the opposite. As we mentioned already, this is due to implementation choices for the write workload. In fact the middleware has to send requests to all the servers, becoming slower as consequence. No significant influence has any other combination of factors.

6.1.2 2k analysis of response times

The same analysis was done for the response times. The only change in the way the tables were filled was made when calculating the average response time for each try of table 12. In fact, if before we were calculating the tries values by summing the throughput of the two middlewares, this time we are averaging the responses. Everything else is calculated in the same way. Analogous conclusions can be made by looking at these results, 15. We notice that the strongest influence is q_c which has an associated variation of 50%, we see that the estimate of q_c is a negative value, and in fact, increasing the number of Worker Threads from 8 to 32 has the effect of decreasing the response time. As we can see in a previous experiment, for example experiment 4 17b the response time of 32 threads (when considering 32*6=192 clients) is almost half of the one measured with 8 threads.

As for the influence given by the factor q_b, we see that it is less strong then the previous , however it is still a good percentage. In fact, as we saw in experiment 3, increasing the number of middlewares helped the response time decreasing, figures 9b and 13b.

Last, as we saw in experiment 4, the throughput decreased when incrementing the number of servers, as a consequence of the implementation choice of sending the sets in all the three servers. As a direct consequence, the response time increases 17b respect to experiment 3. This is confirmed by the variation of the factor q_a, which has a positive mean and a variation of 20%.

6.2 Read-only

6.2.1 2k analysis of throughput

The tables in figures 16, 17, 18 and 19 were calculated in the same way of the write only workload. Even the configurations of the experiments coincide with the previous one. In order to understand this analysis we have to look in detail at table 19. In this table we see that the maximum variation is associated with q_a, 98%, which is the term associated with the number of servers. Moreover, this variation is the only significant one, in fact all the others are almost 0%. This is perfectly in line to what we found previously. In fact, in the previous experiments,

Write-Only	Effect								Throughput	Errors		
Exp	**q0**	**qa**	**qb**	**qc**	**qaqb**	**qaqc**	**qbqc**	**qaqbqc**	**ymean**	**e1**	**e2**	**e3**
1	1	-1	-1	-1	1	1	1	-1	23.41	0.21	0.25	-0.45
2	1	-1	-1	1	1	-1	-1	1	15.25	-0.07	-0.07	0.14
3	1	1	-1	-1	-1	-1	1	1	38.39	-0.21	-0.70	0.92
4	1	1	-1	1	-1	1	-1	-1	20.29	-1.31	0.31	1.01
5	1	-1	1	-1	-1	1	-1	1	19.82	-0.10	0.46	-0.36
6	1	-1	1	1	-1	-1	1	-1	12.97	-0.07	0.07	0.00
7	1	1	1	-1	1	-1	-1	-1	26.18	4.92	-1.43	-3.49
8	1	1	1	1	1	1	1	1	15.19	0.20	-0.14	-0.05
Total	171.51	28.60	-23.18	-44.09	-11.45	-14.08	8.436	5.78				
Total / 8	21.43	3.57	-2.89	-5.51	-1.43	-1.76	1.05	0.72		SSE = 43.41		

Table 13: Table used to measure effects and calculate errors of response times.

Write-Only	Effect^2
SSO	11031.22
SSA	306.75
SSB	201.54
SSC	729.06
SSAB	49.17
SSAC	74.42
SSBC	26.69
SSABC	12.55

Table 14: Table show the sum of squares of the effects.

Write-Only	Mean Estimate	Variation Explained
qo	21.44	——
qa	3.58	21.24%
qb	-2.90	13.96%
qc	-5.51	50.50%
qab	-1.43	3.40%
qac	-1.76	5.15%
qbc	1.05	1.84%
qabc	0.72	0.86%

Table 15: Table shows the mean estimate of the parameter and the related importance (Variation Explained).

