Advanced Systems Lab Report

Autumn Semester 2017

Name: Florian Chlan
Legi: 16-931-933

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

Contents

1 System Overview (75 pts)

This report explains the work I did for the 2017 "Advanced Systems Lab" course at ETH Zurich. The task at hand was to design and implement a middleware system in Java for the memcached Key-Value-store and subsequently analyze its performance through a series of experiments conducted on the Microsoft Azure cloud.

1.1 Component Overview

To start off we would like to give an overview of the implemented middleware system by first highlighting the most important components:

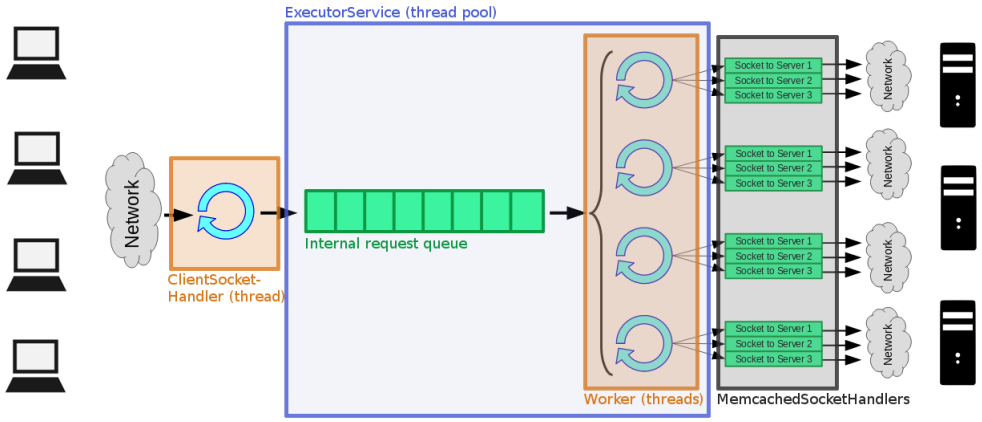


Figure 1: System overview with threads marked in orange and the single request queue

RunMW This is the main-class that boots up the middleware. It parses the command line parameters, initializes the thread pool and starts the ClientSocketsHandler. Additionally it install a shutdown-hook that will react to SIGINT signals by shutting down the middleware gracefully.

ClientSocketsHandler The "network thread". This component runs on a separate thread and sets up and accepts new client connections from the network. It is also responsible for parsing requests and – if successful – submit them to the ThreadPoolExecutor. The `ClientSocketsHandler` employs non-blocking IO by using Java's `Selector`[1] interface.

Request Incoming requests are encapsulated in a Request object containing all necessary information for processing. Request objects are only created for valid requests. Behind the Request class there's a class hierarchy supporting the different types of requests (GET, SET and MultiGET). These subclasses contain the concrete implementation on how to handle requests of different types. In addition, this object also holds timing information gathered during the requests traversal through the system.

[1] https://docs.oracle.com/javase/8/docs/api/java/nio/channels/Selector.html

ThreadPoolExecutor For managing the worker threads and also queueing the requests, we instantiate a `ThreadPoolExecutor`[2] - Java's standard thread pool implementation. For queueing of the requests, a single, unbounded LinkedBlockingQueue is used. The reason for this design decision is that the take()-operation of these queues is blocking and – due to the fact that it uses a linked list – is unbounded. The thread pool is of fixed size with the number of threads being passed on from the command line.

Worker A worker is a generic request processor using Java's `Runnable`[3] interface. It can handle any request types, as it is agnostic to different request types. For each request a Worker object is created and submitted to the ThreadPoolExecutor. The Worker object is not reused after the corresponding request has been processed completely. To prevent the system from setting up new connections to the memcached servers for every request, we use thread-local sockets (that only exist once per thread and are reused by Workers).

MemcachedSocketHandler Finally, this component is responsible for establishing the connection to and communicating with all specified memcached servers. As mentioned before, these components are instantiated on a per-thread basis using `ThreadLocal`[4] to reduce the overhead of establishing socket connections. They keep the connections open until shutdown.

The whole system has been implemented and tested with Java 8 64bit.

[2] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html
[3] https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html
[4] https://docs.oracle.com/javase/8/docs/api/java/lang/ThreadLocal.html

1.2 Request Lifetime

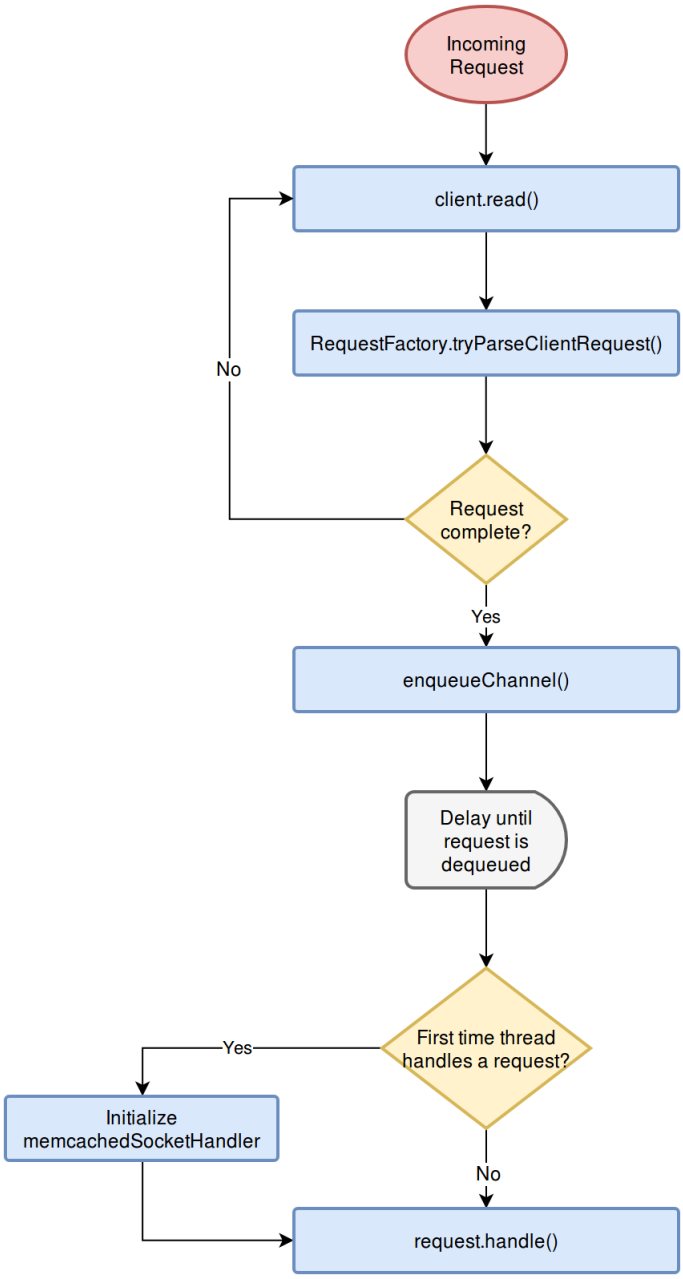


Figure 2: Lifetime of a request from receival until enqueue in the thread pool.

Now we'd like to discuss the lifetime of a request as it passes through the middleware. The first part from receival of a request until it is enqueued is illustrated in Figure 2. Upon establishment of a new client connection, a new request buffer is allocated and any available data is read from the client socket. There is one request buffer per client and it is reused for the whole client connection. It is of fixed size 3000 bytes to accommodate MultiGETs of up to 10 keys with a maximum length of 250 characters each. A RequestFactory tries to parse the incoming message and create a valid `Request` object from it. Should the message received so far be incomplete, the system skips this client and handles other client sockets until more data is received. If successful, the network thread submits the request to the thread pool for processing. At this point, the `RequestFactory` has already dissected the incoming request

by its type, but still all requests are enqueued in the same thread pool queue.

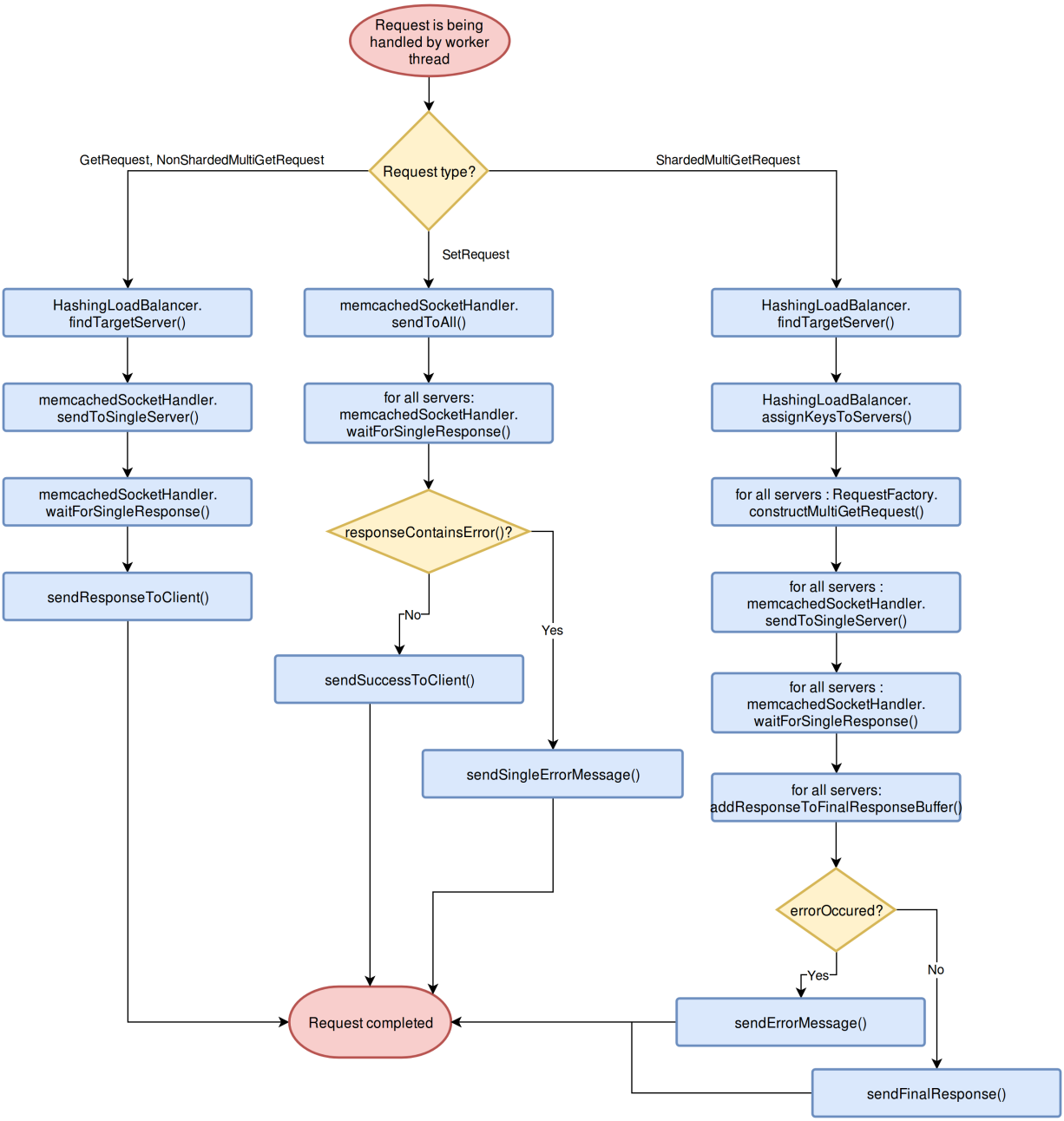


Figure 3: Lifetime of a request from dequeuing until the request is completely processed.

In the `ThreadPoolExecutor`, the threads dequeue requests in a FIFO fashion. Each thread will be able to handle each of the four request types. When a request is dequeued, it is processed as follows: (also see illustration 3)

Single GET-requests are to be forwarded to a single memcached server and the answer is fed back to the client. To determine the memcached server, we hash the requested key and take the modulo of the number of memcached servers. More information on the fairness of the hashing-method can be found in Section 1.4.

SET-Requests are forwarded to each memcached server. Only if all store-operations have been successful, a success message is forwarded to the client.

Non-sharded MultiGETs are similarly handled as single GET-requests. The entirety of the requested keys is hashed and then forwarded to the designated memcached server.

Sharded MultiGETs are handled in a special manner. Similar to the non-sharded case, one server is designated to be the primary server using hashing. This server will be hit (non-exclusively) with the most keys of this request in a fair manner. The remaining keys are then assigned to the other servers in a round-robin fashion, making sure that the other servers get hit with at worst as many keys as the primary server. Again, more information on this procedure can be found in Section 1.4.

Afterwards all the responses from the memcached servers will be collected and the final answer to the client is constructed. For the responses, answer buffers of size 11000 bytes are used to fit MultiGET responses of up to 10 keys. As the request traverses the system, as few memory allocations or object initializations as possible are done. This has been achieved by passing around the request buffers for each request from the network thread to the workers and immediately using them to forward to memcached. Analogously, the response buffer from memached is directly used to send answer back to the client. Through this only few copying operations and almost no String-object instantiations are needed.

Statistics are gathered on a per-request basis and are intermediately stored in the Request object. On completion of the request, they are logged to disk using log4j 2.9.1. No aggregation is done by the middleware, but will instead be handled in postprocessing using python. To keep the disk-usage low and performance impact to a minimum, requests are sampled at random for logging. Through this, about 5 % of the requests are stored to disk. In my opinion this sampling leads to a good compromise between logdata quantity, performance and representativeness of the sample.

The following timestamps are taken throughout a request lifetime:

initializeClockTime The real-world clock time when the Request object is initialized.

firstReadTime The first time that some bytes from this request have been received (If the message is not received by the network thread in one Selector iteration)

previousArrivalTime The time when the previous request (from any client) has been received completely.

arrivalTime The time when the current request has been received completely.

initializeTime The time when the Request object is created.

enqueueTime The time when the request is enqueued in the thread pool queue.

dequeueTime The time when the request is dequeued from the thread pool queue.

beforeSendTime The time before the request is sent out to the memcached server. If there are multiple servers involved, the time is taken before sending the request to the first memcached server.

afterReceiveTime The time after all the answer from the involved memcached servers have been received.

completedTime The time after the answer to the client has been written to the client channel and the request is completed.

lastAfterLogWrite The time after the last logged request has been written to disk.

The requests are logged on a per-thread basis, that means one csv-file per thread will be created (e.g. `requests_Thread-8.csv`). Additionally, a general `mw.log` contains all other log messages from the middleware. All logfiles are placed in a timestamped folder in the `./logs` directory of the middleware repository. Log4j2 is configured to not immediately flush on each log()-invocation, it buffers logs until the buffer size limit configured in the log4j2.xml configuration file is reached. On average the middleware showed a small disk write activity every 10 seconds.

