Advanced Systems Lab Report

Autumn Semester 2017

Name: Andrea Rinaldi
Legi: 17-941-626

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	

Total	

1 System Overview

The starting point of the middleware is the main method inside the class `RunMW`, where, after the parsing of the arguments provided from the command line interface, the call to the `run()` method of the class `MyMiddleware` is executed. `MyMiddleware` implements the `Runnable` interface, so it exposes the method `run()`. This is where the middleware server-socket channel is created, that is configured as non-blocking and bound to the IP address and port provided before.
The implementation of a non-blocking IO instead of a blocking one, has been a straight-forward decision since this enables the `ClientHandler` - that is the thread that manages the clients connections and their requests in input - to move to the next client channel if the previous one is not ready to send a request. Otherwise, that thread would block until there is some data to read, or the data is fully written. In fact, if a client sends a request before another one does not necessarily mean that they will receive their respective replies in that same order, and so, it is not desirable that the `ClientHandler` follows a strictly ordinal loop fashion.

After that, the parameters describing the servers are parsed and stored in a matrix in which each row contains the IP and port of a server, and this 2-D array will then be passed to the constructor of the `WorkerThread` class to create the socket connections with memcached instances. To keep track of the load of each server during the experiment, I create a `HashMap` called `serversLoad` that I instantiate at this point with the servers IP addresses as keys and value 0 and that will be incremented during the experiment through the counter `load` in the class `ServerHandler`.

For the creation of the worker threads, I firstly I instantiate a new `FixedThreadPool` of dimension equal to the number of worker threads specified by the parameters from CLI. Then, I run all those worker threads submitting them to the `Executor`. Moreover, I add them to an `ArrayList` of `WorkerThread` that will be helpful in the `ShutdownHook`, when killing the middleware, in order to log the statistics collected by each worker thread during the experiment. Apart form a number identifying each worker thread and the servers IPs and ports (as mentioned before), the other parameter necessary to build a worker thread is the network queue where the incoming requests are stored.
The choice of the queue implementation has been crucial for obtaining good performances on the cloud. As a matter of fact, in my first attempt I was using the `ConcurrentLinkedQueue` class; while this seemed to work well on local, it flawed on Azure, due to the fact that, while there are no requests in the queue, every worker thread iterate inside a “hot loop” causing the performances to decline drastically as soon as the number of threads increases. The final implementation utilizes a `LinkedBlockingQueue`, that, contrary to the other class, provides the `take()` method that waits if necessary until an element becomes available, avoiding being stuck in many useless iterations.

Moving to the instantiation of the `ClientHandler` I decided to exploit the *Singleton* pattern, since there will be only one object of this class. I set the network queue in this object, that is where it will put the requests from memtier, and I set the `Selector` to allow the thread to handle multiple channels, and thus multiple network connections. Naturally, also `ClientHandler` implements the `Runnable` interface, so that from `MyMiddleware` I can now call the `run()` method.
The “main flow” in `MyMiddleware` essentially ends with a loop in which the server-socket channel accepts the incoming connections from the clients through the method `accept()` that returns a `SocketChannel`. To be consistent with the whole implementation of the connection interface in input, I also configure this channel as non-blocking, and finally I register it with the selector.

Noteworthy are the timers that collect the statistics during the experiment and the `ShutdownHook`. I will now cover these two correlated parts of the middleware quite extensively.
In the `MyMiddleware` class there are two timers and they both call the `scheduleAtFixedRate()` method passing a new anonymous `TimerTask` class as a parameter. In the `run()` method of one of the two `TimerTasks`, I just add the current size of the queue to an `ArrayList` every 50 ms. I found this time interval to work fine empirically, since I can monitor the changes in the queue with a decent granularity without impacting much on the performance.
The other `TimerTask` collects information for a 4 second window, and then it waits 6.4 seconds to gather data again.
Essentially, every worker thread has three `ArrayLists` that are useful for the instrumentation of the middleware:

- `times` that stores, for every request, the waiting time in the queue, the time for parsing the request in the worker thread, the time waiting for the memcached reply, the service time, the response time and the total time in the system. Moreover, every row (that is every request) stores the number (ID) of the worker thread that handled that request and the type of the request (being 0 if SET, 10 if GET or a number from 1 to 9 for the MULTI-GET depending on the number of keys in the request).
- `myThroughput` that stores the throughput of that single worker thread. In fact, when a worker thread handles a request, it increment both the `static AtomicLong` variable `numOfRequests` of the class `WorkerThread` (that contains the total number of requests handled by all the worker threads) and the non static variable `myNumOfRequests` that just stores the number of requests for that single worker thread. By doing so, in the timer, I can add to the `myThroughput` variable of every worker, its throughput for that time window (i.e. `myNumOfRequests / *time passed in the TimerTask*`). Moreover, with the total number of requests in `numOfRequests`, in the same way, I can compute the total throughput during that time window.
- `unproperRequests` that collects the requests that might be not well-formed (i.e. do not start neither with "set" nor with "get"), and the error messages from memcached. I decided to keep both the types in the same `ArrayList` for two reasons: I have never seen any of those two messages (always had an empty file), and because, in case, they would be totally distinguishable.

The ClientHandler class has one `ArrayList`, that is important for the instrumentation, called `interArrivalTimes`, that, as the name suggests, stores the time intervals between two successive arrivals from memtier.

Inside the `run()` method of this `TimerTask`, firstly, I clear the `times` list, the `interArrivalTimes` list, and set the variable `numOfRequests` to 0, so that now, I can collect just the data relative to the 4 seconds windows by doing `Thread.sleep(4000)` and storing the modified "versions" of these, previously cleared, variables in some variables of `MyMiddleware`.
These variables in the `MyMiddleware` class (namely, `throughput`, that contains the total throughput during every one of those time windows, `times` that is the collection of all the times lists of every worker thread, `interArrivalTime`, that collects the interarrival times, and `unproperRequests`) will be helpful when shutting down the middleware and logging that data to files. In the `ShutdownHook`, apart from those lists just mentioned, also the number of set, get and multi-gets, the cache misses and the distinct throughputs of every worker thread are written into files.

Following the path of the requests, I will now start the description of how the `ClientHandler` (that is the net thread) works. While the `Selector` is open, it checks whether the keys in the selected-key set, are ready to submit a new request and, if so, it reads the buffer until the message ends, otherwise the selector moves to the next key. In the first case, after we have read the whole request from the buffer, a `Job` is created and put into the queue. The `Job` constructor requires the message itself, the key (i.e. the clients that made the request), the time we started handling this new request (`arrivalTime`), and the time we put the job in the queue (`queueEntranceTime`).
Finally, we store in the `interArrivalTimes` list the difference between the `arrivalTime` of this last request and the `arrivalTime` of the previous one.

In the next step, one of the worker threads takes the first request out of the queue and it saves in the `queueTime` variable, the time this job has spent in the queue. Then, depending on the first three letters of the message, it starts the parsing and sending procedure for a Set or a Get. If none of those applies, the message is added to the `unproperRequests` list and it is not forwarded to memcached, but an error message is sent back. In case of a Set request, after having incremented the total number of Sets and having put the `type` to 0, the worker thread sends the message to all the servers and save the `sendTime` in a variable. Then, it fetches the responses from all the servers and add them in an `ArrayList`. We also save the time in which all the responses from the server have been received (`receiveTime`). If any of those replies is different from "STORED", this is added to `unproperRequests` and forwarded to the client. Otherwise, we reply back "STORED".

For the get and multi-get requests, we can take in one of two paths. We firstly need to split (into `requests` array) the request so that, if it is a multi-get with sharding enabled we can forward parts of it to all the servers.
If the request is a get or a multi-get with no sharding enabled, the worker thread has to forward the message just to one server. We start by incrementing the corresponding counter for one of the two types (get or multi-get) and set the request `type=10` for get or `type=requests.length` for multi-get. Then, the function `loadBalance()` is called, that increments a counter `serverCount` and takes its remainder by the number of servers (i.e. : `serverCount++; return serverCount % *number_of_servers*`). Lastly, the worker thread sends the request to the identified recipient and fetches the response (always saving `sendTime` and `receiveTime`). By splitting the response through the string "VALUE" we can calculate and save the number of misses. Lastly, if the reply does not end with "END" we add it to `unproperRequests` and, in any case, the reply is sent back to the client.

For the last scenario, that is multi-get with sharding enabled, we compute the number of requests per server (`int requestsPerServer`) and how many requests would remain from that division (`int remainingRequests`). After that, we build a message that contains `requestsPerServer` parts of the initial request, and while there are remaining requests from that division we add one of those to the message. Since we only add one (at most) of these remaining requests to a message, and since the number of remaining requests is certainly smaller than the number of servers, every server will receive at least `requestsPerServer` requests and at most `requestsPerServer+1` requests. Then, the worker thread sends the messages to all the servers, collects useful data to log, and fetches all the responses. If any of the replies does not end with "END", it is sent to the client, otherwise we aggregate the replies by removing the final "END" from every one of those, concatenating the strings and adding a final "END". Lastly, this final response is forwarded to the client.

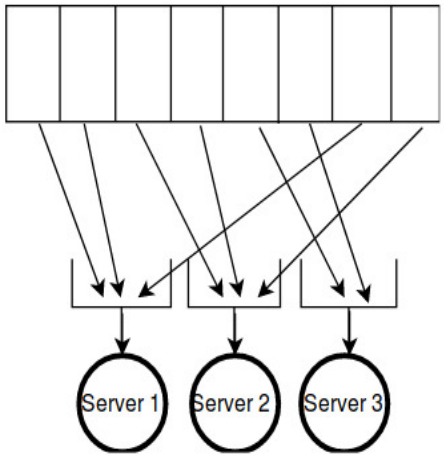

Figure 1.1 How Multi-Get is divided among the Servers

The last thing I want to point out regarding the design and implementation decisions for the middleware, is the connection between the worker threads and the server handlers. `ServerHandler` is the class that effectively connects through the `Socket` with a server and contains the `PrintWriter` and `BufferedReader` to interact with it. Since it is required that, firstly, all the requests are sent to the servers and then all the response are collected (instead of an iterative send-receive), I tried two ways of implementing this.
As a first attempt, I was creating a thread inside `ServerHandler` for every request received from a worker thread so that the worker thread could go on sending the other messages to the other servers. Even though this solution had the advantage of creating only as many `ServerHandler` objects as the number of servers, that caused the performance to decrease by a factor of approximately 30 if compared with the results without the middleware. Moreover, synchronization was needed to avoid mixing the requests and responses of different worker threads. In the other solution, that is the one implemented in the final version, every worker thread has a `ServerHandler` object for every server. As a consequence, the number of `ServerHandlers` (and so `PrintWriters`, `BufferedReader` and `Sockets`) increases linearly with the number of worker threads, but performances are just 3-4 times slower than the ones without middleware and synchronization is not necessary since there is a dedicated channel for every worker thread – server pair.

Lastly, a quick mention to the closing operations for threads and socket. To shut down the worker threads "gracefully", when we have finished collecting data from them, I put a special request in the queue with the message "EXIT" so that, when the the worker thread polls that from the queue, it closes the connections with the servers, and it stops. Additionally, the `ServerSocketChannel` and the `Selector` are closed, and consequently, also the `ClientHandler` thread is stopped.

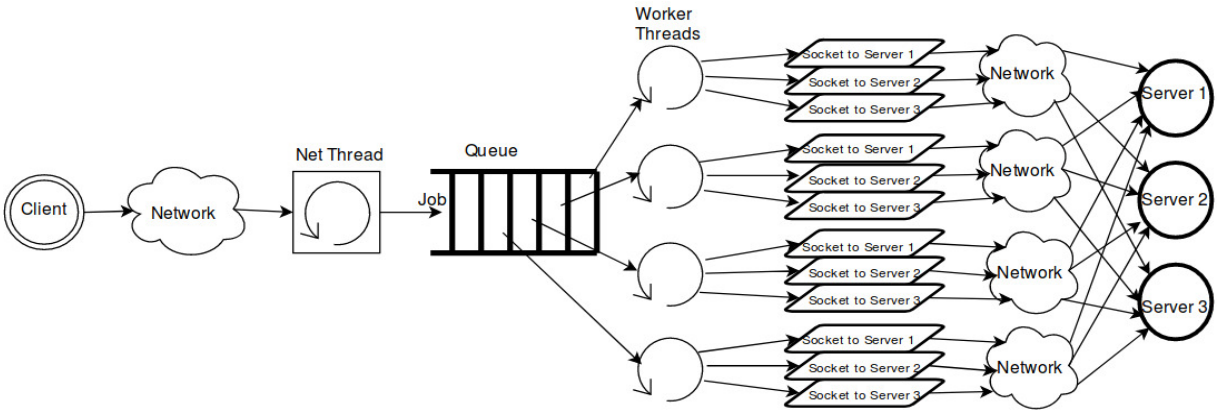

Figure 1.2 Illustration of the System implementation

2 Baseline Without Middleware

2.1 One Server

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[1..50]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	3

2.1.1 Explanation

Looking at the plots in figures 2.1.1 and 2.1.2 it is clear that, for this configuration of the system under test, we have very different behaviors for Gets and Sets. In particular, even though the throughput at 6 clients is approximately equal (`9553.58 ops/sec` for Gets and `9831.53 ops/sec` for Sets), at 24 clients, the system already saturates in the read-only experiment, while the performances rise considerably in the write-only experiment.
Increasing the number of clients, we can see that, for the read-only experiment, the throughput remains stable at nearly `11139.103 ops/sec` while, for sets, it keeps increasing with a trend similar to:

$$c\left(1-\exp\left(-numOfClients\right)\right)+d$$

peaking at `36456.626 ops/sec` for 300 clients.
Since the increment in throughput between 240 and 300 clients, for the write-only experiment, is just of 0.122%, we can state that the system saturates at 240 clients.
A steep increase in the response time for the Gets can be seen over the whole range of client configurations, starting at 0.9526 ms and peaking at 28.1135 ms with 6 and 300 clients respectively. For the write-only experiment the difference in response time between the extreme client configurations is much more moderate since, with 6 clients, the response time is equal to the one for the Gets, and with 300 clients it peaks at 8.3452 ms. Overall, for both the experiments, the relation between throughput X and response time R follows the interactive law $R=\left(N/X\right)-Z$ with an average "think time" Z of 0.4907 ms (N = number of clients).

Name: Andrea Rinaldi Legi: 17-941-626

Now that we have discussed the general behavior of the system in the two cases, we can study in details what are the reasons why the two experiments are so different from one another. This can be explained precisely analyzing the output of *dstat* for the server. In fact, in the read-only experiment, the server is able to send back at most `12850355.2 Byte/sec` and this value is stable around `12 MB/sec` from 24 up to 300 clients. Since the cpu of the server VM is not being utilized at its maximum (avg. of 89% idle processes), we can conclude that the server is network-bound for the read-only experiment.
The results are confirmed if we multiply the observed throughput for the the message size (1024 Byte).
On the contrary, for the Sets, we can notice that, as the number of clients increases, the number of idle processes on the server tends to 0, meaning that the cpu of the server can't handle more than the workload produced by the 300 the clients. To be precise, the server recorded a maximum of `42760186.8 Byte/s` incoming workload and a minimum of idle processes of 0.2%.
Thus, in this case, we can conclude that the server cpu is the bottleneck of the system.