Read-Only	Throughput avg	Measured responses		
Exp	**ymean**	**y1**	**y2**	**y3**
1	2825.073	2917.44	2875.97	2681.81
2	2817.183	2639.68	2906.23	2905.64
3	8003.723	8106.56	7970.37	7934.24
4	8603.746	8587.58	8640.32	8583.34
5	2892.596	2901.58	2885.97	2890.24
6	2898.406	2897.21	2899.67	2898.34
7	8314.123	8714.93	8080.08	8147.36
8	8046.193	6772.69	8688.5	8677.39

Table 16: Table shows different experiments throughput.

Read-Only	Effect								Throughput	Errors		
Exp	**q0**	**qa**	**qb**	**qc**	**qaqb**	**qaqc**	**qbqc**	**qaqbqc**	**ymean**	**e1**	**e2**	**e3**
1	1	-1	-1	-1	1	1	1	-1	2825.073	92.37	50.90	-143.26
2	1	-1	-1	1	1	-1	-1	1	2817.183	-177.50	89.05	88.46
3	1	1	-1	-1	-1	-1	1	1	8003.723	102.84	-33.35	-69.48
4	1	1	-1	1	-1	1	-1	-1	8603.746	-16.17	36.57	-20.41
5	1	-1	1	-1	-1	1	-1	1	2892.596	8.54	-6.40	-2.13
6	1	-1	1	1	-1	-1	1	-1	2898.406	-1.20	1.26	-0.07
7	1	1	1	-1	1	-1	-1	-1	8314.123	400.81	-234.04	-166.76
8	1	1	1	1	1	1	1	1	8046.193	-1273.50	642.31	631.20
Total	44401.04	21534.52	-98.406	330.013	-395.9	334.173	-854.253	-881.653				
Total / 8	5550.13	2691.81	-12.300	41.251	-49.4875	41.771	-106.781	-110.206		SSE = 2773582.23		

Table 17: Table used to measure effects and calculate errors.

for example 3.1 and 3.2 we saw that the throughput of read-only increases only if the number of servers increase. In particular each server can respond to maximum 3000 requests, this is confirmed by the throughputs in table 16 where we see that the throughput is whether around 3000 requests or 9000 (3000 if the experiment is associated with 1 server, 9000 if there are 3 servers). No other factors influence the throughputs, not even combination of factors.

6.2.2 2k analysis of response times

The same analysis of the write-only 2k analysis for the response times was made.

These results confirm the previous conclusions. In table 23 we notice that the strongest and only significant influence is given by q_a (number of servers) which has an associated variation of 98%, we see that the estimate of q_a is a negative value, and in fact, increasing the number of Servers decreases the response time. As we saw in previous experiments, for example in

Read-Only	Effect^2
SSO	739287417.25
SSA	173904546.58
SSB	3647.95
SSC	40896.09
SSAB	58710.00
SSAC	41820.97
SSBC	273512.71
SSABC	291639.92

Table 18: Table show the sum of squares of the effects.

Read-Only	Mean Estimate	Variation Explained
qo	5550.10	——
qa	2691.84	98.04%
qb	-12.33	0.00%
qc	41.28	0.02%
qab	-49.46	0.03%
qac	41.74	0.02%
qbc	-106.75	0.15%
qabc	-110.23	0.16%

Table 19: Table shows the mean estimate of the parameter and the related importance (Variation Explained).

Read-Only	Response Times avg	Measured responses		
Exp	**ymean**	**y1**	**y2**	**y3**
1	62.804	62.731	61.660	64.020
2	63.180	63.300	63.130	63.109
3	21.963	22.110	21.770	22.010
4	19.033	19.010	19.160	18.930
5	63.230	63.169	63.355	63.165
6	62.147	61.735	62.220	62.485
7	21.882	20.265	20.695	24.685
8	23.923	31.805	20.070	19.895

Table 20: Table shows different experiments throughput.

experiment 2 8, the read-only response time of the one server configuration is bigger than the two-servers configuration.

