Advanced Systems Lab Report

Autumn Semester 2018

Name: Luca_Di_Bartolomeo
Legi: 18-961-053

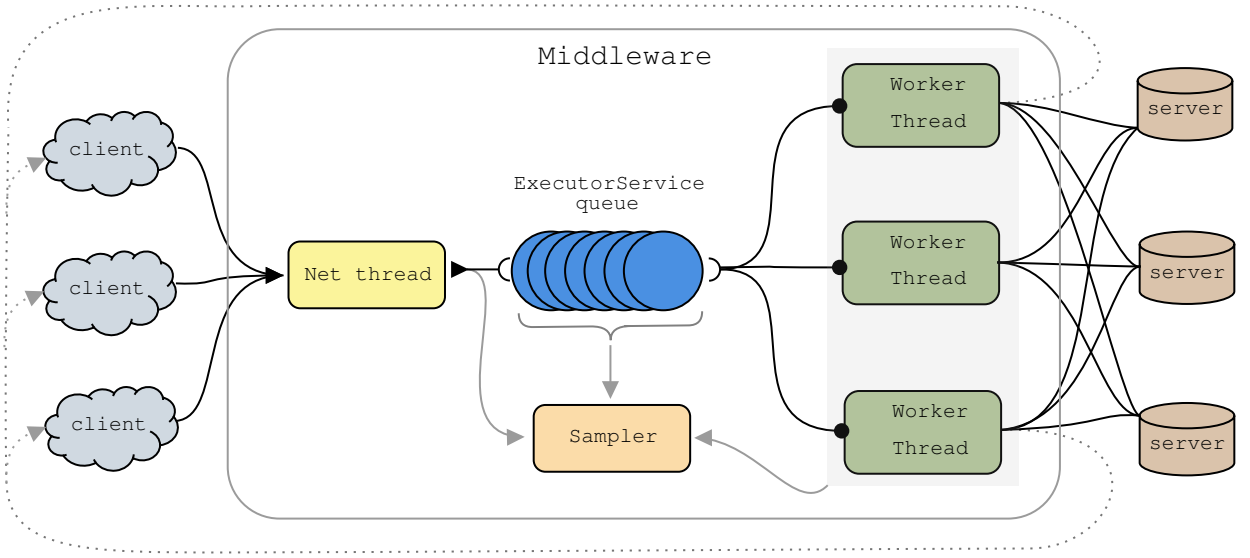
Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

1 System Overview (75 pts)

This section will explain the general functionality of the middleware. It will also explain how requests are parsed and forwarded to the servers.

1.1 System architecture



The middleware is composed of the following components:

- The **Network Thread** which receives requests from the clients and pushes them into the job queue
- The **Job Queue** which holds requests waiting to be accomplished
- The **Worker Threads** which satisfy requests by forwarding them to the servers and giving the answer back to the clients
- The **Logger Thread** which takes care of calculating and storing statistics of the system

Let's look into each component in detail:

1.1.1 Network thread

The network thread is the single thread that takes care of accepting requests from the clients.

It accomplishes that by using *non-blocking I/O*, by using a `java.nio.ServerSocketChannel` object to store all active connections. An infinite loop uses `java.nio.Selector` to cycle through all open connections to read data (if there is data available to read), and also checks if there are any new connection waiting to be accepted.

If there is a request to read from a particular socket, the network thread will read it and checks if the request was correctly received in its entirety. In fact, packet fragmentation is not so rare, and

the network thread must take care of it. A `HashMap` is used to store partially received requests, while waiting to receive them completely. This is done in the following steps:

- The network thread will check if the selector key from which it is reading is present in the `HashMap` of incomplete requests. If it is, any read output will be appended to the incomplete request stored in the hashmap. When an incomplete request string becomes complete, it is removed from the `HashMap`, and the thread will skip to the final step.

- Otherwise, the network thread will read the first word of the request, to understand if it's a `GET` or a `SET`.

- In the case of a `GET`, the request is simply accepted as it is. In fact, a `GET` request is composed of the following fields:
  `get <key1> <key2> ... <key9>\r\n`
  The request is considered complete if there is `\r\n` at the end.

- In the case of a `SET` request, the first line of the request is parsed and splitted into words. In fact, the `SET` request is composed of the following fields:
  `set <key> <flags> <exptime> <bytes> [noreply]\r\n<data>\r\n`
  The network thread will read the fifth word (`<bytes>`), which represents the length in bytes of the `data` section, and checks if all the data section was received, by comparing the `bytes` field to the return value of the `SocketChannel.read()` call.

  If the check fails, and the `data` section is too short, the network thread will store the string received in a `HashMap`, using as key for the `HashMap` the particular key that the `Selector` assigned to the client that sent this incomplete request string.

- The network thread will create a new `RequestData` object, which is a custom defined class containing fields like the request string, the request type (`SET` or `GET`), the size of the request string, and some statistics, like the timestamp of when the request was (completely) received from the client.

  After that, the network thread will push this `RequestData` object to the *job queue*.

1.1.2 Job queue

The queue was realized by using the already existing queue implementation of the `java.util.concurrent.ExecutorService`, which also takes care of feeding jobs from the queue to the worker threads, as soon as there is one idle thread available.

The data structure that holds a particular job is an object of the class `RequestManager`, which is an extension of the java interface `Runnable`. This object is constructed from a `RequestData` object, and contains the code that the worker thread will execute in its `run()` method.

The `ExecutorService`'s queue is unbounded.

1.1.3 Worker threads

Worker threads are objects of the class `InitializingThread`, which are generated at the startup of the middleware by the `ExecutorService` with the help of the `InitializingThreadFactory` class.

The specific `ExecutorService` used is called `FixedThreadPool`, which reuses a specific number of threads specified when it is created. The purpose of this indirect creation process is that in this way the `InitializingThreadFactory` is able to set up all connections that each worker thread needs to have with each `memcached` server as soon as the middleware starts, without waiting for requests to come. The connections to the servers are stored with an array of `Sockets`. They are only closed during the shutdown of the middleware.

The worker thread accepts and executes runnables of the type `RequestManager`. A `RequestManager` object is constructed using a `RequestData` object, which contains information about the request that needs to be processed, and a `SocketChannel` instance, which represent the channel that the worker thread can use to communicate the answer of the request back to the client.

In this way, worker threads do not need to deal with the management (creation and destruction) of networking data structures during their processing of a request.

Let's now move on to the actual processing of a request:

The worker thread reads from the `RequestData` object what kind of request it has to deal with. In the case of a `SET`, it will call the method `query_memcached_set()`. In case of a `GET`, it counts the words that the request is made of. If the words are greater than 2, and the middleware was started with the `-s` flag set to `true`, it will call the method `query_memcached_multiget()`; otherwise it will call the method `query_memcached_get()`.

This is how get, set and multiget request are satisfied:

- (**set**) `query_memcached_set()`: the worker thread forwards the request to all `memcached` servers, and waits for their answer. If all answers are equal to `STORED\r\n`, the worker thread will answer `STORED\r\n` back to the client. If any one of them returns an error, the worker thread will send that error to the client.

  To make this process efficient, it will first send the `SET` request to all servers, and then will wait for all the responses.

- (**get**) `query_memcached_get()`: The worker thread will forward the `GET` request to one of the `memcached` servers, and then send back to the client the response it gets from the server.

  To avoid always asking the same server for `GET` operations, the servers are queried in a round-robin fashion (more details on this in the next subsection).

- (**multiget**) `query_memcached_multiget()`: The worker thread will split the `GET` request into words, and then calculates how many keys it must ask for to each server to achieve an equal distribution of load, in this way: suppose there are k keys and n servers - then, for each of the first $n-1$ servers, `ceil(k/n)` keys will be queried; the remaining keys will be assigned one for each server.

  Then, for each `memcached` server, a `GET` request with the keys calculated is crafted and sent to the server. The worker thread will wait for the answer of all the servers, craft a `GET` response by concatenating all responses it got from the servers, and sends it back to the client.

  If any of the server return an error or a miss, the worker thread will send that back to the client instead.

During the processing of any kind of request, a worker thread will save three timestamps in the `RequestData` object: when it receives the request from the queue, when it sends the request to the `memcached` server, and finally when it receives the answer from the server and forwards it to the client.

1.2 Balance and parallelism

There are two main balance measures that need to be discussed: balance *between worker threads* and balance *between memcached servers*.

The balancing between worker threads is entirely managed by the `ExecutorService`, as it ensures that as soon as a given thread becomes idle, a new job will be sent to it, if there are jobs in the queue. The correct balancing between worker threads can be tested by looking at the middleware logs, where the throughput of each thread is reported.

The balancing between `memcached` servers, instead, is done only for `GET` requests (non sharded). In fact, `SET` requests need to be forwarded to every single server, and multigets too are forwarded to each server.

The balancing is done in a round-robin fashion, by using an `AtomicInteger` object called `rr_counter`. This number keeps track of the last server queried with a `GET` request, and being atomic, can be used simultaneously by all threads . Infact, inside method `query_memcached_get()`, the server to query is calculated in this way: `n = (rr_counter.getAndIncrement() % number_of_servers);`

1.3 Logging

There are some extra fields in the `RequestData` class that help logging. They are timestamps of:
`from_client` - When the request was received by the net thread
`enqueue` When - the request was pushed in the job queue
`dequeue` - When the request was dequeued by one of the worker threads
`to_server` - When the request was forwarded to a `memcached` server
`to_client` - When the answer of that request is received from `memcached` and sent back to the client.

The collection of this information and the calculations needed to provide certain statistics (like average response time) are done by another separate thread, the `Sampler`. It runs an infinite loop with an interval of 3 seconds, in which every request processed by every thread is parsed, and, for each thread, by analyzing the timestamps inside `RequestData`, the following averages are produced: *response time, throughput, queue waiting time, size of the queue, time spent parsing the request, time spent waiting for an answer from the server, time spent by the net thread before pushing the request in the job queue.*

Those values are all relative to a single interval of three seconds, and are calculated individually for each thread. They are then saved in a file called `middleware_log.log` in CSV format.

When the middleware receives the shutdown signal, the Sampler thread will also calculate percentiles of the response time useful for the experiment in Section 5 in the method `percentiles()`, and saves them to a file called `middleware_percentiles.log`.

2 Baseline without Middleware (75 pts)

In this set of experiments we are going to observe the baseline performance of `memtier_benchmark` and of `memcached` when they are directly communicating. Our objective is to understand which are the individual limitations of `memtier` and of `memcached`.

2.1 One Server

In this experiment we use three clients running `memtier` all connected to a single server running `memcached`. We will test with a varying number of clients (1,2,4,8,16,32). Each `memtier` instance will run with two threads. There will be a single `memcached` server answering requests from the client.

2.1.1 Results

Below are the plots of a write-only and read-only tests, measuring the aggregated throughput (the sum of the throughput of each client) and the average response time, depending on the number of virtual clients. It was also plotted the Interactive Law (assuming 0 as thinking time).

