Advanced Systems Lab Report

Autumn Semester 2017

Name: Gianni Perlini
Legi: 13-818-752

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

1 System Overview (75 pts)

In this section we will describe the system implementation and the design choices that we made during the development of the middleware. The section is divided into two subsections, the first giving a general overview of the system and the second one digging into the details of how the system is implemented.

1.1 General Overview

To begin, it is useful to take a look at the overall architecture of the system. *Figure 1* below is taken from the *project description* [2] and shows the internal architecture of the middleware.

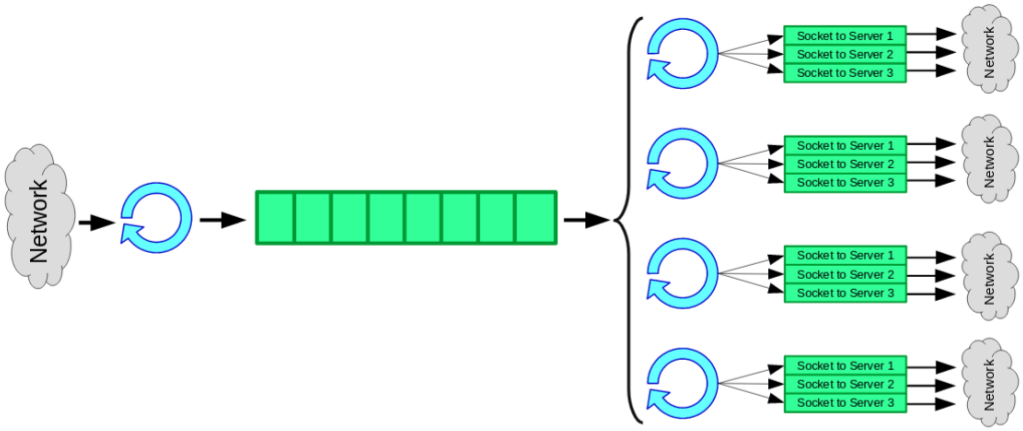


Figure 1: Internal architecture of the middleware

The system acts as a server to the clients (left of *Figure 1*, virtual machines with instances of `memtier_benchmark`) and as a client to the servers (right of *Figure 1*, virtual machines with instances of `memcached`). The middleware is composed of several parts:

- *Network Thread*: This thread is responsible for listening for client connections on a specific port. Upon receival of a request, the network thread reads it and puts it into the *Requests Queue*.
- *Requests Queue*: This queue holds the requests coming from the clients that will then be processed by the *Worker Threads*.
- *Worker Threads*: These are responsible of taking the requests out of the queue, parsing them, sending them to the servers, and send the response back to the clients. As shown in *Figure 1*, each worker thread has open connections to the servers (up to three simultaneously, depending on the number of servers on the experiment), used to forward requests. Idle worker threads are kept into a thread pool waiting for new requests to be processed.

Upon receival of a new request, the middleware acts as follows:

1. The network thread read the incoming request and put it into the queue.

2. One of the worker thread waiting in the pool takes the request out of the queue. It then parses it and, depending on which type of request it is, proceed as follow:

    - *Get*: Get requests are simply forwarded to one of the servers using round robin. The worker thread simply reads the value of an atomic index used to keep track of which is the last server that was forwarded a request. It then increments it and forward the request to the appropriate one.
    - *Multi Get*: This type of request is treated differently depending on weather the mode is `sharded` or `non-sharded`. In the first case the worker thread splits the request evenly in order to keep the load on the server equally distributed in terms of keys. These multiple requests are then forwarded to the servers. In the second case, the worker simply forward the complete request to one of the server as if it were a normal Get operation.
    - *Set*: Set request are forwarded to all the servers to provide robustness. The worker thread sends the request to all of them and after that it waits for them to answer.

3. After sending the request, the worker thread simply waits for the server(s) to respond.

4. Upon receival of the response, the worker thread sends it back to the appropriate client.

In the next subsection we will see how these behaviours are implemented.

1.2 System Implementation

The system is implemented using `Java` and is composed of the following classes:

- *RunMW.java*: This class was provided by the TAs and is in charge of parsing the middleware's arguments and to start the middleware.
- *MyMiddleware.java*: This class is in charge of starting all the different parts of the middleware, as well as to write most of the statistics after the systems started to shut down.
- *NetworkThread.java*: This class models the corresponding thread which is in charge to read requests from the network and put them into the queue.
- *RequestWrapper.java*: This class wraps a single request into an object containing a `String`, a `SocketChannel` and a `long`. The first parameter represents the actual request, the second one binds the request to the client which sent it, while the third one is a timestamp started when the request is put into the queue.
- *Worker.java*: This class encapsulates the jobs that worker threads have to perform while processing a request.
- *LoggingTimerTask.java*: This class is in charge of aggregate statistics every second.
- *QueueLengthTimerTask.java*: This class is used to log the queue length every 100 milliseconds.

We will now see the implementation details of each one of this class.

MyMiddleware.java This class has different types of parameters. The first type is entered by the command line and tells the middleware which IP address and which port the network thread needs to use to listen for connections. It also tells the middleware which IPs and ports the worker threads should use to connect to the servers. Finally, it tells the middleware if it has to expect sharded or non-sharded requests and the number of worker threads to use. The class also has a number of parameters used to write statistics to files. These are represented by `ArrayList<String>` and `ConcurrentHashMap<Long, Long>`. Finally, the class uses a `LinkedBlockingQueue<RequestWrapper>` to represent the queue of requests.
When started, the class first initializes the right number of worker thread in the thread pools. This is done using an `ExecutorService`. It then starts the network thread passing it the right IP and port, as well as the requests queue.
The class also implements a shutdown hook in order to print statistics to files before shutting down. When terminating, the middleware interrupts the network thread and the worker threads, and shuts down the executor. In order to stop the workers, the middleware puts a number of bogus requests on the queue. Finally, the middleware writes statistics to different files before terminating, using methods from the `java.nio.files` package.

NetworkThread.java Other than the ones taken from `MyMiddleware`, the network threads has parameters representing `Selector` and `ByteBuffer` used to listen and to read from the clients connections. Also, the network threads has an `AtomicLong` used to keep track of the arrival rate at the system, and two `Timer`s used to initialize the two `TimerTask`s.
Upon initialization, the network thread sets up the required sockets using the `java.nio` package. In order to know when to stop, the thread keeps a `boolean` called `isRunning` which is set to false by the middleware when shutting down, and which enables the network thread to close the channel and the selector, as well as to stop the tasks. Upon receival of a request, the thread first increment the arrival rate counter and then tries to read the request from the channel. It then creates a new `RequestWrapper` using the request and the socket channel. If it is the first request, the network thread also starts the two `TimerTask`s. The network thread then waits on the `selector.select()` call until a new request arrives.

RequestWrapper.java This class wraps the request using three parameters, *i.e.*, a `String` representing the actual request, a `SocketChannel` binding the request to the client who submitted it, and a `long`, used to keep track of when the request was put into the queue. While the first two are passed by the network thread, the last one is initialized in the constructor and uses the `System.nanoTime()` call to do this.

Worker.java This class has various parameters covering the informations used to connect to the servers, going through the `long`s used to calculate different types of processing times, to the ones used to collect statistics.
Upon invocation of the `run()` method, the worker starts by taking the request out of the queue in order to process it. This is done using the `take()` method of the `LinkedBlockingQueue`, which blocks the thread until a request is ready. The thread then record the current time and uses it to compute the **Time in Queue** altogether with the `putTime`, namely the time recorded when the request was put into the queue. This time is registered into a variable and will later used at parsing time in order to be able to distinguish between the type of operation. Next, the worker checks if the request is a bogus request in order to decide weather to process it, in case it is a legitimate request, or to stop the execution. If the request is not bogus, the thread continues the execution by unwrapping it, taking the string and socket channel out from it. It then starts the `startParseTime` by invoking the `System.nanoTime()` method and then calls

the `parseRequest()` method.
The `parseRequest()` method takes the string request as input parameter and parses it in the following way. It first takes the first three characters of the request, and uses a `switch` to compare them. If the request is a set, the worker records the `endParseTime` and stores it into an `AtomicLong`. The process is the following: the worker keeps two variables, an `AtomicLong` which stores the sum of all the previous parse times and an `AtomicInteger` which stores the actual count, to later be able to compute the average. These two variables are atomic since they are shared between all the workers. The worker then performs the same operation for the time spent in queue (adding the appropriate time accordingly to the request type), initializes the `startSendTime` variable and calls the `handleSetRequest()` method.
In case of a get request, the worker first check how many keys are in the request. If multiple keys are present, the worker updates the respective times accordingly and call the `handleMultiGetRequest()` method, while if only one key is in the request, the `handleGetRequest()` method is called. In case the request does not fall in one of the three categories, the thread adds the request to `wrongRequests` which is an `ArrayList<String>` and call the `sendResponseToClient()` with an error message.
The execution continues with the handle methods. In case of a set request, this method simply calls the `sendSetRequest()` method with the string request. In case of a get, the thread first get the `serverIndex`, which is an `AtomicInteger` used to implement the round robin to decide to which server forward the request. The worker also performs the modulo using the `serverTotal` variable, which stores the number of servers used in the current experiment. It then increments a local `int` to keep count of how many requests have been forwarded to which server, and finally call the `sendRequestToServer` using the appropriate IP and port. In the last case, the worker first checks wheather the multi get is sharded or not. If it is not the case, the thread calls the `sendMultiGetRequest` which will then simply treat the request as a normal get, computing the `serverIndex` and calling the `sendRequestToServer` with the right IP and port. If the multi get is sharded, the worker first computes the number of keys in the request and then calls `splitMultiGetEvenly`. This method compute the `keyCount` statistic, which will be used in *Section 5*, using again an `AtomicLong` and `AtomicInteger` as in the previous cases. It then continues the execution by splitting the key evenly among the servers. To do that, the method first creates a string consisting of keys only, and then splits it according to the spaces in between each key. This create a `String[]` with each index corresponding to one key. The method also create another `String[]` with length equal to the number of servers, which represent the request that each server will have to process. It then iterates over the array of keys, concatenating the key with the request at the index of the appropriate server. This is done again using the `serverIndex` and the modulo. Finally, it concatenates each request with a line termination character and returns. The worker then calls the `sendMultiGetRequest` with a `String[]` as parameter, which represents the different requests to forward to the respective servers.
The execution continues with the send requests to server methods. In case of a get, the worker first chooses the appropriate `Socket`, `BufferedReader` and `PrintWriter` depending on the `serverIndex.` The first three are kept on three different `ThreadLocal<ArrayList<>>`, in order to avoid to keep opening and closing different sockets, reader and writer. The worker then uses the writer to send the request and initializes the `afterSendTime`. Depending on whether the request is a get or a non-sharded multi get, the thread updates the respective times and counters and waits for the response. To read the complete request, the worker keeps reading until the string is equal to `END`. Upon receival, the worker records the time spent waiting for `memcached` and update the miss count depending on the type of request. The miss count is kept as other statistics in an `AtomicInteger` incremented every time the request does not contain the string

VALUE. In the multi get case, the miss count is incremented by the number of missing VALUE string in the complete response. Finally, the worker calls the sendResponseToClient method. For set and multi get, the methods are very similar since the requests are forwarded to all the servers. In both cases, the worker loops over the total number of servers. In every iteration it takes the right sockets and print writers and sends the request to the server. After sending, the worker updates the time to send statistic and then proceeds to receive the responses from all the servers. To do that, it once again loops over the total number of servers and takes the right readers to read the input. In the set case, the worker checks if the response is equal to STORED and increments a local counter to keep track of the number of correct responses. In case of an error, it records the error index. In the multi get case, the worker loops over the total servers but also uses a while loop to be sure to read the whole response. When it reaches an END string, the worker checks whether this is the last server or not. If it is, it breaks out of the loops, if not, it replaces the END string and start reading from another server. When finished, both methods update the corresponding statistics and additionally, in the multi get case the miss count is updated too. Finally, the sendResponseToClient method is called.
The last method uses java.nio to send the response back to the client using the SocketChannel bounded to the request. It then sends the response and updates the **Send Response Back To Client** time. Finally, if it is a legitimate response and no errors were produced, the worker distinguish between the cases and update the corresponding statistics, namely the time to send the response back, the throughput, the total response time and the throughput and response time per worker thread. These last two statistics are later used for the queuing model and are kept in a ConcurrenHashMap mapping the id of the thread which performed the job to the measured throughput or response time. It is important to note that this method always allocate enough space to be able to send a multi get request with 10 keys. This is a waste of resources in case of set and get.
Finally, when the thread is stopping, *i.e.*, it reads a bogus request from the queue, it closes all the resources used for the network operations, and then write three types of statistics to three different files: the server count for each server, the wrong requests that it encountered if any, and the server errors, if any.

LoggingTimerTask.java This class is in charge of saving the statistics from the worker threads to the appropriate middleware lists. The class is composed of three main methods. The first one is the copyStats method, which saves every statistic gathered by the worker threads in the corresponding list in MyMiddleware. To do this, it takes the AtomicLong representing the sum and divides it by the AtomicInteger representing the count, always checking that the latter is non-zero, in order to perform the average of statistics gathered over that second. In the case of the throughputs, since the task is scheduled every second, it simply takes the counter as it is, which represents the number of operations done in one second.
The second method is called writeTHInFile and writes the statistics gathered for every single worker to a specific file.
The last method called is the clearWorkerStats method, which simply resets all statistics counters and sums to zero.

QueueLengthTimerTask.java This class simply takes the queue as parameter and saves the current queue length in the corresponding ArrayList of MyMiddleware.
This operation is done every 100 milliseconds.

To conclude this section, here is a complete list of all the statistics gathered by the system:

- *Queue Lengths*: the number of requests in the queue. This is taken every 100 milliseconds.
- *Throughput*: number of operations done in a second, recorded separately for get, set and multi get.
- *Time in Queue*: the time spent waiting in the queue by each request. This statistic is aggregated every one second.
- *Response Time*: the total time spent by a request in the system, starting from when the request is put in the queue until the time the corresponding response is sent back to the client.
- *Time to Parse*: time needed to parse a specific request, taken for every type of request. This is computed from when the request is taken from the queue (right before the call to the `parseRequest` method) until when the worker knows if the request is a set, get or multi get.
- *Time to Send*: time to send a response to the server, starting after the parse has finished until the request has been sent (to all the servers in case of multi get and set).
- *Time in Processing*: time waiting for memcached, starting after the request(s) is sent, to when the response has been received.
- *Time to Send Response Back*: time needed to send the response back to the client, computed from after the response has been received to when it is sent to the client.
- *Arrival Rate*: number of requests entering the system per second. This is measured at the network thread.
- *Miss Rate*: the number of misses for get and multi get.
- *Key Count*: average number of keys in a multi get request.
- *Server Counts*: number of requests forwarded to a specific server. Used to check if the load is evenly balanced in case of multiple servers.
- *Wrong Requests*: string requests which are neither a set nor a get, nor a multi get, if any.
- *Server Errors*: erroneous responses sent by the servers, if any.
- *Throughput per Worker Thread*: number of requests done per second per every worker. Used in the last section for the queing models.
- *Response Time per Worker Thread*: response time of every request per every worker. Used in the last section for the queing models.
- *Response Time per Request*: response time of every request. Used in the *Section 5* for the response time distribution.

2 Baseline without Middleware (75 pts)

In this experiments we study the performance characteristics of the memtier clients and memcached servers.

2.1 One Server

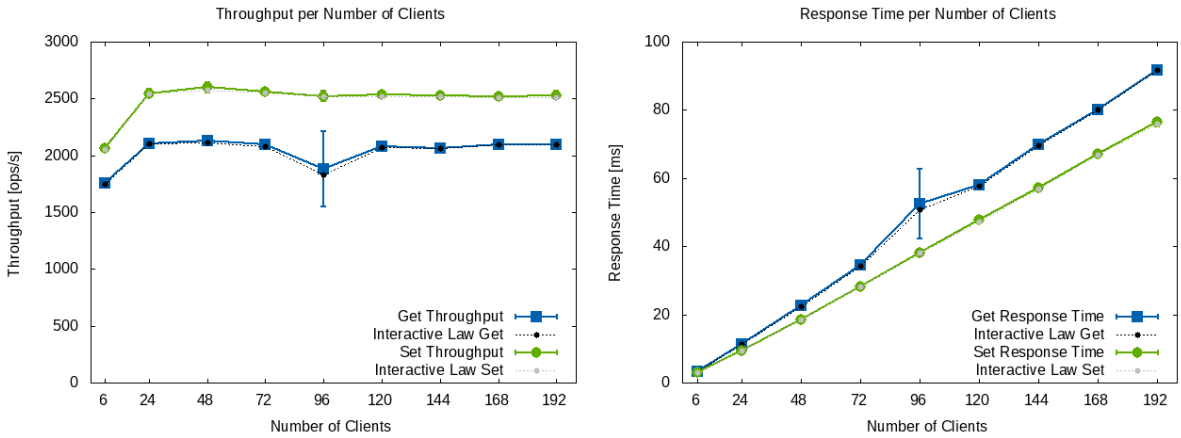
For this part of the experiment we performed a read-only and write-only workload, using 3 load generating VMs, with one memtier (CT=2) each, and varying the number of virtual clients (VC) per memtier thread between 1 and 32. All clients are connected to a single memcached instance. The table below summarize the various parameters and parameters ranges that we used, while below the table is the `memtier` command used for the experiment.

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..32]
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 or more

```
memtier_benchmark --port=9001 --protocol=memcache_text --ratio=x:y --expiry-
    ↪ range=9999-10000 --key-maximum=10000 -d 1024 --server=MEMCACHED_VM_IP --
    ↪ clients=[1, ..., 32] --threads=2 --test-time=80
```

Where the ratio was set to `1:0` for the write-only and to `0:1` for the read-only. The complete script for this experiment can be found under `asl-data/asl2/asl2.1/2.1op.sh` where $op = \{set, get\}$. It is important to note that before each experiment, the memcached server(s) was populated using a separate `memtier` command. The script for memcached population can be found under `asl-data/memcached_pop.sh`.

We show two plots representing the Response Time and Throughput for both types of operation, including the results derived using the **Interactive Law**, and defer the explanation to the next subsection.



