Advanced Systems Lab Report

Autumn Semester 2017

Name: Tim Linggi

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

1 System Overview

1.1 Overview

The goal of this project is to develop a middleware which acts between load generating *memtier*[1] clients and *memcached*[2] servers. *Memcached* is a widely used key-value store, which keeps data in main memory, whereas *memtier* is a load generator and benchmarking tool for, among other systems, *memcached*. The middleware load balances and replicates requests from many *memtier* clients across multiple *memcached* servers.

Within the middleware, as illustrated in Figure 1, a single network thread handles incoming requests, parses them and puts them into a request queue. From thereon, one (of possibly many) worker threads processes the request, sends its to one or more *memcached* servers (depending on the type of request) and relays the answer directly back to the client without involving the network thread. A more detailed description of all components can be found in the following subsections. The middleware is abstracted by the class `Middleware`[3].

The middleware supports three types of request. For storing a single value, the `SET` command is used, whereas for retrieving a single value, the `GET` command is used. It is also possible to query multiple values at once, by using the `MULTI-GET` command. There are two modes for `MULTI-GET`: *Non-sharded* mode means that a `MULTI-GET` request is sent to a single server, while in *sharded* mode, a `MULTI-GET`is split into smaller `MULTI-GET`s, one for each server. The mode is specified as an argument on startup of the middleware. More details are provided in Section 1.7.

Collecting basic statistics like throughput and latency can be done easily with *memtier*. However, for a more in-depth insight into the impact of tuning different parameters (like the number of worker threads or the `MULTI-GET` mode), one has to instrument the middleware to be able to gather various metrics, which are described in Section 1.9.

To test the middleware locally, various tests[4] were written, which can be executed with locally running *memcached* servers. They test the parser and the entire middleware as a blackbox.

1.2 Protocol

The protocol used by the *memcached* servers is specified on their Github project[5]. The middleware supports a subset of this protocol, namely the *set* and *get/gets* commands as well as their possible answers. The next few lines give a brief overview about the protocol's subset used, a more involved description can be found on Github. The following commands are supported:

1. `set <key> <flags> <exptime> <bytes> <noreply>?  CR LF`
2. `STORED CR LF`
3. `get <key>+ CR LF`
4. `gets <key>+ CR LF`
5. `(VALUE <key> <flags> <bytes> CR LF`
   `<data block> CR LF)*`
   `END CR LF`

[1] `https://github.com/RedisLabs/memtier_benchmark/`
[2] `https://github.com/memcached/memcached`
[3] ch.ethz.asltest.middleware.Middleware
[4] src/test
[5] `https://github.com/memcached/memcached/blob/master/doc/protocol.txt`

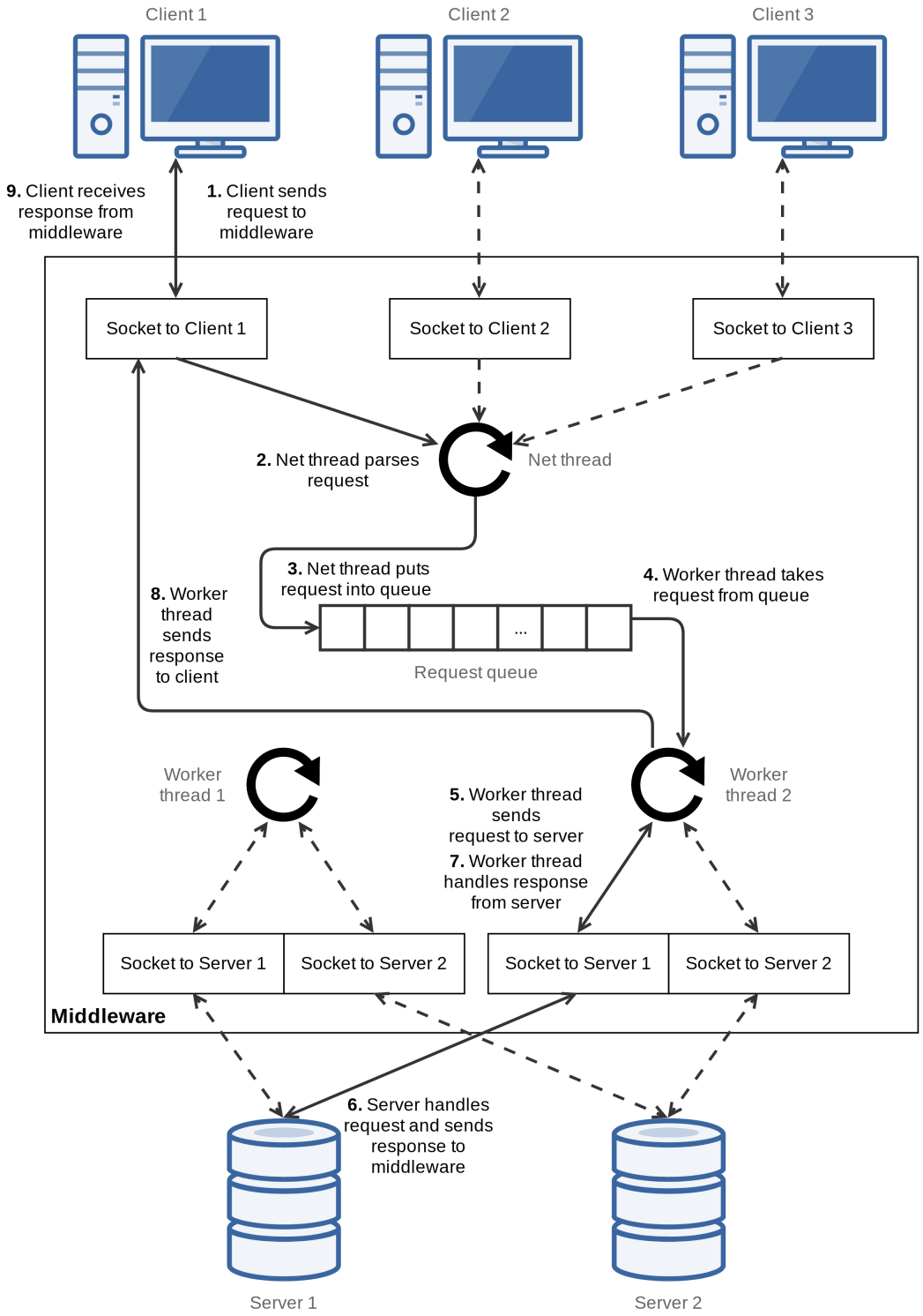


Figure 1: System overview depicted with three client machines, two worker threads and two servers. The solid lines correspond to the path of a single request (marked from 1. to 9.) from Client 1 which is handled by Worker thread 2 and Server 1. The dashed lines correspond to other connections.

6. `<error_string> CR LF`

Item 1 is a `SET` command, and Item 2 is the response of the server, if a `SET` was executed successfully. The `noreply` flag is optional and not used in the middleware. Items 3 and 4 are equivalent `GET` commands which can take one or more keys at once, similarly to Item 5 which can contain multiple `VALUE` blocks at once. However it can also be empty, if no value was found. The error String in Item 6 can have the values `ERROR`, `CLIENT_ERROR` and `SERVER_ERROR`.

All messages which do not belong to the ones listed above, are treated as invalid and handled accordingly.

1.3 Network thread

The network thread[6] ("net thread" for short) handles incoming connections and requests from the clients. The be able to simultaneously handle many connections in a non-blocking fashion, the net thread uses the java.nio package [7]. It is based on the classes `ServerSocketChannel` and `SocketChannel` which correspond to `ServerSocket` and `Socket` in the java.net package [8]. One difference between the two packages is that java.nio supports non-blocking connections, which is essential for reading from multiple clients at once with only one server socket. A `ServerSocketChannel` is used by a server from which a connecting clients' `SocketChannel` can be retrieved, which is then used for communication by reading and writing to and from a `ByteBuffer`. Additionally, a `Selector` is used which acts a multiplexer for the socket channels of different clients.

The middleware acts as a server to the clients and the net thread therefore binds to a `ServerSocketChannel` with the ip and port provided as arguments on startup of the middleware. Initially, the net thread only accepts connections. However, once a client wants to open a connection, its `SocketChannel` and a `ByteArrayOutputStream`[9] (which is basically a variably sized byte array) are saved for later reference and the selector is registered be able to read from this client.

The net thread loops until interrupted (see Section 1.8), always listening to incoming *accept* or *read* operations. An *accept* operation indicates that a new client wants to connect to the middleware and a *read* operation indicates that an already connected client sends bytes. The received bytes are appended to the client's byte stream, parsed and, if parsing is successful, the request is put into the request queue.

To be able to discard invalid requests (as described in Section 1.2) and handle interleaving requests from multiple clients, a simple state machine is applied to each client, as shown in Figure 2. On the one hand, this allows identifying unsupported requests by checking their start String. Supported requests start with the Strings "set ", "get " or "gets ". On the other hand, a request which is split into multiple packets and possibly interleaved with requests of other clients, can be handled correctly, because the bytes of each packets are appended to a per-client byte stream, which is parsed after every time new bytes have been added to it. If parsing is successful, the byte stream is cleared and the request is appended to the queue.

If the request is a `GET` or a non-sharded `MULTI-GET`, the index of the server to which the request should be sent is added to the `Message` object (see Section 1.4). This is possible, since every request passes through the net thread and therefore a simple counter (modulo the number of servers) is sufficient to achieve load balancing between the servers for `GET` and non-sharded `MULTI-GET` requests. This load balancing scheme is called *Round Robin*.

[6] ch.ethz.asltest.middleware.internal.NetThread
[7] `https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html`
[8] `https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html`
[9] `https://docs.oracle.com/javase/8/docs/api/?java/io/ByteArrayOutputStream.html`

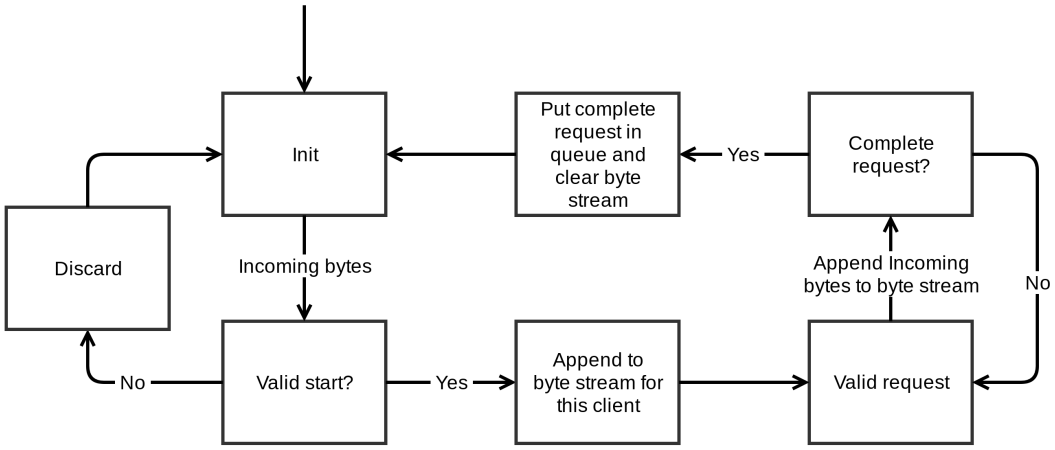


Figure 2: The state machine as it is implemented in the net thread for handling incoming requests of a client.

1.4 Parsing

Both, the net thread and the worker threads, need to parse incoming bytes into a usable abstraction, which can be used for processing the data. For this purpose, the parser[10] verifies incoming bytes and creates these abstractions, if the bytes form a valid message. The parser takes a byte array as input and tries to create one of the following objects (all of which can be found in the message package[11]), which are used for the various types of messages:

- `MessageGetRequest`: Contains one or more requested keys and can therefore be used for `GET`s and `MULTI-GET`s. For `GET`s and non-sharded `MULTI-GET`s, this class also contains the index of the server to which the request should be sent.
- `MessageGetResponse`: Contains the response to a `GET` request. It has a list of zero or more values and can therefore be used for `GET`s and `MULTI-GET`s.
- `MessageSetRequest`: Contains the requested key as well as the flags and expiration time. However, the latter two do not have any functionality within the middleware and only exist for completeness.
- `MessageSetResponse`: Contains a single String with the response from the server to a `SET` request.
- `MessageError`: Contains a single String with any errors coming from the server.

If the parser fails to parse a byte array, indicating an incomplete or invalid message, it returns `null`. In this case, a caller (i.e. the net thread or a worker thread) waits for more bytes belonging to this message, since, per design, the parser is never called for invalid messages.

[10] ch.ethz.asltest.middleware.message.MessageParser
[11] ch.ethz.asltest.middleware.message

1.5 Request queue

The request queue[12] is implemented using Java's `LinkedBlockingQueue`[13]. A blocking queue is preferable to a `ConcurrentLinkedQueue`[14], since a worker thread blocks upon trying to take an item out of the queue (with the method *take*). This means that no resources are wasted on threads which are busy waiting. Additionally, multiple worker threads are not competing for an element in the queue (e.g. by trying to acquire a lock), since the implementation of `LinkedBlockingQueue` simply choses a single thread to wake up, if there are multiple threads blocking on *take*.

1.6 Worker threads

The worker threads[15] extend Java's `Thread`[16] and are stored in a list. They are invoked when the middleware is started and interrupted once the middleware is stopped. Worker threads take `Message` objects (see Section 1.4) out of the request queue and handle them as described in Section 1.7. Every worker thread has a blocking TCP connection to every server.

The number of worker threads is specified as an argument on middleware startup. In theory, there is no limit on the number of worker threads (except `INT_MAX`), however, it is recommended to not exceed 128 worker threads.

1.7 Handling types of requests

Once a worker thread takes a request out of the request queue, it handles it differently, depending on its type. For a graphical overview of how the different requests are handled, refer to Figure 3.

1.7.1 Set

A `SET` request is replicated to all servers. The request is first sent to all servers and afterwards their responses are collected. If all servers respond with a message indicating success, one of them is sent back to the client. If one or more servers respond with an error, one of them is sent back to the client.

1.7.2 Get

A `GET` request is sent to a single server which is determined by the index in the `MessageGetRequest` (see Section 1.4). This is possible, since `SET`s are replicated to all servers and therefore a value can be retrieved from any server. The response of the server is relayed to the client. Non-sharded `MULTI-GET`s are treated equivalently as regular `GET`s, because the request has to be sent to only one server as well and the abstraction (`MessageGetRequest`) is the same.

1.7.3 Multiget

The handling of a `MULTI-GET` depends on the mode of the middleware, i.e. sharded or non-sharded. If it is in non-sharded mode, the request is handled equivalently to handling a `GET` with a single key (see Section 1.7.2). Handling a `MULTI-GET` in sharded mode is slightly more involved. First, the keys are split into n groups where n is the number of servers and for each

[12] ch.ethz.asltest.middleware.internal.RequestQueue
[13] `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html`
[14] `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html`
[15] ch.ethz.asltest.middleware.internal.RequestQueue.WorkerThread
[16] `https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html`

group a separate `MULTI-GET` request is created. Each request is sent to the corresponding server and all the responses are collected. After all responses have been received, the `VALUE` blocks (see Section 1.2) are merged and reordered to match the order of the requested keys. Afterwards, they are sent back to the client as a single list of `VALUE` blocks.

1.8 Shutdown hook

Upon receiving an interrupt (e.g. a `SIGTERM` from Linux), the shutdown hook[17] is invoked. It first interrupts the net thread, the log thread and all worker threads, which in turn clean up their resources, e.g. close sockets and flush files. Once all those threads have terminated, the shutdown hook invokes the `StatsAggregator` (see Section 1.9.2) and waits for its completion. After the statistics have been aggregated and the data is written to a file, the shutdown hook, and with it the middleware, terminate.

1.9 Logging

To collect data for various statistics, several metrics have to be logged. For this purpose, the worker threads and the net thread append lines to a log file using the `StatsLogger`[18], which in turn uses Log4j 2[19], which provides a very efficient API for multiple threads appending to the same file. The data is later aggregated into intervals of five seconds (see Section 1.9.2). For this purpose, the `StatsLogger`, which extends Java's `Thread`, increments the interval id every five seconds. Obviously, there is no explicit need for a separate thread which only increments a counter, however, this design choice was made because it is cleaner and more versatile, if one wants to add different, more complicated aspects to the logging which would require a separate thread.

1.9.1 Collection of data

The net thread only logs the queue length, whereas the worker threads log information regarding the three types of requests. The following three log events can be differentiated. An overview of the meaning of the different variables is given in Table 1.

1. An element is put into the queue
2. A `SET` request has been handled
3. A `GET` or `MULTI-GET` request has been handled

The corresponding data which is logged is the following:

1. $Interval, l_{queue}$
2. $Interval, id, T_{resp}, T_{queue}, T_{service}, type$
3. $Interval, id, T_{resp}, T_{queue}, T_{service}, type, n_{hits}, n_{misses}, n_{keys}$

The queue length is logged every time an element is added to the queue. This gives a reasonably well picture of the queue length. However, if the queue was mostly empty but had to handle bursts of requests, this would not reflect the true queue length variation. However,

[17] ch.ethz.asltest.middleware.internal.ShutdownHook
[18] ch.ethz.asltest.middleware.logging.StatsLogger
[19] `https://logging.apache.org/log4j/2.x/`

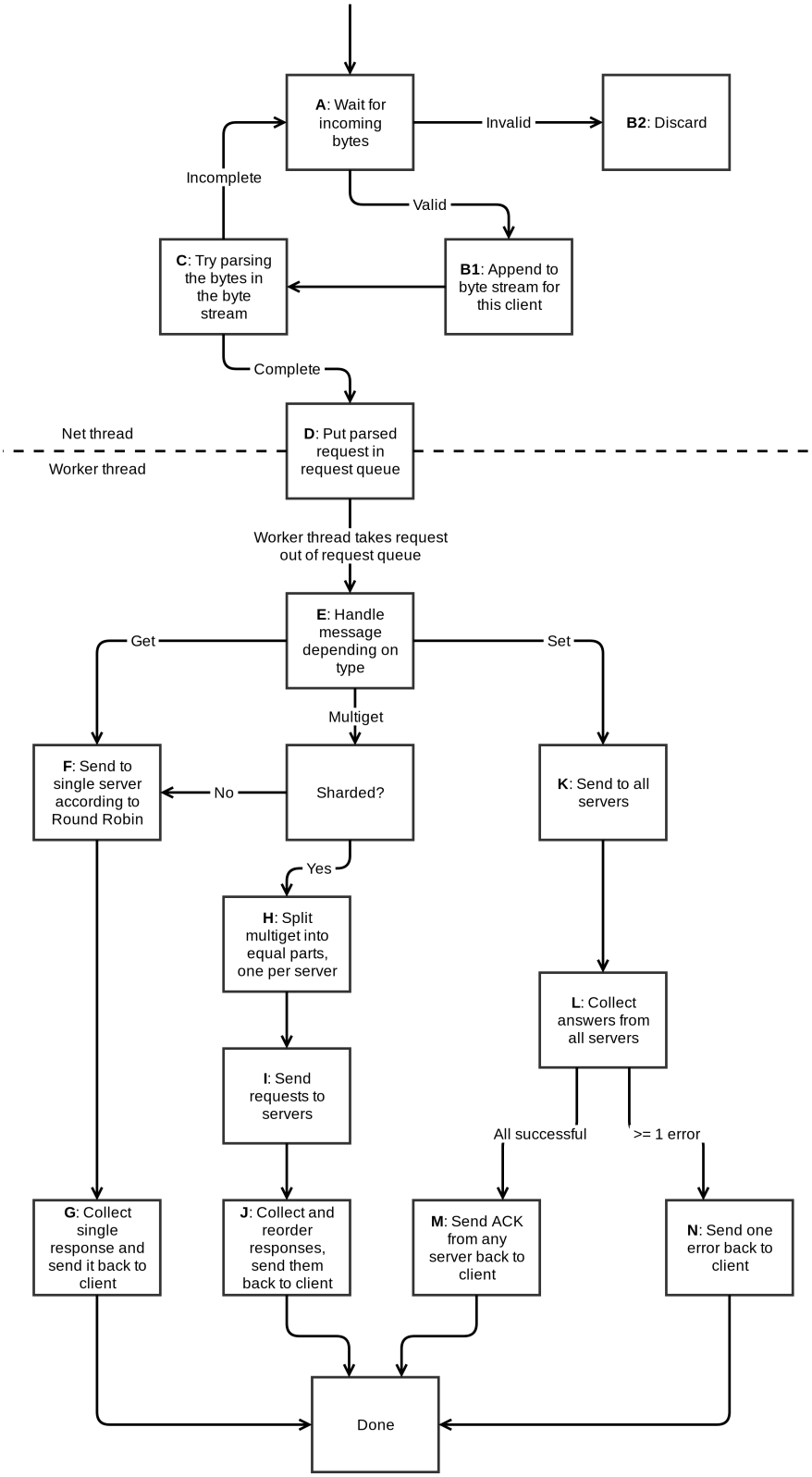


Figure 3: A flow diagram of how one request is processed in the middleware.

