Advanced Systems Lab Report

Autumn Semester 2018

Name: Florian Morath
Legi: 14-931-968

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

1 System Overview (75 pts)

In this section I justify the design choices made for the implementation of the given system description. The goal was to build an efficient and stable system while still keeping it simple to analyze.

1.1 Overall architecture

The main components of the system are the net-thread (`NetThread.java`), the requests (`Request.java`) and the worker threads (`WorkerThread.java`), which will be explained in more detail in the following subsections.

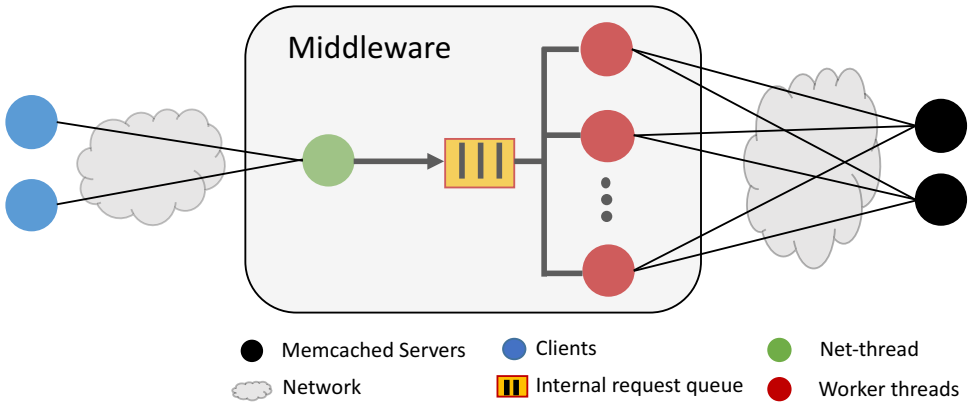

Figure 1: Interaction between components of the system.

Figure 1 gives a simplified overview of the system. First, a client sends a request over a TCP connection to the net-thread. The net-thread reads the message into a buffer, encapsulates it into a *Request* object and puts it into an internal request queue. An available worker thread takes out a request from the queue (if there is one) and processes it depending on its type. The request is sent over a TCP connection to one or multiple servers, after which their response is read into a buffer, then processed and sent back to the client.

`RunMW.java` is the entry point of the system. It parses the command line arguments, such as the addresses of the memcached servers, the address of the net-thread and the number of worker threads in the system. It starts the Middleware by creating an instance of the *Middleware* class, which then starts the worker threads and the net-thread.

1.2 Net-Thread

The `NetThread` class extends the *Thread*[1] class. The net-thread basically does two things: It establishes client connections and puts received requests into a queue.

A *Selector*[2] is used to handle multiple connections with a single thread. A selector can monitor multiple connections for certain I/O events and notify the net-thread when they occur. At first, the net-thread creates a server socket channel which it binds to its address. This channel's sole purpose is to accept new connections from clients. It is registered with the selector for

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

"accept events", i.e. the selector monitors incoming connections on the server socket channel. When this event happens, the net-thread gets notified by the selector and creates a new socket channel for the connection to the client. New socket channels are registered with the selector for "read events", i.e. the selector monitors when data from the clients is ready to be read from. Again, when this event occurs, the net-thread gets notified and reads the data. Note that the net-thread has to call the blocking *select()* method on the selector to be notified for those events.
This design allows the net-thread to monitor multiple channels for I/O events at the same time. It also enables non-blocking I/O between the clients and the net-thread without having the net-thread to constantly poll the medium and wasting CPU resources.

We want to minimize the number of times we allocate *ByteBuffer*[3] objects and reuse them as often as possible to minimize both data copy operations and memory allocation on the heap. The net-thread only needs to allocate one buffer per client connection because we have a closed system, i.e. clients wait for a response before sending the next request. Therefore, before sending a response to the client, we just clear the buffer associated with this client connection such that it can be reused again for a new request from this client. Since we have a fixed number of clients, the number of buffer allocations by the net-thread is also fixed and does not grow with the number of request.
We associate a buffer with each client connection using the *SelectionKey*[4] object. A selection key is created whenever we register a channel with a selector. Most importantly, it contains the channel for which the key was created and an attachment where we can store any object. We use this attachment to associate a buffer with each client connection.

Whenever a full request has been read into a byte buffer by the net-thread, it encapsulates it in a *Request* object and puts it into the internal request queue. More details about the *Request* class and the queue will be given in the next subsection.

1.3 Requests

There are several reasons why we have chosen a *LinkedBlockingQueue*[5] to enqueue and dequeue requests. First of all, we use a blocking queue because the *take()* operation should block until an element becomes available, which is more efficient (in terms of CPU usage) than constantly polling the queue and checking if an element is in there.
We use the linked implementation (in contrast to the array implementation) because of the following reasons: We have constant time for the take() and put() operation because doubly-linked nodes are used as an internal data structure. There is a separate lock for the head and the tail, which may result in better throughput because we can add and remove elements simultaneously. The default capacity is `Integer.MAX_VALUE`, which can be considered unbounded. That's good because we don't know how many requests may be in the queue and also we don't have to allocate space for requests a priori as for the *ArrayBlockingQueue*.
However, there is a disadvantage in choosing the linked blocking queue, namely that the performance may be more variable than with an array blocking queue because of the dynamic allocation of nodes during usage and the more complicated data structure with doubly-linked nodes.

[3] `https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html`
[4] `https://docs.oracle.com/javase/8/docs/api/java/nio/channels/SelectionKey.html`
[5] `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html`

Requests from clients are encapsulated in *Request* objects. The *Request* class contains the following fields:

- **buffer**: The byte buffer into which the request from the client was written into.
- **key**: A selection key which contains the channel over which the request was sent to the net-thread. This channel is used by the worker threads to be able to send the response back to the correct client.
- **type**: The type of the request which is either a Set, a Get (includes Multiget) or an Invalid request.
- **instrumentation fields**: See subsection 1.6.

The constructor of the *Request* class calls a method *parseRequest()*, which first checks if there is an end of line string `\r\n` at the last two written bytes of the byte buffer. If there is none, the request is considered Invalid. Then the type of the request is determined based on the first three chars of the request which should be either 'get' or 'set'. Note that the distinction between Multiget and Get requests is not done at this point in order to avoid code duplication and keeping the code simple.

1.4 Worker Threads

The `WorkerThread` class extends the *Thread*[6] class. The role of the worker threads is basically to establish connections to the memcached servers and to process the requests that the net-thread added to the internal request queue.

The first thing a worker thread does is to establish a permanent connection to each memcached server. The addresses of those servers are passed as an argument to the middleware at start-up. A worker thread opens a socket channel for each server and connects it to the server. The socket channels are stored in an array list, in order to be able to access them later on. The channels are configured to be blocking to keep the code simple and to not waste CPU resources while polling until a whole response has been read from a server. However, it also has disadvantages like failure during blocking calls leading to unresponsiveness and scalability issues if we have a lot of open connections, which in our case we don't have because we can have at maximum three servers.

For the buffer allocations, once again, we want to use as few allocations as possible and reuse them. This is achieved by having each worker thread allocating one buffer and referencing it with a field. This buffer is used to write the response from the servers into. If a request is done being processed, the buffer can be cleared in order to be reused again. Note that because of the sharded mode we actually allocate another buffer to make the reassembly of the responses easier, which will be explained later in more detail. But still the number of buffers doesn't grow with the number of requests and stays proportional to the number of worker threads.

The main loop of the worker thread consists of taking out a request from the queue (this is a blocking operation), processing it depending on its type and when finished taking out another one and so forth.
If the type of the request is `SET` then it will call *handleSetRequest()*, if it is `GET` then it will call *handleGetReuquest()* and if it is `INVALID`, it will send the error message "`ERROR\r\n`" to the client having sent the request.

[6] `https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html`

1.4.1 Handle a Set Request

For a Set request, the worker thread sends it to each server and then waits for a response of all of them. If one or multiple servers respond with an error message, one of them is forwarded to the client. If no error occurred, one of the success messages is forwarded to the client.
This is implemented as follows: The buffer contained in the *Request* object is sent sequentially to each server by iterating over the socket channels. After that, we go over the socket channels again and for each socket channel we write the response into our buffer and check the content in order to find out if the Set request was successfully executed on the server. If it was not, the error message is added to an array list. Note that we always read from a channel until we get an end of line string, which denotes that a full response has been read. And also note that the buffer is cleared before writing into it, such that we can reuse it and not have to allocate a new one for each response. After having processed all responses, if there occurred an error, we just forward the first error message to the client and otherwise the last success message is forwarded to the client because it is still contained in the buffer.

1.4.2 Handle a Get Request

For the **non-sharded mode**, we just forward the Get request (includes Multiget requests) to one server, which is chosen based on a round-robin scheme, and the response from the server is forwarded to the client. We don't have to inspect the response because independently of success or error, it can always be forwarded to the client.
The round-robin scheme works by having a private field *'lastServerIndex'* in each worker thread, which identifies the last server used by this worker thread for a Get request. Whenever a worker thread sends a (non-sharded) Get request, it executes the following code to retrieve the socket channel it uses to write the request to. Note that *'socketChannels'* is an array list containing all established socket channels to the servers.

```
lastServerIndex = (lastServerIndex + 1) % socketChannels.size();
SocketChannel socketChannel = socketChannels.get(lastServerIndex);
```

The empirical evaluation of the load balancing and the justification of this design choice is given in subsection 1.5.

For the **sharded mode**, a Multiget request has to be evenly split into a set of smaller Multiget requests, one for each server. Note that if there are fewer keys in the request than servers, we need to load balance them such that whenever this case occurs, not always the same servers are hit. To simplify the explanation, we ignore this case for the moment.
First we extract the keys from the request and put them into the *'keys'* array. There can be between 1 and 10 keys. Each server is identified with an index which corresponds to the position of its socket channel in the *'socketChannels'* array. We then need to figure out the number of keys each server should handle such that they are evenly distributed and sum up to the total number of keys in the *'keys'* array. We do this based on the index which identifies a server:

```
public static int getKeyCount(int index, int numKeys, int serverCount) {
    int keyCount = 0;
    for (int i = index; i < numKeys; i += serverCount) {
        keyCount++;
    }
    return keyCount;
}
```

getKeyCount() is a method that returns the number of keys for which a server with a specific index is responsible for. *'numKeys'* is the total number of keys in the request and *'serverCount'* is the total number of servers. It works by counting how many times we can add the 'serverCount' to the index until we exceed the 'numKey'. For example if we have 3 servers and 7 keys, index 0 gets 3 keys, index 1 gets 2 keys and index 2 gets 2 keys which sums up to 7 keys and is evenly distributed.
For each server we do the following: Construct the Get request based on its *'getKeyCount()'*. The keys are retrieved from the *'keys'* array in-order, i.e. the server with index 0 retrieves the first n elements, then the server with index 1 retrieves the next m elements and so on.
After all Get requests have been sent, we retrieve the responses from the servers in the same order as we have sent the requests, which simplifies the reassembly of the responses. The responses are reassembled into one response in a separate buffer for simplicity. In case of an error message, an error flag is raised and in the end '`ERROR\r\n`' is sent to the client instead of the reassembled response.
What is left to explain is how we handle the case with fewer keys than servers. Since we don't want to always hit the first n servers in the *'socketChannels'* array, where n is the number of keys, we apply the same round-robin scheme as in the non-sharded mode. But now we need to keep track of which servers we queried and in which order we did it, in order to be able to reassemble the responses in the correct order. For this we use an array called *'usedServers'*. The value of *'usedServers'* at position i tells if we did send a request to a server at iteration i, and if yes to which we did send one.

1.5 Load Balancing

To denote the last server used in the round-robin scheme, we have chosen to use a private field *'lastServerIndex'* in each worker thread instead of a global variable because it avoids having to synchronize access on it. Having all worker threads access the same variable for every Get request concurrently would lead to blocking behavior, which degrades the performance.

We did an empirical evaluation to show that on average all memcached servers are subject to the same load. The experiment was run on the cluster with 1 client, 4 worker threads and 3 memcached servers with Get requests only. Each worker thread printed how many times it hit each of the servers:

```
[Thread-2] ch.ethz.asl.WorkerThread: [0=16208, 1=16208, 2=16208]
[Thread-4] ch.ethz.asl.WorkerThread: [0=15208, 1=15209, 2=15208]
[Thread-5] ch.ethz.asl.WorkerThread: [0=15753, 1=15754, 2=15753]
[Thread-3] ch.ethz.asl.WorkerThread: [0=16307, 1=16308, 2=16308]
```

Server 0 was hit 63'476 times, server 1 was hit 63'479 times and server 2 was hit 63'477 times which shows that the load is distributed equally among the servers.

1.6 Instrumentation

The following timestamps are collected during the lifetime of a request:

- **timeFirstByte**: Time at which the first byte of the request is read by the net-thread.
- **timeEnqueued**: Time at which the request is enqueued into the internal request queue by the net-thread.
- **timeDequeued**: Time at which a worker thread dequeues the request from the internal request queue.
- **timeMemcachedSent**: Time immediately before the worker thread sends the (first) request to a memcached server.

- **timeMemcachedReceived**: Time immediately after the (last) response of a memcached server was read by the worker thread.
- **timeCompleted**: Time at which the worker thread sends the response back to the client.