(a) Average Throughput for set and get

(b) Average Response Time for set and get

Figure 2: Average Throughput and Response Time per number of clients, for both types of operations

2.1.1 Explanation

The two plots depict the throughputs and response times as measured on the clients virtual machines, as a function of the total number of clients sending requests. The overall behaviour of the system matches our expectations. As we can see in *Figure 2a*, after a first rapid increase between 6 and 24 clients, both the plots stabilize meaning that the server is saturated. The point of saturation is thus between 6 and 24 clients. As we would expect, throughput is higher for the set operations. This is because in the case of a get, the server needs to first look in the database to find the value, and then fetch it and send it back. In the case of set the server simply replies with a `STORED` string after receiving the request. We will see in next sections that this behaviour can change when the middleware is in between depending on the setup of the experiment, due to the way set and get are treated. An interesting behaviour can be seen in both *Figure 2a* and *Figure 2b*. We can see that the get throughput suddenly drops at 96 clients, and the standard deviation is quite high with respect to the other points in the plot, telling us that at that moment the throughput was not stable and was greatly varying. Looking at the data stored by `memtier_benchmark`, we can see the following:

```
...
[RUN #1 52%, 46 secs] 2 threads: 30095 ops, 0 (avg: 649) ops/sec, 0.00KB/sec (
    ↪ avg: 664.31KB/sec), -nan (avg: 46.33) msec latency
[RUN #1 53%, 47 secs] 2 threads: 30095 ops, 0 (avg: 635) ops/sec, 0.00KB/sec (
    ↪ avg: 650.17KB/sec), -nan (avg: 46.33) msec latency
[RUN #1 54%, 48 secs] 2 threads: 30095 ops, 0 (avg: 622) ops/sec, 0.00KB/sec (
    ↪ avg: 636.63KB/sec), -nan (avg: 46.33) msec latency
[RUN #1 55%, 49 secs] 2 threads: 30095 ops, 0 (avg: 609) ops/sec, 0.00KB/sec (
    ↪ avg: 623.63KB/sec), -nan (avg: 46.33) msec latency
[RUN #1 56%, 50 secs] 2 threads: 30095 ops, 0 (avg: 597) ops/sec, 0.00KB/sec (
    ↪ avg: 611.16KB/sec), -nan (avg: 46.33) msec latency
[RUN #1 57%, 51 secs] 2 threads: 30095 ops, 0 (avg: 585) ops/sec, 0.00KB/sec (
    ↪ avg: 599.17KB/sec), -nan (avg: 46.33) msec latency
...
```

This is a snippet of the output of `memtier_benchmark` during repetition 1 on the first virtual machine, with 16 clients. The same behaviour took place on the other two VMs at the exact same time. This could due to the network getting congested, meaning that the both the clients and the server had to try multiple time to send network packets.
Taking a look at *Figure 2b*, we can see that the response time behaves as expected and accordingly to the throuhgputs. The response time keeps growing with the number of clients increasing. We can see that at the saturation point, the response time goes from about 3 ms to 12 ms, meaning that even if the throughput increases by about 500 $\frac{ops}{s}$, the system is about 4 times slower. Again we can observe a sudden response time increase at 96 clients, accordingly to the throughput dropping at the same point. We can also see that, again as expected, the response time for set operations is lower than that for get.
Finally, in both plots we can see that the **Interactive Law**, holds. This was computed using the formula presented in class [1], which states that

$$X = \frac{N}{R + Z}$$

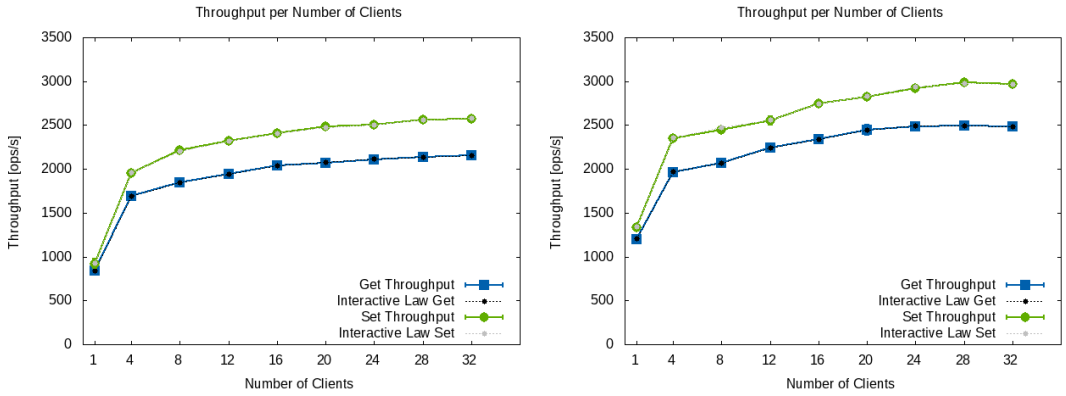
where X is the system's throughput, N the number of clients, R the response time and Z the think time. In our case, we took $Z = 0$ since the clients keep sending requests after receiving a response.

2.2 Two Servers

In this part of the experiment we examine throughput and response time in the following setup. We use 1 load generating VM, with one memtier (CT=1) connected to each memcached instance (two memcache instances in total), and we vary the number of virtual clients (VC) per memtier thread between 1 and 32. Again, below is a summary table. The command is almost the same as the previous part, except that we now use 1 thread and that each instance of memtier is connected to a different server.

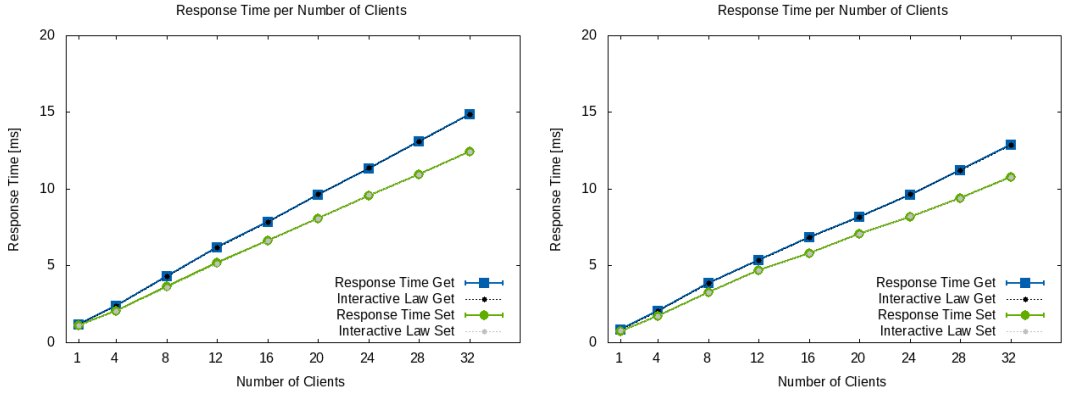
Number of servers	2
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1..32]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	3 or more (at least 1 minute each)

As before, we start by showing the plots for throughput and response time and defer the discussion to the next subsection.



(a) Average Throughput measured on the first client **(b)** Average Throughput measured on the second client

Figure 3: Average Throughput per number of clients, for both types of operations and both clients



(a) Average Response Time measured on the first client **(b)** Average Response Time measured on the second client

Figure 4: Average Response Time per number of clients, for both types of operations and for both clients

2.2.1 Explanation

Figure 3 shows the behaviour of the throughput for both types of operations measured on both clients. As before, we see that the plots first increase and then start saturating. The saturation point is between 1 and 4 clients, since we can see that the plots start getting flat. In general we can notice that set operations have a higher throughput as in the previous situation, and that the second client is faster than the first by about 500 $\frac{ops}{s}$. This is due to the allocation of the VMs. Recall that the two instances of memtier are on the same VM but are connecting to two different servers. The first server is probably further away to the client VM than the second one. This can also be seen by the fact that the second client's throughput stabilizes for 4 and 8 clients but then increases again and stabilizes back at 28 clients, while it starts decreasing at 32.
As in the previous section, we can see that the Interactive Law holds for both operations and for both clients.
On *Figure 4* we can observe the behaviour of the response time. As before, sets perform better than gets and thus have a smaller response time. As seen for the throughputs, the second client perform better than the first, since the response time is lower. The point corresponding to the throughput increasing again after stabilizing can be seen in the response time plot when, at 12 clients, the slope of the graph gets smaller, before increasing again.
Once again, the Interactive Law matches the experimental results perfectly.

2.3 Summary

The following table summarizes some of the results that we found out on the previous two parts:

Maximum throughput of different VMs.

	Read-only workload	Write-only workload	Configuration gives max. throughput
One memcached server	2099.0	2598.8	48 clients for set, 168 for get
One load generating VM	2498.62	2986.5	28 clients for both operations

In this first section we could examine different behaviours of clients and servers corresponding to different setups.
The first result worth noting is that in both cases set operations perform better than get, since they achieve higher throughput and lower response time. As already mentioned, this is due to the fact that the get operation needs to fetch and send the data (1 KB), while the set simply replies with a `STORED` string.
Another similarity in both experiments is the general behaviour of throughputs and response times. Both the metrics behave as expected. The throughput starts by rapidly increasing at the beginning, since the server is under-loaded. It then stabilizes at the saturation point, meaning that the server is working at the maximum capacity. In fact, in certain cases we can notice that the throughput increases a little and this could trick us into thinking that the system is not yet saturated. An example could be the one depicted in *Figure 3a*. Between 4 and 16 clients the throughput increases by about 400 $\frac{ops}{s}$. However, looking at *Figure 4a*, we can see that the response time goes up from 2 ms to roughly 6 ms, meaning that the system is performing three times slower.
As with the throughput, the general behaviour of response time is the same in both experiments. This metric keeps increasing linearly with the number of clients.

Coming to the differences we can see that the second setting performs better than the first one, since the throughput is higher and the response time is slower. At first, this could seem to be contradictory since to a certain degree we would expect the throughput to increase with the number of clients. However, this is exactly what we would expect. Recall that in the first part we used three different client VMs with two memtier threads each, all connected to the same server. This results in a much higher load on the server with respect to what happens in the second experiment, where the load is split into two servers. What the plots tell us is that, in terms of throughput, the system starts saturating around 4 clients, and then keeps a contant performance up to 192 clients. However, in terms of response time, the difference is tremendous. In fact, if we look at *Figure 2a* and *Figure 2b*, we can see that between 96 and 192 clients, set throughput stays more or less constant, but the response time increases from 40 to 80 ms, meaning a degrade in performance by a factor of 2.
This two experiments showed us how important it is to take into account different performance metrics. While one could be mislead by the throughput remaining more or less the same up to a high number of clients, possibly concluding that the system is able to efficiently sustain 192 clients and possibly more, considering the response time too is crucial since it let us understand that the system cannot support such a high load. Finally, we also had the chance to observe the differences between clients connecting to different servers, due to the different allocation of virtual machines in the cloud. In this case we saw that two instances of memtier on the same VM experienced a throughput differing by about 500 $\frac{ops}{s}$ and a response time differing by about 3 ms.

3 Baseline with Middleware (90 pts)

In this section we will see how our middleware behaves when relaying one load generator VM and one memcached server and increasing the number of clients. We will test a system with one and two middlewares.

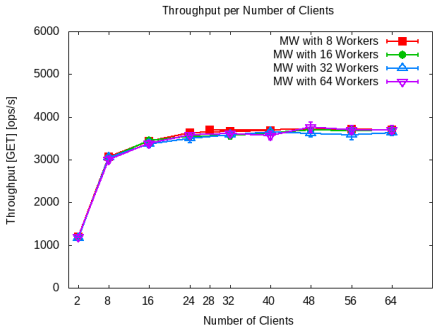
3.1 One Middleware

In this experiment we used one client VM (one instance of memtier with CT=2), one middleware and one server. We run a read-only and a write-only workload with increasing number of clients (between 2 and 64) and measured response time and throughput both at the clients and at the middleware. For this experiment we used four different configurations of worker threads, namely 8, 16, 32 and 64.
Below is the common summarizing table and the command used to start the middleware. The complete script can be found at `asl-data/asl3/asl3.1/bl3.1ssh_controller.sh`. As in the previous experiments, we used `memcached_pop.sh` to populate the server before running the experiment, in order to have a small percentage of cache misses. In this case, we achieved a miss ratio of 0, as can be seen by the clients' output and the `CM` files printed by the middleware.

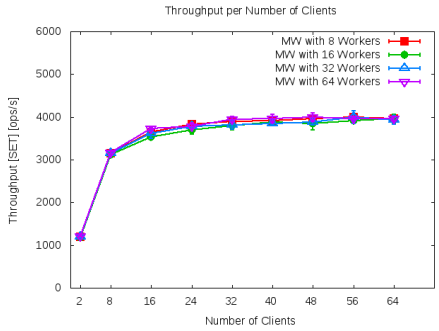
Number of servers	1
Number of client machines	1
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[1..32]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1
Worker threads per middleware	[8..64]
Repetitions	3 or more (at least 1 minute each)

```
java -jar asl17-project-mw.jar -l 10.0.0.7 -p 8999 -t [8, 16, 32, 64] -s false
    ↪  -m 10.0.0.4:9000
```

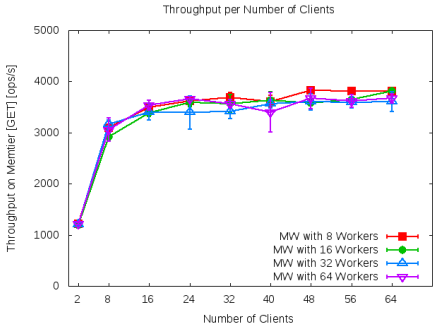
As before, the next eight plots represent the throughput and response time of the system measured at both the clients and the middleware, for both types of operation. The four plots representing the response time can be found at the next page.



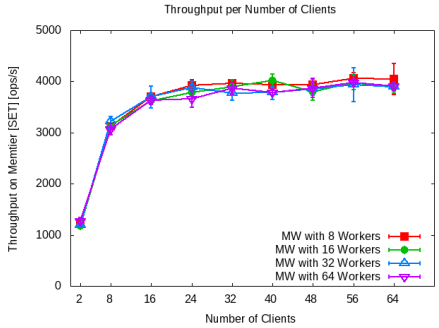
(a) Average Throughput for get measured on the mw



(b) Average Throughput for set measured on the mw



(c) Average Throughput for get measured on the clients

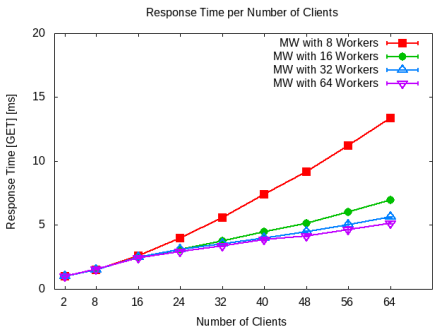


(d) Average Throughput for set measured on the clients

Figure 5: Average Throughput on middleware and clients

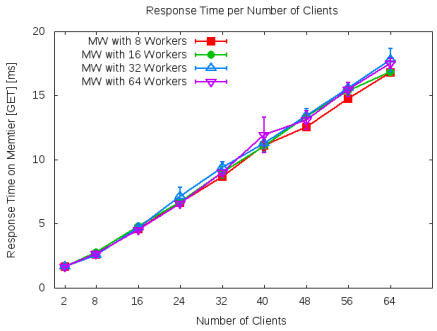
3.1.1 Explanation

To explain the behaviour of the system we will use a number of graphs covering different types of statistics gathered inside the middleware as well as on the clients.
We start by explaining the plots of throughput and response time, represented on *Figure 5* and *Figure 6*. The first one shows the behaviour of throughput both at the clients and in the middleware. As we would expect, we notice the familiar shape of the plots, strictly increasing in the beginning, when the system is under-loaded, and then starting to stabilize when the system is saturated. An interesting behaviour depicted in these figures is that, on both sides, the worker thread configuration does not seems to make a difference, at least in terms of throughput. This is actually understandable given the setup. Since we have a relatively low workload on the middleware (only one load-generating VM), the gain given by the increased number of worker threads is not worth the overhead resulting from the creation and management of them all. In fact, given the implementation of the middleware, a number of atomic operations are performed (such as, for example, some statistics related processing as well as management of the requests

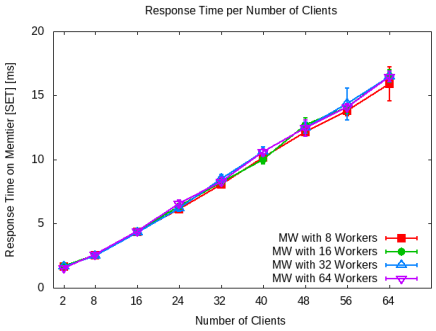


(a) Average Response Time for get measured on the mw

(b) AverageResponse Time for set measured on the mw



(c) Average Response Time for get measured on the clients



(d) Average Response Time for set measured on the clients

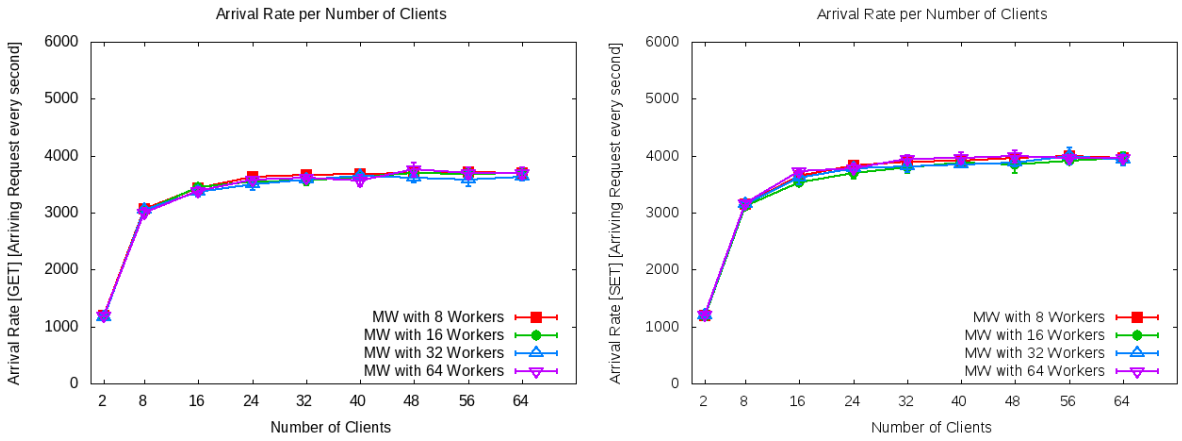
Figure 6: Average Response Time on middleware and clients

queue). However, at this point we cannot give a precise conclusion on the behaviour of different configurations since we are only looking at the throughput. Another interesting behaviour that can be seen from this set of figures is the reduced gap between the performances of set and get operations. This is due to the nature of the set of operation, which is forwarded to all the servers. Even if in this experiment we only have a server, the implementation is such that the worker thread handling the set operation has to go through a number of `for` and `while` loops which slow down the process. However, since we are using only one server, the difference in the middleware is compensated by the server, where sets perform better.

Figure 6 represents the respective response times. The first thing to notice is the difference between the different configurations in the middleware, in contrast with what we have seen on the throughput plots. Here we can see that with 8 workers, the middleware becomes up to 3 times slower with 64 clients, with respect to the better performing configurations. This is another example of what we said at the end of the previous section: examining only one performance metric could lead us to a misunderstaning of the system. As we would expect, the configuration with 32 and 64 threads perform better than the other two. These two performs almost equally which tells us that we are probably reaching the threshold where the gain of having more threads is balanced by the overhead of mananging them all. Another behaviour that we can examine from the plots is that the response time at the clients is in general much higher that the one measured at the middleware. This is expected since the clients are on another VM, and thus the response needs to be sent one more time over the network, which slows down the operation.

We will now see in details some statistics measured inside the middleware to understand how

it behaves internally. The first two figures show the arrival rate, as the number of requests arriving at the network thread per second.

