Advanced Systems Lab Report

Autumn Semester 2018

Name: Philip Junker
Legi: 13-913-389

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

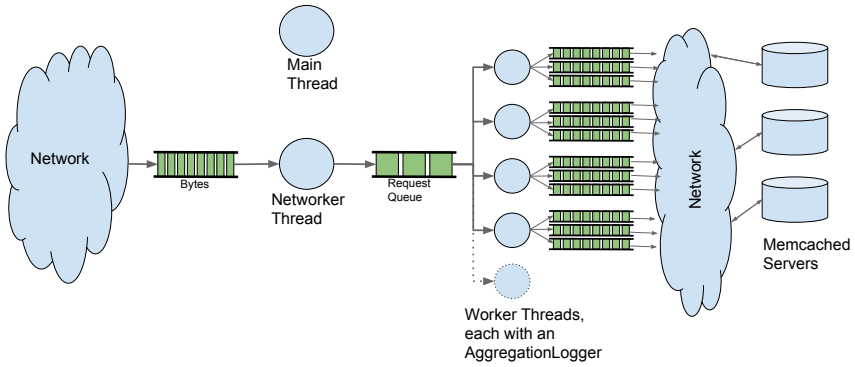


Figure 1: System Overview. Load generating memtier clients were omitted in illustration. All arrows to and from the network are bi-directional. The worker threads return the response in the end to the requesting client.

1 System Overview (75 pts)

My middleware is implemented as a consumer / producer system. It has a single networking thread, the producer, which accepts incoming requests and puts it into the queue. On the other hand it has a configurable amount of worker threads, the consumers, which consume the queue. They are connected via a Java BlockingQueue. For this implementation I chose the LinkedBlockingQueue implementation because it provides thread safety, separate read and write locks and no maximum capacity. All threads are created and started by the main thread. In addition, the main thread also creates the request queue and registers a shutdown hook, which assures that all logfiles are written before the system terminates on a SIGTERM signal. An overview of the system is provided in Figure 1.

1.1 Networker Thread

The networker thread handles the incoming connections from the clients and performs a minimal request parsing to determine whether a request is complete and therefore ready to be processed. For this, the networker uses Java's non-blocking I/O (NIO) mechanisms to set up an event polling loop over all connected sockets in its run method. A Selector, which is responsible to efficiently poll many file descriptors, is used to accept and register new connections. The networker thread reads any connected socket that becomes readable. An empty request structure is initialized during setup of a connection and stored using the Selector's attachment mechanism provided by Java NIO. The request structure, which is being discussed in more detail in the next paragraph, is used to parse and buffer incoming requests over the connection it is attached to and to store logs related to a request.

1.2 Request Parsing

Requests might arrive in multiple packets and therefore might not be complete within a single socket read operation. To make sure only complete requests are forwarded to the worker threads, the networker thread maintains a single request structure per connection into which incoming data is buffered until the request is complete. A single request structure per connection is sufficient because the system is closed, meaning clients wait until their previous request has

been answered before sending the next one. This design choice leads to a minimum number of request object allocations and because request objects are being reused the garbage collector is only needed on system shutdown. Because of the closedness of the system no synchronization on the request object is necessary, even though it is being shared between networker and worker thread, as long as the worker never uses the object after the response has been sent to the client.

Detecting whether a request is complete is done in a simplistic way in order to reduce the work done in the networker thread. The isComplete function only checks whether the last byte of the buffer is equal to a newline character. The rest of the parsing is then done inside the worker thread. By checking the first character in the buffer, the worker thread determines whether it is a get or a set request. Furthermore, if it is a get request, it parses the other arguments to determine whether it is a multiget (more than 1 key) or a get request (exactly 1 key). For a set request no further parsing is required, because it can be forwarded as it is to the memcached servers. During the parsing the buffer data is never being copied during a normal run, neither into other buffers or into String objects. The only exception is in case an error occurs to make it easier to trace the error, and in this case the performance is not the main target anymore. To achieve this, the request object only maintains a list of offsets which indicate the position of space and newline characters in the header of the request. In sharding mode, the offsets will be used by the worker handling the request to create byte buffer slices for each key range.

1.3 Worker Threads

Each worker thread continuously reads blockingly from the queue in its run method. Once it gets a request, it parses its type and then chooses the correct function to handle it (handleGet, handleMultiget, handleSet). In case sharded mode is deactivated, handleMultiget is equivalent to handleGet. Get and non-sharded multiget requests are load balanced among all connected memcached servers using a round-robin scheme. To avoid a shared token among the workers, each worker maintains its own counter which is initialized by the main thread in a way that the worker threads are spread out evenly among the servers and vice versa. Whenever a worker thread wants to access a server it uses the current counter value modulo the number of memcached server is used as an index to a list storing the server connections. On every balanced operation, the worker increases its counter. In expectation, this guarantees an even distribution of load, without incurring any coordination or hashing overhead. Sharded multiget requests are balanced similarly. In this setting a worker thread splits the keys into equally sized portions and sends each portion to one server, starting with its current round-robin index. Because the sharding scheme is a simple division, this leads to one server receiving one key more or less than the others. In expectation, this imbalance is avoided by choosing the starting server using the round-robin scheme and increasing the current round-robin index afterwards by the number of servers used. Each worker thread communicates blockingly with each server by sending requests and receiving responses one after another. Therefore the time to serve a sharded multiget request is at least as long as the slowest server requires to respond. For this project, non-blocking schemes were not considered in order to reduce complexity and keep service time analysis simple. Set requests are not load balanced but replicated to every server to the middleware. Again, replication is handled in a blocking way by forwarding the request to each server one after another.

1.4 Statistics of Requests

During the handling of a request in the middleware, various meta data is being attached to it, such as time and queue size when the request entered the middleware, number of cache misses,

waiting time in queue, service time on memcached servers, total time spent in middleware and information about which servers have been contacted to service this request. This meta data is used to log statistics about the processed requests. The middleware logs into two files using the log4j2 library, one file for errors and one for aggregated request statistics. The error file is synchronized, in order to not lose information in case of a crash. The requests file is buffered and only flushed to file at the shutdown of the system if the buffer limit of 10MB is not reached (for the experiments taken for this report, the file has always been flushed at shutdown). According to the log4j2 documentation, its file appenders are synchronized and hence safe to use in a multi thread environment like the one presented here (with multiple worker threads logging information about threads possibly simultaneously).

1.5 Aggregation of Requests

In order to avoid massive amounts of log data I decided to do a lightweight aggregation inside my middleware. Every worker thread receives the same starting timestamp as well as the desired granularity of aggregation at creation which it uses to initialize its AggregationLogger. Whenever a request is finished for a worker thread, it hands the request object to its AggregationLogger. The AggregationLogger checks if the request timestamp at hand is in the current time window. If that's the case, the AggregationLogger will add the meta data of the request at hand to its sums. If not, it will write the aggregated data of the current time window and set the new period start time such that the request timestamp is in the current time window (this is done by increasing the current start time by the desired granularity, e.g. 1s). The AggregationLogger only writes sums to the log, which means there are no errors in the aggregation data from e.g. division. In the aggregated log data there is one entry per second for every worker. To merge this data into one entry per second, the python script 'postprocessing.py' is used.

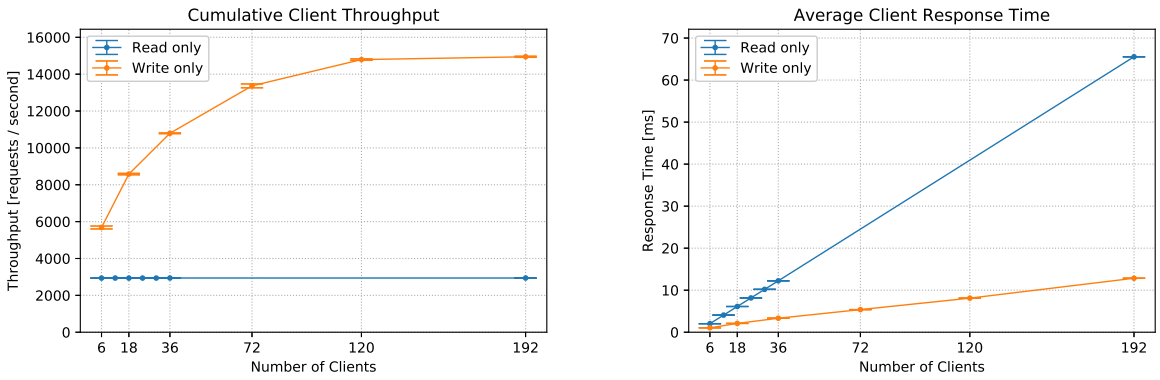
1.6 Preparation of Data

To account for the startup and the cooldown phase, the first 3 seconds and the last 3 seconds of each log were dropped, to account for warmup and cooldown phase respectively. These times were empirically determined by looking plots showing the performance of the middleware over time for various configurations. All aggregation windows are then aggregated into an overall per-run aggregate. For experiments with multiple middlewares and clients, the log files of the middlewares and the memtier clients are aggregated (using 'postprocessing_memtier.py' which calls the script for the middleware postprocessing) before creating the overall per-run aggregate. This is done for all three runs of each experiment, then averages and standard deviation are calculated. The results are then stored in the $aggregated_avg$ directory from which plots were generated. The chosen aggregation approach discards information about standard deviations on the 1 second window of the aggregated log. This is assumed to be acceptable for the setup of this project, because it matches the format in which memtier provides the information as well as we are mainly interested in showing that the obtained results hold across multiple runs.

2 Baseline without Middleware (75 pts)

2.1 One Server

In this section, the maximum capacity of a single memcached server is examined. The measured throughput is presented as the total accumulated throughput from all three client machines. The number of clients represents the total number of virtual clients across all three client machines. All results in this section are based on the output of memtier, which provides per second averages



(a) Throughput (b) Response Time

Figure 2: Baseline without Middleware and one Server

Client	Middleware	Server
25.1 MB/s	100.0 MB/s	12.6 MB/s

Table 2: Network Capacities (outgoing), measured using iperf

per virtual client. The first and last three seconds of each of the three repetitions were cut off as startup and cooldown time, leaving 60 seconds of data per repetition. All plots show averages and standard deviations across three repetitions of each configuration. The sanity of the data was checked with the interactive law for both throughput and response time separately and it was found that the interactive law aligns with the measured data. The configurations of the experiments are shown in Table 1.

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread (write-only)	[1,3,6,12,20,32]
Virtual clients per thread (read-only)	[1,2,3,4,5,6,32]
Workload	Write-only and Read-only
Repetitions	3

Table 1: Experiment configurations for subsection 2.1

2.1.1 Explanation

For read-only workload, the large value size of 4096 bytes causes GET responses to fully exhaust the outgoing capacity of the server's network link, while the CPU remains underutilized. Clearly, the bottleneck in this case is the network upload capacity of the single memcached server. The collected dstat data shows an average upload of 12.41MB/s already for the first configuration with 6 clients (see Table 3). Because this value is very close to the maximum outgoing link capacity of the server machines shown in Table 2, which was measured using iperf, we can conclude that this indeed is the bottleneck in this system for read-only workload. The system is already with 6 clients saturated because of the above mentioned bottleneck. No under- and

Configuration	CPU	Send	Receive
Server, read-only, 6 clients	11.0%	12.41 MB/s	0.44 MB/s
Clients, read-only, 6 clients	5.44%	0.16 MB/s	4.24 MB/s
Server, write-only, 120 clients	93.28%	2.05 MB/s	63.33 MB/s
Clients, write-only, 120 clients	18.09%	20.97 MB/s	0.69 MB/s

Table 3: Machine Stats during Experiments for subsection 2.1

over-saturation of the system has been observed for read-only workload.

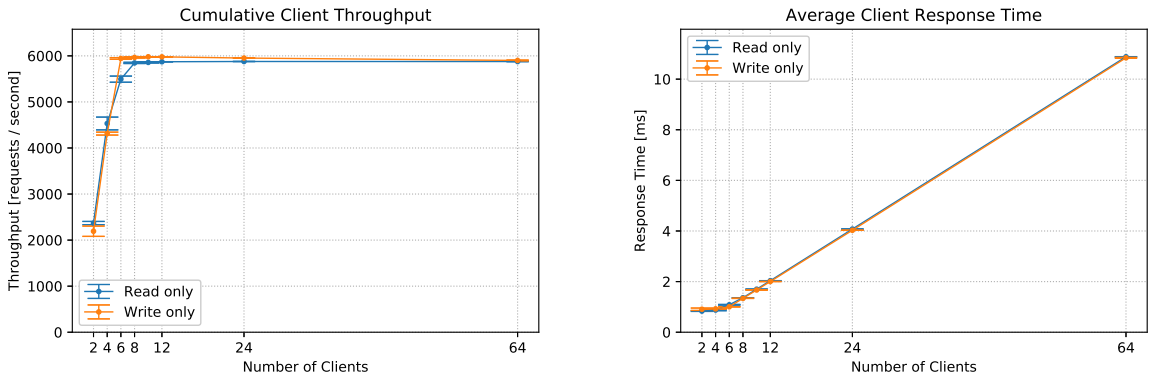
For write-only workload, the request size is large compared to the server response. The outgoing link capacity of the server should therefore not be the problem anymore. And indeed, this time outgoing network activities for both client and server machines stay within their outgoing link capacities. However, for the memcached server a CPU utilization of over 93% has been measured for 120 clients as shown in Table 3. Clearly, the bottleneck for the write-only workload is CPU bound on the memcached server side which implies that further increasing the number clients will not lead to a higher throughput and might put the system into an over-saturated state. Up until 120 clients the system is under-saturated because we observe an increase of throughput with a higher number of clients until this point. After that point the system is saturated because the throughput stays the same while the response time increases linearly with more clients. No over-saturation of the system has been observed for write-only workload.