1.3 Error Handling

Here I'd like to highlight some of the implemented middleware behaviours in case of abnormal situations. The middleware generally checks incoming clients requests for adherence to a subset of the memcached protocol (only GETs, MultiGETs and simple SETs without any flags are supported. Similarly, answers from the memcached servers are dissected. Should memcached send a valid error message to a request, then the middleware will forward this message to the client. If multiple error messages are received (in the case of SET or sharded GET requests), the first message is sent back to the client. If an unexpected response is received, the middleware will log this event and then shut itself down. The same will happen if the connection to one of the memcached servers is lost, the middleware will shutdown. Should we loose the connection to a client on the other hand, then the system will evict the buffers associated with this client and simply close the connection.

1.4 Request Load Balancing

The requirements state that the middleware should support up to three memcached server, upon which load should be distributed evenly. To accomplish this we employ simple hashing using Java's standard `String.hashCode()`[5] method.

A hashing experiment has been conducted to test whether the load is distributed evenly. The experiment can be found in the JUnit test class `ch.ethz.asl.worker.test.HashingLoadBalancerTest`. We conduct tests with completely random key strings as well as keys of the form `memtier-<random number>`, which memtier generates by default. Also we test if MultiGETs are balanced evenly.

```
testSingleKeysWithRandomStrings
Server 0 was hit 332881 times
Server 1 was hit 333248 times
Server 2 was hit 333871 times

testMultiGetsWithPrefix
Server 0 was hit 332959 times
Server 1 was hit 334231 times
Server 2 was hit 332810 times

testSingleKeysWithPrefix
Server 0 was hit 333835 times
Server 1 was hit 333775 times
Server 2 was hit 332390 times
```

[5] https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#hashCode–

```
testShardedServerAssignmentWithPrefix
Server 0 was hit 2166116 times
Server 1 was hit 2165601 times
Server 2 was hit 2166601 times
```

One can clearly see from the experiment above, that the employed load balancing scheme distributes work in a fair manner among the memcached servers and not only for single GETs or MultiGETs in a non-sharded setting. Also with sharding turned on, MultiGETs are split up in a way that the servers have to fetch approximately the same amount of keys.

2 Baseline without Middleware (75 pts)

In these first experiments we study the performance characteristics of the memtier clients and memcached servers. This and all the experiments following after have been conducted on Microsoft Azure. For the clients, VM instances of type Basic_A2, for the middlewares instances of Basic_A4 and for the memcached servers, instances of type Basic_A1 have been used. The nominal runtime of a single run was 82 seconds. Of the experiment data, the first 10 seconds and last 10 seconds were deemed warm-up and cool-down phases and thus have been truncated. All experiments have been repeated three times.

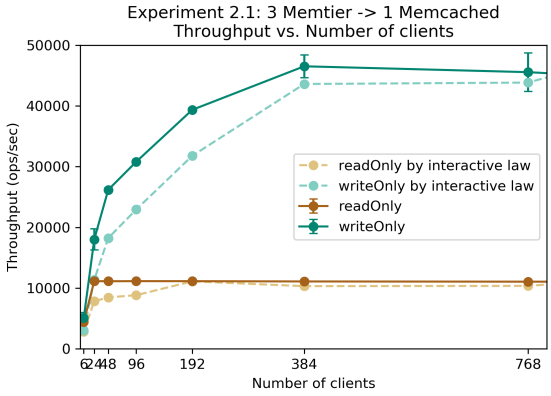
2.1 One Server

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[1,4,8,16,32,64,128,256]
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

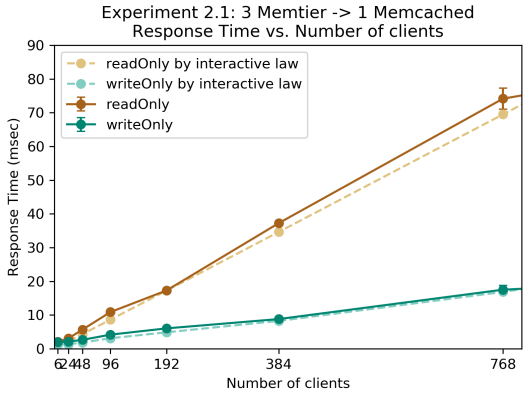
For this experiment we connect three clients to a single memcached instance, varying the number of virtual clients per thread between 1 and 256. Additionally, write-only and read-only workloads with single keys has been tested. For the throughput and response time measures below, the periodic output from memtier (every second) has been extracted. First each timestep is averaged over the three repetitions, and then an average over all timesteps excluding warmup and cooldown phases is taken. To then merge the metrics from multiple memtier processes, the throughput is simply summed up and the response times are aggregated by a weighted average with the weights being the respective throughputs.

Figure 4 shows the results. Two-sigma errorbars have been drawn, hence they show a 95% confidence interval. For some data points, the error bars are hidden by the data points, as the confidence intervals are very narrow. For the write-only workload, Figure 4a shows that the system achieves maximum throughput at around 384 clients. One caveat though is that at 384 clients, the response times have already risen fourfold. The increase in throughput in 4a correlates with the drastic change in CPU load seen in Figure 5b. Hence I conclude that the memcached VMs CPU limits write-performance in this setup.

The read-only workload on the other hand shows a much lower performance, as we already reach saturation around 24 clients. Network performance tests using `iperf` show an asymmetric network performance. A single client VMs could push up to 25 MB/s to the other VMs, while the memcached VM could only push around 12 MB/s (see Table 1). All VMs tested can receive

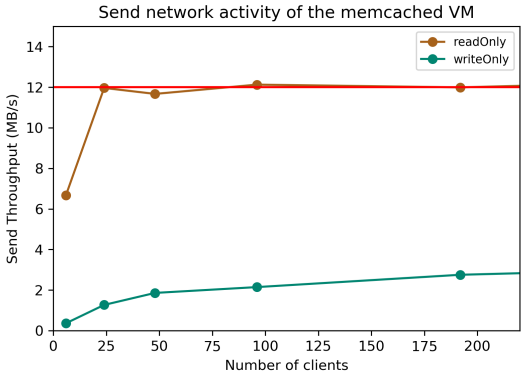


(a) Throughput against number of clients

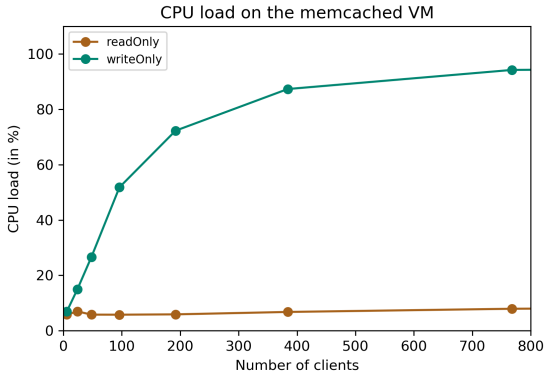


(b) Response Time against number of clients

Figure 4: Experiment 2.1: Throughput and Response Time for one memcached server



(a) Send activity on MC. Red line = max send performance



(b) CPU load on memcached.

Figure 5: Experiment 2.1: Metrics extracted from memcached using dstat

data at a much higher traffic rate than they can send. The CPU does not seem to be used heavily during reads. The send activity of the memcached server quickly reaches a plateau during reads right at the red line. Thus I suspect the sending capacity of the memcached VMs to limit read performance here. Figures 5a and 5b support this.

The response time and throughput calculated with the interactive laws are plotted by the dashed lines. They are reasonably close to the gathered data, so the laws hold.

2.2 Two Servers

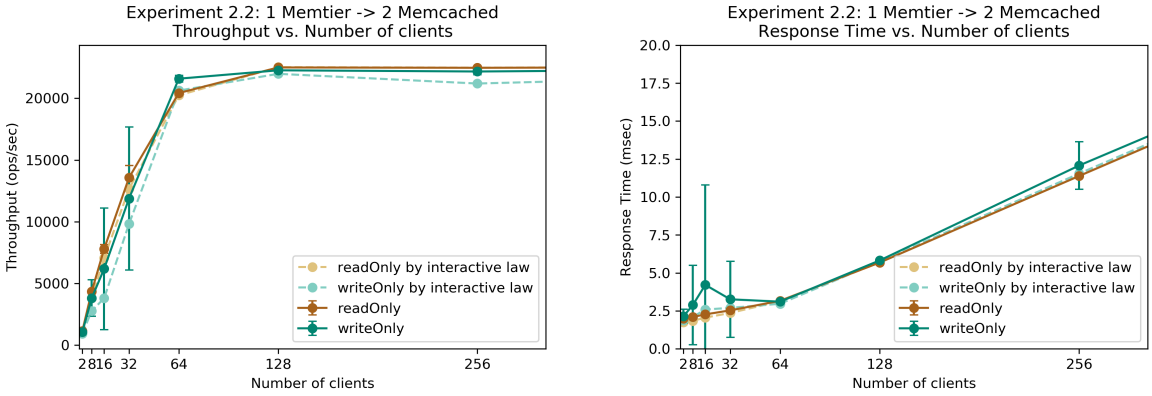
Number of servers	2
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1,4,8,16,32,64,128,256,512]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	3

This experiment setup uses a single client VM to hit two memcached instances at the same

Sending VM	Max throughput (MBit/s)	Max throughput (MB/s)
Memcached VM	101	12.63
Middleware VM	803	100.38
Memtier VM	203	25.38

Table 1: Sending capabilities of the VMs on Microsoft Azure

time. Again, a write-only and read-only workload has been tested and the number of virtual clients per memtier thread has been varied as well. The results can be seen in Figure 6.



(a) Throughput against number of clients

(b) Response Time against number of clients

Figure 6: Experiment 2.2: Throughput and Response Time for two memcached server

One immediate thing that is noticeable are the wide confidence intervals for the write-only workload around the 16 clients runs. I attribute this outlier to network performance fluctuations in the connection between the client VM and memcached server 2. The average pings to server 2 for rep 2 and 3 is `1.762` and `1.199` ms. For rep 3 it is `3.988` ms. The dstat logs do not show any difference in CPU or disk load between the repetitions.

In comparison to the single server experiment in Section 2.1, the performance differences between the write and read workload vanished and saturation is reached at 64 clients. Network statistics on the client show that for writes, the sending limit determined with `iperf` has been reached with that setting. Analogously, reads are also limited by the combined sending limit of the two memcached VMs.[6]

As can be seen from the figures 6, the calculated throughput and response time values using the interactive laws are a good estimation, hence the laws hold.

2.3 Summary

Maximum throughput of different VMs.

	Read-only workload (op/s)	Write-only workload (op/s)
One memcached server	11132 (at 24 clients)	46532 (at 384 clients)
One load generating VM	22502 (at 128 clients)	22270 (at 128 clients)

On experiment 2.1 we tried pushing a single memcached instance to the limit. The big performance gap between read and write only workloads probably stems from network limits.

[6]Graphs have been omitted for conciseness reasons, please check `{server,client}_dstat_logs` Jupyter notebooks

Experiment 2.2 on the other hand tried testing the limits of a single memtier instance. Both read and write only workloads show very similar performance numbers.

When comparing the number of the two experiments, one can observe that for 2.2, the maximum read-only throughput is roughly twice the number from 2.1 as we have doubled the number of memcached servers. For write-only we now get similar throughput numbers as for read-only. To me, this is only a conincidence. The limit lies within the network, but for different reasons. For read-only from the memcached VMs sending capabilities, for write-only from the memtier VM ones. Remember, in experiment 2.1 we utilized three client VMs and subsequently identified the CPU of the memcached server to be the bottleneck for writes. This bottleneck has changed for 2.2.

Finally, the key take-away message from these experiments is that when evaluating the performance of the middleware, one needs to keep in mind the performance limits of the clients and servers. Sufficient client and memcached resources should be provisioned when executing experiments to actually see interesting behaviour in the middleware and not be limited by memcached or memtier.

3 Baseline with Middleware (90 pts)

In the next experiments we connect one client VM to one or two middlewares, both connected to the same memcached server. This should give us a first indication of the performance effects of our middleware.

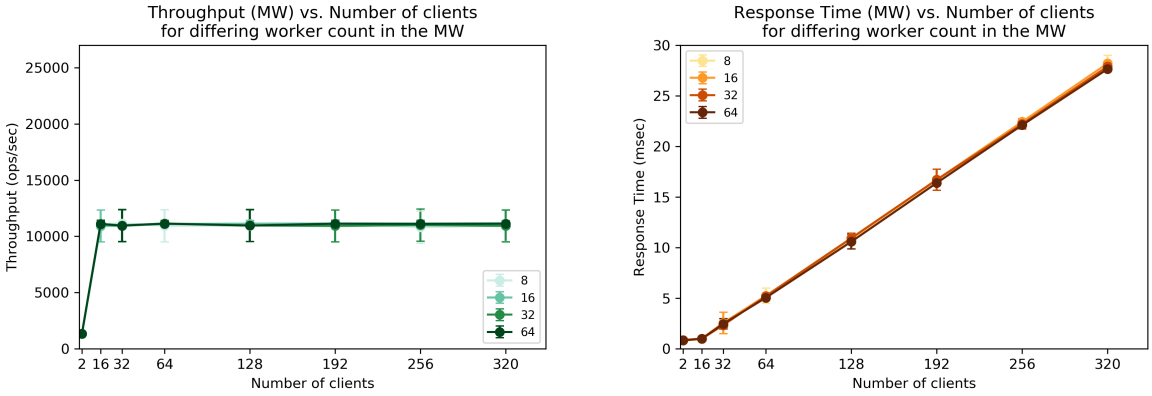
3.1 One Middleware

Number of servers	1
Number of client machines	1
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[1,8,16,32,64,96,128,160]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1
Worker threads per middleware	[8,16,32,64]
Repetitions	3

Here we connect one memtier VM to a single middleware using a single memcached server. We measure the throughput and response times both in memtier and on the middlewares. Section 2 already explained how the metrics are gathered on memtier. For the middleware statistics, we try to keep things consistent. Hence we first merge the request logs from all the threads in the middleware. The throughput is then estimated by counting the number of logged requests within 1-second windows and multiplying the count by 20, because we only log 5% of the requests. The 1-second windows are then combined in a similar manner as for memtier. The response times are averaged analogously. To aggregate metrics over multiple middlewares, we again sum up the throughput of the individual middlewares and use a weighted average for the response times. This is to make the metrics for the middlewares and memtiers comparable. The results can be seen below.