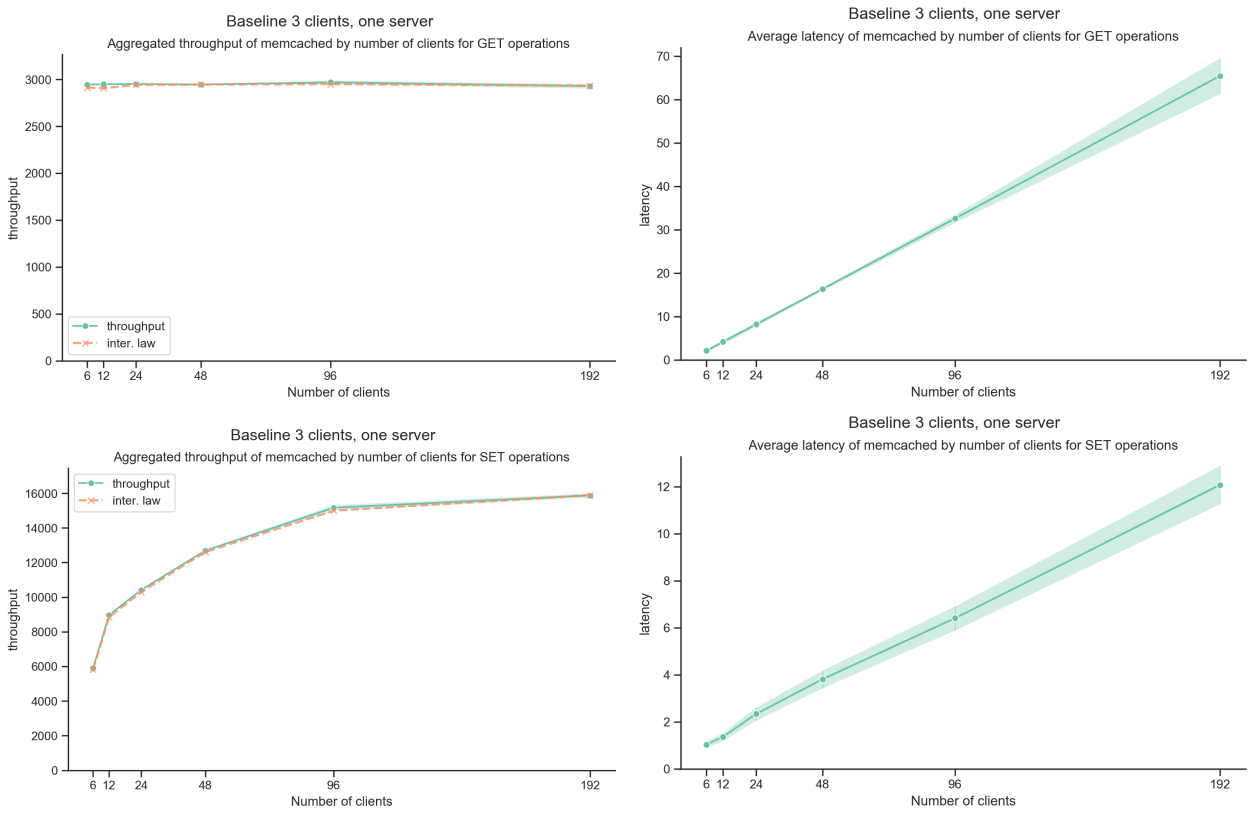


Figure 1: Results of the experiment. On the left column, we have aggregated throughput; on the right one there is average latency.

2.1.2 Explanation

From the plots we can see that the `GET` operations saturate the `memcached` server immediately, starting from 6 clients. Instead, for the `SET` operations, it seems that the `memcached` server is under-saturated for up to 48 clients, and saturates at 96 clients (throughput doesn't grow, but response time does).

We observe a high discrepancy between read-only and write-only throughput. Since in theory the `SET` operation should be more computationally expensive (at least for `memcached`), we can hypothesize that there is a bottleneck at the network level that somehow affects only the read-only operations. To test if we are right, we can look at the upload and download bandwidth of the machines in the following table:

	Download	Upload
Client	`1.3 Gbit/s`	`200 Mbit/s`
Server	`1 Gbit/s`	`100 Mbit/s`

The download speeds were obtained by running a speed test on a public service (`www.speedtest.net`). There is a very useful package on the ubuntu repositories called `speedtest-cli` [1] that lets us do the speed test directly from the command line interface.

The upload speeds, instead, were tested with the `iperf3` tool. Specifically, to measure the upload speed of the client machines (the machines running `memtier`), the command `"iperf3 -s"` was run on one of the server machines, and the command `"iperf3 -c <ip of the server>"` was run on one of the clients. The exact opposite was done to measure upload speed on the server machines.

From those results, we can draw some interesting conclusions. Let's measure the bandwidth consumed by the server machine when uploading results of the `GET` operation: (we assume a lower-bound of 10 bytes for the header, but it can get bigger)

$$(data\_size + header) \cdot throughput \text{ bytes/s} =$$
$$(4096 + 10) \cdot 3000 \text{ bytes/s} = 12318000 \text{ bytes/s} = 98544000 \text{ bits/s} = 98.5 \text{ Mbit/s}$$

This means that the `memcached` machine, with a throughput of 3000 `GET`/s, already reaches its upload bandwidth of 100 Mbits/s. So we can safely confirm our hypothesis, that in case of the read-only experiment the bottleneck is the network of `memcached`.

Another interesting observation we can make is that the response time of the `memcached` server increases proportionally to the number of clients.

2.2 Two Servers

We repeat the same experiments as before, with the same varying number of virtual clients, but this time there will be only one client machine with two instances of `memtier_benchmar` running, each with only one thread, and there are two `memcached` servers listening to requests.

[1] `https://github.com/sivel/speedtest-cli`

2.2.1 Hypothesis

Since we now have the data from the previous experiment, we can take some hypothesis about what will happen in this experiment.

For the `GET` operations, we know for a fact that a single `memcached` server can't go above a throughput of 3000 op/s. Since we're using two servers, the final throughput should be less than 6000 op/s. For the `SET` operations, instead, the client must bear the heavy upload bandwidth. Let's calculate how many op/s can the upload bandwidth of the client manage:

$$\frac{bandwidth}{set\_size}\ \text{op/s} = \frac{200\ \text{Mbit/s}}{(4096+10)*8\ \text{bytes}} = 6088\ \text{op/s}$$

So, similarly to the `GET` operations, we expect a maximum throughput of around 6000 op/s.

2.2.2 Results

Below are the plots representing aggregated throughput and average response time.

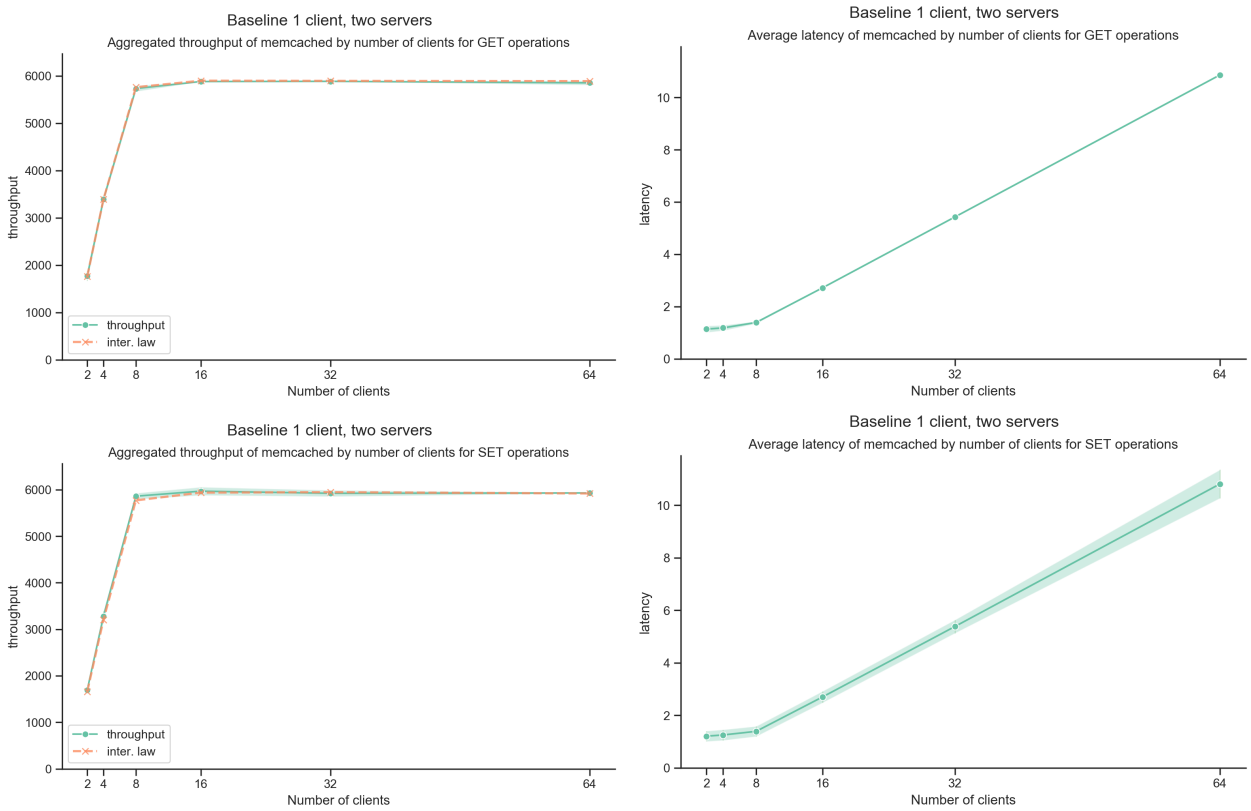


Figure 2: Results of the experiment. On the left column, we have aggregated throughput; on the right one there is average latency.

2.2.3 Explanation

As expected, the maximum throughput of both the read-only and the write-only experiments stays under 6000 op/s, and we already know that the cause is the network bandwidth.

Also, we can observe that the saturation ranges more or less stayed the same: the server is undersaturated between 2 and 4 clients (the throughput grows but the response time does not), and saturated from that point.

If we look at the plots of the average response time, we will find them similar to those of the previous section (in the fact that response time grows linearly), but now we can further inspect what happens with a small number of clients. In fact, we can see that up to 4 clients the servers are under-saturated, and then they saturate.

2.3 Summary

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

Maximum throughput of different VMs.

	Read-only workload	Write-only workload	Configuration gives max. throughput
One memcached server	3000 op/s	15130 op/s	(WO) 96 clients; (RO) 6 clients
One load generating VM	6000 op/s	6000 op/s	(WO/RO) 8 clients

(The maximum throughput for the write-only workload with one memcached server was selected as the point with 96 clients, not 192, as the throughput with 96 clients is a bit lower than the one with 192 clients but the response time is significantly less).

In the experiment with only onle load generating VM, we got the same results in the read-only and write-only workloads and concluded that in the case of the read-only workload, the bottleneck is the upload limit of the `memcached` servers, in the case of the write-only workload, the bottleneck is in the upload limit of the `memtier` clients.

Instead, when there are three load generating VMs, and only one server, we observe a very different result between write-only and read-only workloads. In particular, the read-only throughput is much lower, and we already found out that the reason is the same as before (the upload limit of the `memcached` server). For the write-only workload, we can observe the true behaviour of the `memcached` server, and how it cannot support more than 16000 `SET` operations per second.

In conclusion, we have learned the following about the `memtier` client and the `memcached` server:

- If the `memcached` servers are under-saturated, response time will grow very little, almost staying the same. Instead, if the servers are saturated, response time will increase proportionally to the number of clients.
- Read-only workloads are easily subject to hit the upload bandwidth of the `memcached` machines, even if there are two servers, and thus have an apparently low throughput, compared to the write-only workloads. But network upload limit can be hit during write-only workloads too, if there is only one `memtier` machine.

3 Baseline with Middleware (90 pts)

Now we will proceed to test the performance of the middleware by running some write-only and read-only workloads. We will first test the behaviour of a single middleware against a single `memcached` server, and then we will observe what changes when we add an additional middleware to the system.

3.1 One Middleware

In this experiment, three client machines, each with one instance of `memtier` with two running threads, will send requests to a single middleware. We will vary the number of virtual clients between (1,2,4,8,16,32), the number of worker threads between (8,16,32,64), and the type of workload (read-only or write-only). A single `memcached` server will answer requests from the middleware.

3.1.1 Hypothesis

We can't predict what the performance of the middleware will be, but we can still make some considerations. Assuming that our middleware is efficient and doesn't add any kind of bottleneck/slowdown due to computational load, we should observe:

- An increase in response time, since we are adding a new hop that requests must make. If we run a `ping` test, we can observe that we are adding a 1.0ms overhead on average on the time it took for a request to go from the `memtier` client to the `memcached` server.
- As a consequence of the first item, a decrease in the throughput, because of the interactive law. However, we should make a distinction: in the case of the `GET` requests, since we have only one server, it is very likely that we'll hit the upload bandwidth limit like in the previous experiments; in the case of the `SET` requests, throughput decrease should ideally stay close to what the following formula states (supposing there are enough worker threads to simultaneuosly satisfy all clients):

$$T^t_{middlware} = T^t_{baseline} \cdot \frac{R^t_{baseline}}{R^t_{baseline} + 1.0ms}$$

  where $T^t_{middleware}$ is the (hypothetical) throughput of the middleware with t worker threads, $T^t_{baseline}$ is the throughput of the write-only experiment of the first baseline with t clients (see Figure 1), and $R^t_{baseline}$ is the response time of the write-only experiment of the first baseline with t clients (Figure 1), because each worker thread in the middleware is like a client from the point of view of the `memcached` server.