2.2 Two Servers

In this section, the maximum load generation capacity of a single memtier instance is examined. The measured throughput is presented as the total accumulated throughput of the two memtier threads. As before, the number of clients represents the total number of virtual clients across all three client machines. All results in this section are based on the output of memtier, which provides per second averages per client. The first and last three seconds of each of the three repetitions were cut off as startup and cooldown time, leaving 60 seconds of data per repetition. All plots show averages and standard deviations across three repetitions of each configuration. The sanity of the data was checked with the interactive law for both throughput and response time separately and it was found that the interactive law aligns with the measured data. The configurations of the experiment are shown in table Table 4.

Number of servers	2
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread (write-only)	[1,3,6,12,20,32]
Virtual clients per thread (read-only)	[1,2,3,4,5,6,32]
Workload	Write-only and Read-only
Repetitions	3

Table 4: Experiment configurations for subsection 2.2



(a) Throughput

(b) Response Time

Figure 3: Baseline without Middleware and two Servers

Configuration	**CPU**	**Send**	**Receive**
Server, read-only, 8 clients	11.40%	12.45 MB/s	0.44 MB/s
Clients, read-only, 8 clients	21.34%	0.87 MB/s	24.95 MB/s
Server, write-only, 6 clients	12.50%	0.42 MB/s	12.88 MB/s
Clients, write-only, 6 clients	17.91%	24.66 MB/s	0.83 MB/s

Table 5: Machine Stats during Experiments for subsection 2.2

2.2.1 Explanation

For read-only workload the system is under-saturated until 8 clients. This is derived from the fact that the throughput is increasing significantly while the response time remains almost constant (see Figure 3). From 8 clients on, the throughput does not increase anymore while the response time continues to increase linearly. No over-saturation has been observed during the experiment. On the response time plot a knee is clearly visible at 6 clients. We identified the start of the saturation phase at 8 clients because of the throughput graph which increases by over 500 requests per second from 6 to 8 clients with only a small increase in latency. From this point on the system is saturated. The bottleneck for read-only workload is, as in subsection 2.1, at the outgoing network capacity of the server. As can be seen in Table 5 the server reaches an average outgoing network traffic of 12.45MB/s for 8 clients. This is very close to the maximum outgoing network capacity of the server machines (12.6MB/s) listed in Table 2.

For write-only workload the system is under-saturated up until 6 clients. This is derived from the fact that throughput is increasing significantly while the response time remains almost constant (see Figure 3). From 6 clients on, the throughput does not increase anymore while the response time continues to increase linearly. This time we observe the knee in the response time graph of Figure 3 at 6 clients and it aligns well with the throughput which reaches its peak there as well. From this point on, the system is saturated. No over-saturation has been observed during the experiment. The bottleneck for write-only workload has been identified at the outgoing network capacity of the single memtier client. As shown in Table 5 the client reaches an average outgoing network traffic of 24.66MB/s for 6 clients. This is very close to the maximum outgoing network capacity of the client machines (25.1MB/s) presented in Table 2.

2.3 Summary

This section is concluded by presenting the maximum throughput measured during the experiments in Table 6. The configurations at which the maximum throughput has been identified is stated in the last column. As explained before, it is at the beginning of the saturation phase.

For one memcached server and read-only workload the bottleneck is the network upload capacity of the single memcached server which is exhausted already with 6 clients. For one memcached server and write-only workload the bottleneck is the CPU of the memcached server, which has been measured at 96% in average for 120 clients. For one load generating VM and read-only workload the bottleneck is again the network upload capacity of the memcached servers, measurements of dstat show that the network upload of both servers is in average 12.47MB/s with 8 clients. Compared to the read-only experiment with a single memcached server in subsection 2.1 we observe a maximum throughput which is double as high. This is expected and makes sense because in this experiment we now have 2 memcached servers and because the bottleneck is the upload capacity a throughput increase of factor 2 is observed. For one load generating VM and write-only workload the bottleneck is on the client side where an average network upload of 24.8MB/s has been measured for 12 clients. This is the reason why the maximum throughput measured in subsection 2.1 for write-only workload is not being reached here, because the single load generating VM cannot generate more load due to its maximum network upload capacity.

If we compare read-only and write only workload we observe response time which is over 5x higher for read-only workload compared to write-only workload. Our hypothesis is, that this is due to network congestion at the network interface of the server for read-only workload.

	Read-only workload [req/s]	Write-only workload [req/s]	Configuration gives max. throughput
One memcached server	2937	14791	read-only: 6 clients write-only: 120 clients
One load generating VM	5847	5943	read-only: 8 clients write-only: 6 clients

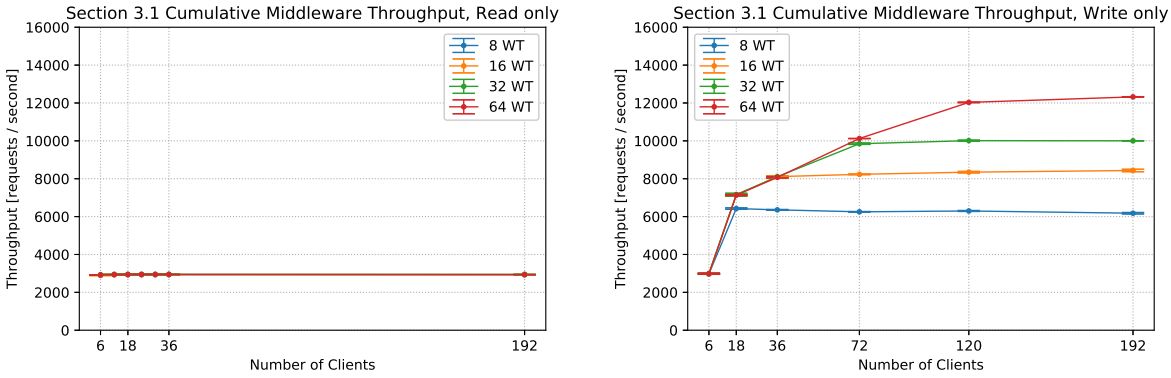
Table 6: Maximum throughput of different VMs.

3 Baseline with Middleware (90 pts)

3.1 One Middleware

After establishing baselines for clients and memcached servers in section 2, this section analyzes how adding my middleware to the system affects the performance. Due to the additional network hops from client to middleware and from middleware to server it is unavoidable that some additional latency is introduced. The middleware is designed to hide this latency with parallelism using its worker threads and, given a sufficient amount of parallelism, avoid becoming the bottleneck of the system.

The results in this section are based on the aggregated request logs from the middleware. The first and last three seconds of each of the three repetitions were cut off as startup and cooldown time, leaving 60 seconds of data per repetition. As a sanity check, the same data



(a) Throughput, read only

(b) Throughput, write only

Figure 4: Throughput of baseline with one Middleware, one Server, subsection 3.1

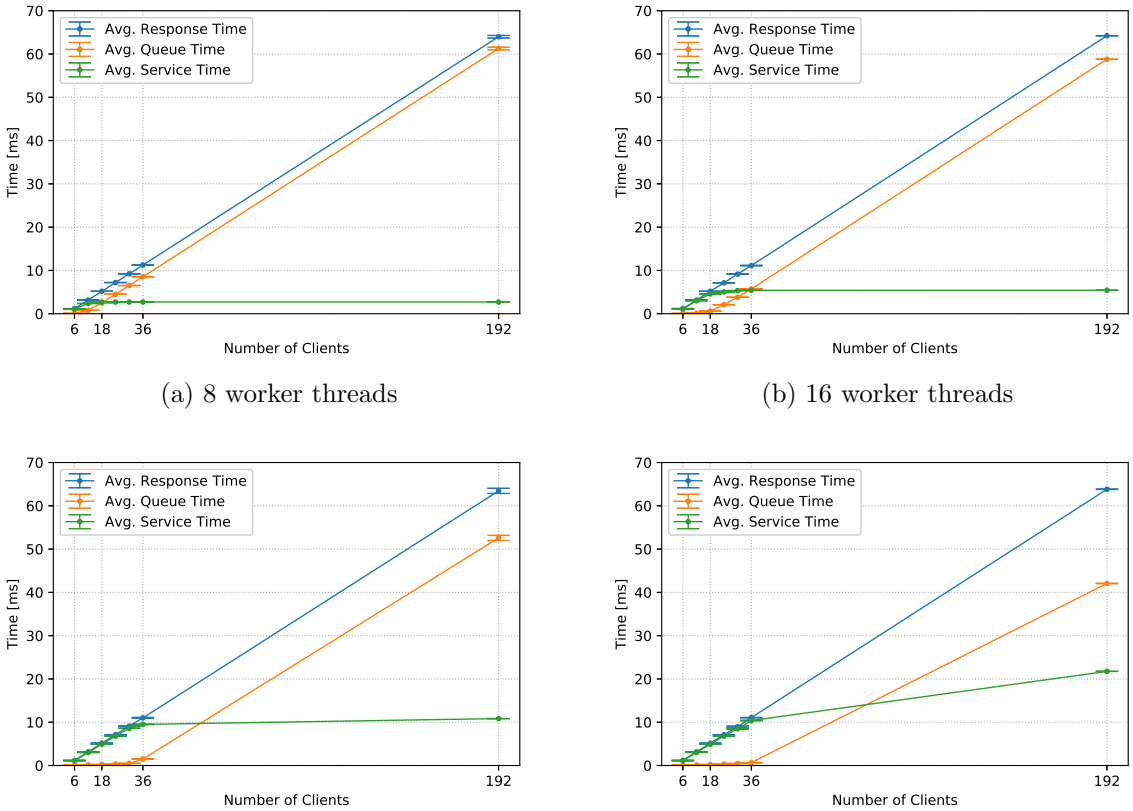
Configuration	**CPU**	**Send**	**Receive**
Server, read-only, 8WT, 6 clients	10.75%	12.20 MB/s	0.43 MB/s
Server, read-only, 16WT, 6 clients	11.35%	12.17 MB/s	0.43 MB/s
Server, read-only, 32WT, 6 clients	13.62%	12.27 MB/s	0.43 MB/s
Server, read-only, 64WT, 6 clients	15.01%	12.24 MB/s	0.43 MB/s

Table 8: Machine Stats during read-only Experiments for subsection 3.1

has been extracted from the aggregated memtier client outputs, compared and found to align well. No misses occurred during the test and all client requests were processed and answered. As before, the number of clients represents the total number of virtual clients across all three client machines. All plots show averages and standard deviations across three repetitions of each configuration. The sanity of the data has been checked with the interactive law for both throughput and response time separately for all configurations and it was found that the interactive law aligns with the measured data. The configurations examined and the setup for this section are shown in Table 7.

