Advanced Systems Lab Report

Autumn Semester 2018

Name: Tobias Krebs
Legi: 12-933-172

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

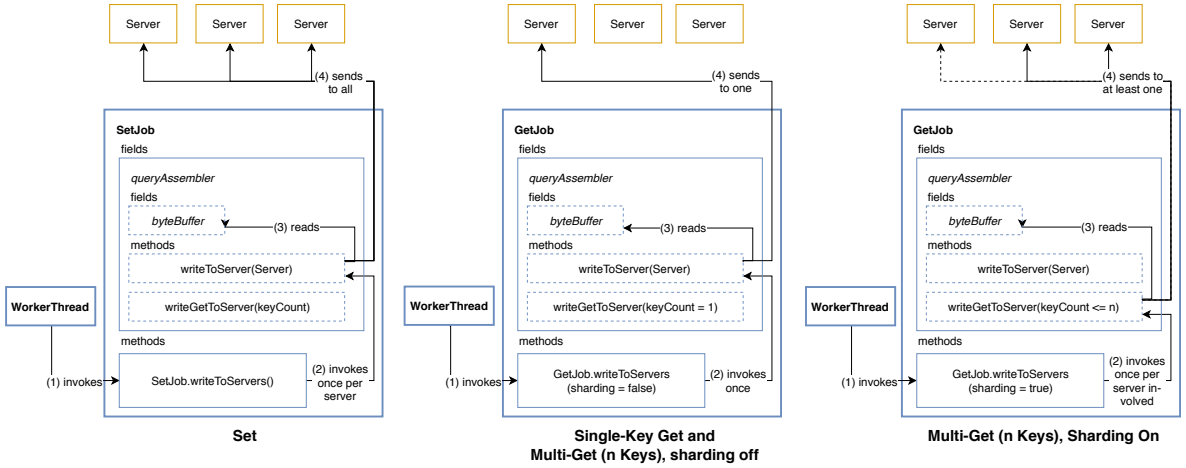


Figure 1: Sequence of events when a `WorkerThread` writes a freshly parsed `Job` to the server(s). Depicted are all four possible sequences, sets (left), single-key gets and multi-key gets with sharding disabled (middle) and multi-key gets with sharding enabled (right).

1 System Overview (75 pts)

1.1 Overview and Basic Architecture

The middleware consists of a multitude of components. The following subsections will highlight individual classes and explain their purpose and important implementation specifics, wherever they account for relevant design decisions. Figure 2 shows a high-level overview over the system. There are no other threads involved in the system other than the `NetThread` and the specified amount of `WorkerThreads`.

NetThread The `NetThread` uses Java's asynchronous `NIO` API to efficiently handle the big amount of connections held. Once a client sends a packet, the `NetThread` will parse it using a `QueryAssembler`. There is exactly one `QueryAssembler` for each client connection to enable the middleware holding parsing states over many connections at the same time. This allows parsing messages in chunks, since messages can be split into multiple packets.

At the time of creating a new `QueryAssembler`, the `QueryAssembler` is given a callback method as an argument. This callback method will be invoked when the `QueryAssembler` has put together a full query and will pass it as a newly created `Job` object. This may either be a `SetJob` or a `GetJob`. The callback method will take this new `Job` and put it into the `JobQueue` 1.1 to be processed by a `WorkerThread` 1.1.

JobQueue The `JobQueue` is a wrapper around a standard Java `ArrayBlockingQueue`. This synchronous queue blocks `WorkerThreads` while they poll `Jobs` to have them wait efficiently. It guarantees FIFO (first-in-first-out) ordering to ensure that queries that wait the longest will be served first (to minimize response time). An `ArrayBlockingQueue` has been chosen over an `LinkedBlockingQueue` to ensure insertion and polling will be fast. Even though a `LinkedBlockingQueue` takes constant insertion and polling time, it is faster to use an `ArrayBlockingQueue`, if the backing array used doesn't have to be extended. This can be prevented by initializing it with a high enough capacity for all experiments.

WorkerThread The middleware will create a `WorkerPool` after starting, which will then instantiate all `WorkerThreads`. Upon creation, the `WorkerThread` instance will create one

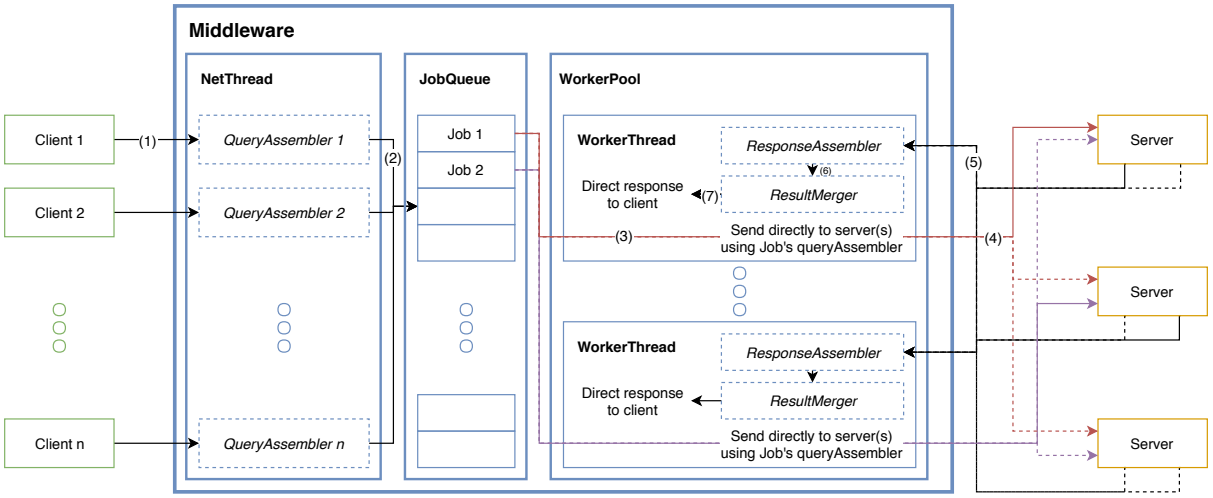


Figure 2: High-level overview over the system. The bracketed numbers follow a query as it is parsed*(1)*, en-*(2)* and dequeued*(3)*, sent to the server(s)*(4)*, has its result(s) parsed*(5)*, merged*(6)* and sent back to the client*(7)*. Note that set operations and non-sharded gets will only use one server (dotted arrows are used for sets and sharded gets only).

connection to each server and store them in an `OffsetList`. A `WorkerThread`'s loop polls the `JobQueue` for a `Job`, and will discern wether it is a `SetJob` or a `GetJob` object. The `SetJob` and `GetJob` classes both have a `writeToServers()` method that will instruct the `Job`'s `QueryAssembler` to send the request to the server(s), depending on job type and sharding. The `QueryAssembler` uses two methods: The first one, `writeToServer` will write the entire Query to a given server. This method is used for sets and non-sharded gets and is either called once for each server (to have the data replicated on all servers in a set) or only for one server (in case of a non-sharded get). The second one, `writeGetToServer` is only used in multi-gets and will construct and send a valid request to the given server and amount of keys. Also see figure 1, which illustrates this process. How the load distribution is managed and how a server is selected in case only one is used is explained in 1.1.

Analogous to how the `NetThread` holds one `QueryAssembler` per client connection, each `WorkerThread` holds one `ResponseAssembler` per connected server to parse incoming responses (possibly split up) individually. The `ResponseAssembler` will create a `Result` object for the completely parsed server response, which will invoke a callback in the `WorkerThread` to have all `Results` collected in a `ResultMerger` (1.1). A `Result` may be of type `StoredResult`, `ValueResult`, `ErrorResult`, `ClientErrorResult` or `ServerErrorResult`, corresponding to possible responses. The `ResponseAssembler` that creates a `Result` after parsing will bind itself to it in order to allow the `ResultMerger` to later access all collected `Results`' data and subsequently write the merged response back to the client (and to minimize data copying).

Assemblers (parsing) An `Assembler` holds an internal `ByteBuffer` that is used to hold the data read from a `SocketChannel` (non-blocking connections to clients in `QueryAssemblers`) or `Socket` (blocking connections to servers in `ResponseAssemblers`). This buffer will be the only storage location for received queries and responses to minimize data copying.

The `Assembler` parsing works for both `Assemblers` as follows: The data source will use the appropriate `Assembler`'s read function to have it copy the new data directly from the network buffer to its own `ByteBuffer`. Once the `Assembler`'s `advance()` method is called, the `Assembler` will begin (or resume) parsing the message. It will do so by iterating through the

new data byte by byte, until it either finishes parsing or the data abruptly ends.

In more detail, the parsing state is saved in a variable that is of type `ParserState`, which is an enum with exactly one symbol for each state the parsing may be in. The `Assemblers` have one byte array for each possible command (such as "`set`" or "`get`" in a `QueryAssembler`) or reply, and for each byte in such an array, the `ParserState` has one state in the form of `WORD_i_j`,x which means that the net expected byte is the j-th byte of the i-th word of the command/reply. For example `VALUE_0_2` expects the next byte to have the same value as is stored in the byte array `VALUE_COMPARABLE[2]`, which is ascii code for `L`. If the current byte matches, the state will be increased by one, setting the state to `VALUE_0_3`, else, the invalid message will be put together in a string and output on the command line. Whenever the message is unclear from just the first byte (e.g. `END` and `ERROR` both start with `E`), the parser has special states that will divert once more data is read. For commands and replies containing words with variable length (such as a `set`'s key), one state will be used until a whitespace character is encountered. Then, the enum will be increased by one, expecting the next word (e.g. transitioning from `SET_1`, which read the `key` word, to `SET_2`, which will read the `flags` word). For parts of the message which have known lengths (such as a datablock following a `set` command), the length was parsed beforehand and stored in a variable called `dataBytesRemaining` to keep track of the exact number of bytes remaining. When reading the word with the known length, the parsing will advance as many bytes as possible (meaning either jumping to the end of the word or, in case the word is cut off by abruptly ending data, the last byte available to enable resuming parsing later) and adjust `dataBytesRemaining` appropriately. In all cases, the `Assembler` will collect all relevant data during parsing, such as how many values are contained in a response to a multi-get and what the positions and lengths of each of them are in the `ByteBuffer`. Once the parsing is done and a full request or response is parsed, it will use this data to create a new object representing the message and invoke the callback given at creation.

ResultMerger The `ResultMerger` merges all parsed `Results` a `WorkerThread` gathers after having received the `Server(s)` response(s). In case of an unsharded get, merging is trivial. For a replicated set or a sharded get (where we expect multiple responses), the `ResultMerger` will check if an error occured in any of the `Results` and if all responses have the same type. It implements a method `writeToClient` that will act as a wrapper over all collected `Result`'s `ByteBuffers` and automatically sequentially send all data present in the `ByteBuffers` containing the responses collected. Together in sequence, they produce a valid response. In case at least one error occurred, the merger's `writeToClient` method will send any error encountered back to the client.

OffsetList The `OffsetList` is a wrapper over a standard Java `ArrayList`. Its purpose is to allow iterating over the list starting from an arbitrary element, looping around if required. By first setting an offset, iteration starts at that index and puts out all items in order. An example with an `OffsetList` of type `OffsetList<ServerConnection>` containing the elements `Server_1`, `Server_2` and `Server_3`: Setting offset `1` will iterate through the elements in this order: `Server_2`, `Server_3`, `Server_1`. Offset 0 iterates as would be expected from a normal insertion-ordered list.

The purpose of this list is to distribute server load evenly. Each `Job` gets assigned an offset index from the `NetThread`, which is then increased for the next `Job`. Because this is done in the `NetThread`, it is guaranteed that consecutive jobs are assigned consecutive offset values. When the `WorkerThread` writes the `Job` to a server, it will do so by iterating through an `OffsetList` of `ServerConnection` with the `Job`'s offset applied (In case of a non-sharded get, only the server with the offset index will be written to). This will balance out the work for the servers because

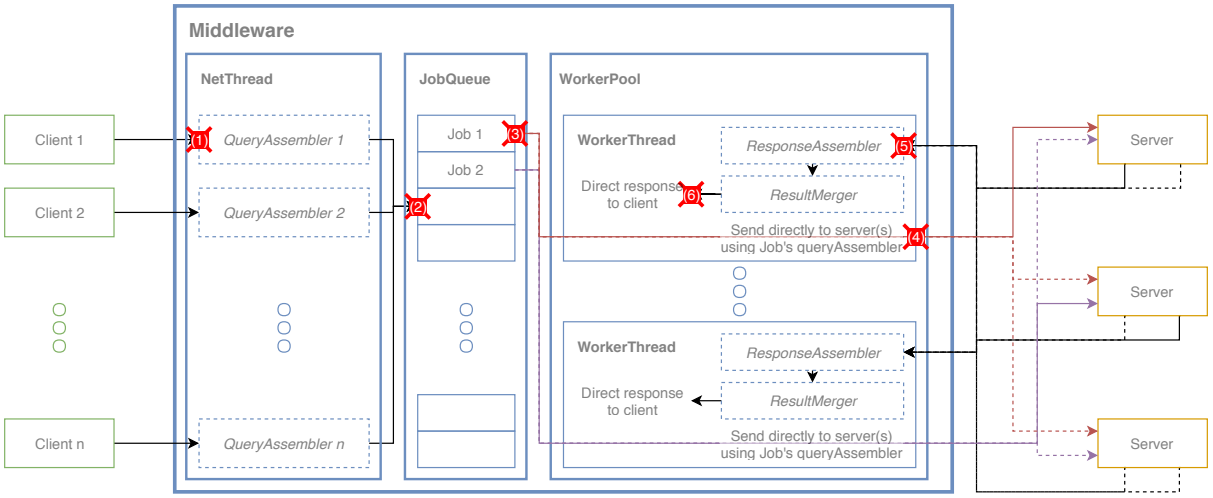


Figure 3: Overview over all measured timestamps. These timestamps are referred to in the following way: *(1)* `clientArrivalTime`, *(2)* `enqueueTime`, *(3)* `dequeueTime`, *(4)* `serverSendTime` (one per server), *(5)* `serverArrivalTime` (one per server) and *(6)* `clientSendTime`.

jobs will select the servers they use with round-robin.

1.2 Instrumentation

Most statistics are saved within `Job` instances, such as timestamps when the middleware has received the request or when the request has been sent to a specific server. To manage this big amount of data, every `WorkerThread` creates two `Statistics` instances upon its creation (one for sets and one for gets), where it will add all finished `Jobs` to a list. As the middleware shuts down, the `WorkerPool` instructs all `WorkerThreads` to dump their statistics. They will then consolidate all `Jobs` and put out two files: A Per-second file (consolidates all `Jobs` that finished in the same one-second window and puts out one line per such window) and histograms (puts out a response-time histogram of all `Jobs` processed with a bucket size resolution of 100μs). The files used to generate all plots are the per-second file. Figure 3 shows all points where a timestamp is taken and saved for a `Job`. From differences between these timestamps, we infer various metrics, all of which are aggregated in the aforementioned one-second windows:

- Index of the current window
- The total number of `Jobs` that finished within this window
- The number of servers, for each of which follows:
  - Server processing time (average, median and 95th percentile)
- Response time (average, median and 95th percentile)
- Queue time (average, median and 95th percentile)
- Enqueue size (average, median and 95th percentile)
- Dequeue size (average, median and 95th percentile)
- `NetThread` time (time spent in `NetThread`) (average, median and 95th percentile)
- Processing time (time the `WorkerThread` was busy with the `Job`) (average, median and 95th percentile)

- Worker time (time from dequeuing until all servers are written to) (average, median and 95th percentile)

Each `WorkerThread` generates a unique file containing its thread ID and writes all one-second windows in order out to this file. Additionally, the file contains the total miss count, the total job count (on line 2) and an error statistic for all error types (line 3).