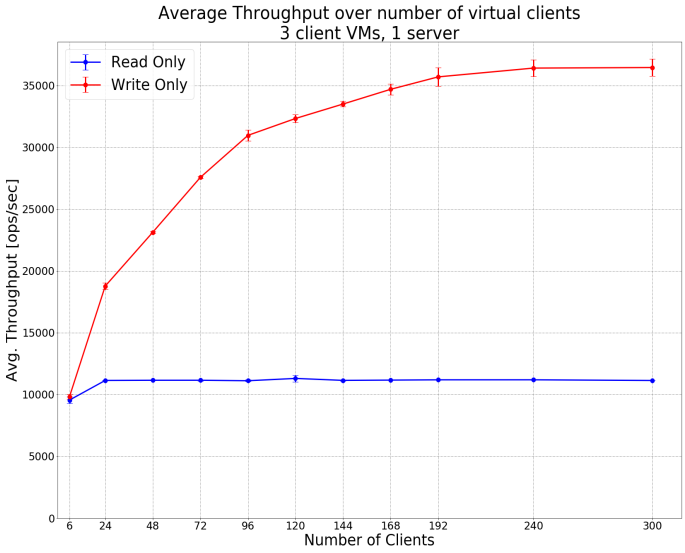

Figure 2.1.1 Plot of the Throughput

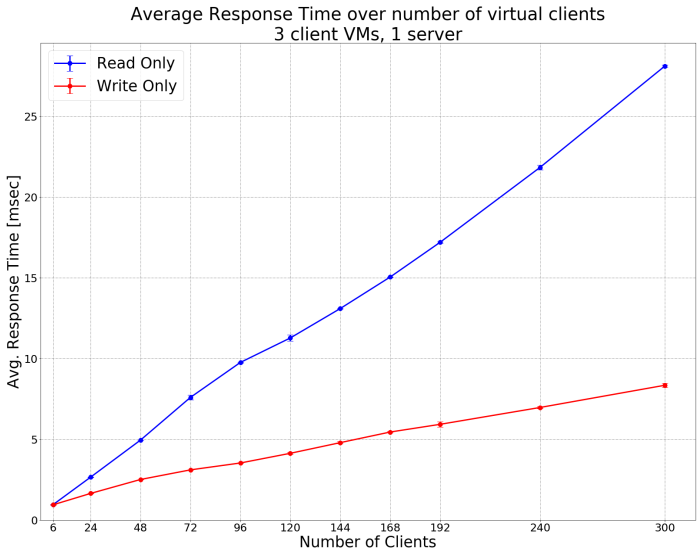

Figure 2.1.2 Plot of the Response Time

2.2 Two Servers

Number of servers	2
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1..50]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	3

2.2.1 Explanation

The analysis of the second part of this section will help us understand the behavior of memtier and memcached and it will also confirm the results and the explanations we gave previously.

At a glance, from figures 2.2.1 and 2.2.2 it is noticeable that, unlike the previous part, with this configuration of the system, we have very similar throughput and response times for Gets and Sets.
The response times follow almost precisely the same trend, starting at 1.3 ms and peaking at 4.8 ms.
Looking at the maximum throughput, for the write only experiment we reach `21240.78 ops/sec` while, for the read-only `20593.37 ops/sec`.
It is not coincidence that the results for the throughput are approximately twice the value we observed for the read-only experiment with one server. As a matter of fact, if we analyze the output of *dstat* for the servers and the client VM, we see that, once again, each server is able to send at most approximately `12 MB/sec` (so the client receives at most twice that Byte/sec). Following the same reasoning, we would expect that also each memtier instance can send up to that workload per second (`12 MB/sec`). This conclusion, in fact, is confirmed by the measurements we get using *dstat* on the client.
As a consequence, the throughput and response time are very similar for Gets and Sets.
To conclude, with this configuration, the system is network-bound in both cases, when sending the 1024 Byte message to the servers in the write-only experiment, and when receiving the 1024 Byte responses from the servers, in the read-only experiment.
In order to have a further proof of our deductions, we can see on Figure 2.2.1 that the system has not completely saturated with this range of client configurations, since the difference in throughput between 240 and 300 clients is 1.98% and 2.23% for sets and gets respectively. If we consider our knowledge about the maximum workload per second allowed by the network (12 MB/sec), we can be confident that the system will saturate at around 22k ops/sec, that would be, in fact, twice the maximum throughput of the read-only experiment with one server.

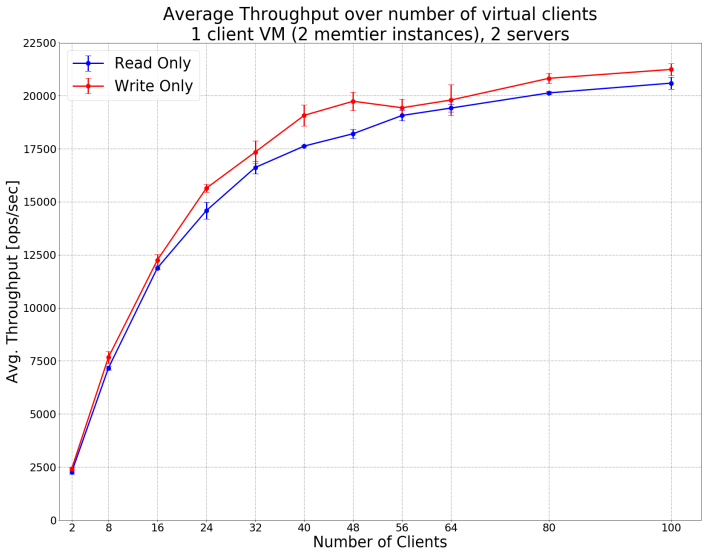

Figure 2.2.1 Plot of the Throughput

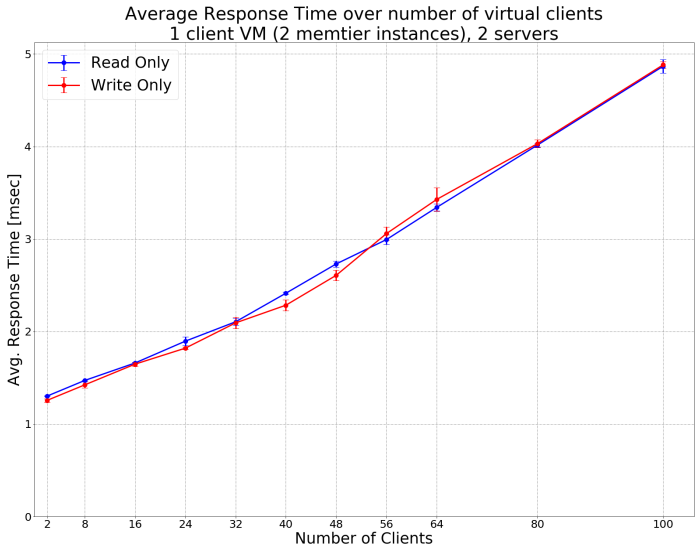

Figure 2.2.2 Plot of the Response Time

2.3 Summary

Maximum throughput of different VMs

	Read-only workload [ops/sec]	Write-Only workload [ops/sec]	Configuration that gives max. throughput [VC]
Memcached Server	11303.170	36456.626	120 Read / 300 Write
Load Generating VM	20593.376	21240.786	100

The results obtained in the two sections are very clear and need to be considered together to be fully understood.

Name: Andrea Rinaldi Legi: 17-941-626

Firstly, let's look at the maximum throughput for the write-only experiment with one server. As previously mentioned, in this case, the clients can generate such a workload that the server reaches its computational power limit, with the number of idle processes close to 0. Consequently, we can state that the server cpu is the bottleneck of the system. This result will also be very useful in the following experiments, since we are now aware of the computational capabilities of the server VMs.
Turning to the read-only experiment with one server, we have already outlined that the system saturates as soon as there are 24 virtual clients generating the load. This is due to the fact that the network upper-bound of approximately 12 MB/sec is reached and, thus, the server cannot send back more than that workload. This outcome relates very closely to the maximum throughputs reached in the experiments with two servers. In fact, as we would expect, the performance for the read-only experiment doubles as there are twice as many servers that can now send at most 12MB/sec each, to the memtier instances. Contrarily, as for the write-only experiment, decreasing the number of client VMs, cause them to be subject to the same network bound of the Gets. In fact, each memtier instance is not able to forward more than that approximately 12MB/s to the servers. The difference between the throughput of the read-only and write-only experiments in Section 2.2, can be explained as an intrinsic property of the memcached server to be faster in processing Set requests than Get requests. This is a result that we will also encounter in the next experiment (Section 3.1).

3 Baseline With Middleware

3.1 One Middleware

Number of servers	1
Number of client machines	1
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[1..110]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1
Worker threads per middleware	[8..64]
Repetitions	3 (80 sec each)

3.1.1 Explanation

In this section we will analyze how the system behaves with one load generator VM, one middleware and one server. We will consider the changes that take place for throughput and response time when changing the number of clients and the number of worker threads inside the middleware.
Looking at Figure 3.1.6 we can see that, as we scale the number of worker threads up, the performance we get are generally higher for larger numbers of clients. In fact, for 2, 8 and 16 virtual clients, the throughput is the same for all the four different configurations of worker threads, but if we look at the results with more clients, it is noticeable that the system behaves differently depending on the number of worker threads. It saturates at `2503.46 ops/sec` and at `4842.868 ops/sec` for 8 and 16 worker threads respectively, while, with 32 and 64 worker threads the throughput and response time follow the same trend up to 120 clients. At this point, the system with 32 worker threads saturates at `8591.34 ops/sec` but, with 64 worker threads, it keeps increasing up to `9998.79 ops/sec` with 170 clients, when it finally saturates.
We will now study the behavior of the system in more detail, starting with the configurations of the middleware with 8, 16 and 32 worker threads. If we take into consideration the results and conclusions we drew in the previous section, we know that the server cannot be the cause of the saturation since, for the write only experiment, it could handle up to `36456.626 ops/sec`.Moreover, in section 2.1 we saw that each virtual machine could upload up to 12 MB/sec to the server, so we can assume that the network is not limiting the performance of the system in these cases.

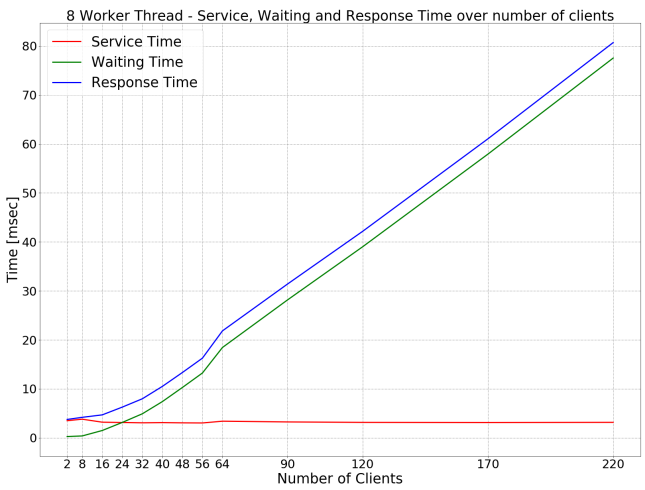

Figure 3.1.1 Times with 8 Worker Threads

Figure 3.1.2 Times with 16 Worker Threads

Figure 3.1.3 Times with 32 Worker Threads

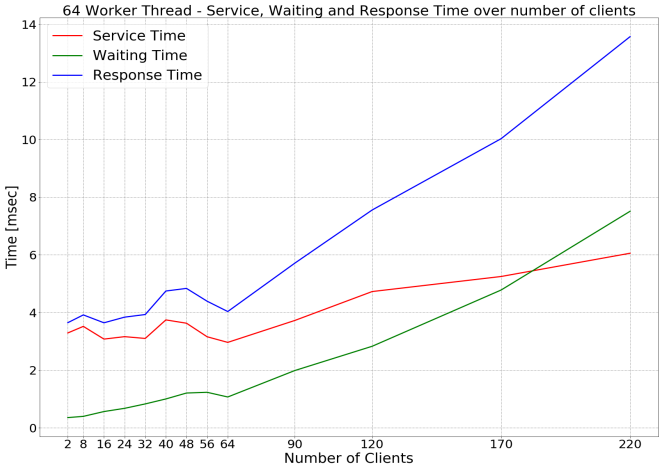

Figure 3.1.4 Times with 64 Worker Threads

Looking at the plots for waiting time and response time (Figures 3.1.1, 3.1.2, 3.1.3), it is clear that the middleware (the worker threads to be more accurate) is, in fact, limiting the performance of the system. Indeed, it happens in every one of these configurations, that the average time spent by a job in the queue gets higher than the service time, and is the main contribution to the increasing response time.

Combining this fact, with the knowledge that the service time remains stable and the server is not saturated, we can conclude that the middleware is indeed the bottleneck of the system.
I decided to analyze separately these three configuration and the one with 64 worker threads, to point out a slight difference between the two cases. What we just mentioned is also valid for the 64-worker-thread configuration of the middleware, with the exception of the reasoning on the network. In fact, looking at the Figure 3.1.4**,** we can see that, once again, the waiting time becomes higher than the service time at a certain point. Although, looking at the average queue length (Figure 3.1.5), we don't (yet) see a significant increase in the average number of jobs in the queue,as much as we do for 8, 16 and

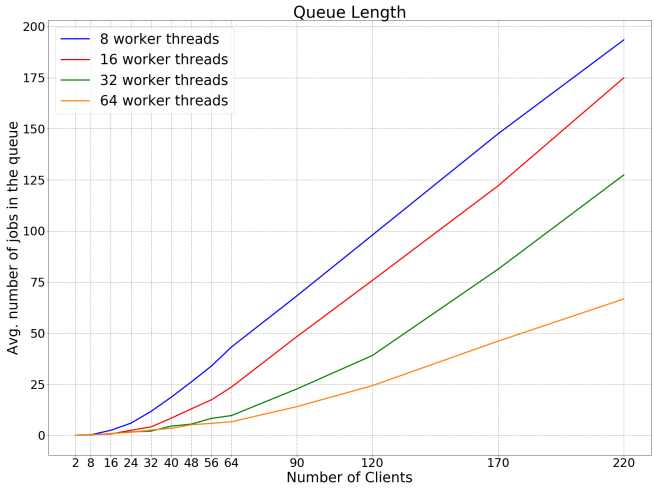

Figure 3.1.5 Plot of the Queue Length

32 worker threads. Additionally, analyzing the response time (Figure 3.1.7), we can show that, for the three previous configurations of worker threads, the difference between the overall increment in response time measured on the client and on the middleware is confined between 2.52% and 5.01%. The same value, with 64 worker threads, reaches 18.43%. This suggests that, for this configuration, the network could also be at its limit, since memtier indicates a much higher response time than the middleware does, as the number of clients increase. To have a further proof, we see that for 170 and 220 clients, 63.5% of the times, the logged throughput inside the middleware is higher than 10k ops/sec and 35.1% of the times it is higher than 11k ops/sec (that, we have seen, is approximately the maximum workload that can be generated with one client VM for Sets).
In conclusion, for the 64 worker threads configuration, we have proved that the middleware remains the main limiting factor of the performance of the system, but also the network is reaching its upper bound, contributing to the saturation of the system.

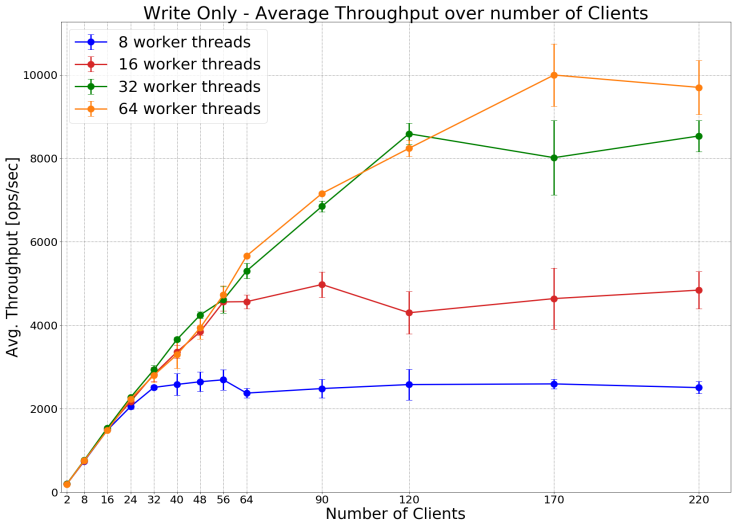

Figure 3.1.6 Plot of the Throughput

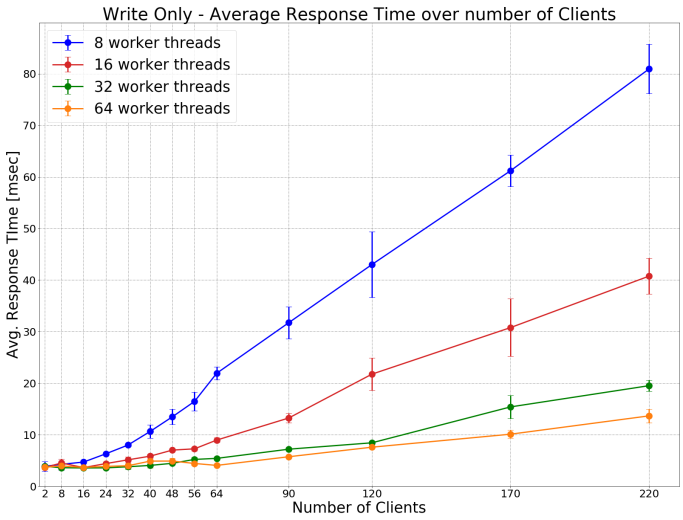

Figure 3.1.7 Plot of the Response Time

Turning now to the read-only experiment, the behavior of the system is exactly the same as the one we just analyzed for the write-only experiment. The only difference is that the maximum throughput for the 32 and 64 worker-thread configurations, is lower. This reflects the measurements and reasoning pointed out in section 2.2, in which we saw that the server was slower when processing Get requests than Set requests.

