Advanced Systems Lab Report

Autumn Semester 2018

Name: David Yenicelik
Legi: 15-944-366

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

1 System Overview (75 pts)

I structure my project (code only) into the following folders. I give a short explanation for each item on how it is used.

I will start out with a diagram that explains the overall structure and refers to each .java file. I will then be more detailed with my project-code structure.

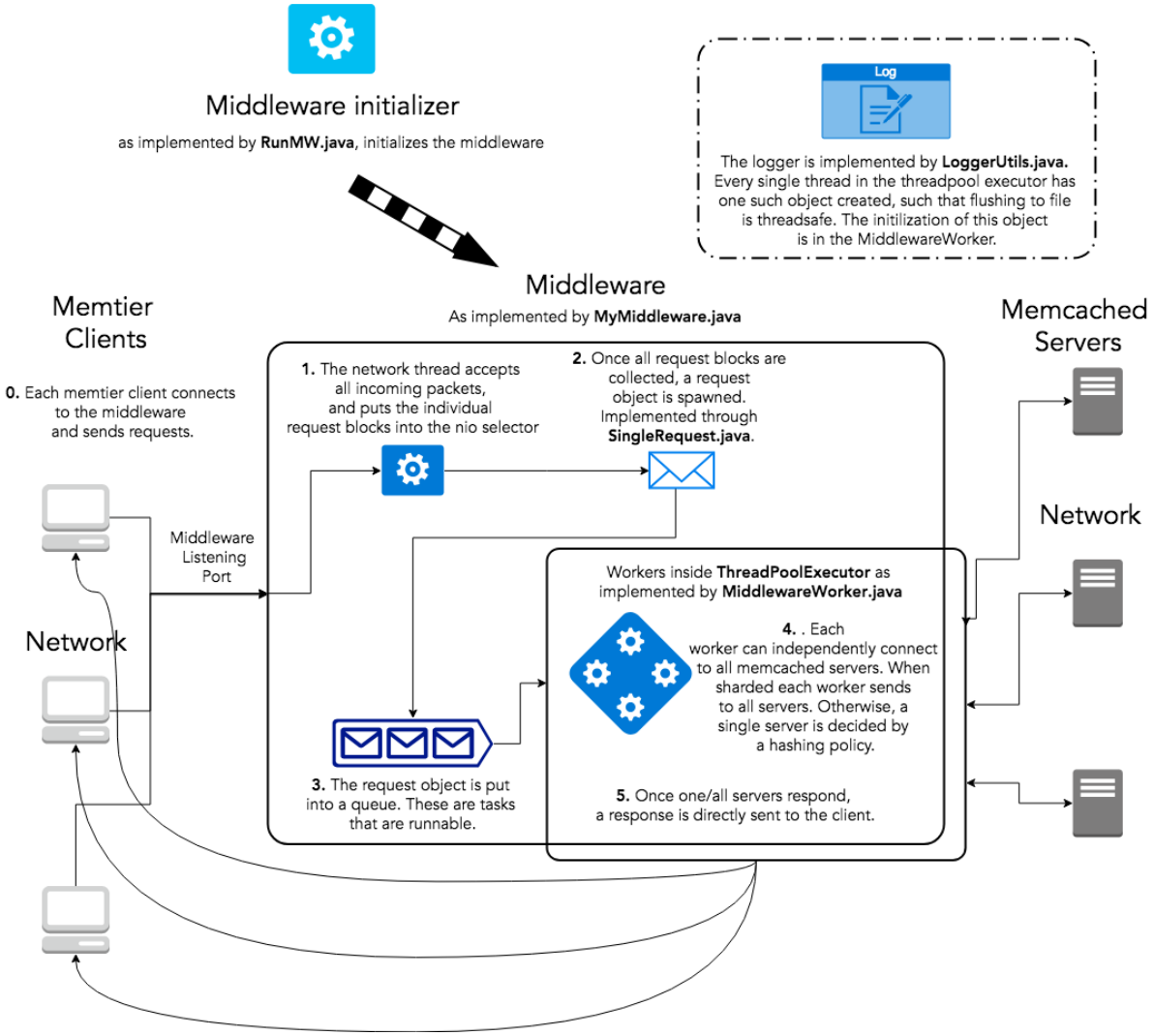


Figure 1: Exp.2.1: A figure with two subfigures

1. **scripts**
   This folder includes all the logic to automatically compile the code, deploy the code, run the experiments (each individual one), and automatically download the logs. This is true both for a local development environment (using docker VM's), and the external "production" environment (using Azure VM's). The local docker and external Azure systems are interchangable with a small change in command.