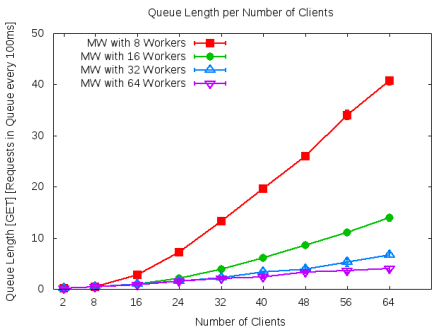


(a) Average Arrival Rate for get

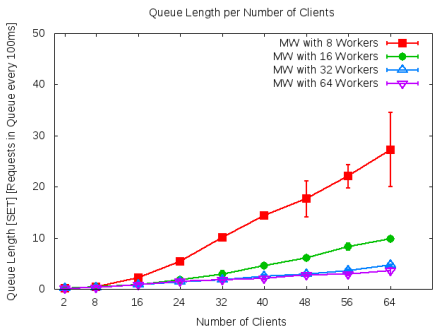
(b) Average Arrival Rate for set

Figure 7: Arrival Rate at the network thread

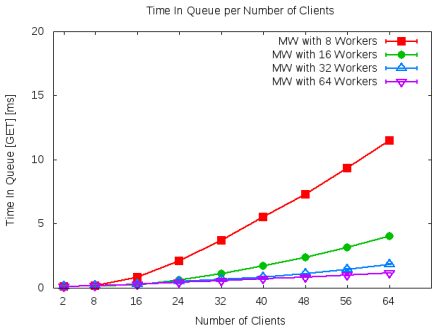
We can see that both are very similar to the respective throughput graphs. Looking at the data, the throughput is usually a bit higher. This tells us that the utilization of the system is almost 100%, since if we take the ratio of the number of jobs entering the system and the jobs leaving it, we get almost 1.
Figure 8 on the next page shows the queue length and the average time spent in the queue. As expected, the queue length grows much faster when there are less threads on the middleware, since they can only perform, in this case, 8 jobs at once. This already suggests that the queue is the bottleneck of our system. The result of the queue growing is clearly that the time spent in the queue by each request grows as well. It is interesting to see that the behaviour of the time in queue is almost identical to the one of the response time. In fact, comparing the two we can see that the time in queue corresponds to a big part of the response time, which supports our claim that the queue is actually the bottleneck. Again, we can see that the difference between the last two configuration is subtle, meaning that the ratio between the gain of having more threads and the overhead of managing them decreased with respect to the same ratio when going from 16 to 32 workers.
Figure 9 depicts the behaviour of the time to parse a request. First of all we specify that the two plots have the same scale in order to be able to compare them more easily. However, since for set the time is smaller, this plot does not capture completely the behaviour of this statistic. Another graph with an adjusted scale is available at
`asl-data/asl3/asl3.1/asl3.1mw_set_finished/TParseSTot/plot/tparse.png`. The same goes for the time to send. These two plots tell us some important information about the implementation of the middleware. The first thing that we should notice is the difference between the two types of operations. We can see that gets can be up to roughly 3 times slower than a set to parse. This is due to the implementation of the `parseRequest` method. Recall that, when parsing, the worker thread will take a substring of the request and then switch on this substring. If the request is a set, the parsing timer is immediately stopped. In case of a get, the worker still needs to check weather the request is a get or multi get by performing an additional split on the request string. For a more detailed explanation of the parsing implementation please refer to *Subsection 1.2*.
Another important behaviour that *Figure 9* depicts is the difference between the worker threads



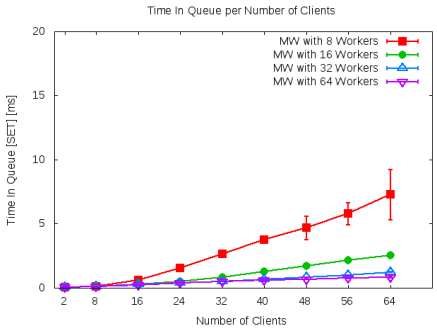
(a) Average Queue Length of get



(b) Average Queue Length of set

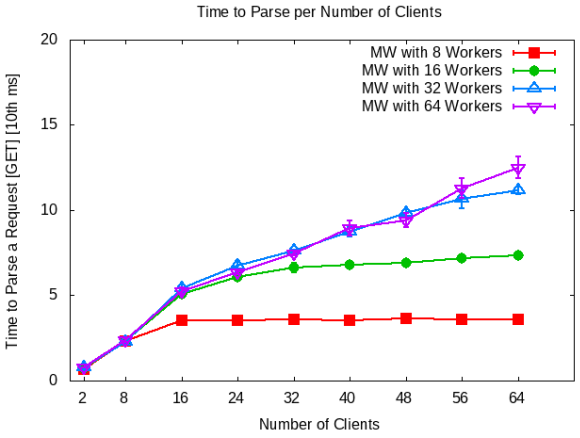


(c) Average Time in Queue of get

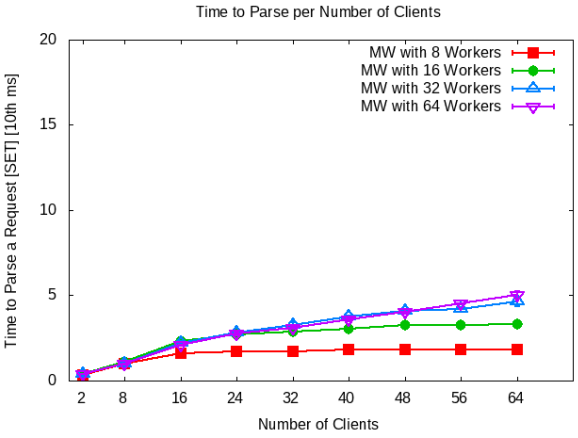


(d) Average Time in queue of set

Figure 8: Average time spent in the queue for both operations



(a) Average Time to parse get requests

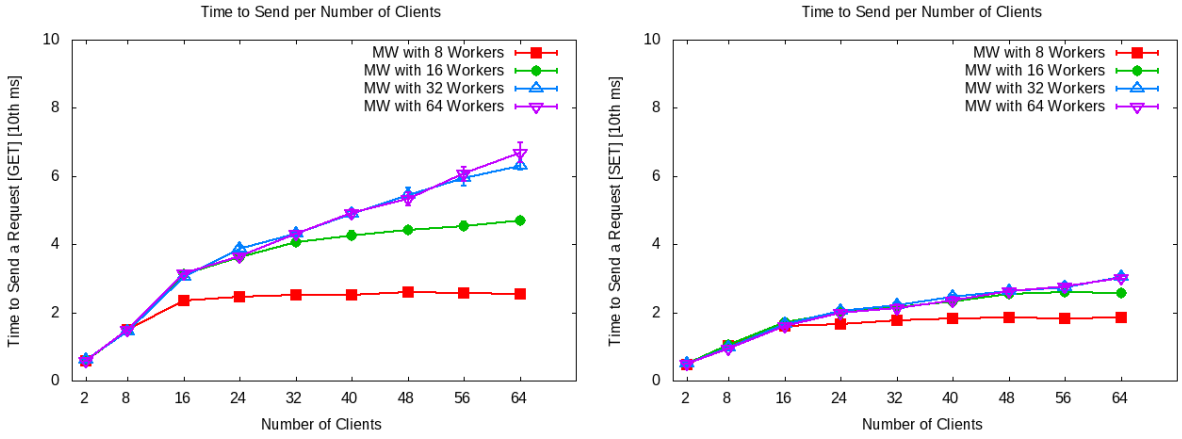


(b) Average Time to parse set requests

Figure 9: Average Time to parse both types of requests

configuration. We can see that the time to parse grows with the number of workers. This is due to how the statistics are gathered. Remember that statistics are collected using an `AtomicLong` and `AtomicInteger` values. This means that, in configurations with more threads, there will be more of them compeating for the atomic values, which will result in more overhead. This same reasoning can be applied to the fact that the time also slightly increases with the number of clients. With a small number of clients, the number of requests will be small and the threads

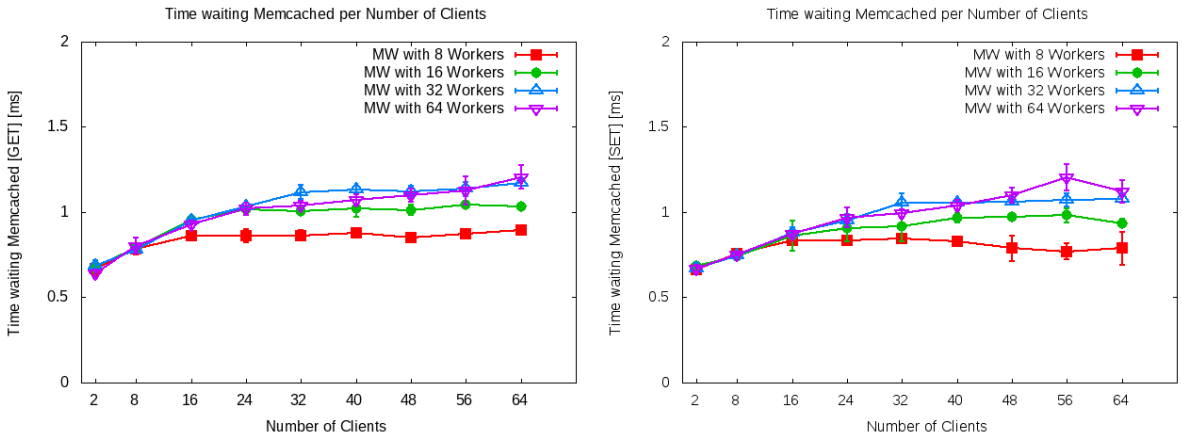
will be waiting most of the time in the pool. With more clients, the requests will increase and this will increase the number of threads at work, which will in turn result again in more threads compeating for the lock.
The next set of figures shows the time to send a request to the server. As for *Figure 9*, *Figure*



(a) Average Time to send get requests to the server **(b)** Average Time to send set response to the server

Figure 10: Average Time to send requests to the server

10 shows the same behaviours that can be explained as before. The time to send a request is roughly half the time to parse it. This is again due to the implementation, since during the parsing the worker needs to do a combination of string operations and switch checking. Interesting enough, the time to send a get is higher than the one to send a set. At first, this might seems a bit misleading. However, this is due to the fact that in this particular experiment only one server was used. Recall from the implementation details that the send time is started after the parsing. In the case of a set, the worker goes through the respective handle methods, which simply consists of a call to the `sendSetRequest`, where it then forwards the request to only one server, and stop the timer. On the other hand, during a get operation the worker needs first to compute the right server index, which imply a read and write of an `AtomicInteger`, which is not needed for sets.
The next statistic we are going to discuss is the time spent waiting for memcached. *Figure 11*

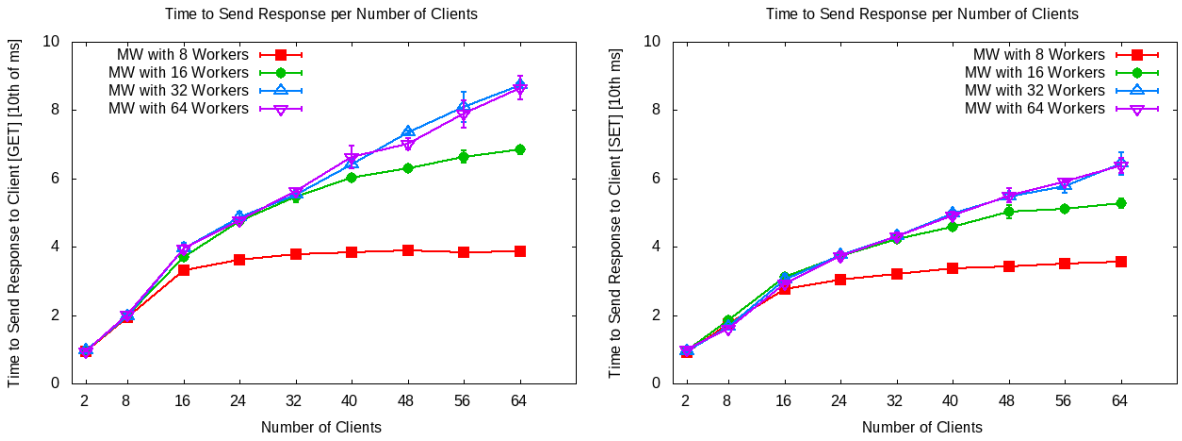


(a) Average Time waiting memcached for get **(b)** Average Time waiting for memcached for set

Figure 11: Average Time waiting for memcached

depicts the behaviour of the time waiting for memcached for both gets and sets. We can see that the operations have roughly the same waiting time, with sets being slightly faster. Differences in the configurations are given by a combination of the fact that more thread will put more load on the server, which will receive more requests in parallel and thus, since it will be more loaded, its response time will increase, and by again the atomic operations involved in collecting statistics.

We finally analyze the time taken to send the response back to the client. We can see that the



(a) Average Time to send get response back

(b) Average Time to send set response back

Figure 12: Average Time to send responses back

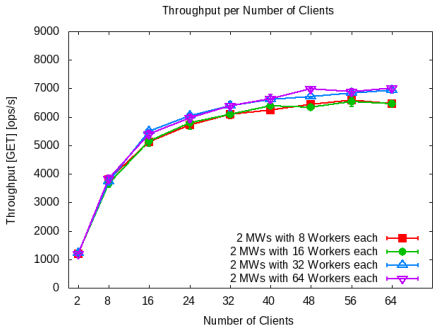
general behaviour is again very similar to the ones depicted in *Figure 9* and *Figure 10*. Gets are slower than sets, which is expected since the worker needs to send a 1KB value in the first case while it only sends a `STORED` string on the second one. Again, the more the workers, the more the time needed, both due to the atomic operations and for the increased number of threads pushing load on the network at the same time.

To conclude this subsection, we briefly analyze the Interactive Law results, that can be found under `asl-data/asl3/asl3.1/3.1-interactive_law.ods`. We can see that in the case of clients, the results almost always perfectly match, and when they don't, they differ very little. This is due to the think time to be not completely 0 even at the clients. On the opposite, we can see that on the middleware the values never matches, and differ greatly. The problem here is due to the fact that at the middleware the think time is never 0. This is because the assumption that when a worker has finished its job another one is ready is almost never true. Workers spend some time waiting in the pool before taking another job. The difference is even bigger when there are more worker threads. This is because, as we have observed when examining the queue length, this is really small in the configurations with more threads, meaning that many worker could spend some time waiting in the pool. Moreover, in this setup we had a relatively low workload on the middleware, which also increased the waiting time of the workers. Given these considerations, we used the experimental data to compute the average thinking time of the workers. The results can be found on the same table. We can observe that, in general, having more worker threads will result in them waiting for a longer time in the pool. This is expected as we previously explained.

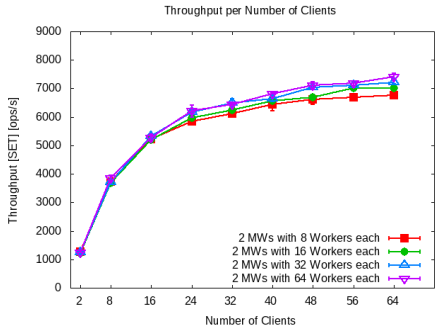
3.2 Two Middlewares

In this experiment we connect one load generator machine (two instances of memtier with CT=1) to two middlewares and we use 1 memcached server. We again run a read-only and a write-only workload with increasing number of clients (between 2 and 64) and measure both the response time and the throughput at both the clients and the middlewares. As before, we repeat the experiment for different number of worker threads inside the middleware: 8, 16, 32, 64.

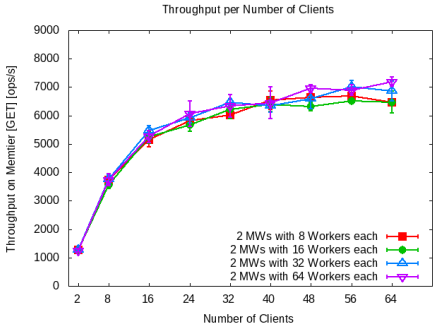
Number of servers	1
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1..32]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8..64]
Repetitions	3 or more (at least 1 minute each)



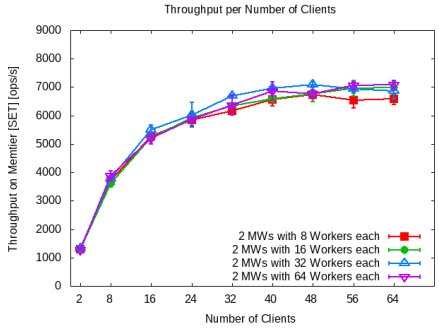
(a) Average Throughput for get measured on the mw



(b) Average Throughput for set measured on the mw



(c) Average Throughput for get measured on the clients

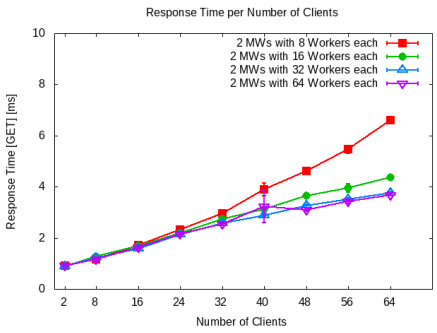


(d) Average Throughput for set measured on the clients

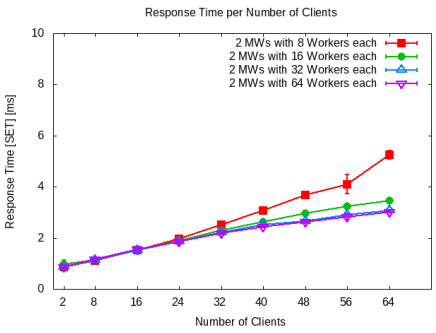
Figure 13: Average Throughput on middleware and clients

3.2.1 Explanation

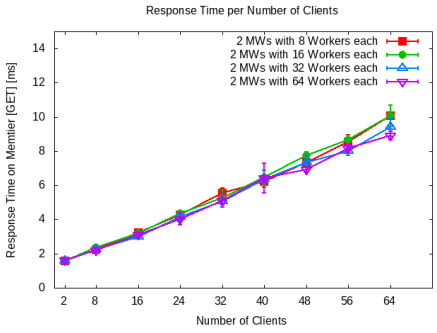
As in the previous experiment, we begin our explanation by observing the behaviours of throughput and response time for both types of operation measured both at the middlewares and at



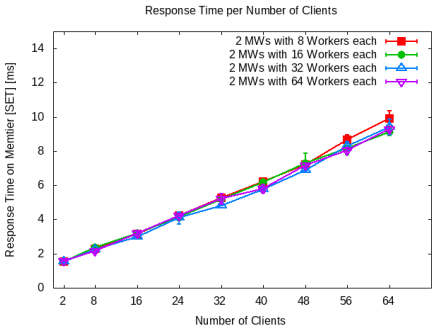
(a) Average Response Time for get measured on the mw



(b) Average Response Time for set measured on the mw



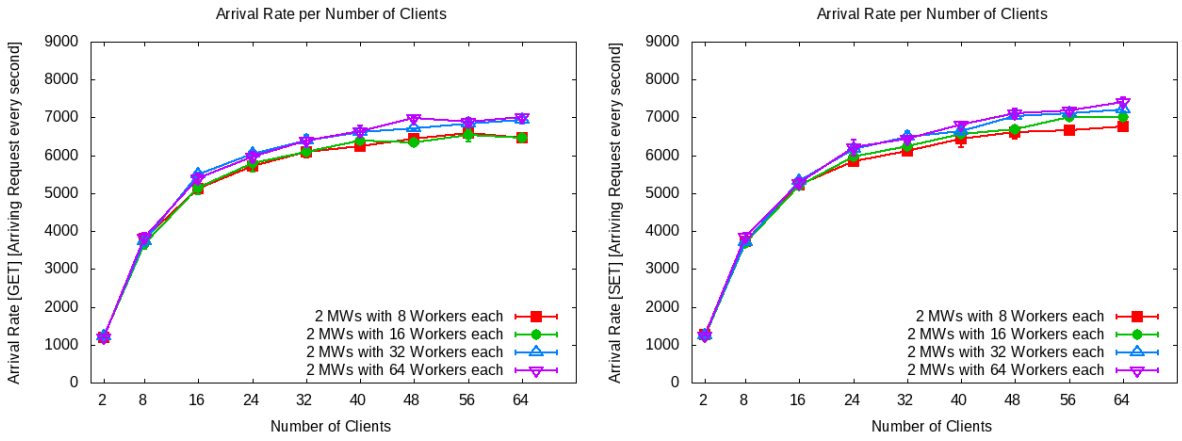
(c) Average Response Time for get measured on the clients



(d) Average Response Time for set measured on the clients

Figure 14: Average Response Time on middleware and clients

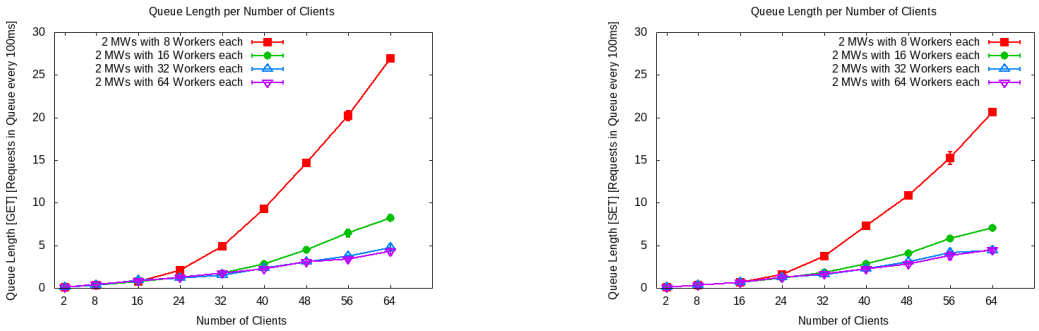
the clients. The first thing that we notice on *Figure 13* is that the throughput almost doubled with respect to the previous section. This is of course expected since we doubled the resources in the system having two middlewares. In fact, this is a first indication that our middleware is the bottleneck. If it was not the case, doubling the resources would not have produced such an increase in performances. Another thing that we notice is that once again set performs a little better than get. We can see in all four graphs that the system saturates with more workload with respect to the previous experiment, a behaviour that we expected since we doubled the middlewares which resulted in the system being able to handle more requests. We also start observing a little distinction between the configurations with less threads (8, 16) and those with more (32, 64) in terms of throughput too.
Figure 14 shows the response time. As can be seen, the response time has decreased by almost a half. Again, this is another indication that our middleware is the bottleneck, since doubling the resources improved the performances by a factor of roughly 2. Again we observe the 8 workers configuration as being the one which experiences the highest response time.
As before, we will now look at the other statistics to see how did their behaviour change with the increased number of middlewares.
Figure 15 shows the arrival rate measured in the new setup. As happened for the throughput, the arrival rate also increased by a factor near 2. This was expected since the middleware is sending back responses to the clients faster, which results in the clients sending more requests than before.