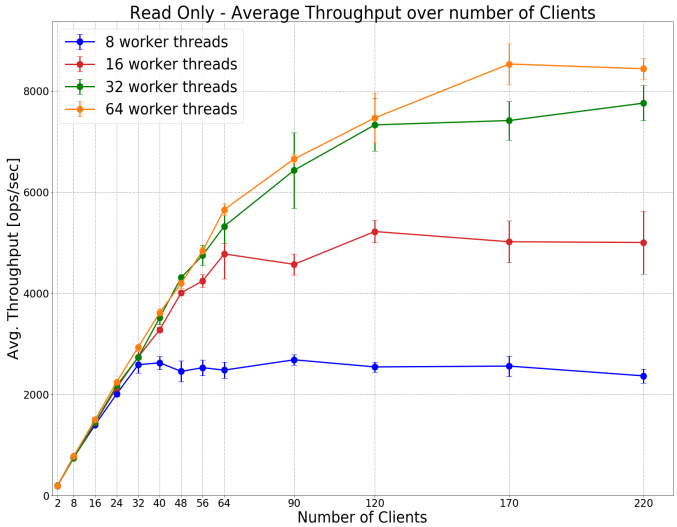

Figure 3.1.8 Plot of the Throughput

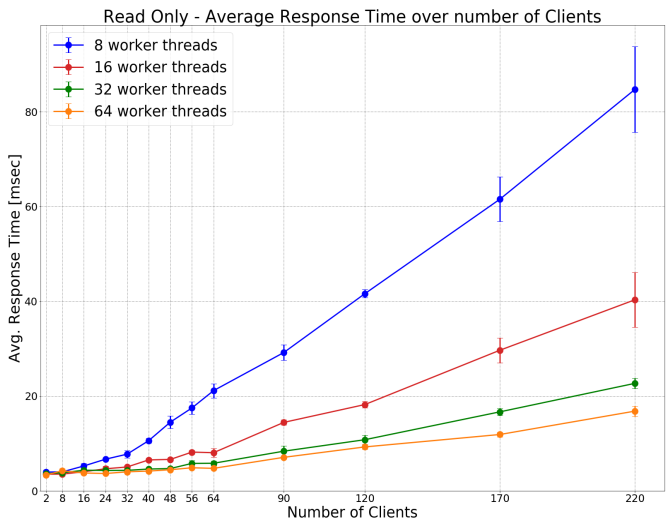

Figure 3.1.9 Plot of the Response Time

We follow now the same reasoning that we did for the write-only experiment.
Looking at the plots for the different time spent in the various components of the system (Figures 3.1.10 - 3.1.13), we see that the waiting time becomes higher than the service time, for different number of clients across the four configurations of worker threads. This means that, from a certain point onwards, the worker threads inside the middleware cannot keep up serving the workload that is generated by the clients.
As before, we can take a bit further the analysis for the 64-worker-thread configuration. In fact, it is noticeable from the queue length plot that the average number of jobs waiting does not increase drastically for this configuration of the system, as it does for the other three. Additionally, looking at the difference between the overall increment in response time measured on the client and on the middleware, this value is confined between 1.95% and 3.94% for the 8, 16 and 32 worker-thread configuration, while it is equal to 15.03% for the 64 worker-thread configuration. This is a useful tool that help us understand that in this last case, the network is playing a limiting role in the performance of the system. This can also be proved since, with 64 worker threads inside the middleware, both at 170 and 220 clients, the throughput is higher than 10k ops/sec 35% of the times. That throughput is approximately half (we have half the number of memtier instances here) the maximum throughput we obtained in Section 2.2 for the read-only experiment, indeed.

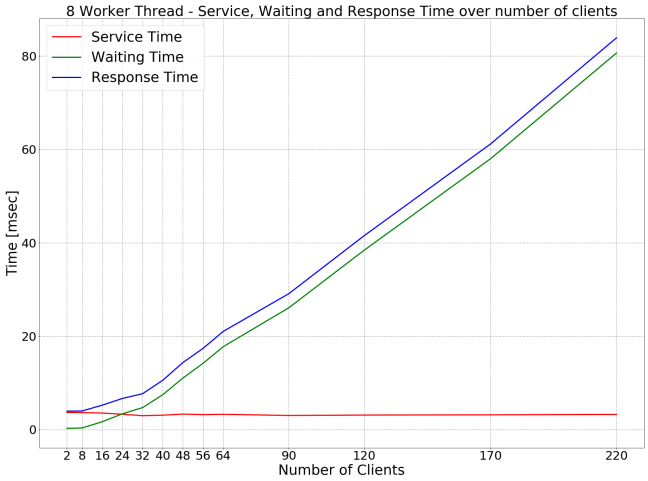

Figure 3.1.10 Times with 8 Worker Threads

Figure 3.1.11 Times with 16 Worker Threads

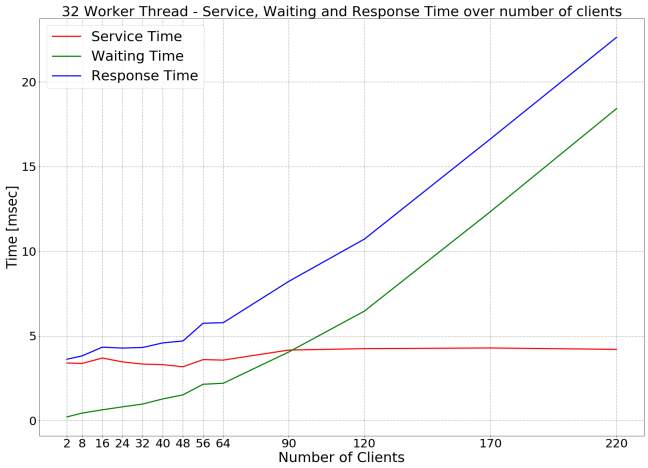

Figure 3.1.12 Times with 32 Worker Threads

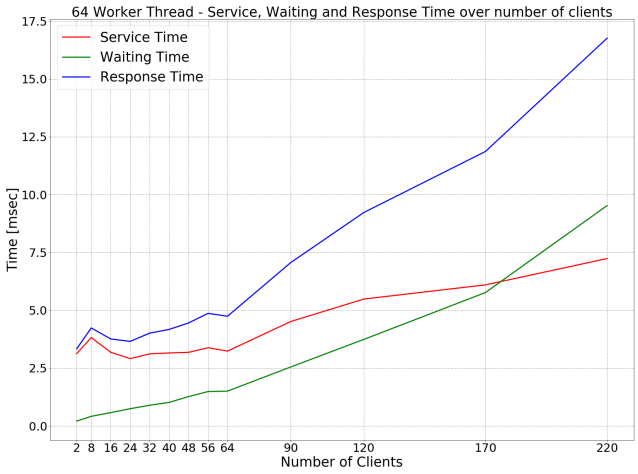

Figure 3.1.13 Times with 64 Worker Threads

Another reason why the throughput is higher for writes than reads, can be explained looking at the average service time for the two cases. In Figure 3.1.14 we show the service time with 32 and 64 worker threads, for the 5 highest numbers of clients, that is the meaningful area to understand the limits of the system. It is clear that the server is, in fact, faster when processing Set requests than Get requests.

To conclude, we can confirm the deductions we made before, stating that the middleware is the bottleneck of the system for all the configurations of worker threads, even though, also the network upper-bound is often reached for the 64 worker thread configuration. The smaller figures of the throughput for the read-only experiment, if compared with the write-only one, can also be explained looking at the intrinsic behavior of the server.

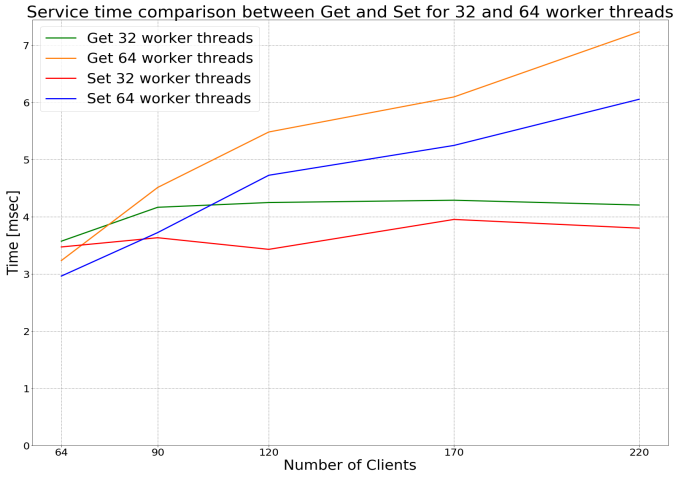

Figure 3.1.14 Service Time comparison Set-Get

3.2 Two Middlewares

Number of servers	1
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1..110]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8..64]
Repetitions	3 (80 sec each)

3.2.1 Explanation

In this section we connect one load generator machine (two instances of memtier with CT=1) to two middlewares and we use 1 memcached server. We will consider the changes that take place for throughput and response time when changing the number of clients and the number of worker threads inside the middlewares and we will analyze the reasons for the system saturation in the different cases.
Starting with the 8 and 16 worker threads configurations, we can see from Figure 3.2.2 that the system saturates at `9632.65 ops/sec` and at `11574.57 ops/sec` respectively, and we can be sure that it is the middleware that is limiting the performance of the system in both

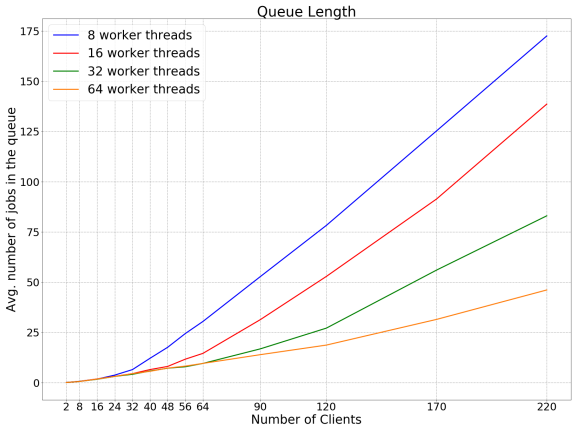

Figure 3.2.1 Plot of the Queue Length

cases. As a matter of fact, if we look at the time in service and the time waiting in the queue (Figures 3.2.4, 3.2.5) we can clearly see that the waiting time becomes higher than the service time at a certain point, and it contributes for the most part to the drastic increment in response time. It is also noticeable that the queue length (Figure 3.2.1) increases very steeply for these two configurations of worker threads, as a result of the saturation of the middleware. This means that the clients generate and send much more workload than the middleware is actually able to serve.
The situation is very similar with 32 worker threads. In fact, we can see in the plot of the throughput that, from 170 to 220 clients, this goes from `12610.09 ops/sec` to `12395.7 ops/sec`, so we can assume it to be stable. Moreover, following the same reasoning procedure as before, from Figure 3.2.6 it is noticeable that, from 170 clients upwards, the waiting time gets higher than the service time, and the time spent in the server is becoming stable. This leads us to the conclusion that the server is actually able to serve that workload without saturating, while the worker threads are the cause of the limit in the performance of the system. Even though a steep increase in the average queue length (Figure 3.2.1) has not yet taken place at 220 clients, we can be confident that, if the clients continue generating that workload, the inflection point in queue length is inevitable.

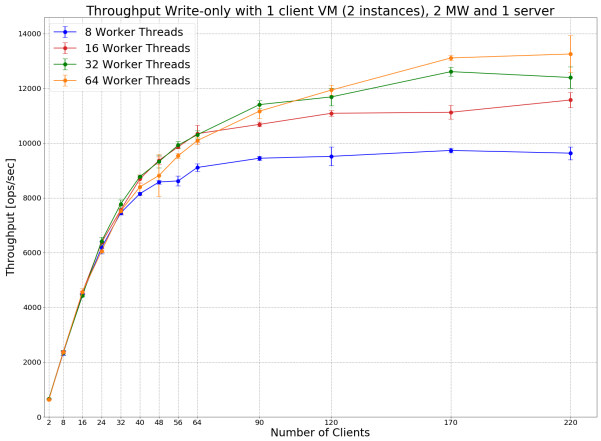

Figure 3.2.2 Plot of the Throughput

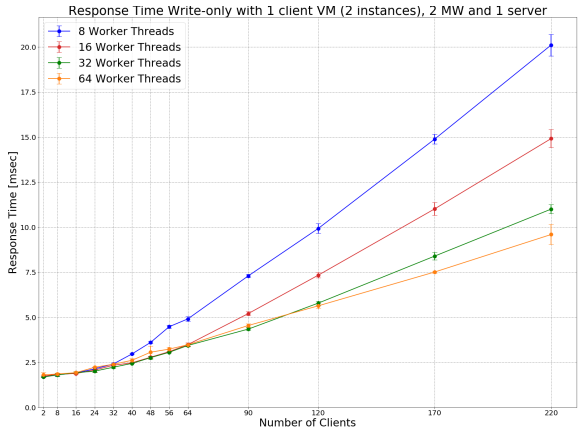

Figure 3.2.3 Plot of the Response Time

As for the 64-worker-thread configuration, contrarily to what happens for the other three, we cannot yet see the waiting time becoming higher than the service time. Although, we can notice in Figure 3.2.7 that between the last two configurations of virtual clients, the waiting time steepness is increasing while the steepness for the service time is decreasing. If we also consider that the server has been proved to be able to handle up to `36456.626 ops/sec` (from Section 2.1), we can deduce that it cannot be the cause of the saturation of the system. What we can expect is that the waiting time will eventually, yet slowly, become higher than the service time. To have a further proof that this speculation is what would really happen, I tried to repeat this experiment, but using two load generator VMs (with two memtier instances each) instead of one. The result is that, indeed, the throughput does not get much higher than what we obtained in the experiment with one client VM. Additionally, going up to 340 clients (I thought it was enough to see the intersection between the two, but it was not) the waiting time is almost as high as the service time, and we see that the steepness of the waiting time is increasing while the one for the service time is decreasing (Figure 3.2.8).

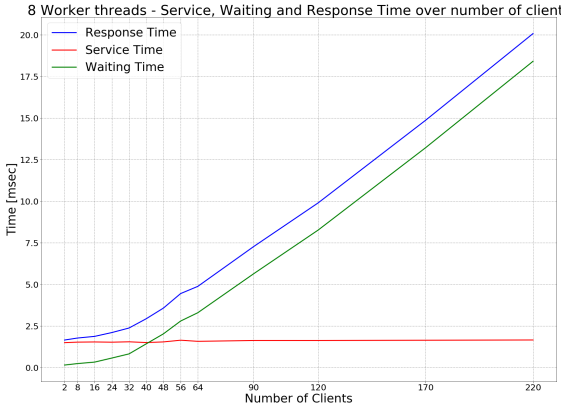

Figure 3.2.4 Times with 8 Worker Threads

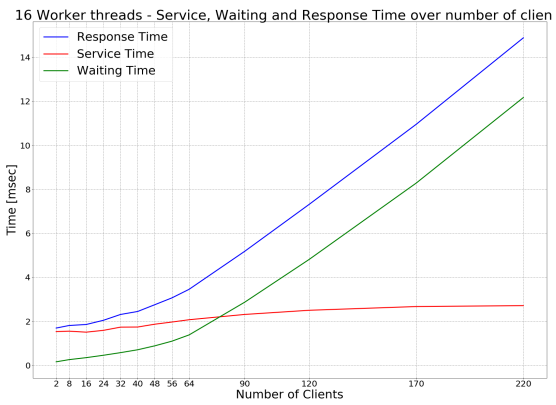

Figure 3.2.5 Times with 16 Worker Threads

This proves that, even though the server is certainly impacting on the performance of the system, we cannot consider it to be the bottleneck of the system, and neither is the client, otherwise we would have doubled the throughput with twice the number of VMs.
In conclusion, the main contribution to response time is the service time up to this point, but the saturation of the system is due to the faster increment of the waiting time over the service time. Therefore, we can deduce that eventually the worker threads inside the middleware will not be able to serve the workload generated by the clients, and this is why the performance of the system are bound.