All those timestamps, the queue length after the request is dequeued and the request type is stored in the *Request* object itself. After the request is completed, the worker calls the method *writeLogLine()* on it, which writes the instrumentation data to a csv-file. Note that we use *Log4j* for logging, which is configured to not immediately flush on each *log()* invocation, but buffers the logs until its capacity of 8'192 bytes is reached. We compared experiments with and without logging in order to check that logging does not affect the performance of the middleware.

Aggregation of log-files is outsourced to python scripts which run after the experiments are finished. A more detailed explanation of the postprocessing can be found in the *README* files of the repository.

2 Baseline without Middleware (75 pts)

The purpose of this section is to study the performance characteristics of the *memtier* clients and *memcached* servers.
The experimental setup is as follows: We let each experiment run for 80s and each experiment is repeated 3 times. The standard deviation of a specific metric is computed over those repetitions and shown as red error bars in the plots. Note that if the standard deviation is very small, one might not be able to see the red error bars in the plots. Throughput and response time are based on the output of *memtier*, whereas other metrics like the CPU utilization are based on the output of *dstat*.

2.1 One Server

In this setup we have 3 memtier instances (each installed on a single VM) connected to a single memcached instance. The overview of the experiment parameters is given 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	{2,4,8,16,24,32,40}
Workload	Write-only and Read-only
Repetitions	3

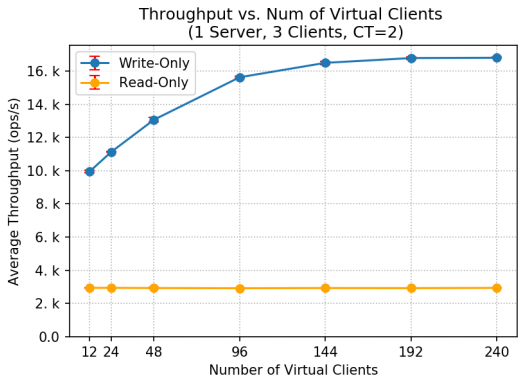

Figure 2: Average throughput.

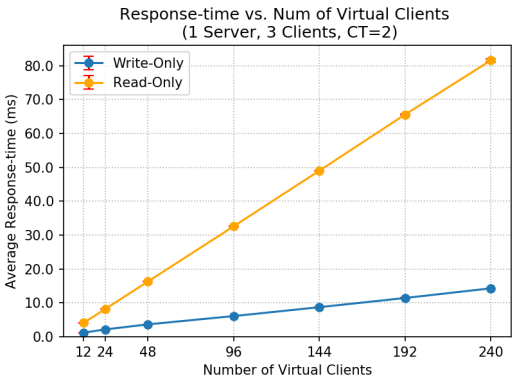

Figure 3: Average response time.

The throughput as a function of virtual clients for read-only and write-only workloads is given in Figure 2, whereas the response time as function of virtual clients for read-only and write-only workloads is given in Figure 3.

The interactive response time law relates throughput and response time in a closed system. It states that $R = N/X - Z$ with response time R, throughput X, think time Z and N number of clients. Z is often assumed to be zero. The interactive law was checked and holds. Note that for the remaining chapters, the interactive law will be checked for all throughput and response time plots, and unless explicitly stated otherwise, it can be assumed to hold.

If we look at the throughput in Figure 2, we can see that for write-only workloads the system saturates at around 144 virtual clients because after that the throughput flattens. The CPU utilization plot of the memcached server, which can be seen in Figure 4, shows that the CPU is the bottleneck here because it almost reaches full utilization at around the saturation point.

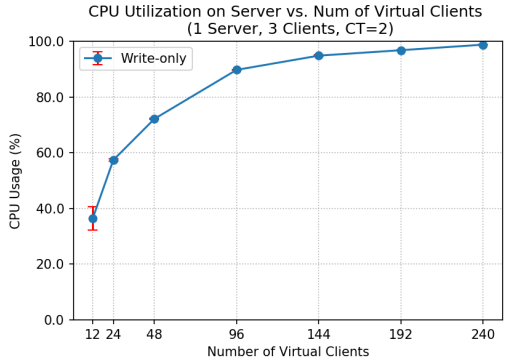

Figure 4: CPU utilization of a server VM during write-only workload.

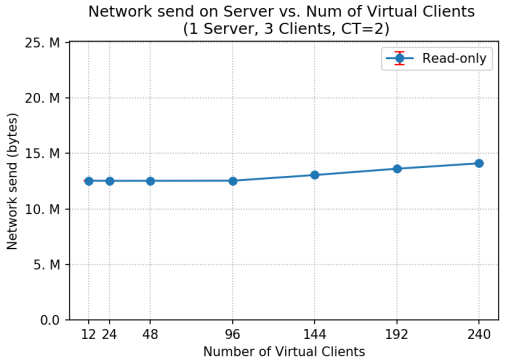

Figure 5: Outbound network activity of a server VM during read-only workload.

The throughput for read-only workloads stays at the same level during the whole experiment. This indicates that the network bandwidth might be a bottleneck here. In order to confirm this, we first need to check the maximum outbound network bandwidth for client and server VMs. This is done with *iperf* and the results can be seen in the following table:

	Maximum outbound network bandwidth
Server VM	12.6 MB/s
Client VM	25 MB/s

Now that we know the maximum outbound network bandwidth, we can check the actually used network bandwidth. Figure 5 shows that indeed the network activity of memcached server is at its limit and thus the bottleneck of read-only workloads. That the network send activity does not stay perfectly at 12.6 MBps I attribute to noise in the cluster. We can compute the maximal throughput based on the maximal bandwidth of the server to support our claim: A single response from the server is around 4kB because it contains a single value corresponding to a key. Thus if we divide 12.6 MBps by 4kB, we get around 3k requests per second which is exactly what we measured. Saturation is already reached for the lowest number of clients which is why we don't observe under-saturation.

Finally we look at the response times in Figure 3. We observe that for read-only workloads the response-time grows faster than for write-only workloads. This makes sense because of the lower throughput for the read-only workloads.

2.2 Two Servers

In this setup we have 2 memtier instances (both run on the same VM), each of which is connected to a single memcached instance. Each memcached instance is installed on a separate VM, i.e. we have two server VMs in total. The overview of the experiment parameters is given 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	{1,4,8,16,24,32,40}
Workload	Write-only and Read-only
Repetitions	3

The throughput as a function of virtual clients for read-only and write-only workloads is shown in Figure 6, whereas the response time as function of virtual clients for read-only and write-only workloads is shown in Figure 7.

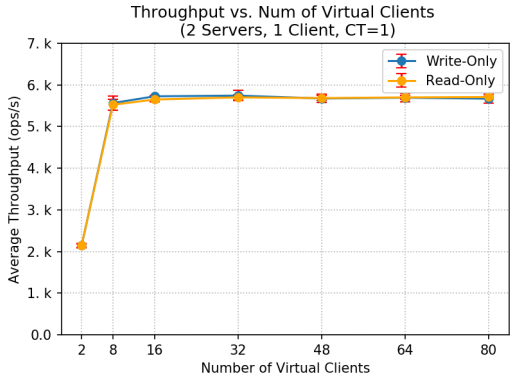

Figure 6: Average throughput.

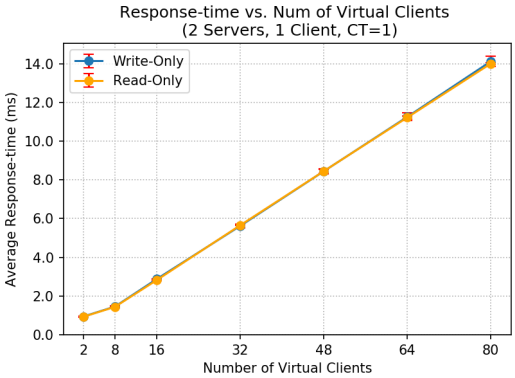

Figure 7: Average response time.

First of all the interactive law was checked and holds. For write-only workloads the system already saturates at 8 virtual clients because from this point on, the throughput flattens and the response time starts increasing significantly, as it can be seen in Figures 6 and 7. The bottleneck here is the outbound network bandwidth of the client VM. This can be seen in Figure 8 which shows that the maximum bandwidth of 25 MB/s (checked with *iperf*) for the client VM is reached. The network send activity not being perfectly at 25 MBps I attribute to noise on the cluster.

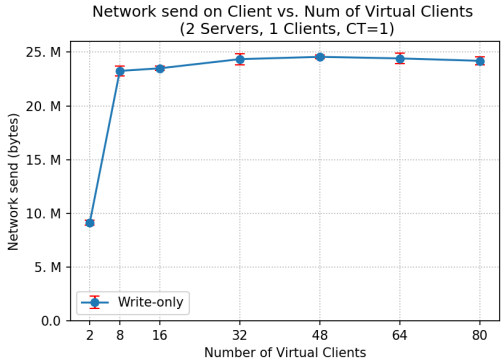

Figure 8: Outbound network activity of the client VM.

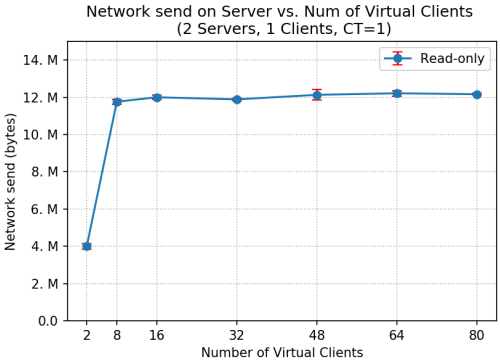

Figure 9: Outbound network activity of a server VM.

For read-only workloads the system also saturates at 8 virtual client for the same reasons, as can be seen in Figures 6 and 7. The bottleneck here is also the outbound network bandwidth, but now of the server VMs and not the client VM. This can be seen in Figure 9, which shows that the maximum bandwidth of 12.6 MB/s (checked with *iperf*) for the server VM is reached at 8 virtual clients, which is exactly the saturation point of the read-only workload.

Note that those results also make sense because for the read-only workloads the responses contain 4 kB values, which mainly affect the outbound bandwidth at the server, whereas the requests just contain the keys which require a few bytes. On the other hand, for the write-only workloads we have the opposite case, i.e. the requests contain the 4 kB values, which leads to an outbound bandwidth problem at the clients and the responses just contain a status code of a few bytes.

2.3 Summary

Maximum throughput of different VMs.

	Read-only workload	Write-only workload	Configuration giving maximum throughput
One memcached server	2'924 reqs/s	15'624 reqs/s	12 clients for read-only, 96 clients for write-only
One load generating VM	5'897 reqs/s	5'976 reqs/s	8 clients for read-only and write-only

The table above shows the maximum throughput for both experiments. Note that when determining the maximum throughput of the system, we take the response time and how it is affected by the increase in load into consideration. For the one server case and write-only workload, going from 96 to 144 clients increases the throughput by 5.6% while increasing the response time by 30%, which is why we chose the 96 client configuration. For the one server case and read-only workload, we choose the lowest number of clients because the throughput remains the same while the response time increases with more clients. For the two server case and write-only workload, going from 8 to 16 clients increases the throughput by 2.8%, while increasing the response time by 50% and similar numbers for the read-only workload.

For the one server experiment, we tested the limits of a single server. For write-only workloads the bottleneck is its CPU and for read-only workloads the bottleneck is its outbound network bandwidth, which is plausible because for the writes it has to process a lot of big requests, and for the reads it has to send back big responses. The reason that reads have a much smaller throughput than the writes is that the server only has a 12 MBps outbound network bandwidth, which is a much more limiting factor than the CPU for the writes.

For the two server experiment, we tested the limits of a single client. For both kind of workloads the bottleneck is the network bandwidth and the throughput is very similar, which I guess is just a coincidence based on the maximal outbound network bandwidth of the client and the server VM.

Comparing the read-only throughput of the experiments, we observe that doubling the number of servers also roughly doubles the read-only throughput on average. This is because in the one server case we did not provide enough server bandwidth resources. Comparing the write-only throughput of the experiments, we observe that the one server case has roughly $2-3$ times the throughput of the two server case. This is because in the two server case we did not provide enough client bandwidth resources.

The conclusion is that the server and client resources need to be carefully selected for the performance analysis of the middleware. If we want the observe the bottlenecks of the

middleware, we need to provide enough resources such that we are not limited for example by the network bandwidth or CPU of the server or client VMs.

3 Baseline with Middleware (90 pts)

In this section we put either one or two middlewares between the *memtier* clients and the *memcached* servers. The purpose of this is to study the performance effects of our middleware. The experimental setup is as follows: We let each experiment run for 80s and each experiment is repeated 3 times. The first and last 10 seconds of an experiment are considered to be unstable and are thus ignored. The standard deviation of a specific metric is computed over those repetitions and shown as error bars in the plots. From the outputs of the middleware we compute average queue length, a detailed breakdown of the response time and throughput of the middleware. In addition, the throughput and overall response time is computed from the outputs of the *memtier* instances. Different kinds of resource usage statistics are computed with *dstat*.

3.1 One Middleware

In this setup we connect three load generator machines to a single middleware and use one memcached server. The overview of the experiment parameters is given 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}
Workload	Write-only and Read-only
Number of middlewares	1
Worker threads per middleware	{8,16,32,64}
Repetitions	3

• **Read-only load:** The throughput and response time as a function of virtual clients for read-only workloads is shown in Figure 10 and in Figure 11.