To increase accuracy for the interacrive law, the `NetThread` holds a table for each host connected and measures all delays between sending a completed job back to a host and receiving a new one from the same host. The `NetThread` will dump this aggregated table when it is requested to shut down.

2 Baseline without Middleware (75 pts)

The goal of these experiments is to explore the limitations of the system without any middlewares involved. In two parts we first try to find out how a memcached server behaves under heavy load. We use only one server and increase the amount of memtier clients connected. In the second part, we test how much load we can generate in memtier by using two servers but only one memcached server.

2.1 One Server

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[1, 2, 4, 8, 16, 32, 48]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	3 (70 seconds each)

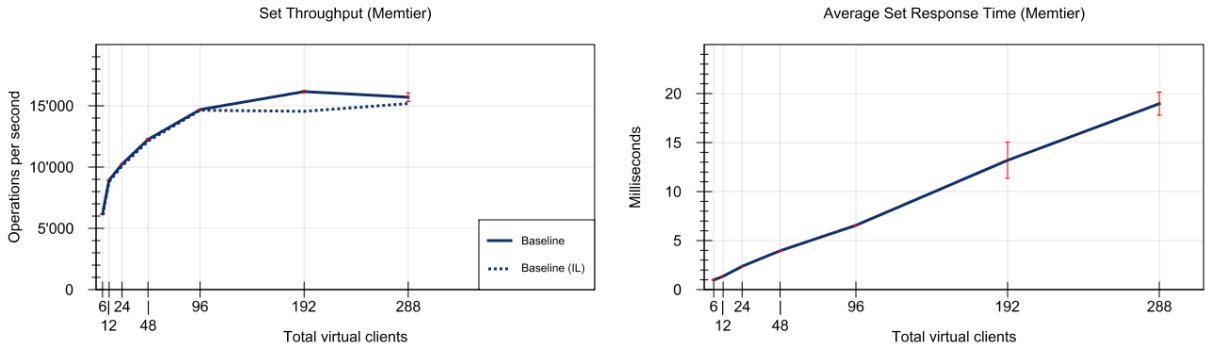
In this experiment, three load generating memtier clients get connected to one server. The clients are run on "Basic A2" (dual-core) virtual machines on the azure platform, the server is run on a "Basic A1" (single-core) virtual machine. The number of virtual clients per thread is varied between 1 to 48 with 6 steps in between, run once for a write-only workload and once for a read-only workload. As with every following experiment in this report, every experiment was repeated 3 times and 5 seconds have been cut of from the beginning and the end of measurements to ensure excluding instabilities during warmup and cooldown phases.

2.1.1 Explanation

The graphs presented in this section have been generated in the same way as those in sections following. Unless stated otherwise, for all plots in this report, the axes used are linear and error bars indicate the standard deviation between repetitions.

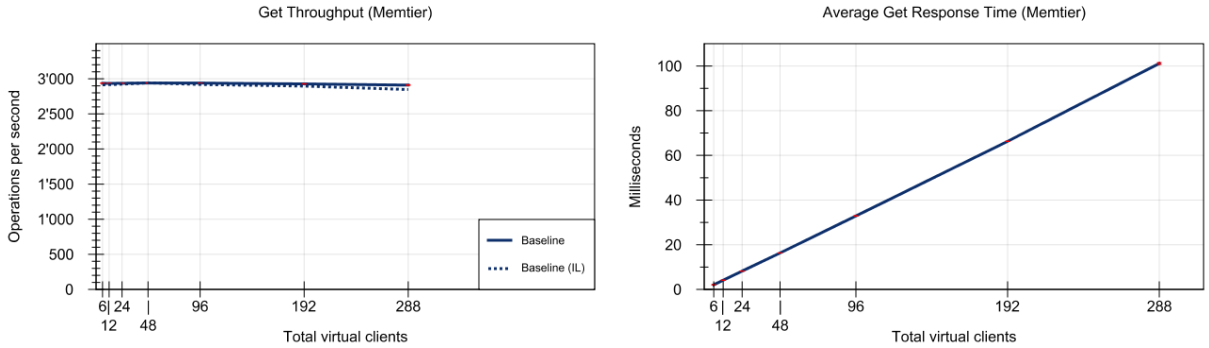
Sanity check In order to validate various metrics and graphs, such as the correlation between throughput and response time, this and future sections will include a sanity check paragraph intended to assess the correctness of the data presented.

An important tool for this is the interactive law, which brings throughput and response time in relation to each other. The basic formula is as follows: $TP = \frac{N}{RT+Z}$, where Z is the thinking time (the time a client waits/thinks after receiving a response and before following up



(a) Accumulated average write-only throughput

(b) Average write-only response time



(c) Accumulated average read-only throughput

(d) Average read-only response time

Figure 4

with a new request) and N, the total number of clients present. The model is best suited for a client-server system such as the one present in this section. The theroetical throughput will be calculated based on the response time using the interactive law and will be plotted alongside the actually measured throughput. This line will be present in all throughput graphs as a dotted line and be suffixed with "(IL)". Here, we assume Z to be zero, since memtier almost immediately follows up a response with a new request, and N to be the total number of virtual clients.

Write-only Regarding plot 4a, the interactive law matches the actual throughput well, except for the data point at 192 virtual clients, where it is quite a bit lower and data point 288, where it is slightly lower. In the corresponding response time graph 4b, the same data points show a significant variance between repetitions. Regarding the first data point, the situation becomes more clear when reading the memtier output: During the first repetition, the response time in one client went from an average of 12.54ms to over 708ms in one line of output. Altough the latencies between hosts is measured using ping, this anomaly was not captured. The reason may be that ping measurements leave gaps of one second between measurements, where maybe the short connection interruption lied completely within. In the Azure platform, such sudden and inconsistent latency changes can happen since the hardware infrastructure is shared between many different customers.

The influence of this sudden increase in latency over the average throughput of the client is very low, since, for throughput measurements, all memtier instances' average throughput will be accumulated. For the response time however, this effect is more noticeable because they are

not accumulated but only averaged over clients. This is the reason the interactive law doesn't fit well for this data point.

The data point at 288 virtual clients is instable too. Again, there was a small spike in latency, this time though visible in the captured ping diagram 5. The spike can be seen in the client logs of all clients, meaning that the server itself probalby has experienced a network anomaly. This time, since all clients of the repetition were affected and thus the smoothing effect of the accumulation over the averages didn't occur, the influence of the anomaly is visible on the throughput plot as well, leaving the errorbar even bigger than at 192 virtual clients (even though the latency anomaly was not as intense). Note that this situation occurs many time throughout the report, and whenever stated to be similarly to the situation here, the ping plots or the log files reveal similar networking anomalies.

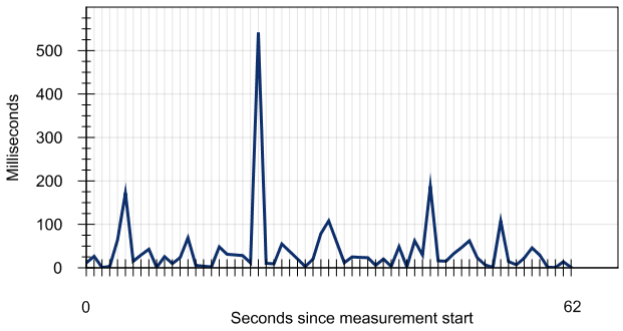


Figure 5: Latency measurement between a client and the server in a repetition with 96 virtual clients total.

Read-only For read-only workload, both throughput and response time have very little variation between repetitions. The IL slightly mismatches the measured throughput for high client counts, which we attribute to network instabilities (as the response time error for 96 virtual clients is slightly enlarged).

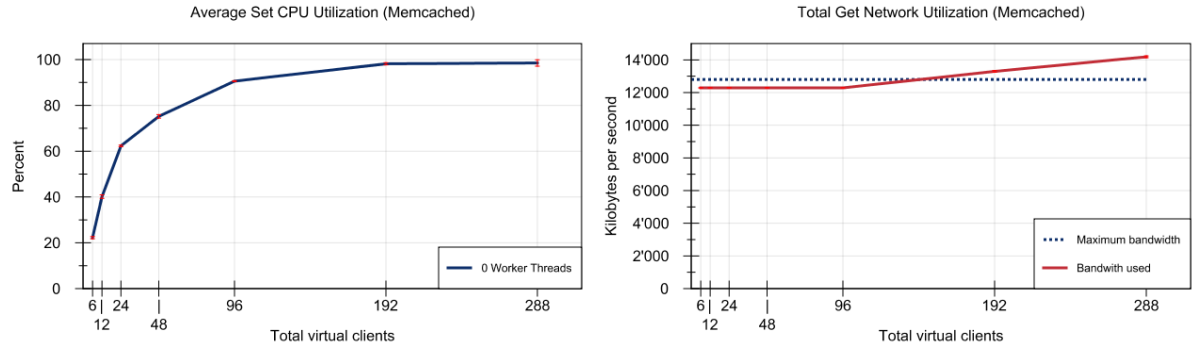
Analysis

Write-only Throughout the report, during analyses we'll define a saturation point and a maximum throughput point which may be different from each other. The saturation point is reached when the system shows an increase in response time, implying that additional load will lead to oversaturation. The maximum throughput point however will maybe be different, since the saturation point may yield very low throughput results, while other configurations yield higher throughput with an acceptable increase in response time.

In write-only workloads, the increase in throughput begins to soften after a total of 96 virtual clients. At this point, the server is saturated, as is visible in the slight knee in the response time graph at 96 virtual clients. After 96 virtual clients, the throughput will only marginally increase, but the response time increase is significant: When we add more virtual clients, and thus more requests to an already saturated system, the throughput stays about equal (because of the saturation) but the response time will start increasing more (a knee will be visible) because on average, a request has to wait longer to be served. We see this behaviour throughout the entire report. An increase from 14'691 ops/s to 16'161 (at 192 virtual clients),

which is about 10%, is met with a corresponding increase form 6.56 ms to 13.2 ms, which is about 101%. Even though we have to assume the actual response time to be slightly lower (since one repetition has experienced a short burst in latency, skewing our measurement to a slightly higher number), a 10% increase in throughput does not warrant nearly doubling the response time. In comparison, going from 48 virtual clients to 96, the increase in response time is not as harsh (65%), because the throughput still increases significantly (20%). Thus, we also select 96 virtual clients as the maximum throughput configuration.

In this experiment, the server is the bottleneck. To illustrate this, we look at its CPU utilization for each experiment. Figure 6a shows that the memcached cpu utilization and the throughput correlate. The cpu utilization reaches its maximum somewhere between 96 and 192 clients (since it is at its maximum at 192 clients), implying that the server becomes oversaturated in between. This also explains the small decrease in throughput noticeable for 192 virtual clients. The server has to manage more connections when it is already saturated, but does not have more cpu capacity to do so. Thus, a smaller timeframe of processing will be available to the server to handle requests than with 192 virtual clients, effectively reducing throughput and increasing response time further. The memtier cpu usage is at its highest at 192 virtual clients with around 14%, which rules the clients out to be a bottleneck. Bandwidth can be ruled out as well since no host's bandwith utilization is maximized.



(a) Average CPU utilization on the memcached server for write-only workloads

(b) Upstream bandwith utilization of the memcached server for read-only workloads

Figure 6

Read-only The throughput hardly varies with more or less virtual clients, as it stays between 2910 ops/s and 2940 ops/s. The reason is that the server again bottlenecks very early on. In contrast to the write-only workload however, memcached does only reach a maximum cpu utilization of about 11%. However, the bandwith utilization is immediately exhausted with only 6 virtual clients as visible in figure 6b. On the Azure platform, all virtual machines used are rate-limited in their upload (send) speed. `iPerf` measurements between different types of hosts have shown that client hosts ("Basic A2" virtual machines) have a send limit of 200 Mbit/s, middleware hosts ("Basic A4" virtual machines) have a send limit of 800 Mbit/s and server hosts ("Basic A1" virtual machines) have a send limit of 100 Mbit/s. Note that download rates are unlimited for all virtual machines (*"Ingress is not metered or limited directly"*, as stated by azure). We attribute the bandwith utilization exceeding the maximum bandwidth at 192 and 288 virtual clients with `dstat`'s imprecision and rounding errors (as well as the Azure platform not very exactly enforcing the rate limits). With a value size of 4096 bytes, a single get reply will amount to approximately 4135 bytes. Not including the overhead of TCP packages, the 100

Mbit/s bandwith limitation for the server will restrict it to be only able to send about 3'170 responses to a get request. This limit is already reached with 6 virtual clients, meaning the saturation point of the system is at 6 total virtual clients. As for the maximum throughput configuration, we choose 6 total virtual clients too, because later the response time increases but the throughput doesn't.

The response time increasing this steadily has to do with the server having plentiful of cpu resources to handle connections and requests. The response time depends mostly on the queue size at the server, which depends on the number of total virtual clients. Since the memcached server runs with one core and thread only, it will only be able to process one request at a time. With more clients in the system, clients have to wait longer to have their responses sent to them. Note that the average response time is throughout significantly higher than in the write-only case. The reason is that clients have to wait longer because the replies, which contain values, are a lot bigger in size, and thus take longer to be sent back to clients.

Note that, in write-only workloads, the bandwidth will not be a bottleneck since the client's maximum bandwith allows sending up to about 6'348 requests per second (sets are slightly smaller with approx. 4129 bytes per request), which is not reached since a single client never sends more than about 5'500 requests. The server's replies are very small, consisting only of the reply `STORED`, having the server use only a small fraction of its available bandwidth.

2.2 Two Servers

Number of servers	2
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1, 2, 4, 8, 16, 32, 48]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	3 (70 seconds)

In this experiment, one load generating memtier client gets conected to two servers (by running two separate memtier instances). The clients are run on "Basic A2" (dual-core) virtual machines on the azure platform, the servers are run on "Basic A1" (single-core) virtual machines. The same parameter values have been chosen as in 2.1.

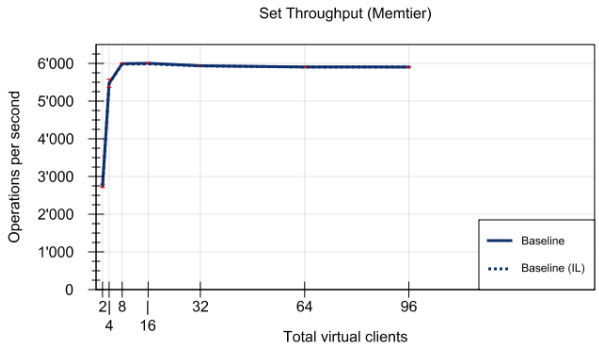
2.2.1 Explanation

Sanity check

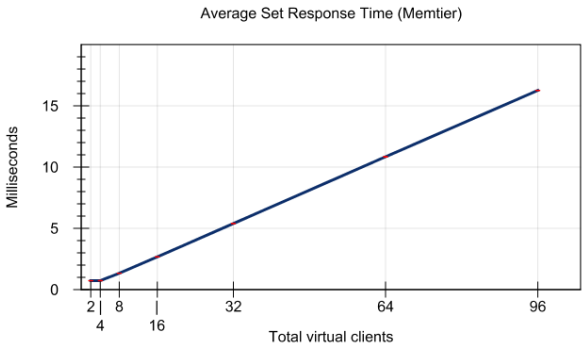
Write-only The interactive law matches almost perfectly with the measured throughput and all errors between repetitions are very small. The data aligning this well implies the network was very stable during these experiments. The reason why the response time still increases after the throughput is stabilized is that by increasing the number of virtual clients, the server queue fills up and individual requests have to wait longer before being processed.