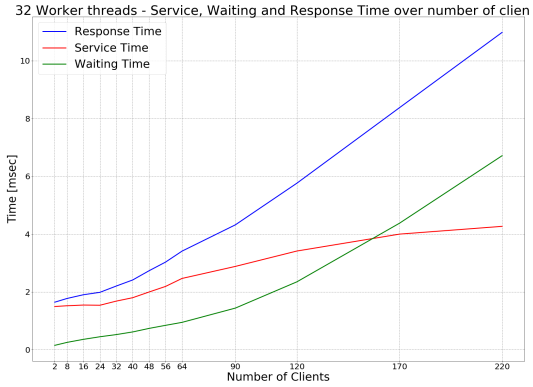

Figure 3.2.6 Times with 32 Worker Threads

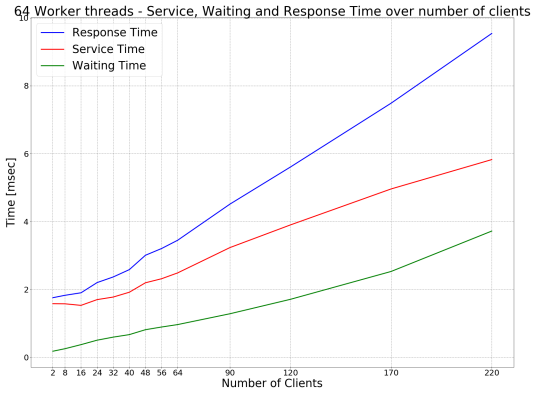

Figure 3.2.7 Times with 64 Worker Threads

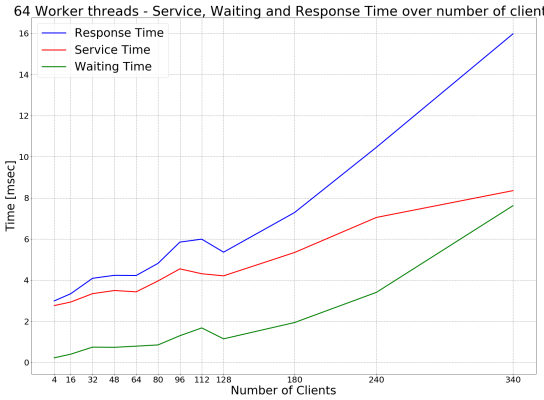

Figure 3.2.8 Times with 64 Worker Threads for the experiment with 2 client VMs

The situation is almost identical for the read-only experiment. In fact, for the 8 and 16 worker-threads configurations, we can see in Figure 3.2.14 that the system gets saturated at `6636.176 ops/sec` and `8202.40 ops/sec` respectively. Following the same reasoning as before, we can now analyze the service and waiting time, and look at the trend of the average number of jobs in the queue. It is noticeable in Figures 3.2.9 and 3.2.10 that, when the waiting times becomes higher than the service time we can see that the throughput starts to plateau and the queue length increases at a faster pace. As a consequence, we can deduce that, in these two cases, the middleware is certainly the bottleneck of the system.

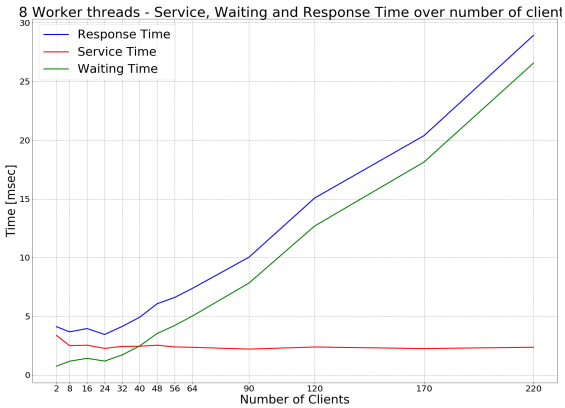

Figure 3.2.9 Times with 8 Worker Threads

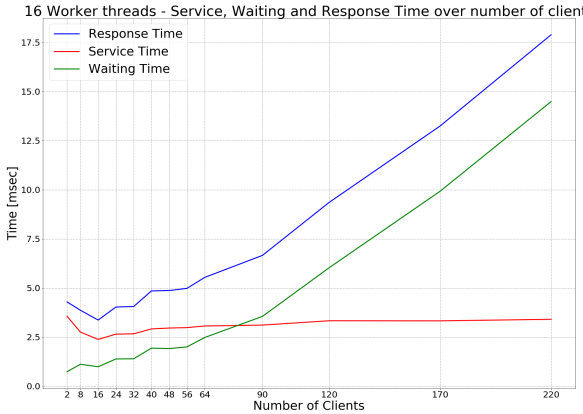

Figure 3.2.10 Times with 16 Worker Threads

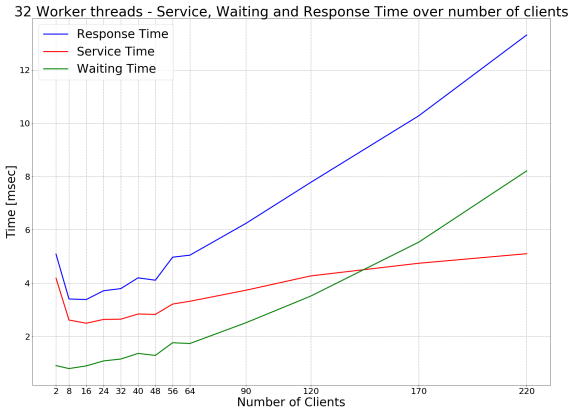

Figure 3.2.11 Times with 32 Worker Threads

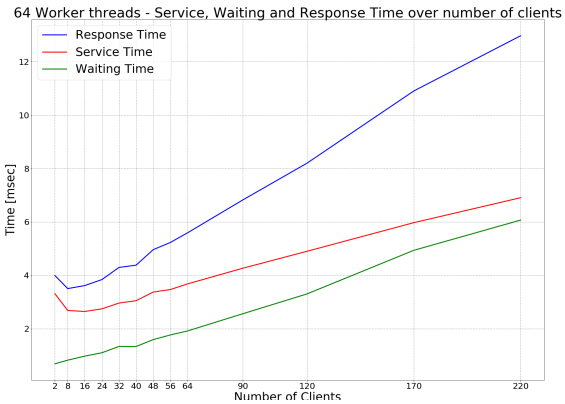

Figure 3.2.12 Times with 64 Worker Threads

The same applies for the 32-worker-thread configuration (Figure 3.2.11), since the waiting time gets higher than the service time between 120 and 170 clients and, as a matter of facts, from that point onwards, the throughput remains stable and the steepness of the queue length (Figure 3.2.13) increases almost like it does for 8 and 16 worker threads.

Unlike what happens for the just discussed configurations, with 64 worker threads inside the middleware, we cannot see the system saturating completely. As a matter of fact, the throughput is still increasing at 220 clients, the queue length (Figure 3.2.13) has not yet experienced an inflection point and the waiting time is still smaller than the service time, even though they are quite close. Just like for the write-only experiment, we can assume that the middleware will eventually saturate if we keep increasing the workload, and so it can be considered as the bottleneck for the system once again. It is also noteworthy that, as we explained and proved in Section 3.1, the throughput for the read-only experiment is overall lower than the one for the write-only, since it takes longer for the server to process get requests than set requests.

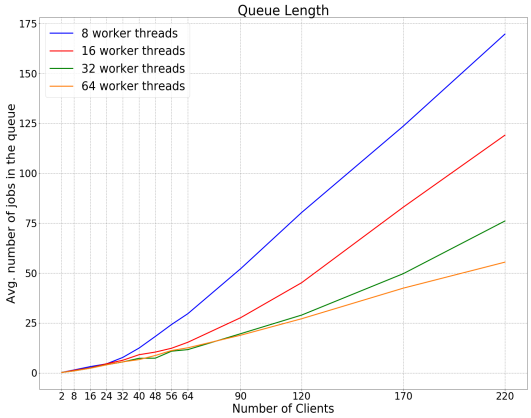

Figure 3.2.13 Plot of the Queue Length

Another interesting point to outline is that, the throughput increment we obtain going from one middleware to two, is much higher for the configurations with smaller number of worker threads. In fact, the throughput for 8 worker threads approximately triplicates, the one for 16 worker threads more than duplicates, while for 32 and (yet more) for 64 worker threads it increases by a much lower factor. The cause of this is the fact that the server needs to manage, read and write to a higher number of connections, since in our implementation of the system we have one socket for each worker thread-server pair, so part of its computational power is being utilized for this activity.

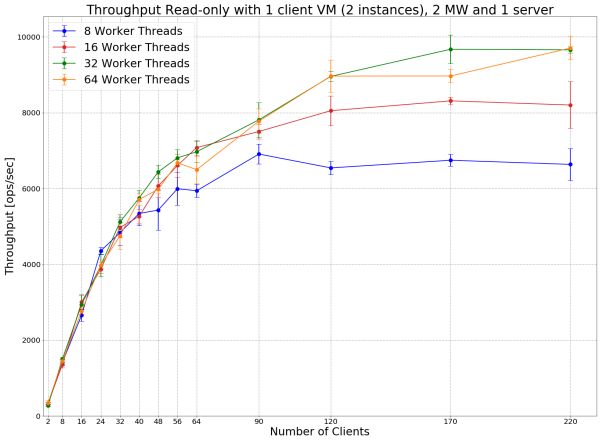

Figure 3.2.14 Plot of the Throughput

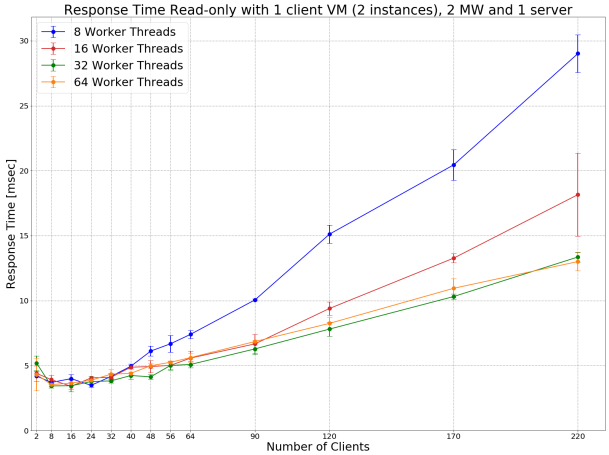

Figure 3.2.15 Plot of the Response Time

3.3 Summary

Maximum throughput for one middleware

	Throughput	Response Time	Avg. Time in Queue	Miss Rate
Reads: Measured on middleware	8442.41	16.83141	9.528035	0.0
Reads: Measured on clients	8473.543	26.11934	n/a	0.0
Writes: Measured on middleware	9701.927	13.6553	7.513028	n/a
Writes: Measured on clients	9727.67	22.78866	n/a	n/a

Maximum throughput for two middlewares

	Throughput	Response Time	Avg. Time in Queue	Miss Rate
Reads: Measured on middleware	9706.4	13.0062	6.067193	0.0
Reads: Measured on clients	9967.33	22.0995	n/a	0.0
Writes: Measured on middleware	13254.72	9.59670	3.717939	n/a
Writes: Measured on clients	13065.29	16.9531	n/a	n/a

Based on the summarizing results reported in the two tables above, it is clear that the middleware is limiting the performance of the system, if compared with the implementation analyzed in Section 2. This difference in performance is especially visible with small number of worker threads, as this is the component than cannot keep up with the workload as soon as the number of clients increases.
Although, considering the results obtained in these two sub-sections, it is clearly visible that in both cases the throughput for the write-only experiment is higher than the one for read-only, and this gap increases the more middlewares we have in our system. Consequently, also the response time and the average waiting time when handling Set requests are smaller than the respective values for Get requests. Moreover, it is noteworthy that the throughput measured on the middleware and on the clients are very similar, but the response time is on average 8.72 ms higher on the latter. As we have already pointed out, this reflects the impact that the network has, especially with this configuration of worker threads (64), when sending the message between clients and middleware.
Lastly, we can highlight the fact that the throughput does not doubles as we add a second middleware to the system. Therefore, even though we have twice the number of worker threads that could potentially double the throughput of the system, in both the implementations, we have one server, that in this case has to manage twice the number of connections. This activity is costly and, as a consequence, we don’t see a linear increase in performance.

4 Throughput for Writes

4.1 Full System

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1..40]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8..64]
Repetitions	3 (80 sec each)

4.1.1 Explanation

As previously explained in the description of the middleware implementation, when a worker threads polls a request from the queue, in case of a Set, it firstly replicates the message to all the servers and in a following loop it fetches the responses from them. Since in this section we are running write-only experiments, we then expect that the performance of the system will decrease compared to the system we analyzed in Section 3.2 where there was only one server.

As a matter of fact, the data reveals that the response time is always higher than the one in the write-only experiment of Section 3.2. The main reason for this increase must be sought in the analysis of how the service time changes from that configuration with one server. In fact, service time is higher than in the previous section for every configuration of the system (i.e. both for worker threads and number of clients), as we can see in Figure 4.1.1 (the lines for Section 4 are the ones above those of the same color for Section 3.2). It is also interesting to outline that this increment is almost completely independent from the number of worker threads, since this value is confined between 1.701x and 1.892x for every configuration of the middleware. In fact, we will see in the 2K Analysis (Section 6) that the effect of interaction between servers and worker threads is equal to 0.297%, meaning that is almost negligible with respect to the impact of the other factors.

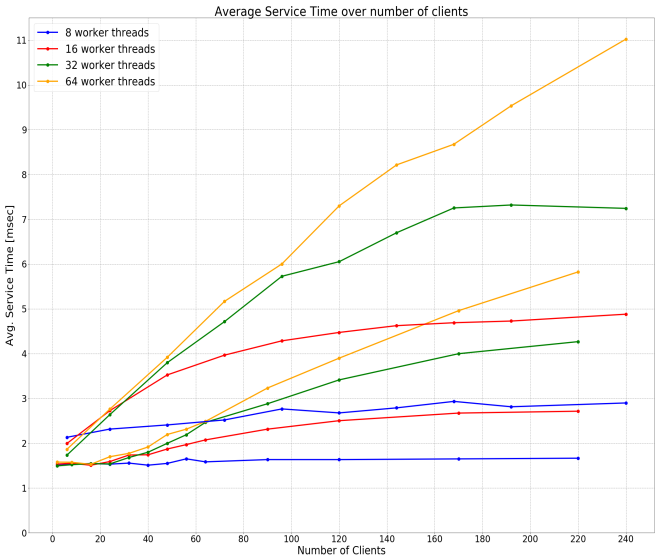

Figure 4.1.1 Service Time comparison with Section 3.2

We can notice that the increment in the service time is not exactly equal to 3 because, as I implemented the middleware, it has to go through a loop for every `ServerHandler` in the `ArrayList` that contains them, when sending the requests. In the base case, the length of the list would be 1, but we can see this procedure of entering the loop and accessing the list as a fixed cost that needs to be considered regardless of the number (>=1) of servers. The same "overhead" is present when retrieving the replies from the `ServerHandlers` in the list, and storing them in an array. As a consequence, it is reasonable that the performance decreases by less than a factor of 3.

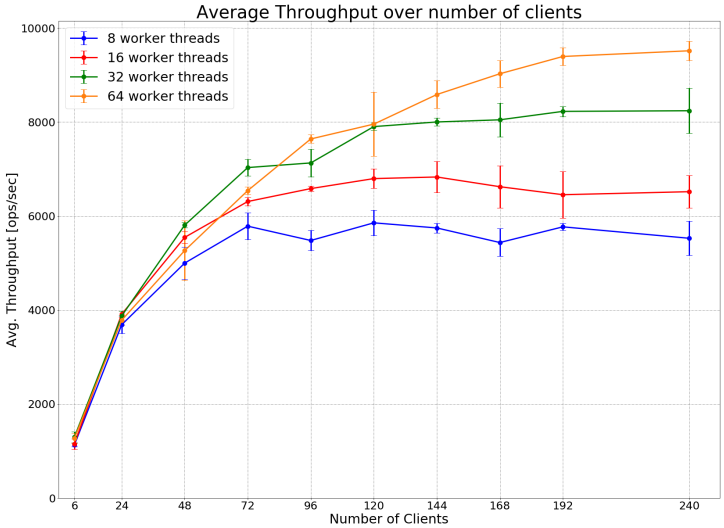

Figure 4.1.2 Plot of the Throughput

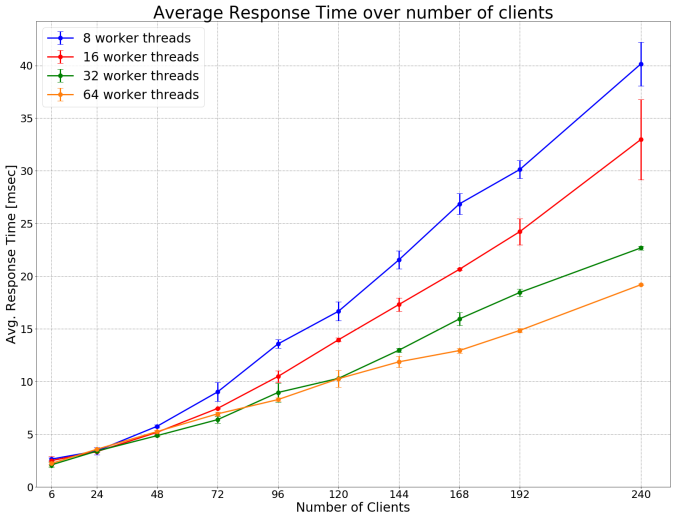

Figure 4.1.3 Plot of the Response Time

We can now make an analysis of the behavior of the system and study how it changes with different configurations of the middleware and number of clients. As well as we approach the discussion of the results in Section 3.2, here we will make a distinction between the 8, 16 and 32 worker threads configurations and the one with 64 worker threads. This is due to the fact that, for the last case, we do not yet see a steep increase in the average number of jobs in the queue, and the waiting time is still lower than the service time at 240 clients. In fact, from Figure 4.1.2 we can see that, for 8 worker threads the system has already saturated at 72 clients and from that point onwards, the throughput remains stable at an average of `5656.716 ops/sec`, peaking at `5854.856 ops/sec` for 120 clients. Moreover the waiting time is almost always higher than the service time (Figure 4.1.5). As for the configuration of the middleware with 16 worker threads inside, the results present an average throughput of `6634.160 ops/sec` where the system is saturated.