7 Queuing Model (90 pts)

7.1 M/M/1

The first model, M/M/1, is used to model processes interacting with one server. Some assumptions are made relative to the arrival rate (lambda) and the service rate (μ), which are considered exponential.

For each worker thread configuration, the parameters have been chosen as follow: μ is computed as : 2*threads / RT. RT is chosen by taking "service_time+parse_time" associated with the maximum throughput configuration (in terms of number of virtual clients). As for λ

Read-Only	Effect								Throughput	Errors		
Exp	**q0**	**qa**	**qb**	**qc**	**qaqb**	**qaqc**	**qbqc**	**qaqbqc**	**ymean**	**e1**	**e2**	**e3**
1	1	-1	-1	-1	1	1	1	-1	62.80	-0.07	-1.14	1.22
2	1	-1	-1	1	1	-1	-1	1	63.18	0.12	-0.05	-0.07
3	1	1	-1	-1	-1	-1	1	1	21.96	0.15	-0.19	0.05
4	1	1	-1	1	-1	1	-1	-1	19.03	-0.02	0.13	-0.10
5	1	-1	1	-1	-1	1	-1	1	63.23	-0.06	0.13	-0.06
6	1	-1	1	1	-1	-1	1	-1	62.15	-0.41	0.07	0.34
7	1	1	1	-1	1	-1	-1	-1	21.88	-1.62	-1.19	2.80
8	1	1	1	1	1	1	1	1	23.92	7.88	-3.85	-4.03
Total	338.16125	-164.5579166667	4.2013833333	-1.59525	5.4152833333	-0.1814166667	3.5126166667	6.4307166667				
Total / 8	42.27015625	-20.5697395833	0.5251729167	-0.19940625	0.6769104167	-0.0226770833	0.4390770833	0.8038395833		SSE = 108.29		

Table 21: Table used to measure effects and calculate errors.

Read-Only	Effect^2
SSO	42882.39
SSA	10154.74
SSB	6.62
SSC	0.95
SSAB	11.00
SSAC	0.01
SSBC	4.63
SSABC	15.51

Table 22: Table show the sum of squares of the effects.

Read-Only	Mean Estimate	Variation Explained
qo	42.27	——
qa	-20.57	98.57%
qb	0.53	0.06%
qc	-0.2	0.01%
qab	0.68	0.11%
qac	-0.02	0.00%
qbc	0.44	0.04%
qabc	0.8	0.15%

Table 23: Table shows the mean estimate of the parameter and the related importance (Variation Explained).

the value corresponding to the first saturation point was used. In my opinion this is a good choice because the M/M/1 model does not take into account the number of virtual clients, therefore it can't predict exactly the queue length, unless it just saturated.

In the following tables, 24, 25, 26, 27 are reported the modeled parameters but also the measured values for experiment 4.1, which let us compare the model and the experiments.

First of all, we see that for all WT configurations ρ is less then one, therefore a stable configuration was studied. Comparing the values of jobs in the system and the one in the queues predicted by the model we see that the nq is usually a few values smaller than the ns. This is easy to understand, in fact, this model hypothesizes that only one server at a time dequeues the requests. Therefore all the others are waiting in the queue. At the same time, it predicts that the majority of the response time is spent in the queue (and in fact the difference between the response time and the waiting time is very small). For all the WThreads we see that this model is pretty good at predicting the queue behavior. In fact, both the queue length and waiting time are predicted with an acceptable error. However, the response time predicted is usually much lower than the real one. If we now compare the waiting time in the different configurations, we see that the model is able to predict that increasing the number of worker threads has the effect of decreasing the waiting time.

To conclude, this model is a good estimate when there is the need to accurately model the queue.