Read-only The same holds for read-only workloads.

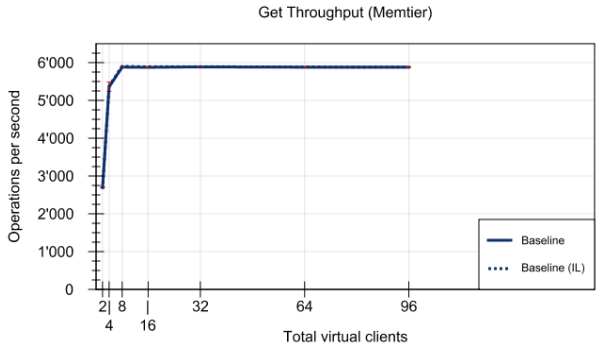
Analysis One interesting question about the data is why both, write- and read-only workloads, behave almost identically.



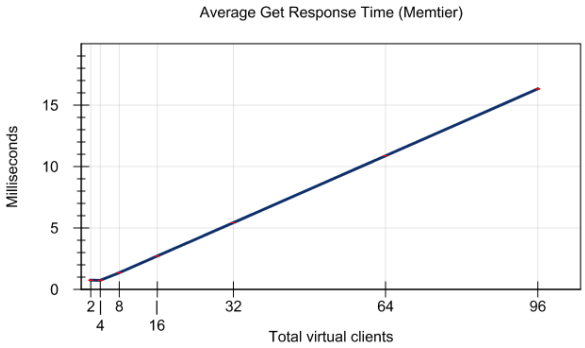
(a) Accumulated average write-only throughput



(b) Average write-only response time

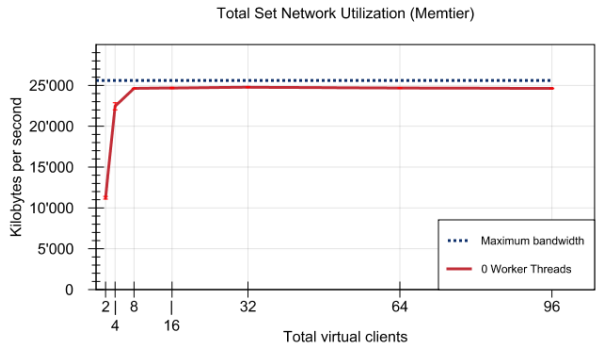


(c) Accumulated average read-only throughput

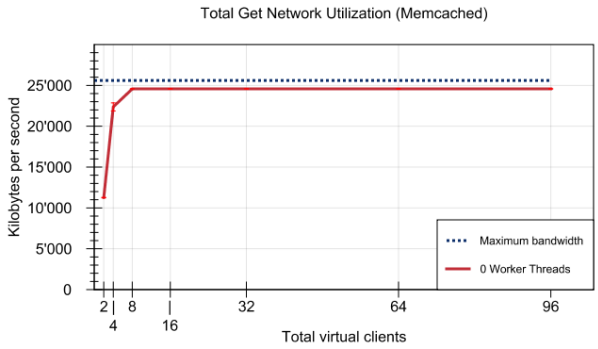


(d) Average read-only response time

Figure 7



(a) Bandwidth utilization of the client host for write-only workloads



(b) Bandwith utilization of the memcached servers for read-only workloads

Figure 8

Write-only For write-only workloads, the maximum throughput is 6'003 ops/s at 8 virtual clients. As mentioned in section 2.1.1, a client host cannot send more than 6'348 set operations per second because of its network rate limit. Since the rate limit will include all tcp headers, but our estimate does not, the effective amount of operations a host can send will be lower. Figure 8a shows that the rapid throughput growth is abruptly halted at 8 clients, where the bandwidth utilization will reach its maximum stay constant for increasing virtual client numbers. The cpu utilization was below 15% for all hosts, ruling out any performance limitations. We

thus conclude the bottleneck to be the client host's network rate limit. The response time is equal for 2 and 4 virtual clients, meaning that the servers are underloaded. In section 2.1 the response time immediately increased, which is not the case here, since we have more servers and less clients. The response time shows a knee at 4 virtual clients, which we conclude to be the saturation point of the system. However, since the system is network-rate-bound, the saturation of the memcached servers is not captured. Assuming no network rate limit, we would expect the throughput to still increase since we know from section 2.1 and the low memcached CPU utilization, that the server's saturation point lies higher. As for the maximim throughput configuration, choosing 8 virtual clients total is not appropriate, because although the throughput increases 10% for both sets and get from the 4 virtual client configuration, the response times increases by 83% and 84% respectively. Thus, we also choose the 4 virtual client configuration, which has the equal response time as 2 virtual clients but almost twice as much throughput.

Read-only For read-only workloads, the maximum throughput is similar at 5879 ops/s at 8 virtual clients. This time, the bottleneck is the network rate limitation imposed by the server hosts. As mentioned in 2.1.1, a server host cannot send more than 3'170 get responses per second. Because there are two servers present, both running on their own host machine, this amounts to a total of 6340 get replies per second, approximately the same as for sets. Figure 8b shows the same pattern as before. The response time is a lot lower here, because using the same amount of clients, they have to wait for a shorter amount of time becasue the servers can serve twice as many requests per timeframe. Again, the system is network-rate-bound. We choose the same system saturation and maximum throughput points as in the write-only case for the same reasons (all numbers are virtually identical).

2.3 Summary

Maximum throughput of different VMs.

	Read-only workload	Write-only workload	Configuration gives max. throughput
One memcached server	2'940 ops/s	14'691 ops/s	write-only: 96 total virtual clients, read-only: 6 total virtual clients
One load generating VM	5'360 ops/s	5'472 ops/s	write-only and read only: 4 total virtual clients

In 2.1, we tried exhausting a memcached server with 3 client machines. For writes, The bottleneck was the memcached server's cpu performance, successfully exhausting the server, for gets, it was its bandwith rate limitation, meaning the server was not exhausted. In 2.2 we tried exhausting the memtier clients, which we could not. The bottleneck was, for both writes and reads, the bandwidth limitations of the client host and servers respectively. We received extremely similar numbers because the rate limit of the single client at 200 Mbit/s for sending sets and the rate limit of two servers at 100 Mbit/s each for replying to gets amount to the same overall bandwidth limitation. The reason is that in the case of gets, the response is very large, meaning the server bandwidth will be exhausted, while in sets the request itself is very large, meaning the client bandwith will be exhausted. In just one experiment, namely the sets in the one server setup, the bottleneck was not a bandwith limitation, but cpu utilization. This experiment did not reach any bandwidth limitations because the memcached server was oversaturated with fewer requests from the clients (which together have an upload limit of

600 Mbit/s) than they would have needed to send in order for the network rate limit to be hit. So, the memcached server was perfomance bound and this experiment reached the highest throughput by far.

Sections 2.1 and 2.2 are very much different in terms of maximum throughput. The reason is that each pair of write-only, read-only, one server and two servers experiments has a different bottleneck - except for one server read-only and two servers read-only, where the bottleneck is server rate limitation, but the results are different because the total rate is twice as large for the second experiment because there are two servers involved, leaving approximately twice the throughput.

The purpose of this experiment was to make two comparisons, namely the differences between write- and read-only workloads, and the differences between having many clients and few servers and having few servers and many clients. We cannot compare write-only experiments to read-only experiments because the network bottleneck leaves them either with almost identical results (two server case) or results that are different from each other because one doesn't measure the characteristic that we wanted to compare (one server case, we have no information about how the memcached server performs under heavy read-only workload because we could not produce it.) We can also not compare the one-client results with the one-server results because again, the changes brought by changing the system could not be captured as we could not generate more than one interesting load because of the network limitation.

So, the main takeaway is the important information gained by the write-only experiment with one server about the server performance under heavy write-only workloads. We don't have this kind of data for read-only workloads because the server was never exhausted because of the bottleneck, but we assume the read-only performance of memcached to be similar. It may be faster because get requests could be cached, or slower, because the server needs to send back more data.

3 Baseline with Middleware (90 pts)

The goal of these experiments is testing the effect introducing one or two middlewares will have on the system. For a number of different thread counts for the middleware(s), we vary the amount of virtual clients to extensively test the performance of the system.

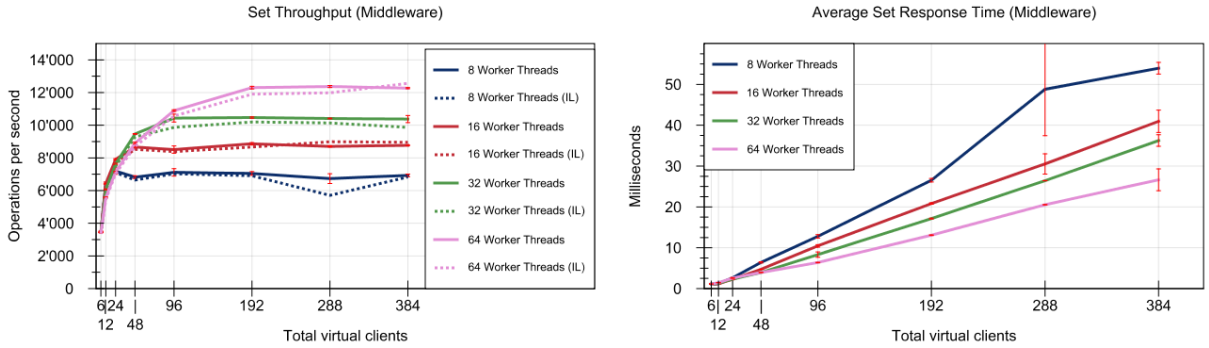
3.1 One Middleware

In this experiment, three load generating memtier clients are connected to one middleware. This middleware is connected to one single server, forwarding requests and relaying the results back to the client. The clients are run on the same hosts as before, as are the servers. The middleware is run on a "Basic A4" (octa-core) virtual machine. For four different amounts of worker threads used in the middleware, we'll vary the amount of virtual clients connected to it. In this baseline experiment, we test the performance of the middleware.

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, 2, 4, 8, 16, 32, 48, 64]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1
Worker threads per middleware	[8, 16, 32, 64]
Repetitions	3 (70 seconds each)

3.1.1 Explanation

All graphs' error bars depict the standard deviation between repetitions, unless explicitly stated otherwise.



(a) Accumulated average write-only throughput(measured in the middleware)

(b) Average write-only response time (measured in the middleware)

Figure 9

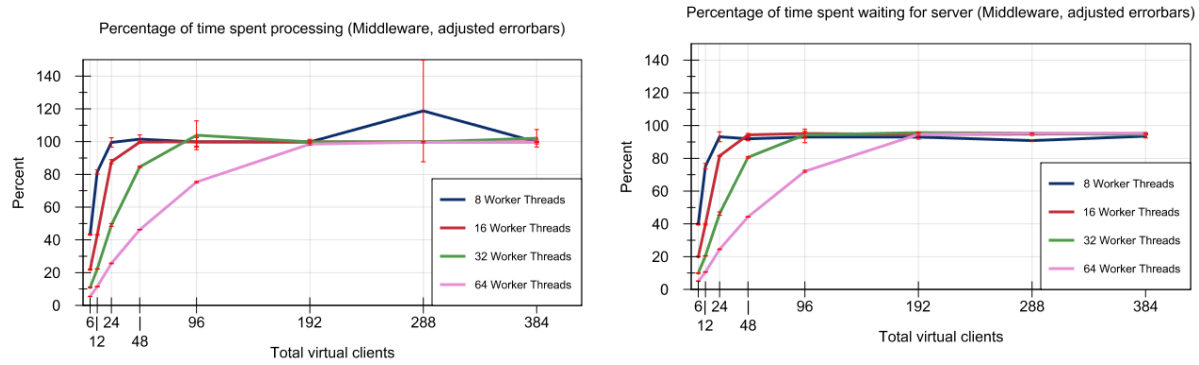
Sanity check As a sanity check for queue graphs, never are there more jobs in the queue than there are virtual clients in the system.

Write-only Figure 9 shows the throughputs and response times measured in the middleware for write-only workloads (Appendix figure 23 has graphs for the clients as well, they are extremely similar). The overall shapes of the figures are very similar, which they need to because the middleware just forwards any requests and replies. However, small discrepancies between the values are attributed to differences in timing and data aggregation and output. To check the throughput matching the response time, we use the interactive law similarly as before in section 2.1.1. The interactive law models a server-client system, which was a good match for the baseline. For this experimental setup however, assuming thinking times to be zero will produce inaccurate results, since the time between two requests from the same client will be significantly longer, now that the middleware is involved. It will not have a new request ready from the same memtier client after receiving a reply, because it will have to be sent back first and a new request has to be sent by the client. As mentioned in 1.2, the network thread in the middleware measures the average time a memtier host takes to follow up a new request after having received a response from it. Averaging all these measured delays and using them as thinking time Z provides more exact results. Small differences stem from the fact that time the middleware spends on a job before sending it to the server is not incorporated in the thinking time, such as parsing. For memtier clients we keep using $Z = 0$.

The theretical throughput calculated using the interactive law fits the measured throughput well. For a few data points, the line does not fit perfectly. The reason for this is extensively explained in 2.1.1 and applies here as well. All inaccuracies in the IL throughput can be ascribed to latency instabilities between hosts during the experiment. The enlarged error bars in the response time graph corresponds to the interactive law not producing an exact fit. Another important check is to make sure that response times are lower in the middleware than in the memtier clients, which is the case for all data points except for 8 worker threads / 288 virtual clients, where a network anomaly took place accross the whole network, skewing the average

even more on the middleware than on the memtier clients. All inaccurracies can be confirmed to be network anomalies by checking the appropriate log files. We thus assume all skewed data points to approximately lie on the straight line between the previous and following points.

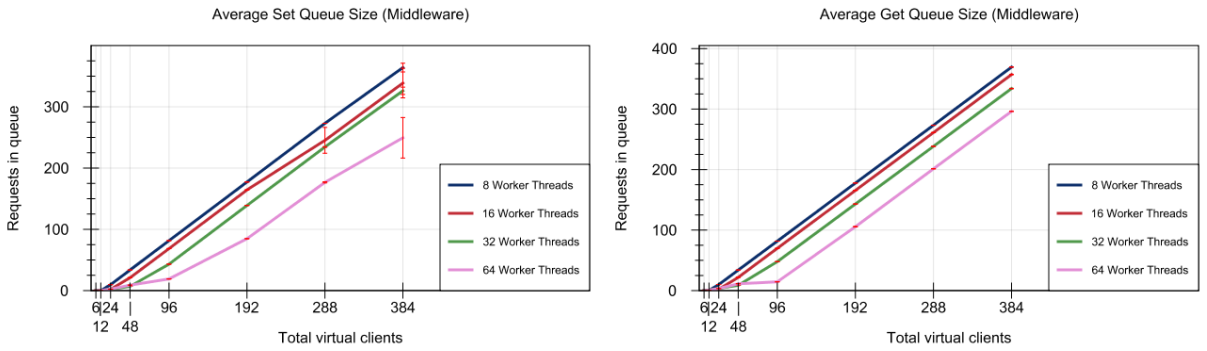
Read-only For read-only workloads the graphs between memtier clients and the middleware are also very similar (again, client graphs are in the appendix 24. As before, we plot the theoretical throughput using the interactive law, which fits very well, except for the data point at 64 worker threads and 192 virtual clients, where measurements experienced latency inconsistencies. The same reasons apply as before (see 2.1.1 again), which can be validated by checking the logs. Here, also, memtier response times are consistenly higher than middleware response times, as expected.