2. **data**
   This folder includes all the raw data that is pulled from the experiment, all the python files which process upon this data, and all the processed data.

3. **figs**
   This folder includes all the figures that are create using the processed data from point 2. I use python to create figures from individual graphs

4. **src**
   I will talk about the code structure of the src directory with more detail in this next section. I take a top-down approach while explaining (i.e. starting with where requests originate, and where they go from there). I will keep this consice, as the *lifetime of a request* section covers some information on what happens in which file.

   (a) **RunMW.java**
   This is the entry point / main class of the program. This is the default implementation by the TA's.

   (b) **SingleRequest.java**
   Is the class which encodes a single request from one of the clients. This can include the SET or GET operation. For logging purposes, this class also includes all possible times which may be of use to calculate the queueing theory (and latency and throughput) later on.

   The request type is parsed by checking the very first character of the request string. If the string starts with a "g", the request type is a GET (later on, it is decided on the fly if it is a MULTIGET by checking if the number of keys is greater than 1). If the string starts with a "s", the request type is a SET. If none of the above cases hold, an error happened (we exit gracefully, as this is unintended behavior).

   (c) **MyMiddleware.java**
   This is the entrypoint of how the Middleware is called. The datastructure we use to connect to the individual clients is the **nio.Selector**. The **nio.Selector** can hold multiple connections from different clients connecting. This datastructure can hold multiple keys for each connections, and fills up each key as long as the request is not done yet. Once the request is complete, it will throw away that key and make space for a new connection. Furthermore, it can parse packages that don't immediately fit into the bytebuffer. The backlog size is bigger than 0 such that multiple requests can made to the same channel at the same time (and these requests are backlogged).

   This class is responsible for fetching the individual requests (spawned as a *SingleRequest* object)using non-blocking IO, putting it to a queue, and spawning *MiddlewareWorker*'s within a ***ThreadPool*** to act upon these requests. We use the NIO selector to handle all connections. The entire logic of this .java class runs in on the main thread of the class, which I may refer to as "Network Thread". This allows for multiple clients to connect to the server in a non-blocking fashion. Whenever a connection is compelte (i.e. the request is complete), this selector spawns a *MiddlewareWorker* and a *SingleRequest*.

   The diagram above symbolizes how this works.

   (d) **MiddlewareWorker.java**
   The *MiddlewareWorker* takes a *SingleRequest* and passes it to the server(s). The **MiddlewareWorker** implements **Runnable**, and is executed by the **ThreadPoolExecutor** by calling **submit**. Depending on whether the request type is a SET or GET, we have different behavior:

   - case SET: The *SingleRequest* is sent to each individual server in a sequential manner. After the sending to the server is done for all servers, it listens to the

response of each individual server. It listens until all servers have responded. If any single item has responded with an error, this SET operation responds with the first error encountered to the client. Otherwise it returns a **STORED** message.

- case GET: The string of the *SingleRequest* is used to calculate a hash for the individual request. This hash is then modulo-ed with the number of server that we can send the request to. This is done to balance the read-workload amongst different memcached server instances. The modulo operation decides which server to send it to. This has **provenly uniformly at random distribution** behavior (proof at: https:eprint.iacr.org2016985.pdf). In short, this proof relies on the fact that hashing is pseudorandom, and that pseudorandomity means that for any input, all output values are uniformly at random distributed. Pseudorandomity keeps it's properties when applied with the modulo operator. Because all operations preserve pseudorandomity, the final operation - and thus the server chosen - is also uniformly at random.
- case MULTIGET (nonsharded): The nonsharded multiget acts exactly like the GET case. Again, the uniformly at random assumption is guaranteed because the hashing algorithm is pseudorandom (and as such provides keys uniformly at random).
- case MULTIGET (sharded): The sharded MULTIGET case acts as follows. The MULTIGET request is first split up into $n = max(keys, buckets)$ where *buckets* is the number of servers in total, and *keys* is the number of keys in total. The split is sequentially, which means that the first thirds of the requests go to the first server, the second third go to the second server, etc. Each individual split up request is then treated as an individual GET request. When the memcached server responds, all the answers are concatenated in a sequential fashion and sent back to the client. Errors are handled and also reduced to the first occuring error if this is the case.

