Advanced Systems Lab Report

Autumn Semester 2018

Name: Mickey Vänskä
Legi: 10-930-386

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

Version: December 16, 2018

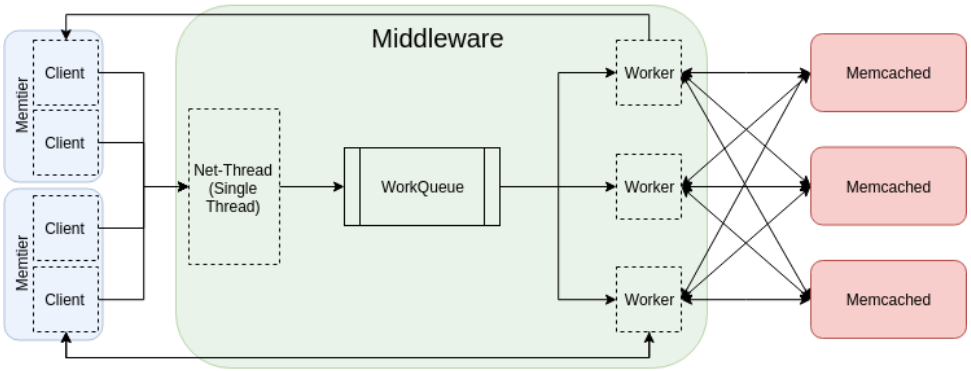


Figure 1 – Basic flow of information in the system. Each box with a dashed border indicates a thread on the respective system which are colour coded into 3 categories, blue for *Client*, green for *Middleware* and red for *Server*. Arrows denote the order of flow of information.

1 System Overview (75 pts)

The abstract flow of requests through the system is visualized in figure 1. The lifetime of a request begins in a thread on memtier, is received by the Net-Thread and upon receiving a complete packet it is stored in the WorkQueue. This queue is shared for any threads in the middleware and they compete for each individual element. The moment a Worker receives the request it processes it locally, checks the type and first sends the request to either one connected memcached (non-sharded GET) or all connected memcached instances (sharded GET and SET). Upon receiving replies from memcached there are checks for what was received and sanity checks included (e.g. receiving a SET reply for a GET reply is clearly an ERROR) before sending back a constructed reply directly from the worker thread to memtier given enough replies have been received. As such the Middleware receives only via the Net-Thread and only sends only via the worker when communicating with memtier instances. Communication between memcached follows a client-server model, the middleware being the client.

1.0.1 System Component Overview

For network communication Java NIO is used for all connections. The Net-Thread uses a `ServerSocketChannel` whereas worker threads employ for each connection with memcached an instance-private `SocketChannel`, all connections configured to be non-blocking. With the use of Java NIO `Channel`s writes are expected to perform more ideal in distribution from the middleware to either memcached or memtier. To manage connections the Net-Thread employs a single `Selector` which handles memtier connections, each worker thread two—one to manage all memcached connections and another to interact with memtier using referenced `SelectionKey`s from the Net-Thread's Selector. To allow handling of multiple requests the Net-Thread spawns a fixed amount of worker threads once at the start, all attached to an `ExecutorService` and upon finishing return gathered thread statistics. This is achieved by worker threads implementing the `Callable` interface. The gathered statistics are to be saved by the net-thread for each finished worker before termination. Termination can be achieved by sending the middleware a `SIGTERM` signal; a shutdown hook is installed right after the middleware is instantiated by the given wrapper class.
For the WorkQueue depicted an `ArrayBlockingQueue` is used, meaning blocking behaviour can be expected for the Net-Thread if the queue fills up with requests and blocking behaviour for workers should not enough requests exist. As such, the only producer is the Net-Thread and the only consumers worker threads. The content of the WorkQueue are `WorkUnit`s which are interpreted requests/replies and designed to be more programmer friendly than simple `byte[]`-

arrays at the expense of more garbage collection. These WorkUnits are the result of successfully receiving data from either party and the product of a `PacketParser`, used by connections of the Net-Thread and worker threads. It is designed in a stateful approach and as such each connection has a logically private instance of a PacketParser attached to it. This binding is done after the first successful registration of new connections.

1.0.2 Request Handling

Before any requests are processed by the middleware any incoming memtier connections must be first `accept()`-ed by the Net-Thread and registered in the attached Selector. Once done the PacketParser is called each time the OS notifies of new data being available on a registered SelectionKey. The PacketParser is designed to read data from a socket into a directly allocated `ByteBuffer` (size = 8192 B), generate a timestamp for the call to read data from the socket, receive data from the socket, parse the received data and extract at most a single request/reply, compact the ByteBuffer and store the result on success. This process is repeated as long as a request is stored ($\rightarrow$ returns a `List` of WorkUnit) and as such follows a greedy approach. The design aims to fully exhaust the data received from a single client before parsing new data. This would be troublesome in the general case of long streams but this is not expected and aims to allow quicker handling of single requests once they get actively processed. In the process of extracting data, three basic steps are done: read the current stream of Bytes linearly until `"\r\n"` is encountered, extract the data and match it Byte-by-Byte to a list of supported headers and extract type-specific header fields as per memcached documentation, and lastly either return the result as no body is expected or try to get the rest of the body. The body is read in a single command as is therefore likely to need multiple repetitions as MTUs are limited to 1500 B but the body being 4096 B. As such internal state is kept over rounds as to what part of a packet is currently expected to be processed. Once enough data is buffered the data is then copied and the result returned. As previously noted this result is a `WorkUnit`. It holds next to the received header and body (as `byte[]`) also fields for the type, type-specific fields (such as the list of keys for GET requests), the SelectionKey from which it was received and also a timestamp-object which keeps track on statistics when a packet was received (the moment of calling to parse incoming data), enqueued into the WorkQueue, dequeued from it, when it was ready to be sent to memcached, when all replies were received for it from memcached and finally when the reply was sent back to memtier.
The PacketParser returns once all data was consumed and returns a list of received packets. These are then stored to the WorkQueue for worker threads to process.

Processing WorkUnits on a worker follows the outline of dequeueing an element, checking its type, sending the request to either one (non-sharded GET) or all/some connected memcached servers (sharded multi-key GET or SET), waiting for the appropriate amount of replies, doing some local processing of the replies, sending it to the WorkUnit's attached SelectionKey and then dequeue another element, thus restarting the loop. Upon dequeueing the element is checked to be of type GET, SET, an invalid packet or anything else (the latter two resulting in logged messages). For SET requests the worker thread prepares for each memcached-instance connected to, a new ByteBuffer with a copy of the request and sends it to all memcached servers. It waits for each server to reply with 3 messages and verifies the content to be only STORED. On a failure an error message is returned, on a match any of the three requests chosen and sent back to the recipient by dereferencing the associated SocketChannel with the original WorkUnit. GET requests are prepared with load-balancing schemes (described in thorough detail later) depending on enabling or disabling sharding. The amount of keys in each GET (GETs with keys > 1 are called *multi-GET*) is of minor significance of load balancing. For the non-sharded case a single packet (a ByteBuffer) is created, the size remembered for load-

balancing purposes and mapped to the server with the smallest load. For the sharded case mappings of packets to servers in the degree of distributed load are generated. Skewed loads result in the degenerated case of a non-sharded multi-GET, good cases in a fair distribution of requests to servers (logically, they have not yet been sent, only prepared). After the packets are created and mapped to servers an expectation of replies is set for each memcached instance and the routine referenced in the distribution of SET requests is called. It is universal enough to also deal with misses and stops listening to the memcached instance once an "END" is received and internal state updated. After receiving all necessary replies sanity checks are again made, received "VALUE" requests serialized and an "END" packet appended to the collection of packets before a ByteBuffer is created from the replies and then sent to the respective memtier instance. In all cases where replies are received from memcached the associated PacketParser is used to read the received data and resulting WorkUnits stored in a result list. These WorkUnits are only processed to infer the validity of the replies from memcached.

1.0.3 System Instrumentation

For the following events in the system performance benchmark results are generated: In the PacketParser on the relating call when a header is correctly parsed ($\rightarrow$ timestamp for package arrived in the system, stored in the respective WorkUnit), on an enqueue and dequeue to the WorkQueue ($\rightarrow$ the current queue size is logged after the operation completed successfully, the only statistic on the Queue; additionally in both cases the WorkUnit gets the timestamp logged for the respective event) and at the following points in the worker thread. The aforementioned dequeue also generates packet-type specific metrics. For all valid types (SET, GET and multi-GET) the count of observed elements is incremented and the time on the queue logged. GETs of multiple keys also log the amount of keys. For invalid packets a counter is incremented. Before communication with memcached occurs but after the keys have already been mapped for servers and packets created a timestamp is stored in the WorkUnit. In case of an unexpected amount of GET replies the number of missing keys is logged for GET requests disregarding their size. It is still possible to infer whether the reason was a single-key or a multi-key GET. Another timestamp is generated after receiving and having processed all request-related replies as well in the WorkUnit (marks stop of memcached communication). In addition to this timestamp the communication with memcached and packet processing is logged for the respective request type. A last timestamp is stored the moment the reply has been successfully written to memtier's SocketChannel. This again also incurs a store of the total duration the WorkItem has been in the system and simulates an RTT through the system in addition to generating a statistic used for creating histograms (with 100 µs precision). Lastly at termination of the each worker thread the observed load distribution is remembered. All datapoints captured by a worker are upon termination accumulated and averaged (where applicable) in relation to the percentage of requests they served.
Timestamps on WorkItems are used to populate the aforementioned type-specific request metrics. All metrics mentioned that are not timestamps are accumulated in 1 second windows and in all cases but the load distribution and counters for requests / cache misses are averaged by the number of observed datapoints. This averaging is a simple arithmetic mean. The metrics on cache misses and request counts are simple counts and not averaged. The load distribution is continuously updated but returns an overall view at the end of the lifetime of the thread.

To start logging a common timestamp is set once the first packet has been fully parsed and ready to be dequeued. A final timestamp is set in the Net-Thread after all worker threads have finished execution.

Machine Type	Throughput [Mbit/s]	SET request	SET reply	GET request	GET reply
Client	201	6077.65	—	$1.32 * 10^6$	—
Middleware	802	24267.73	$2.25 * 10^7$	$5.27 * 10^6$	24261.85
Server	101	—	$1.57 * 10^6$	—	3055.42

Table 1 – Derived maximal requests per second for given upload speeds. Maximum request numbers per seconds are based on sizes of 4131 B, 8 B, 19 B and 4132 B for SET requests, SET replies, GET requests and GET replies respectively. These calculations exclude any network overhead.

1.0.4 Load Balancing GET requests

Each worker thread remembers which memcached instance received how much load and tries to balance requests based on the key-size passed and whether sharding is enabled or not. For non-sharded behaviour the best effort that can be done is returning always the memcached instance with the lowest amount of requests sent. On a tie any of the valid candidates is fine to be selected. For sharded requests this idea is further refined by the use of a sorting network which gives access to the memcached instances in an ordered approach. There the server with the smallest amount of work is greedily chosen and given as many keys as are required to close the gap to the second lowest server. This leads to either exhaustion of requests and one memcached instance is put under load or keys still need to be distributed amongst memcached instances. In this case two servers now exist with equal amounts of smallest load and another, new second lowest server must be found. Again keys are distributed equally such that the first two servers have now at least the load of the third server. As the experiments don't use more than three servers any leftover keys are distributed in an even fashion amongst memcached instances. Any uneven distributions for one multi-key GET request will be evened out by future ones as servers with high load are deprioritized with this algorithm. It is not required that worker threads communicate the amount of load each memcached server was subject to as on average this design converges towards a fair estimate of equal load being distributed to all participants.

1.1 Experimental Configurations

In this report the following notations exist: *Client*, *Middleware* and *Server* which refer to instances of virtual machines on Microsoft Azure of the following configurations.

Basic A1 (1 vcpu, 1.75 GB RAM) are used for *Server* actors, Basic A2 (2vcpu, 3.5 GB RAM) for *Client* actors and lastly Basic A4 (8vcpu, 14 GB RAM) for *Middleware* actors. All instances are running Ubuntu 16.04.5 LTS. Instance specific configurations for the experiments are the following:

1. *Client*s run memtier 1.2.15 to generate load on the system. The memtier configuration is the most variable over the run of experiments. As a basic building block the following command line is used for non-multiget request: `memtier --protocol=memcache_text --expiry-range=259200-259201 --key-maximum=10000 --data-size=4096 --clients=$VIRTUAL_CLIENTS --threads=$THREADS --test-time=$RUNTIME --server=$REMOTE --port=$PORT --ratio=$SET_RATIO:$GET_RATIO`

   For multiget requests the `--ratio` parameter reads in the size of the requested multiget size and also adds the argument `--multi-key-get=$MULTIGET_COUNT`. These variables are set according to experimental paramters.
2. *Middleware*s use OpenJDK 8 to run the middleware software. The middleware accepts as input parameters the number of threads, whether sharding is to be used for MultiGET requests and the memcached instances to connect to. The experiments describe the respective invocations of the middleware. As a basic building block the following

command line is used: `java -jar $JAR_PATH -l $LISTEN_IP -p $LISTEN_PORT -t $WORKER_THREADS -s $IS_SHARDED -m $SERVER_PORT_PAIR[*]`

The last parameter is the list of memcached server IPs with ports to connect to.

3. *Server*s run memcached 1.4.25 to reply to requests sent by memtier. The memcached configuration was configured to listen on any incoming requests with a single thread (`-l 0.0.0.0 -t 1`) and started as a system service.
4. All machines further use dstat to log resource usage in general next to pings between interacting machines. Additionally iperf statistics were generated between each client type to show network limits for upload and download speeds. A tabularized summary can be found in table 1.

The key size for the experiments has been fixed to 4096 B, the maximal key index to 10000 and their lifetime set to 259 200 s (72 h) to prevent misses. Before experiments were conducted each *Server*'s memcached instance was populated with dummy values using the following command: `memtier --protocol=memcache_text --expiry-range=259200-259201 --key-maximum=10000 --data-size=4096 --clients=1 --threads=2 --requests=10001 --ratio=1:0 --key-pattern=S:S --server=$REMOTE --port=$PORT`

Using the included wrapper of memtier which automatically infers the amount of keys used resulted in misses, likely due to a fencepost error in key calculations internally.

Experimental results were gathered for sets of runs. Experimental data was gathered for experiment 2 in a single run, for experiment 3 in a single run and experiments 4–6 were gathered in one single run together (Best effort approaches were made to run everything in one run yet due to financial reasons and Microsoft Azure's cloud being a turbulent environment single experiments with insufficiently comparable data needed to be run multiple times.) Each configuration was tested for 80 s three times with a stable window of 60 s being cut from the logs after 10 s have elapsed.

A note on statistics. During the analysis is has been inferred that queue sizes were not matching with the per-thread queue waiting time statistics gathered. For small loads a generally too high queue size was predicted whereas for high loads too small. The queue waiting times are in their design of the middleware a much better indicator of the queue size and as such the queue size reported is directly derived by the throughput for the window multiplied by the queue waiting time in this report (normalized to 1 s. The reason for incorrect queue sizes inferred by the middleware is due to an imprecise locking approach on statistics gathering.