For read-only workload, lets take a look at Figure 7. One can clearly observe that saturation is already reached at 16 clients as the response time starts climbing when more than 16 clients are used. The number of worker threads does not have any influence on the performance.

Figure 8 shows a detailed breakdown of the response times of selected configurations. While neither the network thread service time nor the worker pre- and postprocessing times are practically relevant, I want to highlight the memcached round trip time (RTT) and average queue



(a) Throughput against number of clients **(b)** Response Time against number of clients

Figure 7: Experiment 3.1: Throughput and Response Time for a read workload with one middleware as measured on the middleware

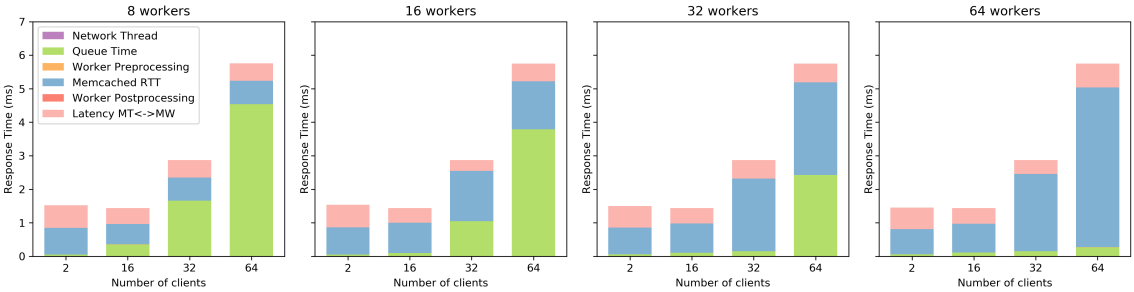


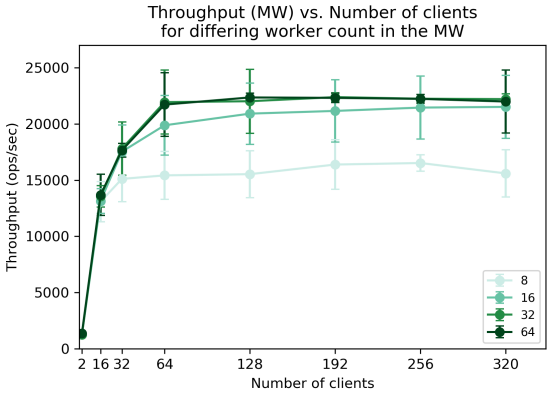
Figure 8: Response time break-down for different read-only configurations for one middleware

time. The former consists of the time it takes to send out the request to the memcached server and wait for the response. The latter is the time the request spent in the queue before being processed by one of the workers. For low worker counts, the arrival rate greatly exceeds the service rate, which leads to congestion in the queue. For a higher worker count, more jobs can be serviced in parallel, but a different bottleneck becomes apparent. Just as in Section 2.1, I suspect the sending capabilities of the memcached VM to be the limiting factor for the read-only performance. The increase of memcached RTT with higher client counts supports this claim.

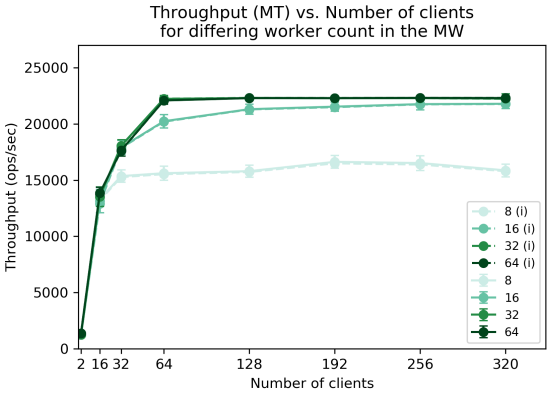
For write-only workload, the situation looks different. Figure 9 shows that saturation can be achieved at around 64 clients, while the configuration with 8 worker threads levels off at a lower throughput than the remaining three configurations. As the 32 and 64 worker levels lie very close together, I assume that their reason for saturation is different from the 8 and 16 worker settings.

An additional thing to notice is the higher standard deviation for the middleware plots than for the memtier ones. I attribute this to the request sampling, which can only estimate the real throughput in expectation. Memtier will always paint a more precise picture in this regard, but nevertheless, comparing Figures 9a and 9b show that the request sampling leads to a very good approximation of the average throughput. The throughput calculated by the response times using the interactive law is plotted in the latter graph as a dashed line (see "8 (i)" for example). The dashed lines are almost not visible as they overlap with the measured throughputs very well.

Comparing the measurements of memtier and the middleware, the diverging response times in Figure 10 immediately stand out, which requires an enormous think time for the interactive

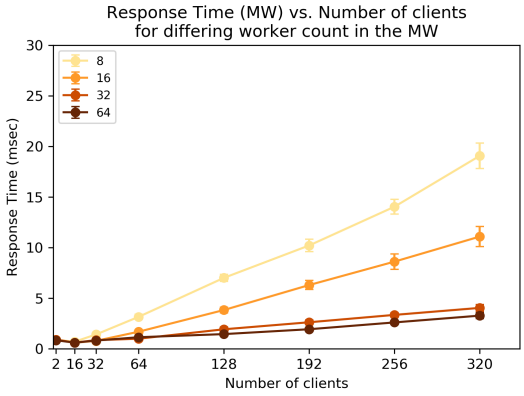


(a) Throughput measured on the middleware

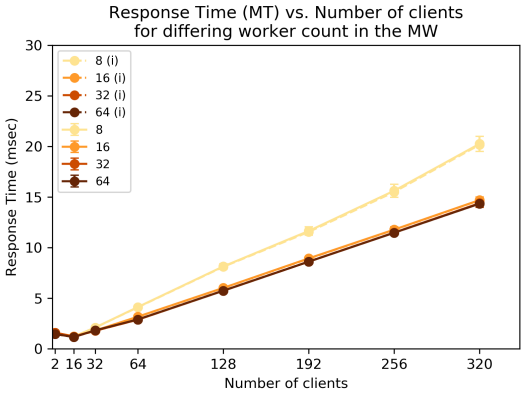


(b) Throughput measured on memtier

Figure 9: Experiment 3.1: Throughput for a write workload with one middleware



(a) Response time measured on the middleware



(b) Response time measured on memtier

Figure 10: Experiment 3.1: Response time for a write workload with one middleware

response time law to hold. In our case, the think time Z does not only include the real think time of memtier, but also the network latency. Independently from that, three possible reasons have been investigated on how to explain these differences:

Delay due to log4j2 write-out: During the experiments, on average only every 20th request will be logged at random. For each thread, the logged requests will be written to a separate file. The time measured to log the request was not captured in the initial setup. The assumption was that expensive disk I/O led to high response times which was overlooked by this sampling method.

An experiment has been conducted where different kinds of log-configuration have been compared[7]. Runs of logging every 20th, 10th and 5th request have been recorded, hoping that if more requests are logged, the response time measured on memtier will rise and the divergence will become even bigger. Indeed more data is written to disk when logging more, which can be seen in dstat excerps, but the response time and throughput levels stayed constant.

Additionally, another experiment has been conducted where the cost of writing the log to

[7]for plots see logs `exp3/logging_effect_test.tar.gz`

disk has been measured[8] on a smaller scale (for a subset of the total configurations). For a majority of requests, the time of logging took less than 0.1ms.

This means that the log write-out cannot explain the response time divergence we observed.

Break-up of request in multiple parts: For the experiments I measured the arrival time as the time when a full and valid message has been received completely. Using the non-blocking I/O with the `Selector` interface could lead to requests not arriving completely within one iteration of the `Selector.select()` method. Hence the aforementioned additional experiment[9] also logged the first contact time of a request at the middleware and how often the network thread had to read from the non-blocking SocketChannels to receive the full message. All messages, even the larger SET requests, were received in a single read from the SocketChannels and the time between the first contact and the arrival of the full message is minuscule.

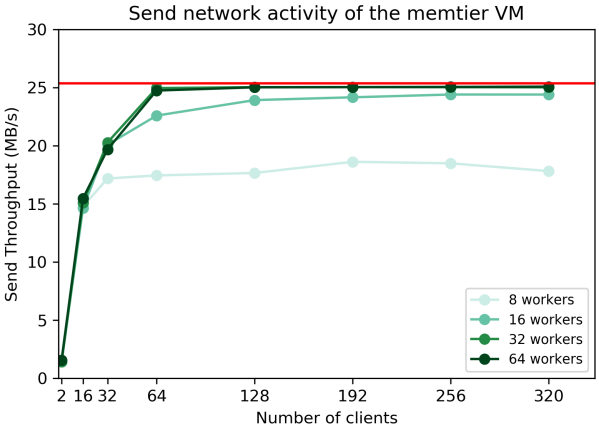


Figure 11: Sending activity on memtier.
Red line = max send performance

Network throughput between MT and MW: The dstat logs of the client VM showed that for worker counts of 32 and 64 and more than 64 clients the network send activity was constantly around 25 MB/s (see Figure 11). We have measured the maximum sending throughput of the VMs in Table 1. The same logs show that the CPU utilization converges between 25 and 40 %, it is highly unlikely that the real memtier thinktime grows that much. (The response times on the middleware do not show a limiting behaviour as seen in 10a).

These tests led to the conclusion that the network sending capabilities of the memtier VM must be the bottleneck in this setup. Figure 12 also shows this in the 2nd, 3rd and 4th subplot, where the total MT response time for high client counts is much larger than the MW response time. (The MT response time can be read from the plots as the total height of the stacked bars. The MW response time includes all measured times excluding the "Latency MT¡-¿MW" colored in light-red.)

For 8 workers, the bottleneck lies in the number of worker threads available. Subplots 1 and 2 of Figure 12 show this through the queue time. With increasing clients, the queue time

[8] see logs `exp3/3_1_log_writeOutTime.tar.gz`
[9] see logs `exp3/3_1_log_writeOutTime.tar.gz`

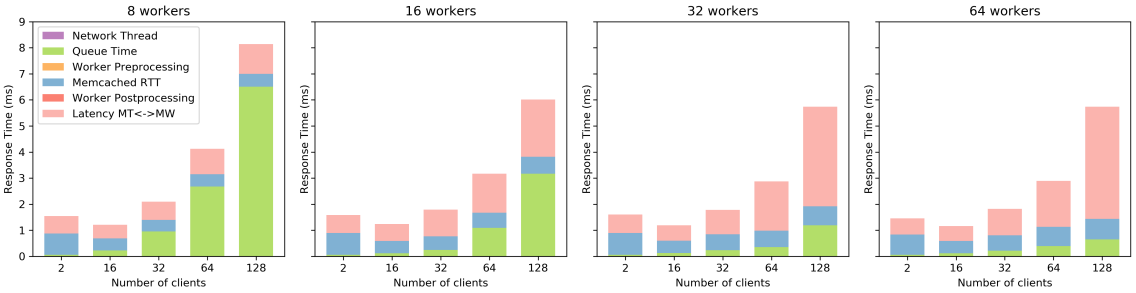
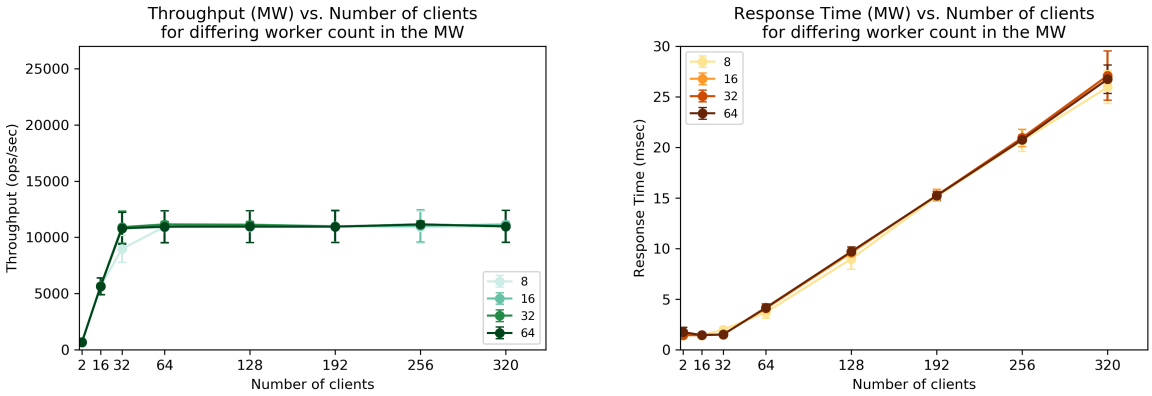


Figure 12: Response time break-down for different write-only configurations for one middleware

rises, while the memcached RTT stays the same. Looking at subplot 2 and also Figure 9, the 16 worker settings bottleneck also seems to be the worker count, as the queue time makes up the biggest part of the total middleware response time.

3.2 Two Middlewares

Number of servers	1
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1,8,16,32,64,96,128,160]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8,16,32,64]
Repetitions	3

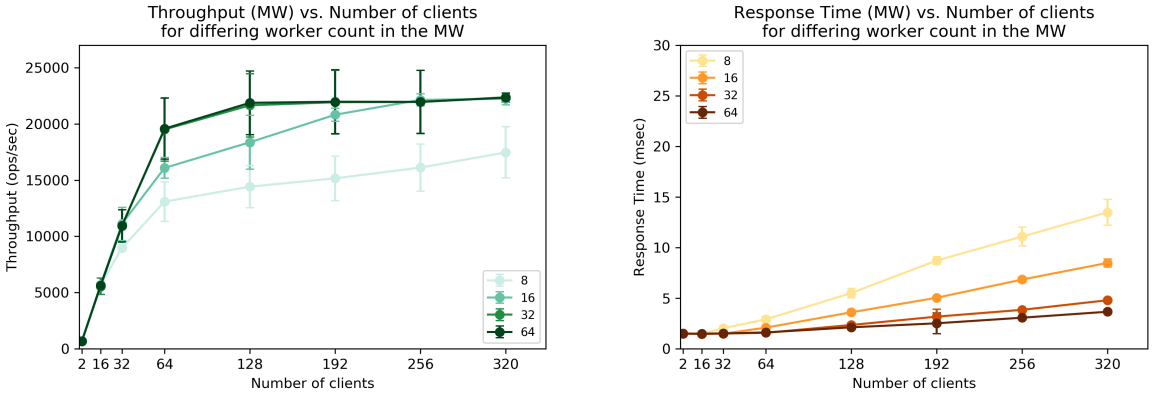


(a) Throughput against number of clients **(b)** Response Time against number of clients

Figure 13: Experiment 3.2: Throughput and Response Time for a read workload with two middlewares as measured on the middlewares

In this experiment we double the number of middlewares to two VMs, keeping the client and memcached VMs at one. The throughput and response times of the read-only workload can be seen in Figure 13. They show a behaviour nearly identical to the one middleware case above. Here I suspect the same bottleneck as in Section 3.1, namely the sending capabilities of the memcached VM.