(e) **LoggerUtils.java**
A helper class. Includes all the logic that is needed to log the requests to hard disk. All the request logic is accumulated to variables, and the mean is flushed to disk every few minutes. For GET requests (and in the interest of experiment 5), GET requests are accumulated into a list, and flushed to disk every 5 seconds. The logging happens in such a way that each individual *MiddlewareWorker* has it's list of requests and all accumulater objects (inside the LoggerUtils.java) to be logged. Because there is a separate Logger for each thread, there are no issues with multithreading as there is no concurrent data access.

Every single request keeps track of the following values, which is then flushed to the LoggerUtils.java (and thus to the file) object when the request has been successfully. All the information in SingleRequest is used to create the following log-informations.

i. timeRealOffset
ii. differenceTimeCreatedAndEnqueued
iii. differenceTimeEnqueuedAndDequeued
iv. differenceTimeDequeuedAndSentToServer
v. differenceTimeSentToServerAndReceivedResponseFromServer
vi. differenceTimeReceivedResponseFromServerAndSentToClient

vii. timeRealDoneOffset

(f) **RequestType.java**
A helper struct definition, which defines the two possible input types (Multi-gets are decided on the fly at a different point as described in *SingleRequest.java*)

1.1 Lifetime of a request

In the following I will talk about how requests enter the middleware, how they are parsed, how they get distributed to servers, and how the middleware communicates these values back to the clients. This is a more detailed version of the above diagram.

1. **Request coming from client to middleware:**
When a request comes from a client to the middleware, I use an **nio.channels.Selector** to accept the request. This datastructure has the following benefits. First, it can distinguish between multiple clients. Second, it can process these individual requests simultaneously in an asynchronous (non-blocking) manner. Third, it fills up multiple channels (one for each connection, and thus, one for each client), which means that if the request does not fit into one network packet, it will just listen for the rest of the packet. I detect if a single request fills up by filling a java **ByteBuffer** until it the request has come to an end (which we can recnognize by waiting for the **END** keyword. We parse the type of request by looking at the very first character (interpreting the bytes) and cross-comparing if this is a *set* or *get*. To distinguish between *multi-get* and *gets*, we later on split the string resulting from parsing the bytebuffer by spaces. If the number of elements after splitting is bigger than 2 (the "get" keyword, and the key), then I parse a multi-get. Else, I parse a get.

2. **The incoming request spawns a SingleRequest object:**
The SingleRequest object is specified in the **SingleRequest.java** java file and is wrapped around a **MiddlewareWorker.java** object which implements a java **Runnable** on which I later on call **submit** using a **ThreadPoolExecutor**. This file keeps track of the statistics described in the LoggerUtils.java class for logging. The SingleRequest takes over the **ByteBuffer** which was created while **nio.channels.Selector** was listening for a complete network packet. This ByteBuffer will later on be passed to the individual server(s) (after some processing).

3. **Submitting a SingleRequest to a MiddlewareWorker using the java ThreadPoolExecutor**:
I use the SingleRequest that was generate before, and spawn a new MiddlewareWorker. This MiddlewareWorker can then be submitted to the java **ThreadPoolExecutor** which contains the number of middleware-threads (as specified per experiment). Each individual middlewareworker contains one instantiated **LoggerUtils** class per thread and thus is threadsafe.

4. **Sending the server response back to the client**:
Each middlewareworker, and thus each individual thread, has a connection to the client that the request came from, and a hashmap of sockets to connect to all the servers. The response is sent back to the client using the description in the previous sub-section. In any case, this goes through the individual middleware-worker thread, and does **not** go through the initial nio.channel.Selector again.