All graphs except histograms, residual plots and QQ plots error have bands/bars given in this report.

2 Baseline without Middleware (75 pts)

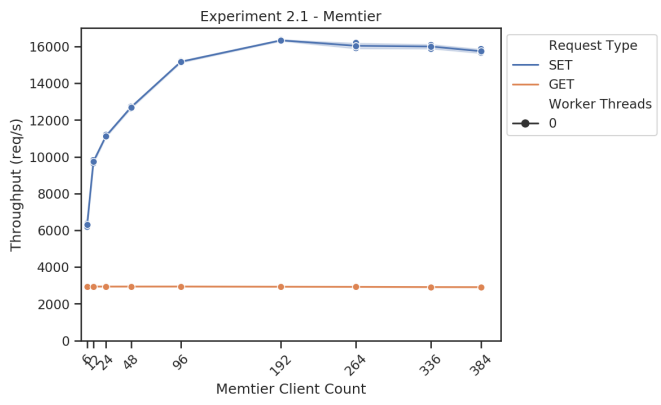
This experiment serves to establish a baseline of the system without the involvement of a middleware. The experimental parameters for experiments run are documented in table 2.

Two configurations are tested, one *Server* with three *Client*s and two *Server*s with one *Client*. The goal is to infer system limits and find out if even without the involvement of a middleware limits below the theoretical maximum are reached.

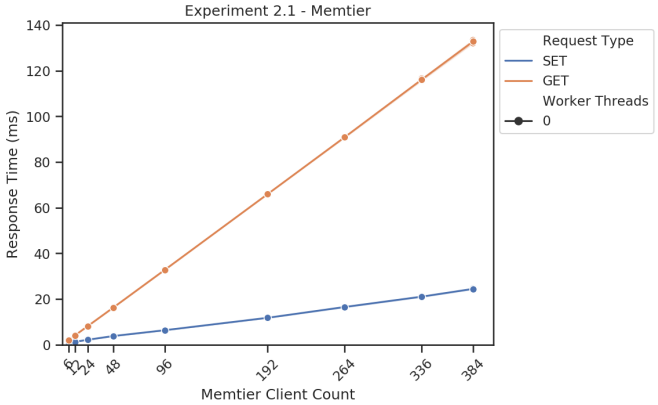
As each memtier instance can only connect to a single target (i.e. *Server*) to keep the behaviour consistent between experiments each *Client* connects, to keep both cores saturated, either with two threads for a single virtual target (as is the case in experiment 2.1) or one thread for a single target (experiment 2.2 —here two physical targets exist).

Number of servers	1 (2.1) / 2 (2.2)
Number of client machines	3 (2.1) / 1 (2.2)
Instances of memtier per machine	1 (2.1) / 2 (2.2)
Threads per memtier instance	2 (2.1) / 1 (2.2)
Virtual clients per thread	[1, 2, 4, 8, 16, 32, 44, 56, 64]
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

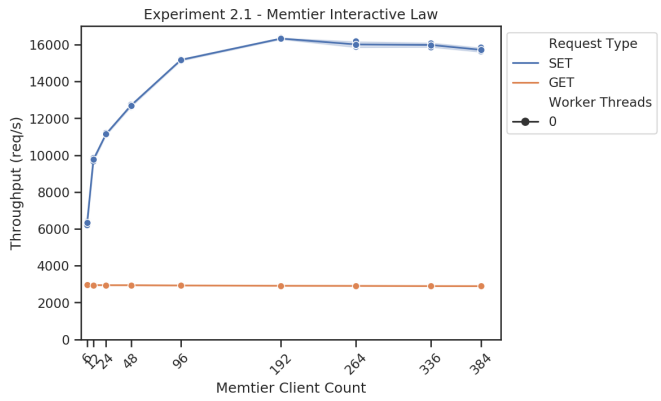
Table 2 – Experimental parameters for experiments 2.1 and 2.2.



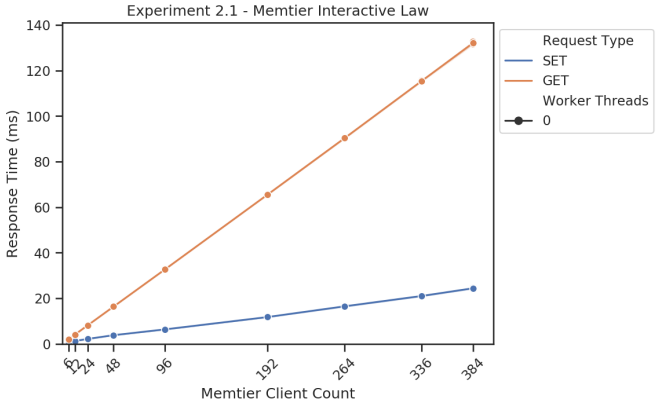
(a) Measured throughput on memtier.



(b) Measured response time on memtier.



(c) Interactive Law – Throughput.



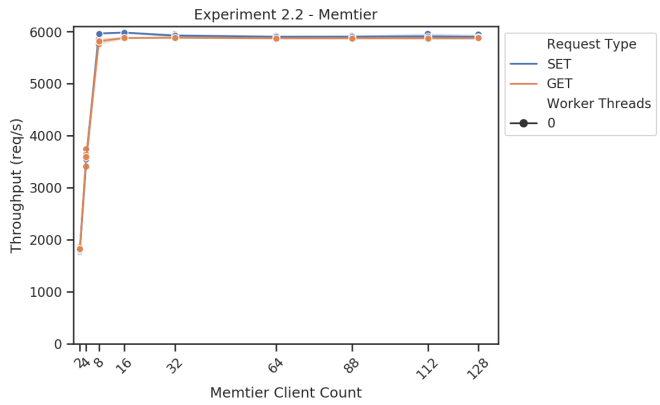
(d) Interactive Law – Response Time.

Figure 2 – Memtier measurements for Experiment 2.1. Error bars are plotted but not visible due to being too small.

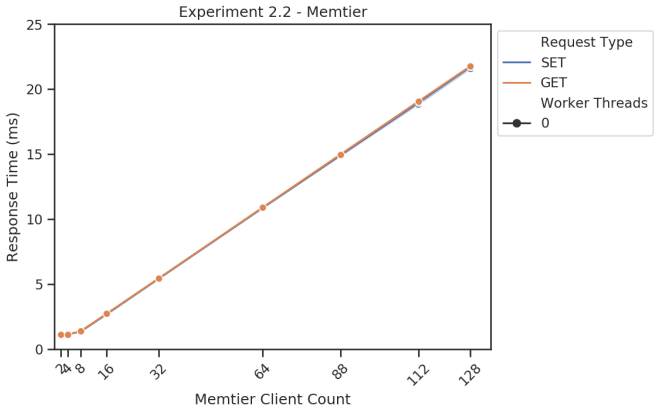
2.1 One Server

2.1.1 Explanation

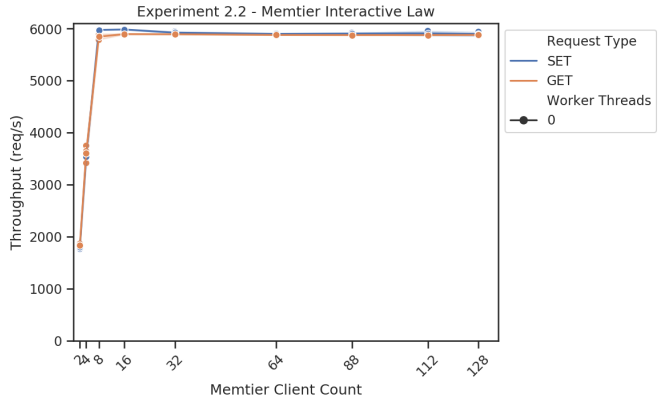
The memcached server is from the start saturated for GET requests as increasing the amount of clients results in no gain of throughput. This behaviour is predicted by table 1 where the theoretical limit is 3055 requests. As such the phase of under-saturation is not observed. The system is at the very least saturated. With the aforementioned 3055 requests a response time of 1.96 ms is expected for 6 clients. The measured data of 2.04 ms indicates a good degree of saturation but a clear statement as to how saturated is not possible. Including the response time graph it becomes clear that the system is still capable of handling more participants but the respective clients will have slower response times.



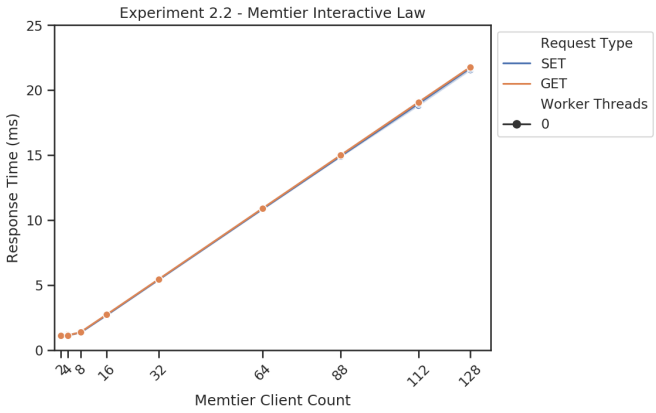
(a) Measured throughput on memtier.



(b) Measured response time on memtier.



(c) Interactive Law – Throughput.



(d) Interactive Law – Response Time.

Figure 3 – Memtier measurements for Experiment 2.2. Error bars are plotted but not visible due to being too small.

The memcached server behaves differently for SET requests. From a visual observation it seems to only saturate at 192 clients with a throughput of 16334 requests—performance degrades thereafter. As the theoretical limit for three memtier instances should be 18233 keys it is interesting to observe only a utilisation of 89%. Observing dstat results shows the system reaching full saturation on the CPU end for counts of 192 clients and beyond (in those cases the system is actually overloaded), meaning even though the network still can accommodate requests the system is CPU bound. As such it is concluded that a single *Server* cannot handle more than 192 clients reasonably.

The interactive law without thinking time holds as seen in figures 2c and 2d.

We conclude therefore that the bottleneck for memcached machines is the upload of data for GET requests and CPU performance for SET requests. The next experiment will clarify the bottleneck for memtier machines.

2.2 Two Servers

2.2.1 Explanation

The throughput of the system behaves as expected when adding another *Server* to the configuration and removing two *Client*s. SETs are reaching the network bound (cf. table 1) as more *Server*s exist which allow handling twice the amount of requests for the same instance of time whereas GET requests also show being limited by the upload bandwidth of both *Server*s

Experiment	Read-only workload	Write-only workload	Configuration
One *Server*	2937.75	16334.11	Clients = 6 / 192
One *Client*	5816.28	5965.34	Clients = 8 / 8

Table 3 – Evaluation of maximum throughputs of baseline experiment. Numbers are rounded to two digits.

(cf. table 1). After reaching the saturation points in throughput a measurable increase in latency can be seen for either request type. These behaviours align with expected behaviour and also reflect well when applying the interactive law (figures 2c and 2d).

We conclude the bottleneck for *Client*s is not immediately reached but very early on. With eight clients being the point of saturation for GET requests for two memcached servers it is natural to assume that four clients are saturating one memcached server. As such the assumption of memcached being already over-saturated in experiment 2.1 can be drawn.

Memtier machines have a maximum throughput of about 5800 requests per second in these experiments. Of note is that the throughput doesn't compare to the previous experiment linearly but can be explained by a variation due to network latencies and also multiple machines resulting in overlaps of commands to memcached whereby some waiting is required for near-simultaneous requests as each memcached instance is run single-threaded.

2.3 Summary

The maximum throughput is easy to find out by visual inspection of the graphs for GET requests (as no gain in throughput is observed after a certain point). SET requests have for the case of three *Client*s and one *Server* more ambiguous result. One can simply look for the maximum amount of requests through the system but not including the increasing response time would be naive. With the limited details the decision of maximum saturation is a combination of limited growth in throughput and still constant growth of response times. For the case of one *Client* and two *Server*s a visual inspection is sufficient.

The difference of how GETs and SETs are bottlenecked has been described in subsections 2.1 and 2.2 and expected at this point. Table 1 summarises the theoretical limits.

For the case of GET with one memcached instance this limit matches with the observed result. For two memcached instances with a single memtier instance this limit is twice as high as two instances are queried. That experiment's respective SET behaviour can be explained by the upload limit of the respective machine (cf. table 1) and not by memcached. This claim is strengthened by the fact that a single instance can handle up to 16334 writes in our experiments.

In either case for the single memtier instance, an increase in steepness for the response time can be observed after the saturation point. The alignment of response times for both request types matches their respective throughput.

The interactive law graphs align in all cases with the measured numbers and as such it can be deduced the thinking time is very close to 0 and all components introducing long response times are observable within this configuration.

3 Baseline with Middleware (90 pts)

This experiment extends on the design of experiment 2.1 by introducing one and two *Middleware*s into the system. The middleware can be configured to have a variable amount of worker threads where each worker thread handles one paired request-reply operation. This parameter is also considered in the evaluation. The configurations are listed in table 4.

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1 (3.1) / 2 (3.2)
Threads per memtier instance	2 (3.1) / 1 (3.2)
Virtual clients per thread	[1, 2, 4, 8, 16, 32, 48]
Workload	Write-Only and Read-Only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1 (3.1) / 2 (3.2)
Worker threads per middleware	[8, 16, 32, 64]

Table 4 – Experimental parameters for experiments 3.1 and 3.2.

3.1 One Middleware

This experiment is similar to experiment 2.1, but here each *Client* connects to the single instance of *Middleware* which itself connects to the single instance of *Server*.

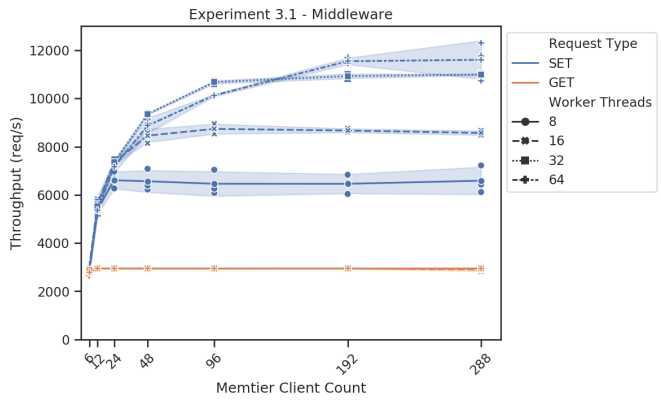
3.1.1 Explanation

Comparing the throughput of GETs with the baseline experiment only a small change in performance for 6 clients is observed (the performance not reaching the stable phase which happens for 12 and more clients). The throughput of SET requests behaves most similarly to the baseline with 64 worker threads yet the throughput for 32 worker threads is in general higher for 48 and 96 connected clients. This observation will be discussed in the following paragraphs and explained with more data. It can be inferred that the throughput of a single client is limited to 11546 requests, a contrast to the measured 16334 without the use of a middleware. This limit occurs for 192 virtual clients. The throughput being lower for fewer worker threads is reasonable and the non-linear scaling highly likely due to the fact that *Middleware*s have only 8 physical cores.