$Interval$	The ID of the interval this log event is in
l_{queue}	The length of the request queue
id	The unique request ID
T_{resp}	The response time
T_{queue}	The wait time in the request queue
$T_{service}$	The service time of the *memcached* servers
$type$	The type of request (`GET`, `MULTI-GET`, or `SET`)
n_{hits}	The number of cache hits
n_{misses}	The number of cache misses
n_{keys}	The number of keys in a `GET` or `MULTI-GET`

Table 1: Description of the different variables used for logging.

since a *memtier* client sends a new request as soon as it got an answer to the last request and because monitoring the queue length with a separate thread would entail an additional overhead, we decided to log the queue length on insertion of an element.

T_{resp}, T_{queue} and $T_{service}$ are calculated by logging a start and an end timestamp for each of the three lengths of time. In particular, T_{resp} gets calculated by subtracting the timestamp when the reply has been sent to the client and the timestamp when the middleware receives the first bytes of the request. T_{queue} is the difference between the timestamps of a request taken out of the queue and the timestamp of it being put into the queue. Lastly, $T_{service}$ is measured by subtracting the timestamp when all *memcached* servers have replied and the timestamp when the first request is sent to a server.

n_{hits}, n_{misses} and n_{keys} are collected by simply counting the cache hits and misses and the number of keys in a `GET` or `MULTI-GET`, respectively. The cache hits and misses are logged to ensure that no data is evicted from the servers during an experiment and the average number of keys is recorded to ensure that the expected `MULTI-GET` size matches the one that *memtier* actually uses. If not stated otherwise, these sanity checks pass in the experiments.

1.9.2 Aggregation of data

Once the middleware is interrupted, the shutdown hook invokes the `StatsAggregator`[20]. This class reads the raw log file described in Section 1.9.1 and aggregates the data into five second intervals. This means that it calculates averages and sums (depending on the type of data) and writes it to a file, one line per interval. All values are collected over the entire five second interval, not per second. One line has the following format, the description of the different variables can be found in Table 2.

- $Interval, \overline{T}_{resp}, \overline{T}_{queue}, \overline{l}_{queue}, \overline{T}_{service}, n_{get}, n_{multi-get}, n_{set}, n_{hits}, n_{misses}, n_{keys}$

Additionally, a histogram for the response time is written to the same file, by simply adding up the number of requests in the various buckets while aggregating the statistics above. The bucket size is $100\mu s$ and trailing buckets with less than ten elements are omitted.

1.10 Experiments setup

The experiments in the following sections were run on Microsoft Azure Cloud [21]. The servers, clients and middlewares were deployed on virtual machines of type *Basic A1* (1 Cores, 1.75 GB RAM), *Basic A2* (2 Cores, 3.5 GB RAM) and *Basic A4* (8 Cores, 14 GB RAM), respectively.

[20] ch.ethz.asltest.middleware.logging.StatsAggregator

[21] `https://azure.microsoft.com`

$Interval$	The ID of the interval
$\overline{T}_{resp}$	The average response time
$\overline{T}_{queue}$	The average wait time in the request queue
$\overline{l}_{queue}$	The average length of the request queue
$\overline{T}_{service}$	The average service time of the *memcached* servers
n_{get}	The number of `GET` requests
$n_{multi-get}$	The number of `MULTI-GET` requests
n_{set}	The number of `SET` requests
n_{hits}	The number of cache hits
n_{misses}	The number of cache misses
n_{keys}	The number of keys in `GET`s and `MULTI-GET`s

Table 2: Description of the different variables used for aggregation of logged data.

On each VM, Ubuntu 16.04 LTS was installed. Each VM has at least 100 Mbps up- and download.

In the cloud, reallocating VMs may change their location and therefore influences the latency and throughput measurements in experiments. For this reason, experiments in the same section were run without shutting down the VMs in between, unless stated otherwise.

As mentioned in Section 1.1, we use *memcached* as servers and *memtier* as load generating clients. One can specify two parameters when using *memtier*: The number of *virtual threads* (CT) and the number of *virtual clients* per thread (VC). Therefore, the total number of virtual clients per *memtier* instance is $CPM = CT * VC$. In the experiments, multiple *memtier* instances (NumMemtier) on multiple client machines can be used. The total number of clients in the system can therefore be described as $NumClients = NumMemtier * CPM$. In all graphs, the "number of clients" refers to $NumClients$.

Before each experiment, the servers were populated, i.e. each key was written once. All requested and written values are 1024B. Therefore, below a throughput of around 100,000 ops/s, the network should never be the bottleneck.

The error intervals depicted in the graphs correspond to the 99% confidence interval. When indicated in the caption, the confidence intervals are present, even though they are barley visible in some graphs due to the low variation of the data.

2 Baseline without Middleware

In this section, a baseline for the system without the middleware, i.e. only *memtier* clients and *memcached* servers, is measured. In Section 2.1, the behavior of the system with one server and three client machines is analyzed and in Section 2.2, the same analysis is done with two servers and one client machine. The goal of these experiments is to find the maximum throughput a client machine can generate, as well as the maximum throughput a single server can handle.

2.1 One Server

To find out the how much load is needed to make the server the bottleneck, this experiments uses a single server machine and many clients. Three load generating VMs, each with one *memtier* instance and two virtual threads (CT = 2) are deployed. The number of virtual clients is varied from 1 to 48 and the experiment is repeated with a read-only and a write-only workload. A detailed configuration can be found in the following table:

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

2.1.1 Hypothesis

This experiment tests how much load a single server can handle before the system becomes saturated. Since there is only one server machine and three client machines, and therefore up to 288 clients (VC = 48), we expect to see a saturation of the servers with a relatively small number of clients. The absolute numbers may differ for the read-only and write only workload.

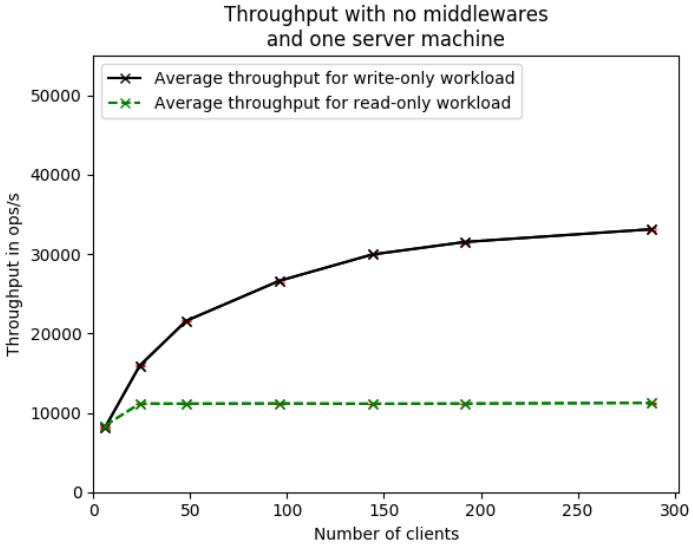


Figure 4: The aggregated throughput as a function of NumClients with 99% confidence intervals.

2.1.2 Explanation

As expected, the system becomes saturated with a small number of clients. For a read-only workload, 24 clients (VC = 4) are needed for saturation, as one can clearly see in Figure 4. From 1 to 24 clients the system is under-saturated and from 24 to 288 clients, the throughput remains constant at around 11,000 ops/s but the system does not get over-saturated. When saturated, the corresponding response time, as depicted in Figure 5 increases linearly as expected. Since the throughput remains constant beyond 24 clients, one can infer that the server is the bottleneck of the system.

However, for the write-only workload, the system behaves very differently. It is not trivial to determine the number of clients which saturated the system when only looking at the throughput in Figure 4. However, the response time in Figure 5 has a clear knee at 192 clients (VC = 32), so one can argue that the system is saturated with 192 clients for a write-only workload. As well as for the read-only workload, no over-saturation occurs, however, the throughput where the system is saturated is around three times higher.

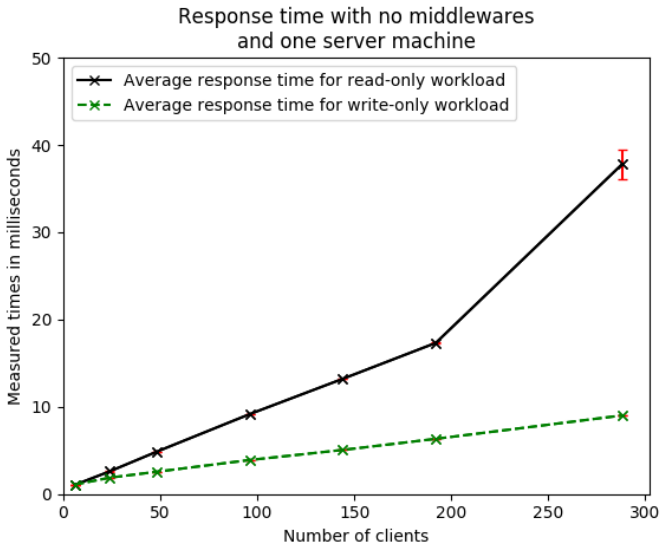


Figure 5: The aggregated response time as a function of NumClients with 99% confidence intervals.

2.2 Two Servers

In this experiment, we want to find out how much load a client can generate without being bottlenecked by the servers. For this purpose, one load generating VM with two *memtier* instances and one virtual thread (CT = 1) as well as two *memcached* servers are deployed. The number of virtual clients is varied from 1 to 48 and the experiment is repeated with a read-only and a write-only workload. A detailed configuration can be found in the following table:

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

2.2.1 Hypothesis

Since there are two servers in this experiment and the single server in the last experiment was bottlenecking the system for a read-only workload, we expect to see saturation at around twice the throughput for the read-only workload from before. We also expect `SET`s and `GET`s to generate equal load for the clients, since *memtier* does not store data and the overhead for generating data to send in `SET`s should be negligible. Therefore, the read-only and write-only workloads should perform similarly.

2.2.2 Explanation

As hypothesized, the systems becomes saturated at around twice the throughput as in the experiment in Section 2.1. One can see in Figure 6 that between 1 and 16 clients (VC = 8), the throughput increases approximately linearly and the response time remains constant,

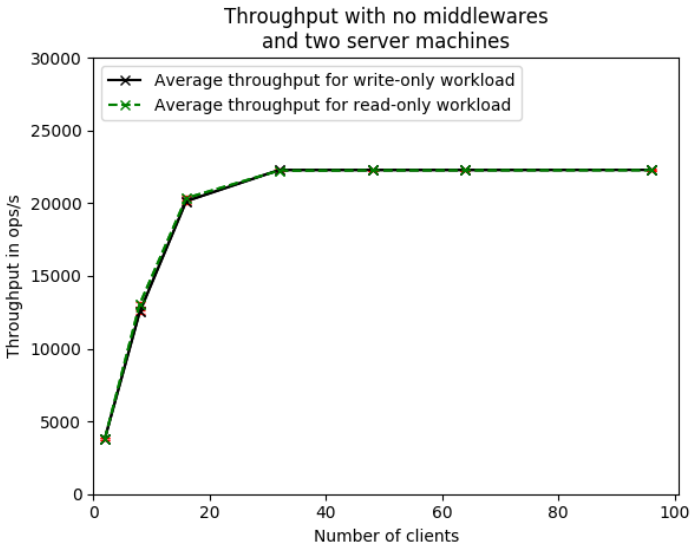


Figure 6: The aggregated throughput as a function of NumClients instance with 99% confidence intervals.

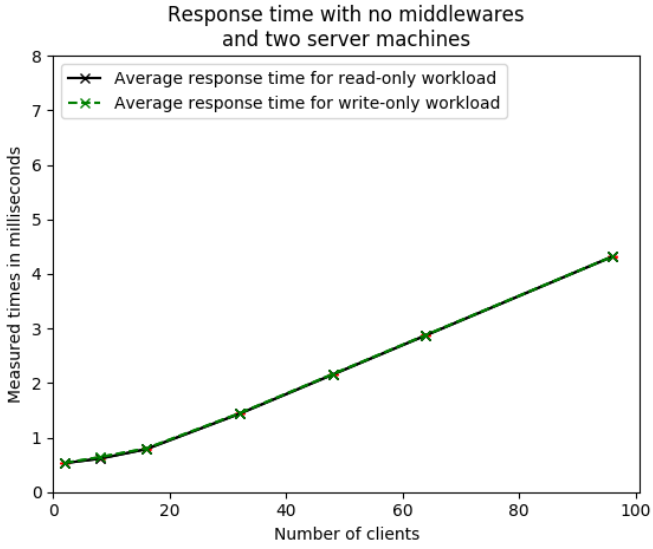


Figure 7: The aggregated response time as a function of NumClients instance with 99% confidence intervals.

as depicted in Figure 7. However, beyond 16 clients, the system becomes saturated, i.e. the throughput stays constant and the response time increases linearly. Again, the system does not become over-saturated.

The difference between the read-only and the write-only workload is negligible, as expected. This is in contrast to the experiment with one server and three client machines from the last section. One can infer that the delay on the clients for generating read-only and write-only workloads does not differ significantly.

Note that the number of clients needed for saturation is much smaller than for the experiment with only one server. This is most likely caused by the single client machine bottlenecking the system. For 16 clients, it already uses all its resources and increasing the number of virtual

clients just leads to the VM's resources being shared by more virtual clients and therefore no increased throughput.

2.3 Summary

The maximum throughput of both experiments for a saturated system with a read-only and write-only workload each can be found in the following table:

Throughput in a saturated system with different experiments and workloads in ops/s.

	Read-only workload	Write-only workload	Configuration gives max. throughput
One *memcached* server	11,152 (VC=4)	31,530 (VC=32)	Write-only, 192 clients
One load generating VM	20,350 (VC=8)	20,137 (VC=8)	Read-only, 16 clients

When comparing the read-only workloads, one can see that when doubling the number of servers (from one to two) the maximum throughput also doubles. This indicates that the server is the bottleneck. Note that the system with one server is already saturated with 24 clients (VC = 4), whereas for one load generating VM, only 16 clients are needed to saturate the system.

On the other hand, for a write-only workload, the throughput decreases from around 30 thousand to 20 thousand, when using less clients. As stated earlier, this indicates that the single client machine cannot generate more load, even when increasing the number of virtual clients on this machine. This is most likely due to the limited resources on the single machine, i.e. increasing the number of virtual clients does not increase the throughput because the overall computing power simply has to be shared by more virtual clients.

Note that values in the table above are not the maximum values over all configurations, but rather the throughputs of the number of clients where the system is saturated. For example, in the read-only experiment with one client machine, the maximum throughput is around 22 thousand with 288 clients (VC = 32). However, this value has little meaning, since the system is saturated with a fraction of this number of clients, so we use VC = 8, as explained in Section 2.2.

In conclusion, one can say that a single server can handle around 11 thousand reads and 30 thousand writes per second, whereas a single client machine can generate around 20 thousand ops/s, no matter if they are reads or writes. Besides the absolute numbers, which may vary for different allocations of the VMs in the cloud, the key insight is that the servers handle writes around three times faster than reads, whereas the clients generate reads and writes at around the same rate, which is approximately twice the read rate of the servers.

3 Baseline with Middleware

In this section, a baseline for the system with middlewares is established. For this purpose, both read-only and write-only workloads are used to measure various metrics when using one and two middlewares.

3.1 One Middleware

First, the system with a single middleware is deployed to test the load which one middleware can handle. The system is configured as follows:

Number of servers	1
Number of client machines	1
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	{1, 2, 4, 6, 8, 10, 16, 24, 32, 48, 64}
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1
Worker threads per middleware	{8, 16, 32, 64}
Repetitions	4

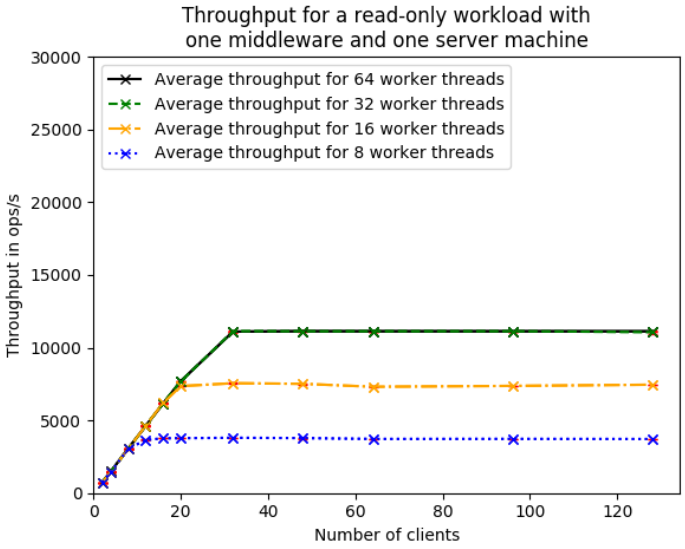


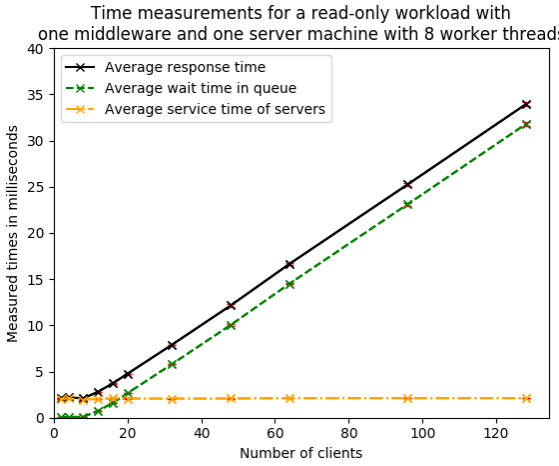
Figure 8: The aggregated throughput as a function of NumClients for different numbers of worker threads for a read-only workload and one middleware with 99% confidence intervals.