(a) Average Arrival Rate for get

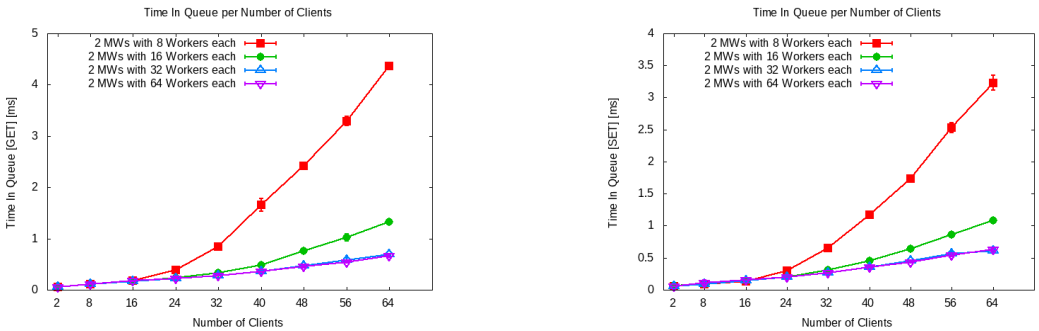
(b) Average Arrival Rate for set

Figure 15: Average Arrival Rate at the network threads



(a) Average Queue Length of get

(b) Average Queue Length of set



(c) Average Time in Queue of get

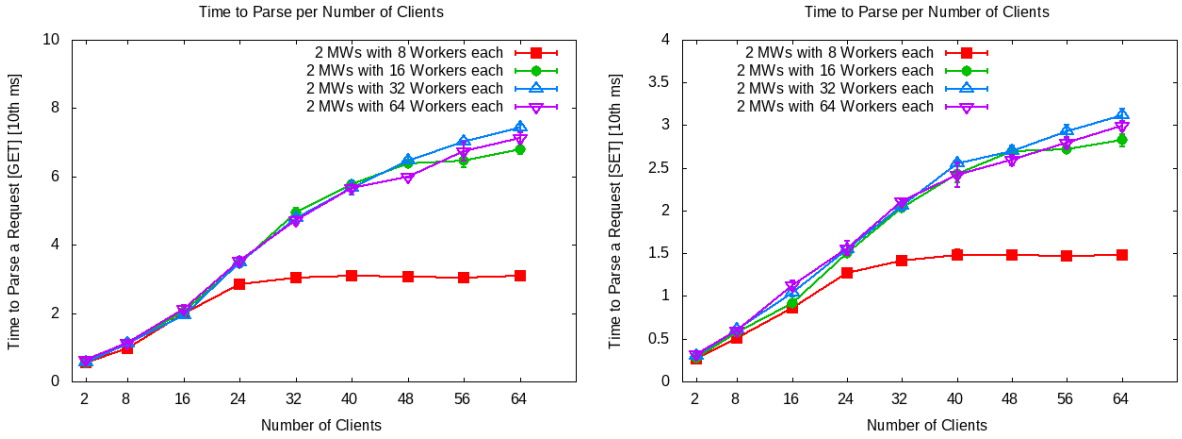
(d) Average Time in queue of set

Figure 16: Average time spent in the queue

As we can see from *Figure 16*, the queue length has decreased as well by a factor, even if not as much as the previous metrics. The intuition behind this is strictly related to the previous graphs of throughput and arrival rate. What we have seen before is that this two have both increased by almost 2 times. This means that our middleware is performing operations two times faster, but also that it has two times more jobs to do. The result is obviously that the queue length can not decrease as much. However, even if the total number of requests in the

queue does not decrease as much as, for example, the response time, we can see that the time spent in the queue by the single requests has halved, and even more. In this case, the same reasoning that we did for the whole middleware can be applied. Recall that in the previous experiment we said that the queue was the middleware's bottleneck, since the time spent in queue was a big part of the total response time. In this case, we can see that doubling the resources (*i.e.*, the number of queues, since two middlewares implies two queues) the time spent in the queue decreased by more than 2.

Figure 17 shows the time spent in parsing for both types of requests. In this case, we used



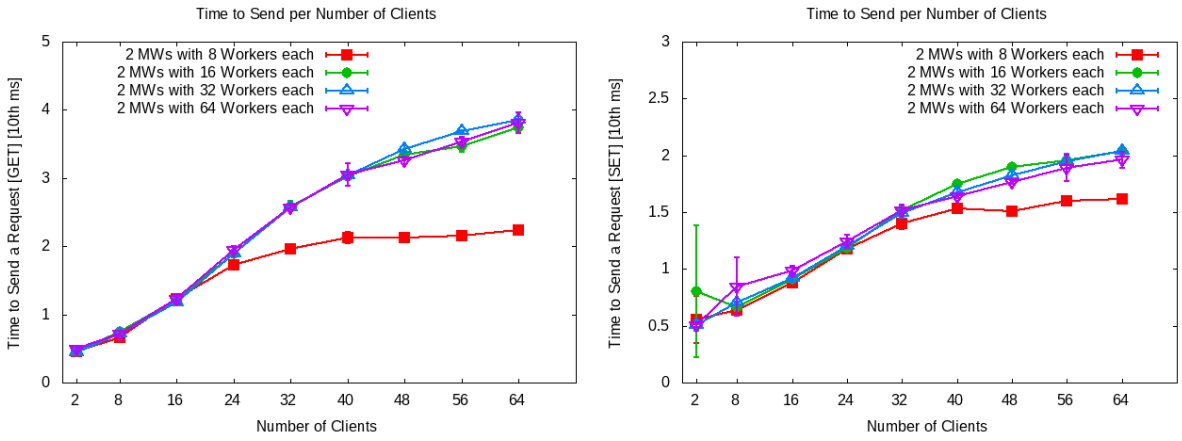
(a) Average Time to parse get requests

(b) Average Time to parse set requests

Figure 17: Average Time to parse both types of requests

two different scales given the difference between the two. As before, sets are parsed faster than gets, which is not surprising since the process that the worker threads need to perform has not changed. We can see that in comparison to the previous experiment, the time to parse decreased as well. This might seems contradictory at first, since one could expect the time to parse to be the same. However, recall that this is the average time to parse requests over the total number of requests performed in a unit of time. Since the throughput has almost doubled, in the same time as before the system is parsing twice as much requests, meaning that the average time will decrease accordingly. Again, we observe that more workers imply a longer time to parse requests. This behaviour has already been encountered and discussed in the previous experiment.

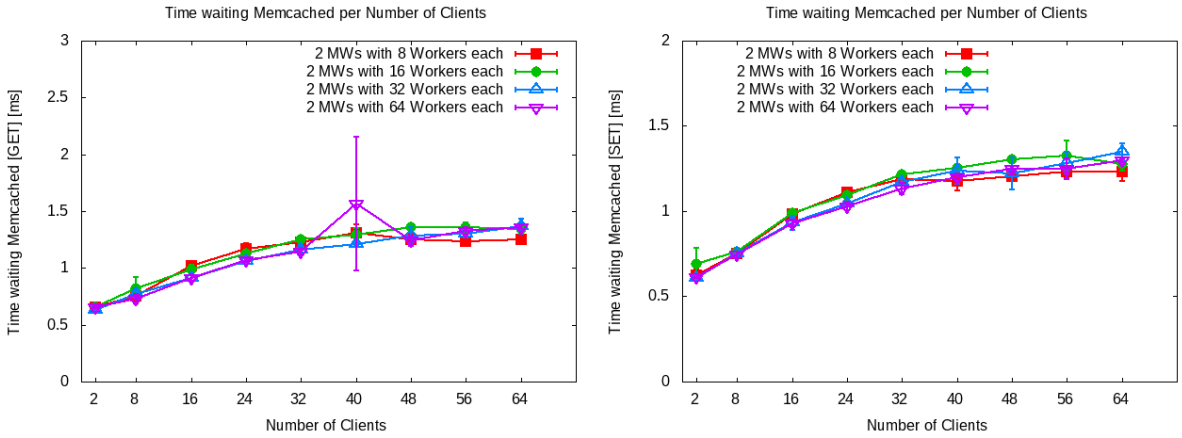
Figure 18 represents the time to send requests to the server. As in the previous experiment, sending is faster than parsing and sending a set is faster than sending a get. Again, the general behaviour is the same as in the previous experiment. As for the time to parse, the time to send has decreased. We can note, however, that the change is not as much important as for previous statistics, particularly for the configuration with less threads. This is probably because we are reaching the lower bound on the time to send the requests. At a certain point, even increasing the resources, the performance boost will stay low or actually be 0. Also, in contrast with the parsing, sending requires access to the network which is more susceptible to the increasing number of requests that are flowing through it. An unexpected behaviour is present in *Figure 18b*. We can observe that for 2 clients and 16 threads, the time to send is higher and highly instable. Looking at the raw data, we can see that this is due to a single unit of time in the first repetition being higher than the others by 3 orders of magnitude. This is probably due to the garbage collector startin the execution and stopping the other processes at that particular time, which makes the average sending time to be highly affected by this outlier.



(a) Average Time to send get requests to the server **(b)** Average Time to send set request to the server

Figure 18: Average Time to send requests to the server

Figure 19 shows the behaviour of the time waiting for the server to reply. In contrast with



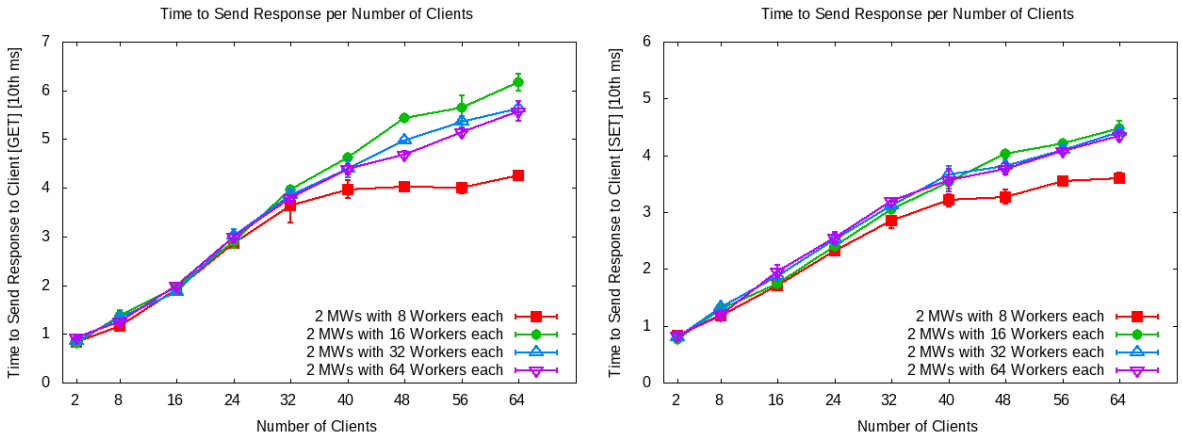
(a) Average Time waiting memcached for get **(b)** Average Time waiting for memcached for set

Figure 19: Average Time waiting for memcached

the others statistics encountered so far, we can see that this waiting time has slightly increased with respect to the previous experiment. This is in fact what we would expect from the server, since the load on it has increased given that the workers are sending more requests than before. Also, we notice a behaviour similar to the sending time at 40 clients and 64 workers. Again, one of the data is higher than the others by 2 order of magnitude, and again we can suppose that this is due to the garbage collector.

The last figure on the next page shows the time taken to send the response back to the client for this particular setup. As for the case of the time to send the request to the server, the time to send the response back has slightly decreased and, for the configurations with less threads, has stayed the same. A similar reasoning to the one made for the time to send to the server can be applied in this case.

We again conclude this subsection by briefly analyzing the Interactive Law results, that can be found under `asl-data/asl3/asl3.2/3.2-interactive_law.ods`. As before, we can see that the law holds for the clients, while it overestimates, sometimes by far, the results of the middlewares. Again, the same reasoning applies. As in the previous section, we used the



(a) Average Time to send get response back

(b) Average Time to send set response back

Figure 20: Average Time to send responses back

experimental data to compute the thinking time. The behaviour is very similar to the previous section, with a higher thinking time with higher number of worker threads. However, this time has almost halved. This is a consequence of the throughput doubling, since to find Z we need to divide by X which is roughly two times higher than in the previous experiment.

3.3 Summary

The following table summarizes the findings regarding the maximum throughput and corresponding response time, time in queue and miss rate, for both the setups.

Maximum throughput for one middleware.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware	3717	1.12 ms	9.34	0
Reads: Measured on clients	3821.8	12.56 ms	n/a	0
Writes: Measured on middleware	3993.55	3.21 ms	0.68	n/a
Writes: Measured on clients	4063.8	13.81 ms	n/a	n/a

Maximum throughput for two middlewares.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware	7007.04	3.68 ms	0.66 ms	0
Reads: Measured on clients	7197.27	8.93 ms	n/a	0
Writes: Measured on middleware	7418.25	3.01 ms	0.63 ms	n/a
Writes: Measured on clients	7096.47	6.89 ms	n/a	n/a

With these two experiments we were able to examine the different behaviours of the system when one or two middlewares were deployed. In the first part we saw that, in terms of throughput, changing the number of worker threads did not make so much difference. In fact, the highest throughput for get operations in this experiment was achieved with 8 worker threads. However, in terms of response time and time spent in the queue we could already notice the difference between configurations, with less threads resulting in higher times. We also noticed

that, since only one server was used, the set operations were still faster than gets, but that the gap was reduced. A similar behaviour was observed on the clients, where the response time was obviously higher but the general behaviour was the same. Also, we saw that the Interactive Law hold in these situations too for the clients, while we observed that the assumption that worker threads are always performing a job is quite optimistic and not met in reality.

In the second part we saw the increased performances of the system as we incremented the number of middlewares. The throughput almost doubled while the response time almost halved. This told us that our middleware was the bottleneck of the system, since doubling the resources corresponded to an increment in performance by a factor of 2. We observed the same behaviour on the queue, which is the middleware's bottleneck. We saw that the length of the queue did not decrease so much because of the mutual increase of the arrival rate and throughput, but we also observed a drastical decrease in the time spent waiting in the queue, which corresponds to an important amount of the total response time. In fact, we noticed that the other times measured did not get affected so much by the increases in the resources, still, the response time got halved due to the important improvement on the time spent in the queue.

4 Throughput for Writes (90 pts)

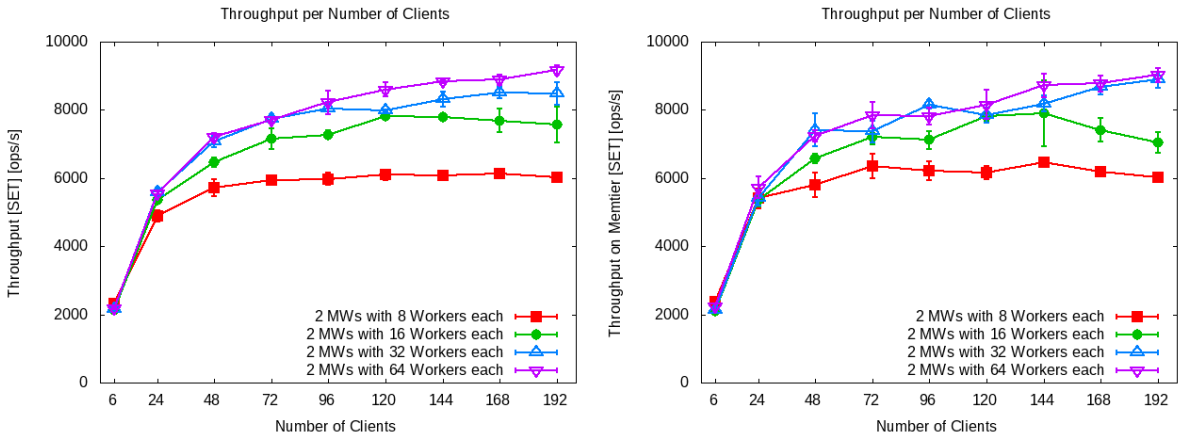
4.1 Full System

In this section we used three load generating VMs, two middlewares and three memchached servers to run a write-only experiment. As before, we measure the throughput and response time both on the clients and on the middlewares, with four different worker threads configurations (8, 16, 32 and 64). To be sure that worker threads forwarded requests to all servers, we kept track of the number of requests sent by each worker to the three servers. The workers behaved as expected. The results can be found under `asl4` in one of the two `finished` folders, under `SL`.

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..32]
Workload	Write-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8..64]
Repetitions	3 or more (at least 1 minute each)

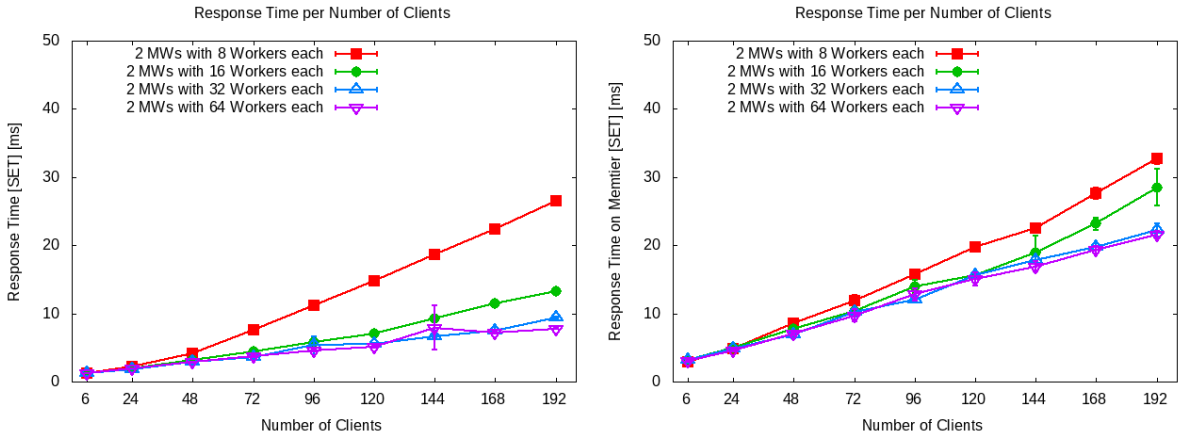
4.1.1 Explanation

As for the previous sections, we start by explaining the graphs representing the throughputs and response times as measured on both the clients and the middlewares. *Figure 21* shows the throughput of the system both at the middlewares and at the clients. The first important notice that we can make comparing with the previous experiments is the difference between the worker threads configurations on the middlewares. While previously, in terms of throughput, all the configuration behaved more or less the same, in this case we can clearly see that with more threads the system performs better. This is expected since we now have a full system with 6 times more clients, and thus, especially the configuration with 8 thread can not sustain such a high load. Moreover, in this case we have a write-only with three servers. Recall that sets



(a) Average Throughput as measured on the middlewares **(b)** Average Throughput measured on the clients

Figure 21: Average Throughput both at the clients and at the middlewares



(a) Average Response Time as measured on the middlewares **(b)** Average Response Time measured on the clients

Figure 22: Average Response Time both at the clients and middlewares

are forwarded to all three servers, so the threads have to do additional work with respect to the previous situations. Given these differences, the system saturates at different points for different numbers of worker threads. We can see that with 8 workers the plot is almost completely flat at 48 clients. For 16 threads, the throughput flattens at 72, but then increase again between 96 and 120 and then starts decreasing. With 32 threads we can see that the plot is flat at 96 and increases slightly again between 120 and 144, while for the 64 workers the throughput keeps increasing slightly until 144 clients. Looking at the throughput measured at the clients, we can observe that with higher number of workers the plot is more unstable. This could be due to the fact that requests have to go through another layer of the network, which is more suscebtible to variation with high number of packets being sent, which is the case with higher configurations. *Figure 22* depicts the response times. These plots confirm what we said previously for the throughputs. The configurations with more workers are the better performing. Looking at both the response time and the throughput, we can draw some more precise conclusions about the saturation of our system. With 8 workers, the response time increases from 1 ms at 6 clients to 4 ms at 48 and to 7 ms at 72, meaning a slow down by a factor of 4 and 7, even if the throughput slightly increases. With 16 workers the response time increases by a factor of 4 between 6 and 72 clients, while to increase by 7 times we need to go up to 120 client. To

have the same increases with the last two configurations, we need to go from 6 to 72-96 in the 32 workers case for a 4 times increase, while for a seven times the clients are 168. Similarly in the 64 threads case, a factor of 4 is met at 96 and of 7 at 164 clients. In general, the 64 threads configuration slows down by a factor slightly higher than 7 between 6 and 192 clients, while the 8 workers configuration slows down 26 times in the same range. Comparing the two configurations with more workers we can also draw some interesting conclusions. Both in terms of response time and throughput, the two performs similarly despite one having two times more workers than the other. A higher difference would probably be observed increasing the number of clients, when the overhead of managing 64 workers would become negligeable in comparison with the gain in terms of throughput and response time over the 32 workers.
We will now take a look at the other statistics gathered during the experiment.