  This formula was derived from the interactive law: if we are adding 1ms of latency, throughput should decrease proportionally to how much the latency increased. For example, on a small load with 6 clients, where response time is circa 1ms, we should see a throughtput of $6000 \cdot (1/2) = 3000$; instead, with a heavy load of 48 clients, where the baseline response time is higher at 4ms, the throughtput with the middleware should be $13000 \cdot (4/5) = 10400$.

- If there are more clients than worker threads, then some clients will need to wait that some worker thread will be free before having their request processed.

  In other words, the middleware is able to simultaneously satisfy as many requests as the number of worker threads. This means that, if there are more clients than threads, there will always be requests in the job queue, worker threads will never wait for a job, but will be always processing and sending requests to the server, and so the system will behave very similarly to a baseline without middleware with the number of clients equal to the number of worker threads.

- A direct consequence of the previous consideration is that no matter the number of clients, as long as they are more than the number of workers, throughtput should stay the same.

3.1.2 Results (read-only)

Here are the plots of the throughput and response time for the read-only experiment, as measured on the middleware: (the interactive law and error metric are present, but not visible because too close to the actual throughput values)

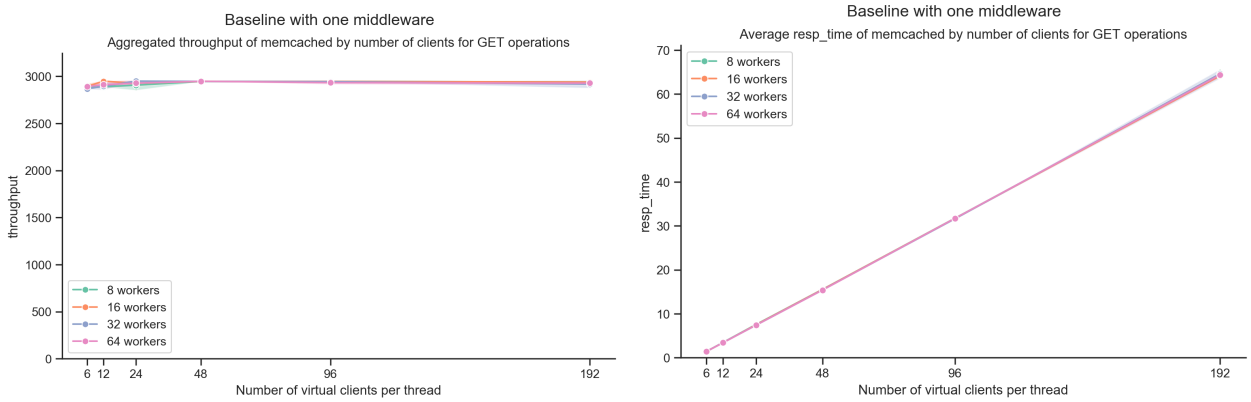


Figure 3

3.1.3 Explaination (read-only)

As we can see, the result we get is very similar to the one we obtained in the first baseline (Figure 1). Like before, the throughput is capped at 3000 op/s because of the upload bandwidth of the single `memcached` server. So we can say that, in this case the middleware doesn't slow down the system (other than the 1.0ms added latency)

3.1.4 Results (write-only)

Here instead are the plots from the write-only workload. The dashed lines represent the interactive law. The interactive law was calculated by adding 1.0ms of *thinking time* to the response time measured on the middleware, since it didn't comprehend the time it took for the request to travel from the client to the middleware and back.

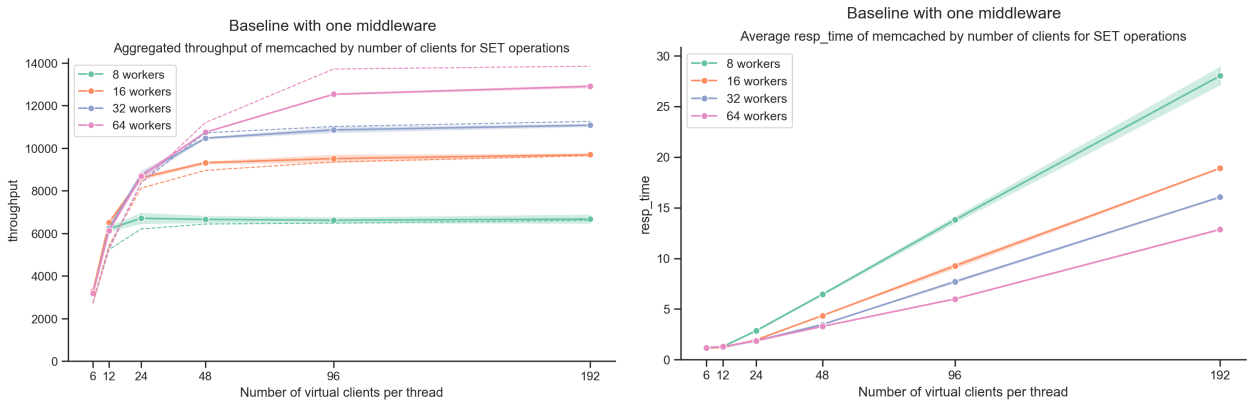


Figure 4

3.1.5 Explanation

This results here are significantly different from the previous baselines, and let us draw some interesting conclusions.

First, we can notice that the interactive law does not hold as close as before, and partly that's because it was calculated by adding the average 1.0ms ping time, which is not guaranteed that stayed constant during the experiment. This is particularly evident with a small load; with a high load this effect is much smaller, in fact we can see that from more than 48 clients, the interactive law is very close to the final throughput, *with the exception of the 64 workers configurations, but we will return on this later.*

Let's now test if our hypothesis were right or not:

- *As long as the number of worker threads is larger than the number of clients, the throughput seems to respect the formula*:

$$T^t_{middlware} = T^t_{baseline} \cdot \frac{R^t_{baseline}}{R^t_{baseline} + 1.0ms}$$

  In fact, we can see that, for example with 6 and 48 clients, the throughput with 64 worker threads is very close to our prediction (respectively, 3000 and 10400). This is good news: the middleware does not add any kind of noticeable slowdown other than the network latency needed to reach it.

- *When the worker threads are less than the clients, throughput should behave like the first baseline (Figure 1).*
  We can observe that this more or less is correct. For example, max throughput for 8 worker threads is around 6000, and in Figure 1 we can see that, even if we don't have the exact point, the throughput we should obtain with 8 clients is a similar number. The same reasoning can be repeated with 16, 32 and 64 worker threads.

- *When the worker threads are less than the number of clients, the throughput should stay the same, no matter the number of clients.*

We can see that this is not always the case. A clear example is that with 16 worker threads, throughput definitely increases between 24 and 48 clients. After 48 clients, though, throughput stays the same. The only plausible explaination is that with even with 24 clients, there are some worker threads idling and waiting for a job to arrive. Let's plot the average queue size to see if this is the case:

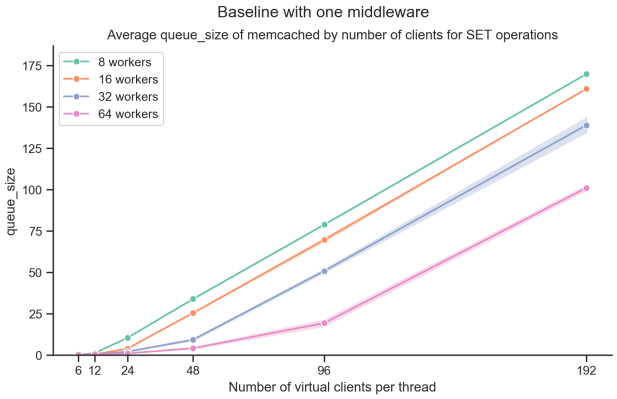


Figure 5: Average queue size of the one middleware write-only experiment

From this plot we can observe how with 24 clients and 16 worker threads, the queue size is, on average, very small. This means that's very likely that there are some threads that go idle waiting for a job to arrive, even if there are more clients than workers. With 48 clients, this no longer happens. Probably, this is an effect of the network latency that separates the client from the middleware, and/or the fact that requests can arrive and be sent in batches due to network spikes.

In conclusion, the results we obtained were more or less what we expected, except for the interactive law for 64 workers, which is considerably different than the throughput. Let's plot the average waiting time for a response from the `memcached` server and the average time spent processing a request by the net thread, to see if something fishy happened with the 64 threads configuration:

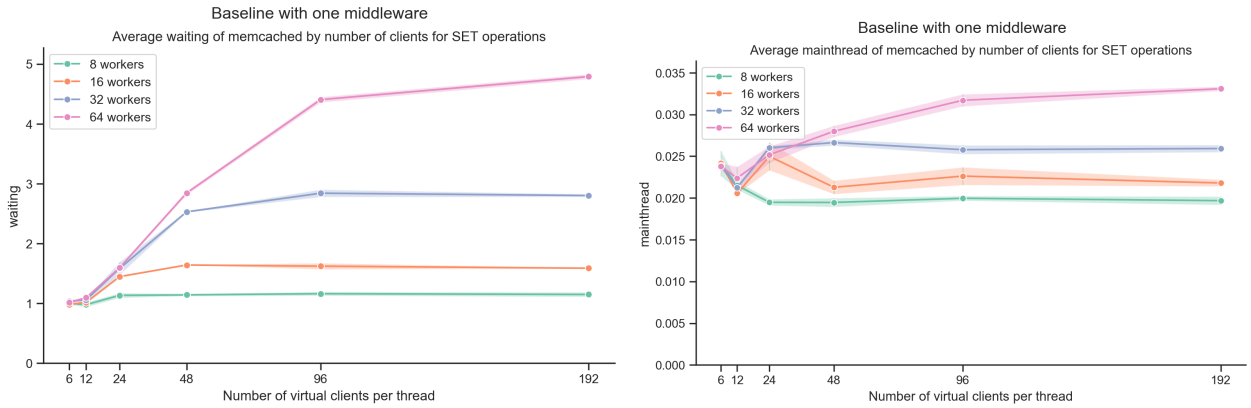


Figure 6: On the left, the average time a worker thread has to wait for a response from the server. On the right, the average time spent by the net thread (mainthread) to process a request.

Both plots do not suggest any strange behaviour: waiting time increases with 64 threads due to the load on memcached side, since we are making much more requests simultaneously (we can confirm this effect in the first baseline, Figure 1), and also the time spent by the main thread parsing requests by the main thread increases too, but not that much to justify the discrepancy between throughput and interactive law.

The most probable conclusion is that the response time reported by the middleware is off, since the throughput is at a value we expected. One reason could be the fact that the execution of so many threads on a single machine makes the net thread wake up more rarely. In fact, response time is measured from when the net thread receives the request, to when a worker thread sends the response back to the client; however, if the net thread is sleeping when the request arrives, the request will wait in the buffer of the network interface before being picked up by the net thread, and that time is not considedered in the measurement of the response time.

3.2 Two Middlewares

Now we test the differences in the results we obtain when we split the load of the `memtier` clients between two middlewares instead of only one.

The number of virtual clients, and the number of threads per middleware, are exactly the same as the previous experiment. The only difference is that we use two instances of `memtier` on each client machine instead of one; however each instance of `memtier` has only one thread, so the total number of clients remains the same.

3.2.1 Hypothesis

Compared to the previous experiment, the only significant thing that changes is that the computational load of the middleware is split in half, since the number of total clients is the same, but now only half are connected to a single middleware simultaneously.