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread (write-only)	[1,3,6,12,20,32]
Virtual clients per thread (read-only)	[1,2,3,4,5,6,32]
Workload	Write-only and Read-only
Number of middlewares	1
Worker threads per middleware	[8, 16, 32, 64]
Repetitions	3

Table 7: Experiment configurations for Section 3.1



(a) 8 worker threads

(b) 16 worker threads

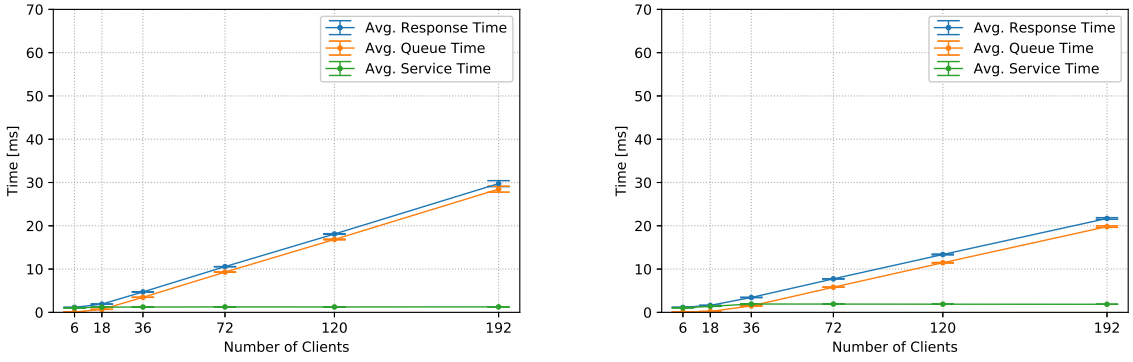
(c) 32 worker threads

(d) 64 worker threads

Figure 5: Response-, Queue- and Server-Service-Times of baseline with one Middleware, one Server, subsection 3.1, Read-only workload

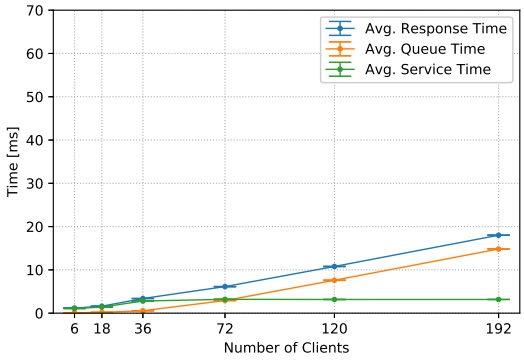
Configuration	**CPU**	**Send**	**Receive**
Server, write-only, 8WT, 192 clients	21.96%	0.86 MB/s	26.51 MB/s
Server, write-only, 16WT, 192 clients	45.52%	1.17 MB/s	36.14 MB/s
Server, write-only, 32WT, 192 clients	65.48%	1.39 MB/s	42.81 MB/s
Server, write-only, 64WT, 192 clients	80.70%	1.71 MB/s	52.65 MB/s
Server, write-only, 128WT, 192 clients	90.87%	1.94 MB/s	59.75 MB/s

Table 9: Machine Stats during write-only Experiments for subsection 3.1

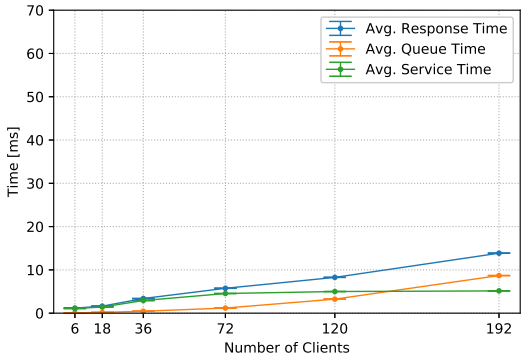


(a) 8 worker threads

(b) 16 worker threads



(c) 32 worker threads



(d) 64 worker threads

Figure 6: Response-, Queue- and Server-Service-Times of baseline with one Middleware, one Server, subsection 3.1, Write-only workload

3.1.1 Explanation

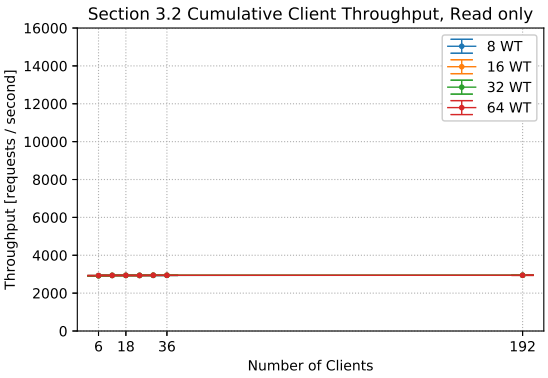
For read-only workload the server reaches its maximum outgoing network capacity of 12.5MB/s (see Table 2) already almost with 6 clients. As Table 8 shows, the measured send capacity of dstat shows for all worker thread configurations a maximum of around 12.2MB/s. With 12 clients this value increases up to 12.4MB/s in average. Because this corresponds to an increase of the total throughput of only 40 requests per second the beginning of saturation phase is observed to be at 6 clients. From this point on, the system is in saturation phase as can be seen in Figure 4 (a) for all configurations of worker threads. Hence, no under- and no over-saturation phase has been observed for read-only workload. A maximum throughput of almost 3k requests per second is reached, which is similar to the maximum throughput for read-only workload measured in subsection 2.1. Like in subsection 2.1 the bottleneck is the single memcached server with its limited outgoing network capacity. Figure 5 shows how the average service time, the time the server needs to respond to a request from the middleware, increases with more worker threads. This is because more connections to the single memcached server are established in parallel when having more worker threads. The figure also shows nicely how the queuing time in the middleware remains almost zero until the point where the number of clients exceeds the number of worker threads. Up until the point where there are at least as many worker threads as clients connecting, because we are in a closed system and clients send requests only one after another, each request is taken from the queue immediately.

For write-only workload the situation is different. This time the server is not the bottleneck until 64 worker threads. Figure 4 (b) shows nicely how the throughput increases with more worker threads. This leads to the conclusion that the middleware is the bottleneck, because if the number of worker threads in the middleware is increased, the throughput increases too. The dstat logs of all machines confirm this hypothesis and show that the network traffic stays within the maximum capacities listed in Table 2 and the CPU usage of all machines stays below 70%. Table 9 shows that for 128 worker threads the CPU usage of the server rises up until 90% which is an indicator that the server is now the bottleneck and more worker threads will not lead to a higher throughput from this point on. For 8 and 16 worker threads the system is under-saturated until 18 clients, as the average response time remains constant up until this point as shown in Figure 6 (a) and (b) and then starts to increase linearly. For 32 worker threads the system is under-saturated until 36 clients, again, the average response time remains constant up until this point and then starts to increase linearly as shown in Figure 6 (c). For 64 worker threads the system changes from under-saturated to saturated at 120 clients, as the average response time remains almost constant up until this point as shown in Figure 6 (d) and then increases linearly. No over-saturation has been observed for all worker thread configurations. For read-only workload we observe again a significantly higher response time compared to write-only workload. This has already been observed and explained in section 2 and we assume that it is due to the same reason in this experiment.

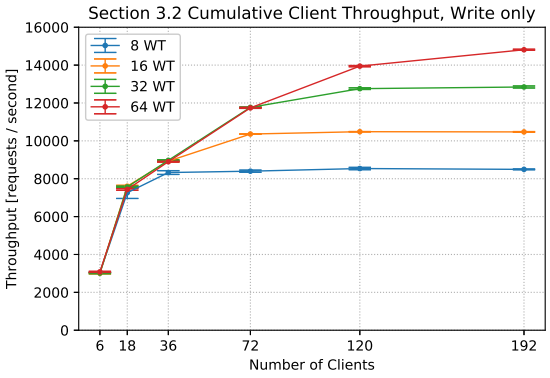
3.2 Two Middlewares

In this section the performance of two middlewares is investigated, splitting the load into two, connecting each client machine to both middlewares.

The results in this section are based on the aggregated request logs from the two middlewares. The first and last three seconds of each of the three repetitions were cut off as startup and cooldown time, leaving 60 seconds of data per repetition. As a sanity check, the same data has been extracted from the aggregated memtier client outputs, compared and found to align well. No misses occurred during the test and all client requests were processed and answered. As before, the number of clients represents the total number of virtual clients across all three



(a) Throughput, read only



(b) Throughput, write only. Please note that for 64WT also 240 and 384 clients have been measured and found that the throughput does not increase further. We leave out these data points to show the same x-range as for the other graphs.

Figure 7: Throughput of baseline with two Middlewares, one Server, subsection 3.2

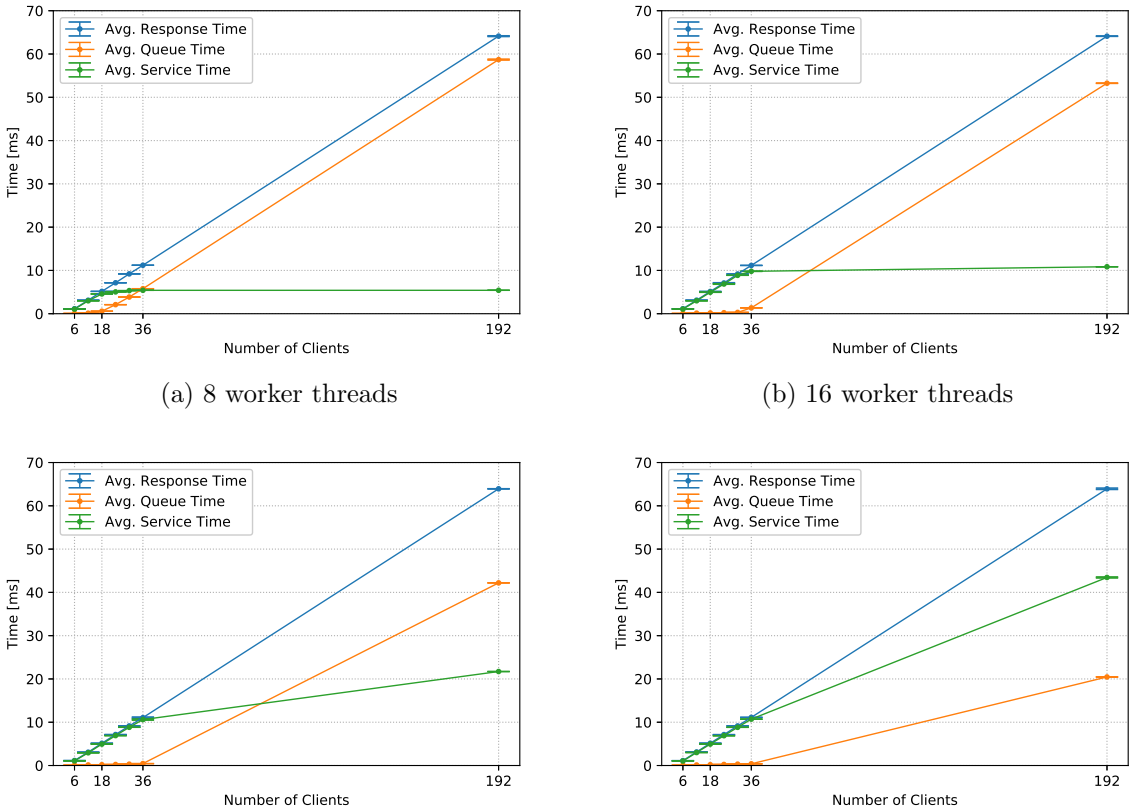
Configuration	**CPU**	**Send**	**Receive**
Server, read-only, 8WT, 6 clients	11.73%	12.29 MB/s	0.43 MB/s
Server, read-only, 16WT, 6 clients	14.25 %	12.29MB/s	0.43 MB/s
Server, read-only, 32WT, 6 clients	14.97%	12.25 MB/s	0.43 MB/s
Server, read-only, 64WT, 6 clients	16.14%	12.26 MB/s	0.43 MB/s

Table 11: Machine Stats during read-only Experiments for subsection 3.2, two middlewares

client machines. All plots show averages and standard deviations across three repetitions of each configuration. The sanity of the data has been checked with the interactive law for both throughput and response time separately for all configurations and it was found that the interactive law aligns with the measured data. The configurations examined and the setup for this section are shown in Table 10.

Number of servers	1
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread (write-only)	[1,3,6,12,20,32]
Virtual clients per thread (read-only)	[1,2,3,4,5,6,32]
Workload	Write-only and Read-only
Number of middlewares	2
Worker threads per middleware	[8, 16, 32, 64]
Repetitions	3

Table 10: Experiment configurations for subsection 3.2



(a) 8 worker threads

(b) 16 worker threads

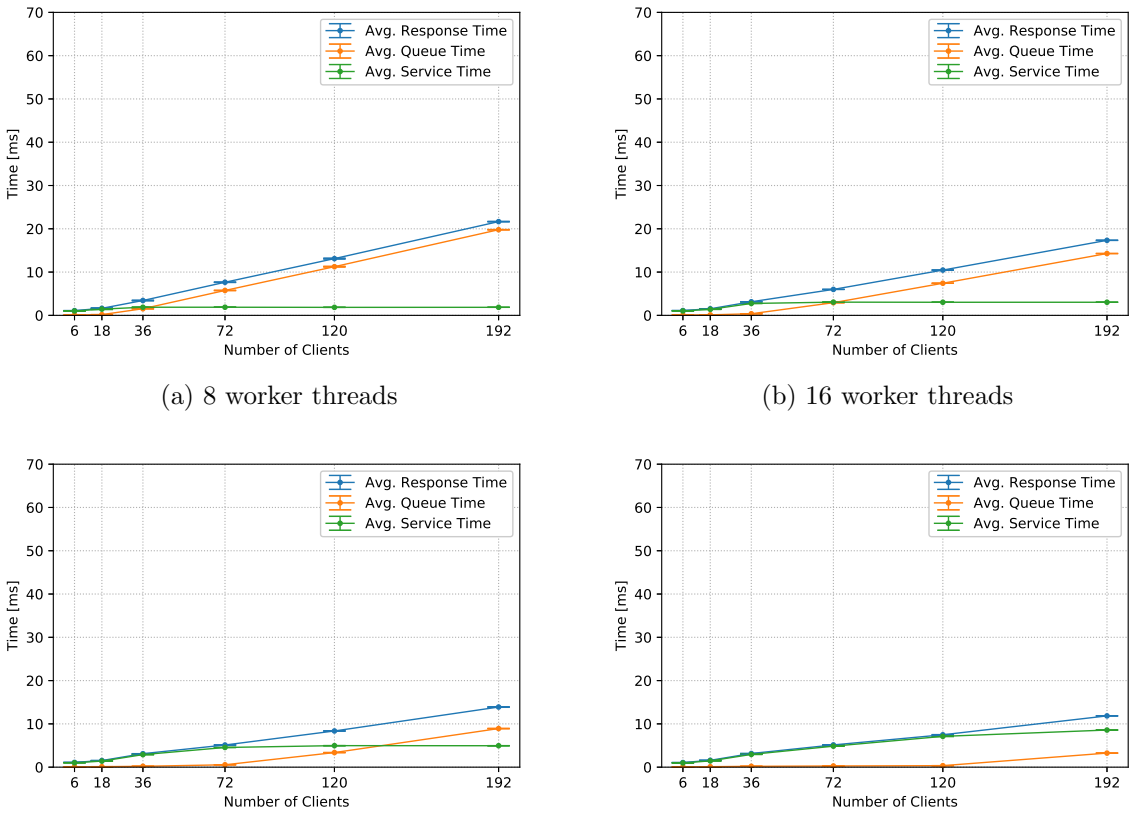
(c) 32 worker threads

(d) 64 worker threads

Figure 8: Response-, Queue- and Server-Service-Times of baseline with two Middlewares, one Server, subsection 3.2, Read-Only workload