As done in the previous sections, we start by showing the arrival rate at the network thread

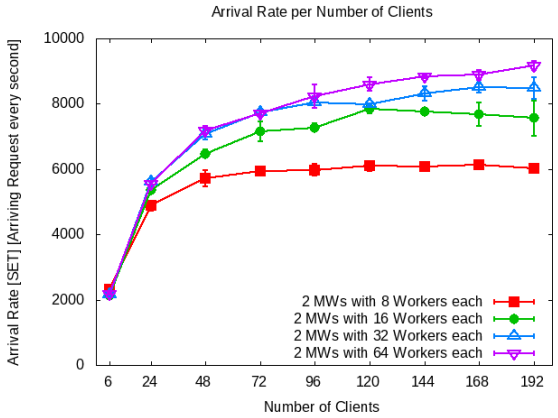
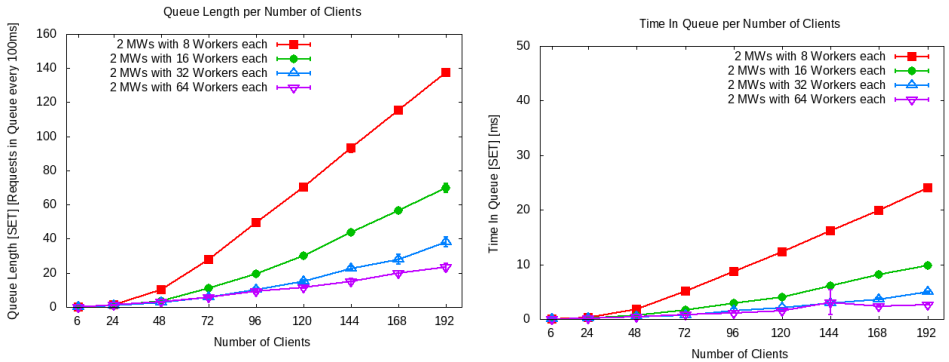


Figure 23: Average Arrival Rate at the network thread in requests per second

for this setup (*Figure 23*). As expected, the behaviour is the same as for the throughput. In general, the latter is slightly higher than the former, which is something we expect. However, the two are really close and in some cases the arrival rate is higher than the throughput. This means that the system is not stable and that, since there are more requests arriving than leaving the system, the queue is going to grow unbounded.

Figure 24 shows the queue length and the time spent in the queue. As for the previous



(a) Average Number of requests waiting in the queue

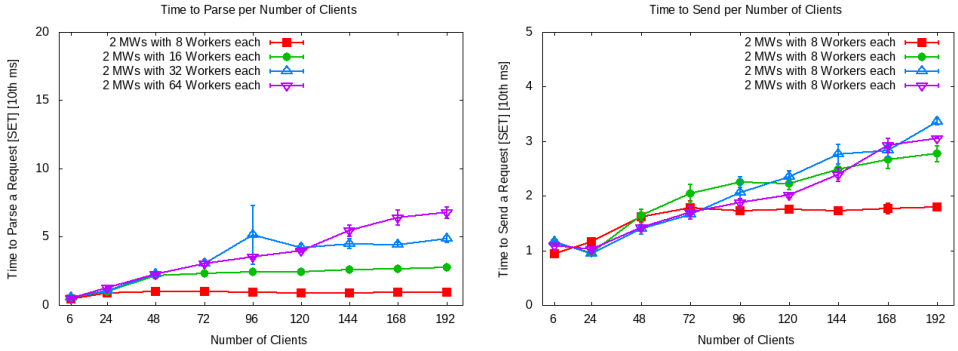
(b) Average waiting time in the queue

Figure 24: Average queue length and time in queue

metrics, we observe that the difference between configurations is important. In terms of queue

length, we can see that even between the 32 and 64 configurations there is a difference of almost 20 requests. This is expected since we have 2 times more threads taking requests out of the queue. However, looking at the time spent in the queue we can see that the difference is not so pronounced. However, we notice that starting from 168 clients, the two graphs start differing. This is the same behaviour that we observed for the throughput and response time, and is expected since as we said the gain of having 64 workers start becoming dominant over the overhead of managing them all. As in previous experiments, we can see that the time in queue corresponds to a big part of the total response time, especially for the 8 and 16 configurations. This is expected since the queue is the bottleneck of the middleware.

Figure 25 shows respectively the time to parse and send requests. As in the previous section,



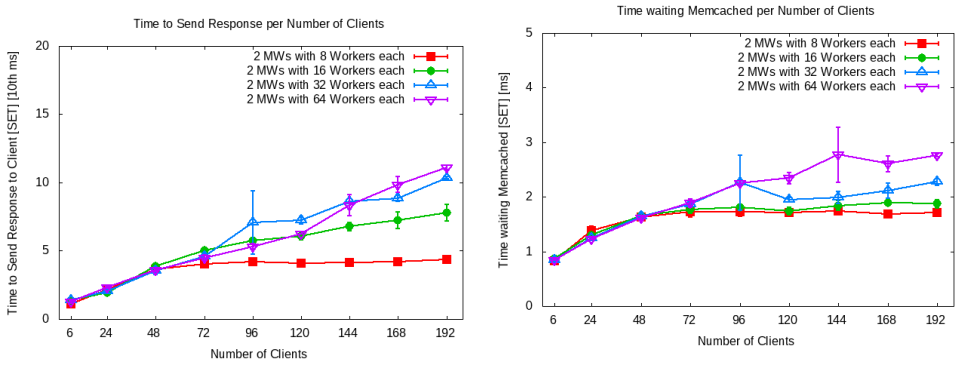
(a) Average Time to parse requests

(b) Average Time to send requests to server

Figure 25: Average time to parse and send requests

the difference between the configurations of workers can be explained by the higher number of threads compeating for the atomic resources. Also recall that every second the `TimerTask` wakes up and will compete for the locks too. Another interesting behaviour is the one observed in *Figure 25a* and *Figure 26a* with 32 workers and 96 clients. We can see that both statistics are highly instable and suddenly make a jump. Looking at the raw data, we can see that in both cases there are a few of them that are higher than the others by 1 or 2 orders of magnitude. This could be due to the java garbage collector waking up and freeing memory, while stopping the workers' execution, a behaviour that we already encountered in the previous section.

The last figure shows the behaviour of the time to send responses back to clients and time



(a) Average Time to send response to client

(b) Average Time waiting for memcached servers to reply

Figure 26: Average Time to send respons to clients and waiting for servers

waiting for the servers. Comparing with *Figure 25* we can see that the first one is the highest time

between the ones related to the workers processing. This is expected given the implementation of the method which handles this operation. Recall that the `sendResponseToClient` method needs to allocate the buffer, convert the string to bytes and write them into the buffer before sending back to the client. Also recall that this method always allocates enough space for up to a response containing 10 KB data. This is a waste in case of a set which only needs space for a `STORED` string. Looking at *Figure 26b* we can see that there is a difference between the configurations which is expected, since the more the threads, the more the load that will be put on the servers at the same time, resulting in a higher response time of the servers. Again we can see some sudden jumps in the graph, and again we can suppose that this is due to the garbage collector.

4.2 Summary

The following table shows some statistics corresponding to the maximum throughput points in all the four configurations.

Maximum throughput for the full system

	WT=8	WT=16	WT=32	WT=64
Throughput (Middleware)	6141.19	7830.92	8503.16	9174.60
Throughput (Derived from MW response time)	7499.34	17061.15	22476.36	24691.75
Throughput (Client)	6473.93	7899.27	8884.87	9020.67
Average time in queue	19.94	4.06	3.53	2.65
Average length of queue	115.40	29.86	27.93	23.26
Average time waiting for memcached	1.68	1.74	2.11	2.76

The table summarizes well what have been said so far. As expected, for both clients and middleware the maximum throughput is achieved with the highest number of worker threads, since more parallelism is added to the system. Also, this is possible because, in contrast with the previous section, in this experiment we had a higher number of clients, which pushed the system to the limit, giving a great importance to the more resources added with more workers. We can observe that the configuration with 8 worker threads is by far the less performant, as we discussed in the explanation. An important notice to make is that the configuration with 16 workers reaches highest throughput at 120 clients while the same happens at 168 and 192 for the 32 and 64 configurations. This is why there seems not to be so much difference between this three configurations looking at the table, while as we have seen the last two configurations are much better performing when we have a higher load. Also as we expected, the 64 worker threads configuration reaches highest throughput at 192 clients, when the load is such that we have the best ratio between gain in performance and overhead of having so much threads. The last row simply tells us the the time waiting for the servers grows with the number of workers, which is what we expected since more threads are sending requests to the servers at the same time, increasing the load that they have to process.

Finally, the second row contains the throughput derived using the **Interactive Law** using measured response time. Recall that this law states that

$$R = \frac{N}{X} - Z \iff X = \frac{N}{R + Z}$$

and that we took our thinking time $Z = 0$. This assumes that, once a job has been processed, another one is immediately ready, and this assumption holds on the clients side, since once they receive a response, they immediately send another request. However, on the middleware

this assumption is not correct, because it assumes that all the worker threads are always working. This is of course not true, especially for the higher worker threads configurations where in many cases some of the workers will idle for some time. This can be observed by the fact that the difference between measured throughput and derived one increases with more worker threads. This is because the lower configuration is the one where the assumption of threads always working is closer to the truth, since the queue length grows and the workers spent less time idle with increasing number of clients. The complete results of **Interactive Law** can be found in `4_interactive_law.ods` under the `asl4` folder. We again used the experimental data to compute the thinking time Z. The behaviour was very similar to the one observed in the previous section.

To conclude this section, in this experiment we pushed our system to the limit and we observed that, as expected, the more the workers the better the performances. We also observed that there is more difference in performance between 8 and 16 workers or 16 and 32 than between 32 and 64, because in the latter case the overhead of managing this number of threads is still important with this number of clients. Finally we saw that, as previously, the queue is the bottleneck and after the system saturates, especially with 8 and 16 workers, the length starts growing unbounded.

5 Gets and Multi-gets (90 pts)

For this set of experiments we will use three load generating machines, two middlewares and three memcached servers. Each memtier instance has 2 virtual clients in total and the number of middleware worker threads is the one that provides the highest throughput in our system, namely 64. In both experiments we again used `memcached_pop.sh` to populate the servers, and we can see in the respective `CM` files that the miss ratio is mostly 0 and when it is not, it is very small (recall that the outputs shown in these files are already percentage, so 0.24 is in fact 0.24%). We also keep track of the server load to be sure that workers are performing round robin when the request is a get. We can see that this is achieved by looking at the `SL` files.

5.1 Sharded Case

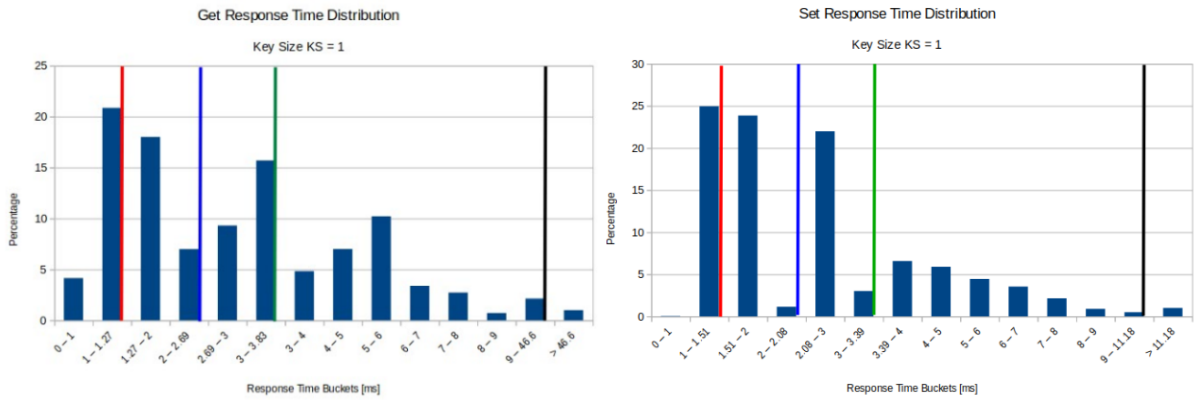
In this experiment we run multi-gets with 1, 3, 6 and 9 keys with sharding enabled.

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	memtier-default
Multi-Get behavior	Sharded
Multi-Get size	[1..9]
Number of middlewares	2
Worker threads per middleware	max. throughput config.
Repetitions	3 or more (at least 1 minute each)

The following figures show the response time distributions for both get and set operations in case of a key size of 1, 3 and 9. We defer the discussion of the case of a key size of 6 to *Subsection 5.3*. In all cases, the bucket size is 1 millisecond, except for cases where the percentile is shown, which are taken in the interval $[t_i; percentile]$ or $[percentile; t_i]$, and the last two buckets, which are always of the form $[9; 99percentile]$ and $[99percentile; t_{max}]$. Also, in each histogram there are four vertical line representing the 25, 50, 75 and 99 percentiles, meaning that the part of histogram on the left of these lines are respectively 25, 50, 75 and 99% of all the requests. The

y-axis always represents the percentage of total requests.

Figure 27 represents the distributions for a key size of 1. We can see that for both sets and gets



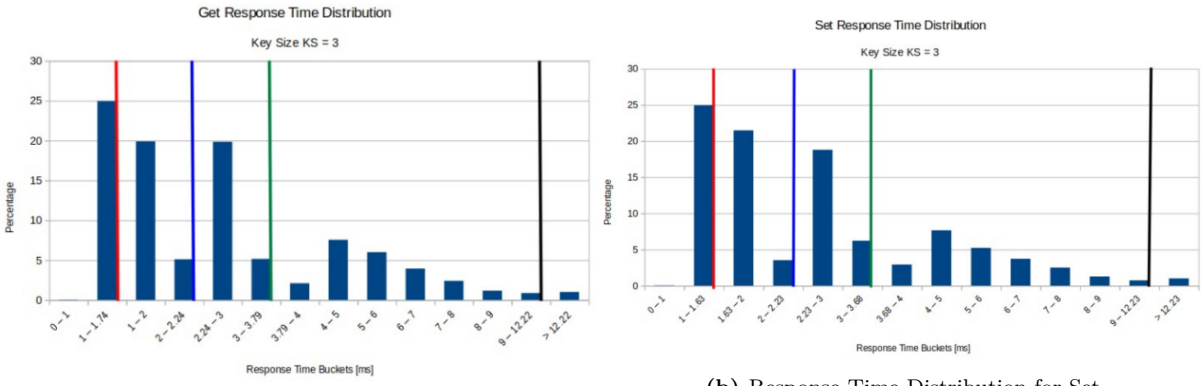
(a) Response Time Distribution for Get

(b) Response Time Distribution for Set

Figure 27: Response Time Distribution for a key size of 1

75% of the requests have a response time of less than 4 ms. Roughly 4% of get requests take less than 1 ms to be completed, while in this same interval the number of sets is less than 1%, which is not surprisingly given that sets are forwarded to all servers. In general the operations behaves similarly since most of them are completed in less than 4 ms, but we can see that sets are more regular since most of the 75% fall in one of the three higher bins.

With a key size of 3 (*Figure 28*), we can see that the operations behave very similar. Altough



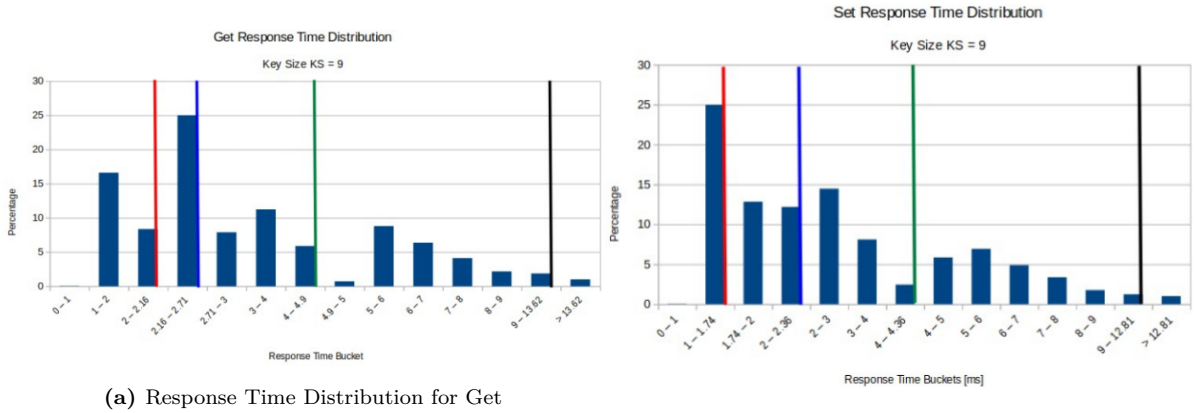
(a) Response Time Distribution for Get

(b) Response Time Distribution for Set

Figure 28: Response Time Distribution for a key size of 3

we can see that sets are still distributed in a similar way to the previous case, which is expected since the key size does not influence directly this type of operations, gets start to become more regularly distributed and in a way similar to sets. This is again not surprising given the implementation of multi gets and sets, which are similar since they both need connections to three servers. Of course, the canging in beahviour of get operations is due to the overhead of splitting the keys eveanly.