Thus, those are the things the we should observe:

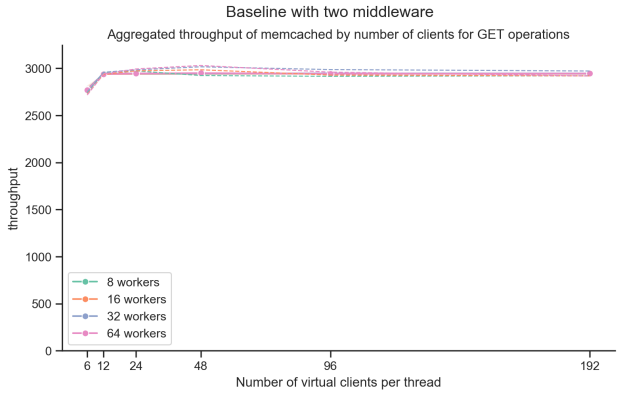
- Since only half of the clients are connected to each middleware, the average queue size should be smaller than before (ideally it should be exactly half).
- The average waiting time that a worker thread waits for the answer from `memcached` should increase, especially with a high number of worker threads, as now the `memcached` server must answer to up to 128 threads simultaneously (while before it was only 64).
- The formula from before, if there are more threads than clients, should hold again, albeit with a slight change:

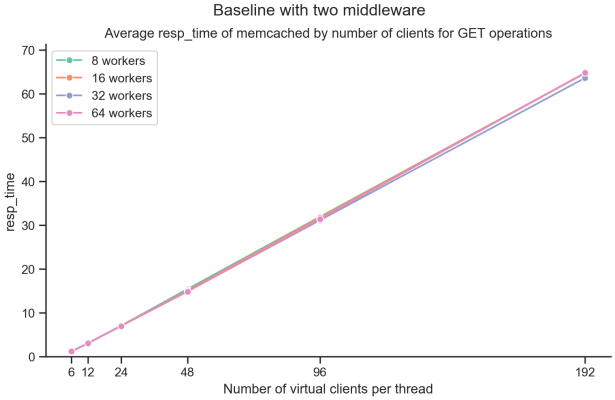
$$T^{t}_{middlware} = T^{2t}_{baseline} \cdot \frac{R^{2t}_{baseline}}{R^{2t}_{baseline} + 1.0ms}$$

  Now instead of $T^{t}_{baseline}$ we have $T^{2t}_{baseline}$ because since we have two middlewares running, it's like having double the number of worker threads as clients to `memcached`.

3.2.2 Results (read-only)

Here are the plots from the read-only workload. (Once again, interactive law and error metric are present, but not very visible).





3.2.3 Explaination (read-only)

Here, too, we can observe the upload limit of the server as the bottleneck of the system. Also, as in all other read-only experiments repeated before, the system is saturated since the start (6-12 clients), and response time grows proportionally to the number of clients.

3.2.4 Results (write-only)

Here instead are the plots from the write-only workload. As before, the interactive law was calculated by adding 1.0ms of *thinking time* to the response time.

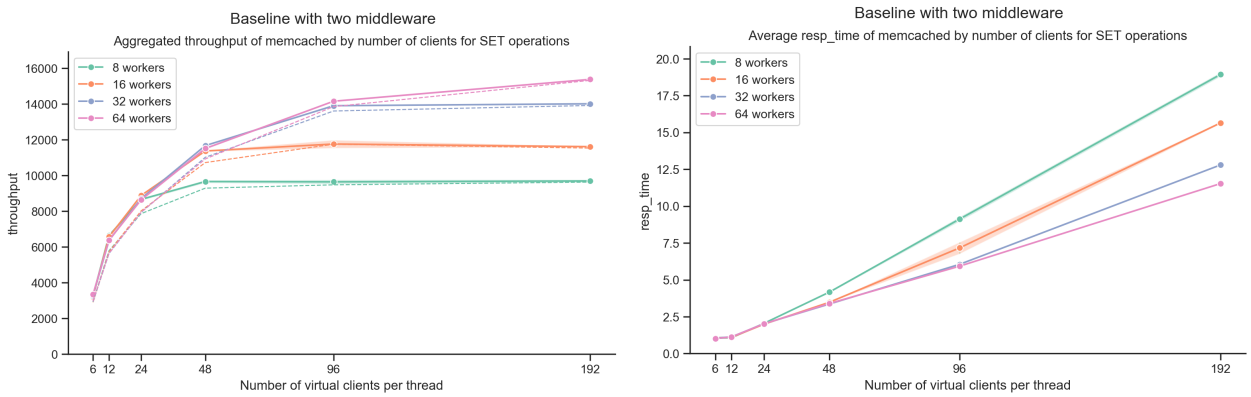


Figure 8

3.2.5 Explaination (write-only)

This time the interactive law follows much more closely the throughput we obtained. The biggest difference from before is that there's no more the discrepancy seen in the previous experiment with 64 workers. This can have many explainations, including the fact that the network was more stable during this run, but the most probable cause is that this time each middleware had half the load, giving more space to the net thread to pick up quickly new requests, resulting in a more correct measurement of the response time.

Now let's have a look if our hypothesis were right or not:

- *With only half of the load, the queue size should me much smaller.*
  Here's a comparison of the average queue size based on the number of clients and workers:

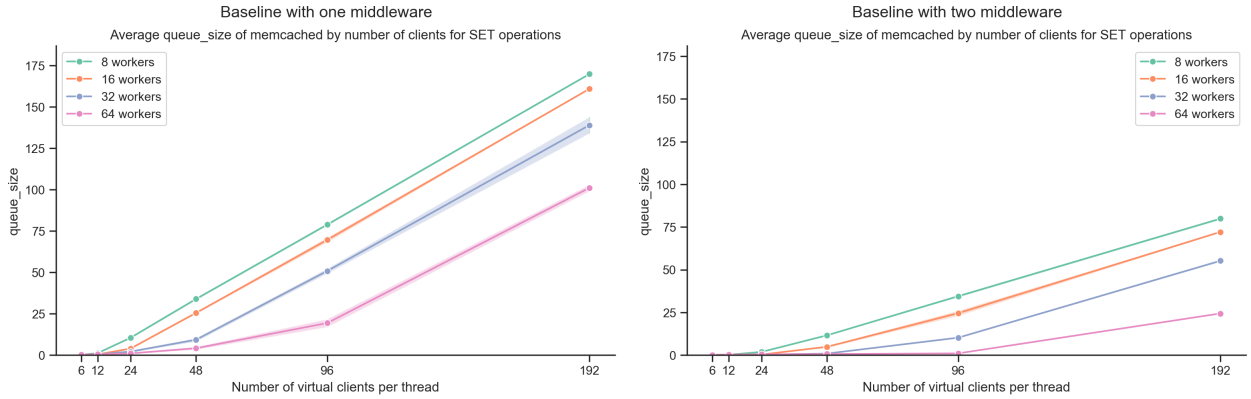


Figure 9: On the left, average queue size with one middleware; on the right, with two middlewares.

From what we can see, the result is as expected, the mean queue size is smaller than before.

- *The average waiting time that a worker waits for the answer from* `memcached` *should increase.*
  Here's a comparison of the average waiting time:

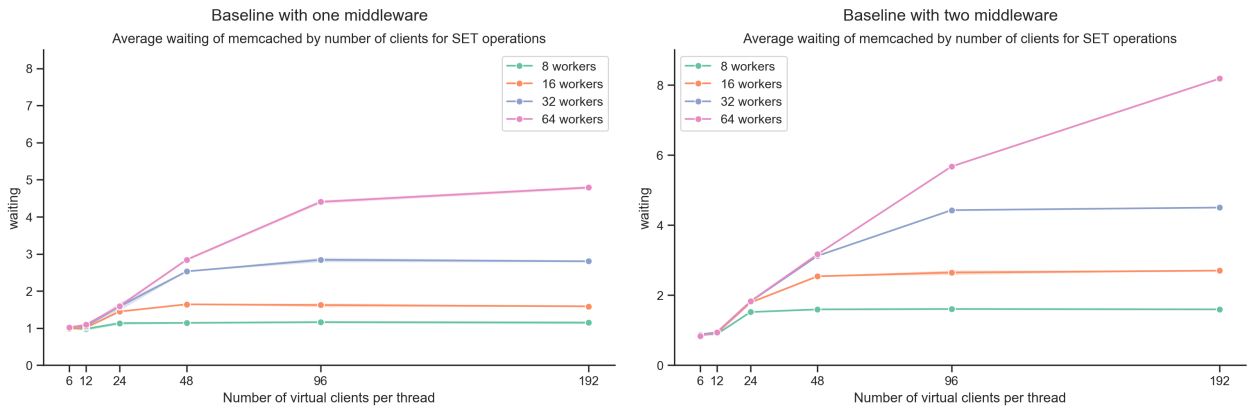


Figure 10: On the left, average waiting time with one middleware; on the right, with two middlewares.

As we can see, waiting time has increased indeed, as now the server must fullfill many more requests simultaneously.

- *If there are more workers than clients, the following formula should hold*:

$$T^{t}_{middlware} = T^{2t}_{baseline} \cdot \frac{R^{2t}_{baseline}}{R^{2t}_{baseline} + 1.0ms}$$

We can see from the plots that this formula doesn't hold as before: for example with 64 worker threads the throughput with 48 clients should be $150000 * (7/8) = 13000$, while the actual throughput is 12000; with 64 worker threads and 24 clients, the thoughput should be $13000 * (4/5) = 10400$, while the actual throughput is around 9000.
We can notice a slight decrease from the ideal throughput that the formula gives us, around 10%, maybe from the fact that the client machines have more difficulties running two separate instances of `memtier`, or more probably, from the fact that machines were shut down and restarted before doing this new experiment, and they could have been relocated, making that 1.0ms ping time measurement not accurate anymore. Unfortunately, ping times were not tested again after this restart of the machines.

3.3 Summary

The maximum throughput configuration, both for one middleware and for two middlewares, was selected as the one with 96 clients. In both cases, the configuration with 192 clients provided a throughput slightly higher, but the response time was almost double, and it did not seem a reasonable choice.

Maximum throughput for one middleware.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware	2950	3.4	0.14	0
Reads: Measured on clients	2946	4.08	n/a	0
Writes: Measured on middleware	12536	5.99	1.46	n/a
Writes: Measured on clients	12467	7.66	n/a	n/a

Maximum throughput for two middlewares.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware	2939	3.08	0.13	0
Reads: Measured on clients	2939	4.13	n/a	0
Writes: Measured on middleware	14155	5.92	0.16	n/a
Writes: Measured on clients	14134	6.79	n/a	n/a

Based on the data gathered from these experiments, we can draw the following conclusions:

- We have learned that when there are more clients than workers, (in other words: when the job queue is never empty) we can model the system by removing the middleware and using as the number of clients the number of worker threads of the middleware.

- *Write-only load:* In the single middleware setup, the bottleneck is *not* the middleware per se, but the low number of threads that it has; we have concluded the throughput is significantly less than the baseline without middleware for the added 1.0ms network latency, and for the fact that the middleware can satisfy up to 64 requests simultaneously, not 192. There may be some overhead due to the high number of threads, but it should not affect performance so much.

  For the two middlewares setup, we can satisfy up to 128 requests simultaneously, because each middleware can run up to 64 threads, achieving over than 10% more throughput than the single middleware. The performance is still not comparable to the baseline, but the causes should be found in the network, not in the computational load: we saw that even at the maximum throughput configuration, with 96 clients, the job queue is almost always empty.

  Response times between the one and two middleware configuration are very similar however, but this is also because we can satisfy more requests simultaneously (in fact, if we look at the average time spent in the queue, we can see that while for two middlewares is almost zero, for the one middleware configuration is almost 1.5 ms)

  The performance obtained is maximum when using the highest number of worker threads available.

- *Read-only load:* no matter the configuration, quickly over-saturates because of the upload bandwidth of the single `memcached` server. We can confirm this by observing that the average

queue time, both for one and two middlewares, is almost zero: most of the time is spent waiting for a response from `memcached`. There is virtually no difference between the one middleware and two middlewares configurations.

The performance obtained is not dependent on the number of worker threads.

4 Throughput for Writes (90 pts)

4.1 Full System

For this experiment we will use all machines available at our disposal. We will observe what performance changes introduce the addition of two new `memcached` servers in the write-only workload of the two middlewares setup. We will use the same variable number of virtual clients as before (1,2,4,8,16,32) and the same variable number of threads as before (8,16,32,64).