Configuration	**CPU**	**Send**	**Receive**
Server, write-only, 8WT, 192 clients	51.19%	1.18 MB/s	36.37 MB/s
Server, write-only, 16WT, 192 clients	65.11%	1.45 MB/s	44.75 MB/s
Server, write-only, 32WT, 192 clients	81.20%	1.77 MB/s	54.75 MB/s
Server, write-only, 64WT, 192 clients	93.17%	2.05 MB/s	63.48 MB/s
Server, write-only, 128WT, 192 clients	95.18%	2.02 MB/s	62.38 MB/s

Table 12: Machine Stats during write-only Experiments for subsection 3.2, two middlewares



(a) 8 worker threads

(b) 16 worker threads

(c) 32 worker threads

(d) 64 worker threads

Figure 9: Response-, Queue- and Server-Service-Times of baseline with two Middlewares, one Server, subsection 3.2, Write-Only workload

3.2.1 Explanation

For read-only workload we observe a similar throughput as in subsection 3.1 for read-only workload. This is no surprise as also with two middlewares the server network upload reaches its maximum upload capacity already with 6 clients. Table 11 shows several machine stats measured with dstat during the experiments for several worker thread configurations. The response, queue and service times plotted in Figure 8 look very similar as in subsection 3.1 if the 8 worker threads configuration is compared to the 16 worker threads graph of subsection 3.1, the 16 worker threads graph is compared to the 32 worker threads graph and so on. This is the case because doubling the amount of middlewares is equal to doubling the amount of worker threads.

For write-only workload we also observe a similar throughput as in subsection 3.1, with the maximum throughput of 8 worker threads being similar to 16 worker threads in subsection 3.1. Figure 7 shows again that the two middlewares are the bottleneck up until 32 worker threads because an increase of worker threads leads to an increase of throughput. For 64 worker threads Table 12 shows a CPU usage of over 93% on the server which implies that the bottleneck now lies at the server and that a further increase of worker thread would not lead to an increase in throughput. Furthermore, Figure 7 shows that doubling the number of middleware is almost equal to doubling the amount of workerthreads. However, we observe a slightly higher throughput for 2 middlewares compared to 1 middleware with double the amount of worker threads from 16 clients on. This can be due to the worker threads being run on 2 different machines which means that there is double the amount of CPU power available for 2 middlewares.

3.3 Summary

This section is concluded by presenting the maximum throughput as well as the average response time and the average time in queue measured at the maximum throughput configuration in Table 13. The configuration which has been identified as the maximum throughput configuration is described in the first column.

For one middleware and read-only workload a maximum throughput of 3k requests per second is reached already with 8 worker threads, which is very close to the maximum throughput for read-only workload measured in subsection 2.1. As in subsection 2.1, the bottleneck in this case is the single memcached server with its limited outgoing network capacity.

For one middleware and write-only workload a maximum throughput of 12k requests per second is observed for 64 worker threads. The bottleneck now lies at the middleware because the throughput keeps on increasing with a higher amount of worker threads. If we compare the maximum throughput to the one of subsection 2.1 for write only throughput we realize that it is around 3k requests per second lower with the middleware in between clients and server and 64 worker threads. Hence we conclude that even for 64 worker threads the middleware is the bottleneck. This is confirmed also by the write-only experiment of subsection 3.2 where the maximum throughput is now at 14.8k requests per second and because this is measured during the configuration with 2 middlewares with 64 worker threads each, this is expected to be equivalend to 1 middleware with 128 worker threads.

Comparing the one middleware system with the two middleware system we observe that, for write-only workload, we would expect that having 2 middlewares is equal to having double the amount of worker threads. During our experiments we observed a significantly higher maximum throughput for the 2 middleware system compared to the single middleware system with an equal total number of worker threads. We concluded that this comes from the additional CPU power because the 2 middlewares are running on two separate machines and hence they have double the CPU power compared to the single middleware system. For read-only workload we measure

exactly the same maximum throughput for single and double middleware systems due to the bottleneck being at the server network upload capacity.

For two middlewares and write-only workload, as mentioned before, a maximum throughput of almost 15k requests per second is measured. This is close to the maximum throughput measured in subsection 2.1 which leads us to the conclusion that at this point the middleware is no longer the bottleneck. In subsection 2.1 we observed the CPU of the single memcached server to be the bottleneck. We expect this to be the same here and Table 12 confirms this hypothesis by showing that the average server CPU usage is at 93% for 64 worker threads per middleware and 192 clients.

For two middlewares and read-only workload a maximum throughput of 3k requests per second is measured. We expect this throughput to be similar as in subsection 2.1, which is actually, with a difference of only 2 requests per second, the case. Table 11 shows that, like in subsection 2.1, the maximum network upload capacity of the single memcached is reached already for 8 worker threads and 6 clients.

As observed in Section 2.1 the response time for read-only workload is much higher compared to write-only workload. We expect this to be the same for this experiment and in addition observe how the queueing time in the middleware increases once the system reaches the saturation phase. Before the saturation phase, the server service time dominates the response time while in the saturation phase the queueing time does.

	Throughput [req/s]	Response time [ms]	Avg. time in queue [ms]	Miss rate
Reads: Measured on MW (8WT, 12 clients)	2939	3.1	0.79	0.0
Reads: Measured on clients (8WT, 12 clients)	2939	4.1	n/a	0.0
Writes: Measured on MW (64WT, 120 clients)	12030	8.27	3.24	n/a
Writes: Measured on clients (64WT, 120 clients)	12034	9.97	n/a	n/a

(a) For one middleware

	Throughput [req/s]	Response time [ms]	Avg. time in queue [ms]	Miss rate
Reads: Measured on MW (8WT, 12 clients)	2935	3.12	0.12	0.0
Reads: Measured on clients(8WT, 12 clients)	2935	4.08	n/a	0.0
Writes: Measured on MW (64WT, 192 clients)	14810	11.84	3.25	n/a
Writes: Measured on clients (64WT, 192 clients)	14813	12.96	n/a	n/a

(b) For two Middlewares

Table 13: Maximum Throughput

4 Throughput for Writes (90 pts)

4.1 Full System

The setup from the previous section is now expanded even further for write-only workload. In this experiment the effect of replication can be observed as each middleware is replicating the SET requests to all three memcached servers.

The results in this section are based on the aggregated request logs from the two middlewares. The first and last three seconds of each of the three repetitions were cut off as startup

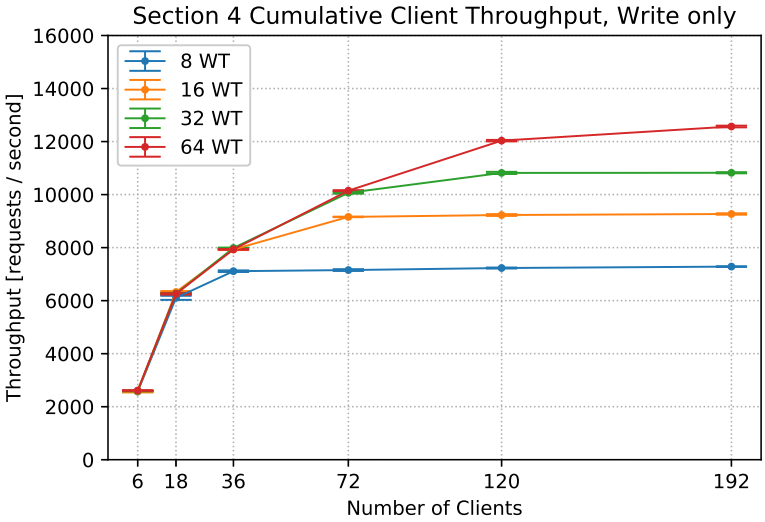


Figure 10: Throughput of full system, write only, subsection 4.1. Please note that for 64WT also 240 and 384 clients have been measured and found that the throughput does not increase further. We leave out these data points to show the same x-range as for the other graphs.

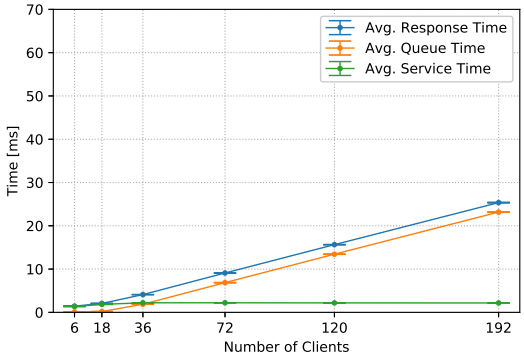
and cooldown time, leaving 60 seconds of data per repetition. As a sanity check, the same data has been extracted from the aggregated memtier client outputs, compared and found to align well. No misses occurred during the test and all client requests were processed and answered. As before, the number of clients represents the total number of virtual clients across all three client machines. All plots show averages and standard deviations across three repetitions of each configuration. The sanity of the data has been checked with the interactive law for both throughput and response time separately for all configurations and it was found that the interactive law aligns with the measured data. The configurations examined and the setup for this section are shown in Table 14.

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, 3, 6, 12, 20, 32]
Workload	Write-only
Number of middlewares	2
Worker threads per middleware	[8, 16, 32, 64]
Repetitions	3

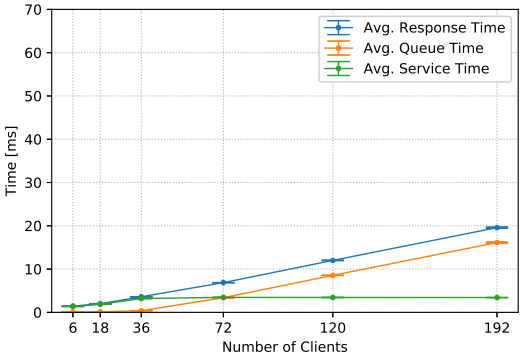
Table 14: Experiment configurations for subsection 4.1

4.1.1 Explanation

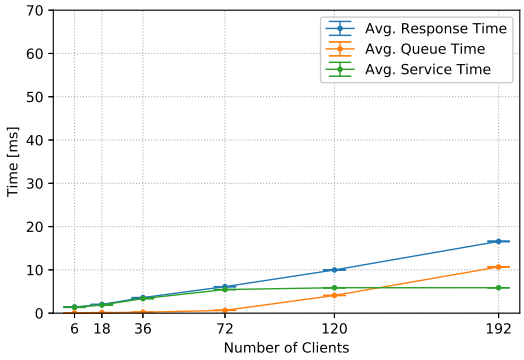
Compared to subsection 3.2 there are now 3 servers instead of 1 in the system. For write-only workload the middleware replicates each request to all 3 servers. This comes with some overhead because the middleware needs to send it to each server one by one and receive it one by one. In addition, because of the way the middleware is built, it has to wait for the slowest server. We



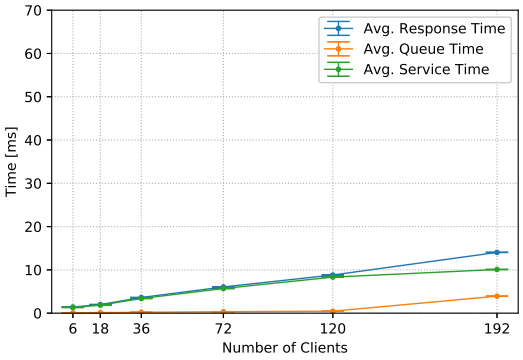
(a) 8 worker threads



(b) 16 worker threads



(c) 32 worker threads



(d) 64 worker threads

Figure 11: Response-, Queue- and Server-Service-Times of full system, Write-only workload for subsection 4.1

measured the ping times between the two middlewares and the single server for subsection 3.2, as well as for this experiment between the middlewares and the three servers and averaged the times per worker thread configuration. We did not observe a higher average ping time for the current section with the replication to three servers when looking at the slowest server. We conclude that the difference in server service time compared to subsection 3.2 has to come from the overhead within the middleware, namely the uploading of the large request payload to 3 instead of 1 memcached server. In Figure 10 we observe a significantly lower maximum throughput for all worker thread configurations compared to subsection 3.2, namely between 12.5% and 20% less throughput from 8 up to 64 worker threads. When looking at Figure 11 we observe that the total response time is slightly higher compared to the one measured in subsection 3.2. At the maximum throughput configuration for each worker thread configuration (see Table 15) a difference of the peak service time (average time waiting for all responses from the memcached servers) compared to subsection 3.2 has been measured of 0.33ms, 0.41ms, 0.9ms and 1.5ms for 8, 16, 32 and 64 worker threads respectively. We also observed that the average time in the queue differs also roughly by this time. The amount of requests in the queue has been observed to be similar compared to subsection 3.2. The aggregated and averaged machine stats collected with dstat do not show any machine bottleneck being reached. While in subsection 3.2 the CPU of the memcached server was the bottleneck, the average CPU utilization of all three memcached servers is now around 80% with 64 worker threads per middleware which is excluding the server CPU as the bottleneck for this experiment. We conclude that the increased service time due to the middleware implementation leads to a lower throughput, which leads to a lower arrival rate at the servers and therefore to less work per server. Hence we identify the bottleneck for this experiment as the middleware and expect the throughput to increase with more worker threads. And indeed, when performing the experiment using 128 worker threads per middleware, a further increase of the maximum throughput can be measured up until 14k requests per second - this confirms our hypothesis that the bottleneck lies at the middlewares until 64 worker threads, namely at the non sufficient parallelization.