In the last case we can see that both operations have a higher response time, given by the distribution being ”shifted” to the right. We can see that the 75 percentile has been shifted by more than a millisecond in the case of gets. This is understandable since we expect the time needed to split the multi get as well the time needed by the servers to fetch all the values to grow with the key size. Since gets will take more time to be completed, this will affect set



(a) Response Time Distribution for Get

(b) Response Time Distribution for Set

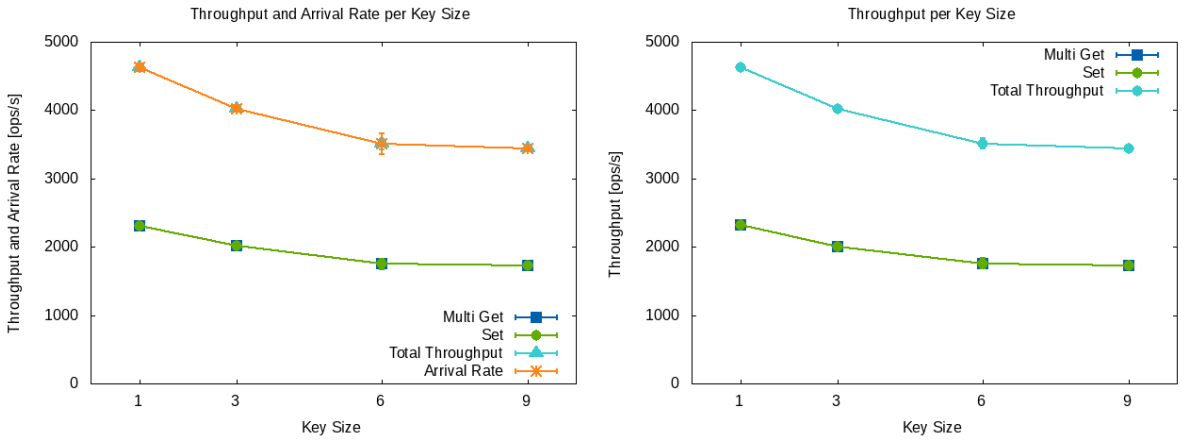
Figure 29: Response Time Distribution for a key size of 9

operations as well since they will have to wait more than in previous situations. As we said, this can be seen on the distribution which is still similar to the previous cases but that also shows that the response time is growing. For example, we can see that the third bin, which almost cover the same interval, went to more than 20% to roughly 12%, and the fourth one increased from less than 5% to more than 10%.

5.1.1 Explanation

As always we start by showing the throughput and response time as a function of the key size both at the middlewares and the clients.

Figure 30 shows the throughput as a function of key size, for both get and set operations, as



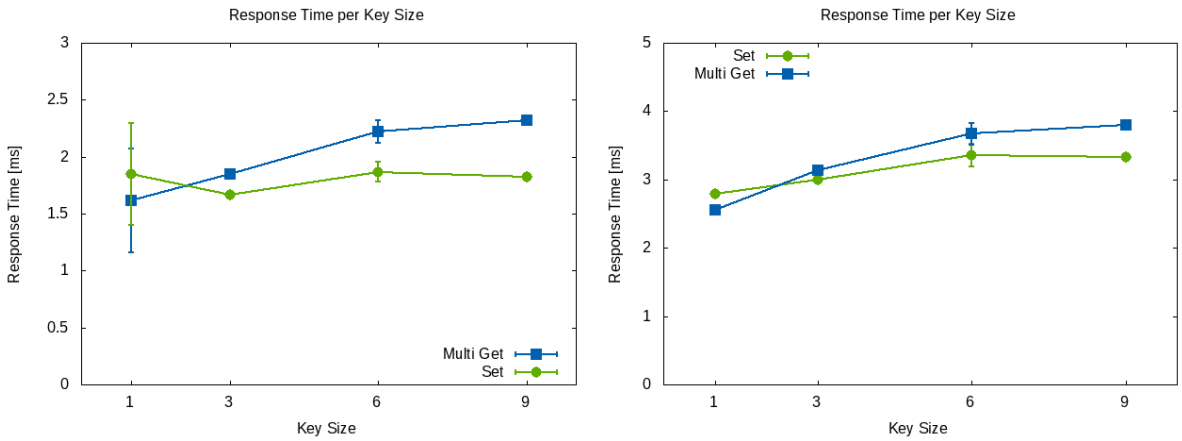
(a) Average Throughput measured on the middlewares

(b) Average Throughput measured on the clients

Figure 30: Average Throughput as a function of key size

well as their sum and the arrival rate at the network thread. We can see that, as one could expect, the throughput decreases with the number of keys in a get request. We notice that both gets and sets have exactly the same throughput. This is because the clients are instructed to send requests with a 1:1 ratio, which result in the same number of operations per unit of time, on average. Also, we observe, as in previous sections, that the arrival rate is very similar to the throughput. Looking at the single data we can see that the former is almost always slightly lower than the latter, which means that the utilization is almost 100%.

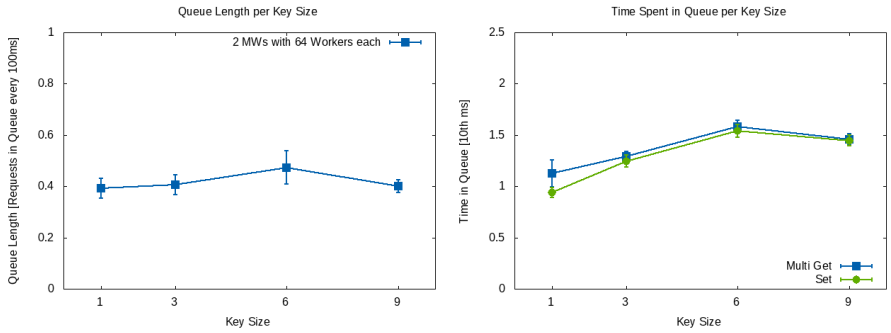
On *Figure 31* the response time as a function of the key size is shown. As we can see, with



(a) Average Response Time measured on the middlewares **(b)** Average Response Time measured on the clients

Figure 31: Average Response Time as a function of key size

a key size of 1, sets have a higher response time than gets. Recall that in this setup we are using three servers, and recall that sets are forwarded to all of them, while multi gets with only one key are treated as normal gets. This explains why we observe this behaviour. We can also see that at the middleware, with one key we have a high standard deviation. This is again probably due to the garbage collector, since looking at the data on the first middleware, we can see that during two period of one second each, we have an increase in response time of one order of magnitude. With increasing number of keys in get operations, we can observe that the response time behaves the opposite. Recall that multi gets are treated similarly to set, since, in the sharded case, the worker needs to forward each chunk of request to the appropriate server. Moreover, multi gets need to be first split into multiple get operations.
We can see that with a key size of 6, the response time can be slightly higher than the other cases, and in general this case shows some instability. This can be due to the implementation of the middleware, that in case of a key size of 6, writes the response time of every request to a file, causing a little overhead. However, in the case of gets, the decrease in slope for response time can also be partially explained by the fact that the gain of splitting requests over multiple servers start becoming dominant over the overhead due to the splitting process. The next set

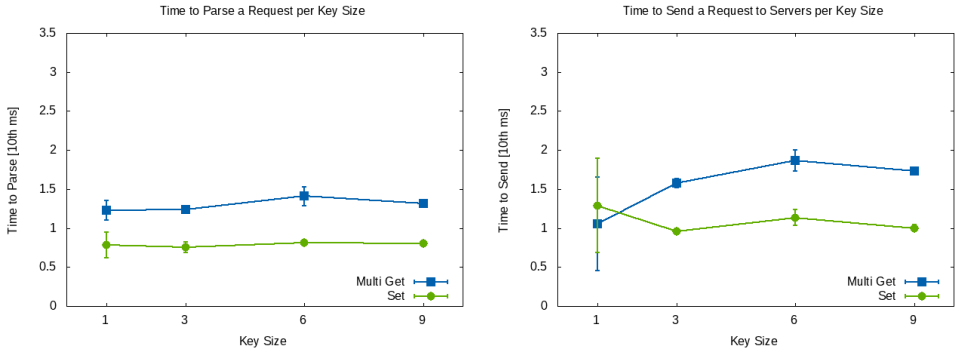


(a) Average Number of requests waiting in the queue **(b)** Average waiting time in the queue

Figure 32: Average queue length and time in queue

of figures shows the average queue length and the time spent in queue as a function of key size. The queue length is almost constant in both the cases, with again at a size of 6 a little

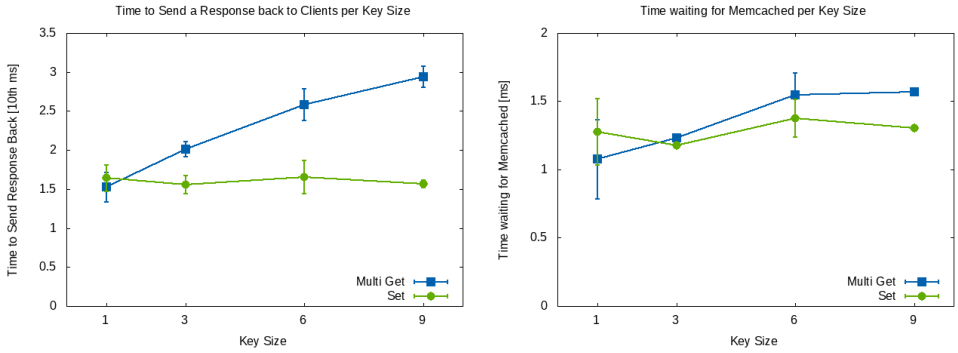
increase probably due to the additional statistics gathered. The time spent in queue increases slightly with the number of keys, but in general for both the statistics we can notice values far below the ones we encountered in the previous sections. This is because this two metrics are mostly affected by the number of clients. In this case, with only 2 clients for each memtier instance (12 clients in total), the workload is not high enough to make the queue grow unbounded.



(a) Average time to parse requests **(b)** Average time to send requests to server

Figure 33: Average time to parse and send requests

Figure 33 shows the average times necessary to parse and send requests to the servers. The time to parse a request is almost constant, except for the little additional overhead at key size of 6. As already observed in previous sections, the time to parse a get request is higher due to the implementation of the `parseRequest` method.
The time to send requests to the servers behaves as expected. With a key size of one, sets require more time to be sent since they need to be forwarded to all servers. As the key size increases, the behaviour changes and multi get are slower, since they also need to be split in multiple pieces and then sent to the appropriate servers. Again, we notice the little overhead at key size of 6, and the effects of the garbage collector at key size of 1.
Figure 34 shows the average time taken to send responses to clients and that workers wait for



(a) Average time to send response to client **(b)** Average time waiting for memcached servers

Figure 34: Average time to send response back to clients and waiting for servers

the servers. The first one is almost constant for the set operations, since they are not directly affected by the incresing keys. For multi get, the time is constantly increasing. This matches our hypothesis since in this case, the actual values corresponding to the keys have to be send back. Since the miss ratio is almost always 0, with 9 keys we have 9 KB of data that has to be sent. In this case we can see that, even if with a key size of 6 we have additional costs due

to the worker constantly recording the response times, the increasing key size has a dominant effect, since we have 3 KB more to send on the network with 9 keys.
As in the case of the time to send, sets are slightly slower to process by the servers when the key size is 1. This is because the worker needs to wait the response from all the three servers before continuing the operations. Again, with increasing key sizes, gets become slower. This is exactly what we would expect since both operations need to wait for three servers to respond, but in case of a multi get each server needs to fetch 1, 2 or 3 values, each one of 1 KB. We can see that the time spent waiting for the servers is the majority of the response time, as it was previously the case for the time spent in the queue. This tells us that in this particular case, the servers are the limiting factor of the system, which is expected since the workload of this experiment is coming from the amount of data that a multi key get operation needs to treat. This data is almost only dealt with at the servers, which neeed to fetch all the values and send them to the middlewares. The workers need to treat the data only when sending them back to the clients. This happens in the `sendResponseToClient` method which, not surprisingly, is the second most time consuming part of the process, meaning that in the middleware this method is now the bottleneck.

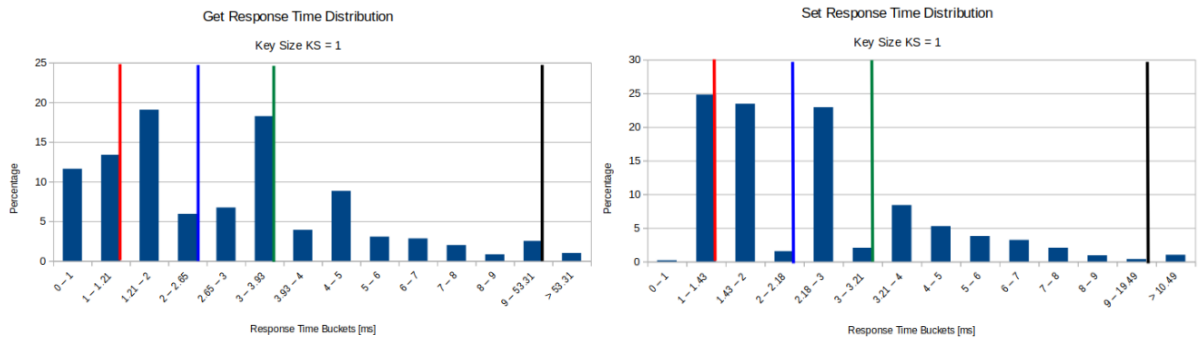
As for the previous section, we again computed the **Interactive Law**. The results can be found under `asl-data/asl5/5.1/5.1_interactive_law.ods`. In this case, the experimental data almost perfectly matched the derived one, meaning that the thinking time Z was approximately zero.

5.2 Non-sharded Case

In this experiment we run multi-gets with 1, 3, 6 and 9 keys with sharding disabled.

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	memtier-default
Multi-Get behavior	Non-Sharded
Multi-Get size	[1..9]
Number of middlewares	2
Worker threads per middleware	max. throughput config.
Repetitions	3 or more (at least 1 minute each)

As in the previous part, we show the response time distributions for both get and set operations in case of a key size of 1, 3 and 9. We again defer the discussion of the case of a key size of 6 to *Subsection 5.3*. As before, the bucket size is 1 millisecond, except for cases where the percentile is shown. In the case of gets with 9 key, the 75 percentile corresponded with one of the interval edges. Thus, we decided to include one more interval, namely $[9; 10]$, to have a consistent number of buckets in each histogram.



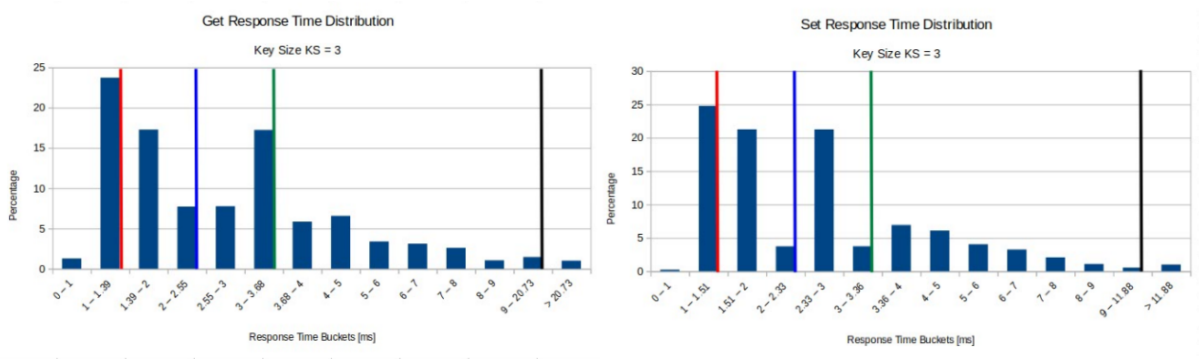
(a) Response Time Distribution for Get

(b) Response Time Distribution for Set

Figure 35: Response Time Distribution for a key size of 1

The behaviours depicted in *Figure 35* are very similar to the ones observed in the previous case, which is normal since with only one key gets are treated the same way for both sharded and non-sharded cases. We can see that in this case more gets fall in the first interval with respect to the sharded case, but on the other hand the 99 percentile is reached later. The set distribution is almost the same as in the sharded case, which is expected since the experiment is basically the same with only one key per get.

Figure 36 shows the distributions for a key size of 3. The right shift is clearly noticeable



(a) Response Time Distribution for Get

(b) Response Time Distribution for Set

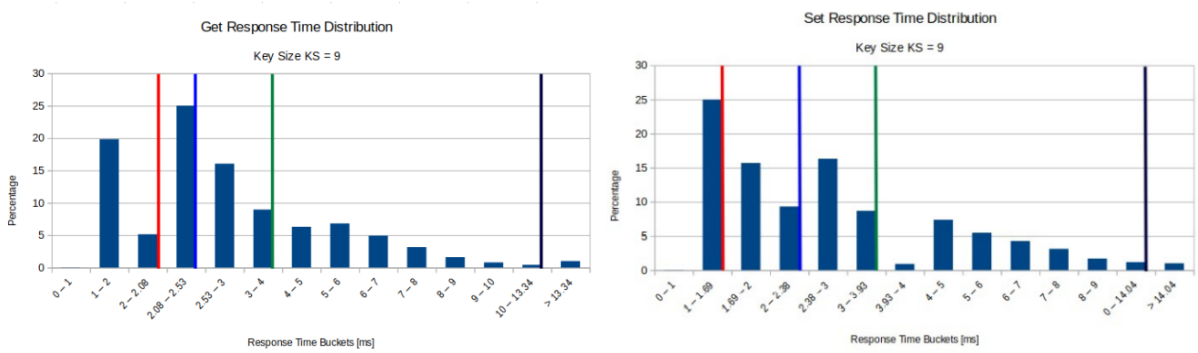
Figure 36: Response Time Distribution for a key size of 3

comparing *35a* and *36a*. Even without the splitting keys overhead, increasing the key size has an impact on the overall response time, since the system is now dealing with values 3 time bigger (3 KB instead of 1KB). However, we can see that this is still quite restricted to the first two bins, which means that we expect the overall response time not to be so much higher than in the case with a key size of 1.

Comparing the cases with 3 and 9 keys we observe that the right shift is even more clearly noticeable. We can see that in *Figure 37* most of the requests fall in the interval $[2.08; 2.53]$, more than 1 ms higher than in the previous figure. Also we can notice that the 25 percentile is reached later. The overall behaviour is due to the fact that the system is dealing with 9KB data instead of 3KB of the previous case.

5.2.1 Explanation

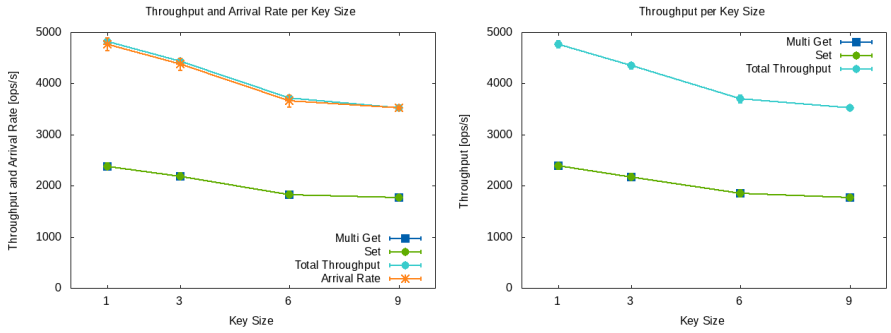
We start our analysis by showing the throughput's plots. Again, the graphs also show the arrival rate at the network thread.



(a) Response Time Distribution for Get

(b) Response Time Distribution for Set

Figure 37: Response Time Distribution for a key size of 9



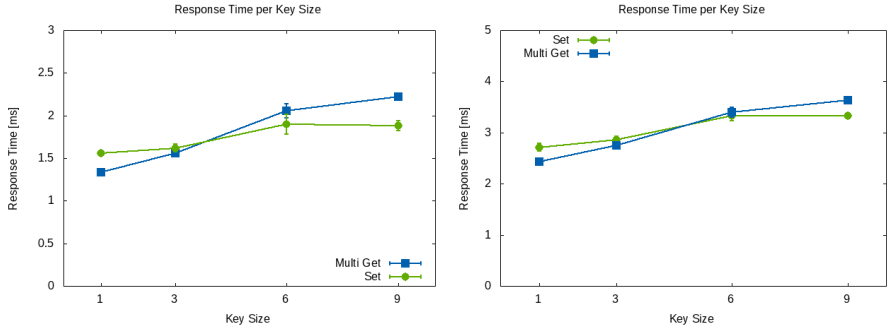
(a) Average Throughput measured on the middlewares

(b) Average Throughput measured on the clients

Figure 38: Average Throughput as a function of key size with sharding disabled

The general behaviour is the same as in the previous section. In fact, since the load is the same, we expect the biggest differences to show up when looking at the times spent to send a request and waiting for the servers, since as we observed in the previous experiment, these are the two statistics that are more affected by the changes in the key size and by the sharding being enabled or disabled, since they are the ones dealing with the data and with the (non-)splitting of gets.