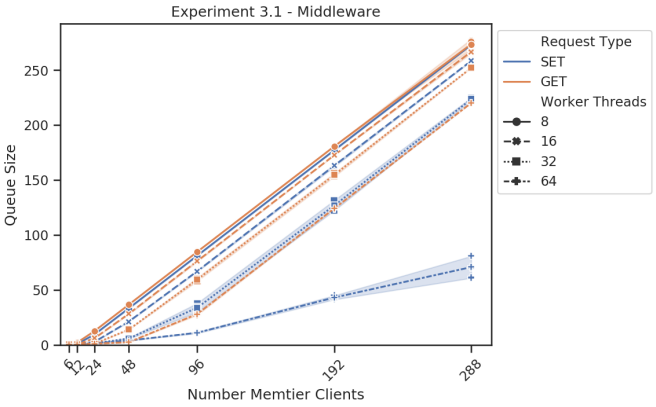
The response rates are also similar to the baseline experiment in general with SETs having a smaller response time than GETs. Looking at the constituents of response time (with detailed timings for queue waiting times and memcached communication and respective packet handling time) it can be observed that for few worker threads and many clients queue times grow from a certain point quasi-linear. The memcached communication and packet handling time does flatten though. The height of flattened sections can be explained due to thread scheduling (the flattened lines are all separated proportional to the number of worker threads divided by the core count of the machine). The reasoning for flattening due to the fact that the middleware cannot handle more requests in parallel and is therefore saturating for the worker thread and client configuration. This can be seen by queues growing for the configuration of active clients and threads.

As such the total response time of the middleware is for a large amount of clients and few worker threads determined by the queueing time and for few clients and many worker threads determined mostly by the communication time with memcached.

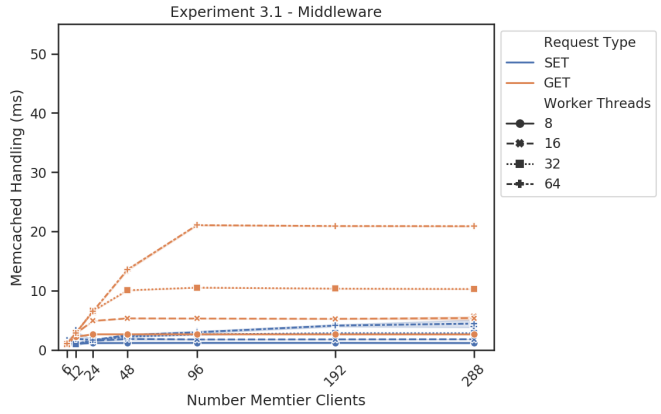
Coming back to the open question of why 32 worker threads perform better than 64 for 48 and 96 clients requires the use of the queue where requests are stored. As long as this queue is not at least the size of worker threads, performance impact can be expected due to worker threads being idle and waiting for a new request to process. This results in an effectively smaller throughput as only some threads are working but the threads waiting for a new element contend on a shared lock on the queue. Only once the queue at 192 clients approaches 64 elements does the middleware become more saturated and is even more so at 288 clients. It is very close to the point of complete saturation when looking at the queue size, yet no real increase in throughput was measured. As the standard deviation for 288 clients and 64 worker threads is rather large



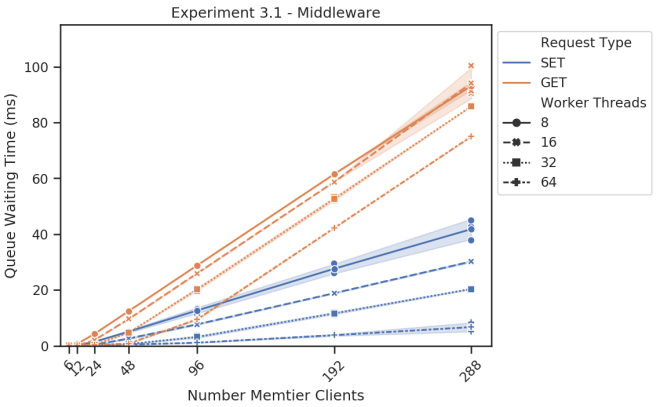
(a) Request throughput.



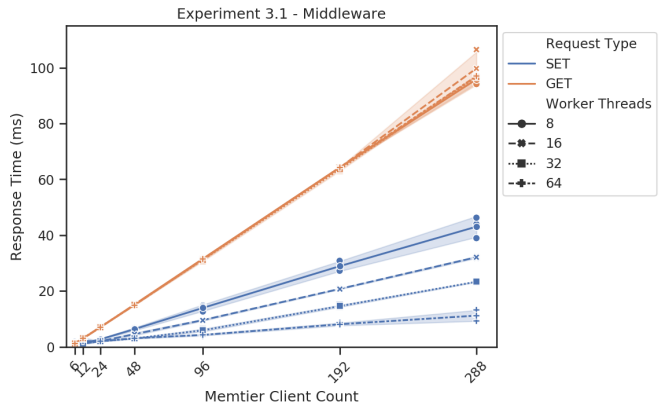
(b) Queue sizes.



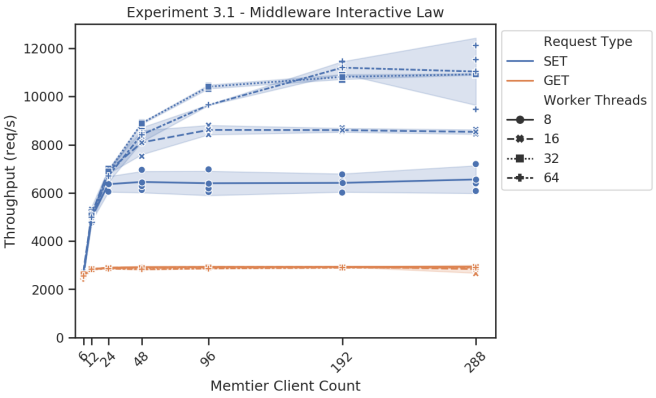
(c) Memcached communication time.



(d) Queue waiting time.



(e) Response times.

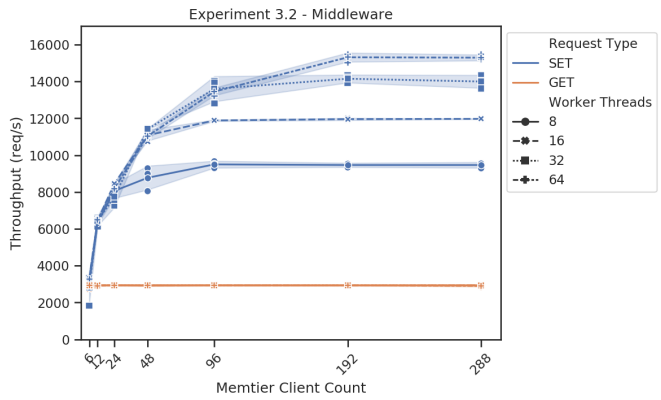


(f) Throughput calculation using the interactive law with an adjusted client count.

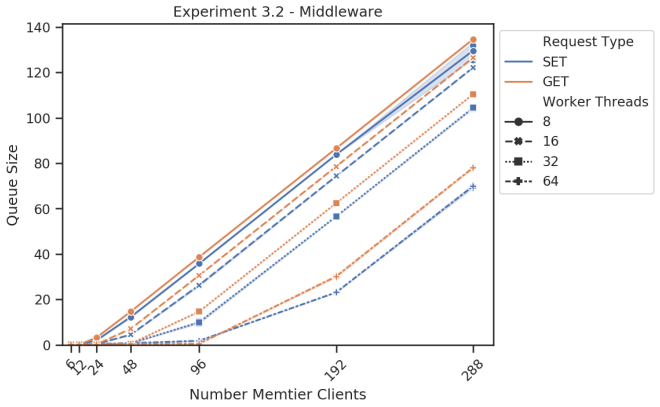
Figure 4 – *Middleware* statistics based on SET and GET requests for various amounts of worker threads and active clients.

compared to previous results allows the claim that beyond a certain amount of threads and connections the cloud environment is not scaling well.

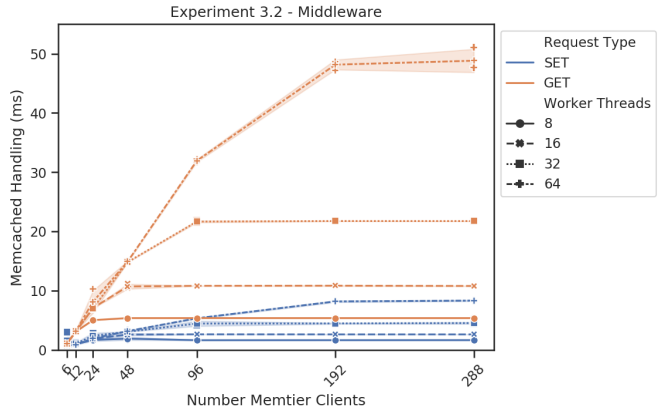
Overall it can be concluded that for 8–32 clients the maximal saturation point has been reached and for 64 threads the saturation point is within a few percent or has been reached.



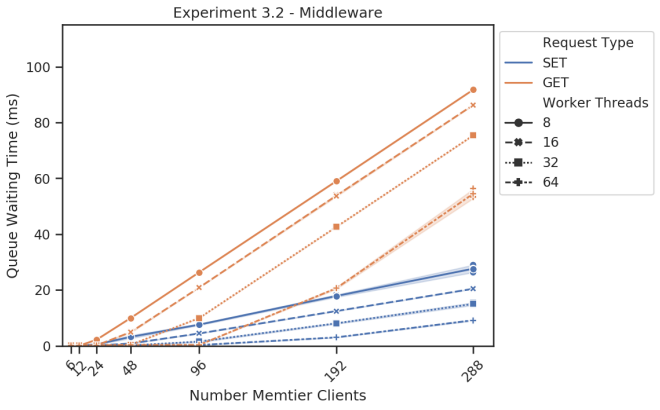
(a) Request throughput.



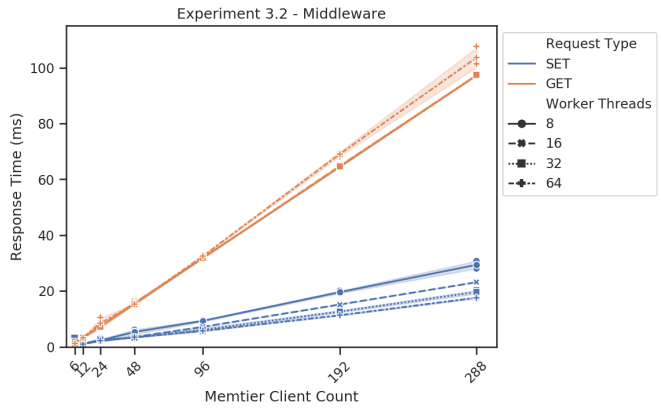
(b) Queue sizes.



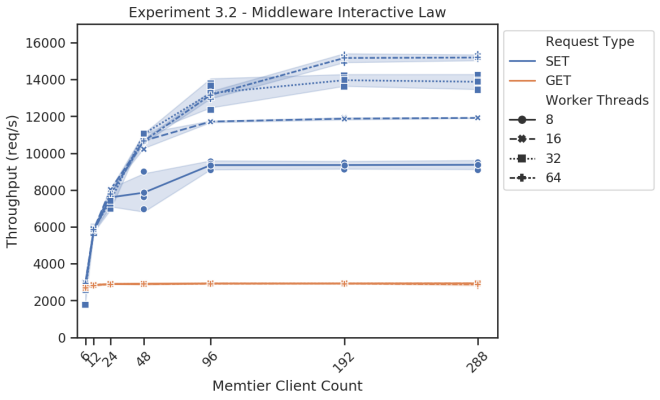
(c) Memcached communication time.



(d) Queue waiting time.



(e) Response times.



(f) Throughput calculation using the interactive law with an adjusted client count.

Figure 5 – *Middleware* statistics based on SET and GET requests for various amounts of worker threads and active clients.

3.2 Two Middlewares

This experiment is virtually identical to the previous experiment with the change of each *Server* running two single-threaded memtier instances where the second connects to the newly added second *Middleware*. As such each *Middleware* is only subject to half the load compared to the previous experiment's single *Middleware*.

		Throughput		Response Time		Queueing Time		Miss Rate	
Type	Machine	Exp 3.1	Exp 3.2	Exp 3.1	Exp 3.2	Exp 3.1	Exp 3.2	Exp 3.1	Exp 3.2
GET	*Client*	2936.83	2923.78	65.32	65.92	N/A	N/A	0.00	0.00
	Middleware	2939.61	2929.16	64.17	69.02	42.17	20.60	0.00	0.00
SET	*Client*	11549.70	15308.52	16.73	12.55	N/A	N/A	N/A	N/A
	Middleware	11546.60	15312.49	8.07	11.29	3.80	3.01	N/A	N/A

Table 5 – Evaluation of maximum throughputs for one and two *Middleware*s in the system with 192 clients and 64 worker threads.

3.2.1 Explanation

Comparing the throughput of GETs with the baseline experiment and the experiment with a single *Middleware* no large differences can be observed except a stable phase from the very beginning. The throughput for SET requests behaves much "cleaner" for increasing amounts of worker-threads, meaning that for not enough saturation by connected clients the resulting performance for SETs behaves the same. Also more throughput occurs throughout the system. The maximum observed throughput is 15312 requests, very close to the baseline experiment of 16334 requests. The middleware shows to be a reasonably small disturbance to the overall system for enough worker threads active. The scaling of throughput for worker threads follows the same reasoning as in experiment 3.1.

The response times reflect the increased throughput with overall lower numbers. There is less spread for SET requests amongst different amounts of worker threads. A comparison between the response times with a single *Middleware* and two *Middleware*s is conducted in subsection 3.3. The memcached communication and packet handling time doubling follows from the fact that with two concurrent *Middleware* SET requests sent to the same *Server* a serialization of requests is expected (as the *Server* is only single-threaded). Thusly larger response times on the end of memcached are observed. This would result in an overall higher response time if not for the fact that as there are now two *Middleware*s, each instance is only subject to half the load generated by *Client*s. As such the queue sizes are also only half of what was measured when compared to a single instance of *Middleware*.

The queue sizes grow disregarding the type for different amounts of worker threads similarly. Even though fewer load is received by each middleware the system still has growing queues. Applying the previously stated argument of the system being fully saturated for queues of size exceeding the current amount of worker threads shows that this experiment as well shows saturation points for 8–32 worker threads and being very close to the saturation point of 64 worker threads.

An artifact of running in a cloud environment can be seen for 6 clients and 32 worker threads where the throughput is low and the response time including memcached communication unexpectedly high. Looking at the respective logs shows a likely network issue on the second run, as no system is overtaxed yet the *Server* is not sending enough data. Statistics generated by ping indicate unusually high response times for communication between *Middleware*s and the single *Server*. This reflects in the memcached communication time and response time of the system. The system is for most of the supposedly stable window overall very shaky.