4.1.1 Hypothesis

The only difference between this experiment and the previous one with two middlewares is the addition of two new servers. This is what we expect from this experiment:

- Since a `SET` request must be forwarded to all present `memcached` servers, now each worker thread has to send three requests instead of one, wait for three answers (one for each server), and then respond back to the client, so the time spent waiting for an answer from `memcached` should increase, and as a consequency, the thoughput of the system should decrease.

  Each `memcached` server, instead, receives exactly the same load as before, so we expect a similar performance from the servers.

- In this experiment the upload capacity of the middleware will be severely tested. By running a test with `iperf3`, we can see that the upload speed limit is about 800 Mbps. The maximum amount of `SET`s that a single middleware can handle is:

$$\frac{bandwidth}{set\_size} \text{ op/s} = \frac{800 \text{ Mbit/s}}{(4096 + 10) * 8 \text{ bytes}} = 24354 \text{ op/s}$$

  Since each `SET` must be sent three times, once per server, we should divide that number by 3, and we obtain 8118 op/s. However, we have two middlewares running, and that means that the upload capacity limits the aggregated throughput at 16236 op/s.

4.1.2 Result

Here are the plots for aggregated throughput (with the adjusted interactive law) and average response time:

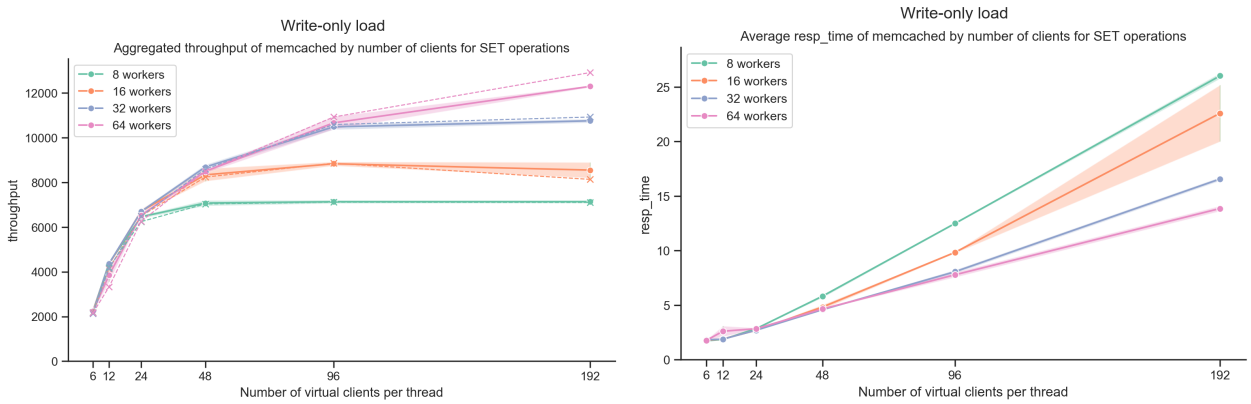


Figure 11

We can see that something wrong happened on the test with 192 clients and 16 workers; maybe a spike in network latency or a very small downtime of one of the machines.

4.1.3 Explanation

Now, let's review our hypotheses:

- *The time waiting for* `memcached` *should have increased.*

  We can immediately see that the maximum throughput reached by the system is much lower than before (this time the max throughput is around 10-12k, while before it was around 14k). However, the slowdown could be introduced by the middleware.

  Let's try to compare average queue length from before:

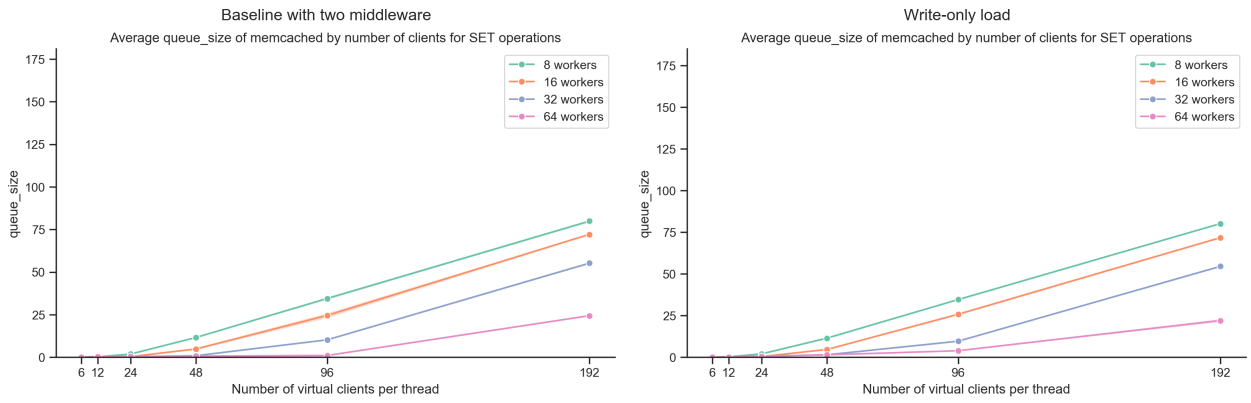


Figure 12: On the left, average queue size of the previous experiment (write-only workload, two middlewares); on the right, average queue size in this current experiment.

We can observe that queue size doesn't change that much from before; we have to look for the slowdown elsewhere. Let's try to look at a comparison of the average waiting time of each

worker thread for an answer from `memcached`:

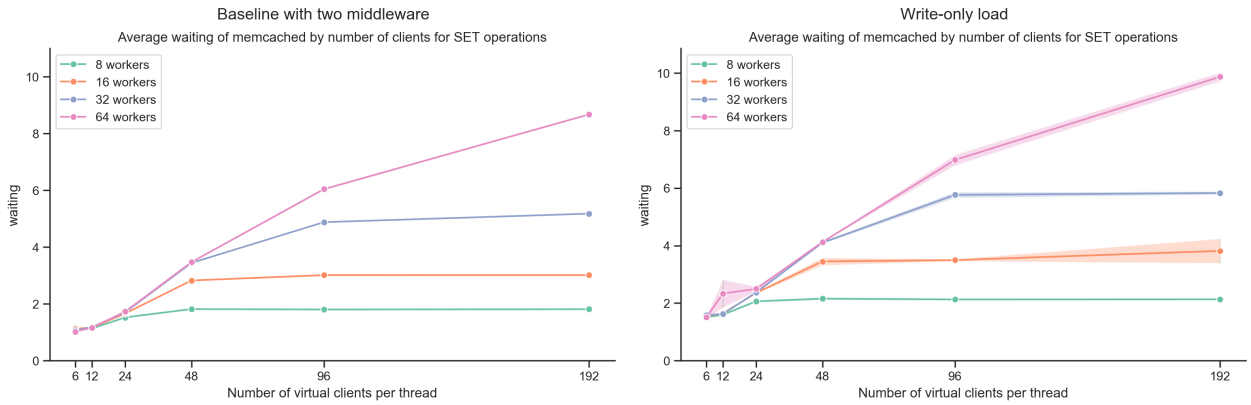


Figure 13: On the left, average waiting time of the previous experiment (write-only workload, two middlewares); on the right, average waiting time in this current experiment.

From the plots we can see that the waiting time is very similar from before, just slightly higher. For example, the waiting time with 192 clients and 64 worker threads is, on average, 8.18 ms for the previus experiment, and 9.86 ms in the current experiment.

This additional 1.5ms of delay could be caused by the fact that each worker thread must wait for the answer of three different servers, not only one. It could also be caused by a slower performance of `memcached`, but unfortunately we have no way of confirming that.

However, this 1.5ms delay justifies the decrease of the throughput. In fact, if we take the max throughput from before (14k), and use the interactive law to determine the decrease caused by an additional 1.5ms delay, we get:

$$throughput * \frac{response\_time + thinking\_time}{response\_time + thinking\_time + delay} = 14155 * \frac{12.5 + 1}{12.5 + 1 + 1.5} = 12739$$

which is close enough to the results we got.

- *The throughput shouldn't be higher than 16236 op/s, because of the bandwidth upload limit of the middleware.*

  From the data we got we can quickly confirm this hypothesis.

4.2 Summary

When selecting the maximum throughput to fill this table, in many cases it was not selected the point with the highest *absolute* throughput, but often it was selected the second highest throughput if the difference in throughput was not significant while the difference in response time was considerable.
For 8 worker threads, the data point selected for max throughput was with 48 clients.
For 16 worker threads, the data point selected for max throughput was with 96 clients.

For 32 worker threads, the data point selected for max throughput was with 96 clients.
For 64 worker threads, the data point selected for max throughput was with 192 clients.

It's for this reason that, for example, the average length of the queue with 32 threads is less than with 64 threads in the following table: because there are a different number of clients.

The "Throughput (Derived from MW response time)" was calculated using the interactive law, with thinking time equal to 1.0 ms.

Maximum throughput for the full system

	WT=8	WT=16	WT=32	WT=64
Throughput (Middleware)	7077	8836	10488	12300
Throughput (Derived from MW response time)	6246	8858	10591	12918
Throughput (Client)	7117	8936	10496	12299
Average time in queue	3.50	6.15	2.10	3.72
Average length of queue	11.49	25.81	9.63	22.01
Average time waiting for memcached	2.15	3.49	5.76	9.86

From the conclusions drawn before, and from the table above, we can derive the following key points:

- As we saw before in the summary of Section 3, the performance obtained is maximum when using the highest number of worker threads.

  The maximum throughput of the system is a bit lower than what we saw in Section 3, but as explained before this is caused by an additional 1.5ms delay in waiting for an answer from `memcached`. This could be caused by the fact that the answer now must be waited from three servers instead of one.

- The average time spent waiting for an answer from `memcached` increases with the number of worker threads used. This is expected: since we are using more threads at the same time, the `memcached` server must satisfy more request simultaneously, and while this could give us higher overall performances, it also means that each request takes more time to process.

- The length of the queue, in all four worker thread configurations under a heavy load, is never very low. This means that the net thread is not slow, and worker threads are never wasting time idling because the net thread isn't fast enough pushing jobs into the queue.

- With a low number of worker threads (8 and 16), the time spent in the queue is higher than the time spent waiting for `memcached`. This means that the main bottleneck of the system is the middleware (or better, the low number of threads), since requests are spending much more time in the queue waiting to be processed than actually being processed by `memcached`.

  Instead, when the number of threads is higher (32 and 64), we start to see that requests spend a lot of time being processed by `memcached`, more than waiting in the queue. This tells us that the `memcached` server is starting to suffer from the load, and we can consider it the bottleneck of the system in this case. (Please note that the average time in queue in the table for 32 threads is smaller than the one with 64 threads, but that is only because

the maximum throughput with 32 threads was selected with 96 clients, while the maximum throughput with 64 threads was selected with 192 clients).

5 Gets and Multi-gets (90 pts)

In this set of experiments we will test the system with a mixed read and write workload, but observing the performance differencies when varying the number of keys in the `GET` requests and by varying the modality in which `GET` requests are forwarded.

We will use all machines available to us: three clients, two middlewares and three servers. The number of virtual clients for each `memtier` instance is fixed to 2.

In this section, results will be aggregated at the end to make confronting plots between the *sharded* and *non sharded* case more easily.

5.1 Configurations

5.1.1 Sharded Case

We will run a mixture of `GET` and `SET` operations, and measure the performance when `GET`s will have 1,3,6 and 9 keys included in a single operation.

The `GET` operations will be *sharded*, meaning that each worker thread will distribute the work equally between multiple servers, in the case of a multi-key `GET`.

The chosen number of worker threads in the middleware is 64. The reasoning behind this is that even though the number of worker threads in a read-only workload doesn't affect the overall performance (as we can see in Figure 3, Section 3), there are some `SET` requests too in this experiment, and Figure 8, Section 3, clearly shows that more worker threads are better in the presence of `SET` requests.

5.1.2 Non-sharded Case

This experiment has the same configuration as the previous one, with only one difference: `GET` requests with more than one key will not be distributed to multiple servers, but they will be simply forwarded entirely to a single server.

For the same reasons as above, we use 64 worker threads on each middleware.