For 8 worker threads the system is under-saturated until 36 clients. From this point on, the system is saturated. This is concluded from the observation that the throughput does not increase with a higher amount of clients. Furthermore Figure 11 shows nicely how until 36 clients the dominating factor of the middleware response time is the server service time and from 36 clients on the dominating factor is the queueing time. This shows that from 36 clients on the middlewares are not able to immediately process incoming requests and the queue starts to grow with more clients. For 16 worker threads the saturation phase starts at 72 clients, for 32 worker threads the saturation phase begins at 120 clients and for 64 worker threads per middleware the saturation phase begins with 192 clients. For all worker thread configuration the same explanation as for 8 worker threads holds, before the saturation phase is reached, the system is under-saturated. The beginning of the saturation phases align well with the ones observed in subsection 3.2. No over-saturation of the system has been observed during this experiment

4.2 Summary

This section is concluded by presenting the maximum throughput as well as the average queueing time, the average queue length and the average server service time measured at the maximum throughput configuration for each worker thread configuration in Table 15. The configuration which has been identified as the maximum throughput configuration is mentioned in the first row of each column.

When comparing the results of this experiment to the results obtained in subsection 3.2

we observe significantly lower maximum throughputs of up to 20%. The reason for this is the replication, for which the middleware has to send the request to 3 instead of 1 memcached servers and wait until it receives an answer from the slowest (or furthest away) memcached server. We have concluded that the increase in total server service time (all 3 servers together) is not due to different server ping times but comes from the overhead of sending the request 3 times instead of only once. Figure 11 shows a slightly higher total server service time compared to subsection 3.2 for all worker thread configurations.

We again observe that the queue time starts to increase roughly at the point where the number of clients is equal to the total number of worker threads in the system (which is 2 times the number of worker threads per middleware) for all worker thread configurations. Before this point there is one worker thread available per client and every request is immediately processed, therefore the queue time and size are almost zero. From this point on the system is saturated, while it is under-saturated before. We did not observe an over saturation during our experiments.

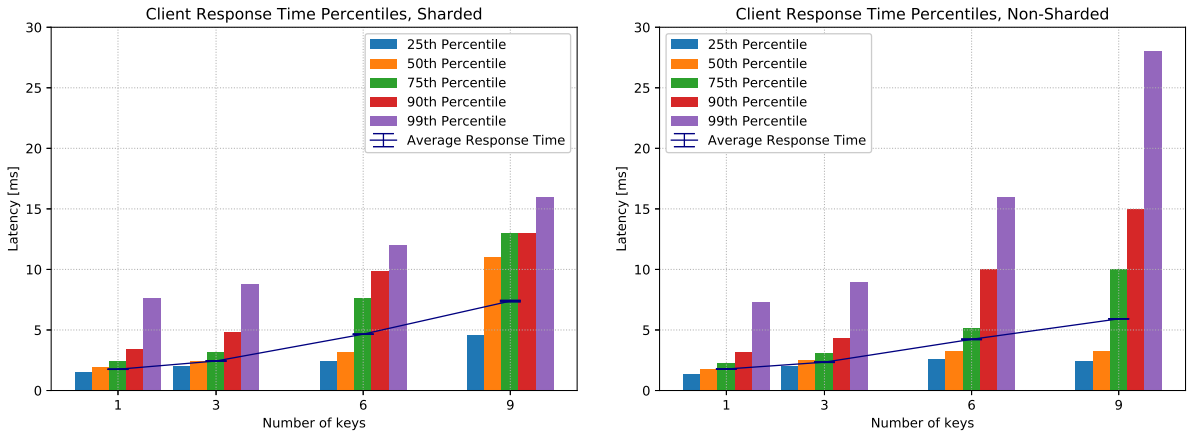
Compared to subsection 3.2 the servers are no longer the bottleneck for any of the worker thread configurations in this experiments. We concluded that for all worker thread configurations the middleware is the bottleneck and that the reason for this is the higher total server service time, which leads to a lower arrival rate at the memcached servers. This is because our system is closed and the clients need to wait for a response from the middleware before sending the next request.

	WT=8, 36 clients	WT=16, 72 clients	WT=32, 120 clients	WT=64, 192 clients
Throughput (MW) [req/s]	7110	9158	10817	12568
Throughput (Derived from MW RT) [req/s]	7045	9148	10810	12549
Throughput (Client) [req/s]	7108	9160	10813	12566
Avg. time in queue [ms]	1.87	3.39	4.09	3.93
Avg. length of queue	4.87	11.68	16.18	19.23
Avg. time waiting for memcached [ms]	2.21	3.46	5.87	10.09

Table 15: Maximum throughput for the full system in subsection 4.1

5 Gets and Multi-gets (90 pts)

This section analyzes the effect on throughput and response time of sharding MULTIGET requests. Table 16 shows the setup and different configurations that were examined for this section.



(a) Sharded

(b) Non-Sharded

Figure 12: Response time and percentiles measured on the client side, sections 5.1 and 5.2

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 and Non-Sharded
Multi-Get size	[1, 3, 6 9]
Number of middlewares	2
Worker threads per middleware	64
Repetitions	3

Table 16: Experiment configurations for subsection 5.1 and subsection 5.2

5.1 Sharded Case

The results in this section are based on the aggregated request logs from the two middlewares as well as the aggregated output data from the three memtier clients. The first and last three seconds of each of the three repetitions were cut off as startup and cooldown time, leaving 60 seconds of data per repetition. No misses occurred during the test and all client requests were processed and answered. All plots show averages and standard deviations across three repetitions of each configuration. The sanity of the data has been checked with the interactive law for both throughput and response time separately for all configurations and it was found that the interactive law aligns with the measured data. The configurations examined and the setup for this section are shown in Table 16. The configuration with 64 worker threads has been chosen because it was the one with the highest throughput in section 3 for write-only workload. Even though we are mainly interested in the MULTIGET workload in this section, 50% of the requests are SET requests, so in order to minimize the effect of the SET request on the performance of our system we chose to set the number of WT to 64.

5.1.1 Explanation

In the sharded mode, each GET request is split into 3 parts for 3 or more keys and requested from each server. With the same explanation as in subsection 4.1 we expect a higher server service time because the way the middleware is constructed it has to wait until it received a response from all servers. However, the overhead is much smaller, because unlike in the case of replication, the sum of the amount of data sent to the server remains almost, except for some overhead for the different TCP packages, the same because instead of 1 large request, each server now gets 3 small requests per multiget request. For the sharded mode we observe that the response time measured at the client remains almost the same for 1 and 3 keys. For 1,3,6,9 keys we measured a normalized average throughput of 3k, 7.5k, 8.7k and 8.8k requests per second respectively. These numbers are normalized, meaning the measured throughput for 3 keys has been multiplied by 3, for 6 keys by 6 and for 9 keys by 9. The configuration at hand is, with the only difference that there are 3 servers, the same as the one in subsection 3.2, where we measured a maximum throughput of almost 3k requests per second for read-only workload. For 3 servers we would expect the throughput to increase by a factor of 3 because the limiting factor in subsection 3.2 has been identified as the network upload capacity at the server. So the maximum throughput is expected to be around 8.8k requests per second. We notice that for 6 keys the maximum throughput of 8.8k requests per second is almost reached and for 9 keys it is reached. The aggregated dstat stats collected during this experiment confirm this. The average response time as well as the percentiles displayed in Figure 12 (a) remain almost the same for 1 and 3 keys because the system is still under-saturated. Also, because the middlewares operate in sharded mode, the servers have to handle only single key requests as the 3 key requests are split up into three single key requests. For 6 keys per request the system almost reaches its saturation phase, with the bottleneck being the server network upload capacity. The servers now have to handle 2 key requests because the middlewares split every 6-key-request into 3 requests with 2 keys each.

5.2 Non-sharded Case

The results in this section are based on the aggregated request logs from the two middlewares as well as the aggregated output data from the three memtier clients. The first and last three seconds of each of the three repetitions were cut off as startup and cooldown time, leaving 60 seconds of data per repetition. No misses occurred during the test and all client requests were processed and answered. All plots show averages and standard deviations across three repetitions of each configuration. The sanity of the data has been checked with the interactive law for both throughput and response time separately for all configurations and it was found that the interactive law aligns with the measured data. The configurations examined and the setup for this section are shown in Table 16. With the same explanation as in subsection 5.1 and because we want to be able to compare it to the sharded case, we also chose for the non-sharded case a configuration with 64 worker threads.

5.2.1 Explanation

In nonsharded mode, each middleware sends the whole (MULTI)GET request to one memcached server. Furthermore the middleware assures load balancing by iterating through the servers for each worker thread. For the non-sharded mode we observe that the response time measured at the client remains almost the same for 1 and 3 keys. For 1,3,6,9 keys we measured an average throughput of 3k, 7.6k, 8.7k and 8.9k requests per second respectively. These numbers have been normalized, meaning the measured throughput for 3 keys has been multiplied by 3, for 6

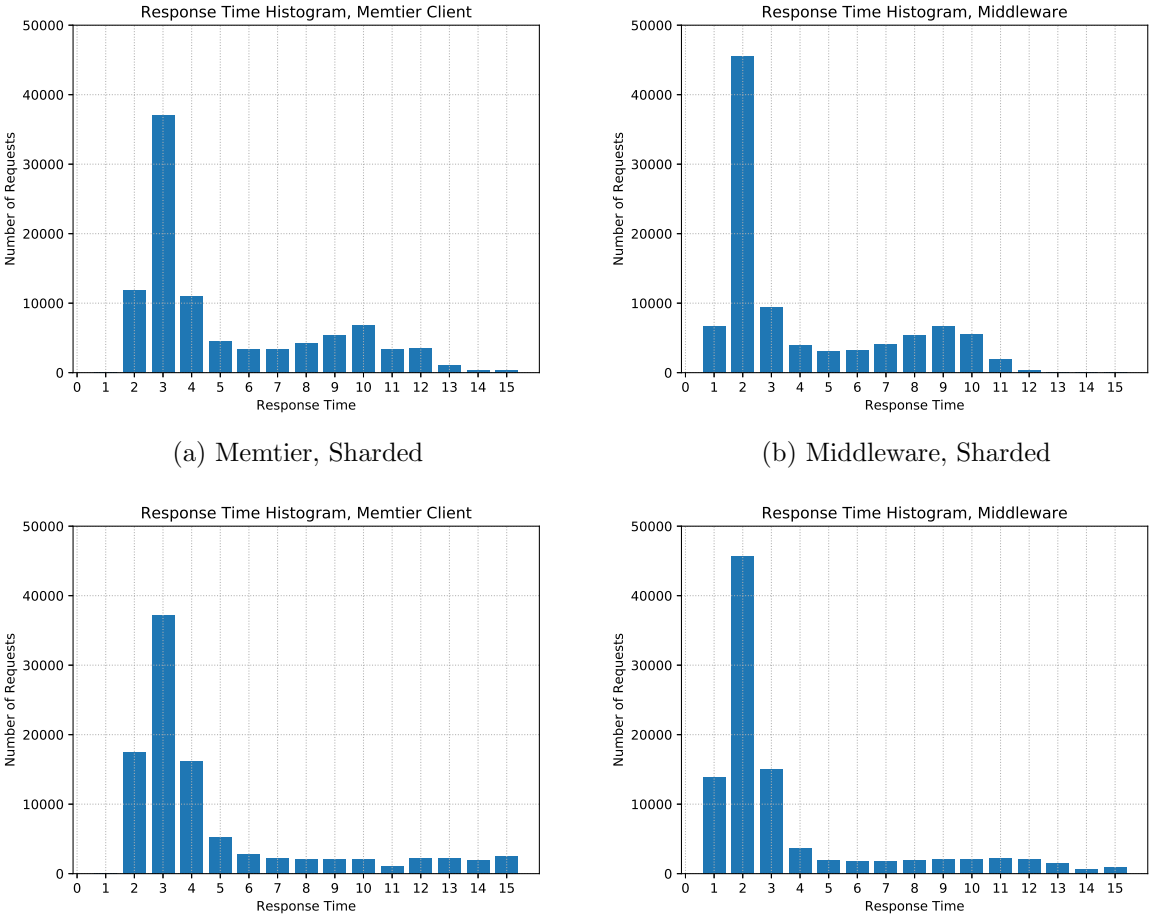
keys by 6 and for 9 keys by 9. Like in the sharded case, the configuration at hand is, with the only difference that there are 3 servers, the same as the one in subsection 3.2, where we measured a maximum throughput of 2935 requests per second for read-only workload. For 3 servers we expect the throughput to increase by a factor of 3 because the limiting factor in subsection 3.2 has been identified as the network upload capacity at the server. So the maximum reachable throughput is expected to be no higher than 8.8k requests per second. We notice that for 6 keys the maximum throughput of 8.8k requests per second is almost reached and for 9 keys it is reached. The aggregated dstat stats collected during this experiment confirm this.