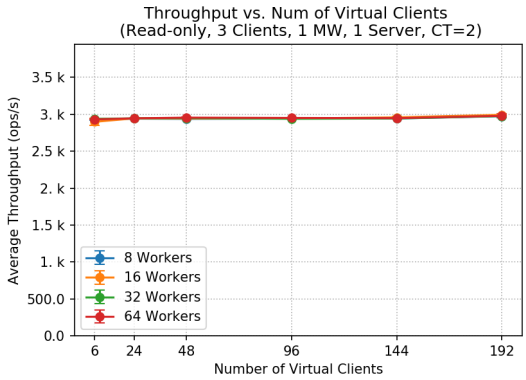

Figure 10: Throughput of middleware

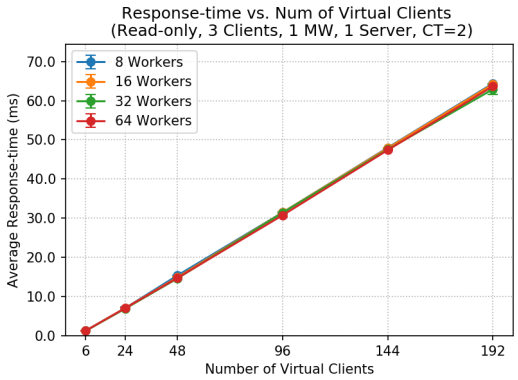

Figure 11: Response time of middleware

The response time of the middleware is computed as the average time a request needs from the first encounter with the net-thread to the point where a worker thread sends the response back to the client. For the interactive law to hold, we can consider twice the network latency between the client and the middleware as the think time Z.

In Figure 10 we can see that the saturation point for all worker settings is already reached at 6 clients. The bottleneck of this setup is the outbound network bandwidth at the server,

which can be seen in in Figure 12. Using *iperf*, we checked that the maximum outbound network bandwidth of a server VM is 12.6 MBps, which is reached at 6 clients and thus we don't see any under-saturation. The bottleneck also makes sense because of the large values the server has to send back for read-only workloads. Different number of workers reach the same maximum throughput, which supports the claim that bandwidth is the bottleneck because it is independent of the number of workers.

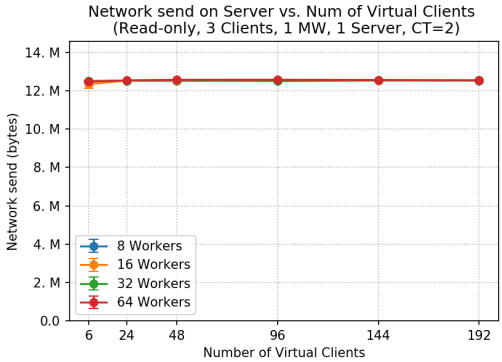

Figure 12: Outbound network activity of the server VM.

In the following, we want to support the claim by looking at the response time in more detail. Figure 13 shows that the response time stays the same for different number of workers, but the queue time shifts to the memcached RTT as we increase the number of workers. This is because we have a lower service rate of workers for small number of workers and thus requests wait longer in the middleware queue. If there are a lot of workers, requests are quickly dequeued, but they then congest at the server outbound queue and have to wait there instead because it is the bottleneck of the setup.

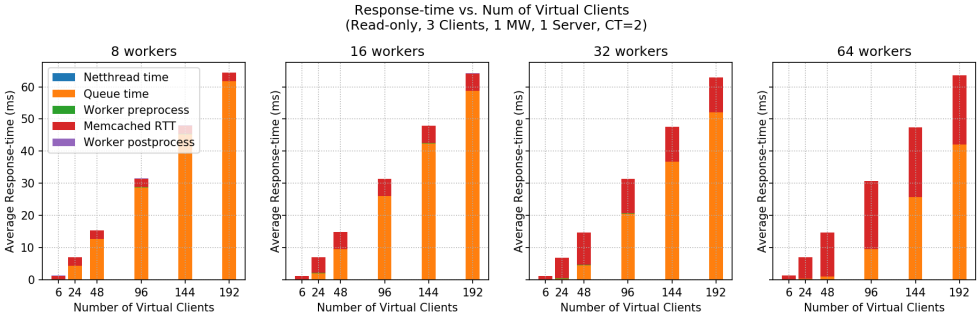

Figure 13: Response time breakdown of the read-only workloads.

• **Write-only load:** The throughput and response time measured on the middleware as a function of virtual clients for write-only workloads is shown in Figure 14 and in Figure 15.

For **8 worker threads** saturation is reached at 24 clients while for **16 worker threads** saturation is reached at 48 clients as can be seen in Figure 14 and 15. In both cases the bottleneck are the worker threads which cannot dequeue requests faster than they are enqueued by the net-thread. This can be verified by multiple metrics. First we can look at the average queue length for different number of clients. Figure 16 shows that for 8 and 16 worker threads the queue length starts increasing more significantly at their saturation points. This happens because the arrival rate of requests into the queue is bigger than the overall service rate of the workers and thus requests start accumulating in the queue.

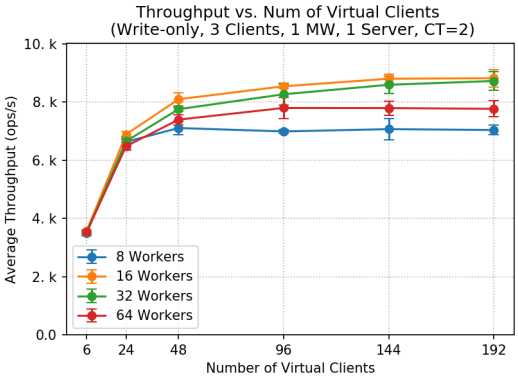

Figure 14: Throughput of middleware

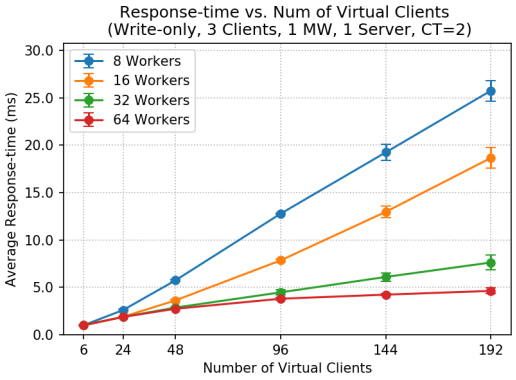

Figure 15: Response time of middleware

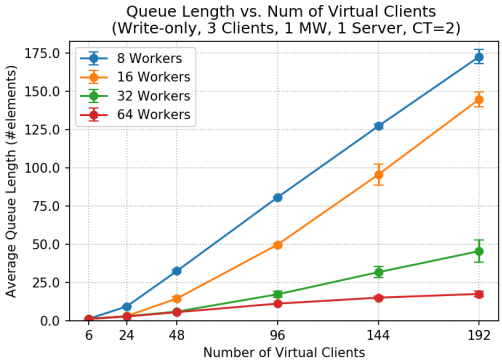

Figure 16: Average queue length for write-only workloads.

We can also look at the response time in the middleware in more detail. Figure 17 shows that for 8 and 16 worker threads, requests spend most of their time in the queue, and once dequeued they are processed quickly, which supports our claim.

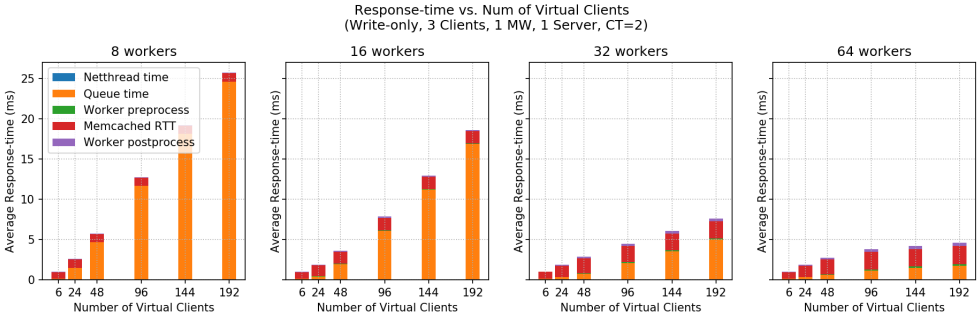

Figure 17: Response time breakdown of the write-only workloads.

If we look at **32 and 64 worker threads**, first of all in Figure 14 we see that in both cases saturation is reached at around 48 clients. The response time in the middleware for 32 and 64 worker threads is decreased in contrast to 8 and 16 worker threads as can be seen in Figure 15. This is because we increased the resources at the bottleneck, which are the worker threads. The queue time is decreased because the overall service rate of the workers is increased (Figure 17). Even though the response time at the middleware is decreased significantly, the overall response time measured at the clients (Figure 18) did not, i.e. there is a new bottleneck now, which is

the net-thread. Requests congest in the inbound network queue of the net-thread. This waiting time makes up roughly the difference between the response time measured on the middleware and on the client and is plotted separately in Figure 19. We see that for 32 and 64 workers the time in front of the net-thread increases significantly at their saturation points.

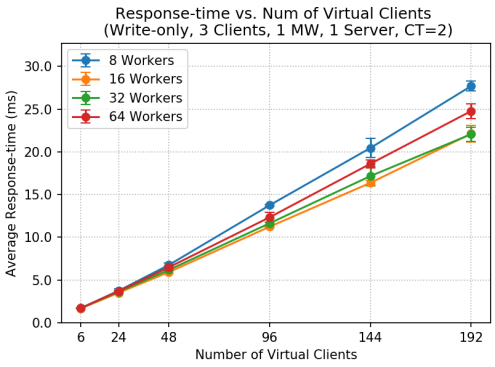

Figure 18: Response time of the write-only workloads measured on client.

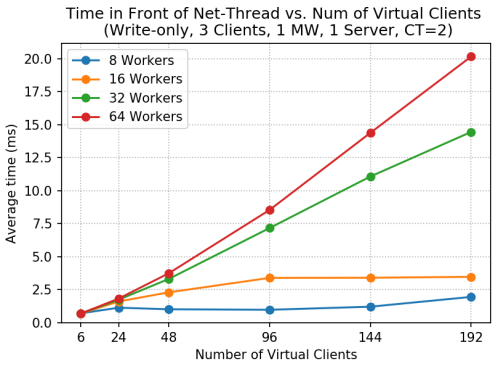

Figure 19: Average time spend in inbound network queue of net-thread.

The increase in inbound net-thread queue time for higher number of workers can be explained as follows: The more worker threads we have, the higher the overhead on the CPU i.e. more threads compete for the same resource and thus there are more context switches (verified with *dstat*), which affects the performance of the net-thread because it gets less CPU time and can process fewer requests. Note that workers are allocated a priori in our system.

So even tough going from 16 to 32 worker threads removes the worker bottleneck and decreases queue time (Figure 17), this time is compensated by the increased waiting time in front of the net-thread, and thus the throughput does not improve (Figure 14). Going from 32 to 64 worker threads has almost no effect on the response time at the middleware because already 32 worker threads were enough to handle the load. So we just have more net-thread waiting time because of the higher overhead on the CPU but without decreased queue time. This leads to a worse throughput than in the 32 worker thread case (Figure 14).

3.2 Two Middlewares

In this experiment we connect one load generator machine (two instances of memtier with CT=1) to two middlewares and use 1 memcached server. The overview of the experiment parameters is given in the following table:

Number of servers	1
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	{1,4,8,16,24,32,40}
Workload	Write-only and Read-only
Number of middlewares	2
Worker threads per middleware	{8,16,32,48}
Repetitions	3

• **Read-only load:** The throughput and response time as a function of virtual clients for read-only workloads is shown in Figure 20 and in Figure 21.

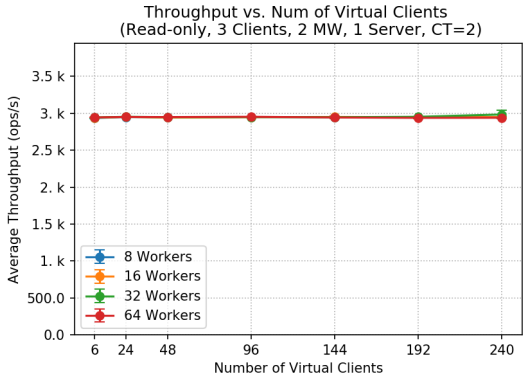

Figure 20: Throughput of middlewares

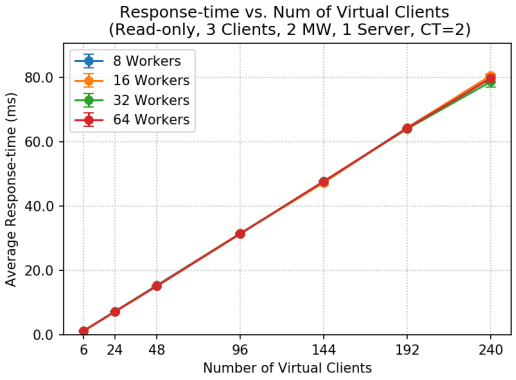

Figure 21: Response time of middlewares

In Figure 20 we can see that the saturation point for all worker settings is already reached at 6 clients. The bottleneck of this setup is the same as for the one middleware case, namely the outbound network bandwidth at the server, which can be seen in in Figure 22. This also makes sense because going from one middleware to two middlewares does not improve the bottleneck at the server. The same reasoning done for the bottleneck analysis in the one middleware case applies here.