3.3 Summary

As already mentioned, in the baseline experiments without the use of a middleware the evaluation of GET and SET requests follows similar base criteria, namely throughput and response time. In case where these are not sufficient, it is possible to use the middleware's

instrumentation and deduce the system behaviour from them.

In the case of GET requests the behaviour between experiments 2.1, 3.1 and 3.2 is comparable. Oddly for six clients the single *Middleware* configuration is not immediately saturated for any amount of worker threads. Only at twelve clients this saturation occurs. It is quite likely that the overhead of the middleware is just high enough to introduce minimal latencies in such a configuration as the system has empty queues and as such it is not a problem of queueing. As experiment 2 and 3 were run on two different instances of machines changes in benchmarking performances are expected, yet the general behaviour should stay comparable. This is the case for GETs in our experiments as well as for SETs.

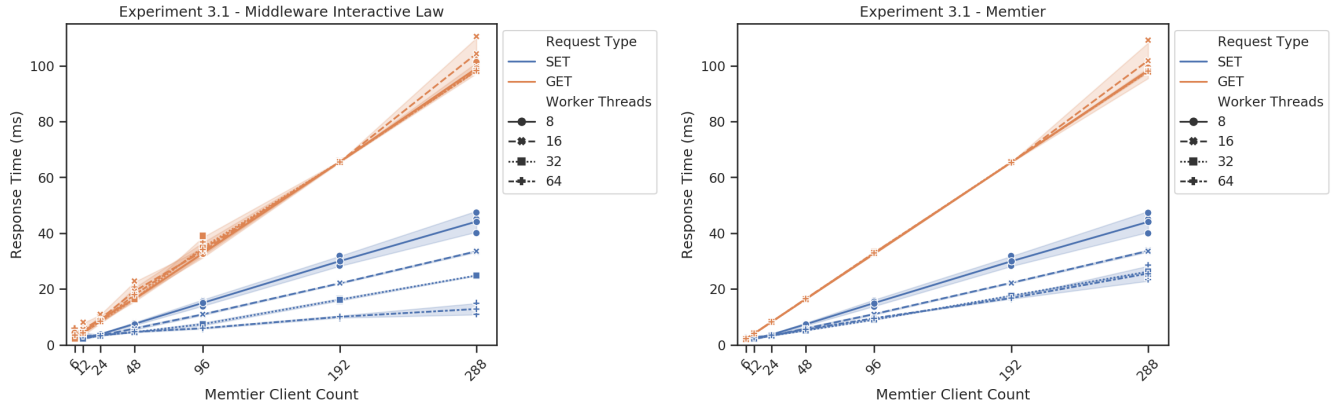
In the case for SET requests the trends with more active clients introducing more throughput does not always match. As long as worker threads are not oversaturated the performance will not degrade. After oversaturating the system the middleware slows down the system as it cannot handle more requests. There is a trend of more worker threads introducing more throughput. Also adding another middleware helps and actually shows to scale better than adding more worker threads when comparing the throughput graphs for the total amount of comparable worker threads between both experiments (for experiment 3.2 half the worker thread count must be evaluated as two middlewares are used). This is to be expected as at some point scheduling overhead on a single system results in slowdowns. Another viable option is a better allocation in the cloud environment.

For a thorough and clear evaluation of system behaviour it is necessary to include the queue and memcached communication time.
As already observed for a given amount of worker threads the memcached communication begins to flatten for a certain amount of active clients. The trend is that more worker threads can handle more clients but the time to process grows. This aligns well with the fact that there is contention for the actual cores of the machine by each worker thread. A detailed explanation on the respective behaviour has been given in subsubsection 3.1.1. The queue sizes also reflect in the queue waiting times appropriately for both experiments in this section. Additionally the queue sizes offer a good metric in evaluating the saturation of the system and aid in defining the maximum throughput for each configuration of worker threads. Once the queue sizes reaches the amount of worker threads there is no more more waiting on the next task for each worker thread but adding more elements than there are worker threads slows down the system as at that point no bijection exists between worker threads and items to process.

As the middleware doesn't include the full response time the interactive law doesn't hold when assuming the whole system is only *Middleware – Server*. To fix this the interactive response time law, stated with

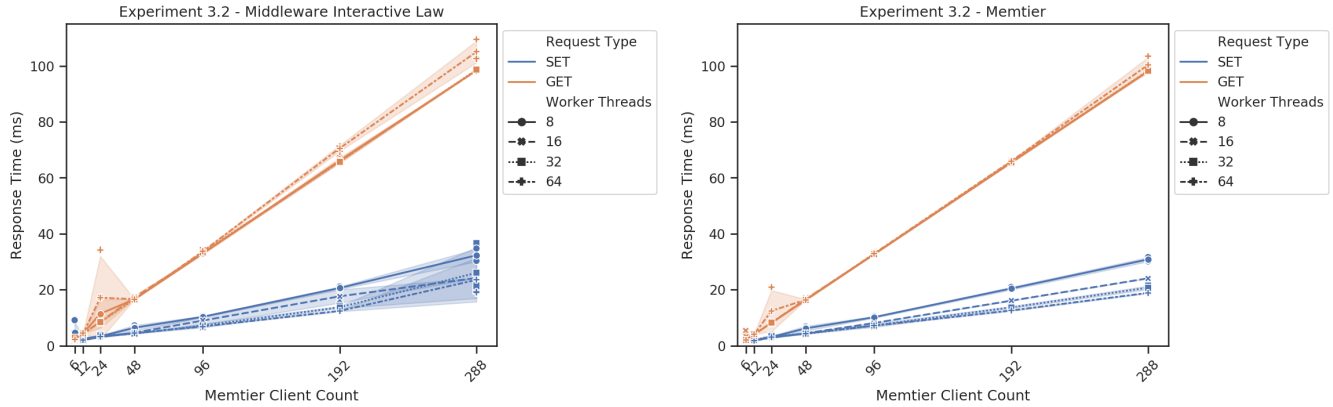
$$R = \frac{N}{X} - Z$$

allows the modification of either N (number of active clients) or setting Z (the thinking time) to a non-zero value. The latter is usually done by using the RTT between *Server* and *Middleware* (and add some minimal extra to allow memtier processing time). To adjust N it is relevant to define what the number of actual jobs are in the current system. This is defined by two components, the queue of the system modelling inactive jobs still to be processed and the active participants which are inferred from the system throughput and memcached communication time. Multiplying the throughput of the system with the time it takes for workers to have completely handled communication with memcached gives a good amount of active workers. Adding these two together and dividing by the observed response time of the middleware gives



(a) *Middleware* interactive law deriving the response time.

(b) *Client* measured response time.



(c) *Middleware* interactive law deriving the response time.

(d) *Client* measured response times.

Figure 6 – Visualizations of the interactive law plotted for experiments 3.1 and 3.2. High loads result in less correct predictions.

a prediction of the throughput on the system in relation to active participants.

$$X_{pred} = \frac{X * \text{Time}_{\text{Memcached}} + \text{Size}_{\text{Queue}}}{\text{Time}_{\text{Middleware}}}$$

In the following the interactive law is applied and the response time derived using the latency from each experiment. Figures 4f and 5f depict the interactive law applied with a modified amount of clients and in figure 6 the application of the thinking time is compared to with measurements from memcached. In general the interactive law holds in both cases but the throughput calculation shows cleaner results and is thusly used onwards. Some outliers are expected for high loads as graceful failures are not the nature of computers.

4 Throughput for Writes (90 pts)

4.1 Full System

This experiment extends on experiment 3.2 and show how load-balancing SET requests amongst multiple *Server*s affects performance. In contrast to that experiment 3 *Server*s are used. The detailed configuration is listed in table 6.

4.1.1 Explanation

Compared to experiment 3.2 a decrease in performance can be observed for throughput. This follows from the fact that each *Middleware* is connecting to three *Server*s instead of just a

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, 2, 4, 8, 16, 32, 48]
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]

Table 6 – Experimental parameters for experiment 4.0.

		Throughput			Average Time		
WT	Clients	*Middleware*	*Middleware* – IL	*Client*	Memcached	Queue	Queue Size
8	48	6680.07	6503.86	6675.16	2.28	3.81	12.72
16	96	8616.11	8448.11	8611.22	3.57	6.27	27.00
32	192	10327.79	10187.26	10323.51	6.01	10.69	55.18
64	288	11469.24	11307.79	11463.20	10.84	10.73	61.54

Table 7 – Evaluation of maximum throughputs of SETs for a full system. IL denotes the Interactive Law applied on the response time applying a thinking time based on the experiment.

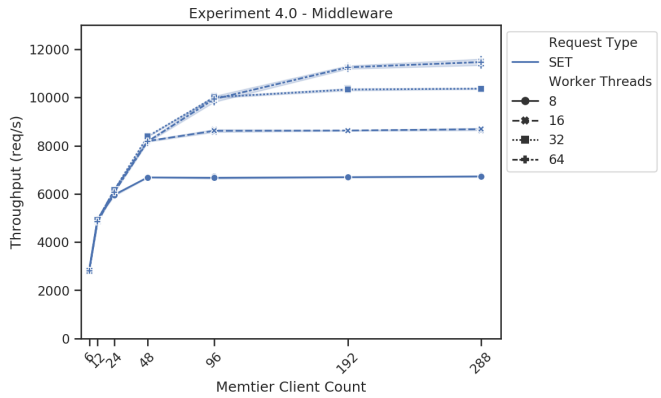
single one and SETs need to be replicated amongst all connected *Server*s. It is relevant to recall the design of the middleware. It may serve multiple requests at once but each request is handled in a single threaded fashion and as such no parallelization for communication between the middleware and memcached occurs.

As in the previous experiment, throughput increases are observed for increasing amounts of worker threads and active clients in the system. In the case of 8 worker threads limited gains are observed for 24 clients, no gains after 48 clients; 16 worker threads have comparable behaviours for 48 and 96 clients; 32 worker threads for 96 and 192 clients and 64 worker threads for 192 and 288 clients. A clear trend can be observed thus with saturation points approaching when three times the amount of clients connect to the current worker thread configuration and actually reached for four times the amount of worker threads. This reflects in the queue sizes seen in figure 7d. For the aforementioned amount of clients a change in steepness can be observed which shows the queue beginning to fill up stronger. As a consequence queue waiting times grow which means the overall response time of the system increases. Yet small throughput increases are expected as now each worker thread can immediately get an element from the queue without having to wait on said queue. This claim assumes the memcached communication time to also stay constant between the amounts of clients for limited and no gains. This holds well for configurations of 8–32 worker threads. For 64 worker this is not strictly speaking observed but overall the gains of throughput are small enough to claim the system is well saturated as indicated by the growing queue.

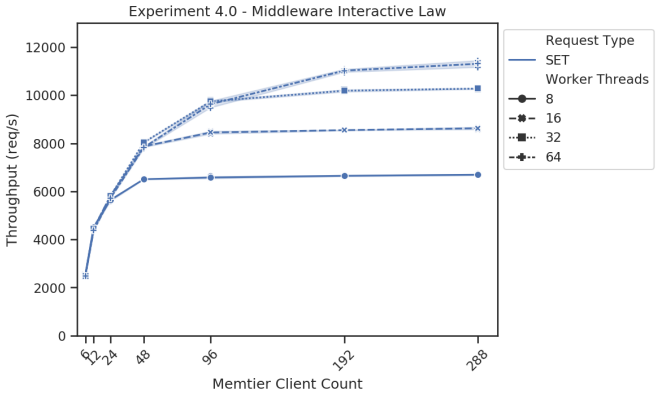
One note on the queue size on the previous claim to hold is that the queue must be larger than the amount of current worker threads. As such each worker thread has the guarantee to pick up a new task without having to wait for said task to be generated.

4.2 Summary

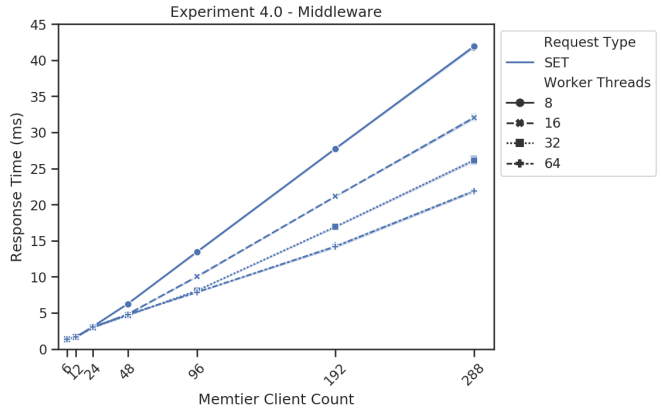
The numbers presented are derived applying the reasoning in subsubsection 4.1.1. It can be observed that the trend to an increase in throughput with the amount of worker threads is valid yet at a certain point diminishing returns are to be expected. The throughput deltas are (rounded to integers) 1936, 1712 and 1141 and illustrate the point of not scaling linearly



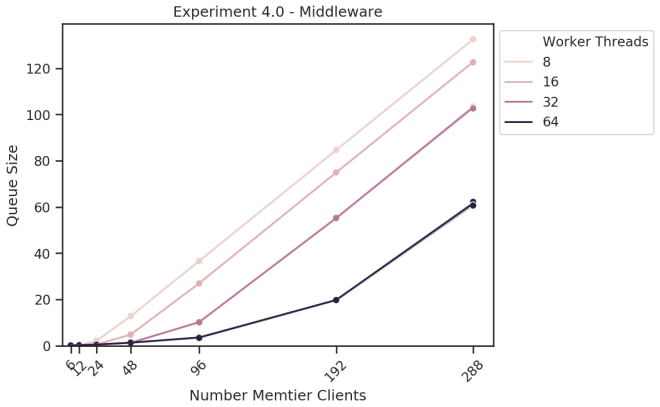
(a) Request throughput.



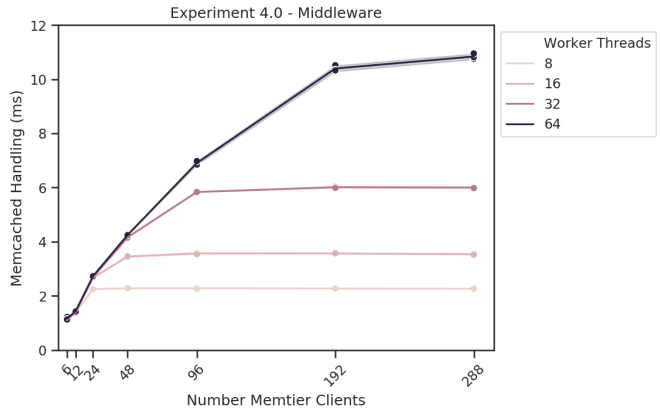
(b) Throughput calculation using the interactive law with an adjusted client count.