(a) Average time percentage worker threads spend processing jobs for write-only workloads

(b) Average time percentage worker threads spend waiting for memcached replies in write-only workloads

Figure 10: Note that the errorbars in these plots have been derived and adjusted from the processing time and server wait time measurements respectively (by being sacled with TP/WT) and do not include errors of throughput measurements involved.



(a) Average queue size for write-only workloads

(b) Average queue size for read-only workloads

Figure 11

Analysis

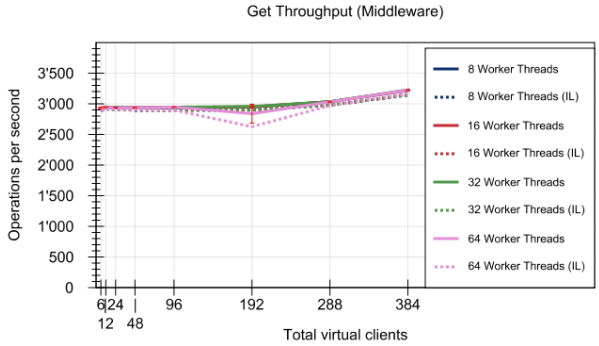
Write-only The response time starts growing immediately. The individual worker thread configurations have different points at which the increase starts growing more quickly, indicating saturation. For 8 worker threads, the knee is visible at 24 virtual clients, for 16 it is visible at 48 virtual clients, for 32 it is visible at 96 virtual clients and for 64 worker threads it is visible at 192 virtual clients. The throughput plot confirms that these saturation points are accurate, as the throughput flattens at the according virtual client count (e.g. the throughput for 32 worker threads flattens at 96 virtual clients, which corresponds to the chosen saturation point).

Using more worker threads implies that the saturation will be occurring with more virtual clients. The reason for this is that the bottleneck lies with the middleware, more precisely the number of worker threads. In order to show this, we show that the worker threads are at full utilization by analyzing the processing time. Recall that the processing time is how long a worker thread is occupied with a job. The measurement starts when a worker thread dequeues a job and ends when it has sent the full reply back to the client. At all other times a worker thread is polling the queue, meaning it is idle and waiting for a job. Figure 10a shows the percentage of time a worker thread spends processing jobs on average. Notice that this plot's data has been calculated from the measured average throughput and processing time and that its error bars are only an approximation derived from the standard deviation between repetitions of the average processing time measured. For all worker thread counts, the throughput does not grow anymore after the worker thread utilization reaches its maxiumum at 100%. More worker threads can work on more jobs at the same time, meaning that maximum utilization will be reached for more virtual clients. All worker threads are then occupied at all times and never need to wait for a job, meaning that any additional workload will necessarily have to queue up and that throughput cannot go higher. As another effect, jobs being forced to wait will mean that the response time increases, which is the reason there are knees visible in the graph. Figure 11a shows the average queue size. The saturation points are visible too in the plot, albeit faintly. The plot shows that using more worker threads helps keeping the queue size low. The higher the ratio of worker threads to virtual clients, the later the queue size starts to increase, because the probability of a worker thread waiting for a new job will be higher with more threads and fewer virtual clients. Similarly, the response time increases more slowly the more worker threads are used: Since there are more worker threads available, a job has to wait for a shorter time until one becomes available.

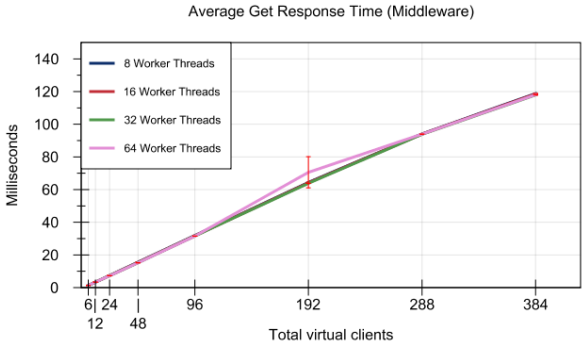
Figure 10b shows the percentage of time a worker thread waits for a memcached reply. Please note again that the for this plot is calculated and that the error bar has been derived from the measured server response time only. For data points where saturation has been reached, the memcached waiting time is about 95%. The memcached server is not yet saturated, as its cpu usage is never higher than 80%, which is to the server's saturation point as shown in 2.1. The middleware's cpu usage never exceeds 38%. No network rate limits are met at any point in the system. The fact that worker threads are mostly waiting on network I/O, and can thus be scheduled out until new packets arrive also explains how the octa-core virtual machine can handle this many threads. We can thus also assume that adding additional worker threads will further increase throughput.

As for the maximum throughput configuration, multiple data points need to be considered. The highest throughput measured was 12'370 ops/s on the middleware at 64 worker threads and 288 virtual clients. This point is unsuited though, since the configuration at 192 virtual clients has a throughput of 12'302, which is less than one percent lower, but shows a 36% lower response time (13.09 ms vs. 20.53 ms). Now, the data point at 96 virtual clients has a throughput of 10'896 ops/s (11% lower) and a response time of 8.83ms (33% lower). Any lower data point will have a reduction of more than 12% from the absolute maximum, which is not appropriate. The 33% decrease in response time when choosing 96 virtual clients is not significant enough

compared against the 11% increase in throughput, so we choose the data point at 64 worker threads and 192 virtual clients.

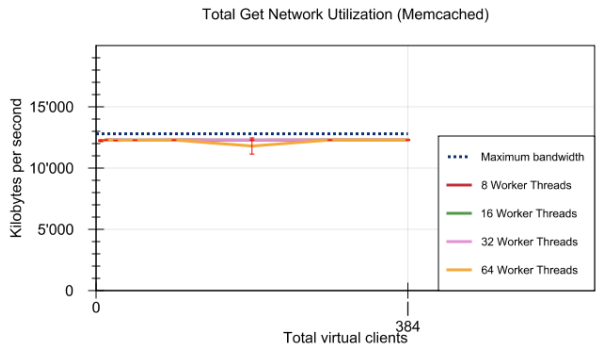


(a) Accumulated average read-only throughput(measured in the middleware)

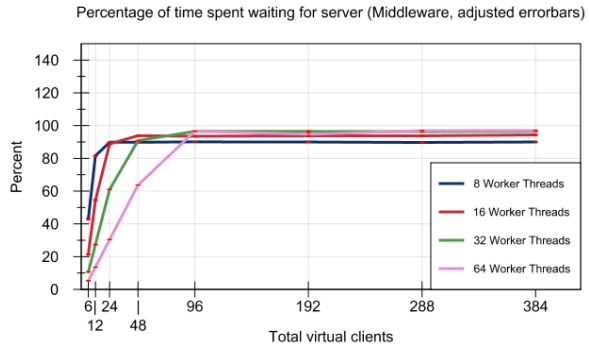


(b) Average read-only response time (measured in the middleware)

Figure 12



(a) Upstream bandwith utilization of the memcached server for read-only workloads



(b) Average time percentage worker threads spend waiting for memcached replies in write-only workloads

Figure 13

Read-only The response time graph does not show any knees whatsoever, implying that the system is immediately saturated. The throughputs and response times are virtually identical for all worker thread configurations. The reason is that here the bottleneck lies in the network rate limitation of the server again. Figure 13a shows the upstream bandwidth utilization of the memcached server and the maximum rate. For every configuration run, the bandwidth is exhausted. The same exact situation from the read-only of 2.1 applies. Again, 100 Mbit/s limits the server to theoretically send at most 3'170 replies back. The only difference in the system is that we have added a middleware in between the server and the clients. Figure 13b shows that the worker threads in the middleware are spending the overwhelming majority of their time waiting for the memcached server to reply. For low virtual client counts some data points are below their maximum. The reason is that there are more worker threads than clients. When accountig for all worker threads that are forced to be idle (because there are no more jobs in the system than virtual clients), it can be seen that the portion of worker threads utilized are most

of the time waiting for the server. As an example, for the data point 64 worker threads and 48 virtual clients, the maximum possible average waiting time is 75% because 25% of worker threads are forced to be idle. The actual average waiting time is 62%, meaning that the worker threads are almost exhausted with waiting for the server responses. Their utilization can still be slightly increased using more virtual clients, which also results in the slight slope decrease in response time and slight increase in throughput. Checking the logs confirms bandwidth limitations are reached for no other hosts and no host reaches full cpu utilization (7% for client hosts, 9% for the middleware and 10% for the server). The queue plot 11b shows behaviour very similar to the write-only case but the lines are more straight, implying the middleware being more stable with lower utilitaztion.

The maximum throughput at 382 virtual clients is reached with 3'218 ops/s, but is less than 10% higher than the throughput at 6 virtual clients, which is 2'924 ops/s but has the lowest response time at 2.06 ms. Thus selecting the data point at 6 virtual clients is a sensible choice for all worker thread configurations.

3.2 Two Middlewares

In this experiment, three load generating memtier clients each run two instances of memtier, each of which is connected to a middleware. Both middlewares are connected to the same server, forwarding requests and relaying the results back to the client. The clients are run on the same hosts as before, as are the servers and middlewares. For four different amounts of worker threads used in the middlewares, we'll vary the amount of virtual clients connected to it. In this baseline experiment, we test the performance of two middlewares.

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, 2, 4, 8, 16, 32, 48, 64]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8, 16, 32, 64]
Repetitions	3 (70 seconds each)

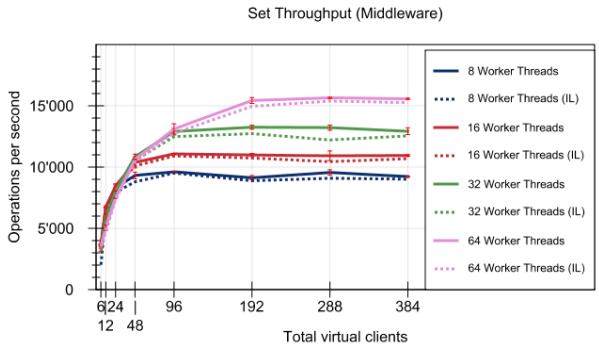
3.2.1 Explanation

All graphs' error bars depict the standard deviation between repetitions, unless explicitly stated otherwise.

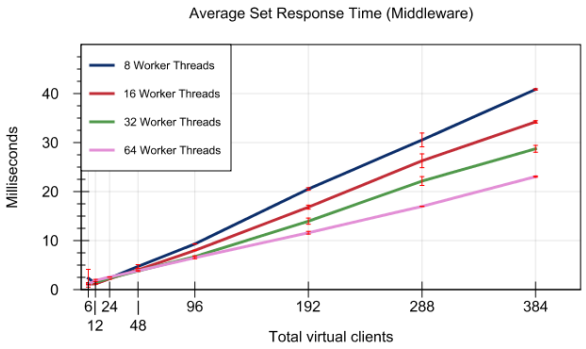
Sanity check As a sanity check for queue graphs, never are there more jobs in the queue than there are virtual clients in the system.

Write-only The throughput and response time graphs match well between the middlewares and the clients (See also appendix 25 for client graphs). The interactive law holds, and where it is not accurate, the same reason holds as before (network anomalies). As expected, all response times are higher in the memtier clients than in the middlewares.

Read-only The graphs for both metrics match well between the middlewares and the clients, too (Appendix 26 for client graphs). The interactive law holds well, too. For 8 worker threads and 6 virtual clients it is extremely inaccurate in the client plots. The response time for this data point is extremely low (close to zero) and a network anomaly happened in the third

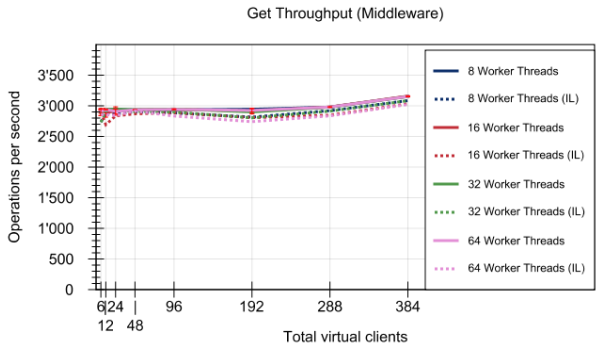


(a) Accumulated average write-only throughput(measured in the middlewares)

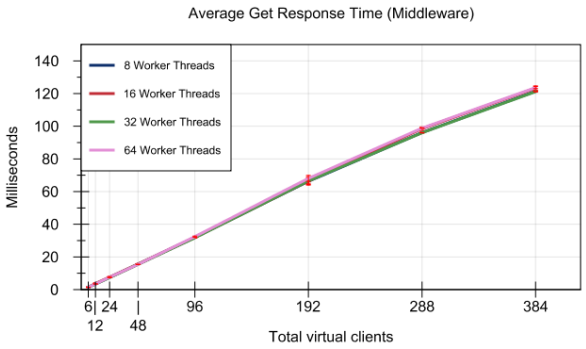


(b) Average write-only response time (measured in the middlewares)

Figure 14



(a) Accumulated average read-only throughput(measured in the middlewares)



(b) Average read-only response time (measured in the middlewares)

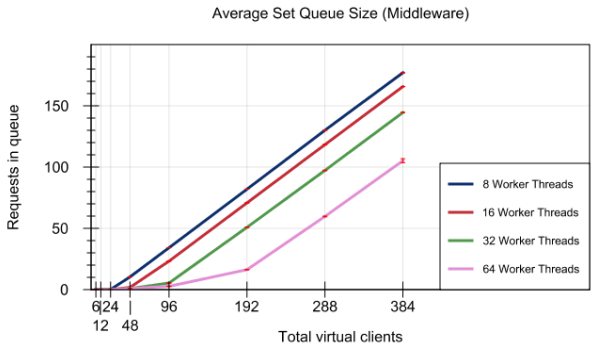
Figure 15

repetition, which is made apparent by the enlarged error bar in the response time graph. The response time being this close to zero will have the interactive law be very suspectible to small variances, because when dividing through the response time, it being close to zero will produce a numerical instability. Again, as expected, all response times are higher in the memtier clients than in the middlewares.

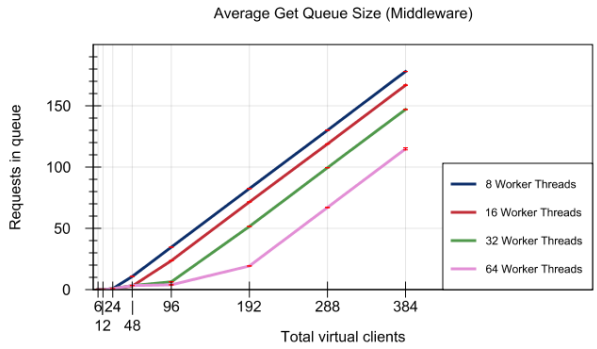
Analysis

Write-only The throughput and response time graphs look very similar as in 3.1, but achieve higher throughputs with lower response times. The highest throughput encountered is 15'638 ops/s at 64 worker threads and 288 virtual clients as measured on the memtier clients.