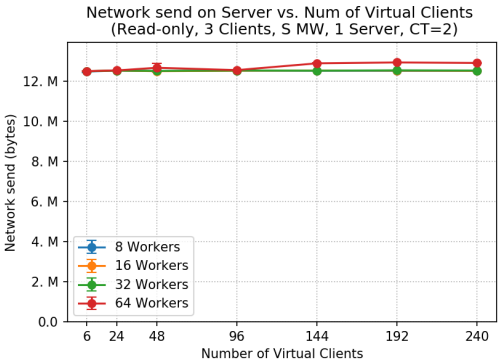

Figure 22: Outbound network activity of the server VM.

We can also look at the response time in more detail (Figure 23) and see that increasing the number of workers shifts the time spent in the queue to the time spent in the outbound network queue of the server because of the server bandwidth bottleneck.

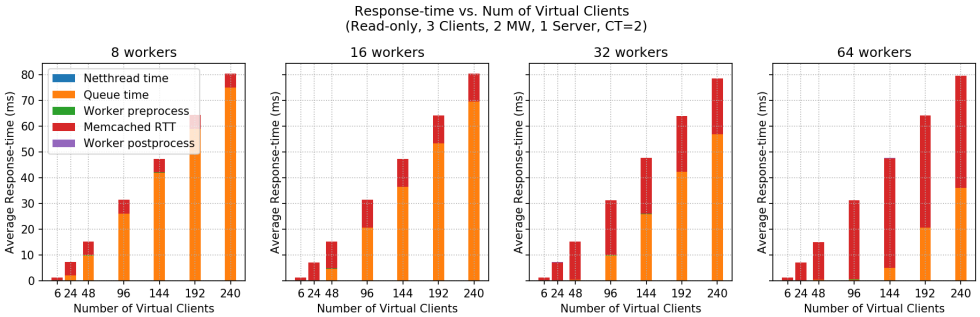

Figure 23: Response time breakdown of the read-only workloads.

- **Write-only load:** The throughput and response time as a function of virtual clients for

write-only workloads is shown in Figure 24 and in Figure 25.

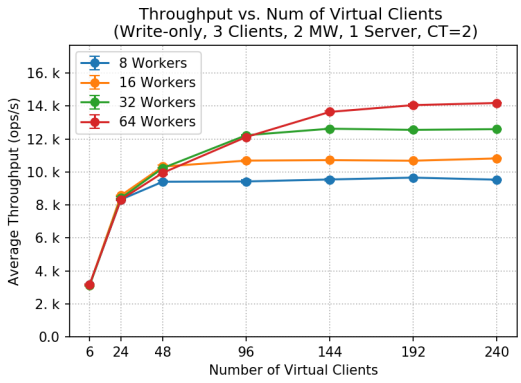

Figure 24: Throughput of middlewares

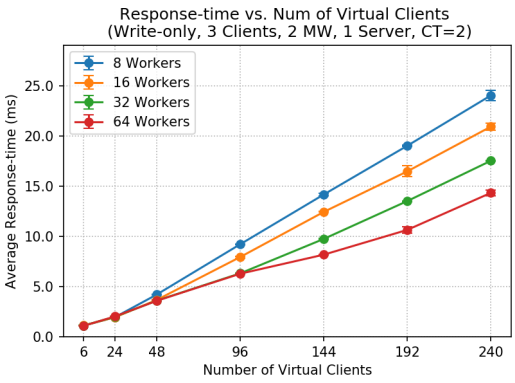

Figure 25: Response time of middlewares

Looking at Figure 24, we see that saturation for 8 workers is reached at 24 clients, for 16 workers at 48 clients, for 32 clients at 96 clients and for 64 workers at 144 clients because at those points throughput starts to flatten. The saturation points are confirmed later in the analysis. In the one middleware case the net-thread is the bottleneck, but having two middlewares removes the net-thread bottleneck because we now have two net-threads sharing the same load. This can be confirmed by looking at the response time graph computed from the memtier outputs, which looks exactly the same as Figure 25 plus 1-2 ms for the network latency between the client and the middleware. Thus no time is lost in front of the net-thread anymore. The bottleneck of this setup are the worker threads. To see this we can first look at the average queue length (Figure 26). The queue length starts increasing significantly at the mentioned saturation points. We also checked the CPU utilization of the server VM (seen in Figure 27) to verify that it does not reach full utilization which means that it is not the bottleneck of this setup.

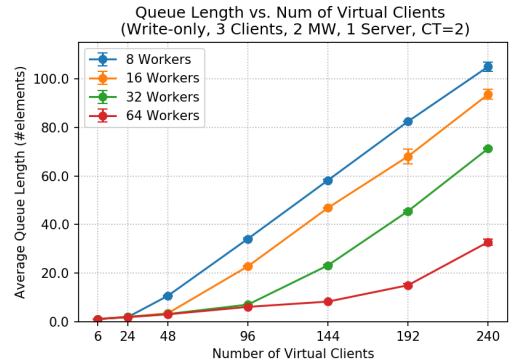

Figure 26: Average queue length for write-only workloads.

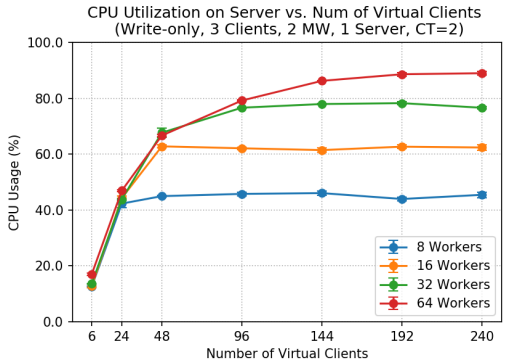

Figure 27: CPU utilization of server VM for write-only workloads.

We can support our claim by additionally looking at the response time of the middleware in more detail (Figure 28). We can see that the queue time starts increasing at the saturation points because the arrival rate of requests into the queue is higher then the service rate of the workers together. Note that if we fix the workers and look at the memcached RTT during saturation while increasing the clients, we see that it stays constant. That is because the workers are at full utilization and the arrival rate of requests into the memcached queue does not increase anymore, which further supports our claim that the workers are the bottleneck.

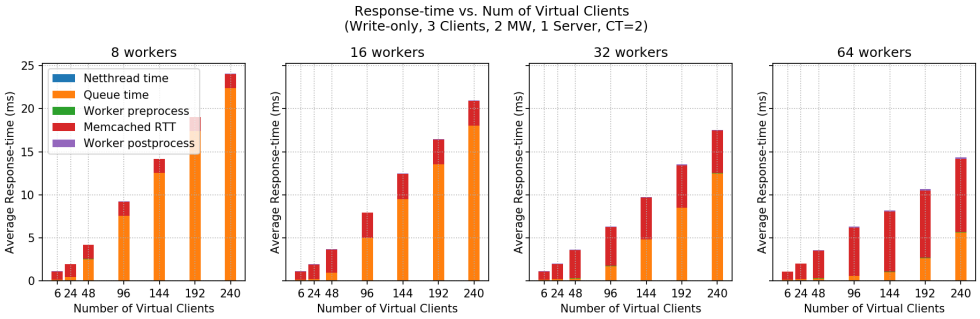

Figure 28: Response time breakdown of the write-only workloads.

3.3 Summary

Maximum throughput for one middleware.

	Throughput	Response time	Average queue time	Miss rate
Reads: Measured on middleware	2'929 ops/s	1.26ms	0.1ms	0.0%
Reads: Measured on clients	2'925 ops/s	2.06ms	n/a	0.0%
Writes: Measured on middleware	8'105 ops/s	3.63ms	1.87ms	n/a
Writes: Measured on clients	8'097 ops/s	5.93ms	n/a	n/a

Maximum throughput for two middlewares.

	Throughput	Response time	Average queue time	Miss rate
Reads: Measured on middleware	2'935 ops/s	1.18ms	0.08ms	0.0 %
Reads: Measured on clients	2'923 ops/s	2.06ms	n/a	0.0 %
Writes: Measured on middleware	13'645 ops/s	8.18ms	0.97 ms	n/a
Writes: Measured on clients	13'603 ops/s	10.6ms	n/a	n/a

First note that again the maximum throughput is chosen by also taking the latency into consideration. For the read-only workload, we choose the lowest number of clients (and workers) in both cases because the throughput remains the same while the response time increases with more clients. For the write-only workload and one middleware configuration, we choose 48 clients and 16 workers because going even further to 96 clients would only increase the throughput by 5.5% while increasing the response time more significantly by 89.5%. For the write-only workload and two middlewares configuration, we choose 144 clients and 64 workers because going to 192 clients would only increase the throughput by 3% but at the same time increase response time by 34.4%.

The response time difference between the client and the middleware is due to the time a request spends between a client and the middleware, which is not part of the response time measured at the middleware.

If we look at the read-workloads, we see that increasing the number of middlewares does not improve the throughput. This is because the bottleneck is the outbound network bandwidth at the server, which puts a hard limit on the achievable throughput.

If we look at the write-workloads, we see that going from one to two middlewares significantly increases the throughput. This is because the bottleneck in the one middleware case is the net-thread and having two net-threads allows them to share the same load, which reduces their utilization.

4 Throughput for Writes (90 pts)

In this section we connect three load generating VMs to two middlewares and three memchached servers. We only run write-only workloads and we want to figure out the effects of replicating data to the servers. The experimental setup is the same as in the previous section.

4.1 Full System

The overview of the experiment parameters is given 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	{1,4,8,16,24,32,48}
Workload	Write-only
Number of middlewares	2
Worker threads per middleware	{8,16,32,64}
Repetitions	3

The throughput and response time as a function of virtual clients for write-only workloads is shown in Figure 29 and in Figure 30.

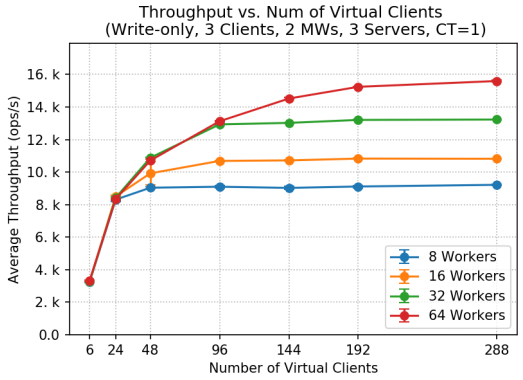

Figure 29: Throughput of middlewares

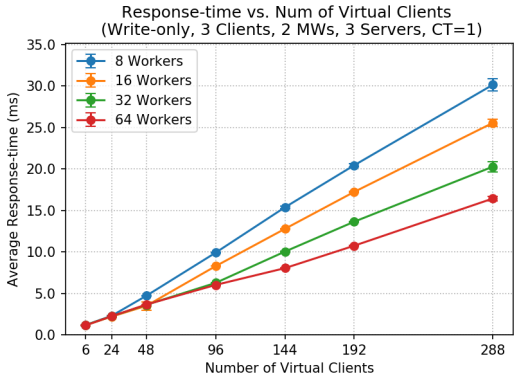

Figure 30: Response time of middlewares

In Figure 29 and Figure 30 we can see that saturation for 8 workers is reached at 24 clients, for 16 workers at 48 clients, for 32 workers at 96 clients and for 64 workers at 192 clients because at those points throughput starts to flatten and we also see the response time increasing more significantly.

First of all, we can compare the response time measured at the middleware with Section 3.2 (write-only workload), where we have one instead of three servers. Comparing Figure 25 and Figure 30 we see that the response time does slightly increase if we take three instead of one server, which makes sense because we have to additionally replicate the data in this experiment. A worker thread sends a Set request from the client to each server, one after the other, and then waits until all Set requests have been executed on all servers, which leads to a longer response time on the middleware.

The bottleneck for all four worker configurations are the worker threads themselves. To see this we can first look at the average queue length (Figure 31). We see that the queue length starts increasing significantly at the mentioned saturation points. We have queue congestion because the arrival rate into the queue is bigger than the combined service rate of the worker threads.

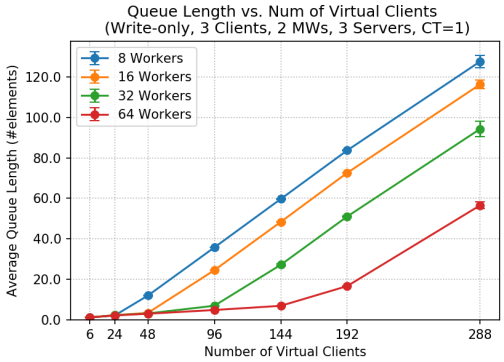

Figure 31: Average queue length.

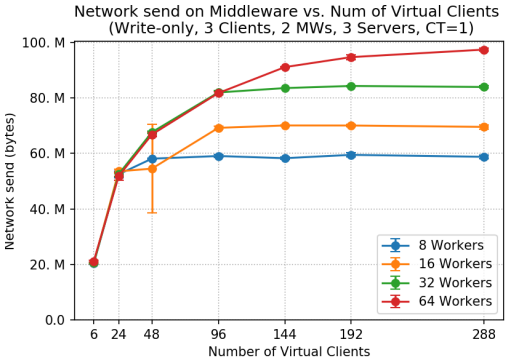

Figure 32: Outbound network activity of the middlewares.