1.2 Some statistics, and observed bandwidths amongst different VMs

I use the program *iperf -c 'VM Address'* to arrive at these statistics. I arrive at the theoretical maximum throughput by checking how many packages could possibly fit in the bandwidth, where the message size is 4KB (i.e. 4096 bytes), and the bandwidth is specified by the value before that. I divide the bandwidth by the packet size to arrive at the theoretical maximum throughput per single VM.

VM From	VM To	Bandwidth	Minimum Latency	Theoretical Maximum Throughput
Client	Middleware	201 Mbits/sec	0.729 ms	6'250 ops/sec
Client	Server	201 Mbits/sec	0.636 ms	6'250 ops/sec
Middleware	Client	804 Mbits/sec	0.611 ms	25'000 ops/sec
Middleware	Server	804 Mbits/sec	0.825 ms	25'000 ops/sec
Server	Client	101 Mbits/sec	0.766 ms	3'125 ops/sec
Server	Middleware	101 Mbits/sec	0.642 ms	3'125 ops/sec

Intuitively, this means that if we have 3 clients and one server, the server must be a bottleneck, and that the clients share the $3'125 ops/s$ (approximately $1'000 ops/s$ per client). **I will use these numbers to derive some explanations later on**.

I am now talking about my statistical methods of how I analyse the experiments. Each plotted value for each experiment in the subsequent report was generating using **three repetitions**, each including a **warm-up**, and a **cool-down** phase of 15 seconds (where the core-length of the experiment was 60 seconds). For some experiments, I use a warm-up and cool-down time (respectively) of approximately 7 seconds, as my experiments became very lengthy at some point, and as this showed to give good results. To arrive at a single rate amongst rates, I use the **arithmetic mean**, which is a common statistically significant measure. As an **error mesaure**, I use the standard deviation, which is also a common statistically significant measure that is tightly coupled with the mean. These are common statistical tools, and according to the **law of large numbers**, will approximate any distribution truthfully when the number of samples go against infinity, which is why I chose these two measures. **Anytime I graph something, the mean of 3 (or more) trials will be used to mark then point. Errorbars are derived from the standard deviation**. The mean μ and standard deviation σ are calculated as follows:

$$\mu = \frac{\sum_i^n x_i}{n} \tag{1}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}} \tag{2}$$

Please be aware, that I use the word *client* to refer to the VM's that host the memtier instances, and the word *server* to refer to the VM's that host the memcached instances. Any virtual machine, I refer to as VM.

Whenever I speak about *virtual client per thread*, I refer to the number of virtual clients per thread **per memtier instance**. The reader should be able to infer that this number needs to be multiplied by the number of memtier-instance-threads to arrive at the total number of virtual clients. I keep this concept constant, so comparison between experiments is sound. In addition to that, I will use the words **latency** and **response time** interchangeably. For the sake of easier typing, I will use approximate values (as far as reasonable), when it comes to discussions on throughput.

2 Baseline without Middleware (75 pts)

These experiments don't use the middleware, and only consist of memtier client instances, and memcached server instances. Each of these experiments have a warm-up time of 15 seconds, and a cool-down time of 15 seconds. I Using high warm-up and cool-down times, I hope to get rid of manipulation of measurements by predominan non-core phases.

2.1 One Server

I use the following setup:

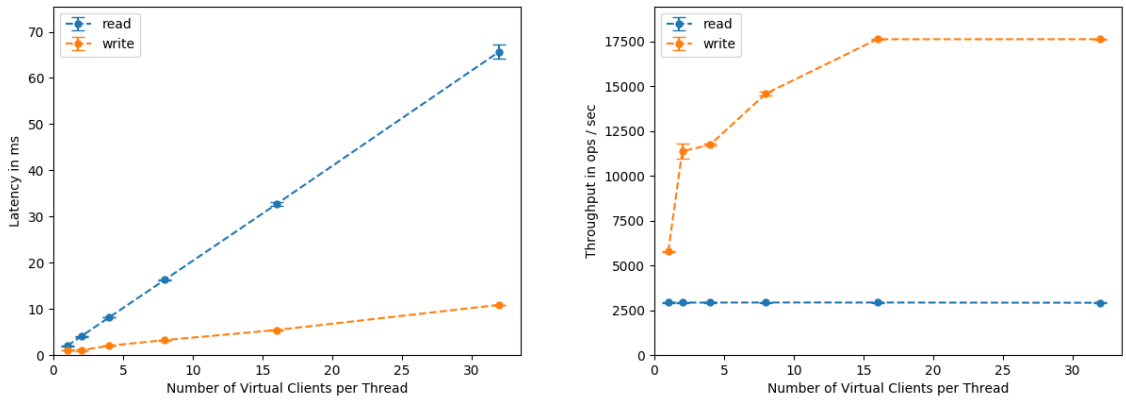
Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[1..32]
Workload	Write-only and Read-only
Repetitions	3 or more