We can notice in Figure 4.1.4 that the queue length shows the first of its two inflection points for 96 clients, that happens right after the intersection between the waiting time and response time (Figure 4.1.6). This is also where we can see the throughput starting to plateau, and we can consider the system to be saturated, since the middleware cannot handle more than that workload per second.
The situation for 32 worker threads is very similar to the previous ones, with the system saturating at 120 clients, because of the "knee" in waiting time we can observe in Figure 4.1.7 for that number of clients. This corresponds to the inflection point in queue length, after which we can see the waiting time becoming higher than the service time and shaping the increasing trend of the response time.

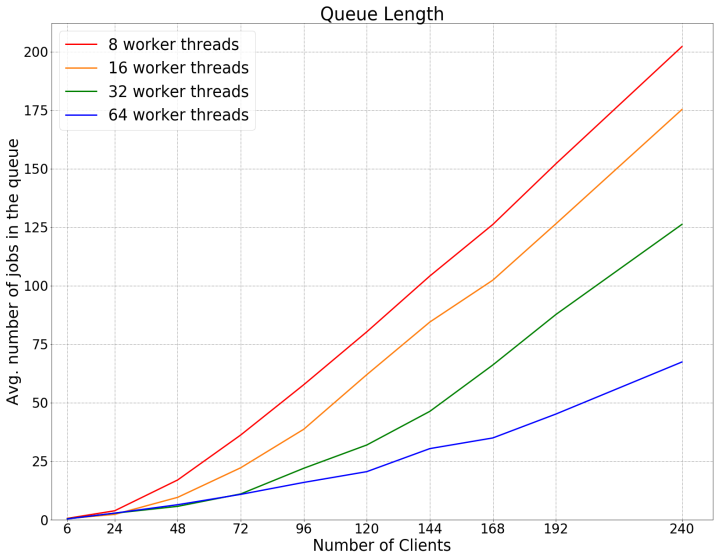

Figure 4.1.4 Plot of the Queue Length

Lastly, we can observe in Figure 4.1.8 that, as already mentioned, with 64 worker threads inside the middleware, the waiting time is still smaller than the service time. Although, each memcached server is processing at most `9515.587 ops/sec` with 240 clients, and we have seen in Section 3.2 that, with the same number of active connections (in both cases we have 128 open sockets per server), the server could handle more than 14k ops/sec. Therefore we can be sure that the servers are not the bottleneck of the system. Moreover, it can be seen that, for 168 clients, the waiting time steepness is increasing, and so is the response time. Following the same reasoning as in Section 3.2, and considering the arguments on the message replication we presented above, we can deduce that the waiting time will eventually get higher than the service time and the middleware can be considered the bottleneck of the system.

Figure 4.1.5 Times with 8 Worker Threads

Figure 4.1.6 Times with 8 Worker Threads

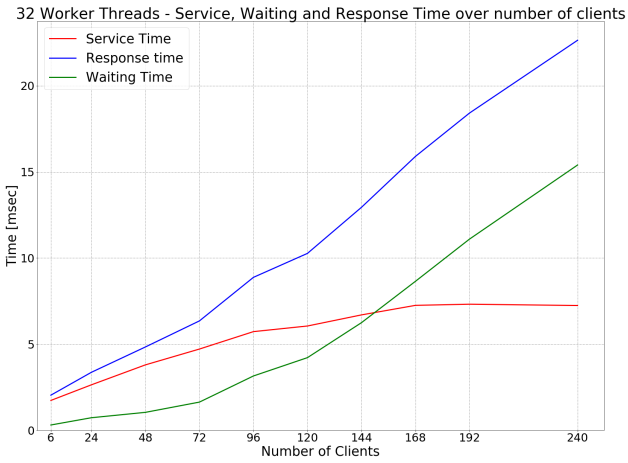

Figure 4.1.7 Times with 32 Worker Threads

Figure 4.1.8 Times with 64 Worker Threads

4.2 Summary

	WT = 8	WT = 16	WT = 32	WT = 64
Throughput (Middleware)	5854.85 (120)	6830.50 (144)	8239.93 (240)	9515.58 (240)
Throughput (Derived from MW response time)	7198.07 (120)	8317.84 (144)	10582.5 (240)	12508.6 (240)
Throughput (Client)	6076.22 (120)	6879.61 (144)	8388.06 (240)	9659.06 (240)
Average time in queue	13.9326 (120)	12.6185 (144)	15.4002 (240)	8.13032 (240)
Average length of queue	80.3036 (120)	84.4806 (144)	126.220 (240)	67.4134 (240)
Average time waiting for memcached	2.56667 (120)	4.43878 (144)	7.05388 (240)	10.7751 (240)

[In the table above every time is in ms, every throughput in ops/sec. In each cell, inside the parentheses is the configuration of clients that has been considered to fill that cell.]

From these aggregated results, we can clearly see that, overall, the system performs better with the highest number of worker threads inside the middleware and, in general, the throughput measured on the clients is very similar, slightly higher, than the data collected through the instrumentation of the middleware. On the contrary, the throughput derived from the middleware response time is much higher than the real one, meaning that the fraction between the increase in number of clients and increase in response time is smaller than 1. This translate to a decrease in the throughput derived from the response time as soon as the waiting time becomes higher than the service time and so the main contribution to response time. As a matter of fact, we can see this trend clearly for 8 and 16 worker threads in Figure 4.1.9.
Regarding the queue length, we can say that for 8, 16 and 32 worker threads, this is where there is the biggest proportion of jobs that are in the system. In fact, there can be at most twice the reported number of worker thread jobs that are being served in a certain moment in time, and for every one of these configuration, this number is lower than the average number of jobs in the queue. On the contrary, for 64 worker thread there are presumably more jobs in service than waiting, and, in fact, we can achieve higher performance and the waiting time is indeed smaller than the service time at 240 clients.
As for the two times reported in the table, since we are analyzing a point in which the system has already saturated, it is noticeable that the average time in the queue is higher than the average time waiting for memcached. As we already pointed out, though, this time spent in the server is heavily influenced by the velocity of the worker thread to retrieve all the responses after having sent all the requests. Thus, the actual time needed for the memcached servers to process the request might be even smaller. This is of particular importance when considering the case with 64 worker threads, in which the reported service time is higher than the waiting time.

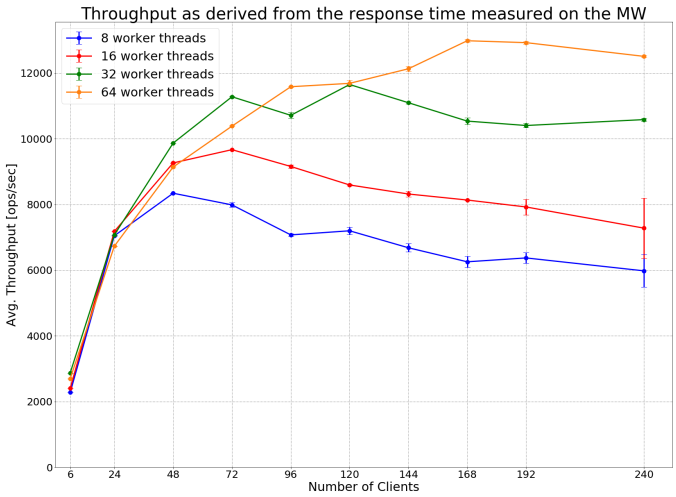

Figure 4.1.9 Throughput derived from Response Time MW

5 Gets and Multi-gets

5.1 Sharded Case

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	2
Workload	`1 Set every Multi-Get`
Multi-Get behavior	Sharded
Multi-Get size	[1..9]
Number of middlewares	2
Worker threads per middleware	64
Repetitions	3 (80 sec each)

5.1.1 Explanation

Even though we have seen that for a total number of 12 virtual clients there is very little difference in performance between the four configurations of worker threads inside the middleware, I decided to equip the system with 64 worker threads to be sure that the performance would not have been much affected by the waiting time.
As for the workload generation, I ran the experiments with the `–ratio=1:$numberOfKeys` when `–multi-key-get=$numberOfKeys`, so that for every multi-get request there is one Set request. Moreover, by doing so, the clients only generate multi-get requests that exactly contains the specified number of keys, not fewer.
If we consider that the server has been proved to be slower when processing Get requests than Set requests, and that in this Section we are generating gets with multiple keys inside, we can expect that, overall, the system will perform worse than in Section 4. Moreover, since we are populating the server before running the experiments so that there are no cache misses, every multi-get request will have a payload of `1024*numberOfKeys` Byte, this being the main cause of the decrease in performance.
From Figures 5.1.1 and 5.1.2, we can see that the throughput of the system with sharding enabled is overall lower than non-sharded one. This is true for every number of keys inside the multi-get requests except for 1, when the throughputs are equal because of course the two types of experiment in this case do not differ from

one another. In this scenario, the clients generates simple get requests so we can prove the consistency of our results showing that, in fact, these measurements are very similar to the ones the read-only experiment we obtained in Section 3.2 where the throughput for 12 clients was approximately 2k ops/sec.

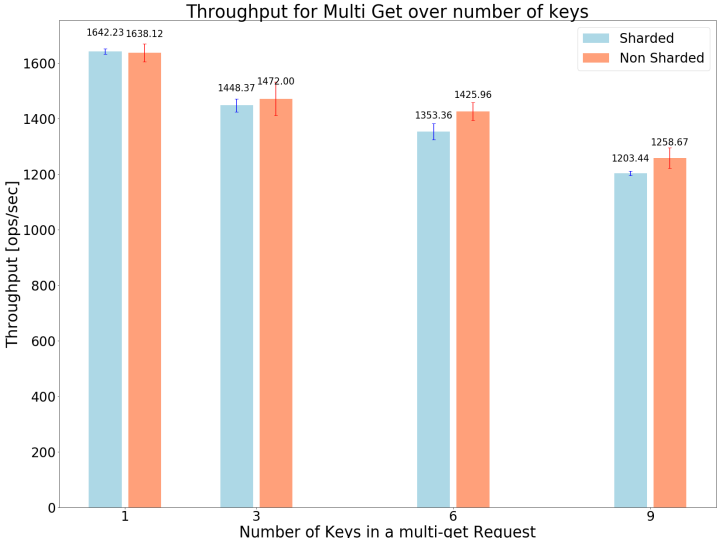

Figure 5.1.1 Plot of the Throughput

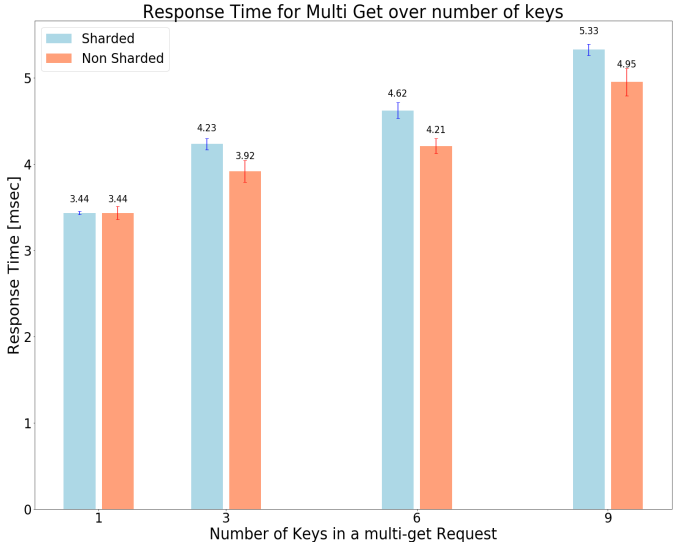

Figure 5.1.2 Plot of the Response Time

Looking at Figure 5.1.3, we can see that, as we would expect, the response time increases with the number of keys inside the request, and so does the service time. In fact, service time is by far the biggest portion of response time, since it is always more than 6 times bigger than the waiting time. Moreover, as we can see in Figure 5.1.4, there is always approximately 0.7 jobs in the queue, as there are fewer clients than worker threads. As explained in the description of the middleware implementation, the service time is the time from when the worker thread takes the request out of the queue until the moment it sends the reply back to the client. To prove that it is not the server that is limiting the performance of the system but the worker threads that have to shard the message and retrieve all the responses, we can notice that the increment in response time is sub-linear in the number of keys. As a matter of fact, if the server was the limiting factor in performance, we would see the service time increasing by a factor approximately equal to the the increment in number of keys. On the contrary, the increment is a smaller portion of the total service time, meaning that the servers can serve the requests faster than the middlewares can send and retrieve them all. This leads to the conclusion that main contribution to the response time is, indeed, the time spent in the middleware both when parsing, sharding and sending the requests and when collecting and aggregating all the responses in the final reply to send to the client.

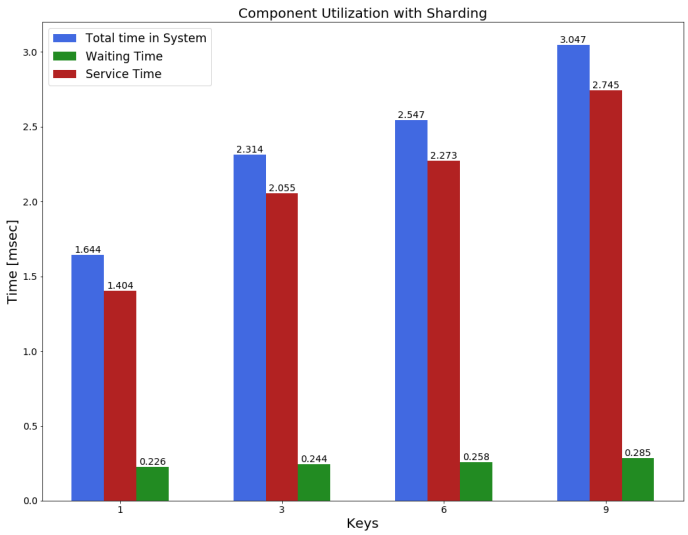

Figure 5.1.3 Times without Sharding

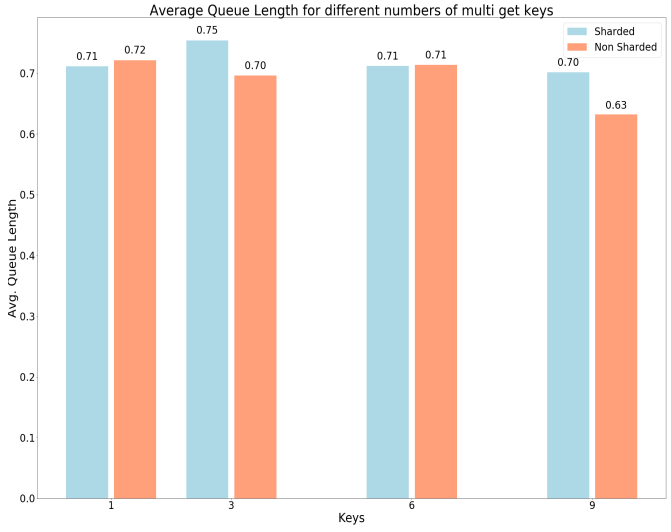

Figure 5.1.4 Plot of the queue length

In addition to that, we can state that in the non-sharded mode the two active servers in a specific moment in time (or one if both the middleware send the request to the same server), they receive 3/2 (or 3, in the unlucky case) the number of requests each server receives in sharded mode. As a consequence, if it was the servers that are limiting the performance of the system we would have seen the throughput for non-sharded being lower than the one for sharded, that is not the case.