We also checked the outbound network bandwidth at the middlewares because the middlewares have to send three times more data than in the one server case due to replication. Figure 32 shows the outbound network activity of a middleware VM. We found out with *iperf* that the maximum outbound bandwidth of a middleware VM is 100 MBps, which is almost reached for 64 workers. But since it is not reached at the saturation point, it is not the reason for saturation and thus not the bottleneck for the 64 workers case.

We can support our bottleneck analysis by additionally looking at the response time of the middleware in more detail (Figure 33). We can also see that the queue time starts increasing at the saturation points. The increase in waiting time for memcached (denoted as "Memcached RTT") for higher number of workers also makes sense because servers need to serve more workers simultaneously.

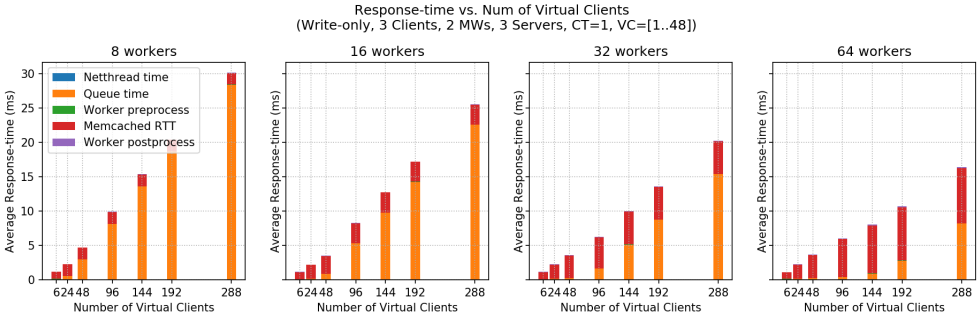

Figure 33: Response time breakdown.

4.2 Summary

Maximum throughput for the full system

	WT=8	WT=16	WT=32	WT=64
Throughput (Middleware)	8'304 ops/s	8'480 ops/s	12'930 ops/s	14'519 ops/s
Throughput (Derived from MW rt)	7'565 ops/s	7'795 ops/s	13'440 ops/s	16'134 ops/s
Throughput (Client)	8'284 ops/s	8'498 ops/s	12'905 ops/s	14'457 ops/s
Average time in queue	0.54ms	0.14ms	1.59ms	0.8ms
Average length of queue	2.16	2.32	6.97	6.93
Average time waiting for memcached	1.67ms	1.98ms	4.51ms	6.99ms

The configurations that give maximal throughput are chosen such that a good trade-off between

the throughput and latency is achieved. For 8 workers, 24 clients were chosen because going from 24 to 48 clients would increase response time by 84%, while only increasing throughput by 8.8%. For 16 workers, also 24 clients were chosen because going from 24 clients to 48 clients would increase response time by 55%, while only increasing throughput by 16%. For 32 workers, 96 clients were chosen because going from 96 to 144 clients would increase response time by 50%, while only increasing throughput by less than 1%. For 64 workers, 144 clients were chosen because going from 144 to 288 clients would increase response time by 28%, while only increasing throughput by 5%.

The throughput can be derived from the MW response time in two ways using the interactive law: Either we consider the round-trip time between the MWs and the clients as the waiting time Z or we adjust the number of requests N by subtracting the average number of requests that are between the clients and the middlewares. We choose the first approach and average the ping times between the clients and the middlewares to get an average round-trip time of 0.8915ms. For 64 workers the throughput derived from the middleware response time is significantly overestimated. This is because the difference between the response time measured at the client and at the middleware is larger than the network latency between them and thus the think time Z is chosen too small. This difference includes the time a request waits in front of the net-thread. The more worker threads there are, the more threads compete for the same resource, which slows down the net-thread.

The highest throughput is reached with 64 worker threads. This is because the workers are the bottleneck in this setup and in general increasing the resource at a bottleneck increases the throughput of a system. Thus we achieve highest throughput with the highest number of workers. With 64 worker threads and high enough load, the maximum outbound network bandwidth of the middleware VMs is almost reached (Figure 32) which means that going beyond 64 workers would not significantly increase the throughput anymore in this setup. In order to reduce the network send activity of the middleware VMs such that one can increase the throughput even more, one would have to either increase the number of middlewares or decrease the replication factor of the key-value pairs.

5 Gets and Multi-gets (90 pts)

In this section we connect three load generating VMs to two middlewares and three memchached servers. We want to understand the effects of increasing the multi-get size on the performance of system. In addition, we analyze if sharding the multigets improves the performance.

5.1 Sharded Case

The overview of the experiment parameters is given 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	ratio=1:<Multi-Get size>
Multi-Get behavior	Sharded
Multi-Get size	{1,3,6,9}
Number of middlewares	2
Worker threads per middleware	64
Repetitions	3

If sharding is activated, the keys inside the multiget requests are evenly split by the workers and distributed among the servers. The average response time measured on the client as a

function of the multiget size and the percentiles for the sharded case are shown in Figures 34 and 35.

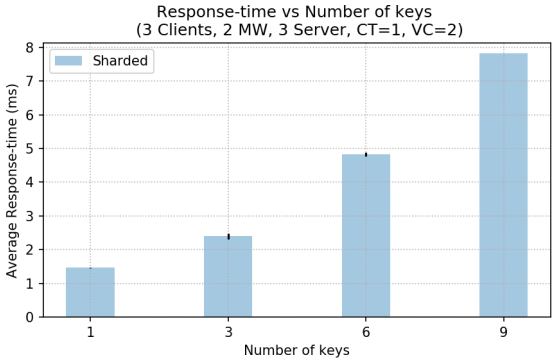

Figure 34: Response time measured on clients.

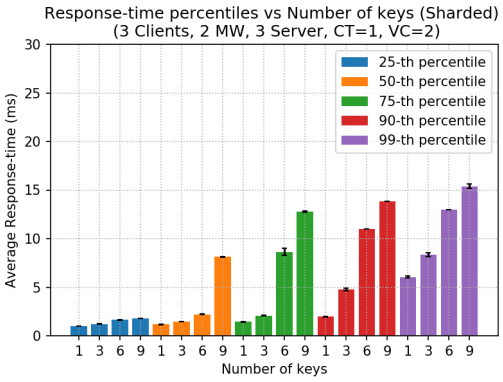

Figure 35: Response time percentiles measured on client.

First of all note that since we are in a closed system and we only have 12 clients, at least 52 workers will be idle at any moment in time. This is not optimal because threads use up memory and we establish more connections to servers than necessary (this depends on how the middleware has been implemented, but in our case the threads are started a priori).

For 3, 6 and 9 keys the bottleneck is the outbound network bandwidth at the server VMs, as can be seen in Figure 37. Note that the maximal outbound network bandwidth of a server VM is 12 MBps. For 1 key, the system is under-saturated because no component reaches full utilization. We checked the CPU usage of the VMs, the network send activity of the VMs, the response time breakdown in the middleware, the average queue length and we also have enough workers because at least 52 workers are always idle.[7] Thus for the 1 key case, the number of clients could be further increased to reach better performance regarding throughput.

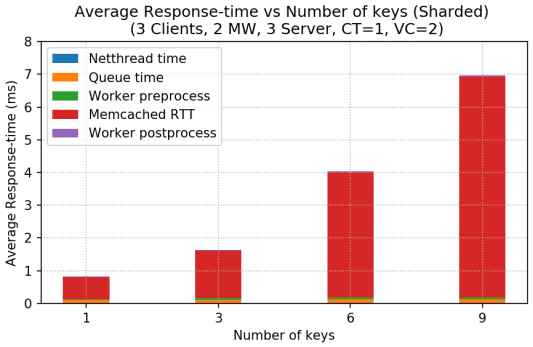

Figure 36: Response time breakdown at middleware.

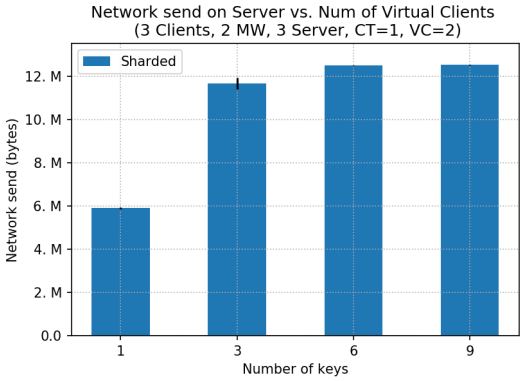

Figure 37: Network send activity on a server VM.

If we look at the response time breakdown at the middleware in Figure 36, we can see that the memcached RTT makes up most of the time. The memcached RTT mainly consists of the network latency between the middleware and the server, the waiting time at the server because of the bandwidth bottleneck and the time it takes the server to write the response into

[7] Those plots are not included in the report to keep it concise, but they can be found in the repository.

the network. We see that the average response time on the middleware (Figure 36) is lower than on the client (Figure 34), which is because of the network latency between the client and middleware. We also see that in Figures 34 and 36, the response time increases with the number of keys. This is because the server has to write more data into the network and also has to process more data. Note that the memcached RTT is what significantly increases the overall response time with increasing key size.

If we look at the percentile plot (Figure 35), we see that individual percentiles increase with increasing number of keys for the same reason as the average response time increases with the number of keys. The percentile plot will be further explained in comparison with the plot of the non-sharded case and in subsection 5.3.

5.2 Non-sharded Case

The overview of the experiment parameters is given 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	ratio=1:<Multi-Get size>
Multi-Get behavior	Non-Sharded
Multi-Get size	{1,3,6,9}
Number of middlewares	2
Worker threads per middleware	64
Repetitions	3

The average response time measured on the client as a function of the multiget size and the percentiles for the non-sharded case is shown in Figure 38 and 39.

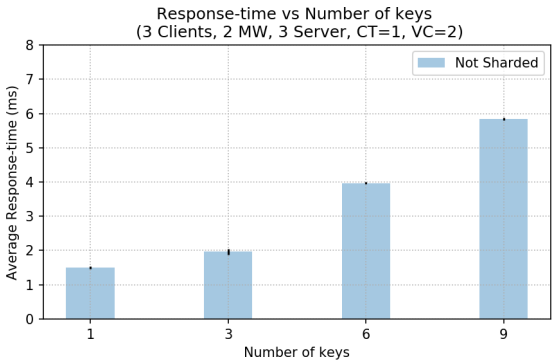

Figure 38: Response time measured on clients.

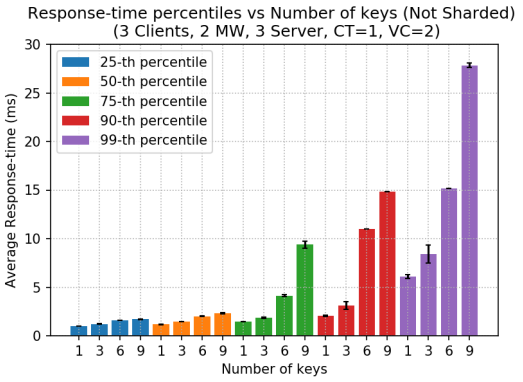

Figure 39: Response time percentiles measured on client.

For the non-sharded case we have the same bottleneck as in the sharded case, namely the outbound network bandwidth of the server VMs (see Figure 41). The same reasoning done in the sharded subsection applies here. For 1 key, the system is also under-saturated because no component reaches full utilization. The memcached RTT makes up most of the response time measured at the middleware (Figure 40) and the response times increase with the key size for the same reasons. In this subsection I mainly focus on comparing the plots with their sharded counterpart.

First of all, looking at the requests with one key (Figures 34 and 38), we see very similar response times for the sharded and non-sharded case because for one key the sharding behaves

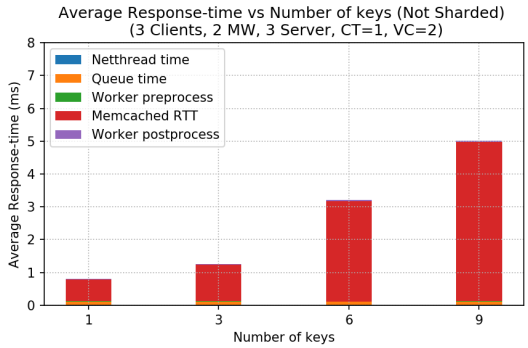

Figure 40: Response time breakdown at middleware.

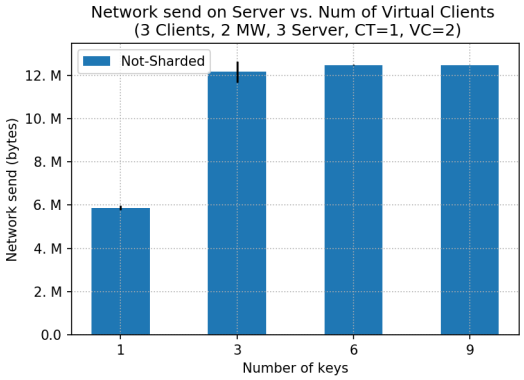

Figure 41: Network send activity on a server VM.

the same as for non-sharding. For 3, 6 and 9 keys, we can observe that the average response times are higher in the sharded case than in the non-sharded case (Figures 34 and 38). This is because in the sharded case a worker has to wait for a response from all servers, i.e. the response time is bound by the slowest server or rather the longest network latency to a server. In this experiment, the network latency between the middlewares and servers does not vary a lot (verified with ping logs), but I guess that if for example the latency to one server was a lot bigger then the other latencies, the effect would be even more significant.