Figure 39 shows the response time at the clients and at the middleware. The general behaviour



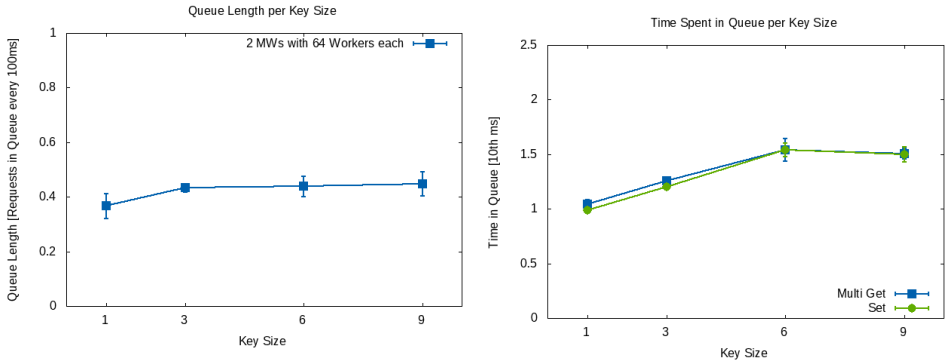
(a) Average Response Time measured on the middlewares

(b) Average Response Time measured on the clients

Figure 39: Average Response Time as a function of key size with sharding disabled

is the same as observed before, with the response time increasing with the key size. However, we can immediately notice two things: the first one is that the difference between sets and gets

is less than in the previous experiment, with set operations being still a bit slower even with a key size of 3. This is normal since sharded is disabled, which means that the workers will skip the splitting part and forward the whole get to one single server. With more keys the same process is done but the server will need more time to fetch all the values and the workers will need more time to send them all to the clients, which is why again set are processed faster when the size is 6 or 9. The second is that the response time at the clients is smaller for the key size of 1 and 3 but higher with 6 and 9 without sharding. This is what we would expect, since sharding with a small number of keys can still introduce a quite large overhead due to splitting and forwarding requests to multiple servers while the gain of splitting the load on the server is not so important with such a small key size. However, the behaviour changes when increasing the keys, since the overhead becomes negligeable and the gain becomes dominant. In the non-sharded cases, a single server will have to deal with an increasing number of values that it will have to fetch. This number grows linearly with the number of keys in get requests, while with sharding enabled the number of such values will be $\frac{k}{s}$ where k is the number of keys and s the number of servers.

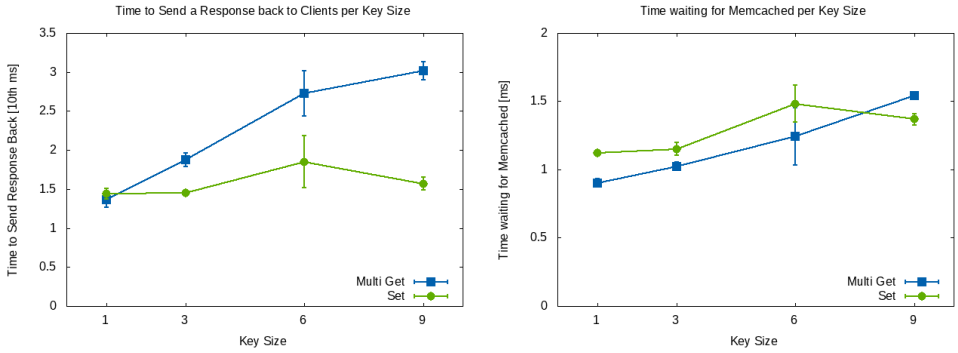


(a) Average Number of requests waiting in the queue

(b) Average waiting time in the queue

Figure 40: Average queue length and time in queue

Figure 40 shows the average queue length and time spent in the queue for the non-sharded experiment. Both behaviours are almost identical to the previous case, and the same reasoning made on the previous section applies.
The average time to parse a request is almost identical to the previous section. This is normal since the process that a worker has to do to parse a request has nothing to do with the sharding and since the number of requests parsed in a unit of time has remained the same. The time to send requests has almost halved for multi get requests. This happens because the entire `splitMultiGetRequestEvenly` method is skipped and the whole request is forwarded to one server. In this case we can see that set operations take longer to be sent, since the overhead to forward the request to all the servers is higher than the one needed to send, for example, 9 keys. *Figure 41* shows the average time to send a response back to the clients and the average time spent waiting for the memcached servers in the non-sharded situation. The firs one is very similar to the previous experiment, since the process is again unchanged. For key sizes of 1 and 3, the second one is smaller compared to the previous experiment, since the overhead of splitting is too high for such a small key size and it is not compensated by the gain of splitting requests over multiple servers. After key size of 3, we can observe that the slope of the graph increases, meaning that, due to the process of fetching more and more values, the server starts saturating.



(a) Average time to send response to client **(b)** Average time waiting for memcached servers

Figure 41: Average time to send response back to clients and waiting for servers

An important consideration to make is that in the sharded case workers need to wait for all three servers to reply, while in this case the time is spent waiting only one server. Making this consideration is important because it tells us that in the case of 9 keys, one server is taking the same amount of time to reply as three servers in the sharded case.

Finally, the results of the **Interactive Law** can be found under `asl-data/asl5/5.2/5.2_interactive_law`. As for the previous experiment, the experimental data almost perfectly matched the derived one.

5.3 Histogram

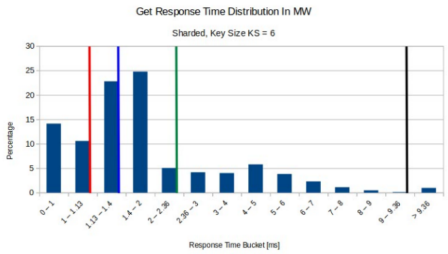
We additionally show 8 histograms representing the response time distribution of sets and gets measured at the clients and inside the middlewares, for the sharded and non-sharded cases. *Figure 42* shows the sharded case while *Figure 43* (on the next page) shows the non-sharded case.

We can observe that gets behave similarly in both the situations. As already seen in the previous section, with a key size of 6 non-sharded gets are slightly faster to process, whith 75% of non-sharded gets being completed in less than 2 ms, and 99% in less than 8.1 ms, while in the sharded case the same happens in less than 2.36 ms and 9.36 ms. Set operations are in both cases very similar, since they are not directly affected by the key size, and since gets are very close too. A similar behaviour can be observed at the clients, with 75% of gets being completed in 3.99 ms and 4.74 ms respectively with sharding disabled and enabled. However, 99% of sharded gets are below 13.95 ms while the same happens in less than 16.87 ms for non-sharded gets. In fact, this is related to what we already discussed in the previous subsection, where we could observe a higher response time at the clients for key size of 9 and sharding disabled.

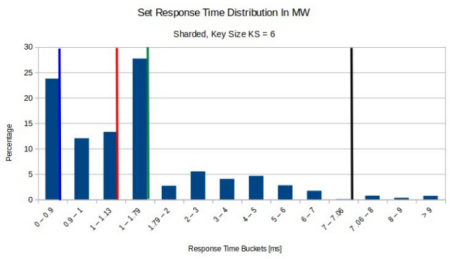
5.4 Summary

In this two experiments we examined the behaviour of the system when faced to sharded and non-sharded multi gets. As expected, in our analysis we observed that the highest times where the ones to wait for the servers and to send the responses back to the clients. The first is due to the time to fetch up to 9 KB of values and, in the case of sharded multi gets, waiting for the replies of all the servers. The second was mostly due to the 9 KB of data that had to be sent back to the clients. We also observed that the time to send a request to the server was between 1.5 - 2 times higher in the case with sharding, a behaviour that we expected since the workers need to split the keys evenly among the servers.

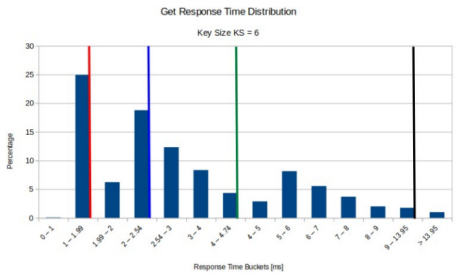
For both the setups we showed histograms representing the response time distribution at the



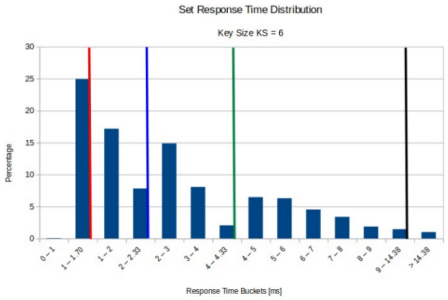
(a) Response Time Distribution for gets inside mws



(b) Response Time Distribution for sets inside inside mws

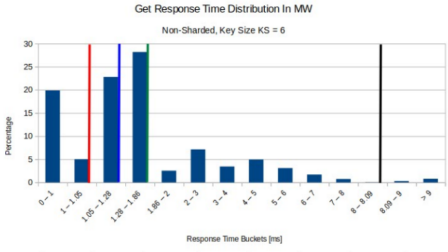


(c) Response Time Distribution for gets at the clients

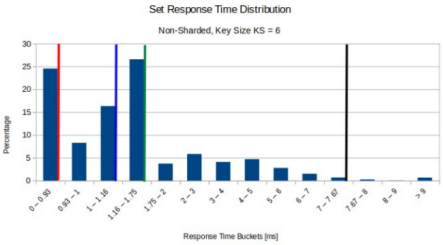


(d) Response Time Distribution for sets at the clients

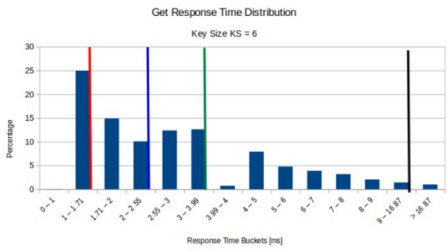
Figure 42: Response Time Distributions for a key size of 6, with Sharding



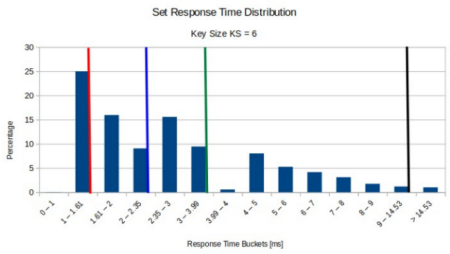
(a) Response Time Distribution for gets inside mws



(b) Response Time Distribution for sets inside inside mws



(c) Response Time Distribution for gets at the clients



(d) Response Time Distribution for sets at the clients

Figure 43: Response Time Distributions for a key size of 6, without Sharding

clients for every key size examined, and we could observe a shift to the right on the distribution, corresponding to the response time increasing with the size of the keys, a behaviour also observed on the average response time graphs.
We also noticed some stable behaviour when inspecting throughput, arrival rate, queue lengths,

time spent in queue and time to parse requests. These results were not surprising since this metrics are most influenced by the increasing workload in terms of number of clients, a behaviour that we had the chance to examine in details in *Section 3* and *Section 4*.

In the end, the highest difference in the two cases was represented by the time to send requests to servers. We recall that, as previously discussed, this difference is due to the time taken to split keys evenly and to send them to three server in the case of the sharded gets, while non-sharded request are simply forwarded as they are to one server, that will have to handle them in their entirety. However, despite this great difference, the final response time was similar in both cases. This was the case because the dominant time is the one waiting for the servers which, as discussed, is a tricky case. We could observe that non-sharded gets seemed to be quickier to process even for key sizes of 6 and 9. However, as mentioned, it is important to notice that in the sharded case this time represent the time waiting for all three servers to respond. This means that we can safely deduce that, at least in the case of a key size of 9, the processing time of a single server is less than in the non-sharded case. This is also expected since comparing the single servers, in one case the load will be of 3 key, corresponding to 3 KB of data to fetch, while in the non-sharded case this correspond to 9 KB of data.

To conclude the section, we give a final brief discussion about wheather to choose sharded or non-sharded multi gets for each of the cases examined, and we make an educated guess for key sizes greater than 9.

- *Key Size = 1*: In this case the request is treated as a normal get by the middlewares, thus making the choice of enabling or not sharding irrelevant.
- *Key Size = 3*: Based on the data that we analyzed in this section, we think that in this case sharding has still a quite large splitting overhead compared to the gain at the servers. Hence, we would probably choose to disable sharding in this situation.
- *Key Size = 6*: In this situation we could observe that the differences between the two cases started to lessen. This is due to the increasing gain at the server which starts becoming dominant over the splitting overhead, while on the other hand with non-sharding, the load on the server starts increasing and becoming dominant over the gain due to not splitting requests.
- *Key Size = 9*: In this case we observed the response time at the clients to be smaller in the sharded situation. We also saw that the plot corresponding to the response time measured on the middlewares flattens in this case, while in the non-sharded case it does not. Given this observations, we conclude that, in this case, sharded multi gets is the best option.
- *Key Size > 9*: Given the analysis that we have done, and the general behaviour that we observed in the 4 above cases, we can conclude that the gain on the servers over the overhead of splitting keys will become more important increasing the key size, in the sharded case. On the other hand the load over the single server will increase linearly with the key size, increasing the processing time in the non-sharded case. We thus think that sharding will be the best option. However, it is important to note that, after a certain number of key, we suppose that the splitting time will again become an important factor in this equation. To increase the benefits of sharding, we should probably revise the implementation of the `splitMultiGetEvenly` and make it more efficient, since this is a critical part with sharding enabled.

6 2K Analysis (90 pts)

In this section we do a 2k analysis of our system under the following setup. We use 3 client machines (with 64 total virtual clients per client VM) and vary the following parameters:

- Memcached servers: 2 and 3
- Middlewares: 1 and 2
- Worker threads per MW: 8 and 32

We repeat the experiment for a write-/read-only and a 50-50 ratio.

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 or more (at least 1 minute each)

Since we have three varying parameters, we start by defining three variables x_a, x_b and x_c as follows:

$$x_a = \begin{cases} -1 & \text{if 2 Servers} \\ 1 & \text{if 3 Servers} \end{cases} \quad x_b = \begin{cases} -1 & \text{if 1 MW} \\ 1 & \text{if 2 MWs} \end{cases} \quad x_c = \begin{cases} -1 & \text{if 8 WTs} \\ 1 & \text{if 32 WTs} \end{cases} \tag{1}$$

To calculate the impact of each parameter and the allocation of variations, we used the sign table method and the equations as explained in [3]. The complete computations can be found under `asl6/asl6_2k_tables.ods`. Here we resume the findings and we explain them.
First of all, it is important to specify that we performed the 2^k analyisis on measurements performed in the middlewares. We organise this section as a set of three paragraphs, corresponding to the write-only, read-only and 50-50 ratio experiments.

write-only *Table 1* summarizes the effects and percentage of variation explained by each factor. The first parameter gives us the estimate mean for response time and throughput

Parameter	Effect	Variation Explained
q_0	189.68	-
q_a	0.13	0.01
q_b	-5.90	16.53
q_c	-10.95	56.97
q_{ab}	0.11	0.01
q_{ac}	-1.09	0.57
q_{bc}	7.06	23.69
q_{abc}	0.46	0.10
error	-	2.12

Parameter	Effect	Variation Explained
q_0	1016.92	-
q_a	-0.60	0.01
q_b	31.92	18.29
q_c	57.60	59.57
q_{ab}	-2.14	0.08
q_{ac}	6.38	0.73
q_{bc}	-32.19	18.61
q_{abc}	-1.46	0.04
error	-	2.68

Table 1: Summary of 2^k analysis for a write-only experiment with repetitions

respectively. We can see that some of the parameters have a negative effect, while other a positive one. In this case we need to be careful and distinguish the cases. Recall that we used the sign method to solve the following equation

$$y = q_0 + q_a * x_a + q_b * x_b + q_c * x_c + q_{ab} * x_{ab} + q_{ac} * x_{ac} + q_{bc} * x_{bc} + q_{abc} * x_{abc}$$

We can see that, for example, q_b is negative for response time. This means that, if we have only one middleware, x_b will be negative and the multiplication will be positive. Thus, we will have $q_0 + q_b$. Since q_0 is the estimate mean response time, this means that having only one middleware will increase the mean estimate response time by q_b. Conversely, if we have two middleware, the same amount will be subtracted. With this consideration in mind, let's analyse the results of the experiment.
The first parameter that we defined is related to the variation of the servers. We can see that q_a has a positive impact on response time and a negative impact on throughput. Thinking about it, this makes sense. Since we are examining the write-only experiment, adding a server means that the worker threads will need to forward each request to one server more. This of course will result in a higher response time and thus in a lower throughput. However, the variation explained tells us that this impact is very low, since on both metrics it accounts for only 0.01%.
The second parameter is related to the variation of the number of middlewares employed by the system. As we can see from *Table 1*, this has a negative effect on response time and a positive one on throughput. This means that adding a middleware reduces the response time and increases the throughput, which is normal since we are adding more resources to the bottleneck of our system, as we discussed in *Section 3*. We can see that in this case, the impact of this parameter is relevant since it contributes to 16.53% and 18.28% of the variation of response time and throughput respectively.
The third parameter allows us to inspect the effect of variation of the number of worker threads. Looking at the respective entries on the table, we see that this is the parameter that affects both the metrics the most. This is not surprising especially because we increase the number of threads by four times. As we observed in previous sections, especially *Section 3* and *Section 4*, the configuration with 8 worker threads is the least performant, since it saturates faster and the growth of the response time is more pronounced, as well as the queue length which starts growing unbounded with higher number of clients. On the other hand, the configuration with 32 workers behaves much better performance-wise and we saw that it is often strictly close to the one with 64, which is the best we tested. The table entries tell us that increasing the number of workers increases performances by reducing the response time by an expected value of 10.95 ms, and increasing throughput by an expected value of 57.60 $\frac{ops}{s}$. Looking at the variation explained, we see that this parameter is by far the one that affect both response time and throughput the most, since it contributes to more than 50% in both cases. This is also closely related to the fact that this parameter is the one that changes the most, since we use 8 and 32 workers.
The q_{ab} parameter is related to the variation of both the servers and the middlewares. This has a positive effect on response time and a negative one on throughput. This means that performances will slightly decrease when both are in the baseline case (2 servers and 1 middleware) and when both are varied (3 servers and 2 middlewares). However, we can see from the variation explained that on both metrics this combination of parameters has a very low impact, accounting for less than 0.1%.
The variation of both servers and worker threads has the opposite behaviour. This happens because the gain of increasing the number of workers is dominant over the overhead caused by the additional server. In fact, we can see that the 4 smaller response time are all achieved when there are 32 workers, and similarly for the throughput. Again, we see that this combination has a little impact on both metrics.
The q_{bc} parameter represents the mutual variation of the number of middlewares and worker threads. We can observe that this is on both cases the second most affecting parameter. This has a positive effect on response time and negative on throughput, meaning that the performances will decrease when both are at the base level (1 middleware and 8 workers) and when

both are at their maximum (2 and 32 respectively). Looking at the singular data, this happens because the first two configurations (2 servers, 1 middleware and 8 workers, the same with 3 servers) are by far the worst performing and in both cases the coefficient is 1 since both parameters are at the baseline. In fact, we can see that the configurations with 2 middlewares and 32 workers are close to the best performing configuration. However, best performances are achieved when using 32 workers and only 1 middleware.
We can finally see that the combination of the three parameters have a positive effect on response time and negative on throughput, meaning that in general better performances are achieved when having 2 parameters increased and one at the baseline. Of course this is a general statement since increasing the number of serves and middlewares is not better performance-wise than having all the parameter at their maximum. As for most of the other combinations, this is not very affective, since it accounts for less than 1%. Finally, the remaining variation that can't be explained is attributed to the errors. This is less than 3% in both cases.