7.2 M/M/m

Same values for λ and service rate where used in this model. Even in this case, I considered better choice to take the first point of the saturation. Therefore both mean arrival time and

8 Threads. 8*6 V. clients.	Mean arrival rate	Traffic intensity (μ = 7438.40)	Mean number of jobs in the system	Mean number of jobs in the queue	Mean response time (ms)	Mean waiting time (ms)
Estimated	7183.02	0.965	28.12	27.16	3.91	3.78
Measured	-	-	-	21.7	5.41	3.25

Table 24: M/M/1 model for 8-threads case. RT used for mu is 2.15 msec

16 Threads. 16*6 V. clients.	Mean arrival rate	Traffic intensity (μ = 9495.54)	Mean number of jobs in the system	Mean number of jobs in the queue	Mean response time (ms)	Mean waiting time (ms)
Estimated	9241.57	0.973	36.38	35.41	3.93	3.83
Measured	-	-	-	45.47	8.97	5.44

Table 25: M/M/1 model for 16-threads case. RT used for mu is 3.37 msec

32 Threads. 16*6 V. clients.	Mean arrival rate	Traffic intensity (μ = 11573.23)	Mean number of jobs in the system	Mean number of jobs in the queue	Mean response time (ms)	Mean waiting time (ms)
Estimated	11012.63	0.95	19.64	18.69	1.78	1.69
Measured	-	-	-	16.78	7.08	1.7

Table 26: M/M/1 model for 32-threads case. RT used for mu is 5.53 msec

64 Threads. 22*6 V. clients.	Mean arrival rate	Traffic intensity (μ = 14004.37)	Mean number of jobs in the system	Mean number of jobs in the queue	Mean response time (ms)	Mean waiting time (ms)
Estimated	12012.32	0.85	6.03	5.17	0.5	0.4
Measured	-	-	-	8.76	9.15	0.81

Table 27: M/M/1 model for 64-threads case. RT used for mu is 9.14 msec

64 Threads. 96 V. clients.	Mean arrival rate	Traffic intensity (mu = 14004.37)	Mean number of jobs in the system	Mean number of jobs in the queue	Mean response time (ms)	Mean waiting time (ms)
Estimated	12012.32	0.85	110.14	0.35	9.16	0.03
Measured	-	-	-	8.76	9.15	0.81

Table 28: M/M/M model for 64-threads case

traffic intensity (rho) were taken the same as the previous experiment.

The data in table 28 refer to the M/M/m model with 64 threads per Middleware. If we compare this with the M/M/1 model, table 27 we can see that the predicted number of jobs in the system increased a lot and it is more accurate than the previous model (M/M/1), in fact in this case the model takes into account the number of worker threads that my system is using.

However, contrarily to the previous case (M/M/1) where the queue time was predicted more precisely, now the queue time is very small and far away from the true value. However, the response time is the same as the measured one. This model assumes that M servers fulfill the requests, therefore the waiting time decreases but the total response time increases because more jobs need to be processed.

Considering the fact that the response time is more similar to the measured ones, I regard this model more accurate for my system, respect to the first one. Since in my opinion, the accuracy of the total response time is more important than only the waiting time.

7.3 Network of Queues

The models (M/M/1 and M/M/m) considered in the previous sections had only one queue. However, in order to have a more accurate description of the system, we should consider a network of queues. These models have several queues. In this case, a job departing from one of these queues arrives at another one.

Considering my system, figure 1, I decided to model the network in the following way.

The interaction between clients and middleware is modeled as a M/M/1 queue. When considering two middlewares, I will use two M/M/1 queues. Secondly, both for one and two middlewares, the part where the worker threads dequeue the requests and send them to the servers is modeled as a M/M/m queue.

7.4 Network of Queues for 1 Middleware

The following tables represent the model's predictions for the main parameters (Throughput and utilization of the i-th device). These tables are filled with the data from experiment 3.1, where one middleware is used.

Tables **??** and **??** show the behavior for 64 worker threads in one middleware, and two different configuration: 6 and 192 virtual clients. The other configurations had similar behavior so I considered useless to put all the configurations. In the table the "Parser" is considered as the server of the M/M/1 model that takes the requests from the client, process them and put them in a queue. While the "Worker Thread" model the server of the M/M/m model. In both tables the parser has visit ratio equals to one, because since we have one middleware, all the throughput has to go through it. Then, each worker thread, has a visit ratio of 1/(tot WT). The service time is taken from the experiments and the utilization is calculated as shown in the table.