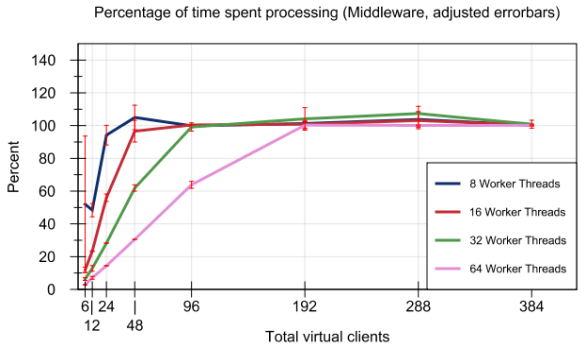
The situation concerning saturation is basically the same as in 3.1. The response times show the same saturation points as before (8 worker threads at 24 virtual clients, 16 at 48, 32 at 96 and 64 at 288), again confirmed by the throughput stabilizing after the point has been reached. We assume that, by now having two middlewares instead of just one, to alleviate the bottleneck encountered in section 3.1. And indeed, the throughput has increased for all worker thread configurations. By adding a second middleware, for each saturation point, the throughput increases on average by 19%. Each step between worker thread configurations doubles the



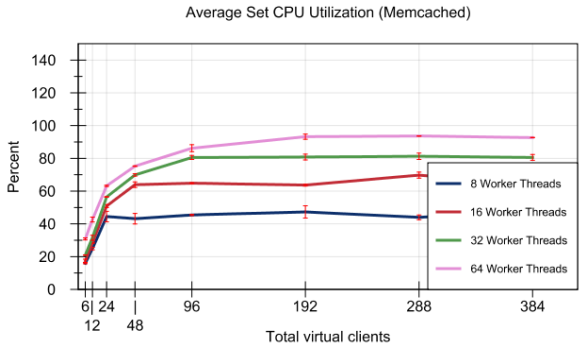
(a) Average queue size for write-only workloads



(b) Average queue size for read-only workloads



(c) Average time percentage worker threads spend processing jobs for write-only workloads



(d) Average CPU utilization on the memcached servers for write-only workloads

Figure 16

amount of them. Adding a second middleware for WT worker threads is thus expected to have a similar effect as choosing the next higher worker thread configuration. By increasing the amount of worker threads in 3.1, the average increase between saturation points was 16%, which confirms our finding.

Because the increase in throughput is still not halted even after using a total of 128 middleware worker threads in the system, we assume the bottleneck to be the worker thread count again. To check if this is the case, we again look at the time spent in processing in figure 16c. We notice that utilization is lower than in 3.1 for few virtual clients, which is because the load is shared between the middlewares, which is also the reason the response time is lower. After utilization reaches its maximum, the saturation points are reached. To make sure the server itself is not yet at its maximum utilization, we check its cpu usage in figure 16d. We see that it reaches its maximum at 93%. The server's saturation point, as we have learnt from section 2.1, is at 96 virtual clients and a throughput of approximately 14'700 ops/s. We reach and even exceed this throughput when using 64 worker threads. In section 2.1, the throughput increased up until a memcached cpu utilization of 98.5%. Although not by much, we assume the throughput could still be increased slightly by using more worker threads (the middleware cpu is only at 16% utilization max). However, the server would become the bottleneck and probably not exceed the maximum throughput of section 2.1.

As for maximum throughput configuration, we pick 64 worker threads and 96 virtual clients. Using the next data point increases throughput by 18% and response time by 70% (which is significant), using the next lower yields 19% less throughput (31% less than maximum through-

put, which is too much) and 38% lower response time (which is not worth the 31% decrease in throughput). At 64 worker threads and 96 virtual clients we have a throughput of 13'102 ops/s and a response time of 7.41 ms.

Read-only This situation also aligns with the one in 3.1. With 3'151 ops/s the highest throughput is virtually equal with the one-middleware case. As was to be predicted, since the server capacity remains unchanged, it still is the bottleneck. The upstream bandwith usage plot is virtually identical to the one presented in 3.1 and is thus ommitted. The saturation point is reached immediately at 6 virtual clients for all worker thread configurations. Since the entire system processes the same load as with only one middleware used, the load is distributed evenly between the middlewares. To confirm this claim, we can look at the queue sizes in figure 16b. When we compare this plot with the queue size where only one middleware was in the system (fig. 11b), we notice that all configurations have shifted down by one. To be more concrete, the line for the configuration with 64 worker threads in 3.1 matches the line for 32 worker threads in 3.2, the line for 32 worker threads in 3.1 matches the line with 16 worker threads in 3.2 and so on. The reason is that the total amount of worker threads in the system is twice as high when using two middlewares. Since the throughput is the same for both experiments, splitting it evenly among two middlewares will lead to both of them processing half as many requests than in the same system that only uses one middleware. Thus, the average queue size will behave the same for one middleware using WT workerthreads, and two middlewares using $WT/2$ workerthreads. This shows very well that the load generated by the clients is evenly distributed among middlewares.

For the maximum throughput configuration, we pick 6 virtual clients for all worker thread configurations because the average throughput increase is less than 1% for 12 virtual clients (avg. 2'900 ops/s vs. avg. 2'935 ops/s), but has an average increase of 48% increase in response time (avg. 2.86 ms vs avg. 4.22 ms).

3.3 Summary

Since both experiments have vastly different performance metrics, finding one experiment that fits both write-only and read-only workloads well is difficult. The four maximum throughput configurations also don't overlap in the same section, because for write-only workloads, increasing the client count increases the throughput as well as the response time, which means we preferrably take a high virtual client count, but for read-only workloads only the response time increases with more virtual clients, which means that choosing the lowest client count is always preferrable. For one middleware we choose 32 worker threads and 48 virtual clients. For two middlewares we choose 64 worker threads and 96 virtual clients.

Maximum throughput for one middleware.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware	2'933 ops/s	15.24 ms	4.83 ms	0
Reads: Measured on clients	2'933 ops/s	16.36 ms	n/a	0
Writes: Measured on middleware	9'468 ops/s	3.94 ms	1.05	n/a
Writes: Measured on clients	9'463 ops/s	5.11 ms	n/a	n/a

Maximum throughput for two middlewares.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware	2'943 ops/s	32.44 ms	0.35 ms	0
Reads: Measured on clients	2'937 ops/s	32.88 ms	n/a	0
Writes: Measured on middleware	13'109 ops/s	6.51 ms	0.24 ms	n/a
Writes: Measured on clients	13'102 ops/s	7.41 ms	n/a	n/a

The configuration chosen for one middleware is a tradeoff between throughput and response time. Further increasing the virtual client count (96 clients) and choosing 64 worker threads would improve the average throughput (of sets and gets weighted equally) to 6'918 ops/s from 6'228 ops/s (+11%), but would also raise the average response time significantly from 10.74 ms to 20.72 ms (+93%) (memtier measurements). Decreasing to 24 virtual clients changes throughput to 5'283 ops/s (-15%) and response time to 5.68 ms (-47%). This table is about maximum throughput, so we choose this datapoint because we would lose more when decreasing than we'd gain when increasing worker counts while still taking response time into account.

For two middlewares, going up to 192 virtual clients would increase the average thoughput from 8'019 ops/s to 9'172 ops/s (+14%) and increase the response time significantly from 20.15 ms to 39.42 ms (+96%). Going down to 48 clients, the average throughput woould go to 6'778 ops/s (-15%) and the response time would decrease to 10.47 ms (-48%). With the same reason as before, we choose this data point. For both tables, no miss rates have occurred, as the servers have been populated before the experiments.

In both experiments, for read-only loads, the bottleneck was again the server upload bandwith limitation. This was expected from our experiences in section 2. The additonal middleware did not change anything about the bottleneck, resulting in very similar throughputs and response times for both experiments. To increase throughput, the network rate limit would have to be increased or a second server would have to be added, which would boost performance in the system as was seen in section 2.2. In the experiment using one middleware, we concluded that the amount of worker threads constituted the bottleneck. In the second experiment, adding a second middleware, and thus doubling the amount of worker threads present in the system, has further increased the throughput and decreased the response time, which confirms our statement. With 64 worker threads however, we start getting close to the maximum throughput of the server as measured in section 2.1. We concluded that adding even more worker threads would further improve throughput and decrease response time, although only slightly as the server is already saturated. After exhausting the server, adding a second server would be the best choice to increase system performance for get requests, for replicated set jobs, section 4 will go into detail.

For write-only workloads, the amount of worker threads are the most important parameters for the performance of the system. In a saturated configuration, adding more worker threads improves the throughput and decreases the response time, because requests need to wait less long for a worker thread to be available and more requests can be served at the same time. For an unsaturated configuration, this will only have a negligible impact since the resources available are already enough to optimally serve the available workload.

For read-only workloads, only the virtual client count makes a difference in system perfomance. The response time is lower with fewer virtual clients, but the throughput is always the same, so the lowest possible virtual client count should be used.

4 Throughput for Writes (90 pts)

The goal of these experiments is testing the effect data replication will have on the system. Here, three servers are used and only a write-only workload is tested. For a number of different thread counts for the middleware(s), we vary the amount of virtual clients to extensively test the performance of the system.

4.1 Full System

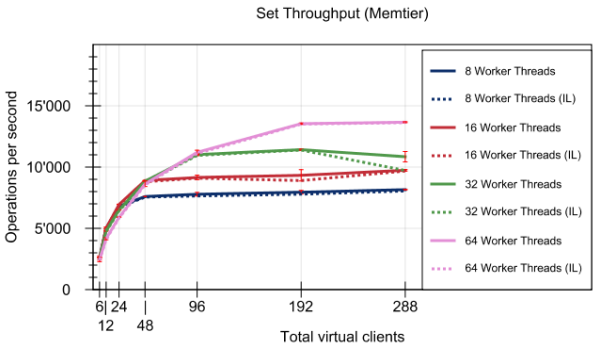
In this experiment, we test a full system: Three load generating memtier clients are connected to two middlewares. Both are connected to a total of three servers, replicating set requests over all of them and collecting and eventually relaying the result back to the client. The clients are run on the same hosts as before, as are the servers and middlewares. For four different amounts of worker threads used in the middleware, we'll vary the amount of virtual clients connected to it. In this baseline experiment, we test the performance of the middleware.

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1, 2, 4, 8, 16, 32, 48]
Workload	Write-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8, 16, 32, 64]
Repetitions	3 (70 seconds each)

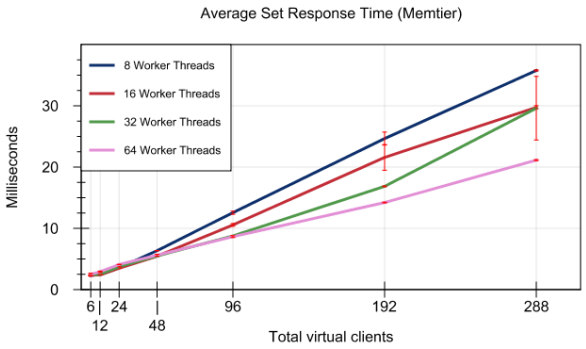
4.1.1 Explanation

Sanity check The throughput and response time graphs match well between the middlewares and the clients. The interactive law holds, and where it is not accurate, the same reason holds as before (network anomalies, see 2.1.1). As expected, all response times are higher in the memtier clients than in the middlewares. In queue graphs, never are there more jobs in the queue than there are virtual clients in the system.

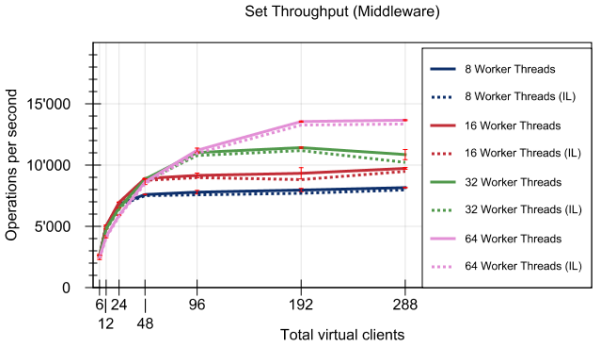
Analysis The same saturation points apply as with write-only experiments in section 3, with the same knees visible in the response time graph. The knees are again visible on the memtier clients and the middlewares. We notice that the highest encountered throughput at 64 workerthreads and 288 virtual clients is with 13'662 ops/s approximately 13% lower than in 3.2 with 15'638 ops/s. The difference to section 3.2 is that we are using three servers. However, since we only have a write-only workload, we need to take into account that writes need to be replicated across all servers. Since in 3.2 the memcached servers were not the bottleneck, we assume them to neither be the bottleneck in this case. Checking the average memcached cpu utilitation reveals it to never exceed 84%, confirming our assumption. Also, bandwith utilization is not close to its maxiumum for any of the hosts. We again take a look at the processing percentage graph (as before, the errorbars are artificial and only based on the processing time errors). Figure 18a reveals that the utilization reaches 100% for all worker thread configurations once saturation is reached. The queue size concurs with the processing percentage: Before reaching maximum worker thread utilization, queue sizes slowly increase as we add more virtual clients. However, once full utilization is reached, adding further clients will just raise queue size and, of course, response time, as requests will need to wait longer before being processed.



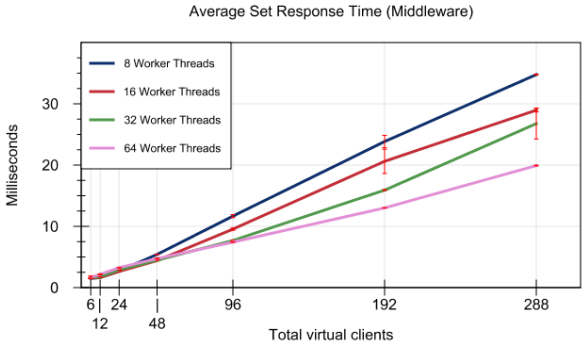
(a) Accumulated average write-only throughput (measured in the clients)



(b) Average write-only response time (measured in the clients)



(c) Accumulated average write-only throughput(measured in the middlewares)

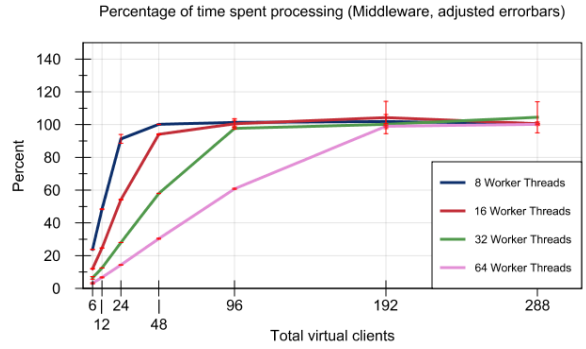


(d) Average write-only response time (measured in the middlewares)