3.1.1 Hypothesis

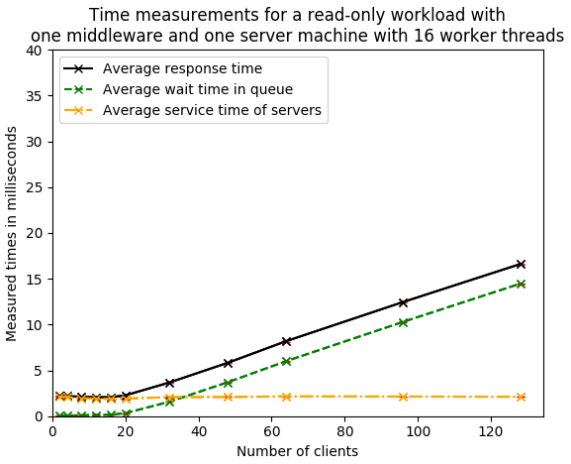
Since only a single middleware is deployed, we will most likely see that it becomes the bottleneck when increasing the number of clients. The more worker threads are used, the more clients should be needed to saturate the system. As seen in Section 2, writes have a higher throughput on the *memcached* server than reads and we therefore also expect the throughput of writes to be higher than for reads when using a middleware.

3.1.2 Explanation

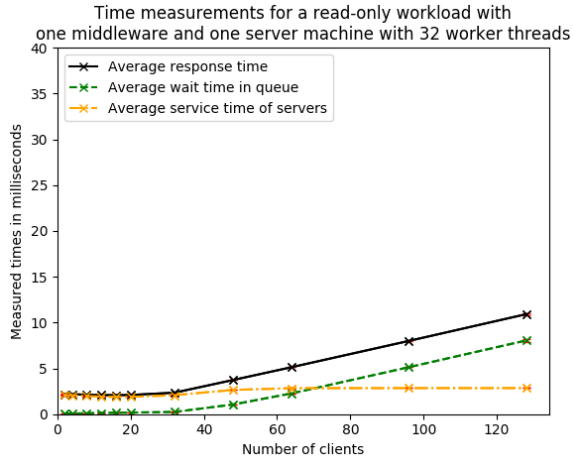
For a read-only workload, Figure 8 shows that the system is saturated with 8 clients (VC = 4) for 8 worker threads, 20 clients (VC = 10) for 16 worker threads and 32 clients (VC = 16) for 32 and 64 worker threads. For 8, 16 and 32 worker threads, the middleware is clearly the bottleneck, as one can see in Figures 9a - 9c. The service time of the servers stays constant while the queue wait time increases linearly when the system is saturated, which causes the response time to increase linearly as well. For 64 worker threads, until 64 clients (VC = 32), the queue wait time is constant while the service time increases once the system is saturated (for 32 clients), as depicted in Figure 9d. Beyond 64 clients, the queue wait time starts to increase linearly, while the service time remains constant. This indicates that the middleware becomes the bottleneck after this point. However, before that, the server is bottlenecking the system.



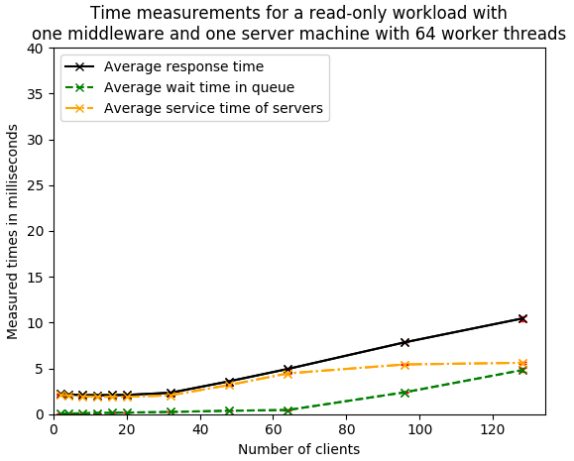
(a) Time measurements on the middleware with 8 worker threads.



(b) Time measurements on the middleware with 16 worker threads.



(c) Time measurements on the middleware with 32 worker threads.



(d) Time measurements on the middleware with 64 worker threads.

Figure 9: Time measurements on the middleware with different number of worker threads for a read-only workload and one middleware with 99% confidence intervals.

For a write-only workload, the results are similar to the read-only workload. However, as already observed in Section 2, the *memcached* server handles write-only workloads with many clients faster than read-only workloads. Therefore, the throughput for writes is higher than for reads, also when using a middleware. For 8 and 16 worker threads, the system behaves almost identically as with a read-only workload: The middleware is clearly the bottleneck as the constant service time in Figures 11a and 11b indicates. The number of clients beyond which the system is saturated is again at 8 (VC = 4) and 20 (VC = 10), respectively. For 32 and 64 worker threads the system behaves differently than for the read-only workload. As illustrated in Figures 11c and 11d, the service time remains constant at around $2ms$, while only the queue wait time increases when the system is saturated.

Overall, one can see that the single middleware, and not the server, is bottlenecking the saturated system for enough clients in all tested configurations, as shown in Figures 9 and 11. Beyond a certain number of clients, the service time remains constant while the queue wait

time increases, regardless of the number of worker threads. Naturally, the number of clients needed to saturate the system becomes larger when increasing the number of worker threads. Over-saturation occurs in none of the experiments.

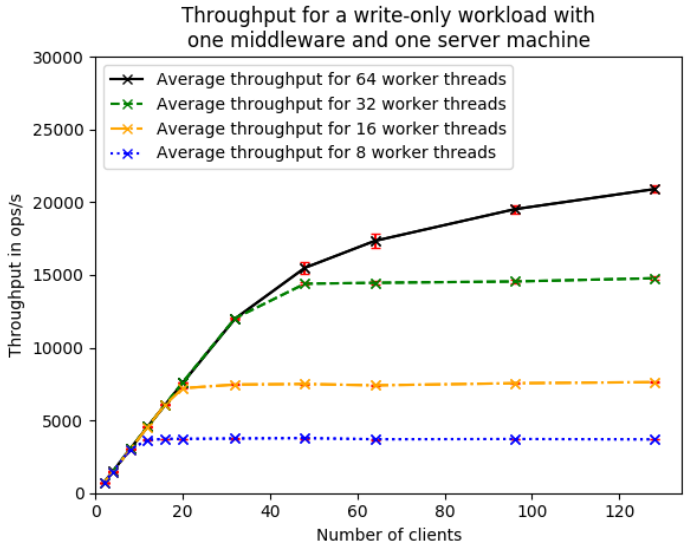


Figure 10: The aggregated throughput as a function of NumClients for different numbers of worker threads for a write-only workload and one middleware with 99% confidence intervals.

3.2 Two Middlewares with one load generating machine

In this section, the system with two middlewares is examined. The parameters of the experiment can be found in the following table:

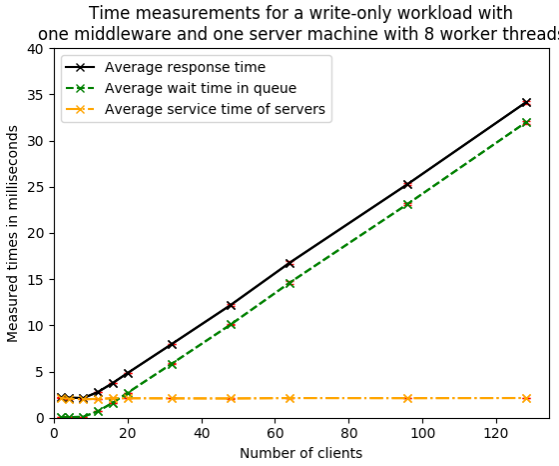
Number of servers	1
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	{1, 2, 4, 6, 8, 10, 16, 24, 32, 48, 64}
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	{8, 16, 32, 64}
Repetitions	4

3.2.1 Hypothesis

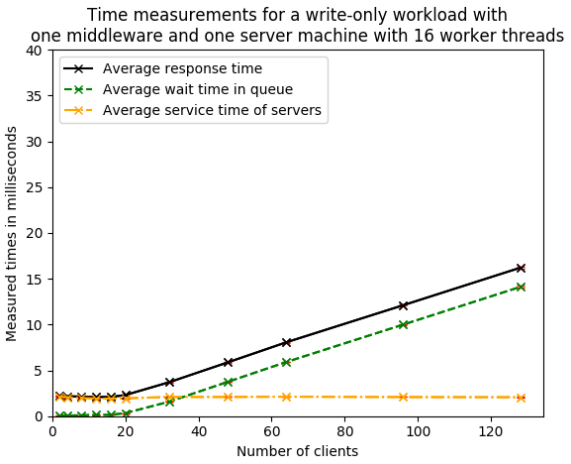
We expect this setup to be able to handle about twice the load as with one middleware before becoming saturated, provided the servers are not bottlenecking the system. Again, with a small numbers of worker threads, we will most likely see bottlenecking with small number of clients. On the other hand, two middlewares might be able to handle the maximum number of clients (128 clients, VC = 64) with 64 worker threads, and therefore making the servers the bottleneck.

3.2.2 Explanation

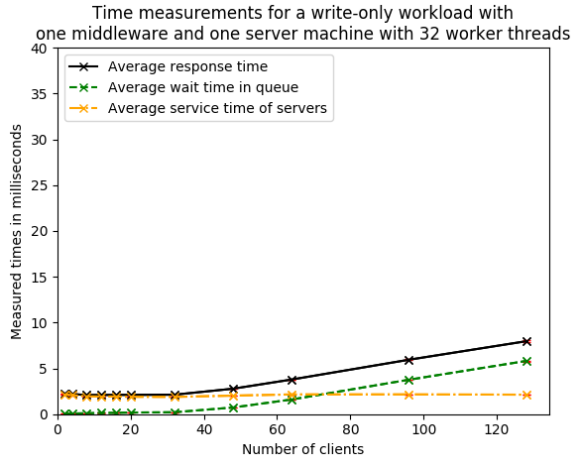
For a read-only workload, the system is again bottlenecked by the middlewares when the number of worker threads is small as illustrated in Figures 13a and 13b. This is similar behavior to



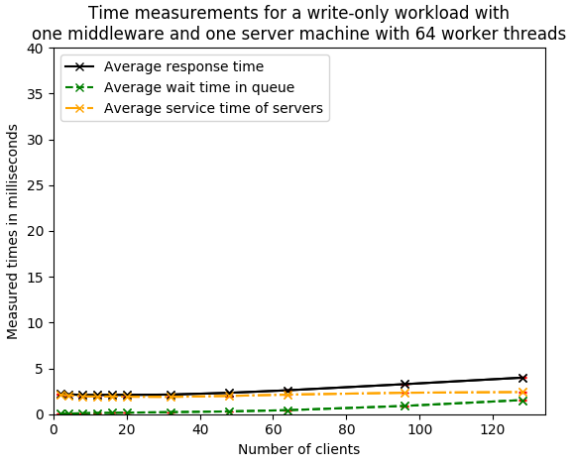
(a) Time measurements on the middleware with 8 worker threads.



(b) Time measurements on the middleware with 16 worker threads.



(c) Time measurements on the middleware with 32 worker threads.



(d) Time measurements on the middleware with 64 worker threads.

Figure 11: Time measurements on the middleware with different number of worker threads for a write-only workload and one middleware with 99% confidence intervals.

the read-only experiment with one middleware in Section 3.1, however, the response time is much lower which one can expect, since we now have two middlewares. When increasing the number of worker threads, the systems behave differently however. With 32 worker threads (Figure 13c), the queue wait time is constant at almost zero and the service time is the main component of the total response time until 64 clients (VC = 32). Beyond that point, the service time remains constant, while the queue wait time starts to increase linearly, indicating that the system is bottlenecked by the middleware. Saturation already occurs with 32 clients (VC = 16). With 64 worker threads, the middlewares are no longer the bottleneck for up to 128 clients, as one can see in Figure 13d. Even though the system becomes saturated with 32 clients, the increased response time is due to an increased service time of the servers and the queue wait time stays constant.

Regarding the throughput, we can see in Figure 12 that it does not increase anymore when using more than 16 worker threads. This corresponds to the similar behavior of the response

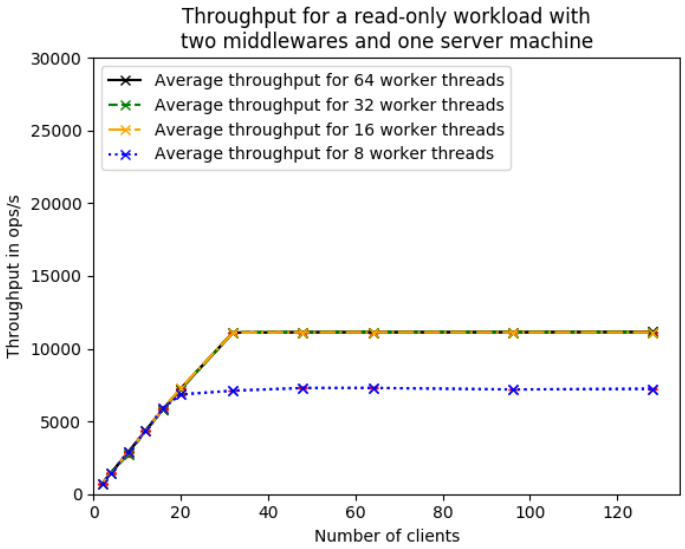


Figure 12: The aggregated throughput as a function of NumClients for different numbers of worker threads for a read-only workload and two middlewares with 99% confidence intervals.

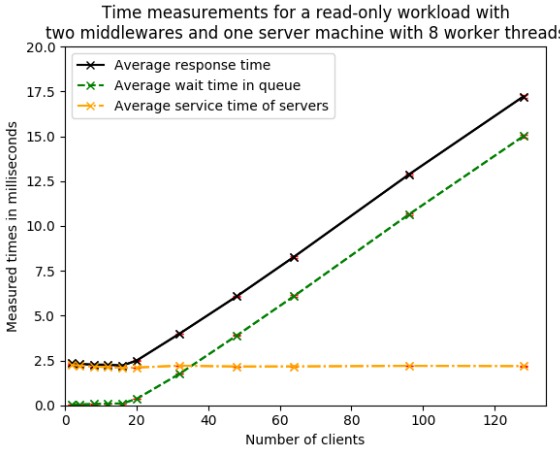
times for 16, 32 and 64 worker threads and supports the claim that the system is saturated with 32 clients, when using 16 or more worker threads.

In Figure 13c, there is an anomaly at 8 clients (VC = 4). As one can see the increased response time is caused by an increased service time. This means, the cause is external to the middlewares and somewhere between the middlewares and the servers, i.e. the network between those two components, the VM running the server or *memcached* itself. Since the experiments were run in a cloud environment, it is not uncommon to see spikes in network traffic or a VM's physical cores being used for another task simultaneously. Due to this and because the spike is clearly an outlier, as indicated by the confidence interval and the measured values for smaller and larger number of clients, we can safely ignore this data point.

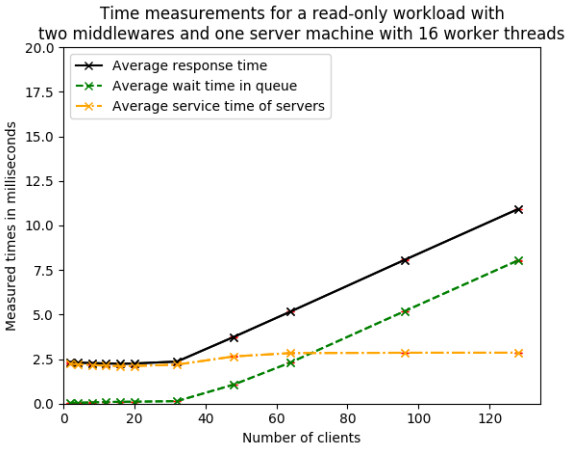
Regarding the write-only workload, we again see similar behavior to the experiment with one middleware, again with smaller response times but similar throughput, as shown in Figure 14. Compared to the read-only workload with two middlewares, we continue to see a larger throughput for write-only experiments. As before, for 8, 16 and 32 worker threads, the middlewares become the bottleneck beyond a certain number of clients, which is indicated in Figures 15a - 15c by the service time remaining constant and the queue wait time increasing linearly. However, with 64 worker threads, the middlewares no longer bottleneck the system with the maximum tested number of 128 clients. As illustrated in Figure 15d, the response time stays constantly at around $2.5ms$, while the queue wait time is almost zero. To find the number of clients for which the middlewares become the bottleneck, even for 64 worker threads, the experiment in the next subsection with two load generating VMs is deployed.

For the configuration in this subsection, the clients are most likely the bottleneck of the system, since we established in Section 2.3 that a single client machine can generate around 20 thousand ops/s and the response time of the middlewares and the service time of the servers remain constant up to 128 clients.

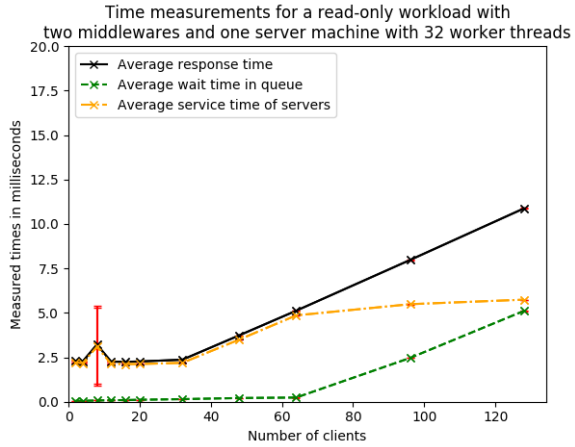
As in the last subsection, no over-saturation occurs.