5.2 Non-Sharded Case

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	2
Workload	`1 Set every Multi-Get`
Multi-Get behavior	Non-Sharded
Multi-Get size	[1..9]
Number of middlewares	2
Worker threads per middleware	64
Repetitions	3 (80 sec each)

5.2.1 Explanation

Turning now to the non-sharded case, we can observe that the situation is similar to the one we just analyzed, although the overall system performance are better. We can see that, for every number of keys inside the multi-get requests, the throughput in the non-sharded case is higher than the one with sharding enabled, except for the simple Get requests where the two approaches perform equally good. Looking at where the requests spend on average their time in the system (Figure 5.2.1), it is noticeable that, just like before, the service time is by far the main portion of the response time. In fact, the waiting time is always confined between 0.232 and 0.261 ms and the queue length tops at 0.72 jobs. Moreover, contrarily to the service time, the waiting time does not increase consistently, since there are fewer clients than worker threads.
Following the same reasoning as before, we can deduce that the main contributions of the service time are the time spent in the middleware and the time in the network.

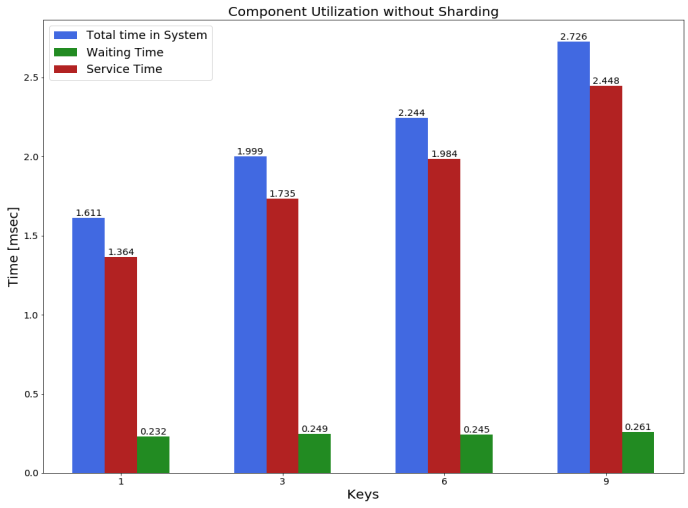

Figure 5.2.1 Times without Sharding

5.3 Histograms

It is clearly noticeable from the following figures that the distribution of the response time seems to present two patterns, one for the results obtained by the middleware and the other for the data measured by the clients. As a matter of fact, both in sharded and in the non-sharded case, the histogram representing the response time distribution as measured on the middleware has just one very high spike while, the histogram for the data collected by the clients shows two peaks, with one being considerably higher than the other.
The reason why this happens is that, as we can see from the memtier output, every client VM might experience a lower or higher latency in the response time depending on the middleware they are interacting with. As a consequence, a smaller portion of requests might have longer response time than others, resulting in a secondary peak that of course is just barely visible, at approximately 5 ms, from the measurements logged inside the middlewares.
Moreover, we can notice that, due to this repartition of the requests in two peaks for the response time distribution inside the clients, the first peak tops at half the percentage of the one reached by the corresponding measurement on the middleware. For instance, in the non-sharded case, when on the middleware we measure a maximum of 20% requests in a 0.25 ms time interval (that is the bucket size), the corresponding peak on the the response time distribution measured on the client tops at approximately 10%.
Aside from this difference in shape between the two measurement points, it is noteworthy that sharded and non-sharded response time distribution are very similar one to the other (inside a measurement approach), with the former being shifted by just over 0.25 ms to the right, because, as we have already seen, performance are slightly worse when sharding is enabled.

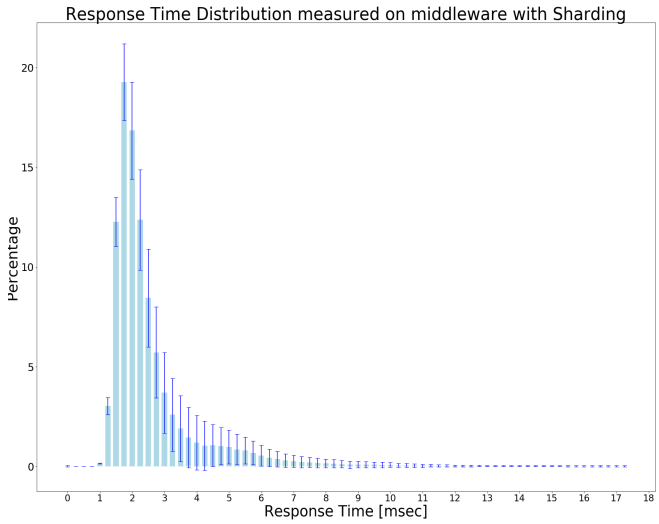

Figure 5.3.1 Resp. Time Distr. MW Sharded

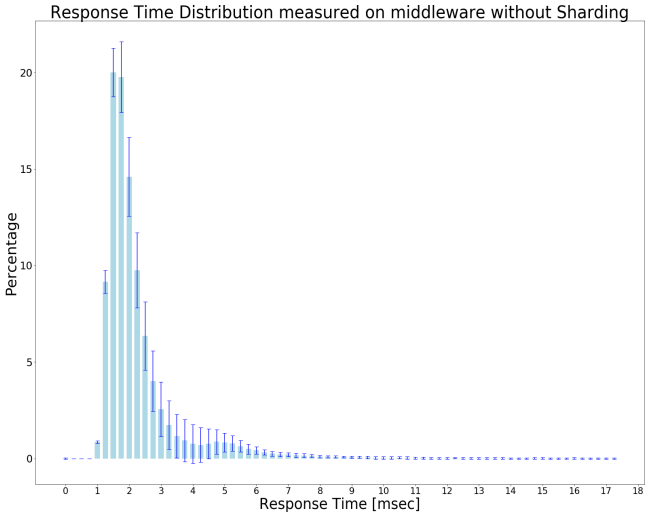

Figure 5.3.2 Resp. Time Distr. MW Non-Sharded

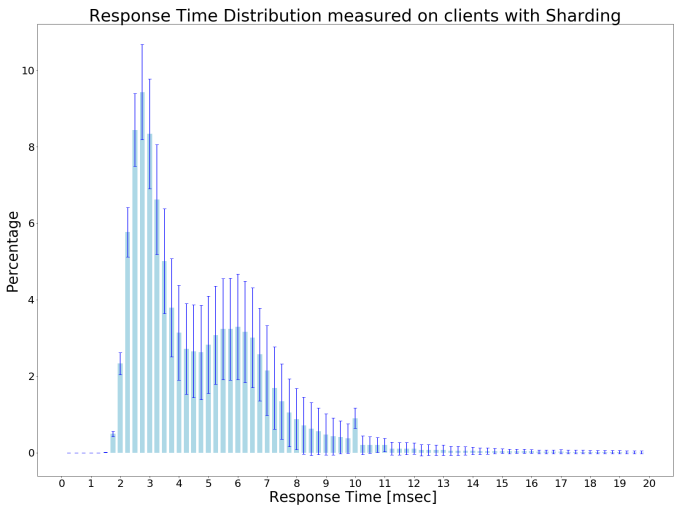

Figure 5.3.3 Resp. Time Distr. Client Sharded

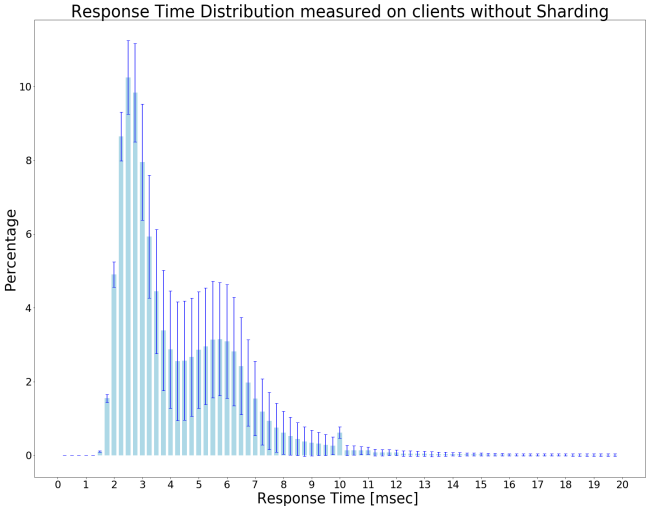

Figure 5.3.4 Resp. Time Distr. Client Non-Sharded

5.4 Summary

Overall, we have seen that without sharding, we can expect better performance from our system as long as we have more than a single key in our multi-get requests, otherwise the two approaches are equally good. We have seen in Figures 5.1.1 and 5.1.2 that the difference in throughput and response time between the two cases is approximately constant across the different number of keys (except for simple Get requests), although, the gap between the two tend to widen with higher number of keys, if we consider the percentiles. In fact, looking at Figure 5.4.1, it is clearly noticeable that the difference in response time between the two increases with the percentile threshold we are considering. Apart from the case with just one key, indeed, the average difference is 0.202 ms for the 25th percentile, 0.397 ms for the 50th percentile, 0.504 ms for the 90th percentile and 0.816 ms for the 99th percentile. This means that, in the Sharded case, we have a tail in the time distribution of requests that is heavier than the one without sharding.
In conclusion, the results have proved that, when the multi-get requests contain such numbers of keys, sharding is never to be preferred, except when multi-get size is equal to 1, where, with very similar performance between the two approaches, we could prefer to spread the workload that comes with each request, over the highest possible number of available servers.

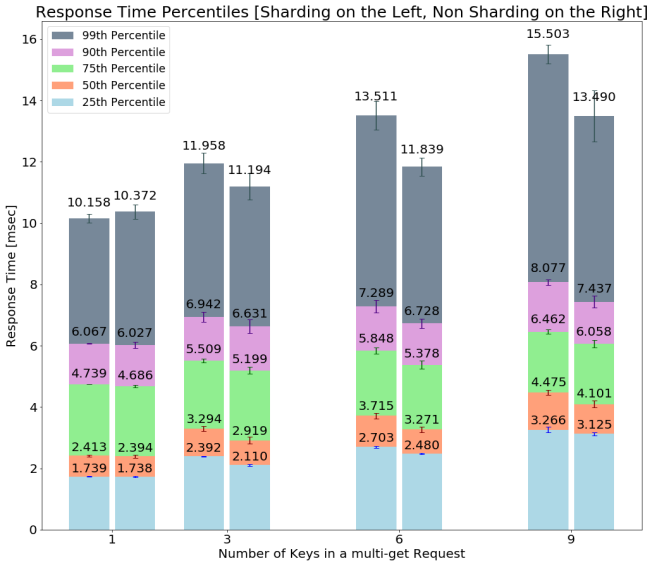

Figure 5.4.1 Response Time Percentiles

6 2K Analysis

Number of servers	2 and 3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	32
Workload	Write-only, Read-only, and 50-50-read-write
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1 and 2
Worker threads per middleware	8 and 32
Repetitions	3 (80 sec each)

In this section we will analyze the effect that different factors have on the performance of the system. In particular we will estimate the contribution made by the number of middlewares, of worker threads and servers, on the performance. We will repeat the experiment for a write-only, a the read-only and a 50-50-read-write workload, replicating each experiment 3 times.

We are going to define three variables:

$$S=\begin{cases}-1 & \text{if number of servers = 2}\\ +1 & \text{if number of servers = 3}\end{cases} \qquad M=\begin{cases}-1 & \text{if number of middlewares = 1}\\ +1 & \text{if number of middlewares = 2}\end{cases} \qquad W=\begin{cases}-1 & \text{if number of worker threads = 8}\\ +1 & \text{if number of worker threads = 32}\end{cases}$$

We identify with y the variable (alternatively throughput and response time) on which we want to analyze the effect of each factor. We also define as q_x the effect of a factor x and as SSX the sum of squares due to x.
In order to calculate the effects of the factors, for every one of the 8 combinations of the factors' levels, we have to solve the set of equations representing the model, given by:

$$y = q_0 + q_S x_S + q_M x_M + q_W x_W + q_{SM} x_{SM} + q_{SW} x_{SW} + q_{MW} x_{MW} + q_{SMW} x_{SMW} + e$$

where e is the experimental error.
The easiest way to analyze a 2^3r design (with r being the number of repetitions) is to use the sign table method, where the last column shows the average throughput (or response time) of the three replications. In the smaller table that follows, I show the single values of the three repetition j for the experiment i, the mean throughput of the three replications and the squared experimental errors e_{ij}^2 .
For every experiment i, we compute the mean throughput $\bar{y}_i$ through the formula:

$$\bar{y}_i = \frac{1}{3}\sum_{j=1}^{r} y_{ij}$$

and the experimental errors as:

$$e_{ij} = \bar{y}_i - y_{ij}$$

The first 8 columns are the sign columns, where the entries in each of them are multiplied by those in the column MEAN TPS, and the sum divided by 8 is entered under the column.
These values are the effects q_x of the factors.
Once the effects have been computed in a 2^3r design, we can calculate the variation explained by each factor, indicated with SSX for the sum of square of factor X, as:

$$SSX = 2^3 \, r \, q_X^2$$

We also compute the sum of the squared errors:

$$\sum_{i=1}^{2^3}\sum_{j=1}^{r} e_{ij}^2$$

so that we can now derive the Total Sum of Squares (SST) as:

$$SST = \left(\sum_{X \in \boldsymbol{F}} SSX\right) + SSE \qquad \text{where} \quad \boldsymbol{F} \quad \text{is set of all the factors.}$$

We can find the sum of square for each factor and the SST on the second-last row of the table.

Lastly, we can compute the percentage of the impact of each factor x on the throughput as SSX/SST , that we reported on the last row of the table.

Our factors are multiplicative, so we should solve the problem with a multiplicative approach; however, we can notice that the maximum ratio across all the experiments $\frac{y_{max}}{y_{min}} = 4.61$ is not very large (in comparison, the book suggests a multiplicative approach in an example where the ratio is 12534). With such a small ratio, the log function that would results out of a multiplicative model can be approximated by a linear function allowing to use a more simple and immediate additive model with very similar results.

Name: Andrea Rinaldi Legi: 17-941-626

Write-Only Experiment

I	S	M	W	SM	SW	MW	SMW	MEAN TPS
+1	-1	-1	-1	+1	+1	+1	-1	4634.23
+1	-1	-1	+1	+1	-1	-1	+1	7953.47
+1	-1	+1	-1	-1	+1	-1	+1	9318.45
+1	-1	+1	+1	-1	-1	+1	-1	13475.14
+1	+1	-1	-1	-1	-1	+1	+1	2916.543
+1	+1	-1	+1	-1	+1	-1	-1	5973.25
+1	+1	+1	-1	+1	-1	-1	-1	5555.206
+1	+1	+1	+1	+1	+1	+1	+1	11389.496
7651.9	-1193.352	2282.599	2045.866	-268.872	176.8829	451.879	242.516	TOTAL / 8
	34178140.7	125046260.6	100453649.1	1735012.73	750901.589	4900683.8	1411539.15	SST = 268716867.4
	12.719	46.534	37.382	0.6456	0.2794	1.8237	0.5252	

Repetition #1	Repetition #2	Repetition #3	Mean TPS	Squared Err. #1	Squared Err. #2	Squared Err. #3
4568.63	4644.06	4690.02	4634.23	4304.234	96.4978	3111.781
7936.57	7915.97	8007.89	7953.47	285.8353	1406.750	2960.81
9096.79	9423.97	9434.6	9318.45	49134.63	11133.7	13490.04
13621.0	13568.54	13235.91	13475.14	21273.19	8720.446	57234.19
2930.04	2856.45	2963.14	2916.543	182.160	3611.208	2171.249
5881.02	6046.42	5992.31	5973.25	8506.372	5353.848	363.283
5642.37	5566.62	5456.63	5555.206	7597.44	130.264	9717.359
11520.26	11370.19	11278.04	11389.496	17099.04	372.747	12422.58
					SSE = 240679.76	

Table 6.1.1 – 6.1.2 Sign table and Errors table for the impacts of factors on Throughput (Write Only)

We can observe that the number of middlewares in the system is the factor that has the main impact on the throughput with 46.534% of the total variation. After that, the number of worker threads is the most influential effect and, that accounts for 37.282% of the total, and with 1/3 of its impact, the number of servers contributes negatively on the performance of the system. As a matter of fact, as we already discussed this issue between Section 3.2 and Section 4, we have seen that the throughput for the write-only experiment decreased with a higher number of servers due to replication. Thus, we can notice that the q_S is negative. As for the effects results of the interaction between the factors, they are very small compared to the relative effects taken individually (they account at most for 1.8% of the total variation), but we can see that they reflect, at a smaller scale, the impact of their contributors.