The average response time as well as the percentiles displayed in Figure 12 (b) remain almost the same for 1 and 3 keys because the system is still under-saturated. We notice here that the response times for 1 and 3 keys do not differ significantly compared to the sharded case For 6 keys per request the system almost reaches its saturation phase, with the bottleneck being the server network upload capacity. Even though the servers have to handle 3 times larger keys compared to the sharded case, the average response time is almost the same. However, the 50th, 90th and 99th percentiles are much higher for the sharded case for both 6 and 9 keys. This indicates that the system is less stable for a higher number of keys in multiget requests. The reason for this could be the larger network packages due to the higher number of keys per request in the non-sharded mode.

5.3 Histogram

Figure 13 shows histograms for both memtier and middleware both for a sharded and a non-sharded configuration. The histograms were generated using the data from subsection 5.1 and subsection 5.2 for the configuration of memtier where 6 key MULTIGET requests were sent. As a sanity check the number of requests per bucket were summed up for both sharded and non-sharded, compared and found to be equal with very small differences due to rounding of the values gathered from the memtier output. For sharded and non-sharded independently we observe a shift from memtier to middleware, namely the histogram of the middleware is shifted 1 bucket to the right, meaning the response times measured at the client are around 1ms longer. This can be explained with the additional round-trip time of each request from client to middleware. The middleware measures its response time from the arrival of the request until before it is returned to the client, so we expect the response time to be 1 round-trip time off compared to the response time measured at the client. Looking at the histogram and the mean ping time of 1ms our hypothesis is confirmed.

Furthermore, for the sharded histograms we observe larger buckets for higher response times. This can be explained with reference to the way the middleware handles sharded multiget requests. For sharded multiget requests the middleware splits the request into equally sized parts (if, like in the case at hand, the number of keys is divisible by the number of servers), sends one part after another to each server and then waits for an answer from each server. Similar as in subsection 3.2 the server service time is now at least the service time of the slowest server for every request plus some overhead for sending 3 requests instead of only one while in the non-sharded case, in average, only every third request is sent to the slowest (or furthest) away memcached server and there is 3 times less requests in the system from the middleware to the servers. Different than in subsection 3.2 the ping times between middleware and servers vary more and we observe differences between the slowest and fastest server of up to 1ms in average ping time during the experiments of subsection 5.1 and subsection 5.2.



(a) Memtier, Sharded

(b) Middleware, Sharded

(c) Memtier, Non-Sharded

(d) Middleware, Non-Sharded

Figure 13: Response Time Histograms

5.4 Summary

The percentiles and response times plot displayed in Figure 12 suggests that in sharded mode we have significantly less outliers for 6 as well as for 9 keys, which means the system provides more often a similar response time in the sharded mode for these key sizes. For 1 and 3 keys we could not observe a difference between sharded and non-sharded mode and hence suggest to choose non-sharded mode for these key-sizes.

For sharded and non-sharded independently we observe a shift from memtier to middleware, namely the histogram of the middleware is shifted 1 bucket to the right, meaning the response times measured at the client are around 1ms longer. This is due to the additional round-trip time of each request from client to middleware. While the middleware measures its response time from the arrival of the request until before it is returned to the client, the client has in addition the round trip time from middleware to client in its response time. The histograms displayed in Figure 13 and the aggregated mean ping time between clients and middlewares of 1ms confirm our hypothesis.

6 2K Analysis (90 pts)

In this section, a 2k analysis for k=3 is being performed based on the theory provided in the exercise sessions [4]. With this, the impact of the number of memcached servers, the number of worker threads and the number of middlewares (as well as any combination of these) on throughput and response time is derived. The following parameter choices are being investigated:

- Worker threads per MW $x_a \in \{8, 32\}$
- Number of memcached servers $x_b \in \{1, 3\}$
- Number of middlewares $x_c \in \{1, 2\}$

For the 2k analysis the parameters will be set to -1 for the first choice and 1 for the second choice.

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

Table 17: Experiment configurations for subsection 6.1 and subsection 6.2

6.1 Read-only Workload

Table 18 shows parameters and combinations of them, measured throughput and response time, allocation of variance and derived effect sizes under a read-only workload. The additive 2k factorial design has been used because ,based on observations during experiments of the previous sections, for read-only workload no multiplicative effects are expected between the tested parameters. The results in this section are based on the aggregated request logs from the middleware(s). The first and last three seconds of each of the three repetitions were cut off as startup and cooldown time, leaving 60 seconds of data per repetition. As a sanity check, the same data has been extracted from the aggregated memtier client outputs, compared and found to align well. No misses occurred during the test and all client requests were processed and answered. For this section the error has been calculated as the sum of squared standard deviation errors over all 3 runs. The configurations examined and the setup for this section are shown in Table 17.

Table 18 (b) shows the allocation of variance per parameter for the throughput and we observe that x_b, the number of memcached servers, is with close to 100% the only dominating factor for the throughput of the system for read-only workload. In Table 18 (a) we see that q_b, the effect size of x_b, is positive, which leads us to the conclusion that more servers lead to a higher throughput. In section 2 we measured for read-only throughput an increase of factor 2 when having 2 instead of 1 servers. We also observed in section 2 that for read-only workload the outgoing network capacity of the memcached server is the bottleneck. Therefore we notice in this 2k analysis that for read-only workload the main factor which positively affects the

I	x_a	x_b	x_c	x_{ab}	x_{bc}	x_{ac}	x_{abc}	y_{mean}	y_{stddev}
1	-1	-1	-1	1	1	1	-1	2936.411	0.904
1	-1	-1	1	1	-1	-1	1	2941.454	3.843
1	-1	1	-1	-1	-1	1	1	8702.579	26.966
1	-1	1	1	-1	1	-1	-1	8803.022	3.089
1	1	-1	-1	-1	1	-1	1	2940.514	2.471
1	1	-1	1	-1	-1	1	-1	2942.596	0.192
1	1	1	-1	1	-1	-1	-1	8804.268	2.778
1	1	1	1	1	1	1	1	8797.934	24.191
q_0	q_a	q_b	q_c	q_{ab}	q_{bc}	q_{ac}	q_{abc}	-	-
5858.597	12.731	2918.354	12.654	11.419	10.873	-13.717	-12.977	/8	-

(a) Parameters and Effect Sizes, Throughput

q_a	q_b	q_c	q_{ab}	q_{bc}	q_{ac}	q_{abc}	Error
0.002%	99.989%	0.002%	0.002%	0.001%	0.002%	0.002%	0.000%

(b) Allocation of Variance, Throughput

I	x_a	x_b	x_c	x_{ab}	x_{bc}	x_{ac}	x_{abc}	y_{mean}	y_{stddev}
1	-1	-1	-1	1	1	1	-1	64.356	0.033
1	-1	-1	1	1	-1	-1	1	64.188	0.094
1	-1	1	-1	-1	-1	1	1	21.098	0.073
1	-1	1	1	-1	1	-1	-1	20.802	0.013
1	1	-1	-1	-1	1	-1	1	63.904	0.047
1	1	-1	1	-1	-1	1	-1	63.966	0.002
1	1	1	-1	1	-1	-1	-1	20.528	0.004
1	1	1	1	1	1	1	1	20.611	0.077
q_0	q_a	q_b	q_c	q_{ab}	q_{bc}	q_{ac}	q_{abc}	-	-
42.431	-0.179	-21.672	-0.040	-0.011	-0.013	0.076	0.019	/8	-

(c) Parameters and Effect Sizes, Response Time

q_a	q_b	q_c	q_{ab}	q_{bc}	q_{ac}	q_{abc}	Error
0.007%	99.991%	0.000%	0.000%	0.000%	0.001%	0.000%	0.001%

(d) Allocation of Variance, Response Time

Table 18: 2K Analysis, Read-only Workload

throughput is the number of servers. This is because each server leads to a throughput increase of roughly 3k requests per second. Furthermore we notice that throughput and response time must be in close relationship. We expect that factors which strongly influence throughput, inversely influence response time in an equally strong way. This is confirmed when looking at Table 18 (c) and (d) where we again observe the allocation of variance for x_b close to 100% but this time the effect size with a negative value.

6.2 Write-only Workload

Table 19 shows parameters and combinations of them, measured throughput and response time, allocation of variance and derived effect sizes under a write-only workload. The additive 2k factorial design has been used because only between x_a, the number of worker threads, and x_c, the number of middlewares we expect a multiplicative effect. This is because having 2 middlewares will double the total amount of worker threads in the system as we have observed

in subsection 3.2, therefore having 32 worker threads and 2 middlewares will actually result in 64 worker threads in total and we have observed in sections 3 and 4 that for write-only workload the middleware is the bottleneck until 128 worker threads in total.

Table 19 shows that for write-only workload x_a, the number of worker threads has the most significant effect on the system followed by the number of middlewares x_c. This aligns with the observations in sections 3 and 4 for write-only workload where we observed that the middleware is the bottleneck due to too few worker threads. The reason for x_a having a significantly higher allocation of variance in comparison to x_c is, that the number of worker threads is increased by a factor of 4 with x_a while it is only doubled with x_c. As mentioned before we would expect here a multiplicative effect, if we have 32 worker threads and 2 middlewares but this effect is not shown by our 2k analysis because we chose to use the additive model. The number of memcached servers, x_b, has the lowest effect for write-only workload and the effect on the throughput is negative, meaning increasing the amount of servers from 1 to 3 actually leads to a decrease of throughput and an increase of latency. As we have seen in subsection 4.1, the decrease of throughput for write-only workload can be explained with reference to the implementation of the middleware. The middleware sends SET requests to each memcached server and responses are received from each server one by one. Besides the sending and receiving, the main difference in response time comes from the dependence on the slowest server response time.

As in subsection 6.1 we also expect here that factors which strongly influence throughput, inversely influence response time in an equally strong way. This is confirmed when looking at Table 19 (c) and (d) where we again observe similar allocation of variances for for all 3 variables but with negated effect sizes compared to the throughput effect sizes.

7 Queuing Model (90 pts)

Note that for queuing models it is enough to use the experimental results from the previous sections. It is, however, possible that the numbers you need are not only the ones in the figures we asked for, but also the internal measurements that you have obtained through instrumentation of your middleware.

7.1 M/M/1

In this section we build a queueing model for each worker thread configuration of the middleware based on subsection 4.1. We use an M/M/1 model based on the formulas given in [1] to model the entire system. This model assumes a single server (in our case the middleware) with an infinite buffer (in our case queue) with first-come-first-serve policy. As input parameters our model takes the service rate μ [requests/second] and the arrival rate λ [requests/second]. The service rate μ we approximate by taking the maximum measured throughput over all runs of the tested number of clients of subsection 4.1. The arrival rate λ is set to the average throughput measured in subsection 4.1 for a given amount of clients, this assumption is made because we are in a closed system. With the model, we predict the expected response time $E[r]$, the expected queue size $E[n_q]$ and the expected queue waiting time $E[w]$.