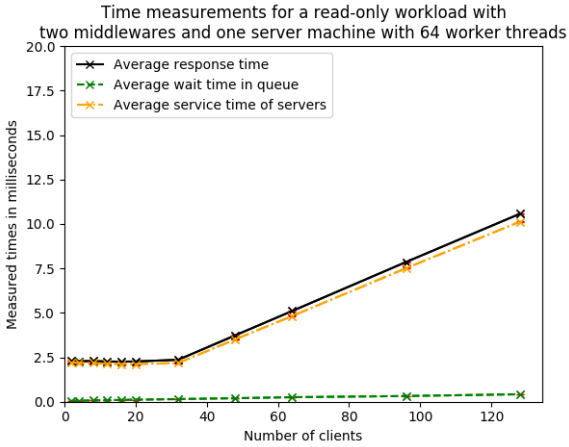
(a) Time measurements on the middlewares with 8 worker threads.



(b) Time measurements on the middlewares with 16 worker threads.



(c) Time measurements on the middlewares with 32 worker threads.



(d) Time measurements on the middlewares with 64 worker threads.

Figure 13: Time measurements on the middlewares for the baseline with different number of worker threads for a read-only workload and two middlewares with 99% confidence intervals.

3.3 Two Middlewares with two load generating machines

Since 128 clients are not enough to make two middlewares the bottleneck of the system, the experiments from the last section are repeated with two load generating VMs, therefore doubling the number of clients. The configuration for these experiments can be found in the following table. Note that between running these experiments and the experiments from the last two subsections, the VMs were shut down, i.e. the absolute values are not comparable, since the VMs have most likely been allocated at different places, changing the network latency between them.

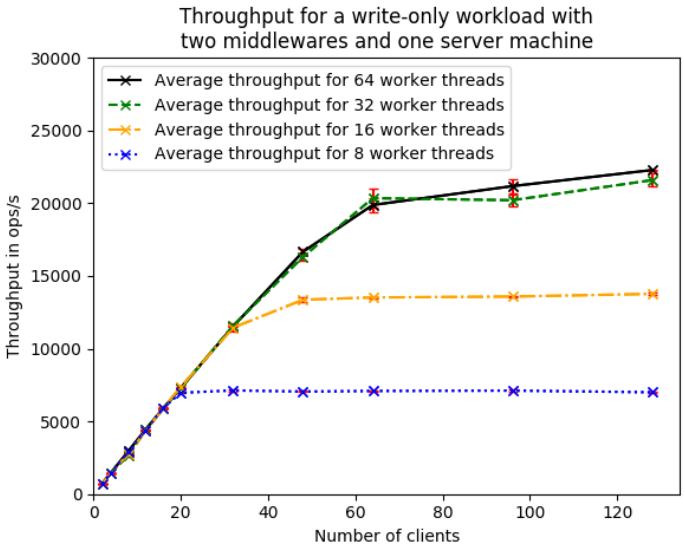


Figure 14: The aggregated throughput as a function of NumClients for different numbers of worker threads for a write-only workload and two middlewares with 99% confidence intervals.

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

3.3.1 Hypothesis

Because we doubled the number of clients for this experiment when compared to using one load generating VM, we expect to see the middlewares bottlenecking the system for more than 128 clients.

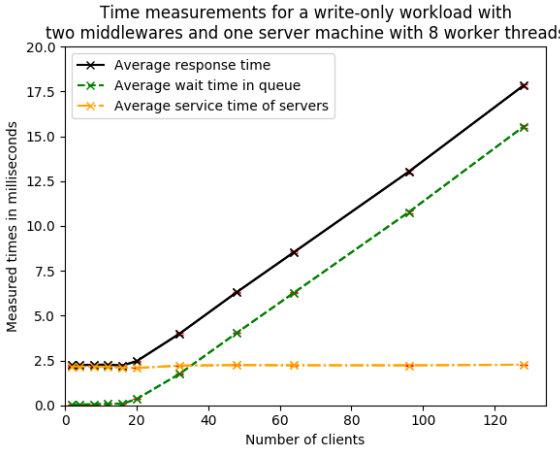
3.3.2 Explanation

As expected, for 192 clients (VC = 48), the middlewares bottleneck the system, both for the read-only and the write-only workload. In Figure 17, one can clearly see the wait time in the queue starting to linearly increase beyond 128 clients (VC = 32), while the service time remains constant beyond this point. This indicates that the middlewares bottleneck the system. Similar behavior can be seen for the write-only workload as illustrated in Figure 19, even though the response time is much smaller in absolute terms than the response times in the last subsection.

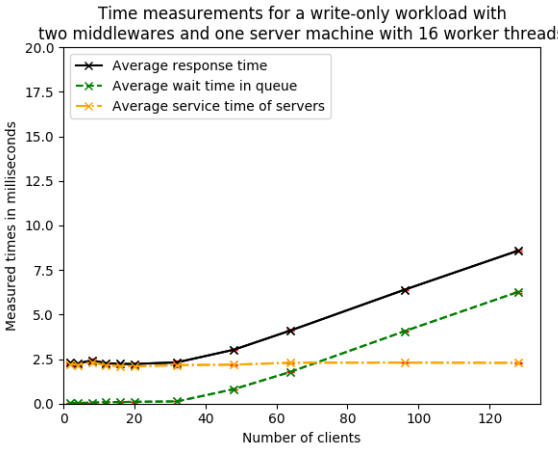
As already established earlier in this section, the system is saturated well before 128 clients, as one can also see when inspecting the throughput in Figures 16 and 18. Again, no over-saturation occurs.

3.4 Summary

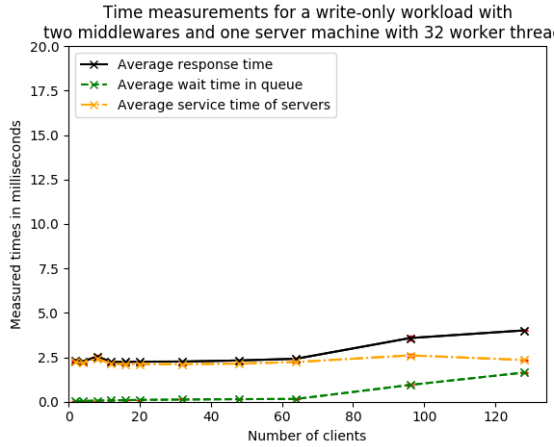
Data for the configuration resulting in the maximum throughput for a saturated system can be found in Tables 3 - 5.



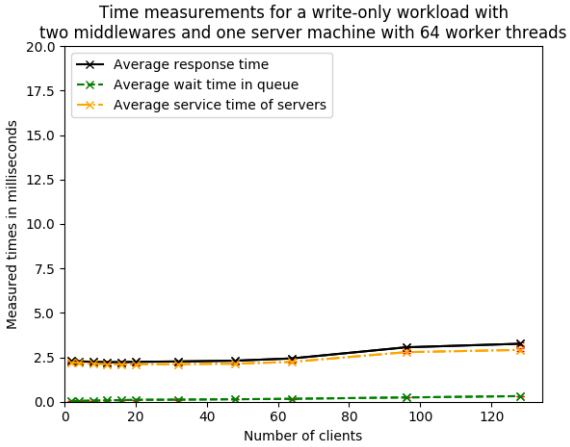
(a) Time measurements on the middlewares with 8 worker threads.



(b) Time measurements on the middleware with 16 worker threads.



(c) Time measurements on the middlewares with 32 worker threads.



(d) Time measurements on the middlewares with 64 worker threads.

Figure 15: Time measurements on the middlewares for the baseline with different number of worker threads for a write-only workload and two middlewares with 99% confidence intervals.

	Throughput (ops/s)	Response time (ms)	Average time in queue (ms)	Miss rate
Reads: Measured on middleware	11,128	2.35	0.24	0
Reads: Measured on clients	11,128	2.88	n/a	0
Writes: Measured on middleware	17,340	2.61	0.44	n/a
Writes: Measured on clients	17,339	3.73	n/a	n/a

Table 3: Data for configuration which leads to maximum throughput for one middleware. Reads: (VC, WT) = (16, 32), writes: (VC, WT) = (32, 64)

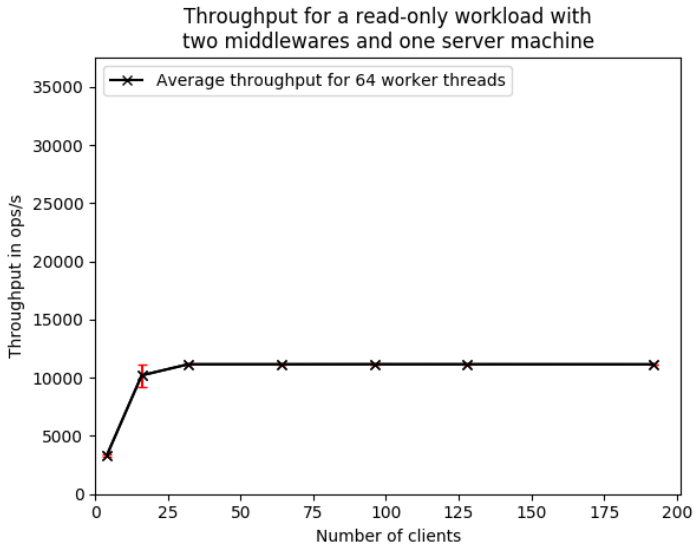


Figure 16: The aggregated throughput as a function of NumClients for 64 worker threads for a read-only workload and two middlewares with 99% confidence intervals.

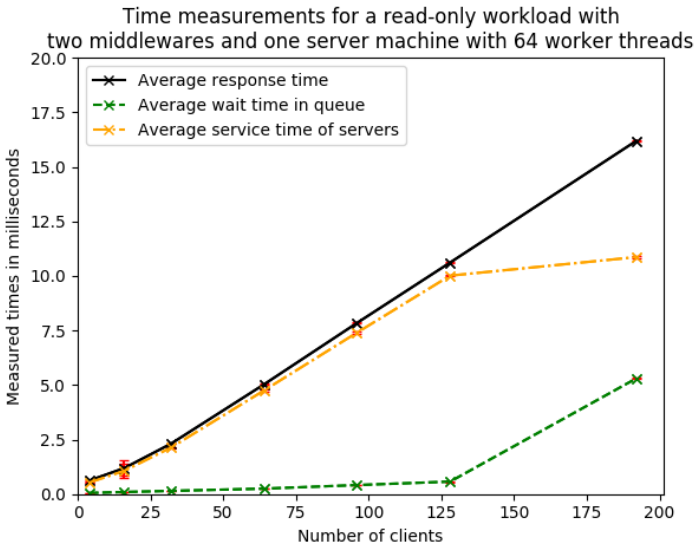


Figure 17: Time measurements on the middlewares as a function of NumClients for 64 worker threads for a read-only workload and two middlewares with 99% confidence intervals.

When comparing the maximum throughput and response times for read-only workloads, one can see that approximately 11 thousand ops/s is the maximum the system can achieve. As established in Section 2.1, this is the maximum, one *memcached* server can handle for a read-only workload.

For write-only workloads, the system is not bottlenecked by the middlewares for the entries in Tables 3 and 4. As one can see, the maximum throughput stays almost the same (at around 20 thousand ops/s) when using the same number of clients but adding a middleware. As concluded in Section 2.3, 20 thousand ops/s is the maximal load a single client machine can generate and therefore we can infer that the clients are the bottleneck for this configuration. However, only half the worker threads (32 instead of 64) are needed to achieve the throughput in Table 4,

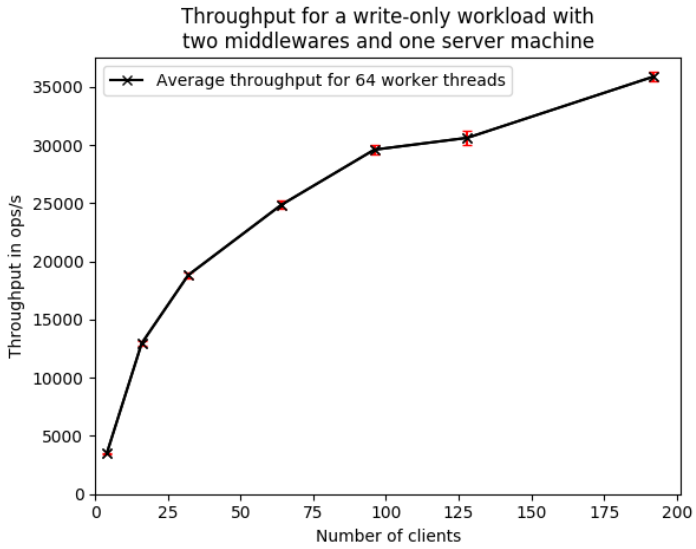


Figure 18: The aggregated throughput as a function of NumClients for 64 worker threads for a write-only workload and two middlewares with 99% confidence intervals.

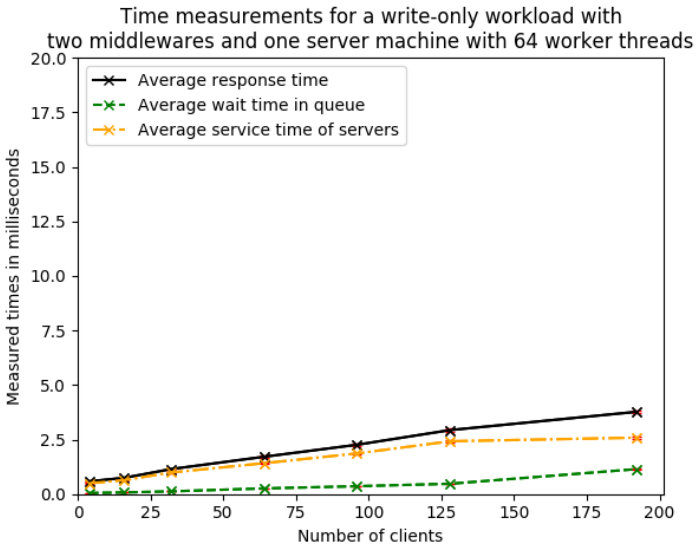


Figure 19: Time measurements on the middlewares as a function of NumClients for 64 worker threads for a write-only workload and two middlewares with 99% confidence intervals.

which makes sense, since the load of the clients is now split across two middlewares. When doubling the number of clients, the middlewares become the bottleneck, as shown in Table 5, since the throughput does not double as well and the queue wait time increases from 0.16ms to 0.47ms. This has already been established in Section 3.3.

The cache miss rate is at zero for all experiments. This shows that no data was evicted from *memcached* when running the experiments, which would have skewed the results. The minor differences between the throughput measurement on the clients and on the middlewares are due to rounding errors, however, they are negligibly small and can safely be ignored.

	Throughput (ops/s)	Response time (ms)	Average time in queue (ms)	Miss rate
Reads: Measured on middleware	11,102	2.36	0.14	0
Reads: Measured on clients	11,102	2.88	n/a	0
Writes: Measured on middleware	20,344	2.42	0.16	n/a
Writes: Measured on clients	20,343	3.20	n/a	n/a

Table 4: Data for configuration which leads to maximum throughput for two middlewares and one load generating VM.
Reads: (VC, WT) = (16, 16), writes: (VC, WT) = (32, 32).

	Throughput (ops/s)	Response time (ms)	Average time in queue (ms)	Miss rate
Reads: Measured on middleware	11,157	2.30	0.15	0
Reads: Measured on clients	11,155	2.90	n/a	0
Writes: Measured on middleware	30,616	2.93	0.47	n/a
Writes: Measured on clients	30,615	4.26	n/a	n/a

Table 5: Data for configuration which leads to maximum throughput for two middlewares and two load generating VMs. Note that this experiment was not done with the same VM configuration as in Tables 3 and 4, therefore the VMs were most likely allocated differently, leading to different network delays.
Reads: (VC, WT) = (8, 32), writes: (VC, WT) = (32, 64).

4 Throughput for Writes

4.1 Full System

In this section, we want to test the full system. For this purpose, an experiment with a write-only workload is run. We use three client and three server machines and two middlewares while varying the number of worker threads inside the middleware. The number of virtual clients is varied between 1 and 64 which means up to 384 clients are run at the same time. The experiment configuration can be found in the following table:

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, 6, 8, 10, 16, 24, 32, 48, 64}
Workload	Write-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	{8, 16, 32, 64}
Repetitions	4

4.1.1 Hypothesis

With 8 worker threads, we expect to see the middlewares being the bottleneck for the system when increasing the number of clients because only 8 worker threads inside each middleware

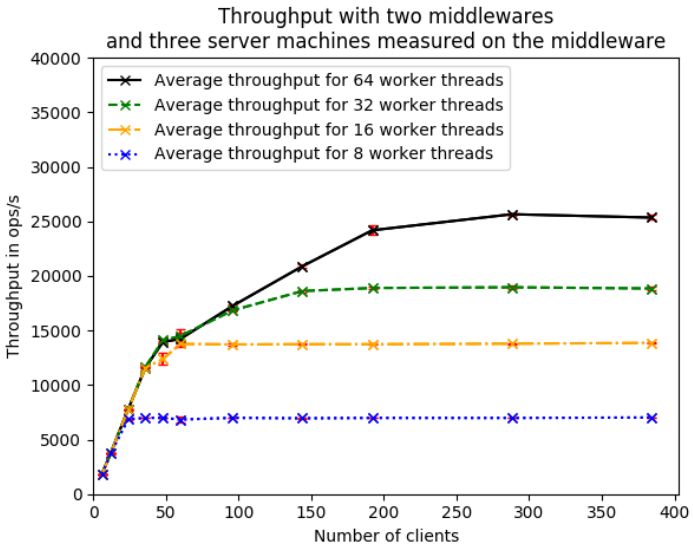


Figure 20: The aggregated throughput as a function of NumClients for different numbers of worker threads with 99% confidence intervals.

will most likely not be enough to handle all incoming requests without a high queue wait time. This implies that the throughput will remain constant beyond a certain number of clients, as well as the service time of the *memcached* server, while the wait time in the queue increases linearly. The increasing wait time in the queue will therefore also be responsible for the linearly increasing response time of the middleware.

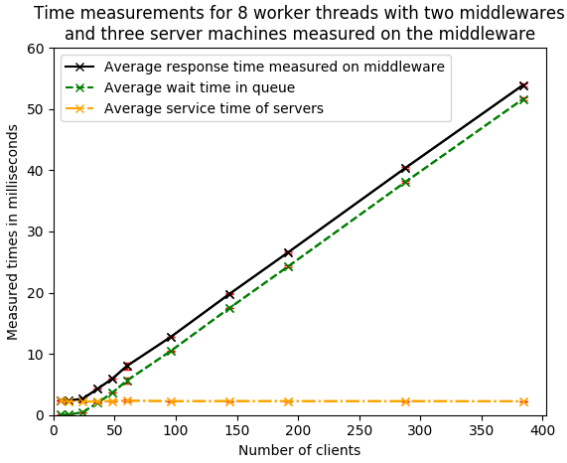
For 64 worker threads, we expect to see much higher throughput and lower response times since the worker threads can handle more requests in parallel. However, the increase will not be a factor of 8, since not the entire system is parallelized. It is possible that a large number of clients will saturate the system, even with 64 worker threads, making the middlewares the bottleneck.