Even though in Figures 36 and 40 we can barely see any latencies other than the memcached RTT, looking at the numbers we observe that the worker preprocessing time is 5x bigger in the sharded case than in the non-sharded case. This is because a multiget request is divided into multiple requests and has to be copied into buffers before sending it to the servers while for the non-sharded case we can just take the buffer and send it to the server. But this time is negligible compared to the memcached RTT.

Looking at the percentiles plots (Figures 35 and 39), we have a larger deviation between percentiles in the non-sharded case than in the sharded case. This is because the response time in the non-sharded case depends on the server, which is chosen based on round-robin, while in the sharded case we are always limited by the slowest server, which leads to less variation in response time.

5.3 Histogram

In this subsection we look at the case with 6 keys inside the multigets, and present four histograms representing the sharded and non-sharded response time distribution, both as measured on the client, and inside the middleware. The bucket size is chosen to be 0.1ms. The histograms can be seen in Figure 42.

Generally we can say that the majority of requests are processed within 3ms for both the sharded and the non-sharded case. Also the histogram plots conform with the percentile plots. For the sharded-case we have 50% of request being processed within 2ms, 75% within 9ms, 90% within 11ms and 99% within 13ms. For the non-sharded case we have 50% of request being processed within 2ms, 75% within 4ms, 90% within 15ms and 99% within 15ms. In the sharded case, the 75th and 90th percentiles are significantly larger than in the non-sharded case, which is because in the sharded case we have a hill around 11ms, which will be explained later.

Comparing the client plots with the middleware plots, we see that the peak of the middleware plots is closer to 0ms than on the client plot. This is because of the network latency between the clients and middlewares which is not included in the middleware response time. We can also

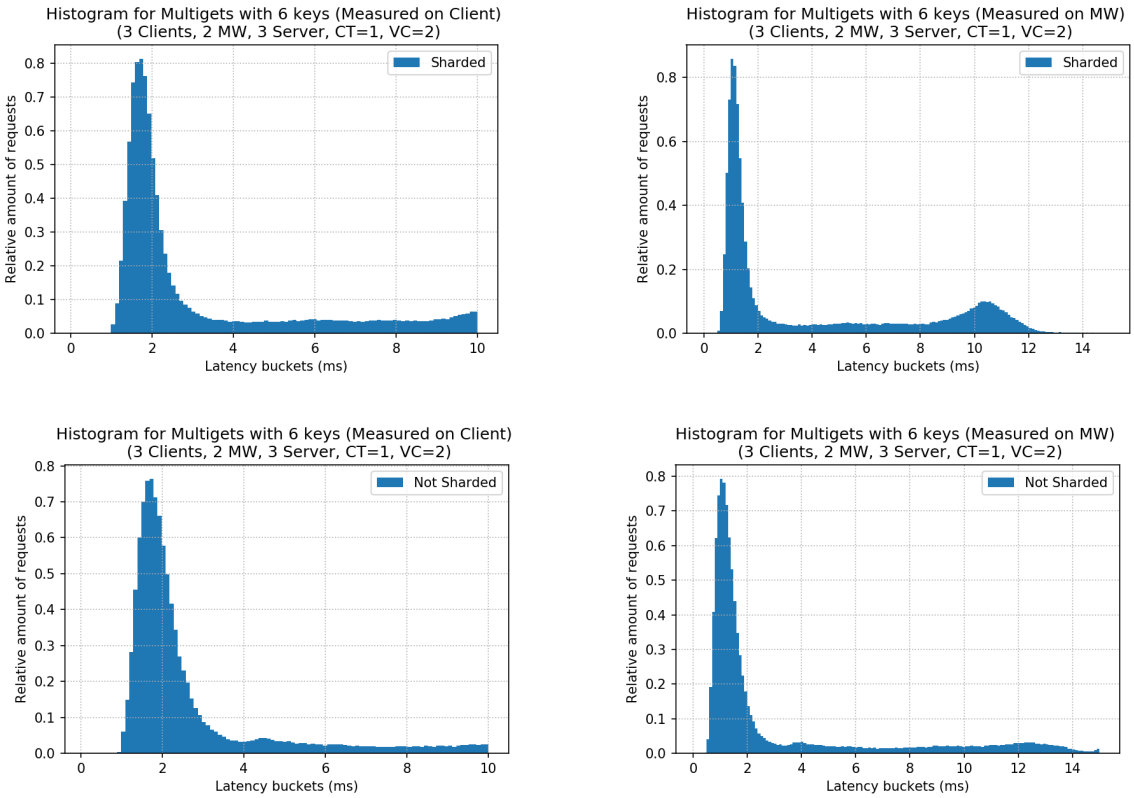

Figure 42: Histograms of the response time distribution of sharded and non-sharded multigets measured on the client (left column) and on the middleware (right column).

see that the distributions on the middleware are a little more narrow than on the clients. This is because the response time measured on clients captures a longer route of requests than the response time measured on the middlewares and thus we have more variability in the response time. Note that we had to cut the distributions measured on clients at 10ms because memtier did increase the bucket size from 0.1ms to 1ms after 10ms, which is too coarse grained. But essentially the continuation of the distributions can be seen in the middleware plots.

Comparing the sharded plots with the non-sharded plots, we can see that in the non-sharded case the distribution around the peak is more spread out than in the sharded case. This is because in the non-sharded case the response time of a request depends on the chosen server by the round-robin load balancer, which is not the same for every request, whereas in the sharded case, the response time is always determined by the slowest server and thus the response time of all requests is bound by this server, which leads to a more narrow distribution around the peak. Since the network latency between a middleware and a server is around 1ms for all servers, we only have one peak in the non-sharded case but I would expect multiple peaks if we had larger differences in latencies.

What remains to be mention is the small hill around 10ms in the sharded case. I assume this is because of the bottleneck at the server which leads to congestion and thus to some requests having to wait longer than others to get processed. The hill almost disappeared in the non-sharded case, which might be because the probability for this delay at the server is smaller due to a request only involving one server and not all servers. This would support the hypothesis.

5.4 Summary

To conclude, non-sharding might be preferred over sharding because of a better average response time. This is because in the non-sharded case not all multigets are limited by the slowest server or largest latency to a server like in the sharded case. Also, sharding has an overhead of dividing multigets, combining responses and sending respectively receiving data from three instead of one server. The larger the service time differences between the servers and the network latency differences to the servers are, the bigger the average response time difference between the sharded and non-sharded case would be for the number of keys analyzed in this section.

6 2K Analysis (90 pts)

In this chapter we want to figure out the effect of different parameters on the response time and the throughput with a 2k analysis.
More specifically, we conduct a $2^k * r$ analysis with three factors ($k = 3$) and with three repetitions ($r = 3$). The analysis is done for two different workloads (read-only and write-only) and for two different response variables (throughput and response time), i.e. in total we conduct four $2^k * r$ analyses.
The three factors we consider are the number of memcached servers, the number of middlewares and the number of worker threads. Each factor can assume two levels (-1 and 1), which represent a specific configuration as given in the following table:

	Number of servers (**A**)	Number of middlewares (**B**)	Number of workers (**C**)
-1	1	1	8
1	3	2	32

To get the input data of the analysis we first need to run the experiments. We let each experiment run for 90s and each experiment is repeated 3 times. The first and last 10 seconds of an experiment are part of the warm-up and cool-down phase and are thus ignored. The overview of the experiment parameters is shown in the following table:

Number of servers	1 and 3
Number of client machines	3
Instances of memtier per machine	1 (1 middleware) or 2 (2 middlewares)
Threads per memtier instance	2 (1 middleware) or 1 (2 middlewares)
Virtual clients per thread	32
Workload	Write-only and Read-only
Number of middlewares	1 and 2
Worker threads per middleware	8 and 32
Repetitions	3

The multiplicative model was chosen for all four analyses. It assumes that the effect of the factors, their interactions and the errors are multiplicative. The multiplicative model does explain our data very well because the unexplained variation of effects is less than 0.65% in all analyses. All we have to do in the multiplicative model in contrast to the additive one, is to take the log of the measurements. Then the calculations remain the same.

Before doing the interpretation of the results of the analyses, we want to explain what has been done in the tables which can be found at the end of this section. Note that all computations are done as explained in chapter 18 of the book "The Art of Computer Systems Performance

Analysis" by Raj Jain[8]. First of all, the input data, which is the measured response for different configurations and repetitions, is colored in yellow. Note that we take the log because of the multiplicative model. The effect variables q can be computed using the sign table method. The signs of the sign table are colored in red, while the effect variables are colored in green. The importance of a factor j is computed as the proportion of total variance in the response that is explained by that factor. It can be computed as SSj/SST, where SST (the Sum of Squares Total) is the total variance of the response y and SSj (the Sum of Squares due to factor j) is the portion of SST that is explained by factor j. The difference between the estimated and measured response values are called the experimental errors and are colored in purple. The Sum of Squared Errors (SSE) is used to estimate the variance of the errors and to compute the confidence intervals for the effects for which the t-value at 16 ($= 2^k(r-1)$) degrees of freedom and 90% confidence was chosen. If 0 is included in the confidence interval, the effect can be considered insignificant.

• **Write-only workload:** The number of middlewares explains 73.5% of the variation of effects on the throughput, which is by far the strongest effect. Then we have the number of servers with 14.7% of variation of effects on the throughput but they have a negative effect. The number of workers explain 7.9% of the variation of effects on the throughput and have a positive effect.
Those numbers make sense: In section 3 we saw that for the write-only workload going from 1 to 2 middlewares almost doubled the maximum achievable throughput because the net-thread bottleneck was relieved. Increasing the workers also increased the throughput because more requests can be processed in parallel, but not as significant as going from 1 to 2 middlewares. Increasing the number of servers has a negative effect on the throughput because of the replication that adds overhead to the system.
Almost all interactions-effects are either statistically insignificant or can be considered negligible, with the exception of the interaction-effect of the middlewares and the workers, which has 2.5% of variation of effects. This is because increasing the number of middlewares also increases the number of workers.
The percentages of variation of throughput and response time are basically the same with small deviations, which is what we expect because of the interactive response time law. But for the response time, the sign of significant effects are opposite to the throughput because they relate inverse proportionally to each other.

• **Read-only workload:** For the read-only workload the variation of effects can be explained entirely by the number of servers. All other effects are statistically insignificant or negligible.
This is because of the outbound network bandwidth bottleneck at the servers. If we fix the number of servers, this bottleneck puts a hard limit on the maximum achievable throughput, which we can also compute as (maximum bandwidth per server * #servers)/ size of a Get response. Thus changing number of workers (8 or 32) or changing the number of middlewares (1 or 2) does not change anything regarding throughput under this bottleneck. But going from 1 to 3 servers increases throughput from 3k ops/s to almost 9k ops/s because we have three times more outbound network bandwidth at the servers in total.
The response time analysis also conforms to the throughput analysis and relates in the same way to it as explained for the write-only workload.

[8]Raj Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, April 1991, ISBN: 978-0471503361.

Write-only throughput

I	A	B	C	AB	AC	BC	ABC	rep1	rep2	rep3	mean	err1	err2	err3
1	1	1	1	1	1	1	1	4.193695362	4.19333405	4.188750891	4.191926768	0.001768594092	0.001407282448	-0.003175876541
1	1	1	-1	1	-1	-1	-1	4.06539252	4.068875221	4.056709058	4.063658933	0.001733586667	0.005216288196	-0.006949874863
1	1	-1	1	-1	1	-1	-1	3.895504688	3.895728437	3.895812217	3.89568178	-0.0001770927671	0.00004665616198	0.0001304366051
1	1	-1	-1	-1	-1	1	1	3.882794769	3.88654067	3.877089954	3.882141797	0.0006529712143	0.004398872338	-0.005051843552
1	-1	1	1	-1	-1	1	-1	4.249095879	4.249333381	4.249366298	4.249265186	-0.0001693073723	0.00006819521679	0.0001011121555
1	-1	1	-1	-1	1	-1	1	4.15842783	4.150473144	4.159125449	4.156008808	0.002419022725	-0.005535663907	0.003116641182
1	-1	-1	1	1	-1	-1	1	4.018411952	4.040001425	4.03831995	4.032244442	-0.01383249052	0.001681475642	0.006075507437
1	-1	-1	-1	1	1	1	-1	3.984262813	3.975020561	3.992338406	3.983873927	0.0003888860207	-0.008853365355	0.008464479334
32.45480164	-0.3879830837	0.8669177481	0.2834347113	0.08860649842	0.0001809239807	0.1596137147	0.06984198829	**Total**						
4.056850205	-0.04849788546	0.1083647185	0.03542933891	0.0110758123	0.00002261549759	0.01995171434	0.008730248536	**Effect variables q (=Total/8)**						

	SSj	Percentage (SSj/SST)	Effects	Confidence low	Confidence high
SSE	0.0005665683345	0.1478139259			
SSA	0.05644907746	14.72719043	-0.04849788546	-0.05061871018	-0.04637706074
SSB	0.2818298932	73.52755249	0.1083647185	0.1062438938	0.1104855432
SSC	0.03012571334	7.859599077	0.03542933891	0.03330851419	0.03755016363
SSAB	0.002944166836	0.7681136272	0.0110758123	0.008954987585	0.01319663702
SSAC	0.00000001227505754	0.000003202481211	0.00002261549759	-0.00209820922	0.002143440215
SSBC	0.00955370172	2.492497501	0.01995171434	0.01783088962	0.02207253906
SSABC	0.001829213748	0.4772297513	0.008730248536	0.006609423819	0.01085107325
SST	0.3832983469	100			