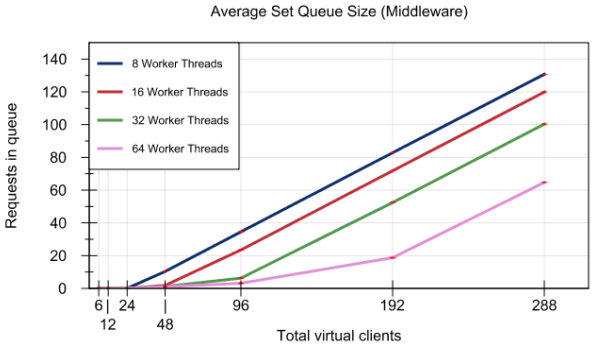
Figure 17

We noted how, in contrast to 3.2, we have lower throughput in this experiment. The response time is consistently higher too (with the exception of seldom outliers due to anomalies). We observe for the 64 worker thread configuration (which was chosen because it is stable in both experiments) the following increases: 1.71ms to 2.5ms, 2.58ms to 2.94ms, 3.24ms to 4.1ms, 4.58ms to 5.61ms, 7.41 ms to 8.64, 12.66ms to 14.21ms and 18.46ms to 21.14. The increases are as follows: 0.79ms, 0.36ms, 0.86ms, 1.03ms, 1.23ms, 1.55ms, 2.68ms. The reason why the response time is higher is that the replication is sequential. The worker thread will write to one server after the other before collecting the results and sending a response back to the client. To confirm this, we take into account the worker time, which has been defined as the timeframe from dequeuing a request until the request is sent to the last server. Figures 18d and 18c show the worker time per request for this experiment and experiment 3.2 respectively. As can be noticed, the values lie consistently lower for section 3.2. The worker time is slightly higher for more worker threads because the server is more saturated as explained in 3.2.1, increasing the response time for requests sent to the middleware. The worker time in 4.1 is throughout significantly higher (most often more than twice as high), meaning that the worker threads spend way more time sending the request to the servers than for unreplicated set requests in 3.2. One may assume the worker time increase at least threefold, but this is not the case because the worker time does neither include request parsing time (which we assume to be consistent, since get requests messages are very similar in size and parsing is byte by byte) nor the time spent receiving replies or parsing them (parsing responses should be very short since we just

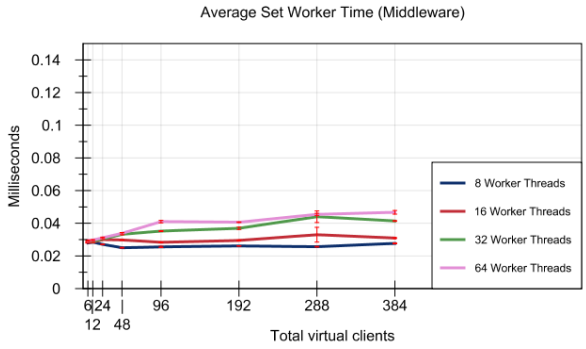
compare 8 bytes). Now, because the average latency to the server is significantly higher than the time it takes writing to all of them (1 ms latency vs. approximately 0.1 ms for sending) and we write to all servers before waiting for the first reply, we can conclude the overhead replication introduces to be significant but not overburdening, which is made apparent by the throughput only slightly decreasing (13% for the highest throughput) compared to section 3.2. So, we can pinpoint the replication factor to be responsible for the increase in response time. As the interactive law confirms, the effect of the higher response time is consequently a reduction in throughput.



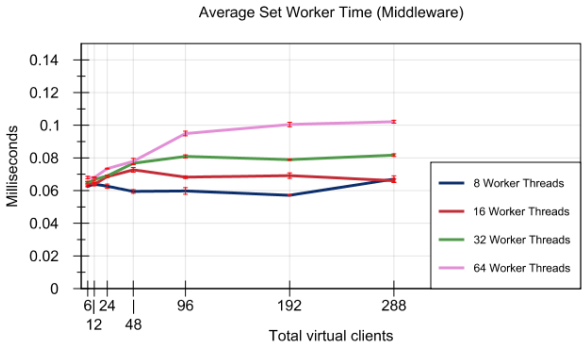
(a) Average time percentage worker threads spend processing jobs



(b) Average queue size



(c) Average worker time per job **from section 3.2** for a write-only workload



(d) Average worker time per job from section 4.1

Figure 18

4.2 Summary

For the maximum throughput table below, each worker thread configuration's saturation point has been chosen, as throughput stays similar after but the response time increase grows.

Maximum throughput for the full system

	WT=8, VC=24	WT=16, VC=48	WT=32, VC=96	WT=64, VC=192
Throughput (Middleware)	6'821 ops/s	8'911 ops/s	11'022 ops/s	13'552 ops/s
Throughput (Derived from MW response time)	6'707 ops/s	8'756 ops/s	10'782 ops/s	13'264 ops/s
Throughput (Client)	6'812 ops/s	8'907 ops/s	11'019 ops/s	13'543 ops/s
Average time in queue	0.54 ms	1.01 ms	1.99 ms	3.63 ms
Average length of queue	0.05 req.	1.82 req.	6.28 req.	18.74 req.
Average time waiting for memcached	1.7 ms	2.67 ms	4.55 ms	7.86 ms

The purpose of this experiment was testing what effect replicating operations over a number of servers has. As expected, the replication makes worker threads take longer to write to servers and collect replies. This increase in response time leads to the decrease in throughput, because the middleware worker threads are fully utilized.

As for the bottleneck, we have shown that again the worker thread count is holding back system performance. Although memcached server cpu utilization reaches 84%, experiment 2.1 has shown that the throughput can be increased further. Average cpu utilization on the middlewares does never exceed 23%, so we conclude that increasing the number of worker threads would boost overall performance, until the memcached servers are exhausted and they become the bottleneck. This will happen before the upload rate limit of the clients will be hit as it did not exceed 73% in this experiment.

In this experiment, again, the most important parameter for system performance is the amount of worker threads present, as it constitutes the bottleneck. This can be observed in the table. Having more worker threads available will mean that requests have to spend less time in the queue on average, because the probability of a worker thread being available is higher because the workload per thread is lower, thus reducing response time and increasing throughput. When fixing the amount of worker threads, especially when saturation is reached (as is the case with the table above), adding more virtual clients will introduce higher queuing times, as for the same amount of worker threads more requests have to be worked on and thus requests have to wait longer for a worker thread to be available. For undersaturated systems the difference will be smaller since there is an abundance of worker threads available for the given amount of requests. Note that in the table, the queue size and time increases with more worker threads because we have increased the amount of virtual clients as well.

5 Gets and Multi-gets (90 pts)

In these experiments, we explore the behaviour of multi-get requests using the full system configuration. Three client hosts each run 2 instances of memtier, and they are connected to two middlewares such that each client host is connected to both of them. Both middlewares are connected with all three servers.

The memcached protocol allows reading multiple keys per request, which can be either be sharded by the middleware processing it or not. Sharding means that the request containing n keys will be split up into multiple requests, each of which will be sent to a different server, containing $<= n$ keys. The middleware then collects the responses and replies to the clients with one message containing all values for the keys requested. Not sharding a multi-get request

means that the middleware just forwards the entire request to a server and relays the response to the client, similar as in gets only requesting one key. In this experiment we test get-requests containing 1, 3, 6 and 9 keys. For all configurations tested we use a total of 12 virtual clients and only vary the multi-get size.

In previous read-only experiments, we have always hit the same bottleneck, namely the upstream bandwith limitation on the servers. For all configurations we immediately were thwarted when trying to check the interactions of client counts and worker thread counts, as all experiments delivered the same results. Therefore, there is no real maximum throughput configuration to choose for this experiment. Since we know that using more worker threads help keep queuing times and sizes low, and that for write-only workloads this parameter was the bottleneck, we use the 64 worker thread configuration.

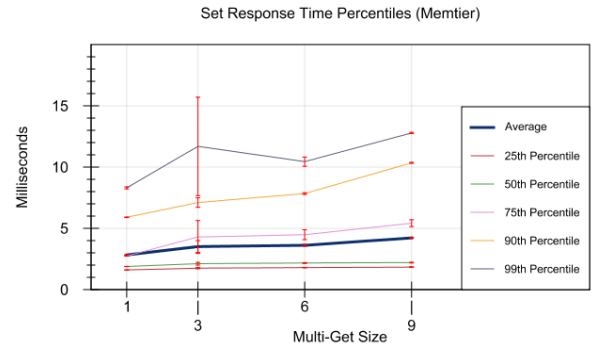
5.1 Sharded Case

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	2
Workload	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 (70 seconds each)

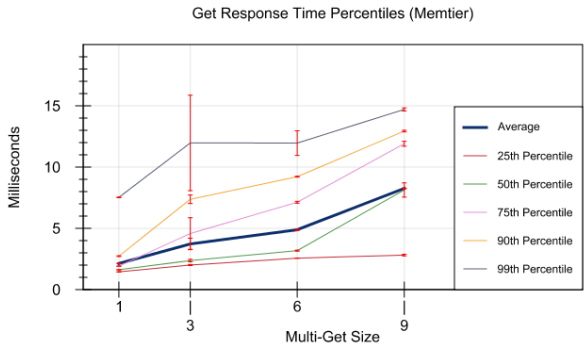
5.1.1 Explanation

Sanity check Both response time graphs show higher percentiles always consistently having higher response times than lower ones. The average is between the median and the 75th percentile. When response times grow, they grow for all percentiles similarly. Error bars are always visible in both graphs, since the data comes from the same experiment. We can again attribute the instabilities to network anomalies on the azure platform as in previous sections. The throughput of sets and gets matches perfectly, as expected, because memtier sends out the same amount of them because of its ratio setting.

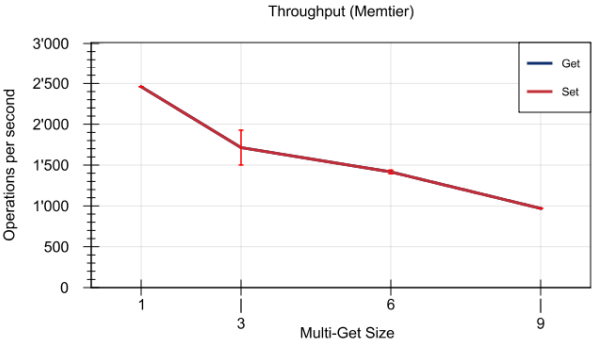
Analysis Since all read-only experiments before this one have had the same bottleneck, namely the bandwidth usage, we have a look at figure 19d to check this metric for this experiment. We notice that the bandwidth is only exhausted (and thus constitutes the bottleneck) when using 6- and 9-keyed multi-gets. The reason for this is that the reply size scales with the value count contained. In this experiment, we are using three servers, so we have 300 Mbit/s upstream available from them in total. For single- and 3-keyed gets, the server's reply load generated with 12 total virtual clients is not enough to have them reach their rate limits. Using 1-key gets, worker threads send one request to a server and wait for the reply. With 3-key gets, because we use sharding, they send a 1-key request to each server and wait for a reply from every server. This increases the bandwidth utilization of the servers, although not threefold as one may suspect: The reason for this is that the throughput has reduced, meaning that less requests get sent in total. This also aligns with the increase in response time in figure 19b: When increasing the key count, the middleware takes longer to reply because of the sharding, thus increasing the response time and lowering throughput. So, we conclude the bottleneck to be the number of virtual clients in the system. Increasing them will increase the workload in the system, which all systems can handle: The worker thread's time is spent less than 2.7%



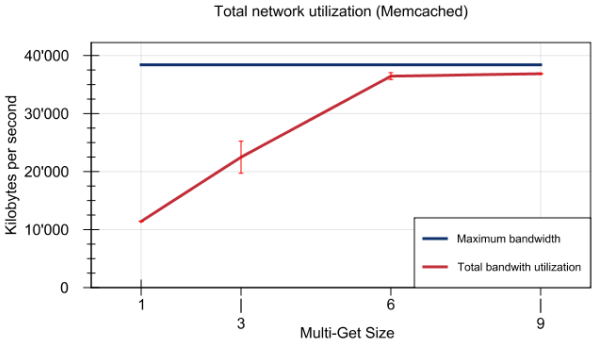
(a) Response time average and percentiles for sets as measured in the memtier clients (sharding on)



(b) Response time average and percentiles for gets as measured in the memtier clients (sharding on)



(c) Accumulated average throughput as measured in the clients (sharding on)



(d) Total bandwith utilization for sharded multi-gets (sharding on)

Figure 19

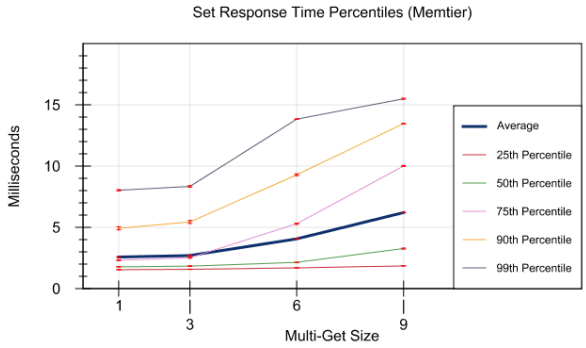
and 4.1% processing requests for multi-get sizes of 1 and 3 respectively and the server's cpu utilization never exceeds 42%. However, with multi-get sizes 6 and 9, the servers' rate limitation will again become the bottleneck.

Note that the response time increases less drastically for set requests in figure 19a. One might suspect the response time of sets to not be affected at all by the key count, but this is not the case. This is because memtier's workload is set to 1:<Multi-Get size>. This means that memtier will send out exactly one set request for each multi-get request. Whenever a set request is queued at a memcached server that is currently working on a get, the set will have to wait until the get request is done. This takes significantly longer than a set, because the memcached server is single-threaded (meaning it cannot handle multiple requests in parallel) and the upload rate limitation means the time a server takes to send back get replies is not negligible. This slightly increases the response time, as is seen in the figure. With more keys used, the server will take even longer to reply because its reply will be twice or three times as big (for 6-key multi-gets from clients gets it has to send back two values, for 9-key multi-gets from clients is has to send back three values). The linearity of this is visible in the slope of the set response time.

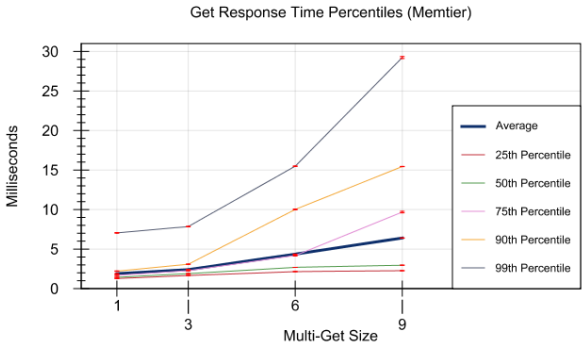
The percentiles in the response time graph show that the average is higher than the median, implying that most requests took less long than the average and that outliers with high response times skew the average to a higher value.

5.2 Non-sharded Case

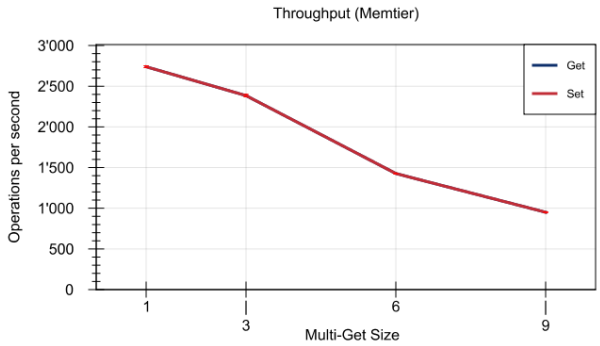
Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	2
Workload	ratio=1:<Multi-Get size>
Multi-Get behavior	Non-Sharded
Multi-Get size	[1, 3, 6, 9]
Number of middlewares	2
Worker threads per middleware	64
Repetitions	3 (70 seconds each)



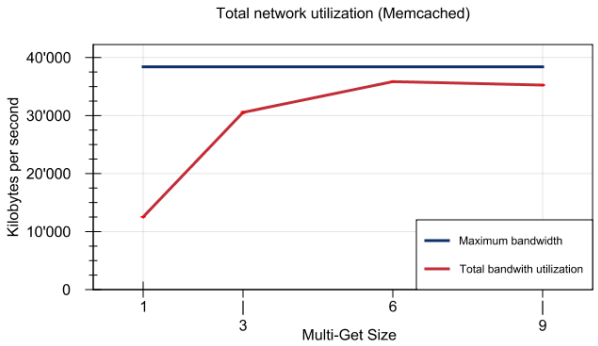
(a) Response time average and percentiles for sets as measured in the memtier clients (sharding off)



(b) Response time average and percentiles for gets as measured in the memtier clients (sharding off)



(c) Accumulated average throughput as measured in the clients (sharding off)



(d) Total bandwith utilization for non-sharded multi-gets (sharding off)

Figure 20

5.2.1 Explanation

Sanity check We use the same sanity checks as in 5.1 and see that all checks hold, for the same reasons.