For 16 and 32 worker threads, we expect behavior somewhere in the middle between the extremes of 8 and 64 worker threads. Increasing the number of worker threads should lower the response time and increase the throughput until a certain number of clients has been reached, beyond which the number of worker threads will not be enough handle all requests without queuing and the queue wait time will start to increase linearly and with it the response time, indicating saturation.

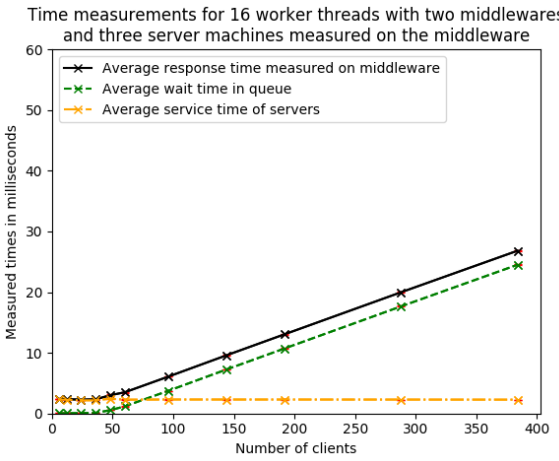
4.1.2 Explanation

As one can see in Figure 20, the throughput first increases when increasing the number of clients and after a certain point it starts to flatten out. Where this point is depends on the number of worker threads in the middlewares, e.g. for 8 worker threads it is at around 24 clients (VC = 4). As expected the throughput remains constant after this point, indicating that the system is saturated. Since the throughput does not decrease when increasing the number of clients, one can infer that no over-saturation occurs.

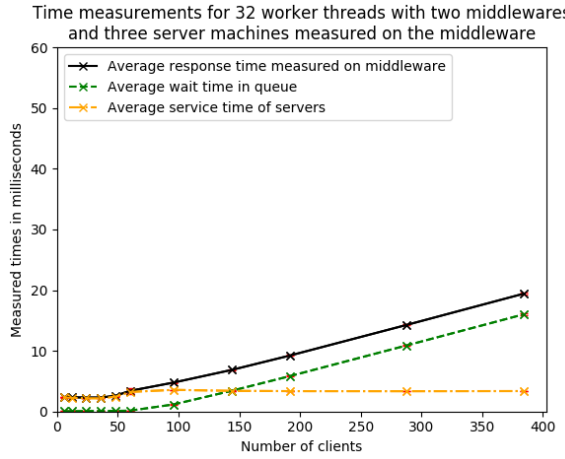
The behavior of the time measurements corresponds to the hypothesis as well. As depicted in Figure 21, the response time decreases when increasing the number of worker threads. Additionally, the response time is dominated by the queue wait time, the more clients and the less worker threads there are. For example, in Figure 21a, one can clearly see that for an increasing number of clients, the service time remains constant while the queue wait time is increasing



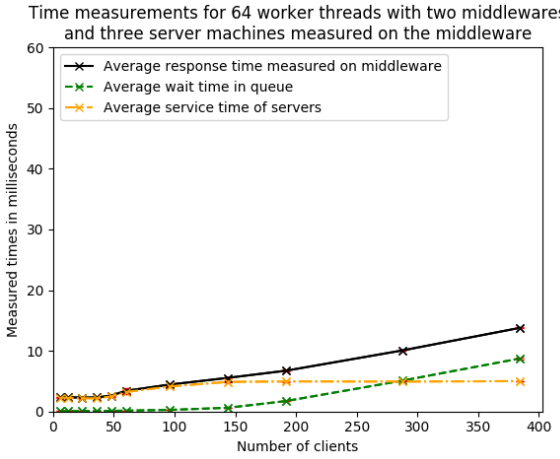
(a) Time measurements on the middleware with 8 worker threads.



(b) Time measurements on the middleware with 16 worker threads.



(c) Time measurements on the middleware with 32 worker threads.



(d) Time measurements on the middleware with 64 worker threads.

Figure 21: Time measurements on the middleware with different number of worker threads with 99% confidence intervals.

linearly.

On the other hand, for 64 worker threads, the queue wait time remains constant until around 192 clients (VC = 32), implying that enough worker threads are available to handle incoming requests, as shown in Figure 21d. Beyond 192 clients, the queue wait time starts to increase, indicating saturation of the system.

As hypothesized, using 16 and 32 worker threads shows behavior between the two extremes of 8 and 64 worker threads. The throughput flattens out (i.e. the system is saturated) at a higher rate, whereas the response time is lower than with 8 worker threads.

Overall, the servers are the bottleneck until the response time starts to increase linearly (after the knee), beyond which, the middlewares become the bottleneck. The clients are never the bottlenecks, which one can infer, since the response time always increases, if the number of clients is large enough and never stays constant (unlike for example in Figure 15d).

	WT=8	WT=16	WT=32	WT=64
Number of virtual clients (VC)	4	10	16	32
Tput. in ops/s (Middleware)	6,854	13,771	16,838	24,182
Tput. in ops/s (Derived from MW resp. time)	8,962	17,165	20,113	28,521
Tput. in ops/s (Client)	6,853	13,771	16,839	24,192
Average time in queue in ms	0.47	1.19	1.17	1.73
Average length of queue	1.80	6.49	6.63	14.53
Average time waiting for *memcached* in ms	2.15	2.27	3.57	4.96
Average resp. time (measured on MW) in ms	2.68	3.50	4.77	6.73

Table 6: Various measured metrics for different numbers of worker threads for the saturated full system for a write-only workload.

4.2 Summary

Table 6 contains measurements for different number of worker threads with the number of virtual clients that achieves the highest throughput in a saturated system. Overall the system behaves as expected when increasing the number of worker threads. As shown in Figures 20, we see an increased throughput when increasing the number of worker threads. However, since not the entire middleware is parallelized, this increase is not linear and we therefore have diminishing returns (in terms of throughput). Similarly, as shown in Figure 21, the response time decreases with more worker threads. Additionally, the number of clients, beyond which the queue wait time is the main factor for an increased response time, becomes larger. This is expected, since more worker threads take requests out of the queue.

When comparing the derived[22] throughput and the measured throughput, one can see that they are not equivalent. This is due to the fact that we derive the throughput on the middleware and not on the clients, which means that the latency of requests between the clients and the middlewares is not taken into account. Therefore, the derived throughput is higher than the measured throughput. However, the derived throughput can be seen as a measurement of what the maximum throughputs of the middlewares themselves are.

Note that the values in the table above were not collected with the maximum overall throughput but rather with the smallest number of clients when the system was saturated. This information can easily be gathered by inspecting Figure 21: the "knee" of the response time (where it goes from constant to linear) is the number of clients beyond which the system is saturated.

5 Gets and Multi-gets

In this section, we want to investigate the performance of `MULTI-GET`s, both in sharded and non-sharded mode. We noticed that when using the *memtier* default ratio of 1:10, it balances the target size with smaller `MULTI-GET`s or `GET`s to achieve that ratio, if the `MULTI-GET` size is not evenly divisible by 10. For this reason, the experiments were done with a 50-50-read-write workload, where one read corresponds to one `MULTI-GET` with the corresponding number of keys, e.g. if the `MULTI-GET` size should be 6, the ratio is set to 1:6. Doing so, resulted in the average `MULTI-GET` size used by *memtier* actually being the `MULTI-GET` size specified

[22]Derived throughput = NumClients / Response time

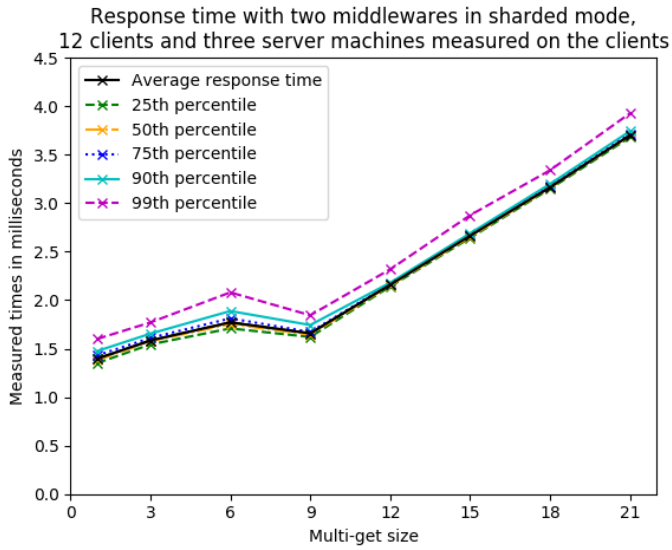


Figure 22: The average response time, as well as different percentiles as a function of the `MULTI-GET` size as measured on the clients, sharded mode.

5.1 Sharded Case

The first experiment is done in sharded mode, which means that `MULTI-GET`s are split into smaller `MULTI-GET`s, one for each of the three servers. A detailed configuration can be found in the following table:

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	50-50-read-write (1 read = 1 multiget)
Multi-Get behavior	Sharded
Multi-Get size	{1, 3, 6, 9, 12, 15, 18, 21}
Number of middlewares	2
Worker threads per middleware	64
Repetitions	3

5.1.1 Hypothesis

Since the overhead of splitting `MULTI-GET`s into smaller part should not be large for small `MULTI-GET` sizes, we expect to see an almost constant behavior of the response time with small `MULTI-GET` sizes. However, for larger `MULTI-GET` sizes, the response time may increase due to the increasing latency at the servers for retrieving multiple values or overhead caused by splitting the messages in the middleware.

5.1.2 Explanation

As expected, the response time remains almost constant between `MULTI-GET` sizes 1 and 9, as illustrated in Figure 22. However, with `MULTI-GET` size 12 and more, the response time starts to increase linearly. As Figure 23 shows, this increase is caused by an increased service time of the servers. It also excludes the network latency as possible cause, since the difference between the response times measured on the client and the middlewares would need to become larger, when increasing the `MULTI-GET` size.

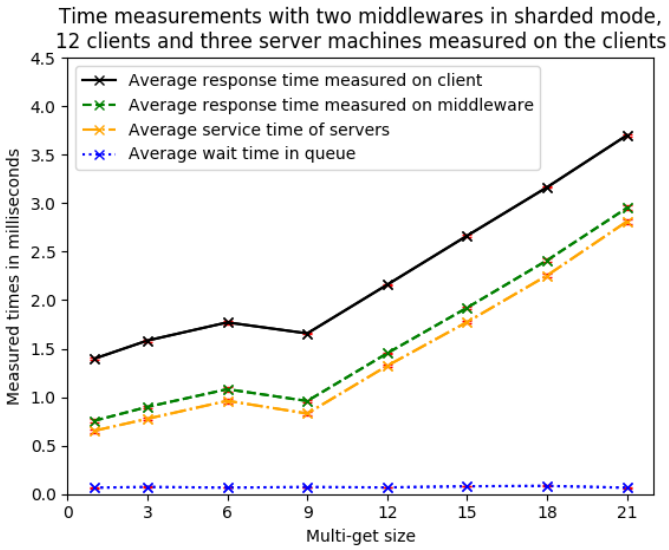


Figure 23: Time measurements on the middleware as a function of the `MULTI-GET` size, sharded mode with 99% confidence intervals.

We can infer that the overhead for splitting the messages is only minor, whereas the servers are saturated beyond `MULTI-GET` size 9, which leads to the increased response times. The different percentiles in Figure 22 do not show any unexpected behavior, there are no outliers, even with the 99th percentile, which indicates stable performance of the system.

5.2 Non-sharded Case

In this experiment, `MULTI-GET`s with sharding disabled are tested. This means that one message contains the keys of multiple values to retrieve which is relayed to a single server by the middleware. The experiment setup can be found here:

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	50-50-read-write (1 read = 1 multiget)
Multi-Get behavior	Non-Sharded
Multi-Get size	{1, 3, 6, 9, 12, 15, 18, 21}
Number of middlewares	2
Worker threads per middleware	64
Repetitions	3

5.2.1 Hypothesis

Since `MULTI-GET` requests are treated the same as `GET` requests (as described in Section 1.7.3), i.e. sent to only one server, we expect to see lower response times than with sharded mode. With larger `MULTI-GET` sizes, the overhead for retrieving multiple values at once for the server will most likely start to increase the response time, as seen in with sharded mode.

5.2.2 Explanation

The response times are overall slightly lower than with sharded mode, which is expected, because there is no splitting and reassembling of messages in the middleware. Also, when increasing

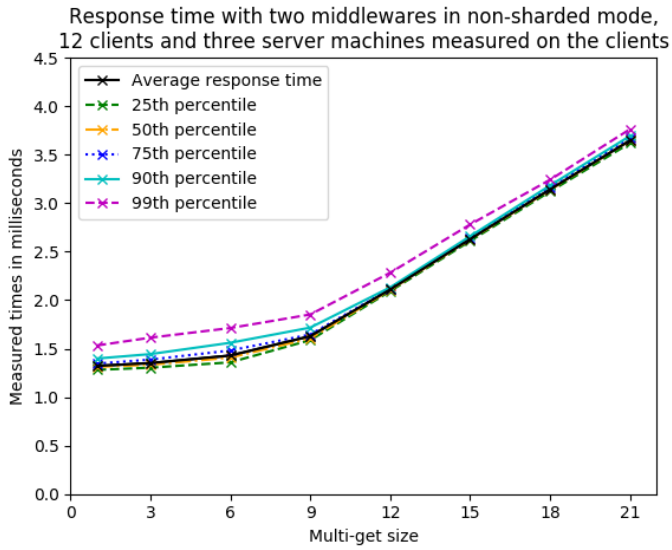


Figure 24: The average response time, as well as different percentiles as a function of the `MULTI-GET` size as measured on the clients, non-sharded mode.

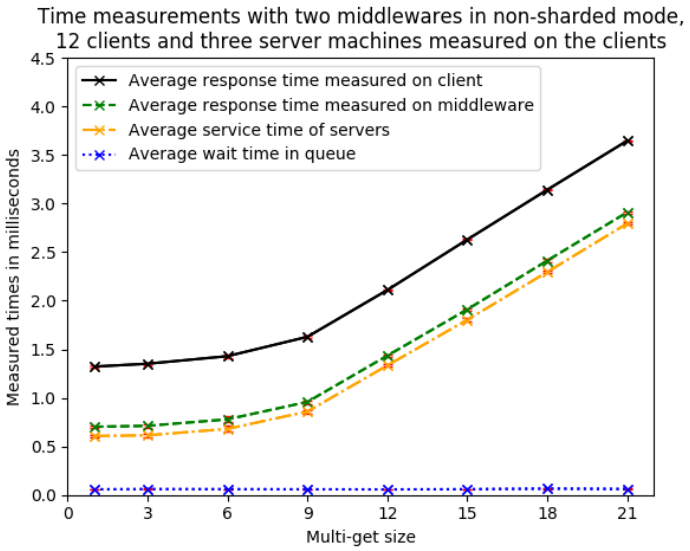


Figure 25: Time measurements on the middleware as a function of the `MULTI-GET` size, non-sharded mode with 99% confidence intervals.

the MULTI-GET size, the response time increases as well, as shown in Figure 24. This is, as in sharded mode, caused by an increased service of the servers time as depicted in Figure 25. Also, the network latency does not contribute any significant increase to the response time, since the difference between the response times measured on the clients and on the middlewares would need to increase with larger MULTI-GET sizes.

Our claim that the servers are saturated beyond MULTI-GET size 9 is further supported by the increasing response times. As with sharded mode, the system is stable, indicated by the different percentiles in Figure 24, which do not show any unexpected behavior.

Note that the observation that system is saturated for both modes with MULTI-GET size 9 may seem counter-intuitive at first. However, the load in terms of values to retrieve per second for one server does not change significantly when using sharded or non-sharded mode, as seen in Figure 27. The difference is that in sharded mode, each server receives multiple smaller MULTI-GET messages and in sharded mode, it receives fewer, larger messages.

5.3 Histogram

For MULTI-GET size 6, histograms of the response times can be found in Figure 26. The bucket size is 100 μs and the number of elements in a bucket is normalized to one second. Figures 26a and 26b show the histogram for sharded mode as measured on the client and the middlewares, respectively. Figures 26c and 26d illustrate the response times as measured on the client and the middlewares in non-sharded mode.

For both modes, the shape of the histogram does not differ significantly, if measured on the clients or on the middleware. This indicates that the network between the clients and the middlewares does not have any significant impact on the curve, besides introducing latency. This latency is clearly indicated by a shift of the curve to larger bucket numbers when measuring on the clients instead of the middlewares, which can be seen for both, sharded and non-sharded mode.

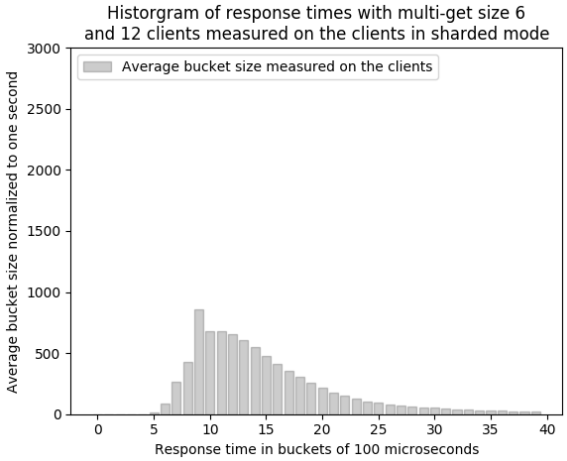
When comparing Figures 26a and 26c, i.e. measurements on the clients for sharded and non-sharded mode, respectively, the difference in the shape of the curves is apparent. Overall the response times with non-sharded mode are lower, which has already been determined in Section 5.2. However, the curve in sharded mode is much flatter than with non-sharded mode. This indicates that the time needed to process a sharded request varies more than processing a non-sharded request. Given that a non-sharded request is just relayed to one server, whereas a sharded request has to be split and the responses reassembled, this is expected.

The difference between the histograms in Figures 26b and 26d, i.e. measurements on the middleware for sharded and non-sharded mode, respectively, is similar to the measurements on the clients, only shifted to smaller bucket sizes because the latency between middlewares and clients is not included. However, the measurements on the clients are slightly flatter when compared to the measurements on the middlewares for their respective mode. This may be caused by the additional variation introduced in the network between the middlewares and the clients.