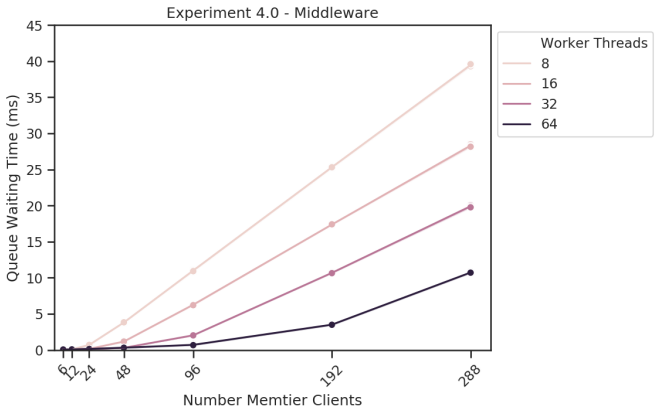
(c) Response times.



(d) Queue sizes.



(e) Memcached communication time.



(f) Queue waiting time.

Figure 7 – *Middleware* statistics based on SET requests for various amounts of worker threads and active clients.

with the amount of worker threads. Additionally the throughput measured by *Middleware*s and *Client*s match up well. The same argument is applicable to the interactive law. Numbers are in general a bit lower than measured but this is expected as numerical evaluations on the middleware are not able to generate infinitely-precise statistics. Verifying the interactive law (figure 7b) shows very similar throughput graphs obtained for the middleware.

A note on the queue sizes. They are for all but the case of 64 worker threads above said amount.

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	ratio=1:<Multi-Get size>
Multi-Get behavior	Sharded (5.1) and Non-Sharded (5.2)
Multi-Get size	[1, 3, 6, 9]
Number of middlewares	2
Worker threads per middleware	64

Table 8 – Experimental parameters for experiments 5.1 and 5.2.

The system is therefore expected to actually be slightly oversaturated for all points whereas for 64 worker threads the point of perfect saturation is reached to within a couple percent.

We can therefore derive that in general between three to four times the amount of clients for worker threads is the optimal setup for write only workloads if the minimum amount of threads is to be used. There is no measurable loss of performance in adding more worker threads.

5 Gets and Multi-gets (90 pts)

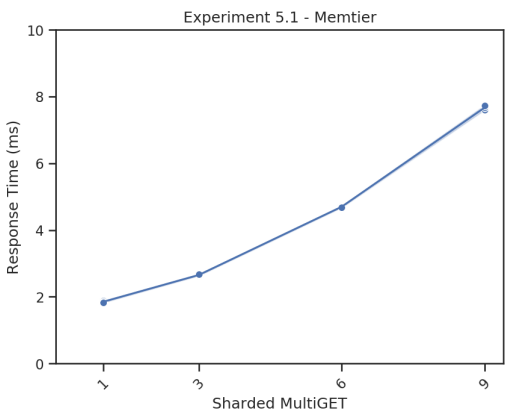
In this section the system is analyzed for mixed workloads whereby SET requests and GET requests are in the ration 1:1 but for GET the amount of keys requested is increased in addition to the middleware possibly splitting up GET requests with multiple keys fairly to each *Server*s. The experiment aims to find out under what conditions this splitting, also called *sharding* is preferable and how the throughput changes amongst differently sized GET requests. In table 8 the experimental setup is documented.

The experiment intends to use the most performant middleware as such the correct number of worker threads needs to be inferred for the case of 12 total clients. This experiment has two limiting factors:

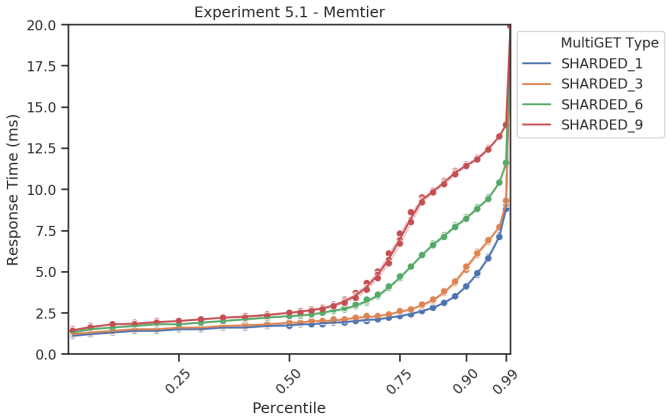
- GET requests are network bound on the end of each *Server*. The number of requests is expected to go down for more keys as replies get larger.
- SET requests are CPU bound on the end of each *Client*. The number of requests is expected to not change due to CPU bounds but because the corresponding amount of GET requests is going down.

From experiment 4 it becomes clear that there is no significantly measurable performance impact using more worker threads than is required for clients. This means that any amount of worker threads beyond 8 is appropriate to use for this experiment for SETs. Experiment 3 is of minimal use as only single *Server* configurations were tested. Yet it can be deduced that the middleware can handle at least one *Server*s load of single-key GET replies without any problems. As no significantly measurable performance impact for SETs exist and multi-key GET requests are not yet observed with three *Server*s active the decision was made to use 64 worker threads to exclude any performance impacts possible due to too little threading on the middleware.

Important in this section's data is the exclusion of the 60 s stable-phase window as histograms are used. Memtier reports histograms for the complete run and not a configurable stable window. This decision was made to allow comparing not only data from the middleware and memtier but also relate it to histograms.

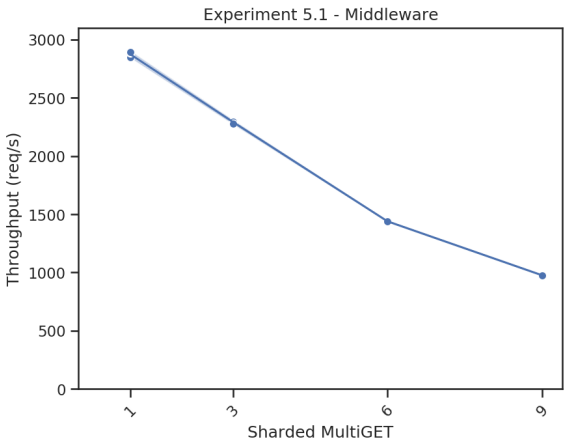


(a) Average response times.

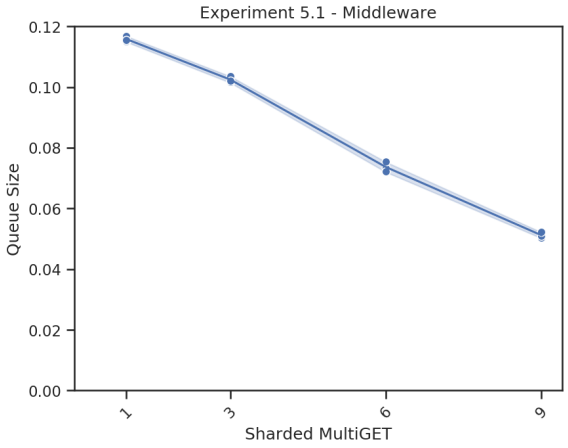


(b) Percentiles of all response times.

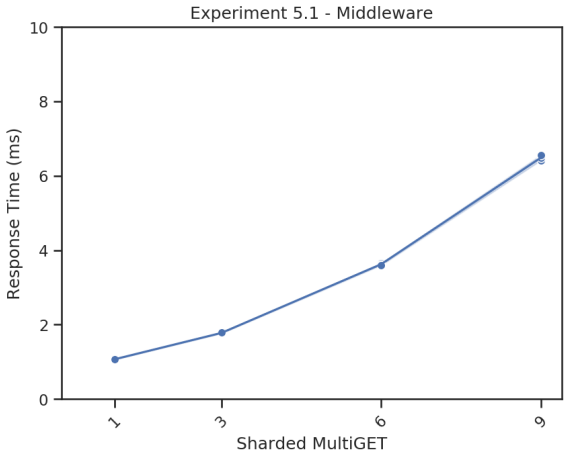
Figure 8 – Response times measured on *Clients*. The graphs include error bars and represent response times for GET packets using sharded behaviour on the *Middleware*.



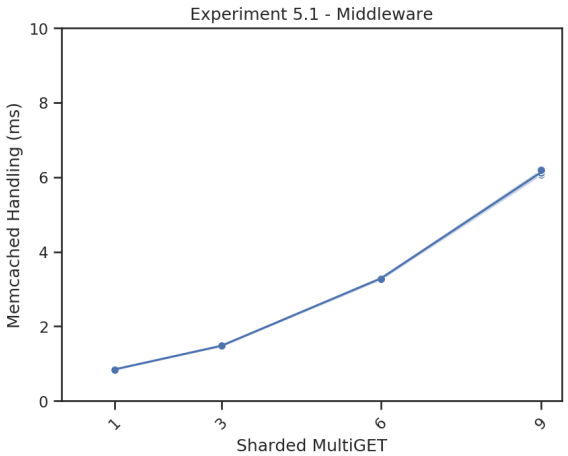
(a) Request throughput.



(c) Queue sizes.

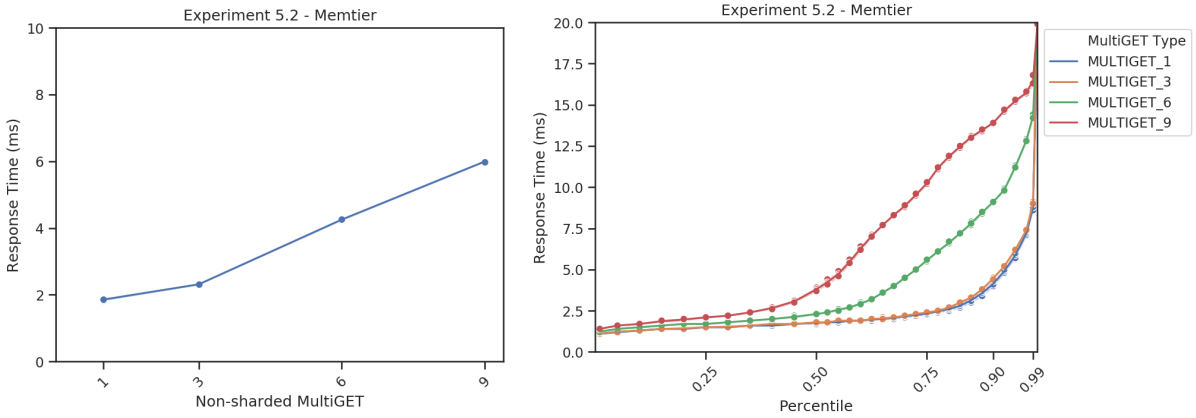


(b) Response times.



(d) Memcached communication time.

Figure 9 – *Middleware* statistics based on GET requests. Requests are defined as consisting of one GET request followed by a variable amount of keys to be requested.



(a) Average response times.

(b) Percentiles of all response times.

Figure 10 – Response times measured on *Clients*. The graphs include error bars and represent response times for GET packets using non-sharded behaviour on the *Middleware*.

5.1 Sharded Case

5.1.1 Explanation

As can be seen in the subfigures of figure 8 the more keys are requested in a single GET request the slower the average response becomes. There is a rough linear behaviour in increasing response times for larger keys which is expected. Also it is of interest to note the stability of the response times; the system looks to be working optimally without outside interference.

It is surprising to observe that the 50-percentile is not representative of the average response time. The 75-percentile represents the value much closer. This signifies that there is a skew in the plot. This is also visible in the histograms obtained for 6-keys (see figure 12b).

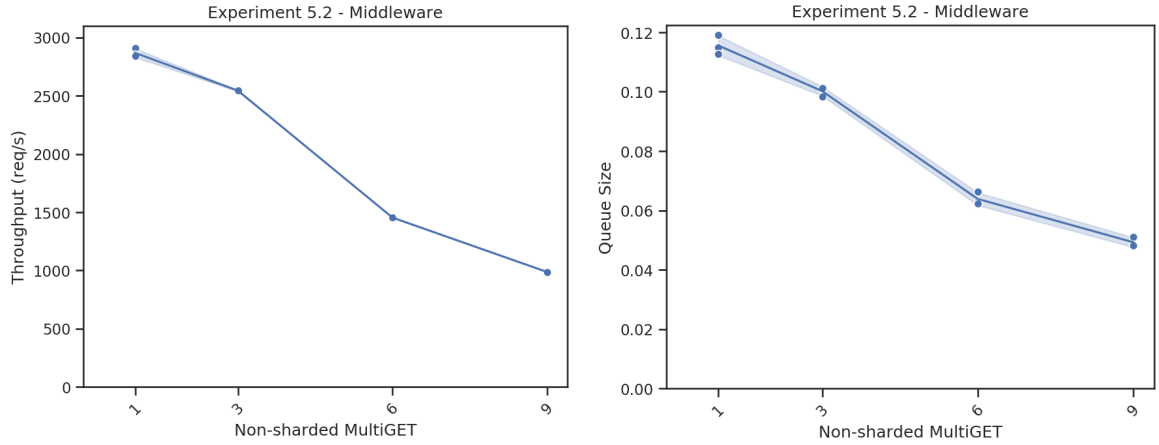
For all requests the queue in the middleware is almost always empty. This shows the middleware performing well in terms of request handling. The hypothesis is therefore the middleware must be spending lots of time on the communication and packet handling with memcached. Figure 9d visualizes the average communication and packet handling time per key size. Comparing the graph with figure 8a the hypothesis is verified. A discussion comparing this behaviour with no sharding follows in subsection 5.4.

The throughput reported by the middleware in figure 9a shows a downwards trend for requests processed. This is expected as even though fewer requests are made, the throughput in keys requested increases. The resulting throughputs in keys are 2879, 6875, 8634 and 8786 keys respectively. The trend downwards is in general hitting a floor due to GET requests being network bound on the side of memcached (cf. table 1). The response time matches quite well with the memcached response times. This indicates that the middleware is easily able to send the requests.

5.2 Non-sharded Case

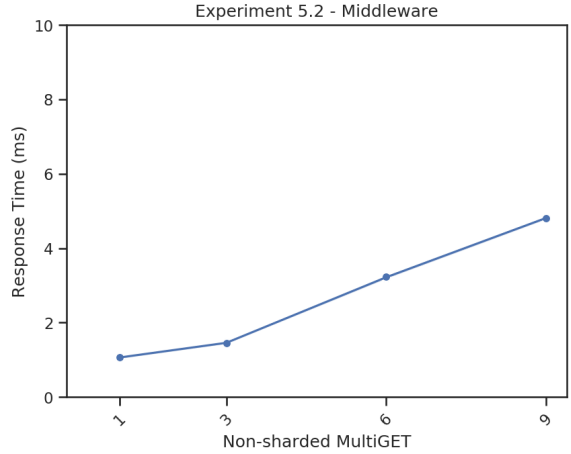
5.2.1 Explanation

The subfigures of figure 10 show similar behaviour to the ones in figure 8. Important differences are the decreased 50-percentile point for mutliGETs of size 9 with higher response times for 6 and 9 keys in the high percentiles. This indicates that sharding helps minimize GET request variance at the cost of a higher average response time. This argument is strengthened when comparing the average response times of figures 10a and 8a. For non-sharding the average response times are therefore lower (about 2 ms for 9 keys and about 1 ms for 6-keys) but higher variances are to

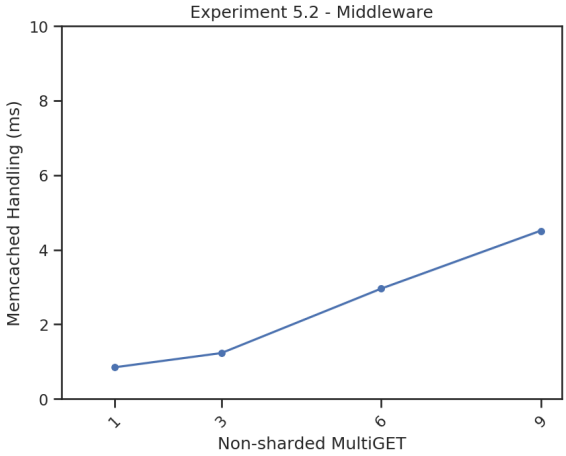


(a) Request throughput.