Parameter	Effect	Variation Explained
q_0	96.77	-
q_a	-19.84	72.00
q_b	-4.34	3.44
q_c	-9.46	16.38
q_{ab}	1.71	0.54
q_{ac}	-1.27	0.30
q_{bc}	5.72	5.98
q_{abc}	0.34	0.02
error	-	1.35

Parameter	Effect	Variation Explained
q_0	2018.32	-
q_a	376.85	74.15
q_b	114.75	6.87
q_c	164.63	14.15
q_{ab}	30.42	0.48
q_{ac}	59.10	1.82
q_{bc}	-65.48	2.24
q_{abc}	-7.48	0.03
error	-	0.25

Table 2: Summary of 2^k analysis for a read-only experiment with repetitions

read-only *Table 2* shows the effects of parameters for a read-only experiment. The first row again tells us that the mean estimate is 96.77 ms for the response time and 2018.32 $\frac{ops}{s}$ for throughput. Looking at the q_a we can immediately observe the first difference between the two experiments. In this case, increasing the number of servers also increase performances. This is expected in the case of a read-only, since the more the servers, the more the load gets split, since in this case every request is forwarded to one server only. We can also see that this parameter has a big impact on the variation and is by far the most affecting one, since in both cases it accounts for more than 70%.
Not surprisingly, increasing the number of middleware or the number of workers again has a positive effect on performances, as we already discussed on previous section. We can again observe that the variation of workers is still important (second most affecting factor with 16.38% and 14.15% respectively) while the variation of middlewares affects for less, with around 3% and 7%.
The combined effect of number of servers and number of middlewares slightly increases the response time but also increases the throughput. For response time, this is mostly due to the fact that the baseline configuration is by far the worst performing. This contribution is however limited, since the variation explained is less than 1%.
The effect of q_{ac} impacts positively the performances, which is what we would expect since both have a positive and impacting effect on both metrics, alone contributing to almost 90% of the variation. In particular, we can observe from the single data that the two best performances configuration are the ones with 3 servers and 32 workers. This mutual variation also contribute less than 1%, as it was the case of the previous one.
On the contrary, the q_{bc} parameter has a relatively high impact in both cases. As in the write-only experiment, this tells us that, in general, the combined effect of having both at the baseline

or both at their maximum decreases performances. This again is due to the baseline configuration and to the fact that, as the high impact of q_a tells us, increasing both the middlewares and the workers has little sense if the servers are not increased. This can also be seen by the fact that with 2 middlewares and 32 threads but 2 servers, the response time is about 108 ms. Instead, with for example 2 middlewares and 8 workers but 3 servers, this drop to 79 ms.
The interaction between all the parameters is again almost negligible since it only account for 0.02% and 0.03%. Finally, the error accounts for less than 1.5% in the variation.

Parameter	Effect	Variation Explained
q_0	155.43	-
q_a	-9.19	39.36
q_b	0.05	0.001
q_c	-6.71	20.95
q_{ab}	-1.50	1.04
q_{ac}	2.15	2.15
q_{bc}	6.78	21.44
q_{abc}	1.61	1.21
error	-	13.85

Parameter	Effect	Variation Explained
q_0	659.25	-
q_a	41.04	34.52
q_b	17.35	6.17
q_c	47.50	46.23
q_{ab}	6.83	0.96
q_{ac}	1.81	0.07
q_{bc}	-21.32	9.32
q_{abc}	-4.25	0.37
error	-	2.37

Table 3: Summary of 2^k analysis for a 50:50 read-write experiment with repetitions, set tables

Parameter	Effect	Variation Explained
q_0	138.82	-
q_a	-11.33	26.66
q_b	-6.19	7.96
q_c	-16.60	57.21
q_{ab}	-2.14	0.95
q_{ac}	0.70	0.10
q_{bc}	3.03	1.90
q_{abc}	0.91	0.17
error	-	5.04

Parameter	Effect	Variation Explained
q_0	659.35	-
q_a	41.06	34.56
q_b	17.30	6.14
q_c	47.52	46.27
q_{ab}	6.75	0.93
q_{ac}	1.80	0.07
q_{bc}	-21.35	9.32
q_{abc}	-4.20	0.36
error	-	2.36

Table 4: Summary of 2^k analysis for a 50:50 read-write experiment with repetitions, get tables

50:50 ratio *Table 3* and *Table 4* show the results for the 50:50 experiment for set and get respectively. A general look at the tables tells us that there are some important differences with respect to the previous cases, especially on the set side. The major difference is given by the effect of q_a. We can observe that in this experiment increasing the number of servers increases the performances, and we can also see that, at least for response time of sets, this is the most affecting factor. This is due to the interaction between sets and gets operations. We saw previously that for a write-only experiment the same parameter had a negative impact on performance, however the variation explained was really low (0.01%). On the opposite, the same parameter affected positively both metrics in case of a read-only and the variation percentage was almost 75%. Since this experiment combine the two operations, the result is that increasing the number of servers increase the performances, even if, of course, by a much less factor in comparison with read-only. This also means that the overhead of forwarding sets to more servers is by far compensated and surpassed by the gain that this parameter has on the read side. Thus, even if single sets are performed more slowly with more server, gets are performed much faster. This result in more set operations being performed and thus the average is done over more operations, giving a smaller result.
The effect of adding a middleware is behaving similarly in both gets and sets, as expected from what we have observed in the two previous situations. In the case of response time for sets, the effect is really low. This is because, looking at the individual values, we observe the following behaviour: when 2 servers are used, increasing the number of middlewares decreases

the response time. When 3 servers are used, the opposite happens. This behaviour was already observed in the write-only case. However, since in this experiment the differences are lower, they compensate each other and the effect is almost 0. This happens because when adding both a server and a middleware, we both add more overhead of forwarding and more load on the servers, since we have more workers doing the jobs.
Once again, the parameter corresponding to the workers has a great impact on the performances. This is expected since we observed the same behaviour on the two previous experiment and since this is a combination of the two. In this case we observe the same behaviour that we already discussed for the q_a parameter. In the previous cases we saw that workers had a greater impact on the write-only experiment than on the read-only one. In this case, we see that combining the two operations gets are more affected by the number of threads. This is normal given the interaction between sets and gets. Since sets are processed faster with more threads, more gets will be performed, incrementing the throughput and decreasing the response time.
The q_{ab} parameter tells us that incrementing the number of servers and the number of middlewares gives us better performances, since both alone are improving factors and since the number of servers is the dominant factor on the experiment. However we can notice that this has a low impact in all the cases.
The parameter corresponding to the mutual effects of servers and threads indicates that in general varying them both slightly increases throughput and response time. However, it is important to note that the highest throughput and lowest response time for all the four cases are achieved with 3 servers and 32 threads. Again, this interaction is not very affecting, since even in the case of the response time for set it only account for 2.15%.
As previously, the variation of both middlewares and worker threads in most cases has an important effect on variation and decreases performances, a behaviour which is consistent with what we have observed in the previous two experiments. The same happens with the interaction with all the factors, which again behaves the same as previously.
Finally, we can see that in most of the cases the variation explained by the error is relatively low. However, we can see that for response time of sets, this parameter is quite high. This is due to an important increase in response time during the second repetition of the experiment with 2 servers, 2 middlewares and 8 workers. In this case the response time was 30 ms higher than the other two repetitions, which caused the error to be higher than in the other situations. A similar behaviour is observed in the get case, but since other configurations achieved higher consistency with each other, the final error was much less than in the set case.

As done in the previous sections, we conclude with a brief commentary on the **Interactive Law** (`asl6/asl6_interactive_law.ods`). In this case, we can see that the derived throughput and response time are almost every time very close to the experimental result, meaning that in this experiment, as in the previous one, the worker threads were almost always processing a request.

7 Queuing Model (90 pts)

In this section we will build a M/M/1 and a M/M/m model based on *Section 4* and a Network of Queues based on *Section 3*.

7.1 M/M/1

We model our entire system as being a single entity with one queue. We take as mean arrival rate λ the average arrival rate measured at the network thread for every configuration of worker threads over each number of clients. This choice seemed quite straightforward since it is exactly what the model is taking as input. As μ, we take the maximum throughput observed in the middleware, for every configuration of worker threads. In this case we could also have taken the average throughput for every configuration. However, in this case both λ and μ would have been almost the same and the utilization would have become 1, making the response time and the number of jobs in the system become infinite. This would not have been a good approach since we already observed that in practice this behavior did not appear and thus the model would have failed to approximate the real system.
To compute the outputs, we used the formulas listed in [3], We thus provide a table summarizing the notation that we used (*Table 5*). We also provide a summarizing table with the findings and we discuss them. The complete computations can be found under `asl7/mm1/mm1_res.ods`.

Symbol	Description
λ	Mean Arrival Rate $[\frac{ops}{s}]$
μ	Mean Service Rate $[\frac{ops}{s}]$
ρ	Traffic Intensity
$E[n]$	Mean Number of Jobs in the System
$E[r]$	Mean Response Time [ms]
$E[n_q]$	Mean Number of Jobs in the Queue
$E[w]$	Mean Waiting Time in the Queue [ms]

Table 5: Summary the notation used in this Section

Worker Threads	λ	μ	ρ	$E[n]$	$E[r]$	$E[n_q]$	$E[w]$
8	5462.06	6141.19	0.89	8.04	1.47	7.15	1.31
16	6583.07	7830.92	0.84	5.27	0.80	4.43	0.67
32	7097.28	8503.16	0.83	5.05	0.71	4.21	0.59
64	7373.79	9174.60	0.80	4.09	0.55	3.29	0.45

Table 6: Summary of our findings for M/M/1 model

Worker Threads	$E[r]$	$E[n_q]$	$E[w]$
8	12.08	56.09	9.81
16	6.04	26.14	3.71
32	4.89	13.65	1.81
64	4.70	9.83	1.32

Table 7: Summary of experimental data from *Section 4*

Discussion *Table 6* shows the findings given by the model while *Table 7* shows the experimental data taken from *Section 4*. We can see that this model is not accurate in estimating the results of the system. Looking at the results, we can see that the model captures the general behaviour of the system, since the configuration with 8 worker threads is by far the less performant, the one with 16 is an in-between, while the two last are the better performing and behaves similarly despite the difference in the number of threads. However, the model is quite optimistic in predicting all the statistics. We see that the response time is almost 10 times smaller than in reality for all the configurations, while the waiting time in queue is again 10 times smaller for the 8 workers configuration down to 3 times smaller for the 64 workers.

7.2 M/M/m

We again use as λ the mean arrival rate at the network threads. To choose μ, we follow the same principle as before but this time we extend the reasoning to each worker thread. For each configuration of worker threads, we take μ as the maximum throughput observed inside the workers. We then multiply this by the total number of threads. We again use the formulas in [3] to compute the results.
As before, we provide a table summarizing the results of the model and we compare them with *Table 7*.

Worker Threads	λ	μ	ρ	$E[n]$	$E[r]$	$E[n_q]$	$E[w]$
8	5462.05	761.90	0.45	7.73	1.41	0.17	0.10
16	6583.07	499.80	0.41	13.43	2.04	0.09	0.04
32	7097.28	295.40	0.37	24.08	3.39	0.02	0.01
64	7373.99	173.90	0.33	42.40	5.75	0.0003	8.72E-05

Table 8: Summary of our findings for M/M/m model

Discussion We can see that in this case our model completely fails to capture the behaviour of the system. The problem in this case is that the resulting throughput for each configuration is way larger than the one measured in reality, since we assumed that each worker was able to process requests at the maximum rate. This results in the Traffic Intensity ρ to be way lower than the one we observed during the experiment. This means that our model behaves as if it were underutilized, and the results are accordingly. In fact, if we compare, for example, the response time of the configuration with 8 worker threads with 6 clients to the one in the table, we see that this are similar (1.18 ms and 1.41 ms), since with 6 clients the system is undersaturated. This however is not the case in general and thus in reality ρ is higher than the one in the model.

7.3 Network of Queues

We build two types of network of queues. The first one is simply composed of a M/M/1 model representing the middleware(s) and another M/M/1 for the server. The second one is composed of WT M/M/1, one for each worker threads, and another M/M/1 representing the server. We consider the network as a delay center.

7.3.1 M/M/1 System & M/M/1 Server

Worker Threads	V_i	X	X_i	S_i	U_i
8	1	3717.08	3717.08	1.70	6.31
16	1	3700.75	3700.75	2.33	8.63
32	1	3646.02	3646.02	2.65	9.68
64	1	3755.71	3755.71	2.64	9.95

Worker Threads	V_i	X	X_i	S_i	U_i
8	1	3986.69	3986.69	1.50	5.97
16	1	3965.71	3965.71	1.86	7.38
32	1	3991.46	3991.46	2.01	8.04
64	1	3993.55	3993.55	2.02	8.09

Table 9: M/M/1 model for the system based on *Section 3.1*, for get (left) and set (right)

Worker Threads	V_i	X	X_i	S_i	U_i
8	1	3717.08	3717.08	0.84	3.11
16	1	3700.75	3700.75	0.95	3.52
32	1	3646.02	3646.02	1.01	3.70
64	1	3755.71	3755.71	0.99	3.73

Worker Threads	V_i	X	X_i	S_i	U_i
8	1	3986.69	3986.69	0.79	3.15
16	1	3965.71	3965.71	0.88	3.51
32	1	3991.46	3991.46	2.01	3.80
64	1	3993.55	3993.55	2.02	3.87

Table 10: M/M/1 model for the server based on *Section 3.1*, for get (left) and set (right)

Section 3.1 - Discussion Since there is only one server and the system is modeled as a single entity, the visit ratio is 1 for both devices, X is the average throughput between 1 and 32 clients, while the service time is the same average computed on the difference between the response time and the time spent in queue. *Table 9* shows the results for the system while *Table 10* for the server. We can see that, as we expected and as we observed in the corresponding section, our middleware is the bottleneck of the system, since it has a higher utilization U_i. We can also observe that in both models the utilization grows with the number of workers. On the server-side, this is normal since the more the workers, the more the load put on the server, since more requests are forwarded at the same time. For the middlewares, this is due to the computation of U_i. Recall that $U_i = X_i * S_i$. In general, more workers implies a higher throughput. Also, the service time of configurations with more worker threads is always higher than the ones with less threads. This is a behaviour that we already observed when examining the statistics in *Section 3.1*. Recall that, with more threads, the time to parse and send a request and to send back a response are higher due to the higher competition for the locks. Thus, the service time is higher and, since X_i is also higher, or at least the same, U_i will be higher as well.

Worker Threads	V_i	X	X_i	S_i	U_i
8	1	6590.22	6590.22	1.81	11.92
16	1	6527.73	6527.73	2.15	14.05
32	1	6924.04	6924.04	2.09	14.48
64	1	7007.04	7007.04	2.11	14.78

Worker Threads	V_i	X	X_i	S_i	U_i
8	1	6752.28	6752.28	1.57	10.63
16	1	7018.33	7018.33	1.80	12.67
32	1	7221.12	7221.12	1.77	12.81
64	1	7418.25	7418.25	1.75	12.99

Table 11: M/M/1 model for the system based on *Section 3.2*, for get (left) and set (right)

Worker Threads	V_i	X	X_i	S_i	U_i
8	1	6590.22	6590.22	1.10	7.24
16	1	6527.73	6527.73	1.13	7.39
32	1	6924.04	6924.04	1.08	7.45
64	1	7007.04	7007.04	1.11	7.78

Worker Threads	V_i	X	X_i	S_i	U_i
8	1	6752.28	6752.28	1.05	7.13
16	1	7018.33	7018.33	1.10	7.73
32	1	7221.12	7221.12	1.07	7.70
64	1	7418.25	7418.25	1.05	7.78

Table 12: M/M/1 model for the server based on *Section 3.2*, for get (left) and set (right)

Section 3.2 - Discussion The general behaviour shown by *Table 11* and *Table 12* is the same as in the previous section. Again, the middlewares have the highest utilization and are the bottleneck of the system, as we already observed in *Section 3.2*. We can again observe that the utilization is higher with more workers on both server and middlewares. The same reasoning made previously can be applied in this case too. We also notice that the utilizations are much higher than in the previous case, which is normal since, on the server side, both S_i and X_i have increased and on middleware side, X_i has doubled and S_i has decreased by a small factor. The increase of S_i in the server is due to the more load that it has to deal with, since the throughput doubled, a behaviour that we observed and explained in the corresponding section while examining the time spent waiting for memcached servers.

To conclude, we saw that this model, even if simplistic, is able to capture the general behaviours of middlewares and servers that we observed on both *Section 3.1* and *Section 3.2*.

7.3.2 M/M/1 Workers & M/M/1 Server

For the M/M/1 model corresponding to the workers, we used the measured service time as S_i, and we used $\frac{1}{WT}$ as V_i. This is an approximation that captures very closely the measured visit ratio. In the following tables we only show one row for each worker thread configuration. The

results are similar for each worker as can be seen in the complete results that can be found under `asl-data/asl7/noq/3.1_noq_workers.ods` and `asl-data/asl7/noq/3.2_noq_workers.ods`. The table for the servers are the same as in the previous model.

WTs	V_i	X	X_i	S_i	U_i	U_{tot}
8	0.125	3308.86	413.61	1.79	0.74	5.82
16	0.0625	3273.64	204.60	2.42	0.50	7.90
32	0.03125	3308.86	101.19	2.70	0.27	8.80
64	0.015625	3278.17	51.22	2.78	0.14	9.11

WTs	V_i	X	X_i	S_i	U_i	U_{tot}
8	0.125	3503.73	437.96	1.62	0.71	5.54
16	0.0625	3433.41	214.59	1.90	0.41	6.44
32	0.03125	3463.05	108.22	2.10	0.23	7.22
64	0.015625	3521.06	55.02	2.11	0.12	7.37

Table 13: M/M/1 model for workers based on *Section 3.1*, for get (left) and set (right)

Section 3.1 - Discussion As the previous model, this one captures the same general behaviour of the middleware. Again, the middleware is the bottleneck of the system, and as previously observed, the utilization grows with the number of workers.

WTs	V_i	X	X_i	S_i	U_i	U_{tot}
8	0.125	5293.11	661.64	1.72	1.13	9.34
16	0.0625	5293.21	330.82	2.24	0.74	11.45
32	0.03125	5550.11	173.44	2.40	0.42	12.86
64	0.015625	5583.34	87.24	2.26	0.20	12.92

WTs	V_i	X	X_i	S_i	U_i	U_{tot}
8	0.125	5404.51	675.56	1.65	1.11	9.25
16	0.0625	5511.22	344.45	1.78	0.61	9.86
32	0.03125	5658.61	176.83	1.80	0.32	10.55
64	0.015625	5724.90	89.45	1.78	0.16	10.37

Table 14: M/M/1 model for workers based on *Section 3.2*, for get (left) and set (right)

Section 3.2 - Discussion Again we can see that the utilization of the middlewares grows since the throughput is much higher than in the previous section. Comparing with *Table 12*, we can again conclude that the middlewares are the bottleneck of the system, since they are the most utilized devices. We observe again the increase in utilization with higher number of worker threads.

To conclude, we saw that both models capture the general behaviour of the system, confirming what we previously discussed in *Section 3*: the middleware is the bottleneck of the entire system, and increasing the number of workers increases the service time since the atomic operations become slower due to the more contentions.
We observed that between the models, the first one outputs a slightly higher utilization on the middleware-side. This is due to the fact that the second one is more fine grained, since it computes the utilization of the single worker threads and then sums the contributions. However, in general the two models are quite similar and both succeed in capturing the system's behaviour in a realistic way.

References

[1] Prof. Alonso G.,
Advanced Systems Laboratory - Slides,
ETH Zürich,
2017

[2] Prof. Alonso G.,
Advanced Systems Laboratory - Project Description,
ETH Zürich,
2017

[3] Jain R.,
The Art of Computer Systems Performance Analysis,
Wiley Professional Computing,
1991