The throughput of write-only workload can be seen in Figure 14a. Comparing it with 9a, the aforementioned and assumed network limit for the 32 and 64 worker settings becomes apparent



(a) Throughput against number of clients **(b)** Response Time against number of clients

Figure 14: Experiment 3.2: Throughput and Response Time for a write workload with two middlewares as measured on the middlewares

again. The 8 and 16 worker throughputs rise a little higher and slower, as their bottlenecks have been alleviated by adding another middleware, effectively doubling the amount of worker threads available. One can observe this as well when comparing Figure 15 with Figure 12, where the average queue time between the two middlewares for the 8 worker setting is now lower. One interesting thing to notice is the increase of memcached RTT when adding another middleware. Also, we cannot observe an improvement in response time, as any reduction in queue time is offset by an increase in latency by the aforementioned network bottleneck.

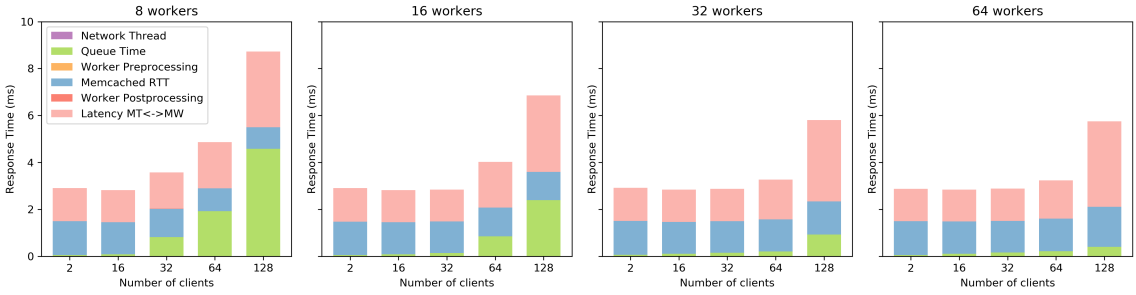


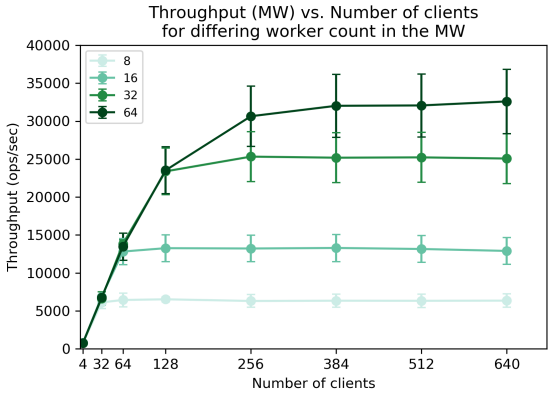
Figure 15: Average measured times within the system for different write-only configurations for two middlewares

3.2.1 Re-run with two memtier VMs

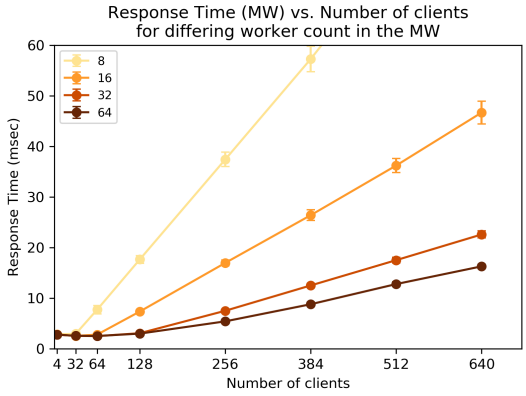
In Section 3.2 the bottleneck in the highest worker setting (64) cannot be attributed to the middleware. Hence we re-run experiment 3.2 using two memtier VMs and analyze the changes.

Saturation for write-only now occures at different levels for the different worker settings (see Figure 16a) and the gap between the response times measured on MT and MW has vanished. (the Middleware response times can be seen in 16b, the plots for memtier have been omitted).

The saturation points have been identified at 32 clients for 8 workers, 64 clients for 16 workers, 128 clients for 32 workers and 384 clients for 64 workers. The average queue length development (Figure 17) after the saturation points when increasing the client count even further show that saturation directly correlates with a linear increase in queue length. Plotting the response time break-down at those configurations (Figure 18), one can see that - again - at



(a) Throughput against number of clients



(b) Response Time against number of clients

Figure 16: Experiment 3.2 re-run: Throughput and Response Time for a write workload with two middlewares and two memtier VMs as measured on the middlewares

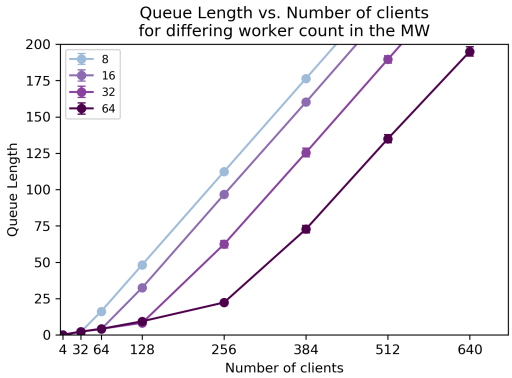


Figure 17: Experiment 3.2 re-run: Average queue lengths for the write-only workload

lower worker counts, the queue time increases and hence this poses the bottleneck. For the 32 and 64 worker settings and client counts beyond the saturation point, the linear increase in queue lengths suggests a similar bottleneck. (The response time break-down also shows this, but has been ommited for client counts larger than 128.)

The re-run did not show any changing behaviour for the read-only workload.

3.3 Summary

Maximum throughput for one middleware.

	Throughput (op/s)	Response time (ms)	Average queue time (ms)	Miss rate (%)
Reads: Measured on middleware	11092	0.98	0.11	0.0
Reads: Measured on clients	11090	1.44	n/a	0.0
Writes: Measured on middleware	21721	1.14	0.39	n/a
Writes: Measured on clients	22090	2.90	n/a	n/a

Maximum throughput for two middlewares.

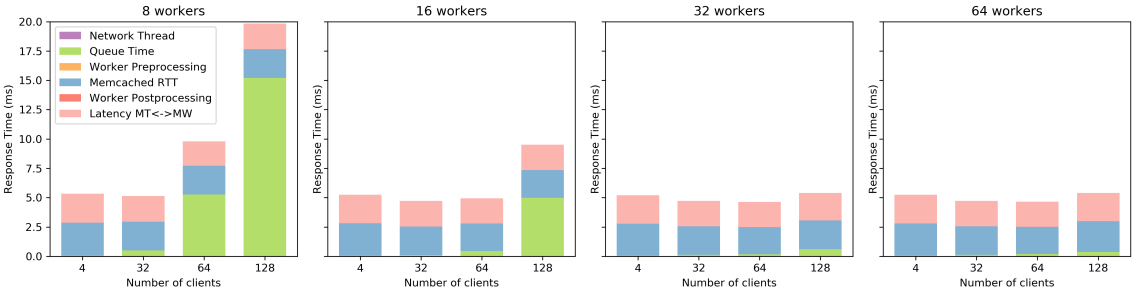


Figure 18: Experiment 3.2 re-run: Response time break-down for different client configurations

	Throughput (op/s)	Response time (ms)	Average queue time (ms)	Miss rate (%)
Reads: Measured on middleware	11136	3.54	0.26	0.0
Reads: Measured on clients	10997	5.74	n/a	0.0
Writes: Measured on middleware	32019	8.81	4.91	n/a
Writes: Measured on clients	32342	11.82	n/a	n/a

Note: The data displayed in the second table with two middlewares is taken from the re-run, not the original network-limited experiment.

Also note: These numbers show the maximum of any repetition, and not the averages reported in the plots above.

Comparing the read-only numbers and taking into account the experiment without middleware, one can clearly see that the middleware is not the bottleneck here. The rise in response time for two middlewares likely stems from the latency between MTs and the MWs being 4-5x higher for the experiments with two middlewares than for the experiments with one middleware. We have shown that saturation has been reached around 16-32 clients. For the following experiments, I expect more memcached servers to alleviate the bottleneck.

Concerning the write-only workload, we have shown that the middleware is not the limiting factor with one memtier VM. As expected, adding another middleware does not change this. Ramping up another memtier VM on the other hand has shown to increase performance by 50 %. The maximum throughput has been reached with 384 clients and 64 workers. The response times for the maximum throughput configuration with two middlewares shows elevated levels. Section 2.1 showed that a single memcached servers peak write throughput occurs around 384 clients and we can also see this in the increased response time. As the middleware now acts as a request buffer for memtier, this also explains the queue time rising to 4.91, even for 64 workers.

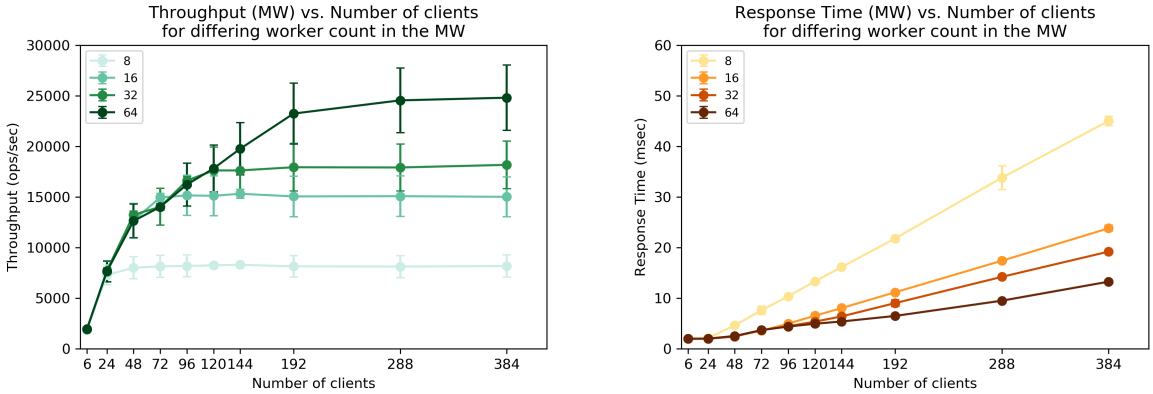
The differences between throughput measured on the clients and on the middlewares likely stems from the request sampling, which is only an estimator of the maximum throughput. Nevertheless, the numbers are reasonably close together.

Finally the interactive response time law holds, which we would like to show with the example of the write-only throughput and response times measured on the clients. With the maximum throughput occuring at 384 clients and the average response time being measured on memtier being `11.82ms`, the interactive law dictates a throughput of 32487 op/s, which is very close to the measured throughput of 32342 op/s.

4 Throughput for Writes (90 pts)

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1,4,8,12,16,20,24,32,48,64]
Workload	Write-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8,16,32,64]
Repetitions	3

In this experiment we connect three load generating VMs to two middlewares and three memchached servers, trying to see the performance implications of replicating SETS to multiple servers. The throughput and response time results can be seen in Figure 19.



(a) Throughput against number of clients. **(b)** Response Time against number of clients

Figure 19: Experiment 4: Throughput and response times for a write workload, two middlewares and three memcached servers as measured on the middlewares.

The throughput plotted in Figure 19a clearly shows the saturation points of the four worker settings. Saturation occurs on different levels so the worker settings have an effect on performance. In general one would expect the throughput of SETs with three memcached servers on average to be lower than with one server, as the requests have to be forwarded to all three instead of only one server. (A direct comparison with experiments in section 3.2.1 is impossible, as the VMs have been restarted between these runs.)

When examining the memcached round-trip time (RTT) for the different configurations (Figure 20) with the throughput plot, a change in regime becomes visible. For the 8 and 16 worker setting, the round trip time for memcached stays low, while the throughput doubles. This is a direct effect of doubling the worker counts. So for these settings, I see the bottleneck in the number of workers available.

The 32 and 64 worker settings do not show this behaviour, the throughput does not double. Even more interestingly, the round-trip time of the memcached servers increases considerably. I see this as a sign of memcached being hit by too many concurrent connections. Thus I identify the bottleneck for these worker settings either on the network or on the memcached side.

4.1 Summary

Maximum throughput for the full system

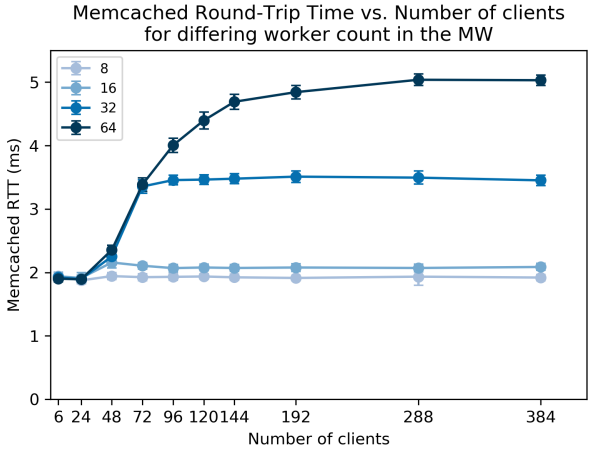


Figure 20: Experiment 4: Average memcached round-trip time for SET requests on the three memcached servers

	WT=8	WT=16	WT=32	WT=64
Throughput (Middleware) (op/s)	8014	15158	17627	23246
Throughput (from MW resp. time) (op/s)	7830	14917	17226	23535
Throughput (Client) (op/s)	8064	15135	17799	23359
Average time in queue (ms)	2.67	2.89	1.90	1.62
Average length of queue	8.99	19.07	9.92	13.03
Average time waiting for memcached (ms)	1.94	2.07	3.44	4.85
Number of clients:	48	96	120	192

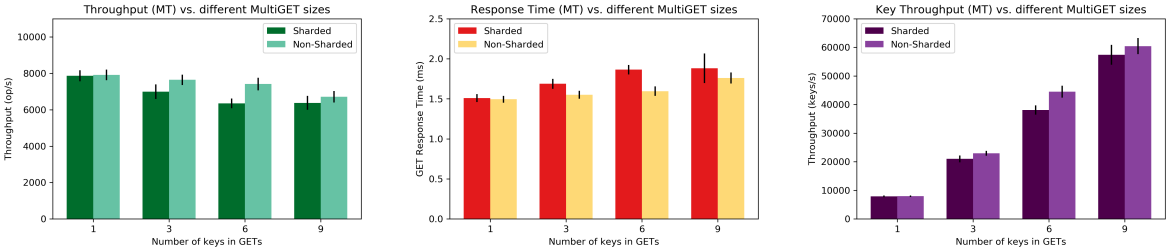
As already analyzed in the previous section, for low worker counts, the increase in workers available directly translates to increases of throughput. The average memcached waiting time stays on similar levels. Further increases of the worker count alleviates this bottleneck, but the memached servers then become the bottleneck. For the table, not the absolute maximum throughput has been chosen, but a reasonable level where the response time doesn't rise too much. This has been decided from the plots in Figure 19. To arrive at the throughput numberes derived from the middleware response time, the interactive law has been applied, assuming a client think time equal to the average ping round-trip-time measured between the clients and the middlewares.