Deviation of error	0.005950673988
Deviation of effects	0.001214676241
t-value [0.95;16]	1.746

	A (server)	B (mw)	C (workers)
-1	1	1	8
1	3	2	32

Write-only response time

I	A	B	C	AB	AC	BC	ABC	rep1	rep2	rep3	mean	err1	err2	err3
1	1	1	1	1	1	1	1	1.089579632	1.089967107	1.094812441	1.09145306	-0.001873428067	-0.001485952837	0.003359380904
1	1	1	-1	1	-1	-1	-1	1.218189462	1.214585701	1.227125087	1.21996675	-0.001777288155	-0.005381049214	0.00715833737
1	1	-1	1	-1	1	-1	-1	1.387617502	1.387427922	1.380785871	1.385277098	0.002340403889	0.002150823397	-0.004491227287
1	1	-1	-1	-1	-1	1	1	1.400812491	1.398960477	1.406797382	1.402190116	-0.001377625841	-0.003229639481	0.004607265322
1	-1	1	1	-1	-1	1	-1	1.034256594	1.034088827	1.033971572	1.034105664	0.0001509297353	-0.0000168375181	-0.0001340922172
1	-1	1	-1	-1	1	-1	1	1.120423787	1.132839827	1.124321457	1.12586169	-0.005437903169	0.006978136623	-0.001540233454
1	-1	-1	1	1	-1	-1	1	1.195339701	1.243646088	1.236075243	1.225020344	-0.02968064312	0.007570845473	0.01105489882
1	-1	-1	-1	1	1	1	-1	1.261416798	1.308439017	1.291043626	1.28696648	-0.02554968231	0.02147253678	0.004077145532
9.770841203	0.4269328459	-0.828066875	-0.2991288699	-0.1240279354	0.008275454064	-0.1414105621	-0.08179078165	**Total**						
1.22135515	0.05336660573	-0.1035083594	-0.03739110874	-0.01550349192	0.001034431758	-0.01767632026	-0.01022384771	**Effect variables q (=Total/8)**						

	SSj	Percentage (SSj/SST)	Effects	Confidence low	Confidence high
SSE	0.002435816266	0.6456268288			
SSA	0.06835187058	18.11704851	0.05336660573	0.04896915593	0.05776405553
SSB	0.2571355311	68.15522164	-0.1035083594	-0.1079058092	-0.09911090958
SSC	0.03355428031	8.893751097	-0.03739110874	-0.04178855854	-0.03299365894
SSAB	0.005768598284	1.528999485	-0.01550349192	-0.01990094172	-0.01110604212
SSAC	0.00002568117749	0.006806940824	0.001034431758	-0.003363018041	0.005431881557
SSBC	0.007498855152	1.987613819	-0.01767632026	-0.02207377006	-0.01327887046
SSABC	0.002508649486	0.6649316843	-0.01022384771	-0.01462129751	-0.005826397907
SST	0.3772792823	100			

Deviation of error	0.01233849734
Deviation of effects	0.002518585223
t-value [0.95;16]	1.746

Read-only throughput

I	A	B	C	AB	AC	BC	ABC	rep1	rep2	rep3	mean	err1	err2	err3
1	1	1	1	1	1	1	1	3.943991144	3.944123885	3.944309547	3.944141525	-0.0001503811867	-0.00001764040758	0.0001680215943
1	1	1	-1	1	-1	-1	-1	3.944129136	3.938696169	3.944097619	3.942307641	0.00182149522	-0.003611472601	0.001789977381
1	1	-1	1	-1	1	-1	-1	3.944151473	3.944345313	3.944448201	3.944314996	-0.0001635227356	0.00003031722495	0.0001332055107
1	1	-1	-1	-1	-1	1	1	3.943904733	3.943978062	3.944347649	3.944076814	-0.0001720819165	-0.00009875213702	0.0002708340535
1	-1	1	1	-1	-1	1	-1	3.466691932	3.466827373	3.466858574	3.466792626	-0.0001006943081	0.00003474686057	0.0000659474475
1	-1	1	-1	-1	1	-1	1	3.466563151	3.466706899	3.466681605	3.466650552	-0.00008740076519	0.00005634732972	0.00003105343547
1	-1	-1	1	1	-1	-1	1	3.467453508	3.466793502	3.467129136	3.467125382	0.0003281255521	-0.0003356337933	0.00000375412062
1	-1	-1	-1	1	1	1	-1	3.465920011	3.466021326	3.466294445	3.466078594	-0.0001585831601	-0.00005726792781	0.0002158510879
29.64148813	1.908193823	-0.001703441579	0.003260927917	-0.002181846713	0.0008832026682	0.0006909885739	0.002500416511	**Total**						
3.705186016	0.2385242278	-0.0002129301974	0.0004076159896	-0.0002727308391	0.0001104003335	0.00008637357174	0.0003125520638	**Effect variables q (=Total/8)**						

	SSj	Percentage (SSj/SST)	Effects	Confidence low	Confidence high
SSE	0.00002009669735	0.001471766738			
SSA	1.365451374	99.99781954	0.2385242278	0.2381247975	0.2389236581
SSB	0.000001088142455	0.00007968930634	-0.0002129301974	-0.0006123604669	0.0001865000721
SSC	0.000003987619079	0.0002920303283	0.0004076159896	0.000008185720086	0.0008070462591
SSAB	0.000001785170654	0.00013073565	-0.0002727308391	-0.0006721611086	0.0001266994304
SSAC	0.0000002925176074	0.00002142231022	0.0001104003335	-0.000289029936	0.000509830603
SSBC	0.0000001790494535	0.00001311255404	0.00008637357174	-0.0003130566977	0.0004858038412
SSABC	0.000002344531023	0.0001716999921	0.0003125520638	-0.00008687820563	0.0007119823333
SST	1.365481148	100			

Deviation of error	0.001120733503			
Deviation of effects	0.0002287687683			
t-value [0.95;16]	1.746			

	A (server)	B (mw)	C (workers)
-1	1	1	8
1	3	2	32

Read-only response time

I	A	B	C	AB	AC	BC	ABC	rep1	rep2	rep3	mean	err1	err2	err3
1	1	1	1	1	1	1	1	1.339417524	1.338966261	1.33887275	1.339085512	0.0003320121605	-0.0001192508843	-0.0002127612762
1	1	1	-1	1	-1	-1	-1	1.33896846	1.348930647	1.338994154	1.342297754	-0.003329293852	0.006632893545	-0.003303599692
1	1	-1	1	-1	1	-1	-1	1.339022951	1.33878952	1.338665163	1.338825878	0.0001970727075	-0.00003635799917	-0.0001607147084
1	1	-1	-1	-1	-1	1	1	1.339240873	1.33914568	1.338801857	1.339062803	0.0001780699741	0.0000828768947	-0.0002609468688
1	-1	1	1	-1	-1	1	-1	1.816144595	1.816234387	1.815995438	1.816124807	0.00001978863402	0.0001095802049	-0.0001293688389
1	-1	1	-1	-1	1	-1	1	1.816134402	1.816014527	1.815858172	1.816002367	0.0001320349666	0.00001216005562	-0.0001441950223
1	-1	-1	1	1	-1	-1	1	1.815473238	1.816037734	1.815763036	1.815758003	-0.0002847644477	0.000274698713	0.00000503286732
1	-1	-1	-1	1	1	1	-1	1.816956187	1.816898318	1.816598504	1.81681767	0.000138517277	0.00008064851741	-0.0002191657945
12.62397479	-1.905430899	0.003046085648	-0.004386394938	0.003943082309	-0.002511939495	-0.001793210918	-0.004157422779	**Total**						
1.577996849	-0.2381788624	0.000380760706	-0.0005482993672	0.0004928852887	-0.0003139924368	-0.0002241513648	-0.0005196778474	**Effect variables q (=Total/8)**						

	SSj	Percentage (SSj/SST)	Effects	Confidence low	Confidence high
SSE	0.00006663343439	0.004893783936			
SSA	1.361500092	99.99315418	-0.2381788624	-0.238906181	-0.2374515437
SSB	0.000003479489165	0.0002555454081	0.000380760706	-0.0003465579478	0.00110807936
SSC	0.000007215172707	0.0005299065944	-0.0005482993672	-0.001275618021	0.0001790192865
SSAB	0.000005830461787	0.0004282087588	0.0004928852887	-0.0002344333651	0.001220203942
SSAC	0.000002366190009	0.0001737809669	-0.0003139924368	-0.001041311091	0.0004133262169
SSBC	0.000001205852024	0.00008856183566	-0.0002241513648	-0.0009514700186	0.000503167289
SSABC	0.000006481561561	0.00047602772	-0.0005196778474	-0.001246996501	0.0002076408064
SST	1.361593304	100			

Deviation of error	0.002040732626
Deviation of effects	0.0004165628028
t-value [0.95;16]	1.746

7 Queuing Model (90 pts)

The goal of this section is to develop an analytical queuing model for each component of the system and for the system as a whole. We derive the performance characteristics that the model predicts and compare them with the results obtained in the experiments. We start with a naive model and progressively improve it to better fit our system.

7.1 M/M/1

In this subsection we build an M/M/1 model based on Section 4 (Throughput for Writes). First of all, we need to define the model boundaries: We choose the net-threads to be the entry point of the system, and the worker threads sending back a response to the clients mark the endpoint of the system. The single service of the M/M/1 model represents all the worker threads of the two middlewares together. Note that the service time also includes the memcached RTT because a service having finished serving a job corresponds in the real system to the event of a worker thread sending back a response to the client. The queue in front of the service represents the internal request queues of both middlewares together. This is a naive model and has several differences to the actual system which are discussed at the end of this subsection.

Input parameters of the model:

- *mean service rate* μ: We are going to analyze each worker configuration separately. That's why for each worker configuration we choose the maximum observed throughput as an approximation of its mean service rate.
- *mean arrival rate* λ: The mean arrival rate is computed as the throughput of the middleware for a specific client load and the analyzed worker configuration. Note that this can only be done because we are in a closed system. For the client load we choose 144 clients because we expect to observe an interesting change in behavior.

Predictions based on the model and the input values: First we list the metrics we want to predict and also the corresponding formulas. The formulas are derived from operational laws and can be seen in chapter 31 of the book "The Art of Computer Systems Performance Analysis" by Raj Jain[9]. We already include the formulas for the M/M/m model of the next subsection. Note that ϱ is the probability of m or more jobs in system:
$\varrho = [(mp)^m/(m!(1-p))] * p_0$ where p_0 is the probability of zero jobs in the system.

Metric	M/M/1	M/M/m
Traffic load/Utilization ρ	λ/μ	$\lambda/(m\mu)$
Mean num jobs in queue $E[n_q]$	$\rho^2/(1-\rho)$	$\rho\varrho/(1-\rho)$
Mean num jobs in service $E[n_s]$	ρ	$m\rho$
Mean num jobs in system $E[n]$	$\rho/(1-\rho)$	$m\rho + \rho\varrho/(1-\rho)$
Mean waiting time $E[w]$	$\rho\frac{1/\mu}{(1-\rho)}$	$\varrho/[m\mu(1-\rho)]$
Mean service time $E[s]$	$1/\mu$	$1/\mu$
Mean response time $E[r]$	$(1/\mu)/(1-\rho)$	$\frac{1}{\mu}(1+\frac{\varrho}{m(1-\rho)})$

Results are displayed in Table 1. First note that the traffic intensities ρ are strictly smaller than 1, thus the stability condition holds. Utilization for 8, 16 and 32 workers is almost at 100%,

[9]Raj Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, April 1991, ISBN: 978-0471503361.

which matches the behaviour of the system in Section 4 because workers are the bottleneck and the system is fully saturated at 144 client load (see Figure 29). For 64 workers, the client load of 144 does not reach saturation yet because there are enough workers that process requests. The model correctly predicts the utilization decrease for 64 workers.

The queue lengths are not accurately predicted, which might be because of the following reasons:

1. We have a fixed number of requests in the actual system because it is closed but in the model the queue length can increase even further because it corresponds to an open system. The model is not closed because it has a fixed arrival rate and there are no clients that wait for a response.

2. The actual system has two internal request queues instead of one queue.

3. Small changes in utilization around 100% result in large changes in the queue length because we only have one very fast service. Reaching high utilization of a single fast service has a more negative impact on the queue length than if multiple slower services reach their limit. This implies that for lower utilization we should have more accurate predictions, which is indeed the case, as can be seen from the queue length and the queue waiting time for 64 workers.

The number of jobs in service is very under-estimated because the model has only one service which cannot process more than one job at a time and thus $E[n_s] \leq 1$. But in the actual system multiple workers can process jobs in parallel and thus we can have more than one job receiving service. This also leads to the number of jobs in system being under-estimated.
Total response times are not accurately predicted because the service time is very underestimated and the waiting time is also not accurate because of the same reason the queue length is not accurately predicted.