(c) Queue sizes.



(b) Average response times.

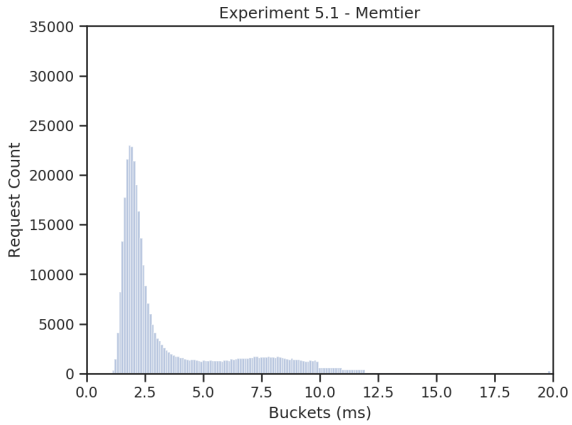


(d) Memcached communication time.

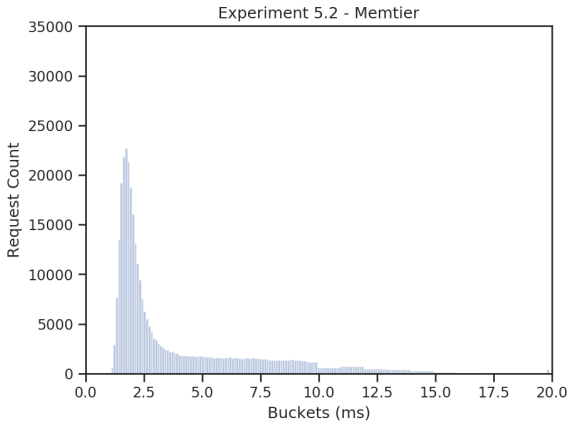
Figure 11 – *Middleware* statistics based on GET requests. Requests are defined as consisting of one GET request followed by a variable amount of keys to be requested.

be expected. For both experiments the response time for keys of size 1 are virtually identical. It is noteworthy to state that the distribution for 3-keys and 1-keys basically overlap. This seems to show a perfect scaling from GETs with 1 key to 3 keys by the system in the case of not sharding.

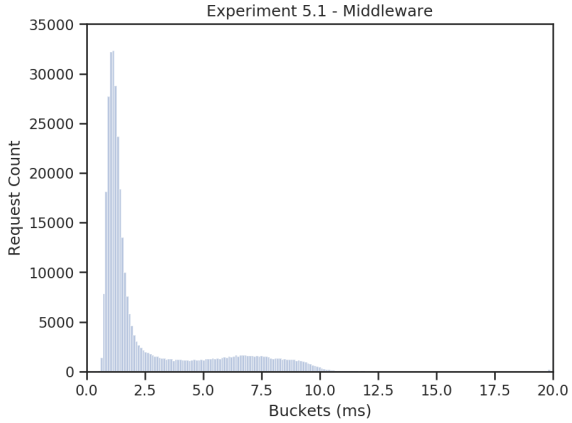
The queue is similarly to the sharded case nearly empty and the middleware spends most of the time communicating with a single *Server* as opposed to multiple ones for GET requests (no sharding occurs). As such only a single connection is used which adds the benefit of being able to receive more contiguous data from a single connection, meaning more stable behaviour once a replies occurs. This behaviour will be compared in subsection 5.4.
The reported throughput has a slightly stronger bent after 3 keys compared to sharded responses but overall the throughputs are comparable between sharded and non-sharded. A closer comparison shows the system performing better for GETs with 3 keys and not using sharding (2543 for non-sharded and 2292 for sharded). The respective keys exchanged are 2868, 7628, 8722, 8887. A higher throughput is achieved compared to the sharded case. Lastly the latency is included which indicates a linear increase in latency from 3 – 9 keys. This trend is reasonable insofar that the network bound is approached and data expected to take longer to send. Again the response time matches quite well with the memcached response times. This indicates that



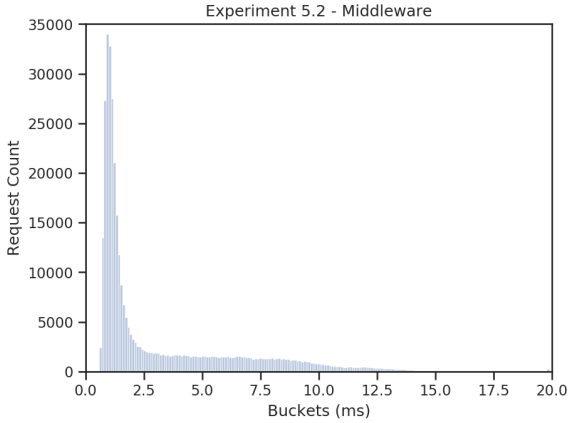
(a) Histogram for sharded multiGET with 6 keys measured on *Client*.



(c) Histogram for non-sharded mutliGET with 6 keys measured on *Client*.



(b) Histogram for sharded multiGET with 6 keys measured on *Middleware*.



(d) Histogram for non-sharded mutliGET with 6 keys measured on *Middleware*.

Figure 12 – Histograms of response times measured for sharded and non-sharded multiGETs of 6 keys. The distribution is based on GET requests and a bucket sizes of 0.1 ms is used, with the last bucket accumulating all values ≥ 19.9 ms. The histogram lacks error bars as the three repetitions' histogram results were extracted and seen as one graph. Thus no error bars are able to be display.

the middleware is easily able to send the requests.

5.3 Histogram

The histograms show consistent behaviour between *Server*s and *Middleware*s; the response times are never smaller for *Server*s than for *Middleware*s and the shape measured being smoother on the middleware than on memtier follows from a constant bucketing of 0.1 ms on the middleware as opposed to memtier which accumulates responses beyond 10 ms into buckets of 1 ms. As such they appear more coarse after 10 ms.

The general shapes are similar between measurements at *Middleware*s and *Client*s for the case of sharded and non-sharded multi-key GET requests. The peaks are much more pronounced on the middleware which makes sense as it is closer to the clients and can process data quicker due to their multithreaded nature. The respective memcached plots share similar peaks; the system can be seen as equally under load in both cases.

Noteworthy in figure 12b is the presence of a small yet measureable second hill between 7.5 ms–10.0 ms. It is also visible on the respective *Client* measurement but doesn't exist for the non-sharded case. This behaviour may be due to servers having a variable response time

and with each sharded request convolving the expected response times from each server. This means there is a coupling of all servers and thus a perturbation towards the tail's end is expected. Due to sharded requests being more evenly distributed the tail must be shorter compared to the non-sharded case. The previous graphs on percentiles reflect well on the distributions displayed. Similar behaviour is observed for the non-sharded graph.

Additionally the histograms show very few outliers in the last bucket. The represented curves are therefore well suited to explain the distribution of response times.

Lastly the previous claim of sharding increasing the response time yet stabilizing the variance shows well in the plots. In general sharded results cut off earlier than the non-sharded counterpart as can be seen when looking at the tails of both experiments. In the non-sharded experiment a long thin tail is observed.

5.4 Summary

The question on how much sharding improves network performance is not easy to answer. Overall the throughput stays mostly comparable for GETs with same key sizes (exclusion being 3-keys).

The only measurable metric that may be of interest is the reduced response time for the non-sharded case. But this only shows strongly for large keys. This seems to indicate that coordinating keys between three *Server*s is more cumbersome than just sending the request to one. Looking at how connection handling and memcached communication is designed on the middleware it becomes clear that distributing the GET requests to the servers is done quickly. Accepting replies and reassembling the responses is the bottleneck. Even though many worker threads exist on the middleware each one handles network communication in a single-threaded approach. As such responses begin to be serialized if they arrive simultaneously on the middleware (the thread can only handle one connection at a time) and introduce the negative side-effect of requiring more state computations and OS-based calls to access data on each of the sockets used for communication. As requests are also guaranteed to be over the size of an MTU and the design of network communication returns once no new data is present in the socket bad interleavings of accessing partially complete data on sockets introduce further overhead. This overhead is not as strongly present for a single connection. Using a single connection introduces the issue of two requests being sent to the same *Server*. As such a larger variation is expected as memcached is run single threaded.

In short it is therefore when expecting averages to be low sensible to not use sharding but enabling it when more consistent response times are requested for increasing key sizes. For 3 keys in a GET the favourite is to not enable sharding.

In both cases there is a measurable increase in throughput for GET as key increase in size. It approaches the upload limit of all three servers. As such the distribution of requests follows a good pattern where the system can show its true performance. It becomes clear that increasing the amount of keys per request is therefore one way to increase performance for GET requests.

The interactive law cannot be simply applied to the throughput numbers obtained in figures 9a and 11a. Calculating the true response time it becomes apparent that the numbers don't match up. This stems from the fact that only half the total requests are displayed and as discussed prior the thinking time is missing. The graph only display GET requests, not both, SET & GET requests. As they are send in a 1:1 ratio though, simply by multiplying the amount of requests for response time calculation gives a very good approximation of the average response time for GETs and SETs.

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

Table 9 – Experimental parameters for 2k analysis.

Sign	Number of *Servers* (**A**)	Number of *Middlewares* (**B**)	Number of worker threads (**C**)
-1	1	1	8
1	3	2	32

Table 10 – Sign table interpretation with respective configuration value.

	Throughput		Response Time	
	Effect	Variation	Effect	Variation
q0	5695	—	44.1	—
qA	2760	98.24%	-21.3	99.73%
qB	173	0.38%	-0.5	0.05%
qC	138	0.25%	-0.4	0.04%
qAB	175	0.40%	-0.5	0.06%
qAC	137	0.24%	-0.4	0.04%
qBC	-136	0.24%	0.4	0.04%
qABC	-136	0.24%	0.4	0.03%
Error	—	0.01%	—	0.00%

(a) 2k3 factors with variation for GET requests.

	Throughput		Response Time	
	Effect	Variation	Effect	Variation
q0	7775	—	26.4	—
qA	-574	7.87%	2.1	9.55%
qB	1149	31.56%	-3.9	32.81%
qC	1530	55.93%	-5.0	54.85%
qAB	171	0.70%	-1.0	2.41%
qAC	-177	0.75%	-0.0	0.00%
qBC	341	2.77%	0.3	0.17%
qABC	115	0.32%	-0.2	0.11%
Error	—	0.09%	—	0.10%

(b) 2k3 factors with variation for SET requests.

Table 11 – 2k3 factor analysis summaries for GET and SET requests. Numbers are rounded for throughput to integers, for latency to a single decimal.

6 2K Analysis (90 pts)

In this section a 2k3 analysis (2k analysis with 3 repetitions) is performed on throughput and response time for experimental parameters listed in table 9 using single keys for GET requests. In a 2k analysis factors are used as parameters to infer system behaviour. Table 10 lists these factors and their interpretation as per sign table (this approach was chosen for the 2k analysis). It is important to note that for a $2^k r$ experiment with $k = 3$ (3 factors observed) and $r = 3$ (three repetitions) the data gets appropriately normalized for the calculation of the q-values (multiplication by $\frac{1}{2^3}$) and the sum of squares (SSQ) is multiplied by $2^3 * 3$ to be able to infer the variation explained.

For 2k analysis two models exist, the additive and multiplicative one. Considering the factors to analyse, it becomes clear that increasing **A** and either **B** or **C**, an additive relationship exists whereas increasing **B** also modifies **C** (but not necessarily vice-versa—a quasi-multiplicative relationship). A mixed model would therefore be optimal under these considerations. Looking at the expected data and including the strength of the multiplicative model for expected values y to exist only for a large enough threshold of $\frac{y_{max}}{y_{min}}$ it is reasonable to evaluate the system with an additive model.

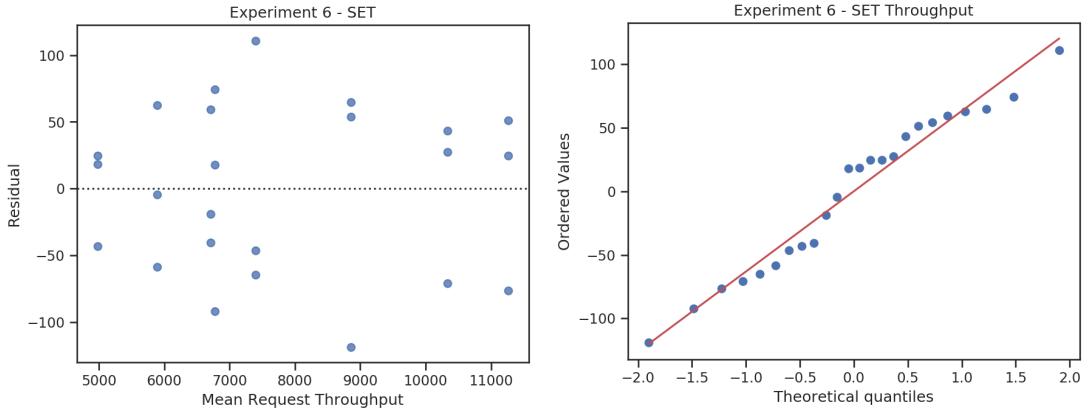
In the case of GET packets the factor **A** is with nearly 100% variational effect for throughput and response time the only relevant factor in determining the system behaviour. Of interesting note is the switch in sign yet including the interpretation that high throughput implies low response times it becomes apparent why a change in sign must happen for corresponding factors. As previously observed we see for three *Client*s and one *Server* immediately a saturation point and the middleware has no measurable effect, either good or bad on the system performance for GET. This matches the results observed. Visualizing the residuals (Figure 13b) and doing a QQ plot (figure 13d) of the measured data for the throughput and response time (the latter omitted for behaving similar to the former) shows very consistent behaviour at either ends of the request throughputs on the residual plot. A few outliers are observed but these cannot be ruled out in a cloud environment. The QQ plot has a very shallow line in the center where the highest likelihood of an observation is for a standard distribution. Both of these plots show an insufficient model according to the book referenced in the lecture (Box 18.1) and a multiplicative model is recommended to be observed. After evaluating that model no reasonable gains were able to be found and the resulting plots are apart from a change of axis labels very similar. Even though the results indicate problems, when including the real system behaviour, both numbers make sense and are not only verified by the 2k3 analysis but also empirically to correlate.