5 Gets and Multi-gets (90 pts)

For the next set of experiments we use three load generating machines, two middlewares and three memcached servers, testing the effects of different sizes for MultiGETs and also the sharded-mode. 64 workers have been used throughout the experiments. The workload ratio in memtier has been set to `1:N`, which means the requests were split evenly between SETs and GETs, while GETs always contained N keys. The analysis in the following subsections focuses now on the GET requests. The SET requests have been omitted, but they generally behaved as expected.

5.1 Sharded Case

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	2
Workload	memtier-default
Multi-Get behavior	Sharded
Multi-Get size	[1,3,6,9]
Number of middlewares	2
Worker threads per middleware	64
Repetitions	3



(a) Throughput for different MultiGET sizes.

(b) Response Time for different MultiGET sizes.

(c) Key throughput for different MultiGET sizes.

Figure 21: Experiment 5: Throughput, Response times and key throughput for different MultiGET sizes as measured on memtier.

Figure 21a shows the overall throughput of the experiments. These number also contain SET requests. The response times of the GET requests only can be seen in Figure 21b. With sharding turned on and with increasing numbers of keys in a single request, the response time naturally rises. Interestingly, there is practically no increase when moving from six keys to nine keys per request.

While the overall throughput drops as expected when the size of the MultiGETs is increased, the number of keys transferred in the same timeframe rises significantly. In Figure 21c we can see that the effective throughput of keys per second sharply rises with increasing MultiGET sizes. Hence we can conclude that processing a single request costs us a huge overhead and this overhead effect can be diminished by using MultiGETs instead of multiple single-GETs.

An analysis of the percentiles in Figure 22 shows that 90 % of the sharded GET request are processed faster than 3.5ms. The 99th percentiles show quite elevated levels. One has to keep in mind though that the reason for this may lie in the design of the middleware. As described in Section 1, each thread in the thread pool contains its own set of connection to all memcached servers (see the `MemcachedSocketHandlers` class). These connections are lazily initialized. That means the sockets are not established during startup of the middleware, but when the connections are needed for the first time. This means that the first request that a thread processes will incur a higher worker pre-processing time than subsequent requests.

One counter-intuitive observation is that sharded requests with 6 keys are slower than the requests with 9 keys. This is something that needs to be investigated.

Through sharding, each server will be hit with one key in the 3-key-multiget case, with two keys in the 6- and with three keys in the 9-key-multiget case. That means that for the sharding experiments with these keysizes, all three memcached servers are involved. The single-key GETs only concern a single memcached server.

In Figure 23 we can see the memcached round-trip-time, which is the time it takes the worker thread from sending out the first of the sharded requests until all three responses have

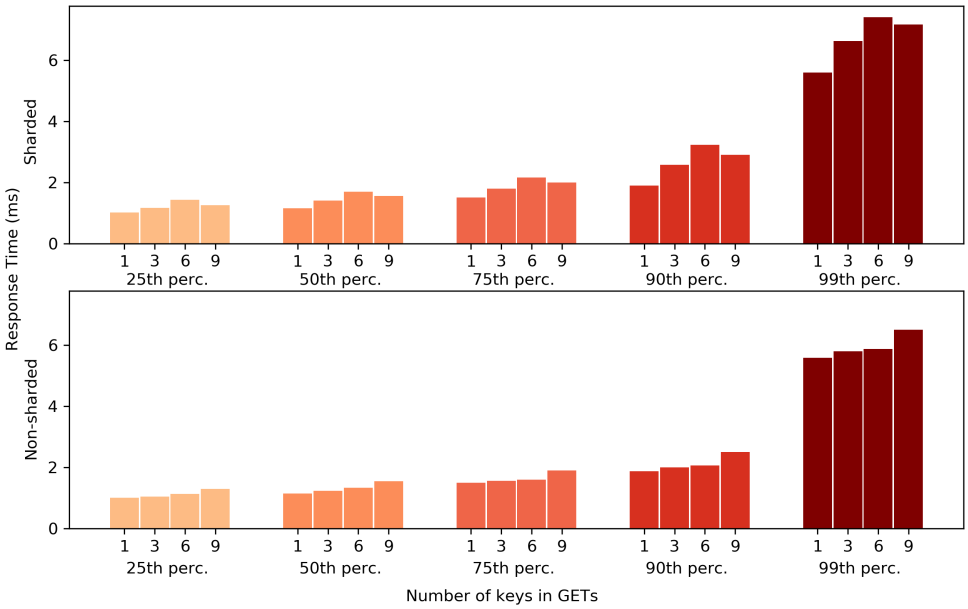


Figure 22: Response time percentiles for the sharded and non-sharded configuration as measured on memtier.

been received. The first column shows the combined memcached round-trip-time. For three out of the four multiget configurations, two clear local maxima become apparent. The plots to the right show the memcached RTT as measured on the individual middlewares. I assume that the three memcached servers treats both of the middlewares fairly. Hence I attribute the difference between the RTTs in column two and column three to the network latency. Middleware two simply has a shorter latency to the memcached servers. While the experiments with keysizes of 1, 3 and 9 clearly show bell curves whose expectation lies between 0.5 and 0.7ms, the 6-key runs show a histogram that is shifted to the right. The maximum not only lies at 0.8ms, the RTTs are also further spread out. For the combined middleware histogram, the two peaks that are so prominently visible in the other plots are completely indistinguishable here.

Note: The histograms in this section have been plotted with the normed-option of matplotlib, which works similar to a density function. We are mainly interested in the shape of the histogram, hence it is safe to ignore the y-axis in the histograms.

The plot above uses data over all repetitions. A further analysis has been conducted whether this behaviour of the sharded 6-key GETs was due to an outlier in the repetitions or whether the repetitions all consistently showed this behaviour. Figure 24 show these results.

The three repetitions are overlaid on top of each other. The histograms areas match almost precisely for all the three experiments and hence there was not much variation in the memcached RTTs between them. Especially there was no outlier, so the deviation was consistent.

Finally, part of the experiment has been re-run a couple of days later to check whether this anomaly also occurs across VM restarts. The results in Figure 25 show no abnormal behaviour anymore. The histograms between the 6- and the 9-key cases have similar shapes and similar response times. From the observations gathered above I conclude that this abnomaly probably not stems from the middleware nor memtier, but from either the network between the middleware and memcached or the memcached servers themselves.

Regarding the bottlenecks, we have seen that the memcached RTT dominates the overall

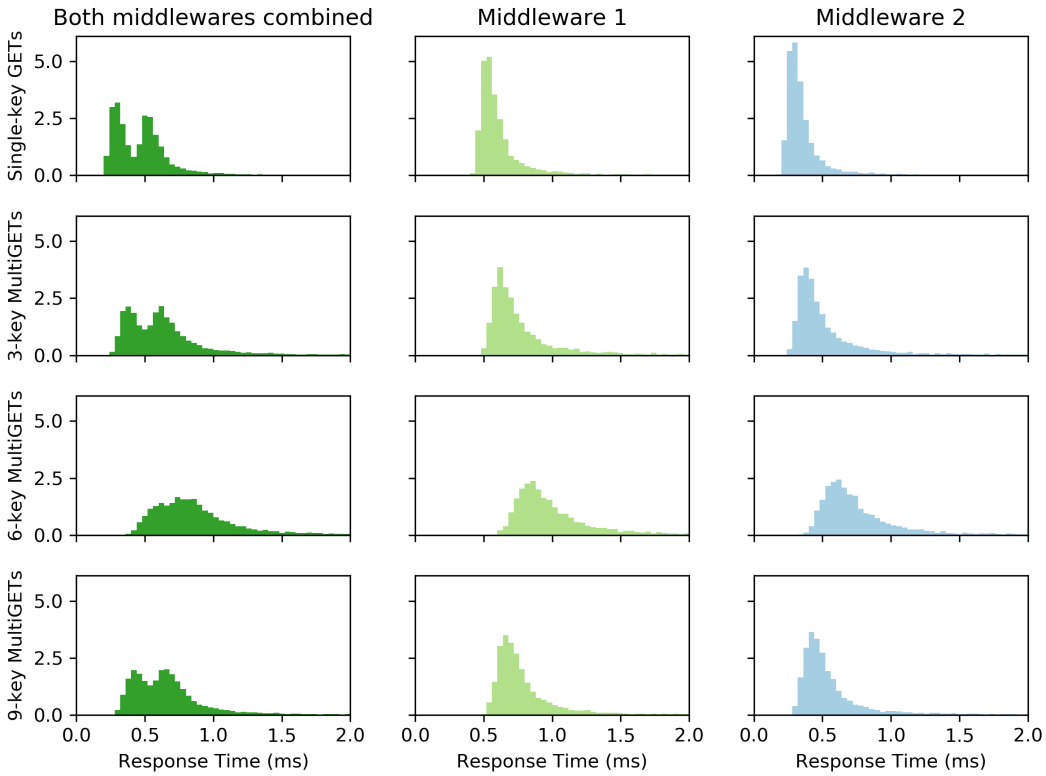


Figure 23: Memcached round-trip time histograms of the sharded requests for 1-, 3-, 6- and 9-key MultiGETs. The first columns shows memcached RTTs of the two middlewares combined, the second and third columns from the individual middlewares alone.

response time. Reducing this waiting time by any ways would see immediate improvements in throughput.

5.2 Non-sharded Case

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	2
Workload	memtier-default
Multi-Get behavior	Non-Sharded
Multi-Get size	[1,3,6,9]
Number of middlewares	2
Worker threads per middleware	64
Repetitions	3

For the non-sharded configuration, MultiGETs are forwarded as-is to one of the memcached server. No split up is needed. The throughput and response time results as well as selected percentiles can be found in Figures 21 and 22. Just as the sharded configuration, throughput declines and response time rises with increasing MultiGET keysizes. The percentiles plot also shows reasonable performance, with more than 90 % of the requests being processed faster than 3 ms.

As we can clearly see in Figure 26, the response time of the middleware is again dominated by the memcached RTT. The average queue time is almost zero and also the worker pre- and post-processing as well as the network thread service time are negligible. The latency

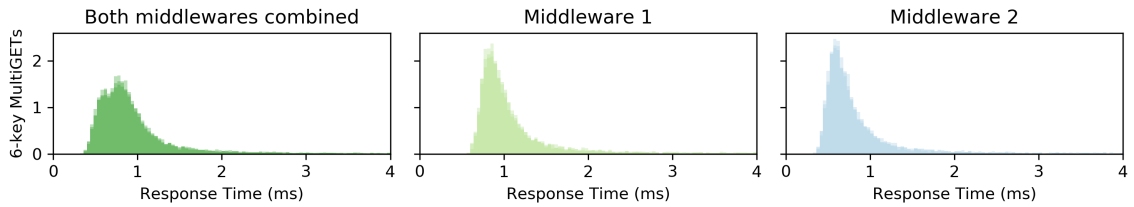


Figure 24: Memcached round-trip time histogram of the sharded requests for 6-key MultiGETs. The three repetitions are laid on top of each other.

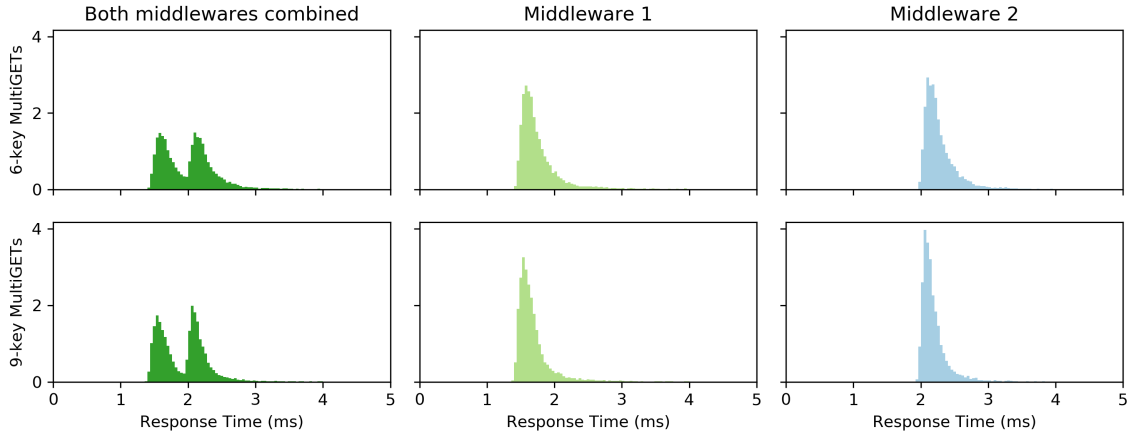


Figure 25: Memcached round-trip time histogram of the sharded requests for 6- and 9-key MultiGETs. Re-run of the experiment above.

between memtier and the middleware makes up about 0.75ms. The average CPU load on the memcached servers rose only by 13 % when moving from single GETs to 9-key-MultiGETs, while the combined network output rose from 3.5 MB/s to 22.6 MB/s, which is an increase of 545 %. Keeping in mind that number of keys processed increased 9-fold, these numbers tell us that MultiGETs are a much more efficient way of requesting keys from memcached. This also supports the observations we did in Figure 21c. Finally, a small increase in the memcached RTT can be seen for larger MultiGETs. Unfortunately it is not clear whether this stems from increased latency due to a better saturated network link or memcached having to fetch more keys for a single request. Nevertheless for the above reasons I identify the bottleneck on the memcached servers side.

5.3 Histogram and Summary

As discussed in Section 5.1, the response time histogram of the 6-key sharded experiment looked unusual. Repeating this part of the experiments gave results that were in line with the other sharded runs. However, to ensure comparability between the non-sharded and sharded modes, the original experiment data has been used for the histograms.