Looking at the write only case, table **??**, we see that the utilization starts from a very low value and then, for the last number of VC it approaches one. In fact, with 192 virtual

64 Worker Threads WRITE-ONLY	Component	Visit ratio (Vi)	Throughput i-th Xi = X*Vi	Service time i-th Si	Utilization i-th Ui = Xi*Si
6 Virtual clients X=2791.57	Parser	1	Xi = 2791.57	-	-
	Worker Thread	1/64	Xi = 43.62	1.10 msec	Ui = 0.047
192 VC X=11965.09	Parser	1	Xi = 11965.09	-	-
	Worker Thread	1/64	Xi = 186.95	5.07 msec	Ui = 0.95

Table 29: Table of model parameters for write-only case with 64 worker threads and one middleware.

64 Worker Threads READ-ONLY	Component	Visit ratio (Vi)	Throughput i-th Xi = X*Vi	Service time i-th Si	Utilization i-th Ui = Xi*Si
6 Virtual clients X=2705.26	Parser	1	Xi = 2705.26	-	-
	Worker Thread	1/64	Xi = 42.27	1.09 msec	Ui = 0.046
192 VC X=2934.77	Parser	1	Xi = 2934.77	-	-
	Worker Thread	1/64	Xi = 45.85	20.56 msec	Ui = 0.94

Table 30: Table of model parameters for read-only case with 64 worker threads and one middleware.

clients the saturation phase is already reached and the worker threads are exploited as much as possible due to the fact that the server increases the service time. This makes the middleware the bottleneck of the experiment.

Regarding the read-only case, analogous behavior can be observed. In fact, when 6 vc are considered, the utilization is very low, because the server is still able to reply fast. However, when the server increases the time it needs to reply, the utilization increases to 0.98.

7.5 Network of Queues for 2 Middlewares

As for the two middlewares case, same configurations, in terms of WTs and Virtual Clients, are used for the tables. The only thing that changes it that now the throughput is split between two middlewares therefore the parser has a visit ratio of half the size and consequently even the worker threads.

Analogous conclusions to the previous section can be made when considering the write only and read-only scenarios. In fact, in both cases, for 6 virtual clients the worker threads are not exploited too much, however when jumping to 192 virtual clients, the utilization approaches 1, due to the increased service time.

Even in this case, the utilizations of the worker threads approach one, therefore the middleware is the bottleneck.

64 Worker Threads WRITE-ONLY	Component	Visit ratio (Vi)	Throughput i-th Xi = X*Vi	Service time i-th Si	Utilization i-th Ui = Xi*Si
6 Virtual clients X=2948.27	Parser	1	Xi = 1474.13	-	-
	Worker Thread	(1/2)/64	Xi = 23.03	1.20 msec	Ui = 0.027
192 VC X=14567.11	Parser	1/2	Xi = 7283.5	-	-
	Worker Thread	(1/2)/64	Xi = 113.80	8.47 msec	Ui = 0.96

Table 31: Table of model parameters for write-only case with 64 worker threads and two middlewares.

64 Worker Threads READ-ONLY	Component	Visit ratio (Vi)	Throughput i-th Xi = X*Vi	Service time i-th Si	Utilization i-th Ui = Xi*Si
6 Virtual clients X=2876.67	**Parser**	1/2	Xi = 1438.33	-	-
	Worker Thread	(1/2)/64	Xi = 22.47	0.91 msec	Ui = 0.02
192 VC X=2890.38	**Parser**	1/2	Xi = 2934.77	-	-
	Worker Thread	(1/2)/64	Xi = 22.58	43.52 msec	Ui = 0.98

Table 32: Table of model parameters for read-only case with 64 worker threads and two middleware.