Table 20 shows the predicted values of these parameters for 8, 16, 32 and 64 worker threads respectively and for the same number of clients as in subsection 4.1. The predicted values were obtained using the described M/M/1 model and the observed values are from subsection 4.1. Figure 14 shows the predicted response times next to the observed response times. We observe that the predicted response times are too low for 6, 18 and 36 clients and too high for 192 clients. We observe that even thought the predicted values are not close to the measured ones

I	x_a	x_b	x_c	x_{ab}	x_{bc}	x_{ac}	x_{abc}	y_{mean}	y_{stddev}
1	-1	-1	-1	1	1	1	-1	6820.136	16.917
1	-1	-1	1	1	-1	-1	1	8557.913	38.062
1	-1	1	-1	-1	-1	1	1	5333.355	174.900
1	-1	1	1	-1	1	-1	-1	7240.880	35.038
1	1	-1	-1	-1	1	-1	1	10573.918	23.666
1	1	-1	1	-1	-1	1	-1	13065.804	12.610
1	1	1	-1	1	-1	-1	-1	8110.258	15.192
1	1	1	1	1	1	1	1	10789.416	59.709
q_0	q_a	q_b	q_c	q_{ab}	q_{bc}	q_{ac}	q_{abc}	-	-
8811.460	1823.389	-942.983	1102.043	-242.029	44.627	190.718	2.190	/8	-

(a) Parameters and Effect Sizes, Throughput

q_a	q_b	q_c	q_{ab}	q_{bc}	q_{ac}	q_{abc}	Error
60.172%	16.093%	21.980%	1.060%	0.036%	0.658%	0.000%	0.000%

(b) Allocation of Variance, Throughput

I	x_a	x_b	x_c	x_{ab}	x_{bc}	x_{ac}	x_{abc}	y_{mean}	y_{stddev}
1	-1	-1	-1	1	1	1	-1	27.219	0.061
1	-1	-1	1	1	-1	-1	1	21.378	0.045
1	-1	1	-1	-1	-1	1	1	35.060	1.193
1	-1	1	1	-1	1	-1	-1	25.521	0.140
1	1	-1	-1	-1	1	-1	1	16.914	0.032
1	1	-1	1	-1	-1	1	-1	13.575	0.050
1	1	1	-1	1	-1	-1	-1	21.570	0.091
1	1	1	1	1	1	1	1	16.604	0.092
q_0	q_a	q_b	q_c	q_{ab}	q_{bc}	q_{ac}	q_{abc}	-	-
22.230	-5.064	2.459	-2.961	-0.537	-0.666	0.885	0.259	/8	-

(c) Parameters and Effect Sizes, Response Time

q_a	q_b	q_c	q_{ab}	q_{bc}	q_{ac}	q_{abc}	Error
60.745%	14.316%	20.760%	0.684%	1.049%	1.853%	0.159%	0.435%

(d) Allocation of Variance, Response Time

Table 19: 2K Analysis, Write-only Workload

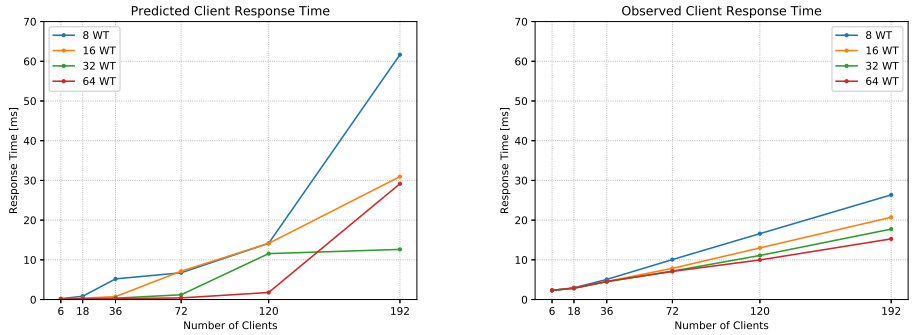


Figure 14: M/M/1 predicted and measured Response Times, Full System, subsection 7.1

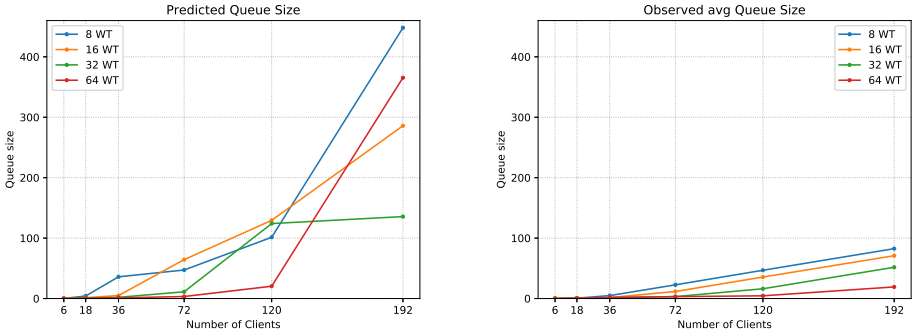


Figure 15: M/M/1 predicted and measured Queue Size, Full System, subsection 7.1

we see a trend that the response time for 64 worker threads is lower than for 8, 16 and 32 worker threads.

This model is too simplistic to model our system accurately because it assumes that there is only a single queue and a single worker. Furthermore it cannot model a closed system which we can see at the queue size which increases higher than the number of clients, in a closed system like ours this is not possible because every client waits for a response before sending the next request.

7.2 M/M/m

In this section we build a queueing model for each worker thread configuration of the middleware based on subsection 4.1. We use an M/M/m model based on the formulas given in [1] to model the entire system. Each worker thread in our system is seen as a server and therefore the number of server m in our model is equal to the number of worker threads inside the middleware. Except for this, the input parameters are the same as for the M/M/1 model described in subsection 7.1, namely takes the service rate μ [requests/second] and the arrival rate λ [requests/second]. The arrival rate λ is set again to the average throughput measured in subsection 4.1 for a given amount of clients, this assumption is made because we are in a closed system and hence a client only sends a request after it got the response from the previous request. The service rate μ is set to the maximum measured throughput over all runs of the tested number of clients in subsection 4.1 divided by m (number of servers in our model = number of worker threads in the system). With the model, we predict the expected response time $E[r]$, the expected queue size $E[n_q]$ and the expected queue waiting time $E[w]$.

Table 21 shows the predicted values of these parameters for 8, 16, 32 and 64 worker threads

# Clients	λ	ρ	$E[r]$	$E[n_q]$	$E[w]$
6	2571.91	0.35	0.21 // 2.33	0.19 // 0.11	0.07 // 0.07
18	6093.61	0.83	0.83 // 2.95	4.22 // 0.62	0.69 // 0.20
36	7107.66	0.97	5.20 // 5.06	35.98 // 4.87	5.06 // 1.87
72	7151.63	0.98	6.74 // 10.06	47.22 // 22.71	6.60 // 6.86
120	7229.38	0.99	14.16 // 16.59	101.38 // 46.78	14.02 // 13.44
192	7283.78	1.00	61.66 // 26.35	448.10 // 82.55	61.52 // 23.17

(a) 8 Worker Threads, $\mu = 7300 ops/s$

# Clients	λ	ρ	$E[r]$	$E[n_q]$	$E[w]$
6	2577.76	0.28	0.15 // 2.32	0.11 // 0.11	0.04 // 0.07
18	6330.40	0.68	0.34 // 2.84	1.45 // 0.70	0.23 // 0.12
36	7930.72	0.85	0.73 // 4.53	4.94 // 1.58	0.62 // 0.38
72	9159.92	0.98	7.14 // 7.85	64.40 // 11.68	7.03 // 3.39
120	9229.22	0.99	14.13 // 13.00	129.41 // 35.57	14.02 // 8.55
192	9267.70	1.00	30.96 // 20.71	285.92 // 70.90	30.85 // 16.13

(b) 16 Worker Threads, $\mu = 9300 ops/s$

# Clients	λ	ρ	$E[r]$	$E[n_q]$	$E[w]$
6	2596.04	0.24	0.12 // 2.31	0.07 // 0.10	0.03 // 0.07
18	6274.84	0.58	0.22 // 2.86	0.78 // 0.66	0.12 // 0.12
36	7991.58	0.73	0.34 // 4.50	2.01 // 1.67	0.25 // 0.21
72	10070.24	0.92	1.21 // 7.14	11.21 // 3.22	1.11 // 0.65
120	10813.54	0.99	11.57 // 11.09	124.08 // 16.18	11.47 // 4.09
192	10820.80	0.99	12.63 // 17.74	135.63 // 51.73	12.53 // 10.68

(c) 32 Worker Threads, $\mu = 10900 ops/s$

# Clients	λ	ρ	$E[r]$	$E[n_q]$	$E[w]$
6	2611.63	0.21	0.10 // 2.29	0.05 // 0.10	0.02 // 0.07
18	6262.69	0.50	0.16 // 2.87	0.49 // 0.60	0.08 // 0.12
36	7933.14	0.63	0.21 // 4.53	1.07 // 1.62	0.13 // 0.21
72	10147.69	0.81	0.41 // 7.09	3.33 // 3.13	0.33 // 0.34
120	12036.31	0.96	1.77 // 9.96	20.40 // 4.52	1.69 // 0.46
192	12565.70	1.00	29.16 // 15.27	365.40 // 19.24	29.08 // 3.93

(d) 64 Worker Threads, $\mu = 12600 ops/s$

Table 20: System Properties, M/M/1 Model. Please note that in the three rightmost columns the values are displayed as *predicted//observed*. $E[r]$ is the response time in ms, $E[n_q]$ displays the number of requests in the queue and $E[w]$ displays is the queue waiting time in ms.

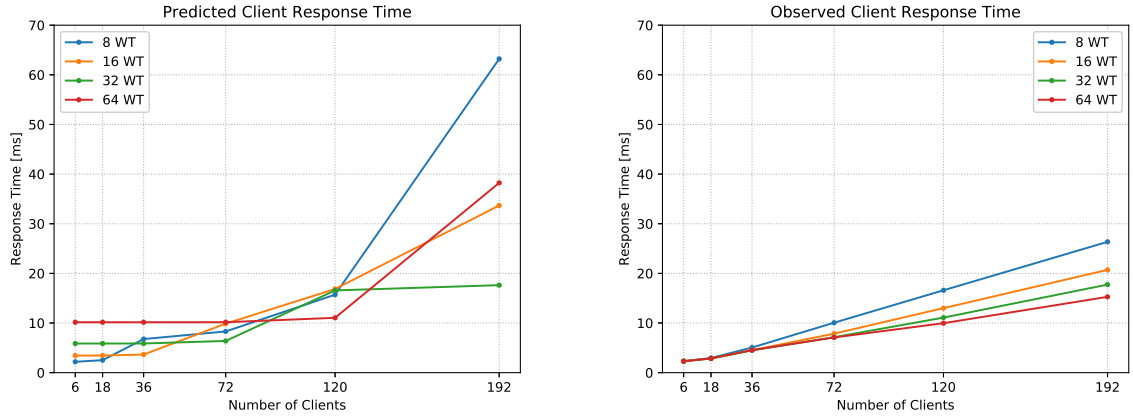


Figure 16: M/M/m predicted and measured Response Times, Full System, subsection 7.2

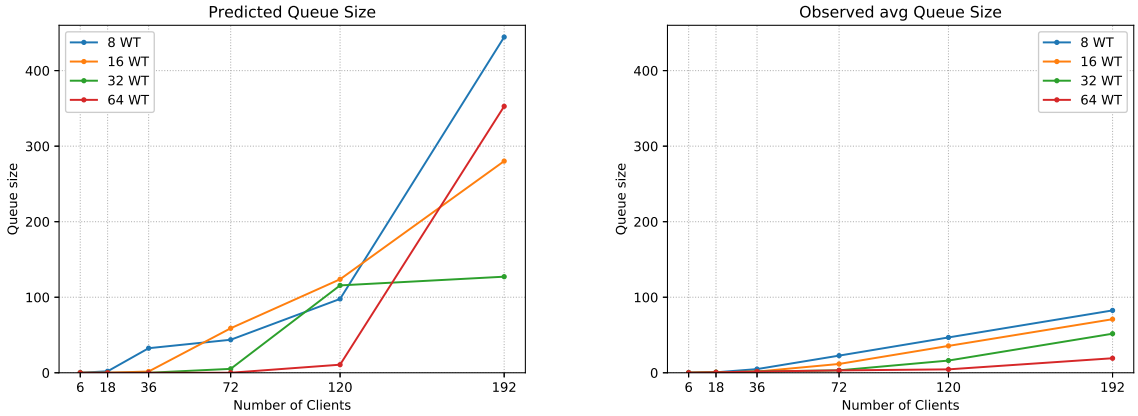


Figure 17: M/M/m predicted and measured Queue Size, Full System, subsection 7.2

respectively and for the same number of clients as in subsection 4.1. The predicted values were obtained using the described M/M/m model and the observed values are the ones measured during the experiments of subsection 4.1. Figure 16 shows the predicted response times next to the observed response times. We observe that compared to the M/M/1 model the response times are now higher for 6, 18 and 36 clients, however they are still not close to the measured values.

This model models our system a bit better than the M/M/1 model because it takes the number of worker threads as an input parameter. However, also this model is not able to model a closed system and also it cannot model the situation at hand where we have 2 queues because of the two middlewares.