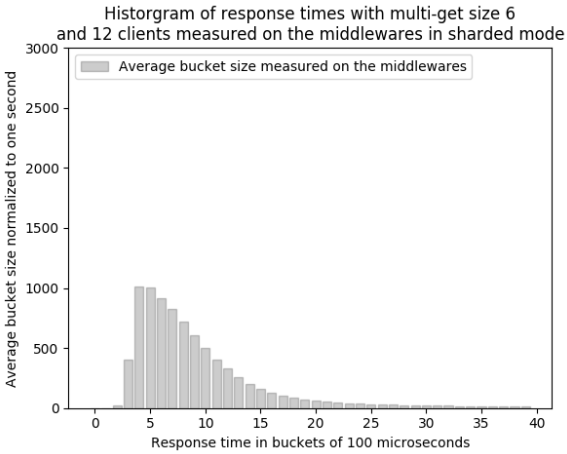
Note that the response time for the 50% writes is also added to the histograms. However, since the write workloads are identical for both modes, their response time distribution is similar, so the difference in for the two modes must come from the read workload.

5.4 Summary

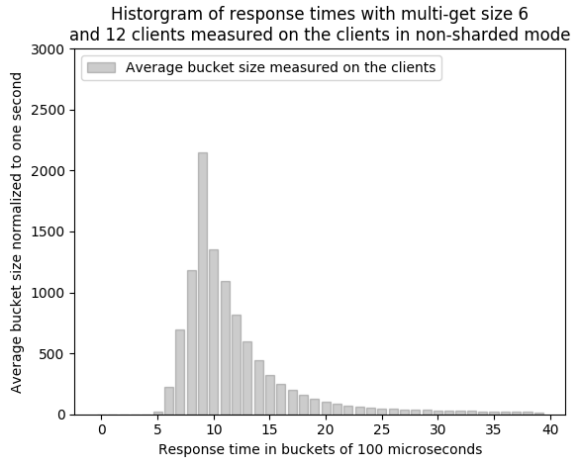
When comparing sharded and non-sharded mode, it becomes obvious that for the MULTI-GET sizes tested in this section, sharded mode is never the preferred option. The response time is larger when sharding the messages than when just relaying them. This overhead comes



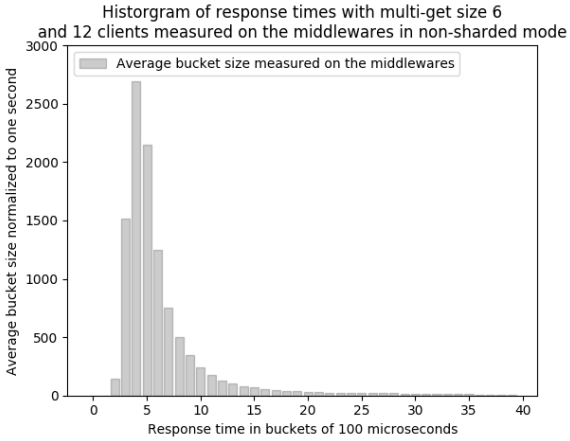
(a) A histogram displaying the response times measured on the clients, sharded mode.



(b) A histogram displaying the response times measured on the middlewares, sharded mode.



(c) A histogram displaying the response times measured on the clients, non-sharded mode.



(d) A histogram displaying the response times measured on the middlewares, non-sharded mode.

Figure 26: Histograms displaying the response times measured on the clients and middlewares for sharded and non-sharded mode and `MULTI-GET` size 6. The bucket size is 100 μs and the number of elements in a bucket is normalized to one second.

from the middleware which has to split the messages, send them to each server, wait for all answers, reassemble the response and send it back to the client (a detailed description of how the middleware handles `MULTI-GET`s can be found in Section 1.7.3). This is also reflected when comparing the throughput in values/s for the two modes, as illustrated in Figure 27.

However, one can conclude that `MULTI-GET`s are preferred over `GET`s. Figure 27 depicts a linear increase in values read per second when increasing the `MULTI-GET` size. Only with `MULTI-GET` size 12 and larger, the curve starts to flatten out, which is caused by the saturation of the servers. However, if using many servers and large `MULTI-GET` sizes, the benefits of load balancing the requests across multiple servers may outweigh the latency increase caused by the middleware, making sharded mode the preferred option. This is however only the case, if the servers are not saturated for sharded messages but are saturated for non-sharded messages, otherwise the two modes would again perform similarly as explained in Section 5.2.2.

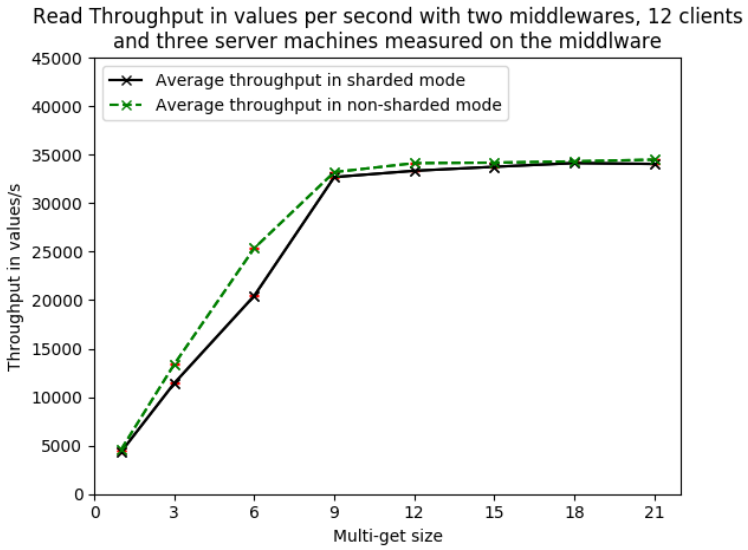


Figure 27: The throughput in values/s (not ops/s) for reads as a function of the `MULTI-GET` size with 99% confidence intervals.

6 2K Analysis

In this section we want to determine the effect of different parameters in our experimental setup. For this purpose, a $2^k r$ factorial experiment with three factors and three repetitions, i.e. a $2^3 3$ experiment, is performed. The goal of this section is to quantify the effect of changing three important parameters in the system for different workloads. In particular we want to investigate the following parameters:

- **A**: Memcached servers: 2 and 3
- **B**: Middlewares: 1 and 2
- **C**: Worker threads per MW: 8 and 32

Throughout this section, we will refer to these three parameters as factors A, B and C. The combined effects are referred to the combination of those three letters, for example the factor of the combined effects of the number of servers and the number of middlewares is referred to as AB.

The experiment is repeated for a write-only, a read-only and a 50-50-read-write workload. The experiment setup can be found in the following table:

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

6.1 Derivation

The mathematical foundation of a $2^k r$ analysis is beyond the scope of this report. However, this subsection aims to explain how the values throughout this section are calculated, using Tables 7 and 8 as examples. The formulæ and explanations are adopted from [1].

In general, a 2^3 analysis is done as follows: Suppose y_0 through y_7 are the eight observed values resulting from the observations of all combinations of the three factors. The model for a 2^3 factorial design is:

$$y = q_0 + q_A x_A + q_B x_B + q_C x_C + q_{AB} x_A x_B + q_{AC} x_A x_C + q_{BC} x_B x_C + q_{ABC} x_A x_B x_C$$

where the q's are the so-called effects of each factor and x_A, x_B and x_C are defined as follows:

$$x_A = \begin{cases} -1, & \text{if 2 servers} \\ 1, & \text{if 3 servers} \end{cases}, x_B = \begin{cases} -1, & \text{if 1 middleware} \\ 1, & \text{if 2 middlewares} \end{cases}, x_C = \begin{cases} -1, & \text{if 8 worker threads} \\ 1, & \text{if 32 worker threads} \end{cases}$$

For better visualization, all combinations of factors A, B and C and their resulting products can be illustrated in a sign table such as Table 7's top left part. Note that q_0 corresponds to column I and does not have a factor x_I, therefore the corresponding column contains all 1's. Instead of the letters of the factors, numbers can be used as subscripts of the q's in the order of their appearance in the sign table.

Since we repeat our experiments three times, we get three different results and we want to isolate the experimental error, i.e. our model evolves to be:

$$y = q_0 + q_A x_A + q_B x_B + q_C x_C + q_{AB} x_A x_B + q_{AC} x_A x_C + q_{BC} x_B x_C + q_{ABC} x_A x_B x_C + e$$

where e is the experimental error and the q's the effects from before. For experiment i and repetition j, this can be expressed as:

$$y_{ij} = q_0 + q_A x_{Ai} + q_B x_{Bi} + q_C x_{Ci} + q_{AB} x_{Ai} x_{Bi} + q_{AC} x_{Ai} x_{Ci} + q_{BC} x_{Bi} x_{Ci} + q_{ABC} x_{Ai} x_{Bi} x_{Ci} + eij$$

Parameter q_j can then be calculated as follows:

$$q_j = \frac{1}{8} \sum_i S_{ij} \overline{y_i}$$

where S_{ij} is the (i, j)th entry in the sign table and $\overline{y_i}$ the average of the three observed values.

Now, if one uses applies all eight combinations of the three factors to the equation above, the result is an equation system with eight equalities and eight unknowns (the q's). Solving this system gives us the values of the q's as shown in the last line of Table 7. q_I indicates the average value (in our example table, the average throughput) and q_x indicates how factor (or the combination of factors) x influences the result. For example, the average throughput in Table 7 is 20,092 and when using three instead of two servers (factor A), the throughput decreases by 1,457.

To quantify the importance of a factor, one uses the the proportion of the variation that is explained by this factor to the total variation. To calculate the variation, different sums of squares are calculated:

$$\begin{aligned} SSY &= \sum_{i,j} y_{ij}^2 \\ SS0 &= 2^3 * 3 * q_0^2 \\ SST &= SSY - SS0 \\ SSj &= 2^3 * 3 * q_j^2 \text{ for } j \in \{1..7\} \\ SSE &= \sum_{i,j} e_{ij}^2 \end{aligned}$$

SSY is the sum of squares of all measured values in all three repetitions, whereas $SS0$ is the average value multiplied by the number of experiments and the number of repetitions. SST is the difference between those two values and therefore the total variation. SSj is the sum of squares for each factor and SSE the sum of the squared errors. In our example, these numbers can be found in Table 8. To calculate the importance of factor j, one simply divides SSj through SST. The larger the fraction, the more important the factor.

6.2 Write-only workload

	I	**A**	**B**	**C**	**AB**	**AC**	**BC**	**ABC**	y_1	y_2	y_3	$\overline{y}$
	1	-1	-1	-1	1	1	1	-1	13,739	13,680	13,921	13,780
	1	1	-1	-1	-1	-1	1	1	11,658	12,287	12,634	12,193
	1	-1	1	-1	-1	1	-1	1	19,988	19,762	19,853	19,868
	1	1	1	-1	1	-1	-1	-1	17,201	17,215	17,460	17,292
	1	-1	-1	1	1	-1	-1	1	24,067	23,864	22,620	23,517
	1	1	-1	1	-1	1	-1	-1	19,992	19,753	20,098	19,947
	1	-1	1	1	-1	-1	1	-1	29,427	29,225	28,439	29,030
	1	1	1	1	1	1	1	1	24,990	25,286	25,046	25,107
q_x	20,092	−1,457	2,733	4,309	−167.74	−416.24	−64.29	79.43				

Table 7: Results of the $2^k r$ analysis for the throughput in ops/s and a write-only workload.

Component	**Sum of squares**	**Percentage of variation**
y	1.04×10^{10}	
$\overline{y}$	9.69×10^{9}	
y - $\overline{y}$	6.83×10^{8}	100
A	5.09×10^{7}	7.46
B	1.79×10^{8}	26.23
C	4.46×10^{8}	65.21
AB	675,300	0.1
AC	4.16×10^{6}	0.61
BC	99,183	0.01
ABC	151,408	0.02
Errors	2.47×10^{6}	0.36

Table 8: Variations for the $2^k r$ analysis for the throughput and a write-only workload.

When inspecting the different percentages each factor contributes to the variations in Table 8, one can clearly see that the factors A, B and C are dominant, whereas combinations of the factors and the errors are insignificant. As illustrated in Table 7, factors B and C, i.e. the number of middlewares and the number of worker threads, increase the throughput by 2,733 ops/s and 4,309 ops/s, respectively, when set to 1, whereas the number of servers decreases the throughput by 1,457 ops/s.

At first, it may seem counterintuitive that more servers decrease the throughput, however, the middlewares replicate writes to all servers, which leads to increased response times and therefore less throughput. On the other hand, increasing the number of middlewares and the number of worker threads increases the throughput, which is expected and has been observed in earlier sections.

Regarding the response time, the results are similar. Increasing the number of servers, increases the response time, whereas increasing the number of middlewares and worker threads decreases the response time as illustrated in Table 9, which also corresponds to the variations, as shown in Table 10. However, in contrast to the throughput, the combination of factors B and C, i.e. the number of middlewares and worker threads, contributes around 7 % to the variation, increasing the response time by 0.73 ms, even more than the number of servers. This is most likely due to the fact that two middlewares, each with 32 worker threads, can handle many

	I	A	B	C	AB	AC	BC	ABC	y_1	y_2	y_3	$\overline{y}$
	1	-1	-1	-1	1	1	1	-1	14.16	14.23	13.95	14.12
	1	1	-1	-1	-1	-1	1	1	16.6	15.75	15.33	15.89
	1	-1	1	-1	-1	1	-1	1	10.0	10.07	10.06	10.04
	1	1	1	-1	1	-1	-1	-1	11.53	11.53	11.32	11.46
	1	-1	-1	1	1	-1	-1	1	8.56	8.66	9.09	8.77
	1	1	-1	1	-1	1	-1	-1	10.1	10.17	10.03	10.1
	1	-1	1	1	-1	-1	1	-1	7.5	7.61	7.76	7.63
	1	1	1	1	1	1	1	1	8.59	8.56	8.63	8.59
q_x	10.82	0.69	−1.39	−2.05	−0.09	−0.11	0.73	0.0				

Table 9: Results of the $2^k r$ analysis for the response times in ms and a write-only workload.

Component	Sum of squares	Percentage of variation
y	2,986	
$\overline{y}$	2,812	
y - $\overline{y}$	173.62	100
A	11.31	6.52
B	46.65	26.87
C	101.15	58.26
AB	0.2	0.11
AC	0.3	0.17
BC	12.87	7.42
ABC	0.0	0.0
Errors	1.13	0.65

Table 10: Variations for the $2^k r$ analysis for the response times and a write-only workload.

requests simultaneously, which increases the load on the servers, which in turn increases their service time. However, clearly, the benefits of having two middlewares and 32 worker threads instead of one middleware and 8 worker threads outweigh this small negative impact on the performance resulting from the combination of factors B and C.

In conclusion, this experiment shows that for a write-only workload the most important factors are the number of middlewares and the number of worker threads. Increasing any of these numbers increases the throughput and decreases the response time. This can be expected, because this results in more resources processing requests. Since writes are replicated to all servers, more servers slightly decreases the performance, however not by much. The additional overhead on the servers when using many middlewares and worker threads also decreases performance slightly. All other factors and the errors have no significant impact. Inspecting Tables 7 and 9 shows that more middlewares and worker threads with a small number of servers yields the best performance.

6.3 Read-only workload

	I	A	B	C	AB	AC	BC	ABC	y_1	y_2	y_3	$\overline{y}$
	1	-1	-1	-1	1	1	1	-1	22,099	21,648	20,857	21,535
	1	1	-1	-1	-1	-1	1	1	20,050	20,095	21,245	20,463
	1	-1	1	-1	-1	1	-1	1	22,275	22,267	22,262	22,268
	1	1	1	-1	1	-1	-1	-1	33,364	33,328	33,364	33,352
	1	-1	-1	1	1	-1	-1	1	22,253	22,310	22,280	22,281
	1	1	-1	1	-1	1	-1	-1	32,345	31,891	31,000	31,745
	1	-1	1	1	-1	-1	1	-1	22,223	22,258	22,266	22,249
	1	1	1	1	1	1	1	1	33,356	33,350	33,448	33,385
q_x	25,910	3,827	1,904	1,505	1,728	1,323	−1,502	−1,311				

Table 11: Results of the $2^k r$ analysis for the throughput in ops/s and a read-only workload.

In contrast to the write-only workload, for a read-only workload, the number of servers

Component	Sum of squares	Percentage of variation
y	1.68×10^{10}	
$\overline{y}$	1.61×10^{10}	
y - $\overline{y}$	7.05×10^{8}	100
A	3.51×10^{8}	49.88
B	8.70×10^{7}	12.35
C	5.44×10^{7}	7.72
AB	7.17×10^{7}	10.18
AC	4.20×10^{7}	5.97
BC	5.41×10^{7}	7.68
ABC	4.12×10^{7}	5.85
Errors	2.65×10^{6}	0.38

Table 12: Variations for the $2^k r$ analysis for the throughput and a read-only workload.

(factor A), has by far the biggest impact on the throughput, as seen in Table 11. Since the middlewares only use one server to query for the value, this performance increase is intuitive, as the load can be spread across more servers and therefore the throughput can be increased. The other two factors, as well as the combination of them, all contribute between 5 and 12% to the variation as illustrated in Table 12, while the errors are not significant.

Also, all factors increase the throughput with the exception of ABC, the combination of all factors, and BC, the combination of the number of middlewares and the number of worker threads. Both of this increases are most likely due to the same reason we have seen for the write-only workload. The two middlewares and 32 worker threads can handle many request simultaneously, therefore increasing the load on the servers and increasing their service time. Increasing the number of servers as well (factor ABC), decreases the throughput less, because more servers are available to process the requests.

	I	**A**	**B**	**C**	**AB**	**AC**	**BC**	**ABC**	y_1	y_2	y_3	$\overline{y}$
	1	-1	-1	-1	1	1	1	-1	8.98	9.16	9.53	9.23
	1	1	-1	-1	-1	-1	1	1	9.85	9.91	9.29	9.68
	1	-1	1	-1	-1	1	-1	1	9.46	9.37	9.35	9.4
	1	1	1	-1	1	-1	-1	-1	6.19	6.18	6.17	6.18
	1	-1	-1	1	1	-1	-1	1	9.53	9.24	9.33	9.37
	1	1	-1	1	-1	1	-1	-1	6.91	6.96	7.8	7.22
	1	-1	1	1	-1	-1	1	-1	10.45	10.53	10.49	10.49
	1	1	1	1	1	1	1	1	6.53	6.5	6.64	6.56
q_x	8.52	−1.1	−0.36	−0.11	−0.68	−0.41	0.47	0.24				

Table 13: Results of the $2^k r$ analysis for the response time in ms and a read-only workload.

Component	Sum of squares	Percentage of variation
y	1,796	
$\overline{y}$	1,740	
y - $\overline{y}$	55.61	100
A	29.28	52.66
B	3.1	5.57
C	0.27	0.48
AB	11.18	20.1
AC	4.12	7.4
BC	5.39	9.69
ABC	1.34	2.4
Errors	0.94	1.7

Table 14: Variations for the $2^k r$ analysis for the response time and a read-only workload.