In the case of SET requests a mixture of factors **C** and **B** shows to be mostly significant with around 55.9%/54.9% and 31.6%/32.8% for throughput and response time respectively. Of minor significance yet measurable are **A** with around 7.8%/9.6% for throughput and response time and **BC** for throughput with 2.8% (which interprets as increasing the number of *Middleware*s and worker threads simultaneously). The change in sign occurs again correctly. The analysis aligns with previous observations and conclusions while bringing up the hypothesis that worker threads result in a larger gain in performance compared to adding more *Middleware*s. The hypothesis seems to not hold though as in Experiment 3 we expect for the case of two *Middleware*s and 16 worker threads less performance than for one *Middleware* and 32 worker threads. This is clearly not the case and as such the belief is the model is unable to correctly differentiate between these two parameters (assuming both *Middleware*s have the exact same throughput and response time behaviour). The factor **A**, adding more *Server*s to the system, does align with results from experiments 3 and 4 where the number of *Server*s changes from one to three and a decrease in throughput is observed. The factor proposes an increase in *Server*s introduces a negative effect as the *Middleware* is designed to share the SET operation with each *Server* in the system. This key distribution introduces additional latencies into the system. Again residual (figure 13a) and QQ plots (figure 13c) were created but only the throughput ones are presented (similar behaviour in either case). The residuals are scattered without anything being possible to interpret and the QQ plot shows a quasi-linear behaviour. Both great signs according to the book as the residual error must be IID and small (it is smaller by a magnitude than the x-labels) and the distribution of errors being uniform at random. As such the model should be good enough to describe system performance per the book. Given empirical data the trend is clearly visible and overlaps but the strength of individual components is questioned to be correct.

7 Queuing Model (90 pts)

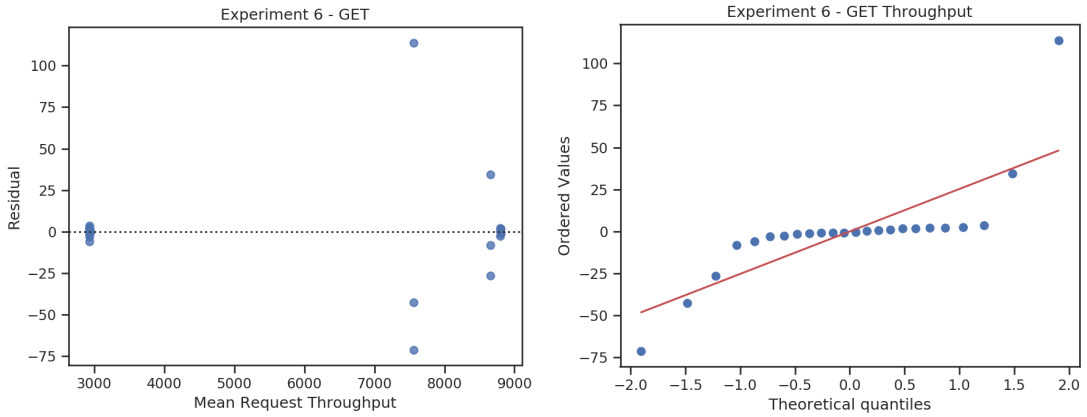
The last section concentrates on theoretical foundations. We apply different modelling techniques and observe how the real system's results differ from computed ones. Explanations for discrepancies and overlaps between computed values and measured results are given.

This section includes the M/M/1 and M/M/m model (which use data from experiment 4 as the basis) as well as a network of queues which tries to attain results for configurations of experiment 3.



(a) Residual plot for SET Throughput.

(c) QQ-Plot for SET Throughput.



(b) Residual plot for GET Throughput.

(d) QQ-Plot for GET Throughput.

Figure 13 – Residual and QQ plots based on memtier measurements for experiment 6.0. Only throughput plots are depicted.

For M/M/1 and M/M/m models the following parameters are part of the system.

Input Parameters:

- μ: The service rate of the system. It is calculated from the maximally observed average throughput from any single run for a given amount of worker threads in experiment 4 disregarding the response time reported. This number is divided by twice the number of worker threads for M/M/m models. The actual value used is documented in the respective table.
- λ: The arrival rate of jobs. It is taken directly from observed results of the average throughput for both middlewares for all repetitions of an experiment. This number is included in the table as well.
- m: The number of services. It is only relevant for the M/M/m models. The service count is double the amount of worker threads configured as two *Middleware*s are used.

Output Parameters:

- ρ: The traffic intensity. For the M/M/m model this is equivalent to U, the average utilization of each server.
- $\mathbb{E}[n]$: Expected number of jobs in the system.
- $\mathbb{E}[n_q]$: Expected number of jobs in the queue.

– $\mathbb{E}[w]$: Expected waiting time for elements in the queue.
– $\mathbb{E}[r]$: Expected response time of the system.
– p_0: Probability of zero jobs in the system. Only relevant for the M/M/m model.
– ϱ: Probability of queueing. Only relevant for the M/M/m model.

Choosing the service rate and arrival rate as described gives for all following calculations the guarantee that $\rho < 1$ holds as the averaged throughput for all repetitions is expected to be lower than any maximally observed value. This is a stability parameter which needs to hold for calculations to be valid as the system would otherwise not be fulfilling the requirements of an M/M/1 or M/M/m model.

7.1 M/M/1

The M/M/1 model is a simple model which is based on the idea of only containing a single queue and service. Jobs to this system arrive with a mean arrival rate of λ and are processed by the service with the mean service rate μ.

In table 12 the computed results for all combinations of worker threads with clients are presented for the M/M/m model. Comparing the received numbers with measured data it becomes clear this model is insufficient to describe the real system's behaviour.

The following observations are made:

1. The trend to a decreased response time for more worker threads to various amounts of utilizations matches for both, the M/M/1 model and the real system. It can be observed that the model predicts response times more to either extreme, either too low or too high yet a reasonable amount of overlapping cases exist. All of them are in the area of 99% utilization. This shows a very narrow region of prediction and thusly is not a great model to predict real system behaviour. The response time the model predicts is compared with the response time of the middleware to a request sent out by memtier.
2. The queue times are in direct correlation to the response times in this model and as such coupled together. This shows in the model predictions where the queue times are in many cases a couple hundred microseconds quicker than the respective response time. This estimate is not valid in the real model as network communication is not modelled. The real system must not only process the packet it must also distribute the SET requests to each *Server* connected, wait for replies and then send back the response to memtier. This is simply not modelled by M/M/1.
3. The queue sizes show also a trend towards smaller sizes for more clients with increasing amounts of worker threads. Queues are mostly modelled too large compared with the real system and therefore overestimate the amount of queueing. This is expected as a single queue is used by the model. As such the observation of the number of elements in the system being larger by at most one element compared to the number of queues can be explained.
   The trend is stems from the model's formulae. The queue size is only determined by ρ and as such actually independent of the number of clients and worker threads. This also shows in some predictions such as 192 and 288 clients for 32 worker threads having larger queues than 16 and 8 worker threads.
4. As can be seen, the estimated number of jobs is always at most one larger than the queue size. This is a fragment of the model's base design and is clearly not reflected by the real system's behaviour where multiple workers are active.

The model is at most a basic approximation as it is a sequential, single queue with a single service actor whereas the middleware is a model of multiple concurrent services with two queues.

					$\mathbb{E}[n_q]$		$\mathbb{E}[w]$		$\mathbb{E}[r]$		
Clients	WT	μ	λ	ρ	Est.	Act.	Est.	Act.	Est.	Act.	$\mathbb{E}[n]$
6	8	6750.87	2840.73	0.42	0.31	0.09	0.11	0.06	0.26	1.31	0.73
	16	8716.58	2841.71	0.33	0.16	0.08	0.06	0.06	0.17	1.32	0.48
	32	10385.53	2804.17	0.27	0.10	0.09	0.04	0.06	0.13	1.34	0.37
	64	11582.18	2795.49	0.24	0.08	0.09	0.03	0.07	0.11	1.38	0.32
12	8	6750.87	4967.22	0.74	2.05	0.21	0.41	0.08	0.56	1.62	2.78
	16	8716.58	4912.86	0.56	0.73	0.21	0.15	0.08	0.26	1.64	1.29
	32	10385.53	4916.40	0.47	0.43	0.21	0.09	0.08	0.18	1.64	0.90
	64	11582.18	4838.18	0.42	0.30	0.21	0.06	0.09	0.15	1.69	0.72
24	8	6750.87	5955.95	0.88	6.61	2.12	1.11	0.71	1.26	3.13	7.49
	16	8716.58	6153.06	0.71	1.69	0.48	0.28	0.16	0.39	3.00	2.40
	32	10385.53	6152.46	0.59	0.86	0.46	0.14	0.15	0.24	3.01	1.45
	64	11582.18	6065.54	0.52	0.58	0.47	0.09	0.15	0.18	3.06	1.10
48	8	6750.87	6680.07	0.99	93.37	12.72	13.98	3.81	14.12	6.25	94.36
	16	8716.58	8187.97	0.94	14.55	4.80	1.78	1.17	1.89	4.81	15.49
	32	10385.53	8378.60	0.81	3.37	1.24	0.40	0.30	0.50	4.65	4.17
	64	11582.18	8169.17	0.71	1.69	1.29	0.21	0.32	0.29	4.77	2.39
96	8	6750.87	6664.63	0.99	76.29	36.64	11.45	11.01	11.60	13.45	77.28
	16	8716.58	8616.11	0.99	84.77	27.00	9.84	6.27	9.95	10.03	85.76
	32	10385.53	10010.38	0.96	25.72	10.18	2.57	2.03	2.67	8.09	26.68
	64	11582.18	9935.81	0.86	5.18	3.54	0.52	0.72	0.61	7.87	6.03
192	8	6750.87	6690.24	0.99	109.35	84.64	16.35	25.32	16.49	27.76	110.34
	16	8716.58	8626.25	0.99	94.51	74.92	10.96	17.38	11.07	21.13	95.50
	32	10385.53	10327.79	0.99	177.87	55.18	17.22	10.69	17.32	16.93	178.87
	64	11582.18	11251.22	0.97	33.02	19.76	2.94	3.51	3.02	14.20	34.00
288	8	6750.87	6721.23	1.00	225.78	132.57	33.59	39.48	33.74	41.90	226.78
	16	8716.58	8681.94	1.00	249.59	122.78	28.75	28.31	28.86	32.04	250.58
	32	10385.53	10366.39	1.00	540.53	103.15	52.14	19.91	52.24	26.13	541.52
	64	11582.18	11469.24	0.99	100.56	61.54	8.77	10.73	8.85	21.87	101.55

Table 12 – M/M/1 calculations for given configurations of Experiment 4 using the formulae listed in the book, Box 31.1. Numbers are rounded to two decimal places for presentation purposes which for the case of ρ makes the system seem unstable yet the actual numbers are < 1.

7.2 M/M/m

The M/M/m model is an extension to the M/M/1 model which still keeps the single queue but has m services acting on the queue. These services will be modelled by the amount of total worker threads in the system. With 2 *Middlewares* m is set to double the value of worker threads (and not explicitly documented in the table).

In table 13 the computed results for all combinations of worker threads with clients are presented for the M/M/m model. Comparing the received numbers with measured data it becomes clear this model is still insufficient to describe the real system's behaviour.

The following observations are made:

1. The response times in this model behave for low utilizations counter-intuitive where low utilization predicts for many workers a higher response time yet this trend not showing for high utilization. This is due to the fact that the workers amongst different configurations don't share the same service time. The service time for few workers is lower than for many workers (or as noted in the table many workers have fewer throughput per worker). It must be remembered that the parameter μ comes from the real system and matches constraints on the real system. With the amount of threading that is close to the actual

physical cores a better utilization is expected whereas for too much threading the system is mostly busy with scheduling overhead and other system maintenance. This reflects in higher throughput per thread. As the model numbers were obtained for the maximum throughput per configuration it is expected that model estimates for low utilization don't match but match much better for high amounts of utilization as can be inferred from the table. The model therefore "adapts" much better to the real system when they both converge in behaviour. It is of interesting note to see the predictions in many cases being below real system measurements. This also matches for the next parameter evaluated, the queue waiting time.

2. This trend also follows for the queue waiting times and gives a much greater delta for estimated queue waiting times and response times. This is much more reasonable and begins to closer match the system behaviour as well for high utilization. For low degrees of utilization the models shows good degrees of approximation with the true system.
3. The queue waiting times need to be evaluated with the constraints that the estimate is halved when compared to the real system as the model assumes one queue but the real system uses two. Even with this constraint the model varies too much to be able to make a statement of it agreeing with the real system. There are definitely overlaps but no definite pattern exists. As in the M/M/1 model this parameter is mostly inferred by the system utilization, the probability of queueing being an additional scaling factor.
4. The probability of queueing goes up for more load (which is expected). The probability seem low compared to the actually observed queue-sizes for experiment 4, meaning queuing expectations should be converging earlier to 1 as queueing is definitely observed. Yet this would give a queue size estimation to the M/M/1 model, not what is reasonable.
5. The probability of 0 jobs in the system is 0 for all experiments. This matches the expected reality.
6. The elements in the system still correlate with the queue size and as has been discussed a reasonable statement cannot be made on the numbers obtained as for small workloads with high threading the number of expected jobs is too high.

Overall the model is better but not in the general case as it may be parallel but still it uses a single queue and the configuration is highly dependent on the service time given (it is fixed and doesn't align for low system utilization inputs). The middleware uses multiple concurrent services (which adapt to a ceiling of the worker thread configuration) but uses two, instead of one queue. Additionally this system merges the worker threads and memcached into one virtual service, clearly not the actual system behaviour.

7.3 Network of Queues

In the following, two network of queue designs are constructed to try and model results obtained on experiments 3.1 and 3.2. The designs include 3 queues and 2 latency centers (which emulate the network where applicable).