It is noticeable that, in fact, we reach the highest throughput values (`13475.14 ops/sec` and `11389.496 ops/sec`) when the number of middleware is equal to 2 and the number of worker threads is 32.

In the following table, we calculate the effects of the factors over the response time. Overall, the results respect the same hierarchy of the factors' importance as before, although, in this case the effect result of the number of middleware in the system is lower than the one for the throughput. In fact, the percentage of variation due to this factor is still the highest one, but it decreases of almost 9 percentage points from the outcomes of the throughput variations. On the other hand, the impact that the number of servers has on the response time rise up to 15.3% and also the intermediate effects experience a large increase. In particular the q_{SW} and q_{MW} now account for 4.9% and 4.58% respectively.

Name: Andrea Rinaldi Legi: 17-941-626

I	S	M	W	SM	SW	MW	SMW	MEAN RT
+1	-1	-1	-1	+1	+1	+1	-1	41.414
+1	-1	-1	+1	+1	-1	-1	+1	24.131
+1	-1	+1	-1	-1	+1	-1	+1	20.608
+1	-1	+1	+1	-1	-1	+1	-1	14.252
+1	+1	-1	-1	-1	-1	+1	+1	65.851
+1	+1	-1	+1	-1	+1	-1	-1	32.142
+1	+1	+1	-1	+1	-1	-1	-1	34.645
+1	+1	+1	+1	+1	+1	+1	+1	16.902
31.2435	6.14178	-9.6412	-9.3863	-1.9701	-3.47675	3.36156	0.62984	TOTAL/ 8
	905.3164	2230.8663	2114.493	93.1580	290.1081	271.202	9.52077	SST = 5920.0278
	15.29243	37.68337	35.71763	1.573607	4.900452	4.58110	0.16082	

Repetition #1	**Repetition #2**	**Repetition #3**	**Mean TPS**	**Squared Err. #1**	**Squared Err. #2**	**Squared Err. #3**
42.019	41.2913	40.931	41.414	0.36649	0.0150	0.2329
24.1863	24.2368	23.97116	24.131	0.00301	0.011106	0.02568
21.105	20.36	20.351	20.608	0.24640	0.05717	0.06619
14.1028	14.1481	14.507	14.252	0.0225	0.01095	0.06485
65.524	67.239	64.79	65.851	0.10696	1.92731	1.12619
32.641	31.749	32.0353	32.142	0.24933	0.15405	0.01141
34.148	34.5725	35.2156	34.645	0.24684	0.00535	0.3249
16.661	17.026	17.019	16.902	0.05810	0.0154	0.0136
					SSE = 5.361933	

Table 6.1.3 – 6.1.4 Sign table and Errors table for the impacts of factors on Response Time (Write Only)

Read-Only Experiment

I	S	M	W	SM	SW	MW	SMW	MEAN TPS
+1	-1	-1	-1	+1	+1	+1	-1	5540.02
+1	-1	-1	+1	+1	-1	-1	+1	9704.87
+1	-1	+1	-1	-1	+1	-1	+1	11880.46
+1	-1	+1	+1	-1	-1	+1	-1	19690.45
+1	+1	-1	-1	-1	-1	+1	+1	4346.8266
+1	+1	-1	+1	-1	+1	-1	-1	9416.5833
+1	+1	+1	-1	+1	-1	-1	-1	9277.41
+1	+1	+1	+1	+1	+1	+1	+1	18935.507
11099.01	-604.9350	3846.94166	3337.8358	-234.565	344.1275	1029.18416	117.900	TOTAL / 8
	8782712.50	355175044.5	267387553.2	1320497.74	2842169.67	25421281.2	333614.5	SST = 662071767
	1.3265	53.6460	40.3864	0.199449	0.4292842	3.839656	0.050389	

Name: Andrea Rinaldi Legi: 17-941-626

Repetition #1	Repetition #2	Repetition #3	Mean TPS	Squared Err. #1	Squared Err. #2	Squared Err. #3
5632.94	5707.37	5279.75	5540.02	8634.126	28006.02	67740.47
9830.38	10091.14	9193.09	9704.87	15752.76	149204.5	261918.7
11846.25	12028.24	11766.91	11880.46	1170.78	21836.95	12895.11
19655.48	19711.23	19704.64	19690.45	1222.90	431.808	201.35
4023.2	4483.04	4534.24	4346.8266	104734.2	18554.07	35123.75
9347.01	9533.32	9369.42	9416.5833	4840.44	13627.44	2224.38
9266.68	9368.91	9196.64	9277.41	115.132	8372.25	6523.79
9266.68	9368.91	9196.64	18935.507	29943.9	4348.96	11469.6
					SSE = 808893.7	

Table 6.1.5 – 6.1.6 Sign table and Errors table for the impacts of factors on Throughput (Read Only)

For the read-only experiment, it is noteworthy that the number of middlewares and worker threads acquire a yet higher importance in determining the performance of the system than before. We can see from the table that more than half (53.64%) of the variation in throughput is due to the increased number of middlewares, and considering the impact of the number of worker threads, they account in total for approximately 94% of the whole throughput variation. In fact, the number of server has a smaller impact on the reads than on the writes since, for Get requests there is no replication. As a consequence, an increasing number of servers still impact negatively on the overall throughput because of the increasing number of connections to manage and the need for the middleware(s) to balance the load appropriately. Although this is a very small, almost negligible impact (1.32%). In fact, as we can see from the table, the maximum throughputs are reached by the system when the number of middleware is equal to 2 and there are 32 worker threads in them. When the number of servers goes from 2 to 3, the mean throughput decreases by 754.953 ops/sec.
The importance of the effects is respected by the intermediate effects, as q_{MW} registers the highest impact among those, accounting for 3.84% of the variation, while the others are all below 0.5%.

The results are once again confirmed in the following table where the effects of the factors on response time are computed. To be more accurate, just like we saw in the write-only experiment, the effect due to the server increases, more than doubling, although its impact is still very small if compared to the results for the number of middlewares and worker threads on the total response time variation. In fact, the smallest response times (9.77577 ms and 10.1522 ms) are the ones with 2 middlewares and 32 worker threads.

I	S	M	W	SM	SW	MW	SMW	MEAN RT
+1	-1	-1	-1	+1	+1	+1	-1	34.6426
+1	-1	-1	+1	+1	-1	-1	+1	19.8266
+1	-1	+1	-1	-1	+1	-1	+1	16.2155
+1	-1	+1	+1	-1	-1	+1	-1	9.77577
+1	+1	-1	-1	-1	-1	+1	+1	44.2909
+1	+1	-1	+1	-1	+1	-1	-1	20.3835
+1	+1	+1	-1	+1	-1	-1	-1	20.6780
+1	+1	+1	+1	+1	+1	+1	+1	10.1522
21.9953	1.880847	-7.79058	-6.96079	-0.67045	-1.647499	2.72004	0.625361	TOTAL / 8
	84.90207	1456.636	1162.862	10.78834	65.142149	177.567	9.385836	SST = 2991.259
	2.838338	48.69642	38.87535	0.360662	2.1777497	5.93619	0.313775	

Repetition #1	Repetition #2	Repetition #3	Mean TPS	Squared Err. #1	Squared Err. #2	Squared Err. #3
34.077	33.643	36.2043	34.6426	0.319915	0.997557	2.447312
19.407	19.019	21.05	19.8266	0.17556	0.652325	1.50471
16.216	16.097	16.325	16.2155	9.00e-06	1.33e-02	1.26e-02
9.7741	9.796	9.7571	9.77577	2.59e-06	4.08e-04	3.46e-04
47.714	42.823	42.33	44.2909	11.7173	2.1524	3.82571
20.5336	20.1326	20.4843	20.3835	0.02253	0.06294	0.01015
20.727	20.512	20.794	20.6780	0.00244	0.02757	0.01359
10.0548	10.1723	10.2296	10.1522	0.00949	0.00040	0.00598
					SSE = 23.97483	

Table 6.1.7 – 6.1.8 Sign table and Errors table for the impacts of factors on Response Time (Read Only)

50% Set - 50% Get

I	S	M	W	SM	SW	MW	SMW	MEAN TPS
+1	-1	-1	-1	+1	+1	+1	-1	5151.13
+1	-1	-1	+1	+1	-1	-1	+1	8441.02
+1	-1	+1	-1	-1	+1	-1	+1	10699.95
+1	-1	+1	+1	-1	-1	+1	-1	16776.07
+1	+1	-1	-1	-1	-1	+1	+1	3667.1
+1	+1	-1	+1	-1	+1	-1	-1	7118.58
+1	+1	+1	-1	+1	-1	-1	-1	7014.04
+1	+1	+1	+1	+1	+1	+1	+1	14344.42
9151.54	-1115.5029	3057.08124	2518.48458	-413.8860	176.9820	833.141249	136.58374	TOTAL / 8
	29864322.2	224297898.4	152226350.3	4111243.8	751743.78	16658984.2	447722.89	SST = 428748435.8
	6.9654649	52.314569	35.5048176	0.95889	0.175334	3.8854915	0.1044255	

Repetition #1	Repetition #2	Repetition #3	Mean TPS	Squared Err. #1	Squared Err. #2	Squared Err. #3
5165.0	5186.66	5101.73	5151.13	192.376	1262.38	2440.36
8637.29	8376.47	8309.3	8441.02	38521.9	4166.70	17350.1
10639.52	10901.28	10559.05	10699.95	3651.78	40533.7	19852.8
16851.91	17053.89	16422.41	16776.07	77183.9	5751.7	125075.4
3697.74	3602.49	3701.07	3667.1	938.8096	4174.452	1153.961
7202.69	7054.47	7098.59	7118.58	7073.931	4110.519	399.7333
7130.41	7036.79	6874.92	7014.04	13541.97	517.5625	19354.37
14338.18	14309.71	14385.38	14344.42	38.97921	1205.015	1677.448
					SSE = 390170.07	

Table 6.1.9 – 6.1.10 Sign table and Errors table for the impacts of factors on Throughput (50%Set–50%Get)

As we would expect, the values in this case are half-way between the two previously analyzed types of experiment. In fact, the number of servers contributes for 6.96% to the performance of the system, while in the write-only experiment this value reached 12.72% and in the read-only it was equal to 1.33%. Once again,

the number of middlewares have the biggest impact on the throughput of the system, contributing for more than a half of its increase, followed by the effect due to the number of worker threads that accounts for 35.5% of the total variation.
In the following table for the study of the effects on the response time, we can see that, just like in the other two experiments, the impact of the number of servers gains a few percentage points to the detriment of the positive impact of the number of middlewares while the effect due to the number of worker threads remains approximately the same at 36%.

I	S	M	W	SM	SW	MW	SMW	MEAN RT
+1	-1	-1	-1	+1	+1	+1	-1	37.267055
+1	-1	-1	+1	+1	-1	-1	+1	22.6823333
+1	-1	+1	-1	-1	+1	-1	+1	17.959777
+1	-1	+1	+1	-1	-1	+1	-1	11.4554444
+1	+1	-1	-1	-1	-1	+1	+1	52.429611
+1	+1	-1	+1	-1	+1	-1	-1	26.985611
+1	+1	+1	-1	+1	-1	-1	-1	27.4570555
+1	+1	+1	+1	+1	+1	+1	+1	13.372777
26.2012	3.860055	-8.6399	-7.5771	-1.0064	-2.304902	2.43001	0.40991	TOTAL / 8
	357.6006	1791.56	1377.92	24.3083	127.50184	141.7192	4.032760	SST = 3827.43209
	9.343097	46.8085	36.0012	0.63518	3.3312633	3.702723	0.10536	

Repetition #1	**Repetition #2**	**Repetition #3**	**Mean TPS**	**Squared Err. #1**	**Squared Err. #2**	**Squared Err. #3**
37.165	37.01	37.6261	37.267055	0.0104153	0.066077	0.1289607
22.222	22.9155	22.9088	22.6823333	0.2112934	0.05436669	0.0513
18.062	17.624	18.193	17.959777	0.0104493	0.11274672	0.0545
11.396	11.266	11.7043	11.4554444	0.0035336	0.0358892	0.061945
52.1331	53.2876	51.868	52.429611	0.087879	0.7362593	0.315407
26.649	27.268	27.0398	26.985611	0.1133070	0.0797434	0.002940
26.9733	27.3201	28.0776	27.4570555	0.2339871	0.0187385	0.385158
13.34	13.424	13.3543	13.372777	0.0010743	0.00262372	0.0003
					SSE = 2.7789873	

Table 6.1.11 – 6.1.12 Sign and Errors tables for the factors' impacts on Response Time (50%Set–50%Get)

In conclusion, the number of middlewares explain the majority of the variation of the throughput, from 46.5% in the write-only experiment up to 53.6% in the read-only experiment. The second factor in order of importance is the number of worker threads inside the middleware(s), that impact on the performance of the system for an average of 37.76% of the total variation, both when considering throughput and response time. Lastly, an increasing number of servers cause the performance of the system to decline. We have pointed out that in all the experiments considered here, its impact on the response time was higher than the one on throughput, although, it is still very small if compared to the importance of the number of middlewares and worker threads. In fact, in the read only experiment, it is even smaller than the intermediate effect of the other two main factors. We have seen that its impact is approximately 1/3 of the one for the number of worker threads in the write-only experiment, since we have to replicate the request to all the servers, while it is almost negligible for the read-only experiment.

7 Queuing Model

7.1 M/M/1

In this section we are going to build a model for each worker thread configuration, assuming that the interarrival times and the service times are exponentially distributed and there is only one server. We will analyze how the measurements obtained in Section 4 compare with the theoretical results of the model for three different numbers of virtual clients for every configuration of worker threads. In particular, the first point refers to an experiment in which the system is under-saturated (i.e. 6 VC), the last one is a situation where the system has saturated (i.e. 240 clients) and the second one is an intermediate state between the two (different for every configuration of worker threads).

Due to the fact that the real implementation of the system that is running is multi-threaded, in an attempt to adapt it to the M/M/1 model, we can assume the service rate to be the maximum throughput ever observed for the configuration of worker threads that we are considering. In fact, in our implementation of the middleware, the throughput is computed as the total number of requests served by all the worker threads over the span of a 4 seconds window (multiple times in a single experiment). Therefore, using this measurement, that is the aggregated result of the whole system as a single unity, is the underlaying idea behind building an M/M/1 model that best adapt to an implementation that exploits multiple servers in practice.

$$\mu = max\{Throughput\}$$

The other crucial input parameter of the model is the interarrival rate. As already mentioned in Section 1, we don't have to make assumptions or hypothesis for it, since during the experiments we keep track and log the time between two successive arrivals. Therefore, we can obtain the interarrival rate through the formula:

$$\lambda = \sum_{i}^{m} \frac{1}{E[\tau]_i}$$

with m being the number of middlewares and $E[\tau]_i$ the average of interarrival times measured on middleware i.
To see if our measurements are consistent across the experiment, we can then verify if λ is approximately equal to the average throughput that we obtained for a particular configuration of clients and worker threads. The results for λ and the mean throughput are reported on the two first columns of the Table
We can then compute the utilization as:

$$\rho = \frac{\lambda}{\mu}$$

Additionally, for each request (in a 4 seconds window), the time it spent in the queue, the time in service and the total time in the system are available. As a consequence, we can calculate the mean number of jobs in the queue as:

$$E[n_q] = arrival\,rate * mean\,waiting\,time$$

These values are reported in the Table 7.1.1 under the column named **λ * mean waiting time**, and see how distant these are from the average of the queue length that we saved each 50 ms and logged (reported under the column **Queue Length**).
In the same way, using the Little's Law, we can compute the mean number of jobs that are in service $E[n_s]$ and the mean total number of jobs in the system $E[n]$ (that we can find in Table 7.1.1 under the columns **λ * mean service time** and **λ * mean total time in system** respectively).
Thus, for each experiment, we should get that:

$$E[n] \simeq E[n_q] + E[n_s]$$

The model predictions we compute are therefore those for the mean number of jobs in the queue and in the system. As reported on the Book *The Art of Computer Systems Performance Analysis* (by Raj Jain) at page 525, Box 31.1, we can compute those values respectively with the formulas: $\frac{\rho^2}{(1-\rho)}$ and $\frac{\rho}{(1-\rho)}$.