5.2 Results

Here are the results for the average response time (measured on the client):
(The interactive law and error metric are not very visible, but present)

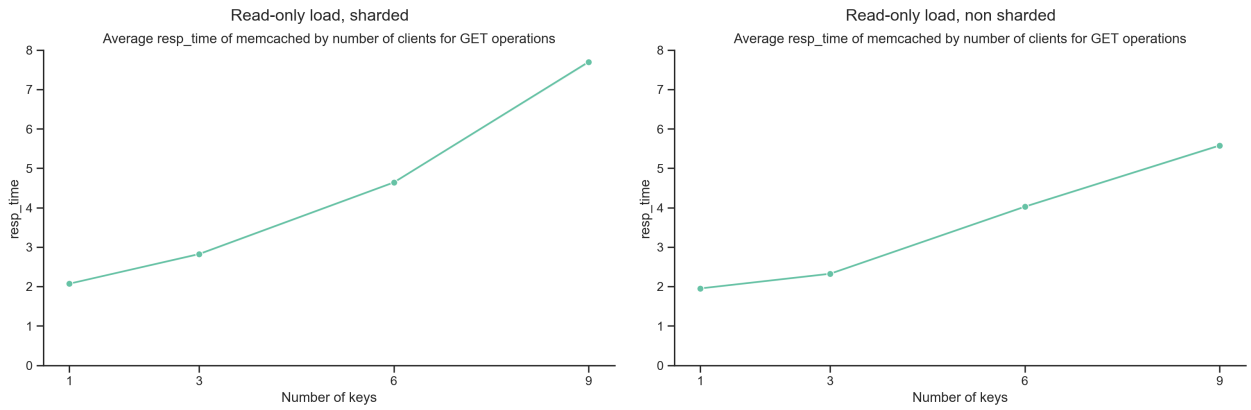


Figure 14: On the left, sharded; on the right, non-sharded.

Here are the results for the percentiles (measured on the client):

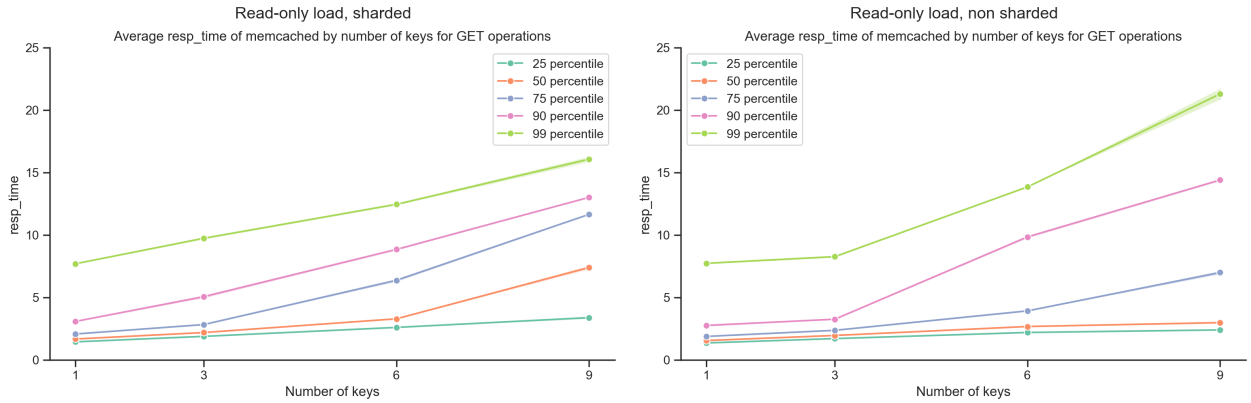


Figure 15: On the left, sharded; on the right, non-sharded.

5.3 Explanation

Looking at the average response time plots of the *sharded* case, we see how the response time increases as we increase the number of keys that are included in a request. The increase in response time is not so high from 1 to 3 keys, but starts to get significant once we use 6 or 9 keys. The same considerations can be made about the *non-sharded* case; in fact, we can also note that the two plots are very similar.

Let's know try to understand if the data we got is reasonable. From our previous analysis done with `iperf3` in Section 2, we know that a single `memcached` server cannot satisfy more than 3000 `GET` requests per second. Let's look at the plot of the adjusted throughput measured on the client, where we account for a multi-get request as many single individual `GET`s. For example, a 6-key multiget is considered as 6 different `GET` operations.

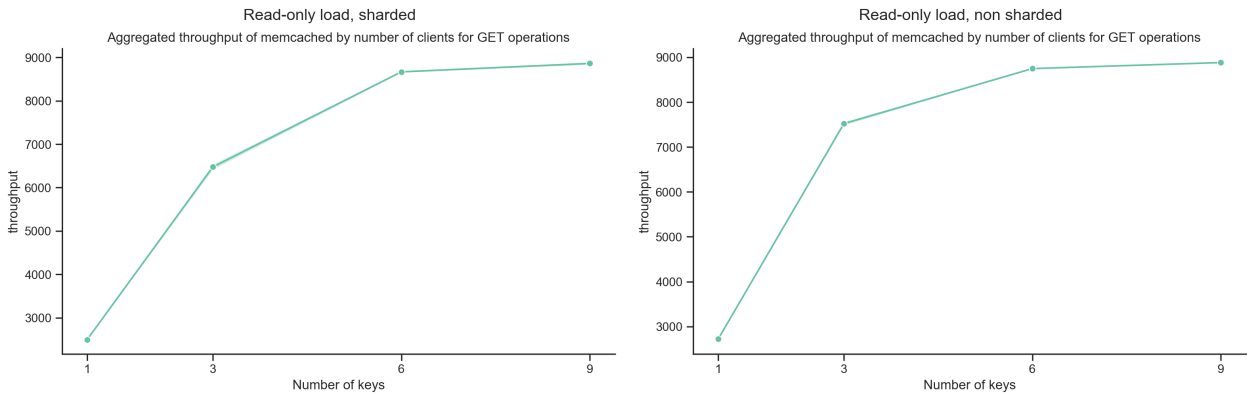


Figure 16: On the left, sharded; on the right, non-sharded.

As we can see, the non-sharded configuration performs a bit better with 3 keys multi-gets, but apart from that the two plots are very similar. The important thing to notice is that the adjusted throughput doesn't get any higher than 9,000 (because, as it was shown earlier, each server can satisfy up to 3000 `GET`s/s).

So, we concluded that the obtained data is consisted with what we saw in previous experiments.

Let's have a look at the percentiles graph now (Figure 15):

Comparing the *sharded* and *non-sharded* case, we can see how with 1 key the results are exactly the same, and this is expected as 1 key multiget requests are treated as normal `GET` requests.

The results are more interesting from 3 keys and higher: the *sharded* configuration has higher values for the 25th, 50th and 75th percentiles; the 90th percentile is more or less the same between the two configurations; the 99th percentile, instead, is lower for the *non-sharded* configuration.

This peculiar behaviour can be explained in the following way: There are three servers answering request, each with a different latency; there is probably one of the servers that performs worse or is further away than the other two, and the response time relative to that server is higher. When using the *sharded* configuration, the middleware always asks all three servers, including the worst one; so the total response time is lower bounded by the response time of the worst server. The *non-sharded* configuration, instead, asks servers in a round-robin fashion, and we can assume that one third of the time it asks the worst server. So, the average response time of the *non-sharded* is lower.

However, ther **worst case** response time is higher, and to explain that let's use an example. Suppose we have a `GET` request with 9 keys; if the *non-sharded* configuration decided to ask the worst server, it will ask all 9 keys. The *sharded* configuration instead will only ask 3 keys to the worst server. The worst server will take more time to answer a `GET` request with 9 keys compared to one with only 3, and so it is plausible that for some requests the *sharded* configuration achieves a smaller response time, and this explains why the corresponding 99th percentile is lower.

5.4 Histogram

Here are the histograms representing the response time distributions:

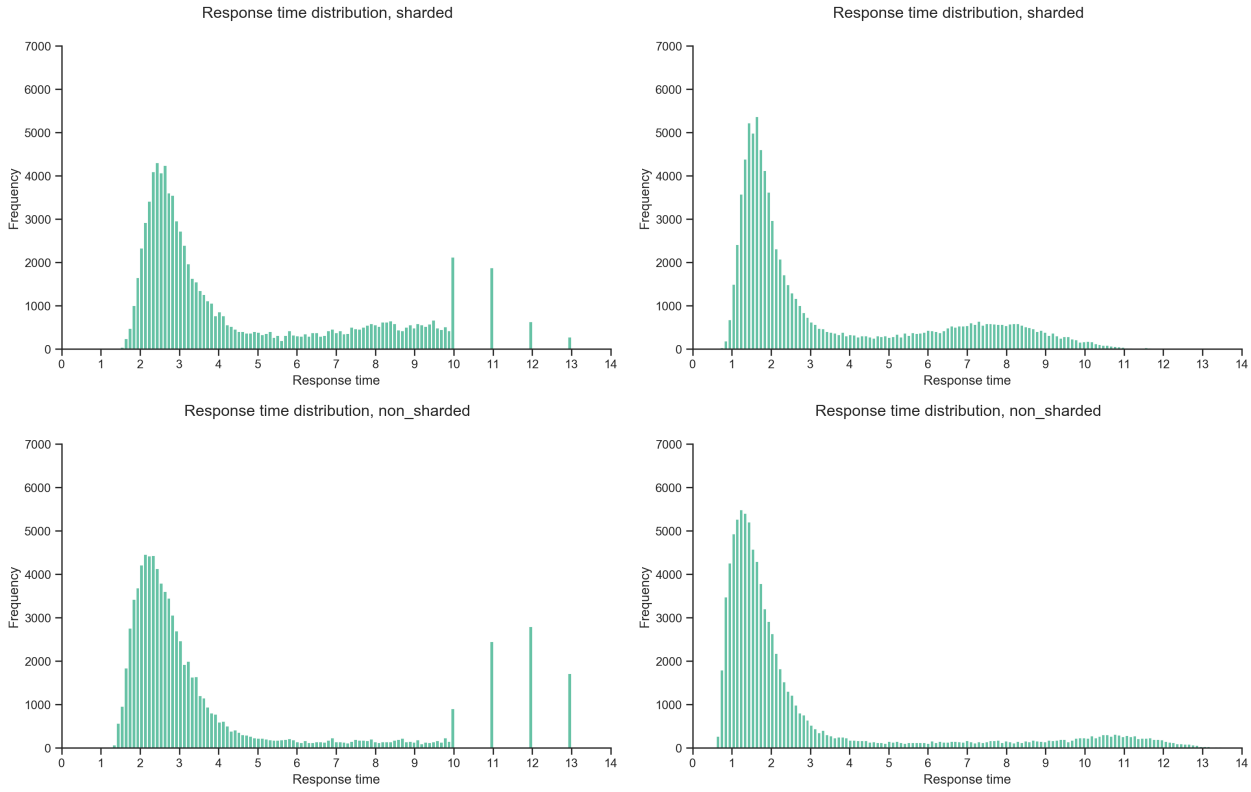


Figure 17: On the top, sharded; on the bottom, non-sharded. On the left, histograms from the `memtier` client, on the right, histograms from the middleware.

The bucket size is 0.1ms (except for the left column, with measurements from `memtier`, which doesn't provide fine-grained statistics for requests that take over 10ms).

As we can see, the histograms confirms the results we got in the percentiles before: the average response time of the *non-sharded* configuration are lower (we can see this because the big bell-shaped wave is shifted to the right, if we compare the plots vertically, both the left column and the right column, the first being relative to measurements on the client, the second one to measurements on the middleware); however, the *sharded* configuration presents much less cases of very high response times: especially if we look at measurements on the middleware (right column), we can see that in the *sharded* case there are almost no requests that took over 10ms, while there are plenty that did so in the *non-sharded* case.

5.5 Summary

The key take-away points from this experiment are:

- With 1 key, there is no difference between *sharded* and non *sharded*.
- We can make the following considerations about the overall throughput (Figure 16): With 3 keys, the non *sharded* configuration performs slightly better.

With 6 and 9 keys, there is no difference between *sharded* and non *sharded*, because we've hit the upload bandwidth limit of the `memcached` servers (9000 op/s).

- The average response time for the *sharded* case is higher than the *non-sharded* for 3, 6 and especially 9 keys. However, ther **worst case** response time is actually lower for the *sharded* case. As explained before, this is because the *non-sharded* configuration may ask an entire request to a server that could be further away than the other two, while the *sharded* configuration always splits request between servers.