Analysis In this section, the only change from 5.1 is that we have disabled sharding. We first again have a look at the bandwidh utilization in figure 20d. The utilization is the same for one key, as sharding does not make a difference (this is also confirmed when looking at all other graphs which yield virtually identical results for one key). For three keys however, the bandwidth utilization is higher with sharding disabled. For 6 and 9 keys, the rate limit is met

(and thus, again, the bottleneck is the upload rate limit of the servers for 6 and 9 keys). The reason the bandwidth utilization is higher with 3 keys is that the response time is lower. That is because sharding introduces an overhead in the middleware when splitting the request: It sends the data sequentially and has to parse more incoming replies, which here falls away. Since here the system is not bottlenecked by bandwith limitations, this change is also reflected in the throughput graph. However, for multi-get sizes 6 and 9, the bottleneck is the bandwidth limitation. Here, for the same throughput we have an increase in response time, which we can also attributed to sharding overheads in the middleware.

With sharding turned off, we see that the influence of higher response times in gets affects the response times of sets even more than in 5.1, increasing them even more for higher multi-get sizes. The reason is the same, however, this time, sets queued at the memcached servers need to wait even longer because responses are way longer because they contain more values, taking longer to be sent back and thus increasing set response times significantly.

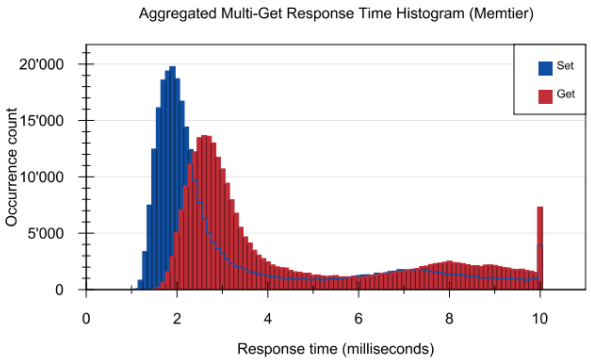
We now can compare the response time percentiles to the ones from section 5.1. Only the 90th and the 99th percentile of multi-get sizes of 6 and 9 are lower with sharding enabled, all other response time percentiles are higher. This means that there are more outliers with high response times when we disable sharding. The reason for this is that, with sharding enabled, there are more messages being sent from the servers to the middlewares, meaning that the connections are used for short intervals when compared to not using sharding, where one big message will be sent for a longer amount of time. Since we operate at the network's bandwidth limit, using one connection longer is more stable thus and produces shorter response times.

5.3 Histogram

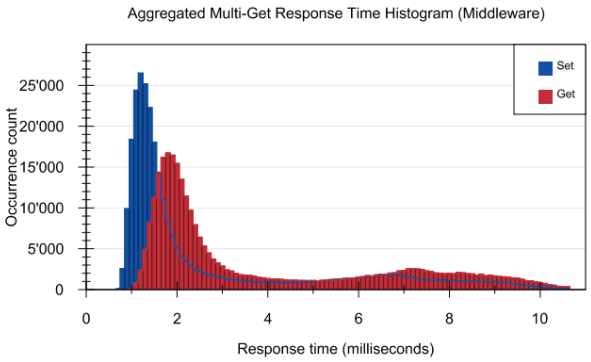
We now present four histograms (figure 21), showing the distribution of response times for sharded and non-sharded multi-gets as measured in the middleware and the memtier clients. For all histograms, outliers above the 99th percentile have been cut off and added to the last bucket in the histogram. The bucket size is 0.1 ms for all histograms. For histograms measured on the memtier clients, the resolution of the cdf in low files emmited by the memtier clients used to calculate them lowers to 1 ms buckets after a measured response time of 10 ms, which is why gaps appear and the values of buckets after 10 ms are higher (they contain all occurrences between them and the bucket 1 ms later).

To conduct a sanity check of the plots, the total amount of operations have been compared as a part of rendering. The difference in amounts of sets and gets is less than 0.01% for all pairs. The difference in amounts between middleware and client is less than 0.3% for all pairs. We also see that the shapes are similar between memtier clients and the middleware, but that the whole graph looks as if shifted left when comparing the middleware's with the client's graph, which again confirms that response times are throughout higher for the memcached clients.

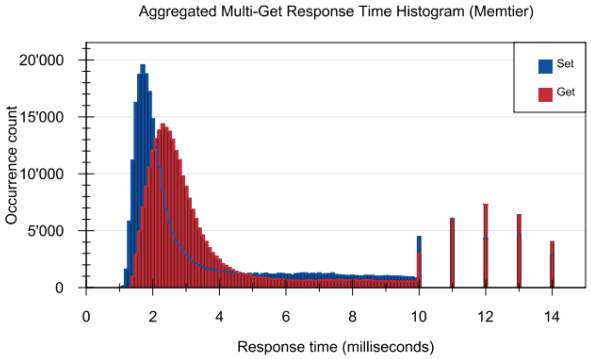
These histograms show the distribution of response times when using 6-keyed multi-gets both with sharding enabled and disabled. They confirm our previous findings as we can see again that sharding increases the response time. The histograms also align with the percentiles shown in the response time graphs. Note that all histograms contain the warmup and cooldown phase. Memtier's cdf cannot be configrued, and to have all histograms be comparable we did not exclude the phases from the middleware data either for the histograms. However, we assume the phases to be negligible because higher response times (i.e. outliers) are more likely to occur once the load is high and the systems saturated.



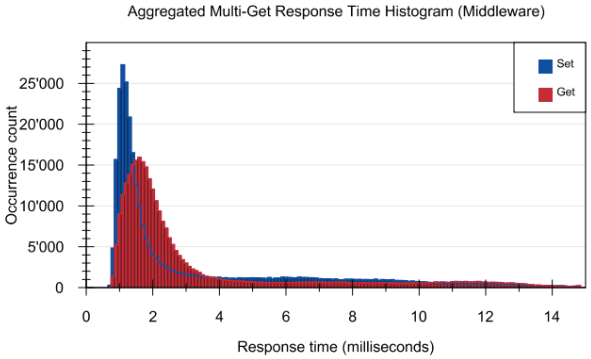
(a) Aggregated response time histogram with sharding enabled, as measured on the clients



(b) Aggregated response time histogram with sharding enabled, as measured on the middlewares



(c) Aggregated response time histogram with sharding disabled, as measured on the clients



(d) Aggregated response time histogram with sharding disabled, as measured on the middlewares

Figure 21

5.4 Summary

The purpose of this experiment was testing what influence sharding has on the performance of multi-gets. The naive assumption is that dividing big jobs into small parts would reduce response time, but we found the contrary to be the case. Sharding introduces an overhead in the middleware as it actively has to create several new requests out of one single request. Then, parsing three instead of just one response and checking them for errors and putting together the response for the client again takes time. It has been observed to lower response times, and thus be more efficient, to just relay the multi-get request untouched to one server and sending the response back as-is to the clients.

We have seen that increasing the multi-get size increases response time and decreases throughput, both when using sharding and when not. It has to be noted that using more keys in a get request also means a client has to send fewer requests to receive the same amount of values back. In that sense, using a higher key count can make sense in general because it delivers more values at the price of a higher response time. When looking at the amount of values read per second, multi-get sizes 6 and 9 deliver about 8'500 values per second, both with and without sharding. However, multi-get size 9 will have higher response time and thus makes no sense to be used. For multi-get size 3, disabling sharding increases the values per second from 5'142 to 7'152. For multi-get size 1, sharding does not make a difference. We conclude that for all multi-get sizes sharding should be disabled for lower response times and higher throughput. The reason is that sharding will introduce a lot of work for the middleware like

sending more messages, parsing more responses and putting together a response for the client. By just relaying messages this overhead falls away and response times are lower.

The most important parameter for performance in this experiment is sharding, since disabling it improves throughput significantly for a multi-get size of 3 keys. For all other key sizes, sharding has a negative effect. Although low, it is not negligible. The multi-get size has a big impact on performance too, but less so on the value per second value, which needs to be considered here.

We have seen in the histograms how the response times are distributed. As expected, the shapes between middleware and memtier measurements were the same. However, the entire figure seems shifted to the left in the middleware plot, because sending back the responses to the clients always takes similar time. We see that sets are closer together and thus have less variance as get requests, for all histograms, whereas gets are more spread, being higher in variance. This aligns with our observations.

6 2K Analysis (90 pts)

Number of servers	1 and 3
Number of client machines	3
Instances of memtier per machine	1 (1 middleware) or 2 (2 middlewares)
Threads per memtier instance	2 (1 middleware) or 1 (2 middlewares)
Virtual clients per thread	32
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1 and 2
Worker threads per middleware	8 and 32
Repetitions	3 (70 seconds each)

We now perform a $2^k r$ analysis with the following variables: x_A is -1 for 1 server and 1 for 3 servers, x_B is -1 for 1 middleware and 1 for 2 middlewares, x_C is -1 for 8 worker threads and 1 for 32 worker threads. We use an additive model for both types of workloads. The reason for this is that the previous experiments (such as 3.1) have shown that the effects on throughput/response time are linear both when adding middlewares and when adding more worker threads. We use the following model:

$$y = q0 + q_A x_A + q_B x_B + q_C x_C + q_{AB} x_A x_B + q_{AC} x_A x_C + q_{BC} x_B x_C + q_{ABC} x_A x_B x_C + e$$

6.1 Write-only

I	x_A	x_B	x_C	TP (ops/s)	TP (mean)	RT (ms)	RT (mean)
1	-1	-1	-1	6738.6, 6918, 6454.82	6703.8	27.73, 27.79, 29.9	28.47
1	-1	-1	1	10121.45, 10200.7, 10184.16	10168.77	18.99, 18.84, 18.87	18.9
1	-1	1	-1	9139, 9031.47, 9154.97	9108.48	22.65, 21.31, 21.02	21.66
1	-1	1	1	13040.97, 13029.8, 13117.87	13062.88	14.78, 14.79, 14.68	14.75
1	1	-1	-1	4566.43, 4742.19, 4619.86	4642.83	42.15, 40.57, 41.65	41.46
1	1	-1	1	7783.88, 7891.78, 7861.7	7845.65	24.68, 24.34, 24.43	24.48
1	1	1	-1	7678.46, 7661.31, 7677.81	7672.53	25.08, 25.13, 25.09	25.10
1	1	1	1	10836.92, 11010.77, 10919.31	10922.33	17.76, 17.46, 17.82	17.68

Table 1: Experimental results for write-only workloads

We can see that parameter C, the number of worker threads, influences the throughput and response time the most with 49.55% and 43.06%, followed by parameter B, the number of middlewares with 33.49% and 29.99%. Parameter A, the server count, contributes to the variations with 16.32% and 16.03%. All interactive effects contribute insignificantly to the

	Mean est.	Variation expl.	Summed squares	90% Cofidence int.
SST	N/A	100%	145645745.11	N/A
q_0	8765.91	N/A	1844187622.84	[8729.84, 8801.98]
q_A	-995.07	16.32%	23764115.73	[-1031.15, -959]
q_B	1425.65	33.49%	48779201.2	[1389.57, 1461.72]
q_C	1734	49.55%	72162165.42	[1697.93, 1770.07]
q_{AB}	100.95	0.17%	244574.93	[64.88, 137.02]
q_{AC}	-120.84	0.24%	350469.06	[-156.92, -84.77]
q_{BC}	67.05	0.07%	107901.34	[30.98, 103.12]
q_{ABC}	-55.31	0.05%	73408.18	[-91.38, -19.23]
SSE	N/A	0.11%	163909.23	N/A

Table 2: 2K analysis results for write-only throughput

	Mean est.	Variation expl.	Summed squares	90% Cofidence int.
SST	N/A	100%	1455.95	N/A
q_0	24.06	N/A	13896.08	[23.9, 24.23]
q_A	3.12	16.03%	233.33	[2.95, 3.29]
q_B	-4.27	29.99%	436.69	[-4.43, -4.10]
q_C	-5.11	43.06%	626.94	[-5.28, -4.94]
q_{AB}	-1.53	3.83%	55.82	[-1.69, -1.36]
q_{AC}	-0.99	1.61%	23.46	[-1.16, -0.82]
q_{BC}	1.53	3.84%	55.91	[1.36, 1.69]
q_{ABC}	0.86	1.22%	17.79	[0.69, 1.03]
SSE	N/A	0.41%	6.01	N/A

Table 3: 2K analysis results for write-only response times

performance, although more so for throughputs. In previous sections we stated that the amount of worker threads in the system is the most important perfomance factor because it is the bottleneck. Parameter B can double the amount of worker threads and paramter C can even increase them by factor 4, which aligns with our previous findings. Both parameters raise the throughput and lower response times significantly, parameter C more intensly because the effects is increased with even more worker threads. We notice that effect A contributes negatively to the perfomcance. Adding a server forces middlewared to replicate their sets to all of them, increasing reponse time and thus reducing throughput, which again aligns with our findings from section 4.1.

6.2 Read-only

I	x_A	x_B	x_C	TP (ops/s)	TP (mean)	RT (ms)	RT (mean)
1	-1	-1	-1	2930.49, 2574.51, 2926.5	2810.5	65.47, 89.26, 65.57	73.43
1	-1	-1	1	2932.73854, 2937.18, 2931.12	2933.68	65.43, 65.33, 65.47	65.41
1	-1	1	-1	2938.07, 2937.76, 2939.9	2938.58	65.37, 65.38, 65.3	65.35
1	-1	1	1	2936.32, 2939.95, 2936.39	2937.55	65.53, 65.43, 65.5	65.49
1	1	-1	-1	6978.52, 7038.85, 7100.91	7039.43	27.56, 27.32, 27.05	27.32
1	1	-1	1	8788.26, 8786.62, 8781.47	8785.45	21.85, 21.85, 21.86	21.85
1	1	1	-1	8804.25, 8801.19, 8797.47	8800.97	21.82, 21.83, 21.85	21.83
1	1	1	1	8804.9, 8807.99, 8821.12	8811.33	21.82, 21.8, 21.77	21.8

Table 4: Experimental results for read-only workloads

We see that parameter A overrules all other parameters by a landslide. For both throughput and response time, all other parameters and interactions between them are insignificant (below 1%). This again aligns with our previous findings. Recall that for all previous read-only experiments (such as 3.2), the network rate limitation imposed the bottleneck. We work here with a total of 192 virtual clients, which exhausts the network rate limit. In 2.2 we were able to double throughput by adding a second server, because that doubled the rate limit and thus the

	Mean est.	Variation expl.	Summed squares	90% Cofidence int.
SST	N/A	100%	185587392.17	N/A
q_0	5632.19	N/A	761316758.85	[5611.58, 5652.8]
q_A	2727.11	96.18%	178491078.90	[2706.5, 2747.72]
q_B	239.92	0.74%	1381501.61	[219.31, 260.53]
q_C	234.82	0.71%	1323350.78	[214.21, 255.43]
q_{AB}	206.93	0.55%	1027722.45	[186.32, 227.55]
q_{AC}	204.28	0.54%	1001516.33	[183.67, 224.89]
q_{BC}	-232.48	0.7%	1297162.19	[-253.09, -211.87]
q_{ABC}	-201.43	0.52%	973801.2	[-222.04, -180.82]
SSE	N/A	0.05%	91258.62	N/A

Table 5: 2K analysis results for read-only throughput

	Mean est.	Variation expl.	Summed squares	90% Cofidence int.
SST	N/A	100%	12320.53	N/A
q_0	45.31	N/A	49270.19	[43.99, 46.63]
q_A	-22.11	95.23%	11732.24	[-23.43, -20.79]
q_B	-1.69	0.56%	68.71	[-3.01, -0.37]
q_C	-1.67	0.54%	67.14	[-3, -0.35]
q_{AB}	0.31	0.02%	2.28	[-1.01, 1.63] (ins.)
q_{AC}	0.3	0.02%	2.13	[-1.03, 1.62] (ins.)
q_{BC}	1.7	0.56%	69.33	[0.38, 3.08]
q_{ABC}	-0.34	0.02%	2.8	[-1.66, 0.98] (ins.)
SSE	N/A	3.05%	375.90	N/A

Table 6: 2K analysis results for read-only response times

amount of responses the servers could send back to the middlewares. This happens here again, meaning that parameter A has the highest contribution in variation. As the other parameters don't change the bottleneck and thus neither the throughput, their contribution to the variation here can be observed to be insignificant. As for response time, adding a second server also has a significant effect, which we also could observe in section 2.2: With twice the server performance available for the same amount of clients, a client needs to wait significantly less long.