The response time shows similar results as the throughput. As depicted in Table 13, all factors decrease the response time, again with the exceptions of BC and ABC. This is most likely for the same reasons as with the throughput. The variations, illustrated in Table 14,

show slightly more accentuated values than for the throughput. For example, the percentage of variation for factors A and AB is much higher than with the throughput, whereas the factors B and C each drop around 7%, making factor C, i.e. the number of worker threads, insignificant for the response time.

The large percentage of variation for factor AB and corresponding decrease in response time is most likely due to the more resources available for processing requests and not having more overhead caused by more worker threads. This overhead is also noticeable when looking at the increased response time for factors BC and ABC, both having 32 worker threads instead of 8.

One can conclude that the number of servers is the most important factor for a read-only workload. This is due to the fact that the middlewares only query one server for a value and therefore increasing the number of servers decreases the load on each server and increases the overall performance. When inspecting the throughput and response time differences for each factor in Tables 11 and 13, one can see that increasing the number of middlewares and the number of worker threads also increases the performance, even though less significantly.

6.4 50-50-read-write workload

	I	**A**	**B**	**C**	**AB**	**AC**	**BC**	**ABC**	y_1	y_2	y_3	$\overline{y}$
	1	-1	-1	-1	1	1	1	-1	16,998	16,937	17,115	17,017
	1	1	-1	-1	-1	-1	1	1	16,052	15,268	15,515	15,612
	1	-1	1	-1	-1	1	-1	1	23,674	23,860	23,877	23,804
	1	1	1	-1	1	-1	-1	-1	22,620	22,538	22,360	22,506
	1	-1	-1	1	1	-1	-1	1	26,663	25,310	27,222	26,399
	1	1	-1	1	-1	1	-1	-1	25,482	25,026	24,455	24,988
	1	-1	1	1	-1	-1	1	-1	33,417	34,316	32,638	33,457
	1	1	1	1	1	1	1	1	30,588	32,002	30,576	31,055
q_x	24,355	−814.45	3,351	4,620	−110.46	−138.72	−69.47	−137.34				

Table 15: Results of the $2^k r$ analysis for the throughput in ops/s and a 50-50-read-write workload.

Component	**Sum of squares**	**Percentage of variation**
y	1.50×10^{10}	
$\overline{y}$	1.42×10^{10}	
y - $\overline{y}$	8.05×10^{8}	100
A	1.59×10^{7}	1.98
B	2.69×10^{8}	33.49
C	5.12×10^{8}	63.67
AB	292,818	0.04
AC	461,871	0.06
BC	115,810	0.01
ABC	452,724	0.06
Errors	5.61×10^{6}	0.7

Table 16: Variations for the $2^k r$ analysis for the throughput and a 50-50-read-write workload.

For a 50-50-read-write workload, we expect to see results between the results for read-only and write-only workloads. However, as illustrated in Tables 15 and Table 16, it is apparent that the number middlewares and the number of worker threads (factors B and C) make up almost all the variation of the throughput, whereas the other factors, interestingly also factor A, the number of servers, are insignificant. Given that these results resemble the variations for a write-only workload shown in Table 8, we can deduce that the writes use more resources in the system and therefore influence the result of a 50-50 workload much more than the reads. This makes sense, since writes are replicated to all servers, whereas for a read, only one server is

queried. Increasing the number of middlewares and the number of worker threads both increase the throughput, which is expected and matches the previous results.

	I	**A**	**B**	**C**	**AB**	**AC**	**BC**	**ABC**	y_1	y_2	y_3	$\overline{y}$
	1	-1	-1	-1	1	1	1	-1	11.66	11.67	11.53	11.62
	1	1	-1	-1	-1	-1	1	1	12.31	12.93	12.72	12.66
	1	-1	1	-1	-1	1	-1	1	8.82	8.73	8.78	8.78
	1	1	1	-1	1	-1	-1	-1	9.28	9.34	9.35	9.32
	1	-1	-1	1	1	-1	-1	1	8.1	8.44	8.06	8.2
	1	1	-1	1	-1	1	-1	-1	8.6	8.65	8.88	8.71
	1	-1	1	1	-1	-1	1	-1	7.28	7.05	7.21	7.18
	1	1	1	1	1	1	1	1	7.73	7.58	7.81	7.71
q_x	9.27	0.33	−1.02	−1.32	−0.06	−0.07	0.52	0.06				

Table 17: Results of the $2^k r$ analysis for the write response times in ms and a 50-50-read-write workload.

Component	**Sum of squares**	**Percentage of variation**
y	2,140	
$\overline{y}$	2,064	
y - $\overline{y}$	76.92	100
A	2.56	3.33
B	25.19	32.74
C	41.99	54.6
AB	0.08	0.1
AC	0.11	0.15
BC	6.49	8.43
ABC	0.1	0.12
Errors	0.4	0.52

Table 18: Variations for the $2^k r$ analysis for the write response times and a 50-50-read-write workload.

The response times for the writes show a similar picture as the throughput. As illustrated in Table 18, the number of worker threads is responsible for most of the variation and decreasing response time, followed by the number of middlewares. Again, the number of servers is insignificant and only slightly increases the response time, as seen in Table 17. However, for the write response time, the variation of the factor BC, i.e. the combination of the number of middlewares and worker threads, is at around 8%, similar to the write-only workload. This is most likely again due to the increased load which increases the service time of the servers. All other factors and the errors are insignificant.

	I	**A**	**B**	**C**	**AB**	**AC**	**BC**	**ABC**	y_1	y_2	y_3	$\overline{y}$
	1	-1	-1	-1	1	1	1	-1	11.49	11.49	11.34	11.44
	1	1	-1	-1	-1	-1	1	1	12.02	12.64	12.43	12.36
	1	-1	1	-1	-1	1	-1	1	8.44	8.35	8.39	8.39
	1	1	1	-1	1	-1	-1	-1	8.7	8.77	8.79	8.75
	1	-1	-1	1	1	-1	-1	1	7.75	8.01	7.69	7.81
	1	1	-1	1	-1	1	-1	-1	7.98	8.03	8.17	8.06
	1	-1	1	1	-1	-1	1	-1	6.6	6.37	6.68	6.55
	1	1	1	1	1	1	1	1	6.85	6.52	6.86	6.75
q_x	8.76	0.22	−1.16	−1.47	−0.08	−0.1	0.51	0.06				

Table 19: Results of the $2^k r$ analysis for the read response times in ms and a 50-50-read-write workload.

In contrast to the read-only workload, the influence of the number of servers for the read response times for the 50-50-read-write workload is almost zero, as seen in Tables 19 and 20. However, this is an almost identical behavior to the write response time. This is most likely

Component	Sum of squares	Percentage of variation
y	1,936	
$\overline{y}$	1,843	
y - $\overline{y}$	92.3	100
A	1.11	1.21
B	32.04	34.71
C	52.0	56.34
AB	0.14	0.15
AC	0.26	0.28
BC	6.22	6.74
ABC	0.1	0.11
Errors	0.43	0.46

Table 20: Variations for the $2^k r$ analysis for the read response times and a 50-50-read-write workload.

caused by the load of the writes dominating the system and therefore influencing the read requests as well, e.g. if they have to wait in the queue behind a write request.

In conclusion, once can see that for a 50-50-read-write workload, the systems behaves very similarly as for a write-only workload. The best results in terms of throughput and response time are again achieved with two middlewares and 32 worker threads, with 2 servers. However, the influence of the number of servers is smaller than for the write-only workload.

7 Queuing Model

In this section, we build various queuing models for our system. We start from a simple M/M/1 model (Section 7.1) and evolve via a M/M/m model (Section 7.2) to a network of queues (Section 7.4).

The terminology for the models, as well as the formulæ for the various metrics, are adopted from [1]. The proofs of the formulæ are beyond the scope of this report, however, they can also be found in [1]. Throughout this section, the maximum observed throughput will be used as the service rate ρ. The term "M/M/m model" is a special case of a queuing model with m servers where the time between successive arrival times and the service times are exponentially distributed, which is a reasonable assumption. Additionally, we assume having infinite buffer capacity, infinite population size and a first-come-first-serve (FCFS) service discipline as simplifications.

7.1 M/M/1

In this subsection, we will build an M/M/1 model of our system. This means that the entire system is modeled with one server and one queue. Therefore, the predictions will most likely not be very accurate, however, the model is easy to understand and a good starting point before using more complex models. The data in this section is based on Section 4 and we build a model for 8, 16, 32 and 64 worker threads each.

Before building our models, we first explain how the values used later are calculated. The input values of the model are the service rate ρ, for which the maximum observed throughput for each number of worker threads is used, and the arrival rate λ. We vary λ based on Table 6, i.e. we use different numbers of virtual clients for which the arrival rate (i.e. the throughput) was measured. All other values are derived as shown in Table 21.

The results of the M/M/1 model can be found in Tables 22 - 25. Note that the values measured and shown in Table 6 correspond to the saturated system for different number of worker threads, as explained earlier. In particular this means that for WT = 8, the measurements for

Metric	Formula
Traffic intensity ρ	λ/μ
Probability of 0 jobs in system p_0	$(1-\rho)$
Probability of $\geq n$ jobs in system	ρ^n
Mean number of jobs in system $E[n]$	$\rho/(1-\rho)$
Variance of number of jobs in system $Var[n]$	$\rho/(1-\rho)^2$
Mean number of jobs in queue $E[n_q]$	$\rho^2/(1-\rho)$
Variance of number of jobs in queue $Var[n_q]$	$\rho^2(1+\rho-\rho^2)/(1-\rho)^2$
Mean response time $E[r]$	$(1/\mu)/(1-\rho)$
Variance of response time $Var[r]$	$\frac{1/\mu^2}{(1-\rho)^2}$
q-Percentile of response time	$E[r]ln(\frac{100}{100-q})$
Mean waiting time $E[w]$	$\rho\frac{1/\mu}{1-\rho}$
Variance of waiting time $Var[w]$	$(2-\rho)\rho/(\mu^2(1-\rho)^2)$
q-Percentile of waiting time	$max(0, \frac{E[w]}{\rho}ln(\frac{100\rho}{100-q}))$

Table 21: Overview of how different metrics for the M/M/1 model are calculated for arrival rate λ and service rate ρ as input parameters (adopted from Box 31.1 in [1]).

Metric	VC = 1	VC = 4	VC = 10	VC = 16	VC = 32
Arrival rate λ	1798	6854	6837	6993	6984
Service rate μ	7021	7021	7021	7021	7021
Traffic intensity ρ	0.25614	0.97611	0.97371	0.996	0.99468
Prob. of 0 jobs in system	0.74386	0.02389	0.02629	0.004	0.00532
Prob. of ≥ 5 jobs in system	0.0011	0.88613	0.87528	0.98014	0.9737
Prob. of ≥ 10 jobs in system	0.0	0.78523	0.76611	0.96068	0.94809
Mean # of jobs in system	0.34434	40.86265	37.03582	248.76123	187.09454
Var. of # of jobs in system	0.46291	1711	1409	62131	35191
Mean # of jobs in queue	0.0882	39.88653	36.06211	247.76523	186.09986
Var. of # of jobs in queue	0.14116	1709	1407	62129	35189
Mean response time	0.19146	5.96214	5.41712	35.57137	26.7887
Var. of response time	0.03666	35.54712	29.34517	1265	717.63467
90-Percentile of response time	0.44086	13.72834	12.47338	81.90609	61.68327
99-Percentile of response time	0.88172	27.45667	24.94675	163.81219	123.36654
Mean waiting time	0.04904	5.81972	5.2747	35.42894	26.64628
Var. of waiting time	0.01637	35.52683	29.32488	1265	717.61438
90-Percentile of waiting time	0.18008	13.58419	12.32905	81.76339	61.54047
99-Percentile of waiting time	0.62094	27.31252	24.80242	163.66948	123.22374

Table 22: Results of the M/M/1 queuing model for different arrival rates for 8 worker threads. Rates are in ops/s and times in ms.

VC = 4 can be found in Table 6. Likewise the values for the pairs (WT = 16, VC = 10), (WT = 32, VC = 16) and (WT = 64, VC = 34) can be found in 6 as well. Naturally, values for different combinations were measured as well, however, we will restrict ourselves to compare the measured and compared values for these four cases.

In general, one can observe that, if the utilization is closer to 100%, the predicted values approach infinity (for unbounded variables). This lies in the nature of a queuing model because metrics like the response time and the number of jobs in the system increase unlimitedly when the utilization approaches 100% (which is also the reason why a system should never be put under 100% load).

When increasing the number of worker threads or decreasing the number of clients, which decreases the arrival rate, the response time and the wait time become smaller as expected. Also, when increasing the arrival rate, the probability of having more jobs in the system increases, which is intuitive.

One can see that the values predicted with the M/M/1 model are only a rough estimate of the measured values. Especially when the utilization approaches 100%, the difference between

Metric	VC = 1	VC = 4	VC = 10	VC = 16	VC = 32
Arrival rate λ	1802	7734	13771	13717	13737
Service rate μ	13859	13859	13859	13859	13859
Traffic intensity ρ	0.13004	0.55809	0.99364	0.98974	0.99125
Prob. of 0 jobs in system	0.86996	0.44191	0.00636	0.01026	0.00875
Prob. of ≥ 5 jobs in system	4×10^{-5}	0.05414	0.9686	0.94973	0.95701
Prob. of ≥ 10 jobs in system	0.0	0.00293	0.93819	0.90198	0.91587
Mean # of jobs in system	0.14948	1.26291	156.22515	96.43381	113.29105
Var. of # of jobs in system	0.17182	2.85785	24563	9396	12948
Mean # of jobs in queue	0.01944	0.70482	155.23151	95.44407	112.29979
Var. of # of jobs in queue	0.02487	1.98829	24561	9394	12946
Mean response time	0.08294	0.16328	11.34484	7.03049	8.24686
Var. of response time	0.00688	0.02666	128.7053	49.42786	68.01063
90-Percentile of response time	0.19098	0.37597	26.12245	16.18831	18.98909
99-Percentile of response time	0.38196	0.75195	52.2449	32.37662	37.97817
Mean waiting time	0.01079	0.09113	11.27268	6.95834	8.1747
Var. of waiting time	0.00167	0.02146	128.70009	49.42265	68.00543
90-Percentile of waiting time	0.02179	0.28074	26.05006	16.11578	18.91661
99-Percentile of waiting time	0.21277	0.65672	52.17251	32.3041	37.9057

Table 23: Results of the M/M/1 queuing model for different arrival rates for 16 worker threads. Rates are in ops/s and times in ms.

Metric	VC = 1	VC = 4	VC = 10	VC = 16	VC = 32
Arrival rate λ	1806	7734	14486	16838	18892
Service rate μ	18956	18956	18956	18956	18956
Traffic intensity ρ	0.09526	0.40802	0.76419	0.88826	0.9966
Prob. of 0 jobs in system	0.90474	0.59198	0.23581	0.11174	0.0034
Prob. of ≥ 5 jobs in system	1×10^{-5}	0.01131	0.26062	0.55296	0.98313
Prob. of ≥ 10 jobs in system	0.0	0.00013	0.06792	0.30577	0.96654
Mean # of jobs in system	0.10529	0.68924	3.24074	7.94922	293.37127
Var. of # of jobs in system	0.11638	1.16428	13.7431	71.13938	86360
Mean # of jobs in queue	0.01003	0.28122	2.47654	7.06097	292.37467
Var. of # of jobs in queue	0.01204	0.58979	12.39492	69.46212	86358
Mean response time	0.05831	0.08911	0.22371	0.4721	15.52896
Var. of response time	0.0034	0.00794	0.05005	0.22288	241.14844
90-Percentile of response time	0.13426	0.20519	0.51511	1.08705	35.75674
99-Percentile of response time	0.26852	0.41038	1.03023	2.17409	71.51348
Mean waiting time	0.00555	0.03636	0.17096	0.41935	15.4762
Var. of waiting time	0.00062	0.00516	0.04726	0.22009	241.14566
90-Percentile of waiting time	0.0	0.1253	0.45495	1.03111	35.7039
99-Percentile of waiting time	0.13143	0.33049	0.97006	2.11815	71.46064

Table 24: Results of the M/M/1 queuing model for different arrival rates for 32 worker threads. Rates are in ops/s and times in ms.

the predicted and the measured values becomes very large due to the afore-mentioned reasons.

For some metrics of particular interest, a comparison between the measured and the predicted values can be found in Section 7.3.

7.2 M/M/m

Next, the system is modeled with an M/M/m model, where each worker thread inside the middleware is represented as one server, i.e. m = number of worker threads. As before, the service rate μ is fixed at the maximum observed throughput and the arrival rate λ is varied, based on Table 6. The formulæ for calculating the values can be found in Table 26.

Since we do not measure the throughput of each worker thread separately, the service rate per worker was calculated by simply dividing the maximum observed throughput by m. This is of course not exactly true, however, it is a well enough approximation for our model.

The results of the M/M/m model can be found in Tables 27 - 30. We can observe that

Metric	VC = 1	VC = 4	VC = 10	VC = 16	VC = 32
Arrival rate λ	1842	7749	14220	17254	24182
Service rate μ	25652	25652	25652	25652	25652
Traffic intensity ρ	0.07182	0.30208	0.55436	0.67262	0.9427
Prob. of 0 jobs in system	0.92818	0.69792	0.44564	0.32738	0.0573
Prob. of ≥ 5 jobs in system	0.0	0.00252	0.05235	0.13767	0.74449
Prob. of ≥ 10 jobs in system	0.0	1×10^{-5}	0.00274	0.01895	0.55427
Mean # of jobs in system	0.07738	0.43282	1.24395	2.05457	16.45099
Var. of # of jobs in system	0.08336	0.62016	2.79136	6.27581	287.08607
Mean # of jobs in queue	0.00556	0.13074	0.68959	1.38194	15.50829
Var. of # of jobs in queue	0.00639	0.22683	1.92969	5.15077	285.25469
Mean response time	0.042	0.05586	0.08748	0.11908	0.6803
Var. of response time	0.00176	0.00312	0.00765	0.01418	0.46281
90-Percentile of response time	0.09671	0.12861	0.20142	0.27419	1.56645
99-Percentile of response time	0.19342	0.25723	0.40285	0.54837	3.13289
Mean waiting time	0.00302	0.01687	0.04849	0.08009	0.64132
Var. of waiting time	0.00024	0.0016	0.00613	0.01266	0.46129
90-Percentile of waiting time	0.0	0.06175	0.14982	0.22696	1.5263
99-Percentile of waiting time	0.08281	0.19036	0.35124	0.50115	3.09275

Table 25: Results of the M/M/1 queuing model for different arrival rates for 64 worker threads. Rates are in ops/s and times in ms.