I will test out the response time and latency for a different number of virtual clients per thread on the client-side (per memtier instance), namely [1, 2, 4, 8, 16, 32]. I will talk about read-only operations, and write-only operations. As there is no difference in pre-populating the servers in the case of writes, for both experiments I pre-populate the server the same way as using the following command, which sequentially generates and stores each key in the server. **Wheneverin future experiment setups I refer to prepopulating the server, I will refer to this command**:

```
memtier_benchmark -s {SERVER_IP} -p {PORT}
--protocol=memcache_text --clients={VIRTUAL_CLIENTS_PER_THREAD} --threads={THREADS}
--requests=15000 --ratio=1:0 --data-size=4096
--expiry-range=9999-10000 --key-maximum=10000 --key-pattern=S:S
```

.

The clients and servers don't use multi-gets, and there is no middleware involved. I run 3 repetitions of each configuration and plot the mean and standard deviation of the trials for each possible configuration.



(a) Latency (b) Throughput

Figure 2: Exp.2.1: Latency and Throughput per number number of virtual clients per memtier-thread. Comparison of read-only and write-only values.

For the **read-only experiments**, the system becomes saturated even with 1 virtual client per thread (per memtier instance), and does not oversaturate as the system is stable. This is because the throughput reached approximately almost $3000 ops/s$, which conforms to the observation of the maximum theoretical throughput per **server** in section 1 (minus some network overhead).

This stability is indicated by the error (which is very small, and thus for some points intersects with the measured points). The error metric (standard deviation) is explained in section 1.

For the **write-only experiments** the system (in this case the server), is undersaturated until 16 virtual clients per memtier-threads, after which it reaches a stable saturated phase, and never reaches over-saturation (due to stability of the system). This stability of the system is indicated by the small error bars, (which are hardly readable for this reason) This is because the throughput reached approximately almost $18000 ops/s$ at 16 virtual clients per memtier-threads, and is increasing to this number before that, which conforms to the observation of the maximum throughput per **server** in section 1.

As a sanity check, the interactive law holds, as the throughput flattens out after a square-root-like growth, while the response time still increases linearly.

For **read-only operations**, the bottleneck is the upload bandwidth of the server, as 3 clients are trying to download a load of 100Mbit/s, and thus must each share appr. 33Mbit/s. This corresponds to the total upload bandwidth of $3000 ops/sec$ (which was empirically proven in section 1 through an additional experiment), is thus divided amongst three servers. This also proves to be a valuable sanity check. As the number of threads increases, more requests are able to be generated. Because the network bandwidth stays constant, but we introduce more virtual clients, the round-trip time of individual requests increases. This implies a linear increase in latency, as can be seen from the graph. This linear increase in the response time provides a third sanity check.