The histogram of sharded and non-sharded responsetimes measured at memtier and the middlewares for the 6-key MultiGET runs has been plotted in 27. The number of buckets is 40. To keep the histogram useful, only response times between 0 and 4 ms have been plotted. The response times extracted from the middlewares have been plotted directly, for memtier an interpolation step has been added to estimate the number of requests at response times where the memtier logs do not provide that number. This was also necessary for merging of multiple

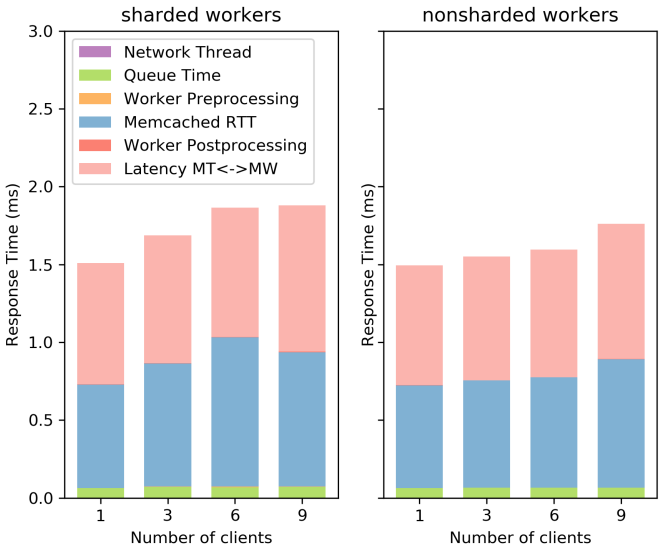


Figure 26: Response Time breakdown of the MultiGET requests for both sharded and non-sharded mode.

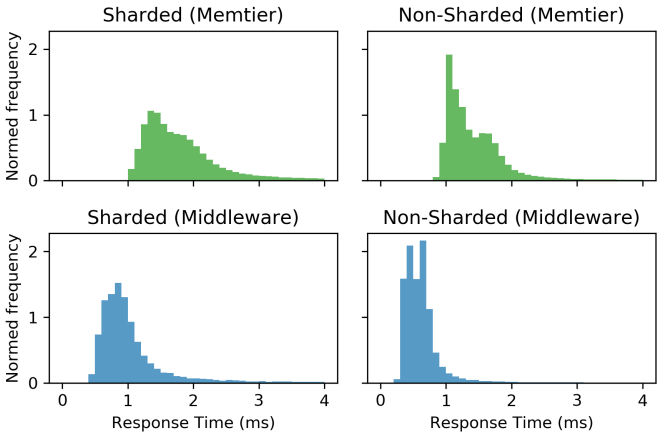


Figure 27: Response Time histograms for 6-key MultiGETs.Averaged over memtier VMs / middlewares.

client logs.

From the histograms we would expect to see a bell curve with a hard threshold on the left side. This threshold would be absolute minimum round-trip-time of a request.

Immediately one notices the shift between the memtier and middleware response times. This gap of about 1ms on average can likely be attributed to the network round-trip-time. Of course the three clients have different round-trip-times to the middlewares. This is the reason why the shapes of the memtier histograms do not match the curves of the middlewares. Breaking this down into the contributions of the individual clients with overlapping histograms makes this apparent (see Figure 28). While all clients exhibit a very similar round-trip-time to middleware 2, client 2 seems to have a RTT to middleware 1 that is about 1ms faster than the other two clients.

Furthermore, the non-sharded middleware histogram shows two peaks, where the sharded histogram shows only one. A break-down of the combined histogram into the two middlewares in Figure 29 explains the two local maxima. It seems that in both experiments, middleware

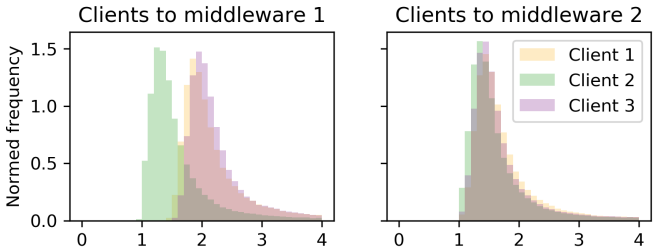


Figure 28: Break-down of the response time histogram for the individual memtier clients in the sharded 6-key MultiGET case.

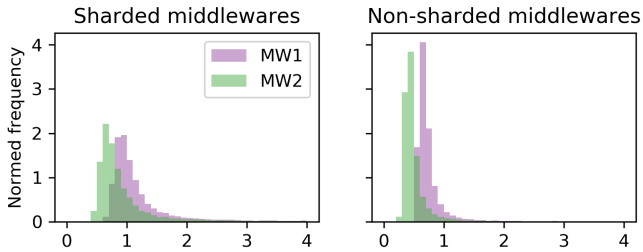


Figure 29: Break-down of the response time histogram for different middlewares in the sharded 6-key MultiGET case.

2 could process requests faster than middleware 1. I attribute this to middleware 2 having a lower latency to the memcached servers, its memcached RTT is between `0.26` and `0.30ms` lower than for middleware 1. For the sharded case, the standard deviation of the response times simply is higher, which means that the two modes are mixed together to what appears to be a single curve.

I believe that the reason for the higher spread of response times for sharded requests lies in the fact that more servers are involved in a single request. Even slight delays in the response time of a single memcached server can slow down the overall processing time. These random fluctuations are what can be seen here. This is the very same reason why SET requests for three servers on average take longer than SET requests involving a single server.

Additionally, the worker thread has to do a lot more work for the sharded requests than for the non-sharded ones. In the latter case, the worker uses the hashing method explained in Section 1.4 to figure out which server is responsible for the whole request and then immediately forwards the request to the designated server. For postprocessing, the worker simply forwards the memcached response to the client. This keeps computation time in the worker at a bare minimum. Sharded requests on the other hand need much more pre- and post-processing. After dequeueing, we determine the primary server using hashing, need to split up the request and finally construct separate GET requests for every memcached server. During post-processing the worker needs to extract, take care of the order of keys requested and then construct a final answer for the client. This extra work lengthens the pre- and post-processing time by 200 % from 0.002ms to 0.006ms. Still this is very much insignificant for the overall response times.

All-in-all, the experiments suggest that the non-sharded mode is the preferred choice for the MultiGET workloads tested above. The sharded mode consistently performs worse. On a final note: Of course, we cannot make a blanket statement about configurations with more keys, maybe at some point the tipping point is reached where it is favourable to split up MultiGETs instead of hitting single memcached servers with gigantic requests.

6 2K Analysis (90 pts)

6.1 Introduction and Model Validation

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

In the next chapter we conduct a $2^k r$ experiment on our system. While keeping the client setup fixed at 192 clients, we vary the number of middlewares, number of memcached servers and number of worker threads per middleware. We want to dissect the effects of three factors, thus $k = 3$. Each experiment is repeated three times, hence $r = 3$. We decided to use the $2^k r$ analysis instead of the 2^k one to also account and see the effect of experiment errors.

Symbol	**Factor**	**Level -1**	**Level +1**
MW	Number of Middlewares	1	2
MC	Number of Memcached Servers	2	3
WT	Number of Worker Threads	8	32

Table 2: Short symbols and levels used throughout the experiments for the three factors.

The $2^k r$ analysis requires factors to be mapped to **-1** and **+1** levels. Our mapping and also the symbols for the factors can be seen in Table 2. We are interested in conducting the analysis for both average throughput and average response times as well as for three types of workload, namely a write-only workload, a read-only one and a workload where a 50:50 ratio of GET and SET requests have been issued. The metrics are gathered from the memtier VMs.

The analysis is closely based on Chapter 18 of Raj Jains book[10], which introduces both an additive as well as a multiplicative model. We want to start by checking the model assumptions on whether an additive model is suitable or whether we have to change to a multiplicative model. Raj Jain describes the ratio $\frac{y_{max}}{y_{min}}$ as a suitable first approach to check whether an additive model fits our situation. The ys are the experiment metrics used for the analysis, in our case this is the average throughput and average response time. Table 3 shows the calculated ratios.

We can see that the ratios for response times and throughput are very close together, which I attribute to the interactive response time law. With a factor of five to six, the ratios are on a reasonable level.

Next we would like to check that the model errors are independent and normally distributed around zero. We show this by plotting the predicted response versus the residuals and through

[10]Raj Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, April 1991, ISBN: 978-0-471-50336-1

Metric/Workload	**Read-Only**	**Write-Only**	**Fifty-Fifty**
Throughput	5.38	5.86	5.21
Response Time	5.39	5.85	5.24

Table 3: Ratios of y_{max}/y_{min}

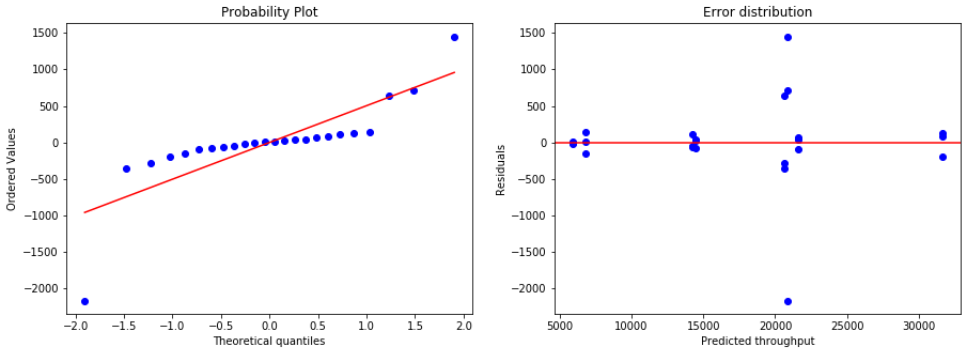


Figure 30: Distribution of the errors and residuals for the read-only experiment.

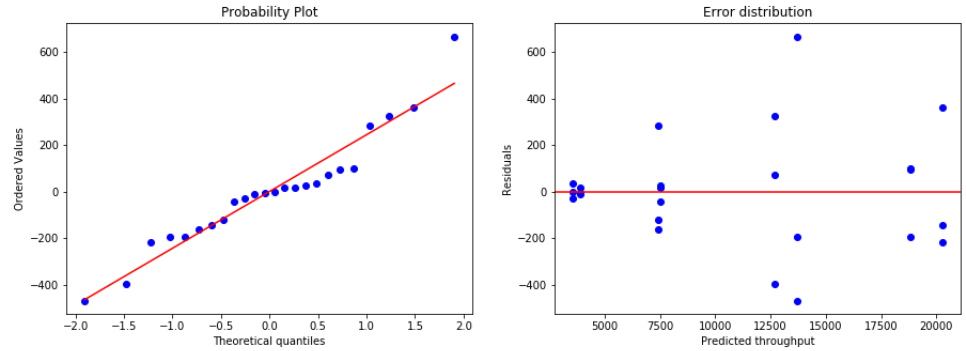


Figure 31: Distribution of the errors and residuals for the write-only experiment.

a quantile-quantile plot. For conciseness reasons, plots 30, 31 and 32 shows this only for the throughput metric. As seen on the right subplots, there doesn't seem to be apparent trend in the residuals. They are also spread out evenly around zero.

For the quantile-quantile plot we would expect a straight line for the model assumptions to hold. The write-only experiment shows a linear trend, the read-only and fifty-fifty workloads though present long tails. Hence we cannot take the additive model for the two latter experiment analysis, but a multiplicative might be better suited.

Figures 33 and 34 show the same plots for log-transformed data. The QQ-plot of the fifty-fifty experiment now shows a linear trend, which means the multiplicative model is more applicable for that workload. The read-only workload did not show a big difference after log-transforming the data.

Similar to this discussion about the throughput, the response time metric was analyzed. The graphs have been omitted. It seemed like an obvious choice to choose a multiplicative model where there is already one used for the throughput. This means that I decided to take a multiplicative model for the response times of the read-only and fifty-fifty workloads as well. Additionally, Raj Jain mentions an example[11] of measuring the time required to process some workload. I hence conclude that doing a 2^k analysis for a metric like a response time should use a multiplicative model and have thus used a multiplicative model for the write-only workload as well.

[11]Raj Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, April 1991, ISBN: 978-0-471-50336-1, page 304

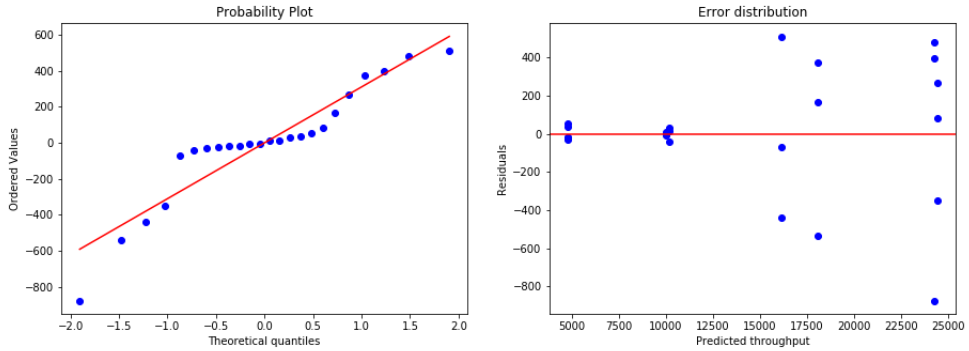


Figure 32: Distribution of the errors and residuals for the fifty-fifty experiment.

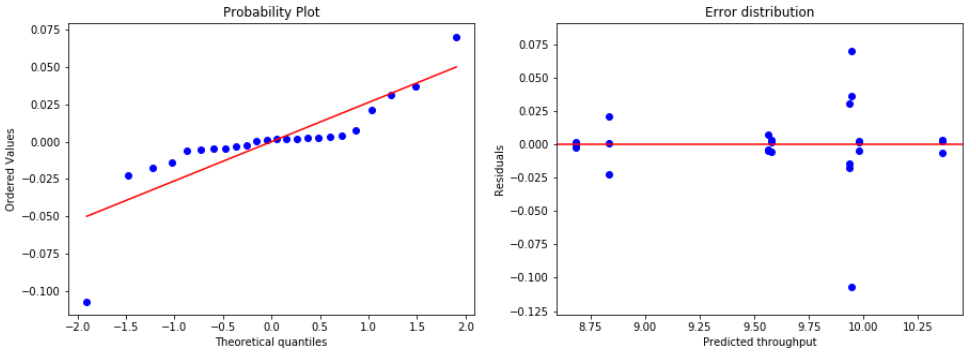


Figure 33: Distribution of the errors and residuals for the write-only experiment with log-transformed data.