Metric	Formula
Traffic intensity ρ	$\lambda/(m\mu)$
Probability of 0 jobs in system p_0	$(1 + \frac{(m\rho)^m}{m!(1-\rho)} + \sum_{n=1}^{m-1} \frac{(m\rho)^n}{n!})^{-1}$
Probability of queuing $\delta = P(\geq m$ jobs)	$p_0 \frac{(m\rho)^m}{m!(1-\rho)}$
Mean number of jobs in system $E[n]$	$m\rho + \rho\delta/(1-\rho)$
Variance of number of jobs in system $Var[n]$	$m\rho + \rho\delta(\frac{1+\rho-\rho\delta}{(1-\rho)^2} + m)$
Mean number of jobs in queue $E[n_q]$	$\rho\delta/(1-\rho)$
Variance of number of jobs in queue $Var[n_q]$	$\delta\rho(1+\rho-\delta\rho)/(1-\rho)^2$
Mean response time $E[r]$	$\frac{1}{\mu}(1 + \frac{\delta}{m(1-\rho)})$
Variance of response time $Var[r]$	$\frac{1}{\mu^2}(1 + \frac{\delta(2-\delta)}{m^2(1-\rho)^2})$
Mean waiting time $E[w]$	$\delta/(m\mu(1-\rho))$
Variance of waiting time $Var[w]$	$\delta(2-\delta)/(m^2\mu^2(1-\rho)^2)$
q-Percentile of waiting time	$max(0, \frac{E[w]}{\delta} ln(\frac{100\delta}{100-q}))$

Table 26: Overview of how different metrics of the M/M/m model are calculated for arrival rate λ, service rate *per worker* ρ and number of worker threads m as input parameters (adopted from Box 31.2 in [1]).

the values are overall closer to the measured values, however, when the utilization is close to 100%, the difference between measured and predicted values is still large for some metrics. As discussed in Section 7.1, this is due to the unbounded increase of certain variables when approaching 100% utilization.

Overall, the intuition that with more worker threads the queue is shorter and therefore the wait time and the response time are smaller is clearly reflected in Tables 27 - 30. Also, when increasing the number of virtual clients and therefore the arrival rate, the inverse is true, i.e. the response time and the wait time become larger. When having less worker threads or more clients, the probability of queuing is larger as well.

7.3 M/M/1 vs. M/M/m

To compare the measurements with the predictions, we fix the number of clients and increase the number of worker threads. This results in a mostly constant arrival rate, whereas the utilization decreases when increasing the number of worker threads, i.e. m. The comparison can be found

Metric	VC = 1	VC = 4	VC = 10	VC = 16	VC = 32
Arrival rate λ	1798	6854	6837	6993	6984
Service rate μ per worker	877.67656	877.67656	877.67656	877.67656	877.67656
Number of services m	8.0	8.0	8.0	8.0	8.0
Traffic intensity ρ	0.25614	0.97611	0.97371	0.996	0.99468
Prob. of 0 jobs in system	0.12884	6×10^{-5}	7×10^{-5}	1×10^{-5}	1×10^{-5}
Prob. of queuing	0.00134	0.92394	0.91645	0.98705	0.98282
Mean # of jobs in system	2.0496	45.56346	41.73107	253.50727	191.83765
Var. of # of jobs in system	2.05265	1713	1411	62133	35194
Mean # of jobs in queue	0.00046	37.75457	33.9414	245.5393	183.88018
Var. of # of jobs in queue	0.00078	1698	1396	62117	35178
Mean response time	1.13963	6.64802	6.10388	36.25002	27.46783
Var. of response time	1.29827	36.63963	30.43848	1266	718.72101
Mean waiting time	0.00026	5.50865	4.96451	35.11065	26.32846
Var. of waiting time	0.0001	35.34146	29.14031	1265	717.42285
90-Percentile of waiting time	0.0	13.25667	12.00073	81.44237	61.21903
99-Percentile of waiting time	0.0	26.98501	24.47411	163.34846	122.9023

Table 27: Results of the M/M/8 queuing model for different arrival rates for 8 worker threads. Rates are in ops/s and times in ms.

Metric	VC = 1	VC = 4	VC = 10	VC = 16	VC = 32
Arrival rate λ	1802	7734	13771	13717	13737
Service rate μ per worker	866.17135	866.17135	866.17135	866.17135	866.17135
Number of services m	16.0	16.0	16.0	16.0	16.0
Traffic intensity ρ	0.13004	0.55809	0.99364	0.98974	0.99125
Prob. of 0 jobs in system	0.12485	0.00013	0.0	0.0	0.0
Prob. of queuing	0.0	0.02334	0.97033	0.95236	0.95931
Mean # of jobs in system	2.08066	8.95893	167.48782	107.67597	124.54118
Var. of # of jobs in system	2.08066	9.2409	24568	9401	12953
Mean # of jobs in queue	0.0	0.02947	151.58958	91.84018	108.68117
Var. of # of jobs in queue	0.0	0.10305	24536	9370	12922
Mean response time	1.15451	1.15832	12.16271	7.8501	9.06579
Var. of response time	1.33288	1.33411	129.92486	50.64858	69.23091
Mean waiting time	0.0	0.00381	11.00821	6.6956	7.91129
Var. of waiting time	0.0	0.00123	128.59198	49.3157	67.89803
90-Percentile of waiting time	0.0	0.0	25.78073	15.84518	18.6465
99-Percentile of waiting time	0.0	0.13838	51.90318	32.03349	37.63559

Table 28: Results of the M/M/16 queuing model for different arrival rates for 16 worker threads. Rates are in ops/s and times in ms.

Metric	VC = 1	VC = 4	VC = 10	VC = 16	VC = 32
Arrival rate λ	1806	7734	14486	16838	18892
Service rate μ per worker	592.38385	592.38385	592.38385	592.38385	592.38385
Number of services m	32.0	32.0	32.0	32.0	32.0
Traffic intensity ρ	0.09526	0.40802	0.76419	0.88826	0.9966
Prob. of 0 jobs in system	0.04743	0.0	0.0	0.0	0.0
Prob. of queuing	0.0	1×10^{-5}	0.10111	0.41161	0.97715
Mean # of jobs in system	3.04841	13.05652	24.78183	31.69628	318.55853
Var. of # of jobs in system	3.04841	13.05662	29.27101	84.70988	86371
Mean # of jobs in queue	0.0	0.0	0.32769	3.27201	286.66724
Var. of # of jobs in queue	0.0	1×10^{-5}	2.3442	44.5858	86308
Mean response time	1.68809	1.6881	1.71072	1.88242	16.86219
Var. of response time	2.84966	2.84966	2.85927	2.99538	243.87218
Mean waiting time	0.0	0.0	0.02262	0.19432	15.17409
Var. of waiting time	0.0	0.0	0.00961	0.14572	241.02252
90-Percentile of waiting time	0.0	0.0	0.00248	0.66798	35.39776
99-Percentile of waiting time	0.0	0.0	0.51759	1.75502	71.1545

Table 29: Results of the M/M/32 queuing model for different arrival rates for 32 worker threads. Rates are in ops/s and times in ms.

Metric	VC = 1	VC = 4	VC = 10	VC = 16	VC = 32
Arrival rate λ	1842	7749	14220	17254	24182
Service rate μ per worker	400.81152	400.81152	400.81152	400.81152	400.81152
Number of services m	64.0	64.0	64.0	64.0	64.0
Traffic intensity ρ	0.07182	0.30208	0.55436	0.67262	0.9427
Prob. of 0 jobs in system	0.01009	0.0	0.0	0.0	0.0
Prob. of queuing	0.0	0.0	1×10^{-5}	0.00182	0.53959
Mean # of jobs in system	4.59646	19.33287	35.47889	43.05149	69.20944
Var. of # of jobs in system	4.59646	19.33287	35.47931	43.14493	315.0321
Mean # of jobs in queue	0.0	0.0	1×10^{-5}	0.00373	8.87686
Var. of # of jobs in queue	0.0	0.0	5×10^{-5}	0.01904	222.14442
Mean response time	2.49494	2.49494	2.49494	2.49515	2.86202
Var. of response time	6.22472	6.22472	6.22472	6.22477	6.58942
Mean waiting time	0.0	0.0	0.0	0.00022	0.36709
Var. of waiting time	0.0	0.0	0.0	5×10^{-5}	0.3647
90-Percentile of waiting time	0.0	0.0	0.0	0.0	1.14674
99-Percentile of waiting time	0.0	0.0	0.0	0.0	2.71319

Table 30: Results of the M/M/64 queuing model for different arrival rates for 64 worker threads. Rates are in ops/s and times in ms.

Metric	VC=4 WT=8	VC=4 WT=16	VC=4 WT=32	VC=4 WT=64
Arrival rate λ	6854	7734	7734	7749
Traffic intensity ρ	0.976	0.558	0.408	0.302
Response time M/M/1 predicted	5.962	0.163	0.089	0.056
Response time M/M/m predicted	6.648	1.158	1.688	2.495
Response time measured	2.678	2.283	2.293	2.287
Variance of response time M/M/1 predicted	35.547	0.027	0.008	0.003
Variance of response time M/M/m predicted	36.64	1.334	2.85	6.225
Variance of response time measured	0.001	0.001	0.0	0.001
Queue length M/M/1 predicted	39.887	0.705	0.281	0.131
Queue length M/M/m predicted	37.755	0.029	0.0	0.0
Queue length measured	2.175	2.176	2.184	2.179
Queue wait time M/M/1 predicted	5.82	0.091	0.036	0.017
Queue wait time M/M/m predicted	5.509	0.004	0.0	0.0
Queue wait time measured	0.467	0.067	0.069	0.068

Table 31: Comparison of measurements and predictions of the M/M/1 and M/M/m queuing model for different numbers worker threads and 4 virtual clients. m is the number of worker threads. Rates are in ops/s and times in ms.

in Table 31.

As observed before, a utilization close to 100% yields inaccurate predictions, which can be seen for $m = 8$. For this reason, this comparison will only deal with the other three cases (i.e. 16, 32 and 64 worker threads), since those values allow a more meaningful comparison.

Regarding the response time, the predicted values of the M/M/m models are much closer to the measured values than the predictions of the M/M/1 model, almost being exact for $m = 64$. On the other hand, the variance of the response time is predicted more accurately by the M/M/1 model. The same holds for the queue wait time, for which the M/M/1 model clearly does a better job at predicting the values than the M/M/m model. The queue length is predicted unsatisfyingly by both models, however, the M/M/1 predict values slightly closer to the measured values.

In conclusion, it is hard to determine the "better" model. If one is just interested in the response time, the M/M/m model seems to be more accurate. On the other hand, for measurements regarding the queue, the M/M/1 model might be the better option. Usually, the response time is the more important metric, since the predictions of internal metrics (like the queue wait time) are of less interest, if the response time and with it the throughput is satisfiable. However,

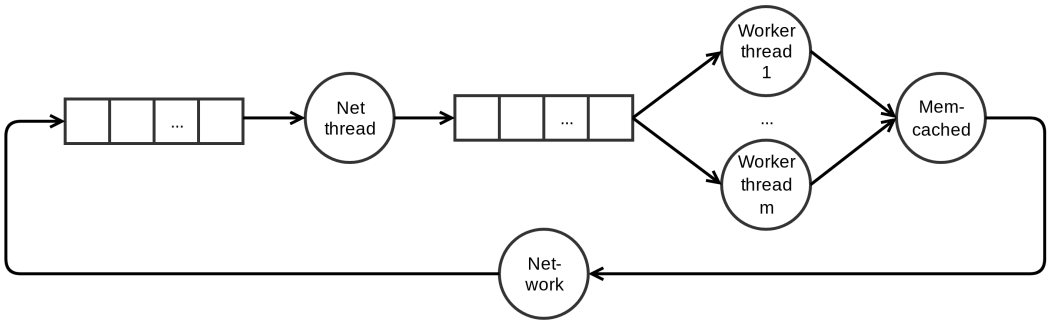


Figure 28: A schematic overview of the queuing network used for the extended MVA. The net thread is modeled as an M/M/1 system, the worker threads as an M/M/m system and the server and the network as delay centers.

internal metrics might indicate general issues with the system, e.g. if the probability of queuing is high, even for a low arrival rate.

In any case, factors which were not modeled, either internal (like the service time of the servers) or external to the middleware (like the network latency) also influence the behavior of the system. This is an inherent drawback of the simplifications a single queuing model makes. To overcome these limitations, a network of queues can be used to model the system more accurately, as described in the next section.

7.4 Network of Queues

In this subsection, we model the entire system as a network of queues. The model, as well as the input data used to predict values, is based on Section 3.1.

7.4.1 Design

For the queuing network, we use three different kinds of service centers: Delay centers, fixed-capacity centers and load-dependent centers. In our model these correspond to M/M/∞, M/M/1 and M/M/m systems. A schematic overview of the model can be found in Figure 28. As illustrated, the net thread is modeled as an M/M/1 system, the worker threads as an M/M/m system and the single server, as well as the network, as delay centers.

Modeling the net thread as an M/M/1 system is intuitive, since there is only a single thread accepting messages. Since the network is a system with fixed delay, we can model it as a delay center, i.e. an M/M/∞ system. Naturally, this is not true for arbitrary large loads on the network, however, we did not notice any significant changes in the network delay during an experiment.

On the other hand, modeling the worker threads and the server is not trivial. As explained in Section 1.9, we only measure the total response time, the queue wait time and the service time of the servers. Therefore, we cannot easily model the server as a fixed-capacity or load-dependent center, because the service time of the server is contained within the 6service time of a worker thread, which we did not measure. However, when inspecting Figures 9 and 11, one can see that the service time is mostly constant in those experiments (with the exception of 64 worker threads and a read-only workload), thus we decided to model the server as a delay center. The worker threads are modeled as an M/M/m system with m being the number of worker threads, as in Section 7.2.

Metric	VC = 4 WT = 8	VC = 32 WT = 8	VC = 4 WT = 32	VC = 32 WT = 32
Throughput predicted	2,977	4,252	2,953	20,524
Throughput measured	3,049	3,721	3,031	11,124
Response time net thread predicted	0.05	0.054	0.049	0.355
Response time net thread measured	0.035	0.028	0.034	0.022
Response time worker threads predicted	0.05	12.396	0.054	0.054
Response time worker threads measured	0.105	14.525	0.101	2.271
Response time server predicted	2.066	2.066	2.084	2.084
Response time server measured	1.961	2.117	1.98	2.838
Response time network predicted	0.52	0.537	0.522	0.625
Response time network measured	0.52	0.537	0.522	0.625
Total response time predicted	2.687	15.053	2.709	3.118
Total response time measured	2.621	17.208	2.637	5.756
Utilization of net thread predicted	0.131	0.186	0.127	0.879
Utilization of worker threads predicted	0.138	0.865	0.147	0.669
Request queue length predicted	0.149	52.701	0.159	1.105
Request queue length measured	1.961	2.117	1.98	2.838

Table 32: The measured data and the data predicted by the queuing model for a read-only workload. Rates are in ops/s and times in ms. In this experiment, NumClients = 2 * VC.

To circumvent the issue of not having measured the service time of the worker threads, we make the assumption that the queue wait time accounts for most of the service time of the worker threads, of course excluding the service time of the servers, but this was measured. Therefore, the delay between taking an element out of the queue and sending it to the server, as well as the delay between getting a response from the server and sending it to the client does not contribute to the service time of the worker threads. However, these delays are minor and as one can see in the next subsection, the model is still accurate. The service time of the net thread is calculated by subtracting the queue wait time and the service time of the servers from the total response time of the middleware. Finally, the network delay can be calculated by subtracting the response time measured on the middleware from the response time measured on the clients.

For evaluating the model and calculating the predictions, we use the extended MVA algorithm, which can be found in Box 36.1 in [1].

7.4.2 Evaluation

Since the experiment in Section 3.1 was run for a read-only and write-only workload, we model the system for both workloads. A comparison of the predicted and the measured values can be found in Table 32 for the read-only workload and in Table 33 for the write-only workload. For both workloads, the system is modeled for 4 and 32 virtual clients, with 8 and 32 worker threads.

For both experiments, there are many similarities in terms of the accuracy of the predictions. The predictions of network delay and the service time of the servers tend to be very close to the measured values. This is due to both of these components being modeled as delay centers, i.e. the response time is not expected to change when varying the number of clients or the number of worker threads. Any differences between the measurements and the predictions are due to the component not behaving exactly like a delay center.

The predictions of the queue lengths for 32 virtual clients and 8 worker threads are very inaccurate. This may be due to an inherent difficulty to predict queue lengths, as we have seen very inaccurate predictions of queue lengths for the M/M/1 and the M/M/m model, or caused by the measurement technique used, as described in Section 1.9.1.

The predicted throughputs, and with it the predicted response times for all components,

Metric	VC = 4 WT = 8	VC = 32 WT = 8	VC = 4 WT = 32	VC = 32 WT = 32
Throughput predicted	2,894	4,228	2,871	20,645
Throughput measured	3,014	3,702	3,056	14,457
Response time net thread predicted	0.052	0.056	0.045	0.234
Response time net thread measured	0.035	0.029	0.036	0.024
Response time worker threads predicted	0.046	12.411	0.049	0.049
Response time worker threads measured	0.102	14.62	0.102	1.603
Response time server predicted	2.144	2.144	2.168	2.168
Response time server measured	1.993	2.13	1.953	2.156
Response time network predicted	0.523	0.526	0.525	0.65
Response time network measured	0.523	0.526	0.525	0.65
Total response time predicted	2.764	15.137	2.787	3.1
Total response time measured	2.653	17.306	2.616	4.432
Utilization of net thread predicted	0.131	0.191	0.115	0.828
Utilization of worker threads predicted	0.123	0.86	0.131	0.635
Request queue length predicted	0.132	52.475	0.14	1.007
Request queue length measured	1.993	2.13	1.953	2.156

Table 33: The measured data and the data predicted by the queuing model for a write-only workload. Rates are in ops/s and times in ms. In this experiment, NumClients = 2 * VC.

are very close to the measured values, with the exception of the experiment with 32 virtual clients and 32 worker threads. However, the cause is different for the two workloads. For the write-only workload, the difference is caused only by an inaccurate prediction of the worker thread service time, whereas for the read-only workload, additionally, the service time of the server is predicted poorly. This is most likely caused by the issues with modeling the worker threads and the servers, explained in Section 7.4.1.

For both workloads, we can observe an increased utilization of the net thread and the worker threads when increasing the number of clients, which is expected. For 32 virtual clients and 8 worker threads, the utilization of the worker threads is around 86% for both workloads, making them the bottleneck. This corresponds to the measurements in Section 3.1. When increasing the number of worker threads to 32 and again using 32 virtual clients, the net thread now has the highest utilization, making it the bottleneck. Note that the server and the network as delay centers do not have a meaningful utilization value, since the are modeled to have an infinite number of servers.

In conclusion, the network of queues predicts data much better than the M/M/1 and M/M/m models. This can be expected, since the model is much more involved and requires more detailed input values and with it a deeper understanding of the system. With more time measurements within the middleware, an even more accurate model might be possible.

References

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