A visualization of the network of queues for 2 *Middleware*s can be seen in figure 14 with the difference to model 1 being the existence of only 1 *Middleware* instead of two for placement of components. With two *Middleware*s the requests are split up between both in an even fashion such that both middlewares experience only half the throughput.
As can be seen memcached is modelled with a queue, more specifically using an M/M/1 approach as this reflects the experimental setup. Instances of memtier are assumed to be queue-less and their performance being infinite. This matches with experimental setups where memtier was not the cause for slowdown (cf. experiment 2). The middleware is modelled with 2 queues, an M/M/1 queue which emulates the single network thread followed by an M/M/m queue which

					$\mathbb{E}[n_q]$		$\mathbb{E}[w]$		$\mathbb{E}[r]$					
Clients	WT	μ	λ	ρ	Est.	Act.	Est.	Act.	Est.	Act.	$\mathbb{E}[n]$	ϱ	p_0	U
6	8	421.93	2840.73	0.42	0.00	0.09	0.00	0.06	2.37	1.31	6.73	0.00	0.0	0.42
	16	272.39	2841.71	0.33	0.00	0.08	0.00	0.06	3.67	1.32	10.43	0.00	0.0	0.33
	32	162.27	2804.17	0.27	0.00	0.09	0.00	0.06	6.16	1.34	17.28	0.00	0.0	0.27
	64	90.49	2795.49	0.24	0.00	0.09	0.00	0.07	11.05	1.38	30.89	0.00	0.0	0.24
12	8	421.93	4967.22	0.74	0.50	0.21	0.10	0.08	2.47	1.62	12.28	0.18	0.0	0.74
	16	272.39	4912.86	0.56	0.00	0.21	0.00	0.08	3.67	1.64	18.04	0.00	0.0	0.56
	32	162.27	4916.40	0.47	0.00	0.21	0.00	0.08	6.16	1.64	30.30	0.00	0.0	0.47
	64	90.49	4838.18	0.42	0.00	0.21	0.00	0.09	11.05	1.69	53.47	0.00	0.0	0.42
24	8	421.93	5955.95	0.88	3.98	2.12	0.67	0.71	3.04	3.13	18.10	0.53	0.0	0.88
	16	272.39	6153.06	0.71	0.10	0.48	0.02	0.16	3.69	3.00	22.69	0.04	0.0	0.71
	32	162.27	6152.46	0.59	0.00	0.46	0.00	0.15	6.16	3.01	37.91	0.00	0.0	0.59
	64	90.49	6065.54	0.52	0.00	0.47	0.00	0.15	11.05	3.06	67.03	0.00	0.0	0.52
48	8	421.93	6680.07	0.99	89.76	12.72	13.44	3.81	15.81	6.25	105.60	0.95	0.0	0.99
	16	272.39	8187.97	0.94	9.91	4.80	1.21	1.17	4.88	4.81	39.97	0.64	0.0	0.94
	32	162.27	8378.60	0.81	0.27	1.24	0.03	0.30	6.19	4.65	51.90	0.06	0.0	0.81
	64	90.49	8169.17	0.71	0.00	1.29	0.00	0.32	11.05	4.77	90.28	0.00	0.0	0.71
96	8	421.93	6664.63	0.99	72.71	36.64	10.91	11.01	13.28	13.45	88.51	0.94	0.0	0.99
	16	272.39	8616.11	0.99	79.22	27.00	9.19	6.27	12.87	10.03	110.85	0.92	0.0	0.99
	32	162.27	10010.38	0.96	18.36	10.18	1.83	2.03	8.00	8.09	80.05	0.69	0.0	0.96
	64	90.49	9935.81	0.86	0.35	3.54	0.04	0.72	11.09	7.87	110.16	0.06	0.0	0.86
192	8	421.93	6690.24	0.99	105.74	84.64	15.80	25.32	18.17	27.76	121.59	0.96	0.0	0.99
	16	272.39	8626.25	0.99	88.93	74.92	10.31	17.38	13.98	21.13	120.60	0.93	0.0	0.99
	32	162.27	10327.79	0.99	169.38	55.18	16.40	10.69	22.56	16.93	233.03	0.95	0.0	0.99
	64	90.49	11251.22	0.97	22.26	19.76	1.98	3.51	13.03	14.20	146.60	0.65	0.0	0.97
288	8	421.93	6721.23	1.00	222.12	132.57	33.05	39.48	35.42	41.90	238.05	0.98	0.0	1.00
	16	272.39	8681.94	1.00	243.89	122.78	28.09	28.31	31.76	32.04	275.76	0.97	0.0	1.00
	32	162.27	10366.39	1.00	531.89	103.15	51.31	19.91	57.47	26.13	595.77	0.98	0.0	1.00
	64	90.49	11469.24	0.99	88.44	61.54	7.71	10.73	18.76	21.87	215.19	0.87	0.0	0.99

Table 13 – M/M/m calculations for given configurations of Experiment 4 using the formulae listed in the book, Box 31.2. Numbers are rounded to two decimal places for presentation purposes which for the case of ρ makes the system seem unstable yet the actual numbers are < 1.

simulates the workers. The approach to model the network thread with an M/M/1 queue follows from real system behaviour where packets queue up on a network card if they are not processed quick enough. The M/M/m queue for the workers reflects the design of the middleware in which a single queue is used amongst workers. The decision to leave out modelling the reply stems from the fact that the correct pathway would involve the path back from memcached to each worker thread that communicated with it and then have it reply back to memtier. With the design not allowing bidirectional flows through service centers this cannot be correctly modelled.

The software package `queueing` from *GNU Octave* is used to model the previously designed networks using Mean-Value Analysis. The parameters for both networks are defined as follows:

- n: The number of requests in the system. This is equal to the amount of currently active clients.
- S: The average service time per actor. The actors are defined as follows for the network of queues:
  - S_{client}: It is set to 0 as we assume *Client*s to never be the bottleneck.
  - $S_{network}$: The latency of the network. As a reasonable approach results from ping are taken and halved. This leads to unreasonably low throughput for few clients as

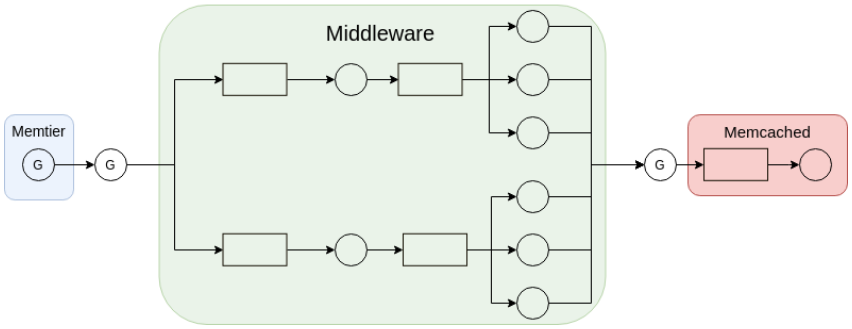


Figure 14 – Design of the network of queues. All rectangles define queues, circles delay centers. Circles with a "G" define delay centers which allow utilizations above 1 as they model the network.

has been inferred by trial and error and set to 0.3 ms.

- $S_{netthread}$: The service time of the network thread. This is inferred from the maximum throughput observed in SET requests for the configurations of one and two *Middleware*s. For the latter case this number is also halved as two instances exist and on average either has only experienced half the workload.
- S_{worker}: The service time per worker. It is inferred for each worker thread and request type separately to model the actual system behaviour whereby worker threads have more sending to do for SET compared to GET requests. It is derived by taking the response time of the whole system and subtracting the queuing time and memcached communication time.
- S_{server}: The service time for memcached. This is inferred from the maximum throughput per request type as GETs are network bound whereas SETs are CPU bound. For all analysis with SET this is $\frac{1}{16335}$ (from experiment 2.1) and for all analysis with GET this is $\frac{1}{2940}$ (from experiment 3.1 as the throughput is slightly higher compared to experiment 2.1).

– V: The visit ratios from one service center to the next. It is in general set to 1 but for 2 *Middleware*s the ratio between the network delay center and each M/M/1 queue is halved to model the experimental parameters correctly.

– m: The number of identical servers for each node. This is only relevant for the M/M/m queues which model multiple concurrent worker threads. In contrast to the previous M/M/m model the number is set to actual amount of worker threads.

We observe the results of the MVA in terms of throughput (X), response times (R), queue sizes (Q) and system utilization (U).

7.3.1 One *Middleware*

For this analysis the following fixed parameters were chosen: $S_{network} = 0.3\,\text{ms}$, $S_{netthread} = \frac{1}{11546}$, $S_{worker\_GET} = 1.094\,205\,\text{ms}$, $S_{worker\_SET} = 0.172\,963\,\text{ms}$, $m = 64$.
The bottlenecks are expected to be memcached for GET requests and the Net-Thread for SET requests. For GET requests it has already been determined that a single *Server* is bottlenecking the system and needs no further proof. The utilizations expected align with expectations for memcached but the worker threads are unrealistically low taxed. It looks as if they are "sleeping" for the most time. This would be correct with the model but the real behaviour of worker threads waiting for replies is impossible to model and as such cannot be included in the analysis. It is expected that worker threads are in general expected to under-perform. The response times match general trends but for memcached response times the numbers don't

# MW	Type	Parameter	m	Delay Center	Net-Thread	Worker Threads		Memcached	
						Act.	MVA	Act.	MVA
1	GET	U	6	0.69	0.20	—	0.04	—	0.79
			24	0.88	0.25	—	0.05	—	1.00
			192	0.88	0.25	—	0.05	—	1.00
		R	6	0.30	0.10	0.09	1.09	1.00	0.80
			24	0.30	0.12	0.20	1.09	6.50	6.35
			192	0.30	0.12	1.09	1.09	20.91	63.50
		Q	6	0.69	0.24	0.19	2.52	—	1.84
			24	0.88	0.34	1.03	3.21	—	18.68
			192	0.88	0.34	124.11	3.21	—	186.68
		X	6		MVA: 2311.40 / Measured: 2794.72				
			24		MVA: 2940.00 / Measured: 2938.86				
			192		MVA: 2940.00 / Measured: 2939.61				
	SET	U	6	1.80	0.52	—	0.02	—	0.37
			24	3.45	0.99	—	0.03	—	0.70
			192	3.46	1.00	—	0.03	—	0.71
		R	6	0.30	0.14	0.10	0.17	1.16	0.09
			24	0.30	1.11	0.13	0.17	1.66	0.20
			192	0.30	15.64	0.17	0.17	4.10	0.21
		Q	6	1.80	0.86	0.17	1.04	—	0.52
			24	3.45	12.78	1.35	1.99	—	2.32
			192	3.46	180.67	43.02	2.00	—	2.41
		X	6		MVA: 5985.93 / Measured: 2778.38				
			24		MVA: 11511.69 / Measured: 7177.16				
			192		MVA: 11545.00 / Measured: 11545.60				
2	GET	U	6	0.84	0.18	—	0.06	—	0.95
			24	0.88	0.19	—	0.06	—	1.00
			192	0.88	0.19	—	0.06	—	1.00
		R	6	0.30	0.16	0.09	0.26	1.04	1.14
			24	0.30	0.16	0.12	0.26	8.12	7.14
			192	0.30	0.16	0.26	0.26	48.16	64.28
		Q	6	0.84	0.22	0.08	0.36	—	3.18
			24	0.88	0.24	0.21	0.38	—	21.01
			192	0.88	0.24	30.14	0.38	—	189.01
		X	6		MVA: 2784.95 / Measured: 2936.91				
			24		MVA: 2940.00 / Measured: 2934.91				
			192		MVA: 2940.00 / Measured: 2929.16				
	SET	U	6	1.82	0.40	—	0.00	—	0.37
			24	4.08	0.89	—	0.01	—	0.83
			192	4.56	0.99	—	0.01	—	0.93
		R	6	0.30	0.19	0.10	0.11	0.83	0.08
			24	0.30	0.77	0.11	0.11	1.94	0.29
			192	0.30	11.01	0.11	0.11	8.17	0.89
		Q	6	1.82	0.58	0.09	0.33	—	0.53
			24	4.08	5.20	0.37	0.75	—	3.93
			192	4.57	83.84	23.08	0.84	—	13.51
		X	6		MVA: 6081.54 / Measured: 3272.89				
			24		MVA: 13609.05 / Measured: 8186.25				
			192		MVA: 15218.43 / Measured: 15312.49				

Table 14 – MVA analysis for one and two *Middleware*s at select clients. The intervals show increasingly saturated systems and as such are of interest in the presentation of the model. As the delay center behaves the same before or after the middleware only one center's results are listed. For the experiments with two *Middleware*s the average is reported.

match for 192 clients. This is expected when including the queue sizes. The queue sizes are incorrectly distributed. Memcached is supposedly having 186 requests in the M/M/1 queue for 196 clients, something that is a violation of the system design as each request sent expects a reply. With the system being limited to 64 worker threads at most 64 requests can buffer on memcached in the actual system at any time. This is another flaw of the model. Most requests are proven to be caught in the work-queue because worker threads must wait for memcached replies. The throughput numbers are unexpectedly low for six clients but quickly plateau towards the maximum throughput of memcached (which reflects in maximal utilization).
SET requests indicate a high utilization of the Net-Thread with memcached increasing in utility for more clients. This is a reflection on the model definition and as such only noteworthy to mention but cannot be further elaborated on. As assumed, the utilization of worker threads is assumed too low. The response times show a clear delay being caused by the Net-Thread but such an observation was not made in the system. The model shows memcached replying much quicker than is the case, likely another issue of assuming perfect scaling in the system (where more load doesn't introduce any overhead). The queues are interesting in that the worker threads have virtually no queueing in the model. The Net-Thread is modelled to be the point of queueing. This is a contradiction to previously gathered data where queuing was observed for worker threads. Again the argument of threads "sleeping" in the model can be made but actually worker-threads are busy for longer than just their service time (which is non-trivial to model). The queue in the Net-Thread could be an indicator though that there is a large amount of work incoming and can be the reason why the throughput isn't higher for the real system. Lastly the throughput is predicted in all instances too high (where model limits are not reached). Again the incorrect processing of requests by the model compared to the system are at fault.

7.3.2 Two *Middleware*s

For this analysis the following fixed parameters were chosen: $S_{network} = 0.3\,\text{ms}$, $S_{netthread} = \frac{2}{15310}$, $S_{worker\_GET} = 0.256\,949\,\text{ms}$, $S_{worker\_SET} = 0.109\,886\,\text{ms}$, $m = 64$.
Again the bottlenecks are predicted to be memcached for GET requests and the Net-Thread for SET requests. For GET requests the utilizations are reasonably predicted (excluding worker threads) with the Net-Thread experiencing higher loads. The response times align much better for this model with memcached actual and predicted values being close enough (for high loads still wrong results are calculated) and the worker thread response time being stable. The queues are still incorrectly inferred by the model and have been previously explained. It is noteworthy to mention the queue size predicted for memcached being comparable to the single *Middleware* case. Lastly the throughput matches much closer which can be explained by spreading the load over two *Middleware*s. This reflects in utilization values. The Net-Thread is less taxed and the workers show on average more utilization.
For SET requests the utilization shows the Net-Thread still to be the major bottleneck but memcached following close after. This is expected as both middlewares are able to achieve throughputs very close to the maximum which can be handled by one *Server*. The utility of the workers is even lower than in the single *Middleware* model. This is explained by the fact that each worker receives half the amount of work, meaning they are "sleeping" even longer. The response times are still incorrectly predicted for SET requests and the Net-Thread is expected to bottleneck the system. The only stable component for response times is the response time of the workers which is comparable throughout. The queue sizes are again incorrectly placed in the system with the Net-Threads queue filling up despite being able to handle the expected amount of requests. Compared to the single *Middleware* case roughly half the elements are only predicted. This matches with the model differences (1 queue vs 2 queues). The throughput is

still estimated incorrectly and stems from the aforementioned fact of worker threads processing data much quicker than happens in reality where a feedback loop exists with memcached.

7.3.3 Conclusion

Before drawing conclusions a note on response times predicted. For the case of GET requests these are very close to the expected response time of the real system whereas the Net-Thread models the true response time quite closely for SET requests.

To summarise, the network of queues models has shown improvements over M/M/1 and M/M/m models yet issues still exist:

1. It becomes increasingly complex to model queues correctly where feedback loops exist (such as worker threads waiting for memcached results or worker threads replying to memtier once a reply is received).
2. The model is only accurate if the resource given scale linearly. Usually hardware performance is not being able to be modelled linearly, especially not complete systems which are subject to a collection of unknowns.