- From the histograms we can observe how response times measured on the client are greater by more or less 1 ms than response times measurements made on the middleware.

  This is consistent with previous findings in Section 3, where we found out that the round trip time between the client and the middleware machine was 1.0 ms.

6 2K Analysis (90 pts)

The different factors and their assigned names used in this 2^k analysis are shown in the following table:

Letter	Parameter	Value for -1	Value for +1
A	Number of servers	1	3
B	Number of middlewares	1	2
C	Number of worker threads	8	32

We will use an additive model to estimate how much each factor affects the overall performance. If we'll observe that the additive model is not sufficient, we will use the multiplicative model too.

Our basic model looks as follows:

$$y_{ij} = q_0 + q_A x_{A_i} + q_B x_{B_i} + q_C x_{C_i} + q_{AB} x_{AB_i} + q_{AC} x_{AC_i} + q_{BC} x_{BC_i} + q_{ABC} x_{ABC_i} + e_{ij} \quad (1)$$

with e_{ij} being the error due to repetitions.

We can now calculate the parameter estimation q_j as follows:

$$q_j = \frac{1}{2^k} \sum_{i=1}^{2^k} S_{ij} \overline{y_i} \quad (2)$$

with S_{ij} being the (i, j)th entry in the sign table, and $\overline{y_i}$ being the mean of the r repetitions.

We can now proceed to calculate several sums of squares that we will use to determine the effect

of single effects on the variation of y

$$\text{SSY} = \sum_{i=1}^{2^k} \sum_{j=1}^{r} y_{ij}^2 \tag{3}$$

$$\text{SS0} = 2^k r q_0^2 \tag{4}$$

$$\text{SST} = SSY - SS0 \tag{5}$$

$$\text{SS}j = 2^k r q_j^2, j = 1, 2, \ldots, 2^k - 1 \tag{6}$$

$$\text{SSE} = SST - \sum_{j=1}^{2^k-1} \text{SS}j \tag{7}$$

Since the experiment was done with repetitions, there will be an additional "Error" factor in the tables to account for variation of results in individual repetitions.

Also, confidence intervals will be listed in the tables for each factor's effect. The chosen confidence level is 90%. We assumed that experiment results were distributed along a Gaussian distribution, and so we used the Student's t distribution critical value of 1.746, taken from a precalculated table[2].

6.1 Write-only

Here are the results for the write-only workload:

	Throughput			Response time		
Factor	Variation	Effect	Confidence Interval	Variation	Effect	Confidence Interval
A	**19.40%**	-1098	(-1155 , -1041)	**18.54%**	3.6997	(3.4948 , 3.9046)
B	**26.62%**	1286.7	(1229.7 , 1343.8)	**24.77%**	-4.275	(-4.480 , -4.071)
C	**53.09%**	1817.2	(1760.1 , 1874.2)	**44.03%**	-5.701	(-5.906 , -5.496)
AB	0.45%	168.15	(111.12 , 225.18)	4.05%	-1.728	(-1.933 , -1.523)
BC	0.14%	-92.08	(-149.1 , -35.05)	2.54%	-1.368	(-1.573 , -1.163)
AC	0.00%	-11.14	(-68.17 , 45.883)	4.93%	1.9075	(1.702 , 2.11)
ABC	0.02%	34.547	(-22.48 , 91.576)	0.84%	0.7887	(0.5838 , 0.9936)
Error	0.27%			0.30%		

As we can observe, results for throughput and response time are pretty close. The only noticeable difference is that the signs of the effect where there is a high variation are opposite; but that makes sense, since an increase in the response time would mean a decrease in the throughput.

The variation effect is high on each individual factor (highlighted in bold), and low when factors are combined: this means that factor have a small co-variance. This is especially true for the throughput results; looking at the response time results, we get some co-variance between factors, but always below 5%.

[2] `https://en.wikipedia.org/wiki/Student%27s_t-distribution`

According to the results, what affect the performance the most is factor C, the number of worker threads. This reflects our results in Section 3 (baseline with two middlewares) and Figure 11, Section 4; more worker threads means more requests accomplished at the same time (in fact, the "effect" on the throughput is positive, and the "effect" on the response time is negative).

Another important factor however is B, the number of middlewares. Increasing the number of middlewares brings a positive effect to the throughput, and we already saw this in the summary of Section 3 (comparing the write-only throughput between one and two middlewares). In fact, adding another middleware is very similar to duplicating the number of worker threads. The variance is not as high as factor C, probably because factor C determines a four-fold increase in worker threads (between 8 and 32).

Also, we can't ignore the variance caused by factor A (the number of servers). This time, however, the "effect" is negative on the throughput and positive on the response time; in fact, having more servers is worse for the performance of our system, because a middleware will have to forward a `SET` request to three servers instead of only one, and wait for the answer of each of them. We already encountered this behaviour in Figure 13, Section 4, when comparing the results with the baseline of Section 3 which had only one server.

6.2 Read-only

Here are the results for the read-only workload:

	Throughput			Response time		
Factor	Variation	Effect	Confidence Interval	Variation	Effect	Confidence Interval
A	**99.28%**	2819.5	(2805.5 , 2833.6)	**99.78%**	-21.49	(-21.64 , -21.34)
B	0.13%	101.66	(87.625 , 115.70)	0.01%	-0.213	(-0.366 , -0.060)
C	0.13%	101.82	(87.787 , 115.86)	0.05%	-0.486	(-0.638 , -0.333)
AB	0.12%	99.854	(85.813 , 113.89)	0.06%	-0.527	(-0.680 , -0.374)
BC	0.13%	100.87	(86.837 , 114.91)	0.01%	-0.197	(-0.349 , -0.044)
AC	0.10%	-88.32	(-102.3 , -74.28)	0.01%	0.1886	(0.0359 , 0.3413)
ABC	0.10%	-90.03	(-104.0 , -75.99)	0.05%	0.4935	(0.3408 , 0.6462)
Error	0.01%			0.03%		

As we can see, results for throughput and response times are almost the same, where a single factor, A (the number of servers) takes up all the variance. The "effect" is positive for throughput and negative for response time; this means that our system performs better with multiple servers than with a single one. From our analysis done with `iperf3` in Section 2, in the read-only workload, we already know that the read-only experiment is prone to suffer from the upload bandwidth limit on the `memcached` machine instances, so it's not surprising that adding more instances of the server will increase throughput and decrease response time.

We can further confirm this by looking at the summary of Section 2, where we observe that using two servers instead of one for the read-only workload causes the throughput to significantly increase.

All other factors have negligible variation.

6.3 Conclusion

We can state that the 2^k analysis confirmed our previous findings in discovering which component of the system was affecting the performance.

We also saw that the variance is almost always attributed by factors composed by the variation of a single component, instead of factors composed by a combination of component variation. For this reason, we can assume that the additive model we used is enough for our purpose, and we do not need to make further observations with a multiplicative model because co-variances are already very low.

7 Queuing Model (90 pts)

7.1 M/M/1

Let's start by modelling our middleware as a system with a single processor, distinguishing between each number of worker threads.

The two inputs to the model are the arrival rate λ and the service rate μ.

We can assume that the arrival rate λ equals to the throughput measured on the middleware for a given configuration of clients and worker threads. This is because the system is closed, and each `memtier` instance that sent a request will wait for the answer of that request before sending another one.

As for the service rate, μ, we will take the highest observed throughput that the middleware has reached in any given repetition, with any given number of clients for a given configuration of worker threads, because that should be the *maximum* performance of our system can reach.

We will analyze our model based on the following parameters:

$\boldsymbol{\mu}$		Service rate
$\boldsymbol{\lambda}$		Arrival rate
$\boldsymbol{\rho}$	$= \frac{\lambda}{\mu}$	Traffic intensity (Utilization)
$\boldsymbol{E[n_q]}$	$= \rho^2/(1-\rho)$	Expected mean number of jobs in the queue
$\boldsymbol{E[r]}$	$= \frac{1/\mu}{1-\rho}$	Expected mean response time
$\boldsymbol{E[w]}$	$= \rho\frac{1/\mu}{1-\rho}$	Expected mean waiting time

Here are the results: (*Note: the values corresponding to the average queue size gathered from the middlewares were doubled, because the M/M/1 models a single queue, while with two middlewares we are calculating the average between the two*).

Clients	λ	μ	ρ	Response time	$E[r]$	Queue size	$E[n_q]$	Queue time	$E[w]$
6	2241.55	7229	0.310	1.74	0.20	0.12	0.14	0.08	0.06
12	4391.24	7229	0.607	1.86	0.35	0.56	0.94	0.11	0.21
24	6465.18	7229	0.894	2.84	1.31	4.02	7.57	0.62	1.17
48	7077.13	7229	0.979	5.82	6.60	22.98	45.71	3.51	6.46
96	7138.61	7229	0.988	12.49	11.10	69.22	78.25	10.20	10.96
192	7141.17	7229	0.988	26.03	11.42	160.34	80.60	23.73	11.29

Table 1: 8 worker threads

Clients	λ	μ	ρ	Response time	$E[r]$	Queue size	$E[n_q]$	Queue time	$E[w]$
6	2247.52	8872	0.253	1.78	0.15	0.12	0.09	0.08	0.04
12	4341.33	8872	0.489	1.90	0.22	0.42	0.47	0.11	0.11
24	6715.53	8872	0.757	2.69	0.46	0.84	2.36	0.17	0.35
48	8340.11	8872	0.940	4.82	1.88	9.36	14.75	1.19	1.77
96	8836.88	8872	0.996	9.84	28.73	51.64	252.90	6.16	28.62
192	8553.37	8872	0.964	22.60	3.14	143.50	25.91	18.60	3.03

Table 2: 16 worker threads

Clients	λ	μ	ρ	Response time	$E[r]$	Queue size	$E[n_q]$	Queue time	$E[w]$
6	2216.59	10844	0.204	1.82	0.12	0.14	0.05	0.09	0.02
12	4369.51	10844	0.403	1.89	0.15	0.40	0.27	0.11	0.06
24	6699.36	10844	0.618	2.70	0.24	1.24	1.00	0.17	0.15
48	8689.04	10844	0.801	4.58	0.46	3.00	3.23	0.31	0.37
96	10488.64	10844	0.967	8.06	2.81	19.28	28.54	2.10	2.72
192	10759.31	10844	0.992	16.57	11.79	109.08	125.88	10.53	11.70

Table 3: 32 worker threads

Clients	λ	μ	ρ	Response time	$E[r]$	Queue size	$E[n_q]$	Queue time	$E[w]$
6	2236.08	12342	0.181	1.77	0.10	0.18	0.04	0.10	0.02
12	3853.31	12342	0.312	2.61	0.12	0.40	0.14	0.12	0.04
24	6532.68	12342	0.529	2.83	0.17	1.04	0.60	0.18	0.09
48	8500.18	12342	0.689	4.65	0.26	2.94	1.52	0.34	0.18
96	10666.45	12342	0.864	7.78	0.60	7.84	5.50	0.59	0.52
192	12300.69	12342	0.997	13.86	24.47	44.14	299.96	3.72	24.39

Table 4: 64 worker threads

As we can observe from the tables, ρ is always smaller than 1, which means that the system is stable. If we increase the arrival rate, the utilization (which is equal to ρ) increases too. The utilization with 8 worker threads increases very quickly, reaching saturation levels already when setting the arrival rate equal to what we get with 24 clients. Instead, with 64 worker threads, the utilization increases more slowly, reaching saturation levels only when we set the arrival rate equal to what we get with 96 or 192 clients.

Let's make comparisons between the data from experiments and the predictions made by the model:

- Response time - $E[r]$:
  We can see how the response time and its prediction are not always close for 8, 16 and 32 worker threads: they are very low when the system is undersaturated, and then they get more accurate as we increase the arrival rate. For 64 worker threads in particular, the predicted response time is heavily underestimated. One major motivation for this could be that the M/M/1 model does not take into account the increasing parallelism that we are using between the different configuration. When using more threads, and obtaining a higher service rate, the M/M/1 model just assumes we're using a faster processor, and predicts a very low response time in case the arrival rate is low. Moreover, the M/M/1 model is not aware of the base latency that is present between the client and the middleware, which contributes too to the difference.