7 Queuing Model (90 pts)

7.1 M/M/1

Here, we model the entire system as a single M/M/1 queue. The inputs are λ, the arrival rate, and μ, the service rate. As the service rate we will use the highest ever encountered throughput for each individual worker thread configuration, because we know the can system handle that amount of requests. For the arrival rate we use the average throughput (we use the maximum throughput table 4.2 from section 4.1 for the corresponsing worker thread configuration). Because that value is lower than the service rate, the utilization $\rho = \lambda/\mu$ will be < 1 and thus the system stable. Using the model, we calculate the service time $E[s] = 1/\mu$, the mean number of jobs in the system $E[n] = \rho/(1-\rho)$, the mean number of jobs in the queue $E[n_q] = \rho^2/(1-\rho)$, the mean response time $E[r] = (1/\mu^2)/(1-\rho)^2$, and the mean waiting time $E[w] = \rho\frac{1/\mu}{1-\rho}$.

We can see that the model is significantly off. It treats the entire system as a black box with only one queue and one worker. In reality, we have one queue per middleware and many workers polling them in parallel. The service time in the model decreases with more worker threads, whereas in reality it increases because using more worker threads will lead to higher wait times for the server responses, but the model does not model multiple workers and is thus inaccurate. The mean number of jobs is inconsistent, as it is higher with 16 worker threads than with 32. This is because the utilization is slightly higher with 32 worker threads, skewing

	8 WT		16 WT		32 WT		64 WT	
λ	6821		8911		11022		13552	
μ	8152		9153		11422		13662	
	mod.	meas.	mod.	meas.	mod.	meas.	mod.	meas.
ρ	0.84	N/A	0.97	N/A	0.96	N//A	0.99	N/A
$E[s]$ (ms)	0.12	3.02	0.11	4.44	0.09	6.78	0.07	10.58
$E[n]$	5.12	24	36.82	48	27.56	96	123.2	196
$E[n_q]$	4.29	0.05	35.85	1.82	25.59	6.28	122.21	18.74
$E[r]$ (ms)	0.75	3.56	4.13	5.45	2.5	8.77	9.01	14.21
$E[w]$ (ms)	0.63	0.54	4.02	1.01	2.41	1.99	9.02	3.63

Table 7: Results of the M/M/1 model vs. actual measurements

the model. The queue size is throughout way too low in the model, which may be linked to the formula deriving it, where we divide by values close to zero, producing unstable results. The response time is way too low as well and again breaks the trend between 16 and 32 worker threads, however it is somewhat close for 16 and 64 worker threads. The average queue time is always higher than measured values and again breaks the trend of increasing between 16 and 32 worker threads.

7.2 M/M/m

Here, we again build a model, still with one queue but with multiple workers polling it in parallel. The inputs are again λ, the arrival rate, μ, the service rate and additionally m, the amount of service nodes. We choose $m = 2 \cdot WT$, which is the total amount of workers in the system (there are two middlewares). We choose μ and λ similarly as before as the highest encountered throughput for the worker thread configuration (divided by m, since it concerns one service only) and the maximum throughput configuration as defined in table 4.2. We again have the utilization $\rho = \frac{\lambda}{m \cdot \mu}$ guaranteed to be less than 1, meaning the system is stable. We calculate the same parameters as before, with formulas adjusted to the model as follows: $E[s] = E[r] - E[w]$, $E[n] = m\rho + \frac{\rho\varrho}{(1-\rho)}$, $E[n_q] = \frac{\rho\varrho}{(1-\rho)}$, $E[r] = \frac{1}{\mu}(1 + \frac{\varrho}{m(1-\rho)})$, $E[w] = E[n_q]/\lambda$ with the probability of queueing $\varrho = \frac{(m\rho)^m}{m!(1-\rho)}p_0$ and the probability of zero jobs in the system $p_0 = \left[1 + \frac{(m\rho)^m}{m!(1-\rho)} + \sum_{n=1}^{m-1} \frac{(m\rho)^n}{n!}\right]^{-1}$.

	8 WT		16 WT		32 WT		64 WT	
m	16		32		64		128	
λ	6821		8911		11022		13552	
μ	509.5		286.03		178.47		106.73	
	mod.	meas.	mod.	meas.	mod.	meas.	mod.	meas.
ρ	0.84	N/A	0.97	N/A	0.96	N//A	0.99	N/A
ϱ	0.4	N/A	0.83	N/A	0.7	N/A	0.89	N/A
p_0	0	N/A	0	N/A	0	N/A	0	N/A
$E[s]$ (ms)	1.96	3.02	3.5	4.44	5.6	6.78	9.37	10.58
$E[n]$	17.09	24	34.86	48	65.47	96	130.68	196
$E[n_q]$	0.3	0.05	3.43	1.82	1.74	6.28	8.11	18.74
$E[r]$ (ms)	2.65	3.56	6.93	5.45	7.34	8.77	17.48	14.21
$E[w]$ (ms)	0.3	0.54	3.43	1.01	1.74	1.99	8.11	3.63

Table 8: Results of the M/M/m model vs. actual measurements

This model presents us with a significant improvement over the M/M/1 model. However, there are still differences to the real system, an important of which is that service nodes do not share the server- or the network-resources, meaning that the model again is oversimplified. We see that the service time is lower than the measured values but is close and becomes closer the more worker threads are present in the system. The average amount of jobs in the system is

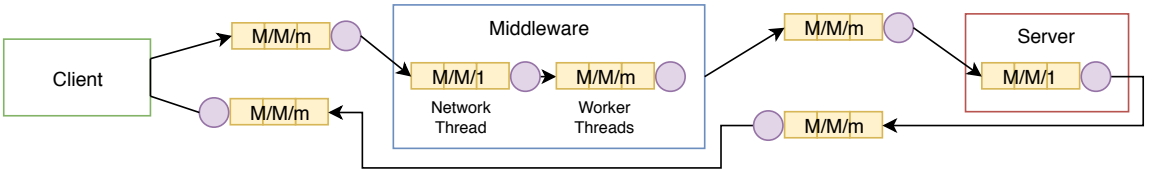


Figure 22: Model overview

again too low but the trend is correct. As for the queue size, we notice again that the trend is broken between 16 and 32 worker threads (the reason being again that the utilization is higher for 16 worker threads). It is too high for 8 and 16, but too low for 32 and 64. This is because the model assumes service nodes can complete jobs independent of how many service nodes there are, but in reality, worker threads will need to wait longer when more worker threads are present because the servers will be more congested. We can observe that the response time predicted by the model is close to the real values, alternately being too low and too high. The average queue time also fluctuates but is not that much off, except for 64 worker threads. The reason is again, that the model is oversimplified with respect to resource sharing between worker threads/service nodes.

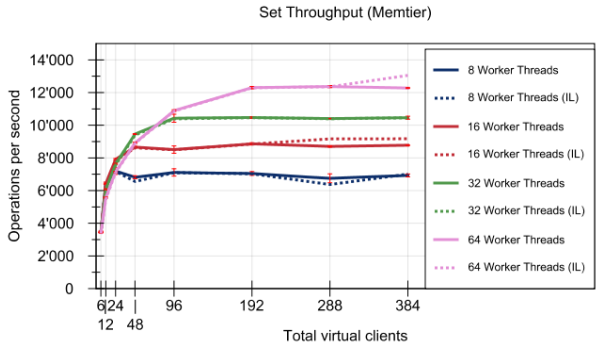
7.3 Network of Queues

To accurately model the system for section 3, we need multiple different components. Previous models only assumed one component with one queue, here we use a more intricate network of multiple such components to model individual aspects of the system more accuratley. See the model chosen in figure 22.

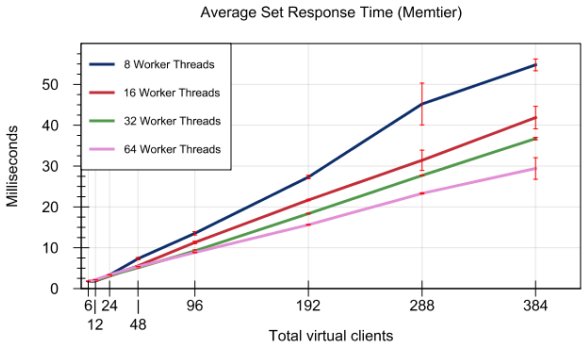
We model our system as follows: The clients will be modelled only as the two network links, because jobs never get queued or need to wait in memtier clients. We have one M/M/m model for both up-and download, which we will choose appropriately to mimic the bandwidth limitations present on the links in our real system (Since the limitations differ for the directions, we choose two different service times). The middlewares will be represented with an M/M/1 model and an M/M/m model. The first one mimics the network thread, which is single threaded, the second one mimics all worker threads, i.e. we use the model from 7.2. For the server, using a M/M/1 model seems to be the best fit, as memcached is using only one thread. The network will again be modelled as two M/M/m components to accurately capture the bandwith limitations, and again we choose different service rates to reflect the different (up- and down-stream) limitations. Now, there are several advantages and disadvantages when choosing this system. In this model, contrary to the real implementation, middleware worker threads will not be blocked until receiving a response from the server. This behaviour has been observed to have significant influences over the performance. One possible solution is modelling the server and the middlewares together as M/M/m components, this has another drawback though, since the increase in server response times with more workers in the middleware cannot be captured.

8 Appendix

8.1 Section 3

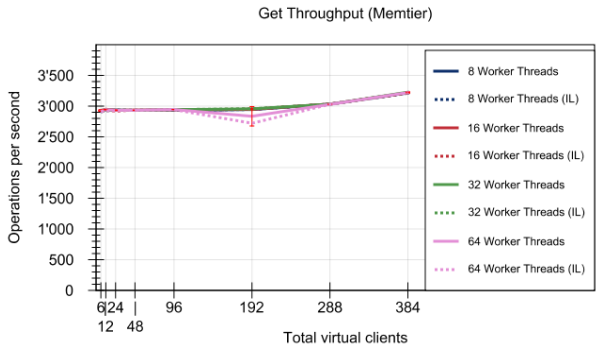


(a) Section 3.1, accumulated average write-only throughput (measured in the clients)

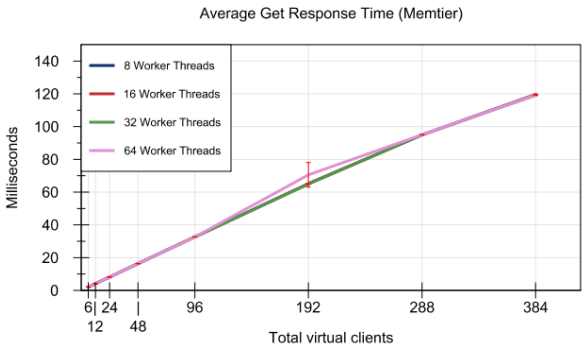


(b) Section 3.1, average write-only response time (measured in the clients)

Figure 23

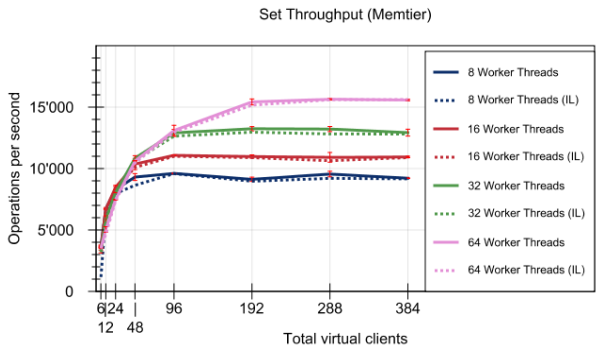


(a) Section 3.1, accumulated average read-only throughput (measured in the clients)

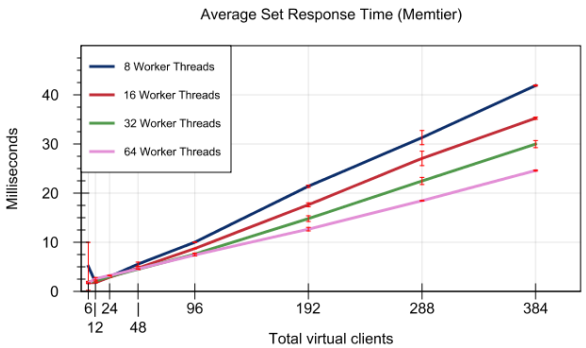


(b) Section 3.1, average read-only response time (measured in the clients)

Figure 24

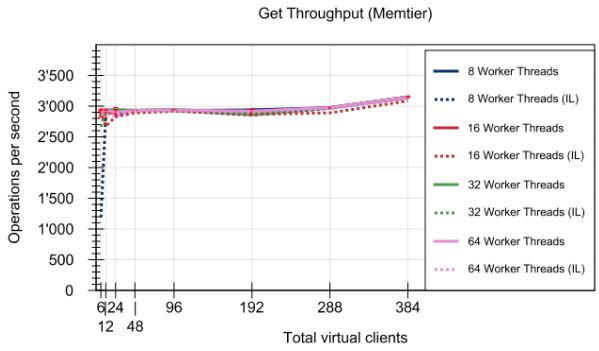


(a) Section 3.2, accumulated average write-only throughput (measured in the clients)

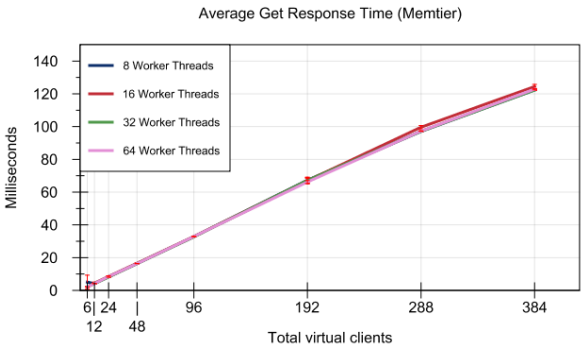


(b) Section 3.2, average write-only response time (measured in the clients)

Figure 25



(a) Section 3.2, accumulated average read-only throughput (measured in the clients)



(b) Section 3.2, Average read-only response time (measured in the clients)

Figure 26