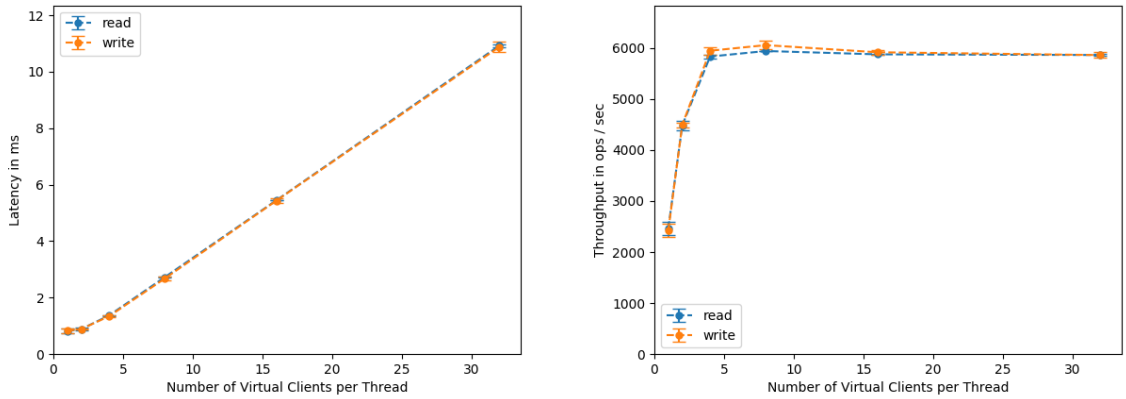
For **write-only operations**, the bottleneck is the upload bandwidth of the thee clients, as 3 clients are trying to upload a load of 200Mbit/s each. The server only responds with message suchs *STORED*, which take up almost no bandwidth compared to the actual message itself. Thus, the total bandwidth is the additive bandwidth of each individual client, which means approximately $3 \times 200 Mbit/s = 600 Mbit/s$, which corresponds to approximately $18000 ops/sec$ (same calculation as in section 1.2). This also proves to be a valuable sanity check. I know that storing the individual requests is not the bottleneck, as a local docker experiment proves that higher throughputs can be achieved locally. The clients are not able to generate enough load with 1, 2 or 4 virtual clients per thread, and only generate a maximum load with 16 virtual clients per threads, where the system finally saturates the upload bandwidth of the client VMs. This can also be seen from the graph, which plateaus around 16 virtual clients per thread. As a sanity check, the latency graph shows how the latency barely increases for up to 4 virtual clients per thread. Only after the 4th virtual client per thread, does the response time increase linearly, which underlines the saturation phase.

2.2 Two Servers

I use the following setup:

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..32]
Workload	Write-only and Read-only
Repetitions	3 or more (at least 1 minute each)

I will test out the response time and latency for a different number of virtual clients per thread on the client-side (per memtier instance), namely [1, 2, 4, 8, 16, 32]. As there is no difference in pre-populating the servers in the case of writes, for both experiments I pre-populate the server the same way as in experiment 2.1 (Baseline without middleware and 3 clients). The clients and servers don't use multi-gets, and there is no middleware involved. I run 3 repetitions of each configuration, each having a length of 90 seconds (such that the warm-up and cool-down times of 15 seconds respectively even out).



(a) Latency (b) Throughput

Figure 3: Exp.2.2 Latency and Throughput per number of virtual clients per memtier-thread. Comparison of read-only and write-only values.

For **throughput for the read-only experiments** both server virtual machine, are using their maximum capacity to upload all the values that are uploaded by the server virtual machines. The combined throughput of the two servers is $2 \times 100 Mbit/s$ which corresponds to a compound $200 Mbit/s$. This conform to the maximum theoretical throughput of $6000 ops/sec$ when using the same calculation as in section 1.2. The graph underlines these observed values.

For **throughput for write-only experiments**m the server can respond fast enough to all requests by the clients, because the response message (which is usually "STORED") consumes almost none of the bandwidth. Both memtier instances are using their maximum capacity to upload all the values to from the *client* machine, to the server virtual machines. The system is undersaturated initially as the clients cannot create enough load, and then becomes saturated after an increase of virtual client threads. This can be seen from the plot which indicates that there is minimal to no oversaturation (there is a slight decrease in throughput, but is below some error measures). The inflection point can be seen on the graphs at 4 virutal clients per memtier-thread. The repsonse times start to become more steeper after this point, and the throughput flats out. This conform to the $6000 ops/sec$ when using the same calculation as in section 1.2.

For **read-only operations**, the bottleneck is the upload bandwidth of the server virtual machines, as each server is uploading 100Mbit/s each, which compounds to 200Mbit/s together.