6.2 Analysis of Write-Only

	Throughput			Response Time		
	Effect	**Variation (%)**	**90% conf. int.**	**Effect**	**Variation (%)**	**90% conf. int.**
I	10982.65	-	(10879.19, 11086.11)	3.03	-	(3.02, 3.04)
MW	2518.77	17.62	(2415.31, 2622.22)	-0.28	19.72	(-0.29, -0.27)
MC	-250.98	0.18	(-354.44, -147.52)	0.01	0.01	(-0.00, 0.02)
WT	5383.56	80.47	(5280.10, 5487.02)	-0.55	78.41	(-0.56, -0.54)
MWMC	-78.88	0.02	(-182.34, 24.58)	0.01	0.02	(-0.00, 0.02)
MWWT	651.37	1.18	(547.91, 754.83)	0.07	1.34	(0.06, 0.08)
MCWT	-372.22	0.39	(-475.68, -268.76)	0.04	0.36	(0.03, 0.05)
MWMCWT	-16.58	0.00	(-120.04, 86.88)	-0.01	0.05	(-0.02, -0.01)
Error	-	0.16	-	-	0.09	-

Looking at the throughput results first, WT immediately stands out as the dominating factor, explaining more than 80 % of the variation. MW comes second place with more than 17 %. Together (and including their combined effect MWWT) they explain the variation almost exclusively. Another thing to be kept in mind is that increasing the number of middlewares from 1 to 2 means that also the number of worker threads is doubled. For all three factors MW, WT and MWWT, the effect shows a clearly positive number, which means that increasing any of either MW or WT will greatly increase throughput. The mean throughput without any effects is around 11,000 op/s. This can be directly read from the table, as no log-transformation has been applied here.

The 90%-confidence intervals for the combined effects MWMC and MWMCWT include zero, which means these effects can be seen as statistically not significant. The effect of MC and MCWT both is negative. That means that if we increase the number of memcached servers,

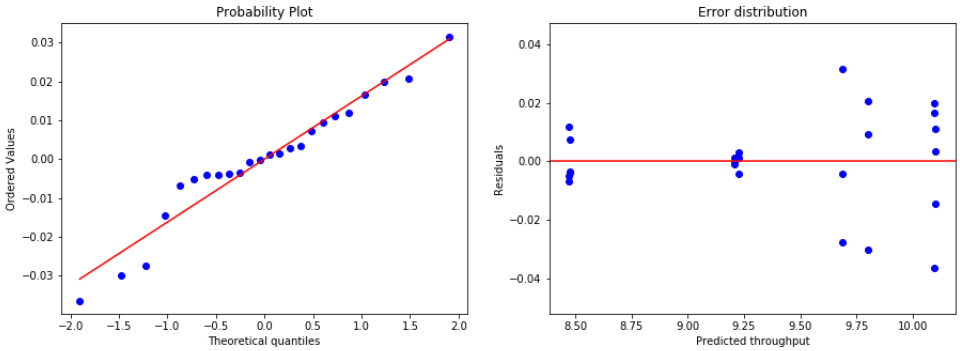


Figure 34: Distribution of the errors and residuals for the fifty-fifty experiment with log-transformed data.

throughput slightly drops. This is in line with our system, where each SET request has to be replicated to all the servers and only when all servers have replied, can we send back the answer to the client and complete this request. Because we always have to wait for the slowest server, increasing their number could slow things down.

Due to very stable experiments, the experiment error only accounts for 0.16 % of the variation. This also led to the narrow confidence intervals we see in the table.

The explanation of variation for the response time shows very similar numbers to the throughput side. The signs of the big effects, which are MW and WT again, are flipped, as an increase in these resources will reduce response times. It comes to no surprise that the variation numbers between response time and throughput are relatable, as the interactive response time law dictates.

In comparison to the throughput side, the response time effect numbers are not easily interpretable, as they have been log-transformed. Interestingly, the combined multiplicative effect of MWWT shows a positive sign on the response time as well as throughput side. For the latter it makes sense that a combination of number of middlewares and worker count has a positive effect on the throughput, but for the response time this result is counter-intuitive.

6.3 Analysis of Read-Only

	Throughput			Response Time		
	Effect	**Variation (%)**	**90% conf. int.**	**Effect**	**Variation (%)**	**90% conf. int.**
I	9.61	-	(9.60, 9.62)	2.55	-	(2.53, 2.56)
MW	0.26	22.83	(0.25, 0.27)	-0.26	23.23	(-0.28, -0.25)
MC	0.03	0.27	(0.02, 0.04)	-0.03	0.28	(-0.04, -0.02)
WT	0.45	66.31	(0.43, 0.46)	-0.45	66.02	(-0.46, -0.43)
MWMC	0.06	1.35	(0.05, 0.08)	-0.06	1.36	(-0.08, -0.05)
MWWT	-0.15	7.09	(-0.16, -0.13)	0.15	7.05	(0.13, 0.16)
MCWT	0.07	1.57	(0.06, 0.08)	-0.07	1.43	(-0.08, -0.05)
MWMCWT	0.03	0.30	(0.02, 0.04)	-0.03	0.34	(-0.05, -0.02)
Error	-	0.28	-	-	0.29	-

Again, WT dominates the varation explained by two thirds with MW taking second place. Both of these factors clearly have a positive effect on the throughput. The combined effect MWWT now explains 7 % of the variation, which is a big increase from the 1.18 % observed for the write-only workload. However, the MWWT effect is now negative. We have shortly discussed in the previous subsection that an increase in middlewares also doubles the number of available worker threads. The negative interactive effect here shows that there's a limit on how far we can increase the number of workers per middleware or the number of middlewares (and thus the number of overall workers).

As none of the 90%-confidence intervals include zero, all of them need to be deemed statistically significant. The magnitude of the varation though still shows limited effect of the remaining factors.

An interesting observation is that the number of memcached servers practically doesn't seem to have an influence on throughput. One would expect that at least a small influence can be seen as GETs are distributed evenly among all available servers. But in Section 3 we have already seen that a worker count of 8 threads per middleware combined with a high number of clients quickly leads to congestion in the queue as not enough requests can be processed in parallel. Increasing this four-fold to 32 had a huge effect on throughput. This effect can also be seen here and completely dwarfs any improvement that could be gained by distributing GETs among more memcached servers.

The response times effects are similar to the througput ones. Again, the effect signs are flipped, as a higher throughput leads to a lower response time. Finally, the experiment error explains around 0.29 % of the variation, both in throughput and response times. While this number is still reasonably low, a 2-3 fold increase can be observed when comparing with the write-only workload.

6.4 Analysis of 50:50 workload

	Throughput			Response Time		
	Effect	**Variation (%)**	**90% conf. int.**	**Effect**	**Variation (%)**	**90% conf. int.**
I	9.38	-	(9.38, 9.39)	2.77	-	(2.77, 2.78)
MW	0.28	20.12	(0.27, 0.28)	-0.28	20.58	(-0.29, -0.27)
MC	-0.02	0.08	(-0.02, -0.01)	0.02	0.09	(0.01, 0.03)
WT	0.54	77.07	(0.53, 0.55)	-0.54	76.64	(-0.54, -0.53)
MWMC	0.01	0.03	(0.00, 0.02)	-0.01	0.03	(-0.02, -0.00)
MWWT	-0.10	2.52	(-0.10, -0.09)	0.10	2.47	(0.09, 0.10)
MCWT	-0.01	0.04	(-0.02, -0.00)	0.01	0.04	(0.01, 0.02)
MWMCWT	0.02	0.07	(0.00, 0.02)	-0.02	0.07	(-0.02, -0.01)
Error	-	0.07	-	-	0.08	-

Just as for the two previous workloads, the 50:50 workloads variation is dominated by MW and WT. Looking at the the variations being explained by the WT and MW factors for the write-only and read-only workload, the equivalent values for the 50:50 workload lie in between. That makes sense, as this workload contains both GET and SET requests.

While the confidence intervals for the MC, MWMC, MCWT factors are close to zero, they do not contain it and are hence statistically significant. Nevertheless, it is safe to say that due to those low numbers, these effects do not have a great influence on the performance of our system.

Finally, the experiment error of this workload is again very low and the results for the response time metric are comparable.

7 Queuing Model (90 pts)

In this final sections we would like to build models that should help us to predict the behaviour of our system. For that we use part of the numbers gathered during the experiments and try to predict other characteristics. A comparison then shows us the suitability of these models.

7.1 M/M/1

At first we try to model our system as an M/M/1 system, an unbounded queue with a single service. We have worked with the following assumptions (just as described by Raj Jain):

- The interarrival and service times are exponentially distributed, for the duration of the experiment they are constant.
- There is a single queue and the worker threads are modeled by a single server.
- The service discipline is First Come, First Served (FCFS).
- There is no limitations in queue sizes or number of clients (Raj Jain calls this buffer or population size limits)
- The job flow balance holds, no jobs are lost in the system.

The input to our model is the mean arrival rate λ and the mean service rate μ. We can assume that the system we are testing is closed, as memtier only sends out the next request when the previous one has been processed completely. Hence the average throughput of the experiment is a good approximator of the mean arrival rate λ. For the mean service rate μ, we assume that for a fixed worker setting, the maximum throughput ever observed is a reasonable estimation of what the workers can achieve at max.

We will create a model for each of the four worker configurations tested in Section 4. The 192 client-run has been chosen as it should show different behaviours for the different worker settings. These are the configurations in detail:

Configuration 1: 192 clients / 64 workers: $\lambda = 23359$, $\mu = 25552$

Configuration 2: 192 clients / 32 workers: $\lambda = 17940$, $\mu = 18419$

Configuration 3: 192 clients / 16 workers: $\lambda = 15093$, $\mu = 15200$

Configuration 4: 192 clients / 8 workers: $\lambda = 8166$, $\mu = 8215$

7.1.1 Discussion

	Conf 1: 64 workers		**Conf 2: 32 workers**	
	Predicted	**Measured**	**Predicted**	**Measured**
Arrival Rate λ (op/s):	23359	23359	17940	17940
Service Rate μ (op/s):	25552	–	18419	–
Traffic Intensity ρ:	0.9142	–	0.9740	–
Interarrival Time τ (ms):	0.0428	0.0833	0.0557	0.1109
Mean Service Time $\mathbb{E}[s]$ (ms):	0.0391	4.8475	0.0543	3.5020
# Jobs in System $\mathbb{E}[n]$:	10.65	130.04	37.51	103.61
# Jobs in Queue $\mathbb{E}[n_q]$:	9.74	13.03	36.53	41.27
Mean Response Time $\mathbb{E}[r]$ (ms):	0.4560	6.48	2.09	9.02
Mean Waiting Time $\mathbb{E}[w]$ (ms):	0.4169	1.62	2.04	5.40

Table 4: M/M/1: Model-Measurement comparison for configurations 1 and 2.

Let us start with a comparison of the model and the measured results from configuration 1. We can see from comparing the numbers in Table 4 that the model is not a good fit. The traffic intensity ρ is < 1, hence the system is stable. The predicted interarrival time is about half of what has been measured in the middlewares. This can be explained by taking into account that we used two middlewares for the experiment, which the M/M/1 model does not suppport. The divergence in the mean service time stems from the very same reason, but also because the

	Conf 3: 16 workers		Conf 4: 8 workers	
	Predicted	**Measured**	**Predicted**	**Measured**
Arrival Rate λ (op/s):	15093	15093	8166	8166
Service Rate μ (op/s):	15200	–	8215	–
Traffic Intensity ρ:	0.9930	–	0.9941	–
Interarrival Time τ (ms):	0.0663	0.1301	0.1224	0.2393
Mean Service Time E (ms):	0.0658	2.0777	0.1217	1.9119
# Jobs in System $E[n]$:	141.06	98.11	169.28	96.55
# Jobs in Queue $E[n_q]$:	140.07	66.34	168.29	80.64
Mean Response Time $E[r]$ (ms):	9.35	11.15	20.73	21.81
Mean Waiting Time $E[w]$ (ms):	9.28	9.06	20.61	19.88

Table 5: M/M/1: Model-Measurement comparison for configurations 3 and 4.

model does not take into account the parallelism of the worker threads. The number of jobs in the system is the sum of number of jobs in service and the number of jobs waiting in queue. The model predicts 10.65 jobs on average to be in the system, while 9.74 of them are waiting in the queue. This falls in line with the assumptions of the M/M/1 model, which assumes a single queue and a single service. The prediction comes reasonably close to the actual average queuelength of 13, even though the system under test actually contains two queues (one per middleware). The prediction for the number of jobs in the system though is considerably off, because - again - the model does not include parallel processing of jobs servers. In Figure 35 we can see a comparison of the predicted and actual response time percentiles (as measured on memtier). Not only does the predicted percentile curve diverge considerably from the actual one, the predicted response times are also between 10x and 15x too low. The reason for that is that the model assumes a single service to handle all the requests alone, which of course would require the response time to be lower.

Furthermore we want to analyze configuration 4 with 8 workers. Looking at the model output (Table 5), the traffic intensity ρ almost reaches unstable territory, but is still < 1, which we see as a requirement for continuing our analysis. As the traffic intensity equals the service utilization, we can see that the system is practically fully saturated here. The predicted interarrival time again is about half of the actual one, as the model does not take the second middleware into account. The same is true for the worker service time. The number of jobs in the system again is equal to the number of jobs in the queue + 1. As the system has reached saturation in this configuration and as we have identified the number of worker threads as the bottleneck for 8 workers, the measured queue lengths reflects this. The predicted queue length is about double the size of the real one, as our experiment setup was using two middlewares and hence two queues. Nevertheless the predicted value captures the saturated state to some degree of precision. Additionally, the model correctly identifies the queue time to be the dominant factor for the overall response time and also predicts the measured waiting time reasonably well. Finally, the predicted percentiles (Figure 36) underestimate the actual response time distribution.

For configuration 2 and 3 a similar reasoning can be applied. While full saturation is barely being reached with 32 workers, the 16 worker setting model closely resembles the situation with 8 workers.

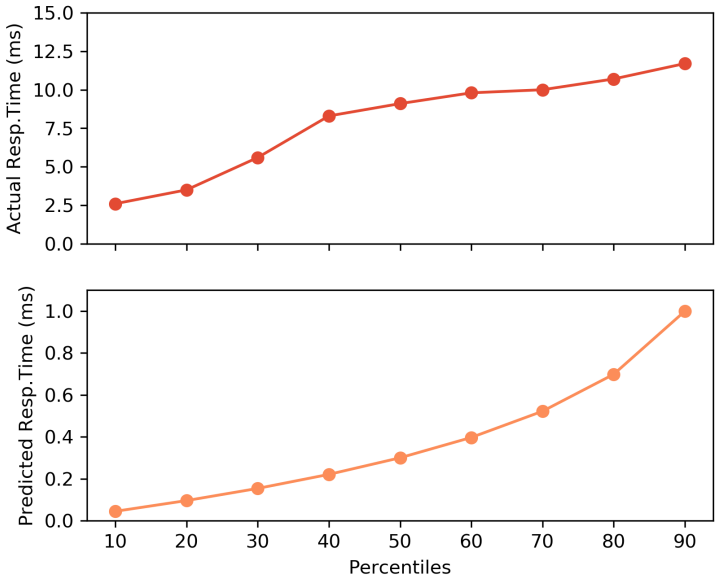


Figure 35: M/M/1: Response time percentiles predicted by the model and as measured on memtier (192 clients / 64 workers).