- Queue size - $E[n_q]$ and Queue time - $E[w]$:
  We can observe how both parameters are very closely predicted by the model, for the majority of the data points. This is also thanks to the fact that the queue size data gathered from the middleware was doubled to simulate having a single queue instead of two parallel ones.

7.2 M/M/m

Now, let's model our middleware as a system with a multiple processors, distinguishing between each number of worker threads.

The two inputs to the model are the arrival rate λ, the service rate μ, and the number of parallel processors m.

λ remains unchanged from before, as it equals to the throughput measured on the middleware for a given configuration of clients and worker threads, because the system is closed. As for the service rate, μ, we will take the highest observed throughput that any single given worker thread has reached in any given repetition, with any given number of clients, because that should be the *maximum* performance that a single processor can reach. As for the number of processors, m, it will be equal to the total number of worker threads at our disposal. Since there are two middlewares, for the 8 worker thread configuration m will be equal to 16; for the 16 worker thread configuration, m will equal 32, and so on.

This time however, for space constraints, the formulas used to calculate the parameters will not be shown, and we will analyze only one configuration, the one with 32 worker threads. Here is the result:

Clients	λ	μ	ρ	Response time	$E[r]$	Queue size	$E[n_q]$	Queue time	$E[w]$
6	2216.59	183	0.189	1.82	5.46	0.14	0.00	0.09	0.00
12	4369.51	183	0.373	1.89	5.46	0.40	0.00	0.11	0.00
24	6699.36	183	0.572	2.70	5.46	1.24	0.00	0.17	0.00
48	8689.04	183	0.742	4.58	5.47	3.00	0.04	0.31	0.00
96	10488.64	183	0.895	8.06	5.70	19.28	2.49	2.10	0.24
192	10759.31	183	0.918	16.57	5.88	109.08	4.49	10.53	0.42

Table 5: 32 worker threads

Here, too, ρ is always smaller than 1, which indicates a stable system. The utilization however increases more slowly than before (M/M/1 model), and this is expected as we are using the maximum throughput a single thread has ever obtained as μ, and the M/M/m model assumes that all processor are independent , with only the job queue shared between them, while in our system they share the same resources (CPU and memory), and more importantly, they share the same `memcached` servers.

This is a particularly important point: since we're doing a write-only workload, and `SET` request must be forwarded to all three servers, we could argue the fact that the performance of each worker thread depends on the performance of the worst server. This invalidates the assumption that each of the m processor is independent, and, as a result, we do not expect very accurate result from this model.

Let's make comparisons between the data from experiments and the predictions made by the model:

- Response time - $E[r]$:
  The response time is not predicted very correctly. We can observe how it stays constant until we use an arrival rate correspondent to less than 96 clients, and this is expected, as since the model thinks we have 64 independent processors, having less than 64 clients will not incur in penalties in the response time. We can confirm this behaviour by looking at the predicted queue size: it is zero, until we use an arrival rate corresponding to 96 clients.

- Queue size - $E[n_q]$ and Queue time - $E[w]$:
  The predictions here are not very close either, but at least they follow the same trends of the data we obtained: queue time and queue size are very low until we use an arrival rate corresponding to 96 client and higher, at which point they rise sharply. Both parameters are heavily underestimated compared to data from the experiments, but probably the reason is the same as before, that the system does not *really* have m independent processors, because they share the same `memcached` servers.

7.3 Network of Queues

In this final section we will try to model our system as a network of queues, a model in which jobs departing from one queue arrive at another queue. In particular, our model will be a *closed*

queueing network, in which the jobs are finite and there are no external arrivals or departures. In this way we can build a complex model that fits better our system, and we are able to perform a more detailed analysis of the bottleneck.

We will use an M/M/1 system to model the network thread inside the middleware. The service time was measured subtracting to the average response time as measured on the client (excluding the RTT between client and middleware) the average response time measured on the middleware (excluding the enqueueing time).

We will use an M/M/m system to model the worker threads inside a single middleware, as there are multiple processes that share the same job queue. The service time was obtained from the middleware logs, and it's the average time spent by a worker thread processing a single request before forwarding it to the server. The m parameter was selected as equal to 64, as we want to model our best-performant system (the one with the highest number of worker threads).

We will use an M/M/1 system to model the `memcached` server. The service time was obtained from Section 2. For simplicity, we will include the network time spent in sending and receiving a request from the server in the service time. Since we saw that the maximum throughput for a write-only load is 16,000, the service time will be $1/16,0000$. For a read-only throughput, instead, the service time will be $1/3,000$.

We will model the clients as a delay center, with delay equal to the RTT between the middleware and the clients (1.0ms). We do not model them as a queue since from previous experiments we already excluded them from being the bottleneck of the system.

The service times of the middleware threads (both worker and network threads) do not change between write-only and read-only. They change between the two different configurations (one and two middlewares); that is probably because when we use two middlewares, each of them is under half the load compared to the configuration with only one middleware, and so its performance may change (although, the differences are not that strong).

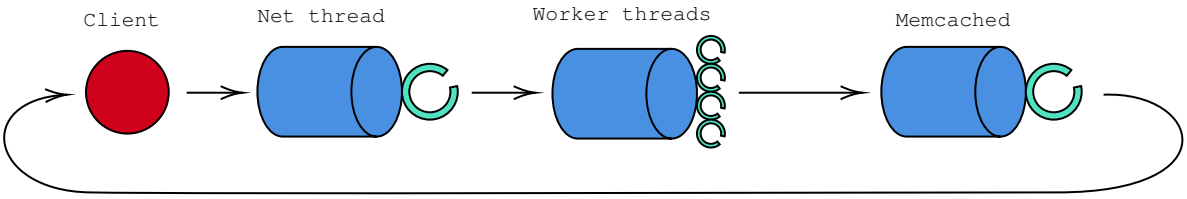
Here is a detailed table of every input parameter given to the model:

Component	***One middleware***		***Two middlewares***	
	Read-only Service time	**Write-only Service time**	**Read-only Service time**	**Write-only Service time**
Network thread	0.081 ms	0.081 ms	0.104 ms	0.104 ms
Worker thread	0.012 ms	0.012 ms	0.023 ms	0.023 ms
Server	0.0625 ms	0.3 ms	0.0625 ms	0.3 ms

Since one of the queues in our network is M/M/m, we will need to use the extended MVA algorithm.

7.3.1 One middleware

This is a very simple diagram of how our network of queues looks like: (In red, delay centers; in blue, queues)



Here is a table of the utilization that the algorithm gave us as output for both the read-only and write-only experiments, depending on the number of jobs that circulate in the system:

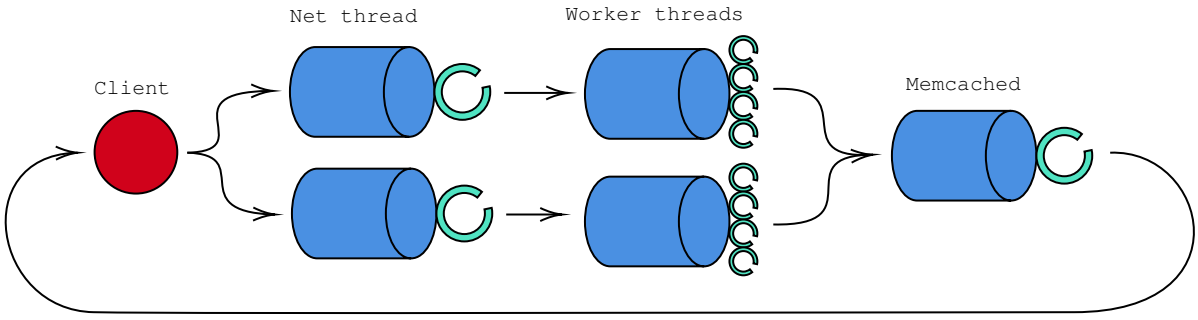
Write-only workload						
	6 jobs	12 jobs	24 jobs	48 jobs	96 jobs	192 jobs
Network thread	0.218	0.428	0.786	0.998	1.000	1.000
Worker threads	0.004	0.008	0.015	0.019	0.019	0.019
Server	0.169	0.330	0.607	0.771	0.772	0.772
Read-only workload						
	6 jobs	12 jobs	24 jobs	48 jobs	96 jobs	192 jobs
Network thread	0.171	0.238	0.243	0.243	0.243	0.243
Worker threads	0.003	0.004	0.005	0.005	0.005	0.005
Server	0.705	0.982	1.000	1.000	1.000	1.000

We can immediately notice how in the read only experiment the bottleneck is the server, as its utilization quickly gets very close to 1. This is in line with our findings in Section 2; in fact, for the way we modeled the server, network latencies are included in the service time of the server, so it makes sense that the server is highlighted as bottleneck.

For the write-only experiment, instead, we can see how the model highlights the bottleneck as the network thread, as its utilization gets very close to 1 from when there are 48 jobs in the system.

Worker threads have very low utilization values: this is because they do not comprehend network time, but only the cpu time needed to parse the request before sending it to the server.

7.3.2 Two middlewares



(In red, delay centers; in blue, queues).

Here is a table of the utilization that the algorithm gave us as output for both the read-only and write-only experiments, depending on the number of jobs that circulate in the system:

	6 jobs	12 jobs	24 jobs	48 jobs	96 jobs	192 jobs
Write-only workload						
Network thread	0.099	0.196	0.385	0.698	0.833	0.833
Worker threads	0.001	0.003	0.006	0.010	0.012	0.012
Server	0.119	0.236	0.462	0.838	1.000	1.000
Read-only workload						
	6 jobs	12 jobs	24 jobs	48 jobs	96 jobs	192 jobs
Network thread	0.085	0.142	0.156	0.156	0.156	0.156
Worker threads	0.001	0.002	0.002	0.002	0.002	0.002
Server	0.547	0.906	1.000	1.000	1.000	1.000

We have another confirmation of the server being the bottleneck in the read-only experiment. But the more interesting part is the write-only experiment: we can see from the table very high utilization values for both the network thread and the `memcached` server; however, ultimately it is the server that gets flagged as the bottleneck. This is in line with our findings in the summary of Section 4, when we found out that the majority of the response time of the middleware is made of waiting for an answer from the server, not of the time spent in the job queue.

7.3.3 Throughput estimates

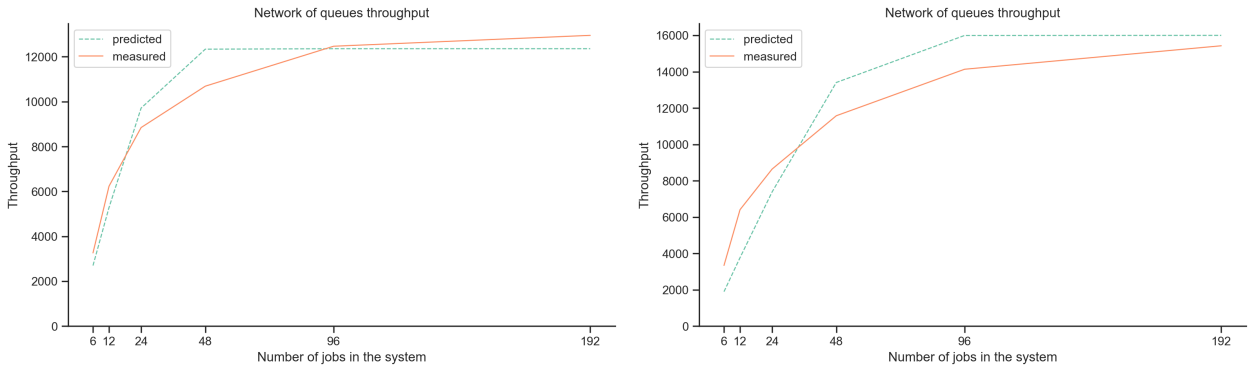


Figure 18: On the left, predictions of the model for one middleware (write-only). On the right, predictions of the model for two middlewares (write-only).

We can observe how both models are not very far off from the data measured from the experiments, but sometimes the trend are a bit different: in the one middleware configurations, the model gets completely saturated when there are 48 jobs, much more earlier than the real middleware. The same thing happens with the two-middlewares model: the prediction gets saturated more quickly.