7.3 Network of Queues

Using a network of queues we are able to model our system more accurately. In this section we will explore the setup of 1 middleware and 2 middlewares as well as read-only and write-only workload. We will compare the results to the measurements of subsection 3.1 and subsection 3.2. The network of queues allows us to use load dependent service times and to model a closed system, which should lead to more accurate predictions.

# Clients	λ	ρ	$E[r]$	$E[n_q]$	$E[w]$
6	2571.91	0.35	2.19 // 2.33	0.00 // 0.11	0.00 // 0.07
18	6093.61	0.83	2.52 // 2.95	1.97 // 0.62	0.00 // 0.20
36	7107.66	0.97	6.77 // 5.06	32.53 // 4.87	0.00 // 1.87
72	7151.63	0.98	8.30 // 10.06	43.72 // 22.71	0.01 // 6.86
120	7229.38	0.99	15.72 // 16.59	97.77 // 46.78	0.01 // 13.44
192	7283.78	1.00	63.21 // 26.35	444.42 // 82.55	0.06 // 23.17

(a) 8 Worker Threads, $\mu = 7300/(2 * 8) = 456 ops/s$

# Clients	λ	ρ	$E[r]$	$E[n_q]$	$E[w]$
6	2577.76	0.28	3.44 // 2.32	0.00 // 0.11	0.00 // 0.07
18	6330.40	0.68	3.45 // 2.84	0.06 // 0.70	0.00 // 0.12
36	7930.72	0.85	3.65 // 4.53	1.68 // 1.58	0.00 // 0.38
72	9159.92	0.98	9.87 // 7.85	58.92 // 11.68	0.01 // 3.39
120	9229.22	0.99	16.85 // 13.00	123.78 // 35.57	0.01 // 8.55
192	9267.70	1.00	33.68 // 20.71	280.22 // 70.90	0.03 // 16.13

(b) 16 Worker Threads, $\mu = 9300/(2 * 16) = 291 ops/s$

# Clients	λ	ρ	$E[r]$	$E[n_q]$	$E[w]$
6	2596.04	0.24	5.87 // 2.31	0.00 // 0.10	0.00 // 0.07
18	6274.84	0.58	5.87 // 2.86	0.00 // 0.66	0.00 // 0.12
36	7991.58	0.73	5.88 // 4.50	0.03 // 1.67	0.00 // 0.21
72	10070.24	0.92	6.39 // 7.14	5.19 // 3.22	0.00 // 0.65
120	10813.54	0.99	16.57 // 11.09	115.68 // 16.18	0.01 // 4.09
192	10820.80	0.99	17.63 // 17.74	127.20 // 51.73	0.01 // 10.68

(c) 32 Worker Threads, $\mu = 10900/(2 * 32) = 170 ops/s$

# Clients	λ	ρ	$E[r]$	$E[n_q]$	$E[w]$
6	2611.63	0.21	10.16 // 2.29	0.00 // 0.10	0.00 // 0.07
18	6262.69	0.50	10.16 // 2.87	0.00 // 0.60	0.00 // 0.12
36	7933.14	0.63	10.16 // 4.53	0.00 // 1.62	0.00 // 0.21
72	10147.69	0.81	10.16 // 7.09	0.05 // 3.13	0.00 // 0.34
120	12036.31	0.96	11.05 // 9.96	10.71 // 4.52	0.00 // 0.46
192	12565.70	1.00	38.23 // 15.27	352.76 // 19.24	0.03 // 3.93

(d) 64 Worker Threads, $\mu = 12600/(2 * 64) = 98 ops/s$

Table 21: System Properties, M/M/m Model. Please note that in the three rightmost columns the values are displayed as *predicted//observed*. $E[r]$ is the response time in ms, $E[n_q]$ displays the number of requests in the queue and $E[w]$ displays is the queue waiting time in ms.

7.3.1 One Middleware

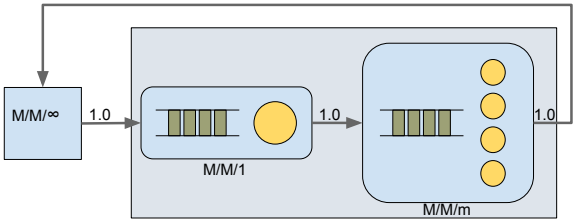


Figure 18: Network of Queues for 1 Middleware

Figure 18 shows our network of queues model for 1 middleware. We model the network latency as an $M/M/\infty$ center, the network thread as an M/M/1 center and the worker threads together with the servers as an M/M/m center with m equal to the number of worker threads in the system. We decided to model the worker threads together with the servers because we observed in sections 3.1, 3.2 and 4 that the response time is dominated by the server service time as long as the system is under-saturated. In the saturated phase the queueing time starts to become the dominating part in the middleware response time, for the network of queues model we only need to input the server service time, because the queueing time will be predicted by the model. Furthermore we observed that the server service time is the same for every worker thread (given one specific number of worker threads), which means that the model as described in Figure 18 should be able to model our system up to a reasonable accuracy. To predict the values, Octave [3] has been used with an extension for queueing models from which we used the implementation for closed networks [2, ch. 5.2.2]. The parameters m_i defines the number of servers per center. As described above we set it to 0 for the $M/M/\infty$ center, to 1 for the M/M/1 center and to 'number of worker threads' for the M/M/m center. The μ_i parameters define the service time per center. For the $M/M/\infty$ center we set μ_i to 0.001s which is approximately the average network roundtrip time between clients and middlewares we measured during the experiments in section 3. For the M/M/1 center we set μ_i to 0.00001s which is approximately the average time a request spent in the network thread in the middleware. For the M/M/m model we used a load dependent service time which we set to the average server service time measured in subsection 3.1.

Table 22 (a) shows the measured and predicted values for the network of queues for write-only workload. The number of worker threads in the M/M/m center in our model was set to 8 worker threads because this was the worker thread configuration which achieved the highest throughput in subsection 3.1. While the model predicts the throughput quite accurately for 6 and 18 clients, its predictions are too high for 36 and 72 clients. For the response time we observe the best prediction for 36 clients. From the utilization we see that the utilization of the M/M/m center is predicted to be 96% for 72 clients. This is close to the measured beginning of the saturation phase in subsection 3.1 of 120 clients. The way our model is built we can only conclude that the bottleneck must be at the worker/server interaction.

Table 22 (b) shows the measured and predicted values for the network of queues for read-only workload. The number of worker threads in the M/M/m center in our model was set to 64 worker threads because this was the worker thread configuration which achieved the highest throughput in subsection 3.1. For read-only workload the model predicts the throughput very accurately for all client configurations. Also the response time prediction is close to the measured values, only off by a constant factor for all clients. From the utilization we see that the utilization of the M/M/m center is predicted to be 97% already for 16 clients. This is close to the measured beginning of the saturation phase in subsection 3.1 of 6 clients. The way our model is built we

can only conclude that the bottleneck must be at the worker/server interaction.

The predicted queue length is too high for both types of workload and all clients. We conclude that while the model predicts throughput and response time very well for read-only workload, it fails to do so for write-only workload and for the queue length for any type of workload. Our model could be further extended on the client as well as on the server side, which might improve the prediction accuracy.

7.3.2 Two Middlewares

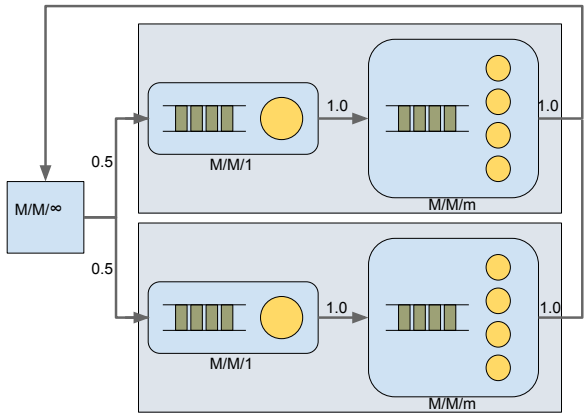


Figure 19: Network of Queues for 2 Middlewars

Figure 19 shows our network of queues model for 2 middlewares. The only difference compared to the queueing model with 1 middleware is, that there are now 2 M/M/1 centers and 2 M/M/m centers which correspond to 1 middleware each. The parameters m_i define the number of servers per center. We again set m_i to 0 for the $M/M/\infty$ center, to 1 for the M/M/1 center and to 'number of worker threads' for the M/M/m centers. The μ_i parameters define the service time per center. For the $M/M/\infty$ center we set μ_i to 0.001s which is approximately the average network roundtrip time between clients and middlewares that were measured during the experiments in section 3. For the M/M/1 center we set μ_i to 0.00001s which is approximately the average time a request spent in the network thread in the middleware. For the M/M/m model we used a load dependent service time which we set to the measured average server service time in subsection 3.2.

Table 22 (c) shows the measured and predicted values for the network of queues for write-only workload. The number of worker threads in the M/M/m center in our model was set to 8 worker threads because this was the worker thread configuration which achieved the highest throughput in subsection 3.2. For write-only workload Table 22 (c) shows that the most accurate throughput prediction is achieved for 36 clients. For 6 and 18 clients the prediction is too low while for 72 clients it is too high. The same goes for the response time but inversely, meaning it is predicted too high for 6 and 18 clients and too low for 72 clients. From the utilization we see that the utilization of the M/M/m center is predicted to be 84% for 72 clients. This is quite far from the measured beginning of the saturation phase in subsection 3.2 of 196 clients. The way our model is built we can only conclude that the bottleneck must be at the worker/server interaction.

Table 22 (d) shows the measured and predicted values for the network of queues for read-only workload. The number of worker threads in the M/M/m center in our model was set to 64 worker threads because this was the worker thread configuration which achieved the highest

# Clients	6		18		36		72	
	pred.	**meas.**	**pred.**	**meas.**	**pred.**	**meas.**	**pred.**	**meas.**
Throughput [req/s]	2985	2973	5979	6425	8975	6356	11937.5	6252
Resp. Time [ms]	2	1.1	3	1.9	4	4.7	6	10.6
Queue Len. [#req]	3	0.4	12	1.7	26.9	5.9	59.9	15.0
Utilization [%]	0.05	-	0.18	-	0.46	-	0.93	-

(a) 1 Middleware, Write-Only, 64 Worker Threads per Middleware

# Clients	6		12		18		36	
	pred.	**meas.**	**pred.**	**meas.**	**pred.**	**meas.**	**pred.**	**meas.**
Throughput [req/s]	2985	2909	2606	2939	2667	2936	2667	2937
Resp. Time [ms]	2	1.2	4.6	3.2	6.7	5.2	13.5	11.2
Queue Len. [#req]	2	0.5	4.6	1.0	6.7	4.3	13.5	22.1
Utilization [%]	0.37	-	0.97	-	1	-	1	-

(b) 1 Middleware, Read-Only, 8 Worker Threads per Middleware

# Clients	6		18		36		72	
	pred.	**meas.**	**pred.**	**meas.**	**pred.**	**meas.**	**pred.**	**meas.**
Throughput [req/s]	1993	3047	5980	7260	8977	8326	11978.8	8398
Resp. Time [ms]	3	1.1	3	1.6	4	3.4	6	7.6
Queue Len. [#req]	1.5	0.1	4.5	0.7	7.6	1.8	10.2	3.1
Utilization [%]	0.03	-	0.09	-	0.21	-	0.84	-

(c) 2 Middlewares, Write-Only, 64 Worker Threads per Middleware

# Clients	6		12		18		36	
	pred.	**meas.**	**pred.**	**meas.**	**pred.**	**meas.**	**pred.**	**meas.**
Throughput [req/s]	2985	2927	2986	2936	2699	2936	3068	2939
Resp. Time [ms]	2	1.1	4	3.1	6.6	5.2	11.7	11.2
Queue Len. [#req]	1.5	0.2	4.5	0.8	7.6	1.1	16.5	4.8
Utilization [%]	0.18	-	0.55	-	0.84	-	0.95	-

(d) 2 Middlewares, Read-Only, 8 Worker Threads per Middleware

Table 22: Predicted values for Network of Queues Model

throughput in subsection 3.2. For read-only workload Table 22 (d) shows good predictions for the throughput and the response times. From the utilization we see that the utilization of the M/M/m center is predicted to be 84% for 18 clients. This is quite close to the measured beginning of the saturation phase in subsection 3.2 of 6 clients. The way our model is built we can only conclude that the bottleneck must be at the worker/server interaction.

The predicted queue length is again too high for both types of workload and all clients. Also for the model with two middlewares we conclude that while it offers a good base, it is unable to predict write-only throughput accurately. For read-only workload on the other hand the predictions are good, except for the queue length which is predicted too high. Same as in the one middleware model, the model at hand could be extended on the client as well as on the server side, which might improve the prediction accuracy.

References

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

[2] Moreno Marzolla, *A Queueing Package for GNU Octave*, https://www.moreno.marzolla.name/software/queueing/queueing.html

[3] GNU, *Octave: A scientific programming language*, https://www.gnu.org/software/octave/

[4] ASL exercise session 7, 2018, *2k Factorial Design*, https://www.systems.ethz.ch/sites/default/files/07-2k-analysis.pdf