7.2 M/M/m

Seeing that the M/M/1 model in Section 7.1 fails to capture the intricate parallelism of the system under test, we move on to a slightly more complex M/M/m model. Compared to M/M/1, M/M/m assumes that there is more than one service available, that means multiple jobs can be processed in parallel. The model still assumes a single queue to feed the services. Similar to M/M/1, the choice of input parameters has a great effect on the accuracy of the model. For the mean arrival rate λ we again chose the average throughput. The mean service rate μ though has now been calculated from the measured average service time, which consists of $s = t_{preprocessing} + t_{memcached} + t_{postprocessing}$. From the logfiles, s can be found by taking `completedTime - dequeueTime` and averaging over them. The mean service rate then is $\mu = \frac{1}{\mathbb{E}[s]}$. Finally we need to find a suitable m, which denotes the number of service available to process jobs in parallel. Setting $m = num\_workers * num\_middlewares$ captures all worker threads in the system under test.

For the M/M/m analysis, we will focus on the two interesting worker configurations. Again, 192 clients are used:

Configuration 1: 192 clients / 64 workers: $\lambda = 23359$, $\mu = 206$, $m = 128$

Configuration 2: 192 clients / 8 workers: $\lambda = 8166$, $\mu = 523$, $m = 16$

7.2.1 Discussion

Table 6 shows an obvious match between the mean arrival rate and the mean service time, because we used those as the input parameters of our models. The service rate μ is now much lower than in the M/M/1 model, not only because we now calculate the service rate from the service time, but also because of the parallelism m that this model assumes. μ now correctly

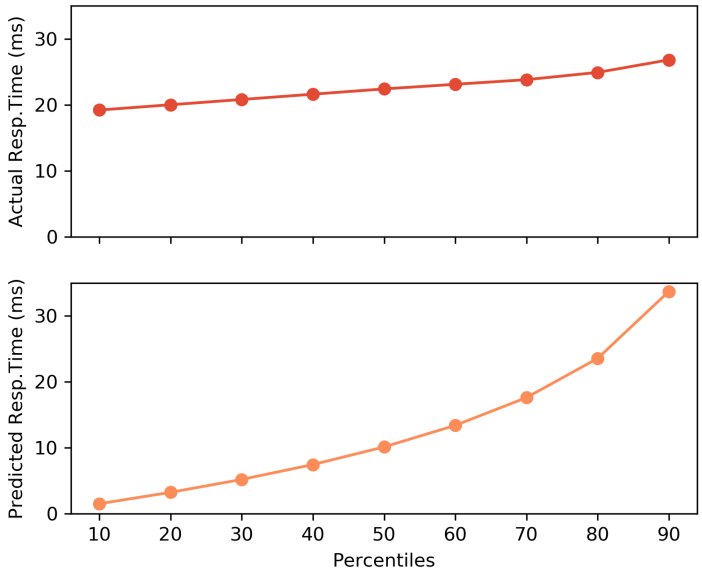


Figure 36: M/M/1: Response time percentiles predicted by the model and as measured on memtier (192 clients / 8 workers).

models the service rate of a single worker thread. The lower service rate for 64 workers stems from a significantly higher memcached response time, probably due to the memcached servers being hit by more requests in parallel.

The model correctly identifies the susceptibility to queueing for configuration 2, where only $8*2 = 16$ workers are available. The predicted numbers of jobs in the queue are too low though, partly because M/M/m still a assumes a single queue. The same reasoning can be used for the mean waiting time, which is also too low.

Finally I identify two major downside for using the M/M/m model for our system under test. Firstly, the aforementioned two middlewares contain one request queue each, which is not considered by the model at hand. Second of all, the model combines the middleware and memcached servers and completly abstract the latter away by just including the whole communication with memcached in the worker service time. Also all the workers are deemed identical, even though there are obvious differences in the worker service time between the two middlewares.

7.3 Network of Queues

In this final subsection we would like to use a more sophisticated network of queues to model our system under test. While we used data from experiment 4 (see Section 4) for the M/M/1 and M/M/m models, this network of queues will be evaluated against the re-run of experiment 3.2, where two memtier VMs create load on two middleware VMs which are both connected to a single memcached VM (see Section 3.2.1 for details). To keep things consistent with the other queueing models above we will focus on the write-only workload. The runs with 256 clients (64vc) will be analyzed.. This choice of clients should show interesting behaviour as the 8 worker setting is already and the 64 worker setting is on the verge of becoming fully-saturated. Figure 16 shows this. A diagram of the model developed can be found in Figure 37.

The following input parameters have been used for the models:

	Conf 1: 64 workers		Conf 2: 8 workers	
	Predicted	**Measured**	**Predicted**	**Measured**
Arrival Rate λ (op/s):	23359	23359	8166	8166
Service Rate μ (op/s):	206	–	523	–
Traffic Intensity ρ:	0.8846	–	0.9759	–
Interarrival Time τ (ms):	0.0428	0.0833	0.1224	0.2393
Mean Service Time $\mathbb{E}[s]$ (ms):	4.8475	4.8475	1.9119	1.9119
Probability of zero jobs p_0:	0.00	–	0.00	–
Probability of queueing ϱ:	0.1179	–	0.8901	–
# Jobs in System $\mathbb{E}[n]$:	114.14	126.26	51.61	96.26
# Jobs in Queue $\mathbb{E}[n_q]$:	0.9038	13.03	36.00	80.64
Mean Response Time $\mathbb{E}[r]$ (ms):	4.8862	6.4844	6.32	21.81
Mean Waiting Time $\mathbb{E}[w]$ (ms):	0.0387	1.62	4.41	19.88

Table 6: M/M/m: Model-Measurement comparison for configurations 1 and 2.

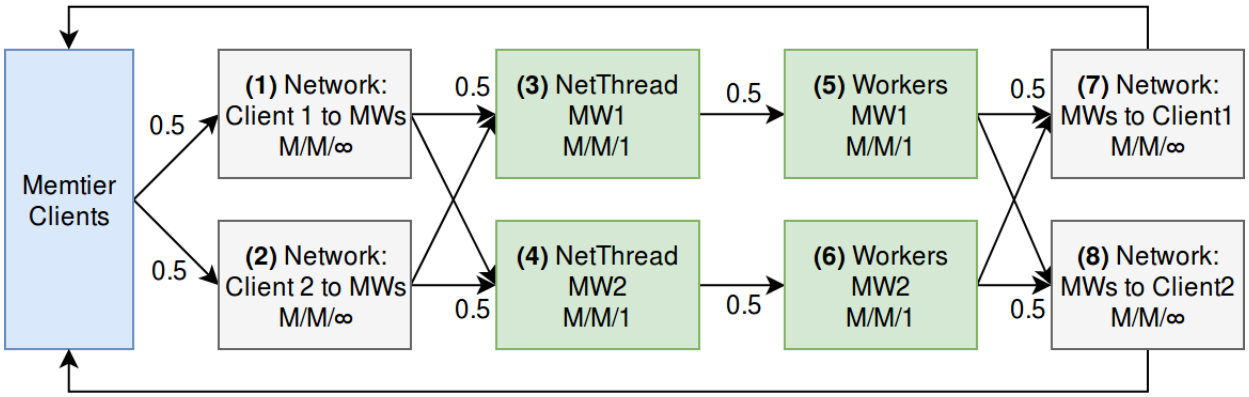


Figure 37: Network of Queues: Model diagram for experiment 3.2 (re-run).

Configuration 1: 256 clients / 64 workers: $\mathbf{S} = (0.0014006, 0.0012272, 0.0000048, 0.0000056, \frac{0.0039281}{64}, \frac{0.0035088}{64}, 0.0014006, 0.0012272)$

Configuration 2: 256 clients / 8 workers: $\mathbf{S} = (0.0012930, 0.0010774, 0.0000048, 0.0000040, \frac{0.0025422}{8}, \frac{0.0035088}{8}, 0.0023935, 0.0010774)$

7.3.1 Model

For our network we have combined the memtier clients into a single unit, as closed-systems assume jobs are originating from a single source. The two middlewares are modelled separately as they are indenpendent and also contain their own set of queues. The network latencies between memtiers and middlewares are designed as two delay centers whose service times are equal to the average pings measured on the respective clients. The ping logs show that the latencies from one particular client to the two middlewares are very close together. It seems thus that averaging over the measured latencies of a single client is a reasonable approximation. To keep things simple this approach was taken in favour of a more detailed network with four delay centers between the clients and the middlewares. As each memtier VM will simulate equally many clients for each middleware, the visiting ratios of these delay centers are 0.5 each. Afterwards I have decided to model the network threads as a M/M/1 queue with Javas

`Selector` API being the queue and the average `enqueueTime - arrivalTime` as the the service time. Again the visiting ratios of both queues are 0.5, as both memtier VMs simulate an equal number of clients for each middleware. After being processed by the network thread, the job enters the worker queue of its respective middleware, which has also been designed as an M/M/1. Here the service time is measured as the average of `completedTime - dequeueTime` on the respective middleware. Finally after being processed by the workers, two delay centers account for the time of transferring the response back to the clients. They incur the same delay as the delay centers at the beginning of the network.

There are two big caveats of the model at hand:

Memcached in Workers MW queue: Our network does not model the memcached server as a separate queue. The round-trip-time to as well as the service time of memcached are included in the Workers MW queue. The reason for that is that having a separate queue for the memcached server and potentially also including the network latency between the middleware and memcached as a delay center does not model the middleware behaviour correctly. If we would design the model like that, the service time in the Workers MW queues would only include the worker pre-processing time (from dequeueing the request in our middleware until sending it out to the memcached server). This would highly underestimate the service time of the middleware workers.

A second way to make this issue visible is that such a model would assume that as soon as the middleware has sent out the request to memcached, it is done processing and the request leaves the middleware. This is not the case in our system, where a worker thread will send out the request and wait until it has received the answer from memcached before it finishes processing.

Denning and Buzen (1978)[12] call this assumption "Single-Resource Possession": *"A single job may not be present (waiting for service or receiving service) at two or more devices at the same time."*.

Workers MW queue as M/M/1: The worker threads available dequeue jobs from a single queue request queue and then process them in parallel. We have discussed in Section 7.2 that this design comes close to how our middleware is implemented. Unfortunately, the Mean-Value-Analysis-algorithm as described in the book and as it is implemented in the `queueing` package for Octave[13] led to erroneous results due to numerical instabilities. The documentation of the package mentions this here[14] as well. Hence even though M/M/m queues would have better reflected our system design, limitations in the algorithm made a meaningful MVA impossible and we had to revert to M/M/1 queues. Instead of the real mean service time $\mathbb{E}[s]$ we have thus divided it by the number of workers w. ($\mathbb{E}[s]_{new} = \frac{\mathbb{E}[s]}{w}$)

7.3.2 Discussion

Finally let us compare the results of the model with the actual values measured on our system. The measured network thread utilization has been calculated by $U_3 = \lambda_{MW1} * S_3 = X_3 * S_3 = X_5 * S_3$. Due to the job flow balance and as the visiting ratio of center 5 directly follows center 3, we can take X_5 to calculate the utilization of the network thread for MW1. For MW2 a

[12] cited by Raj Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, April 1991, ISBN: 978-0-471-50336-1, page 551/552

[13] http://www.moreno.marzolla.name/software/queueing/

[14] http://www.moreno.marzolla.name/software/queueing/queueing.html#index-qncsmvaap

	Conf 1: 64 workers		Conf 2: 8 workers	
	Predicted	**Measured**	**Predicted**	**Measured**
Utilization Netthread MW1 U_3:	0.08	0.00	0.02	0.00
Utilization Netthread MW2 U_4:	0.09	0.00	0.01	0.00
Utilization Workers MW1 U_5:	1.00	0.92	1.00	0.98
Utilization Workers MW2 U_6:	0.89	0.88	0.94	0.99
Total Average Response Time R (ms):	7.85	8.22	40.68	39.68
Queue Length Workers MW1 Q_5	161.82	23.35	224.96	112.98
Queue Length Workers MW2 Q_6	8.3682	21.12	16.10	111.77
Throughput Workers MW1 X_5 (op/s):	16293	15055.91	3146.9	3097.04
Throughput Workers MW2 X_6 (op/s):	16293	16081.08	3146.9	3300.54
Total Throughput MT (op/s):	32586	30825.57	6293.8	6420.49

Table 7: Network of Queues: Model-Measurement comparison.

similar reasoning is used. The calculated utilization rates are very small, but due to rounding they are displayed as 0.00. While the model predicts a higher network thread usage between 0.08 and 0.09 for 64 workers, it still clearly shows that the network thread is not the bottleneck.

Differences in the utilization of the workers queues likely stem from different service times between the middlewares. MW1s network latency to the memcached server is slightly higher, leading to an overall higher service time of `3.9281ms` for 64 workers (compare with `3.5088ms` for MW2.). Hence the model predicts a higher utilization for the workers in MW1. It has successfully identified the worker queues as the bottleneck, even with the caveat that we modelled it as M/M/1 queues.

The total average response time has been predicted very well for both configurations, the queue lengths of service centers 5 and 6 though show large deviations. While the measured average queue lengths between the two middlewares are roughly equal, the network of queues shows fairly large queue lengths for MW1. This can again be explained by the deviation in service times. Even a miniscule response time difference of `150` microseconds in the case of the 8 worker configuration can lead to wildly different queue length estimates. The model has also been tested by setting the service times worker MW service centers to the combined average response time. As expected this led to the resulting queue lengths being almost equal, still these numbers were too high.

The throughput estimates by the model on the other hand match the observed throughputs very well.

The utilizations and response times that the mean-value-analysis produces for the delay centers have been omitted from table, as they are nonsensical and do not contribute to the analysis.

A Logfiles and Scripts

All logfiles of the experiments have been stored in Polybox.
The link is: `https://polybox.ethz.ch/index.php/s/IRzqWxO5kESWMas`.

For each of the experiments a subdirectory has been created with all the log archives available to download.

The source code repository can be found at
`https://gitlab.ethz.ch/fchlan/asl-fall17-project`. Scripts that have been used for deploying the VMs and running the experiments on Microsoft Azure are in the `./scripts` subdirectory of the git repository.

Postprocessing has been achieved by mainly using python and Jupyter notebooks (.ipynb files). All postprocessing scripts are located in the experiments `postprocessing` subdirectories in `./scripts`.

Upon opening the Jupyter notebooks, the experiment directory needs to be specified. This is usually a variable called `exp_dir` and should point to the path of the unzipped experiment logs.