	8 workers		**16 workers**		**32 workers**		**64 workers**	
	predicted	measured	predicted	measured	predicted	measured	predicted	measured
μ [req/s]	-	9'249.94	-	10'891.57	-	13'258.34	-	15'670.26
λ [req/s]	-	9'025.93	-	10'716.8	-	13'023.1	-	14'519.22
ρ	0.9758	-	0.984	-	0.9823	-	0.9265	-
$\mathbb{E}[n_q]$	39.32	59.7	60.33	48.39	54.38	27.2	11.69	6.94
$\mathbb{E}[n_s]$	0.98	16	0.98	32	0.98	64	0.93	128
$\mathbb{E}[n]$	40.29	75.7	61.32	80.39	55.36	91.2	12.61	134.94
$\mathbb{E}[w]$ [ms]	4.36	13.6	5.63	9.78	4.18	5.12	0.8	0.84
$\mathbb{E}[s]$ [ms]	0.11	1.77	0.09	2.99	0.08	4.91	0.06	7.19
$\mathbb{E}[r]$ [ms]	4.46	15.37	5.72	12.77	4.25	10.03	0.87	8.03

Table 1: Predicted and measured metrics of the M/M/1 model for all worker configurations under the load of 144 clients.

To conclude, we can say that the model is too simple and cannot predict the experiment results very well because of the following differences to the actual system: In reality we have two internal request queues and not only one because we have two middlewares. The worker threads correspond to multiple services that run in parallel and not to a single sequential service. The model represents an open system because jobs arrive at a fixed arrival rate, and there is no flow of requests where finished jobs are again enqueued at the beginning. But the actual system is

a closed one. Another difference is that memcached servers and workers are both in the same service instead of considered independently from each other.

7.2 M/M/m

In this subsection we build an M/M/m model based on Section 4. The model consists of multiple services that process and dequeue jobs from a single queue in parallel. A service corresponds to a single worker thread and again the service time includes the memcached RTT. The queue in front of the services represents the internal request queues of both middlewares together like in the M/M/1 model. This is a more accurate model of our system than the M/M/1 model and should lead to better predictions.

Input parameters of the model:

- *mean service rate* μ: There are multiple ways to approximate the mean service rate. Either we take the inverse of the shortest observed service time where the service time consists of the worker preprocessing time, the memcached RTT and the worker postprocessing time. Or we take the maximum observed throughput and divide it by the number of services m. We have chosen the second approach because it lead to better predictions.
- *mean arrival rate* λ: Average throughput of the middleware like in the M/M/1 model.
- *number of services* m: We have as many services as total number of workers in the system i.e. m = worker configuration x 2.

	8 workers		**16 workers**		**32 workers**		**64 workers**	
	predicted	measured	predicted	measured	predicted	measured	predicted	measured
μ [req/s]	-	578.12	-	340.36	-	207.16	-	122.42
λ [req/s]	-	9'025.93	-	10'716.8	-	13'023.1	-	14'519.22
ρ	0.9758	-	0.984	-	0.9823	-	0.9265	-
$\mathbb{E}[n_q]$	35.85	59.7	54.87	48.39	46.35	27.2	3.73	6.94
$\mathbb{E}[n_s]$	15.61	16	31.49	32	62.86	64	118.6	128
$\mathbb{E}[n]$	51.46	75.7	86.36	80.39	109.21	91.2	122.33	134.94
$\mathbb{E}[w]$ [ms]	3.97	13.6	5.12	9.78	3.56	5.12	0.26	0.84
$\mathbb{E}[s]$ [ms]	1.73	1.77	2.94	2.99	4.83	4.91	8.17	7.19
$\mathbb{E}[r]$ [ms]	5.7	15.37	8.06	12.77	8.39	10.03	8.43	8.03

Table 2: Predicted and measured metrics of the M/M/m model for all worker configurations under the load of 144 clients.

Predictions based on model and input values: Results are displayed in Table 2. First note that the traffic intensities ρ are strictly smaller than 1, thus the stability condition holds. The utilization predictions do match the reality pretty good as in the M/M/1 case. We now have higher service times, respectively lower service rates because we consider m services instead of a single one that models all the worker threads together. That's why the service time is now predicted more accurately than in the M/M/1 model where the service time was very short because of the extremely fast service, which lead to underestimated response times. Also the number of jobs in the service is predicted more accurately because it has no upper bound of 1 job anymore and thus also the total number of jobs is predicted more accurately. Note that still

the response times are mostly predicted too optimistic since this is a more ideal model than our system because of the following:

1. If there are jobs in the queue and there are available workers, jobs will be dequeued and processed. But in our system this may not be the case, because a worker may be ready to dequeue but his queue is empty while the queue in the other middleware is not.

2. Services operate at a fixed ideal service rate, which in a real system may vary. Also an open system has a fixed arrival rate which can be considered more ideal than a closed system that always has to wait for jobs to finish because if a job of a client is processed very slowly, the client cannot send another job during the processing time.

To conclude, we can say that the M/M/m model is more accurate than the M/M/1 model but the predictions are still not perfect because of the mentioned differences to the real system. In both models the whole system is modeled as a black box. To be able to more accurately predict the behavior of the system, we need to model each component of the system which is done in the next subsection.

7.3 Network of Queues

In this subsection we build a network of queues based on Section 3 (Baseline with Middleware). This is an even more accurate model of our system and should lead to better predictions than the M/M/m model. We consider both the one and two middleware configuration and analyze both read-only and write-only workloads. The number of worker threads is set to a constant.

7.3.1 System with one middleware

The model is shown in Figure 43. It is a closed queuing network, where jobs keep circulating from one queue to the next, i.e. we have a fixed number of jobs which we set equal to the number of clients. The network is modeled as a delay center. Delay centers have no queuing and the service time is independent of the load. The network delays can be computed from the measured ping times.
The memtier clients are modeled as a delay center because their queuing time is negligible and we adjust their service time to handle numerical instabilities. We only need a single delay center because the network delay between the clients and the middleware is roughly the same for all clients.
The net-thread is simulated with an M/M/1 model. It consists of one service because we have one net-thread and the queue represents the Java Selector in front of the net-thread which enqueues arriving requests from the clients. The service time is computed from the difference of timestamps when the net-thread reads a request and when it enqueues it.
The workers and the internal request queue is simulated with an M/M/m model where m is the number of workers. The service time includes the worker preprocessing time, the memcached RTT and the worker postprocessing time. We include the memcached servers in this component because of the Single-Resource Possession condition by Denning and Buzen (1978)[10] which states that a job may not be present at two or more devices at the same time. If memcached servers were modeled as another component, then as soon as a worker sends a job to a memcached server it would take another job from its queue instead of waiting for the response of the memcached server. This would not model the behavior of our system.

The model is implemented using the *Octave* programming language and the *queuing* package[11]. Since we use load-dependent service centers (for the workers), we cannot use the basic

[10] The Art of Computer Systems Performance Analysis by Raj Jain (chapter 32.2).
[11] Version 1.2.6 downloaded on `https://www.moreno.marzolla.name/software/queueing/`

MVA algorithm because they cannot be modeled using operational laws[12]. In this case we have to use the extended MVA algorithm, which is implemented in the mentioned queuing package of Octave.

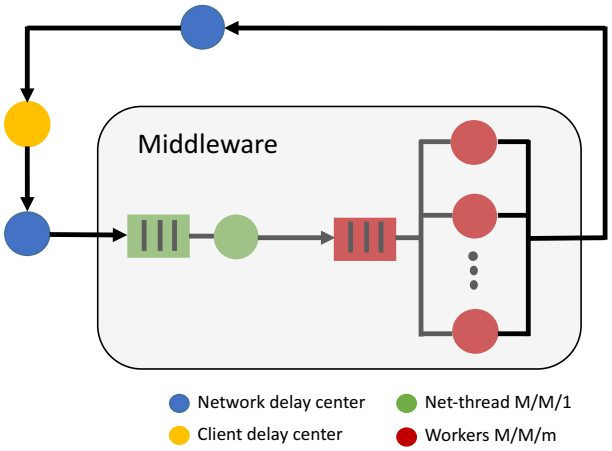

Figure 43: Network of queues model for one middleware.

The **input parameters** of the model are:
- worker service time and number of workers
- net-thread service time
- delays of the delay centers
- number of jobs circulating (= number of clients)
- visit ratios

Predicted metrics (from the output of Octave) and measured metrics for 8 workers and 48 client-load are shown in Table 3. The predictions for the system throughput and the response time of the middleware are quite accurate. The predictions for the length of the internal request queue are slightly overestimated. This is because $Q_{workers}$ actually not only includes the jobs waiting in the queue but also those receiving service at the workers. Note that there can be at most 8 jobs receiving service at the workers at the same time since we have 8 workers.
In contrast to the previous models, a network of queues allows us to do a bottleneck analysis. For the read-only workload, the bottleneck in our system of Section 3.1 was the memcached server respectively its outbound network bandwidth. This is correctly predicted by the model since $U_{workers} > U_{netthread}$ and based on our model $U_{workers}$ includes the memcached server. For the write-only workload, the bottleneck in our system (for 8 worker threads) were the worker threads, which is also correctly predicted as $U_{workers} > U_{netthread}$. The bottlenecks are at 100% utilization which conforms to our system which is also saturated at 48 client-load with 8 worker threads for both read-only and write-only workloads.

If we compare read-only and write-only net-thread utilization, we see that the net-thread utilization for write-only is 5x higher than for read-only. This is because the Set requests are bigger and thus the net-thread needs to read more data in contrast to the Get requests which just contain a key.

Another interesting point to mention is that if we do the write-only workload analysis for 64 workers, we have 58.11% utilization for the net-thread and 26.65% for the workers. Thus the model predicts that the bottleneck is on the net-thread and not on the worker threads anymore which exactly corresponds to the bottleneck analysis done in Section 3.1 as for higher number

[12]Exercise session slides on network of queues (page 20).

of workers the worker bottleneck is removed.

	read-only		write-only	
	predicted	measured	predicted	measured
throughput X [req/s]	2'964.4	2'938.08	7'255.6	7'112.42
response time R_{mw} [ms]	15.26	15.4	5.59	5.74
queue length $Q_{workers}$	45.16	34.31	40.17	32.51
utilization $U_{netthread}$ [%]	6.28	-	30.02	-
utilization $U_{workers}$ [%]	100	-	100	-

Table 3: *Network of Queues for 8 workers and 48 client-load (one middleware).*

7.3.2 System with two middlewares

The model is shown in Figure 44. Essentially everything is the same as in the one middleware case, but now the middleware part is duplicated i.e. the net-thread component is duplicated and also the worker thread component is duplicated. The traffic is split equally at the client among both net-threads.

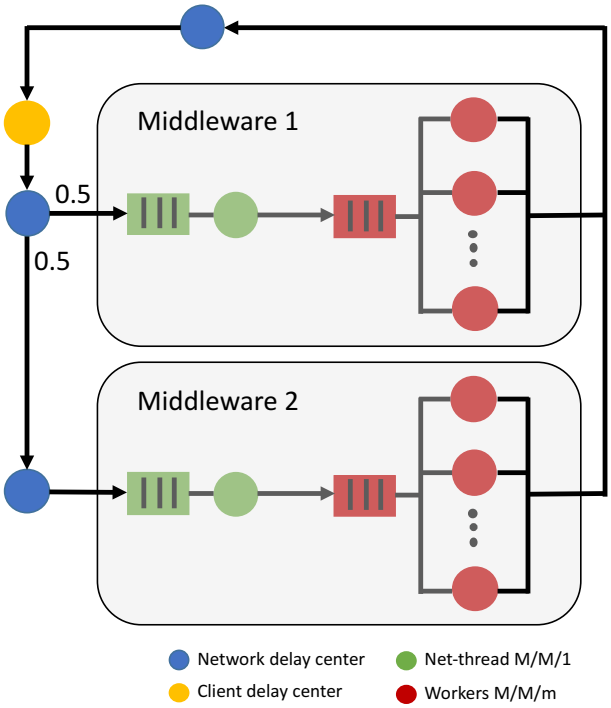

Figure 44: Network of queues model for two middlewares.

Predicted metrics (from the output of Octave) and measured metrics for 8 workers and 48 client-load are shown in Table 4. Again the model predicts the throughput and response time of the middlewares quite accurately. The queue length is slightly overestimated for the same reason that was given in the one middleware case.
For the read-only workload the predicted bottleneck are the worker threads because $U_{workers} > U_{netthread}$. This conforms to our bottleneck analysis in Section 3.2 where the memcached server is the bottleneck, because again the servers are part of the M/M/m component representing the worker threads. For the write-only workload, the predicted bottleneck are also the worker

threads for the same reason, which conforms to our bottleneck analysis in Section 3.2 where the worker threads are the bottleneck. The bottlenecks are almost at 100% utilization, which conforms to our system which is also saturated at 48 client-load with 8 worker threads for both read-only and write-only workloads.

If we compare the one middleware model predictions with the two middlewares model predictions, we see that especially for the write-only workload the utilization of the net-thread is significantly reduced going from one to two middlewares. This is because the two net-threads share the same load and we have less congestion in front of the Java Selector objects for the two middlewares configuration.

	read-only		**write-only**	
	predicted	measured	predicted	measured
throughput X [req/s]	2'869.2	2'944.13	9'184.1	9'406.65
response time R_{mw} [ms]	15.66	15.32	4.27	4.21
queue length $Q_{workers}$	22.44	11.63	19.47	10.67
utilization $U_{netthread}$ [%]	2.06	-	13.98	-
utilization $U_{workers}$ [%]	97.24	-	96.68	-

Table 4: *Network of Queues for 8 workers and 48 client-load (two middlewares).*

To conclude, we can say that the network of queues model provides accurate predictions of our system. But also network of queues have their limitations, and for example do not allow us to model the memcached servers and workers separately from each other.