Name: Andrea Rinaldi Legi: 17-941-626

W	C	λ [ops/s]	TPS [ops/s]	μ [ops/s]	ρ	Queue Length	λ * mean waiting time	$\frac{\rho^2}{(1-\rho)}$	λ * mean service time	λ * mean total time in system	$\frac{\rho}{(1-\rho)}$
8	6	1117.148	1130.132	6238.895	0.1791	0.59544	0.534378	0.039125	2.55320	3.117713	0.218187
8	24	3587.059	3686.600	6238.895	0.5749	3.83353	3.981678	0.78559	8.897047	12.97037	1.36055
8	240	5482.291	5527.078	6238.895	0.8787	202.182	208.555	8.919319	16.34427	225.06189	9.79804
16	6	1168.555	1151.744	7286.292	0.1604	0.50656	0.54472	0.0310570	2.39465	2.97388	0.191434
16	48	5518.491	5545.435	7286.292	0.7574	9.57956	9.17248	2.386245	19.94934	29.27964	3.143625
16	240	6485.286	6517.606	7286.292	0.8901	175.3035	186.6918	11.11882	32.72182	219.6117	12.00889
32	6	1283.592	1295.346	8687.8125	0.1477	0.43599	0.424645	0.025811	2.29991	2.764231	0.17355
32	72	6843.436	7031.069	8687.8125	0.7877	11.0351	11.12923	2.960686	32.2605	43.580216	3.74839
32	240	8267.618	8239.937	8687.8125	0.9516	126.22	128.6972	18.72408	60.82578	189.7941	19.6757
64	6	1275.105	1270.505	9756.0208	0.1307	0.3590	0.4942	0.019716	2.60616	3.141349	0.150415
64	96	7393.113	7639.645	9756.0208	0.7578	15.9525	16.58442	2.376934	44.46467	61.26601	3.134734
64	240	9304.685	9515.587	9756.0208	0.9537	74.41341	76.44290	19.66214	103.26552	180.06377	20.61588

Table 7.1.1 Measurements and M/M/1 Model predictions for our system

In the table the first two columns represent the number of worker threads per middleware and the total number of clients, respectively. We can already see that the results for λ and the average throughput are approximately similar, as we would expect, meaning that the results obtained in the middleware are consistent across different types of measurements. This is also proved if we compare the queue length as measured through the `TimerTask` ever 50 ms and the average number of jobs in the queue computed using the Little's Law. Lastly, also the condition $E[n] \simeq E[n_q] + E[n_s]$ holds. In fact, if we sum the average number of jobs in the queue and the average number of jobs that are being served we approximately get the average number of jobs in the system for each experiment.
We can appreciate the bounty of the assumption of μ being equal to the maximum throughput by looking at the utilization ρ. In fact, for every configuration of worker threads, it reflects the real state of the system, as it increases from 0.15 when the system is under-saturated, up to approximately 0.95 when the system has saturated. We know that a system utilization should not go higher than that value, as it could become unstable, so we can assume that these results fit a good model to the real state of the system.

Although, as we can see from the average number of jobs in the queue and in the system, computed with the mentioned formulas, the model flaw in representing the real implementation of the system in certain aspects.

As a matter of fact, the M/M/1 model does not consider two crucial factors, meaning, the number of worker threads and the number of clients that concurrently sends requests to the middleware. Let's take for example the experiments with 8 worker threads for each middleware and 240 clients. We can see that the average number of jobs in the queue as measured in the middleware is equal to 208.555 while the predicted value is 8.919319. If we consider the fact that, as soon as memtier starts, each of the 240 clients concurrently send a request, and that there are only 8 worker threads, the real measurements seem to be reasonable. Moreover, looking at the output file for the queue length, it is noticeable that there are long sequences of numbers that fluctuates around 120 [jobs], that alternate with shorter sequences of rows with very small, if not null, outputs. This means that the clients actually send the requests following a certain periodicity, or equivalently, that the interarrival rate is not very consistent during the experiment. Consequently, there are times in which the middleware is in an almost idle state and others when there is a lot of workload both in service and in the queue. This is a situation that cannot be captured by the average interarrival rate, and so the model prediction differs from the real measurements. Lastly, since the average number of jobs in the system as predicted by the model just adds ρ to the average number of jobs in the queue, (while in practice there are 8 worker threads), also the last column values differs from the real measurements for that same reason.

Name: Andrea Rinaldi Legi: 17-941-626

7.2 M/M/m

W	C	λ [ops/s]	TPS [ops/s]	μ [ops/s]	ρ	Queue Length	λ * mean waiting time	$\frac{\rho\varrho}{(1-\rho)}$	λ * mean service time	$m\rho$	λ * mean total time in system
8	6	1117.148	1130.132	11639.751	0.09597	0.59544	0.534378	1.1557e-12	2.55320	1.5356	3.117713
8	24	3587.059	3686.600	7693.631	0.46623	3.83353	3.981678	0.004140	8.897047	7.45979	12.97037
8	240	5482.291	5527.078	5963.163	0.9193	202.182	208.555	7.53350	16.34427	14.7097	225.06189
16	6	1168.555	1151.744	17881.024	0.06535	0.50656	0.54472	6.2890e-28	2.39465	2.09125	2.97388
16	48	5518.491	5545.435	11225.63	0.49159	9.57956	9.17248	0.0002105	19.94934	15.73111	29.27964
16	240	6485.286	6517.606	7685.021	0.84388	175.3035	186.6918	1.4314	32.72182	27.00437	219.6117
32	6	1283.592	1295.346	53482.748	0.0240	0.43599	0.424645	3.6308e-80	2.29991	1.5360	2.764231
32	72	6843.436	7031.069	16749.90	0.236	11.0351	11.12923	2.1100e-10	32.2605	26.14819	43.58216
32	240	8267.618	8239.937	10171.717	0.8128	126.22	128.6972	0.31525	60.82578	52.01949	189.7941
64	6	1275.105	1270.505	101406.1	0.01257	0.3590	0.4942	1.9137e-192	2.60616	1.60950	3.141349
64	96	7393.113	7639.645	28876.441	0.25602	15.9525	16.58442	6.75779e-37	44.46467	32.7713	61.26601
64	240	9304.685	9515.587	13041.899	0.71344	74.41341	76.44290	0.000442	103.26552	91.32103	180.06377

Table 7.2.1 Measurements and M/M/m Model predictions for our system

In this section, we build an M/M/m model showing the measurements and the model predictions for the same configurations of worker threads and clients as we did for the M/M/1 model. As before, the first two columns represent the number of worker threads and the number of clients, the column named **λ** shows the interarrival rate that, as we have already seen, is very similar to the average **TPS**.
Looking at the service rate **μ**, this is computed considering the maximum service rate observed for a specific configuration of the system and multiplying that value for the number m of worker thread. To be more accurate, as we consider the lambda as the sum of the interarrival rates at the two middlewares, then, also for m, we need to sum the number of worker threads in the two middlewares.
It is noteworthy that the utilizations are generally lower than their counterparts in the M/M/1 model, and that is probably due to the fact that we are multiplying the maximum service rate for the number of worker threads, while it is very unlikely to happen that all the worker threads register the overall maximum service rate simultaneously.

As for the M/M/1 model we compute the prediction for the average number of jobs in the queue $\left(\frac{\rho\varrho}{(1-\rho)}\right)$ that should be similar to the measurements of the average queue length also reported on the table. Although, it is clear that this similarity does not hold, since, as already explained in the previous sub-section, we cannot take into consideration in the model, with the only mean interarrival rate, the number of clients that are generating the workload.
Contrarily to the M/M/1 model though, we now can consider the number of worker threads that serve the requests coming into the middleware. As a consequence, I decided to show in this table the mean number of jobs in service as predicted by the model $(m\rho)$, rather than the mean number of jobs in the system. It can be seen, indeed, that, for all the configurations of worker threads and clients, the values predicted by the M/M/m model are quite similar to the ones measured through the instrumentation of the middleware (showed on the column **λ * mean service time**).
This means that this type of model actually fits better our implementation of the system, as it can provide a good approximation of the system utilization and also the predicted values for the average number of jobs served by the worker threads are close to the measured ones.

7.3 Network of Queues

In this section we are going to build a network of queues that simulates our system, based on the experiments ran in Section 3.
We have shown in the previous two sub-sections, when modeling our system implementation as M/M/1 and M/M/m, that the mean interarrival rate is approximately equal to the average throughput. Therefore we can assume the system to be a closed one, meaning that the completed jobs, exiting the system, immediately reenter. The system is modeled as follows:
The network between clients and middleware(s) can be considered as a delay center, as the jobs spend approximately the same amount of time in this device, regardless of the number of jobs in it.
The net thread can be optimally modeled as an M/M/1, since it is the only thread that manages the incoming requests. Because every job that enter the system has to go through the net thread, this device visiting ratio V_{nt} is equal to 1 and the throughput X_{nt} equal to the throughput of the system X.
Lastly, the worker threads and the network queue constitute the final serving device that, as shown in Section 7.2, can be modeled with a good degree of accuracy as an M/M/m. Of course, it needs to be considered as a load-dependent service center, since the service time changes considerably depending on the workload. It is worth mentioning that the server should also be modeled as an M/M/1, although, we are not able to measure the exact amount of time the server needs to handle a request. In fact, we have the time that every jobs spend in the server at our disposal, but this is heavily impacted by the middleware, as the server might be faster in processing the requests than the worker thread to fetch the response. Consequently the best approximation that can be done considers the server's service time as the minimum registered of these time intervals.

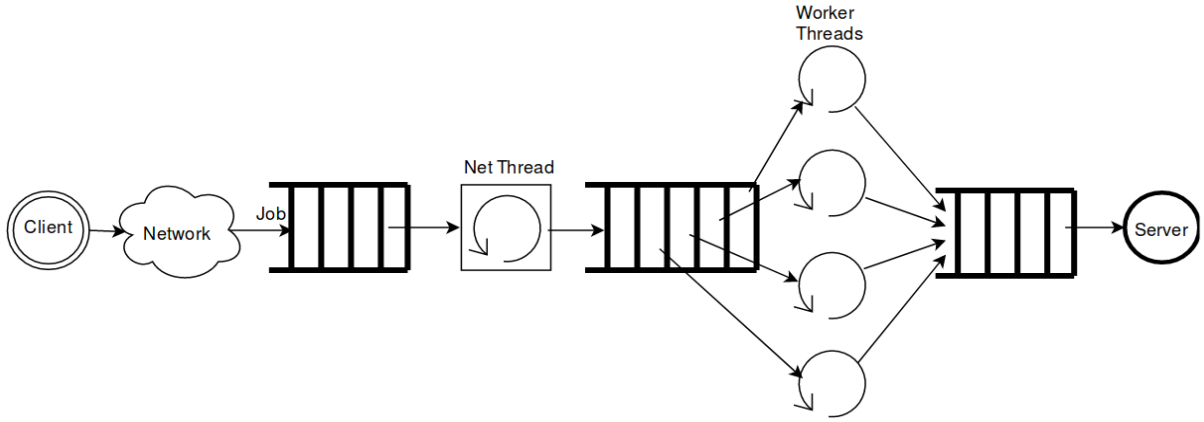

Figure 7.3.1 Illustration of the Network of Queues Model

To have a complete perspective of how this queuing model fits the real implementation of the system, I will now go through the results obtained for the Set and Get requests, both with one and two middlewares.
I decided to consider, as a sample configuration, the one with 8 worker threads and 48 clients in total, when the system is saturating. Therefore, since we already explained in Section 3 that the middleware is the bottleneck of the system in this case, we expect to see the utilization of the M/M/m model being quite close to 1, while the net thread and the server service centers could handle a much higher workload.
In the following tables are reported the commands and the output, using a queueing tool for GNU Octave that allows us to analyze our network. In particular, for the study of the model with one middleware, Q1, Q2 and Q3 identify in order the three queues shown in the figure above from left to right. These queues are defined with three parameters: the server type, the mean service time and the number of identical servers at the node. In the function qnsolve, we specify the fact that the system is a closed one, the number of requests in the system, the list of queues in the network and lastly the vector of visit ratios.
Looking at the results for the model with one middleware, we can see that, indeed, the utilization of the M/M/m model is approximately 1, as the system is saturating, while the other two serving centers are under-utilized. This is also reflected in the average number of requests Q at each center, as the almost totality of the jobs is waiting in the second queue. We are aware that the description of the server device is an approximation that is considering a lower bound, so the actual results (U, Q, R) might be higher than the outcomes showed in the table. Although, as explained in Section 3, the bottleneck would be the middleware anyway, and in particular the component constituted of the worker threads and the queue before them.

Therefore, we would have the highest utilization in the M/M/m serving device, even with higher values describing the server. As a consequence, the throughput X, that is shaped by the slowest of the devices, peaks at 2618.3 ops/sec and 2407.9 ops/sec for Set and Get requests respectively, result that is very similar to the measurements of the system (2646.3887 and 2455.6109).

SET	**GET**
Q1 = qnmknode("m/m/m-fcfs", 1.4879e-05, 1) Q2 = qnmknode("m/m/m-fcfs", 0.00305543, 8) Q3 = qnmknode("m/m/m-fcfs", 5.19e-05, 1) > [U R Q X] = qnsolve("closed", 48, {Q1,Q2,Q3}, [1 1 1]) U = 0.038959 1.000000 0.135889 R = 1.5483e-05 1.8257e-02 6.0062e-05 Q = 0.040538 47.802202 0.157259 X = 2618.3 2618.3 2618.3	Q1 = qnmknode("m/m/m-fcfs", 2.9662e-05, 1) Q2 = qnmknode("m/m/m-fcfs", 0.00332254, 8) Q3 = qnmknode("m/m/m-fcfs", 5.34e-05, 1) > [U R Q X] = qnsolve("closed", 48, {Q1,Q2,Q3}, [1 1 1]) U = 0.071422 0.999992 0.128575 R = 3.1944e-05 1.9842e-02 6.1279e-05 Q = 0.076915 47.775539 0.147546 X = 2407.8 2407.8 2407.8

Table 7.3.1 Results of the Network of Queues model for the Section 3.1

The situation is very similar in the case with two middlewares inside our system. The input are identified as follows: Q1 and Q2 are the two M/M/1 queues of the service centers with the net threads, Q3 and Q4 are the queues served by the worker threads, and Q5 is the last queue for the jobs entering the server.
In each of the two middlewares we have 8 worker threads, so we have 16 in total this time, and we keep analyzing the configuration with 48 clients. In order to define the visit ratio as accurately as possible, I took the proportion of requests that each of the two middlewares received and served, while for the server the visit ratio remains 1.

It is clearly noticeable that, also in this configuration of the system, the bottleneck is the service center that comprise the worker threads, as the utilization of the M/M/m model is more than 0.97 in both cases, while the net threads and the server are under-utilized (between 6.4% and 12.6%). It is worth mentioning that, as already pointed out, the utilization, queue length and response time of the server are smaller when considering Set requests than Get requests. Even though the computed throughput seems to be higher than the one measured during the experiments, the response time is a close approximation of the experimental results, that are 3.61ms and 6.68 ms for Set and Get respectively.

Since this tool makes use of the operational laws to compute the resulting queueing network model, it could be proved that all these relationships hold with the results presented so far.

SET	**GET**
Q1 = qnmknode("m/m/m-fcfs", 2.4707e-05, 1) Q2 = qnmknode("m/m/m-fcfs", 2.2032e-05, 1) Q3 = qnmknode("m/m/m-fcfs", 0.00151284, 8) Q4 = qnmknode("m/m/m-fcfs", 0.0015849, 8) Q5 = qnmknode("m/m/m-fcfs", 6.4e-06, 1) > [U R Q X] = qnsolve("closed", 48, {Q1,Q2,Q3,Q4, Q5}, [0.50953, 0.49046, 0.50953, 0.49046 1]) U = 0.12675 0.10880 0.97015 0.9783 0.0644 R = 2.82e-05 2.47e-05 4.43e-03 5.04e-03 6.84e-06 Q = 0.1451 0.1220 22.7431 24.9207 0.0688 X = 5.1302e+03 4.9383e+03 5.1302e+03 4.9383e+03 1.0068e+04	Q1 = qnmknode("m/m/m-fcfs", 2.5143e-05, 1) Q2 = qnmknode("m/m/m-fcfs", 2.9126e-05, 1) Q3 = qnmknode("m/m/m-fcfs", 0.0025619, 8) Q4 = qnmknode("m/m/m-fcfs", 0.00250976, 8) Q5 = qnmknode("m/m/m-fcfs", 1.42e-05, 1) > [U R Q X] = qnsolve("closed", 48, {Q1,Q2,Q3,Q4, Q5}, [0.49489 0.50510 0.49489 0.50510, 1]) U = 0.0765 0.0904 0.9745 0.9744 0.0873 R = 2.7225e-05 3.2021e-05 7.8471e-03 7.6760e-03 1.5557e-05 Q = 0.0828 0.09945 23.88012 23.8419 0.0956 X = 3043.2 3106.0 3043.2 3106.0 6149.2

Table 7.3.2 Results of the Network of Queues model for the Section 3.2

Name: Andrea Rinaldi Legi: 17-941-626