The total upload bandwidth of 6000ops/sec (which was empirically proven in section 1 through an additional experiment), is thus divided amongst amongst two memtier servers and thus two memtier instances on the same machine (as can be seen in the client logs). This also proves to be a valuable sanity check. For up to 4 virtual clients per threads, the response time stays constant, simply due to the fact that the bandwidth is not saturated, and the requests don't have to wait on each other to be processed (but can be processed in parallel). This implies an almost constant latency for up to 4 virtual clients per thread, and then an increase in latency, as can be seen from the graph.
For **write-only operations**, the bottleneck is the upload bandwidth of the client virtual machines, as the single client is uploading 200Mbit/s each, which compounds to 200Mbit/s together. Once the client spawns 4 virtual clients per thread, the client VM upload bandwidth starts to saturate as the client upload bandwidth of 200Mbit/s is reached, as each memtier instance in the client is actually able to generate approximately $3'000 ops/sec$, which compounds to $6'000 ops/sec$ when ran on two memtier instances. Because the network bandwidth stays constant, the round-trip time of individual requests increases, except for up to 4 virtual clients per thread. For up to 4 virtual clients per threads, the response time grows slowly, simply due to the fact that the bandwidth is not saturated, and the requests don't have to wait on each other to be processed (but can be processed in parallel).

Both read-only and write-only operations are stable, as can be seen from the very small (and this barely readable) error metric. Furthermore, the interactive law holds for both experiments, as the throughput flattens out after 4 virtual clients, while the latency starts increasing. This servers as a sanity check.

2.3 Summary

The following table summarizes the quantiative results. I then compare the read-only and write-only workloads. This is a summary. Because this report is meant to be concise, for any explanations, please see the above sections.

Maximum throughput of different VMs.

	Read-only workload	Write-only workload	Configuration gives max. throughput
One memcached server	2940	17633	VC=16
One load generating VM	5939	6054	VC=8

I first compare read-only and write-only operations. I start with **read-only operations** first. For **one memcached server and three clients**, the bottleneck is the server, which is bottlenecked to serve at most $3'000 ops/sec$. In contrast, using **one load generating VM**, the bottleneck is the upload bandwidth of upload bandwidth of the server (2 times the upload bandwidth of a server) $6'000 ops/sec$. In both cases, a bottleneck occurs exactly if the messages take over the entire network/upload bandwidth, and never happens when only messages such as "GET" or "STORED" are sent back. The bottleneck as such is the same for both configurations, being the server which must send enough data back to the client. I continue with **write-only operations**. For **one memcached server and three clients**, the bottleneck is the client, as the server has a response which virtually fills none of the bandwidth. For **one load generating VM**, the bottleneck is the upload bandwidth of the client, as the client cannot generate more than $6'000 ops/sec$ of bandwidth-filled requests. The bottleneck as such is the same for both configurations, being the client which mus send enough data to the server (to be

stored). All these values conform to the measurement done in section 1.

I proceed with comparing the two server configurations. For **one memcached server**, any read-operation is bottlenecked by the number of servers, because we only have one server, this value can maximally achieve $3'000 ops/sec$. For any write-only operations, the bottleneck is the number of clients. Because we have 3 clients, this value can maximally achieve $18'000 ops/sec$. The compound upload bandwidth of the client machines is at approximately $3 \times 200 Mbit/s$, whereas the compound upload bandwidth of the server machines is approximately $100 Mbit/s$. This means that the clients can send almost any load of operations they want. In contrast, write-operations don't suffer this penalty, as the server has almost no bandwidth to give away, as in this case the server only responds with "STORED", and never with a datapacket which is of size $4KB$ (which would fill the bandwidth).

For **one load-generating**, any read-operations are bottlenecked by the number of servers. Because there is always 2 servers, the maximal throughput is $t6'000 ops/sec$. Any write-operation is again bottlenecked by the number of clients. Because we have one client, this value is capped by the $6'0000 ops/sec$.

For one memecached server, the "balance" is heavily destroyed, whereas one can see that one client and two servers both can handle the same volume in throughput. As such, the configuration with only one memcached server, and the configuration with one load generating VM are in polar contrast to each other, showing that the servers have relatively few bandwidth compared to the client machines, but that the server is fast enough for as long as the operation is only a sending a simple "STORE" (i.e. bandwidth is not fully used up).

Any numbers that deviate from the theoretical maximum arise due to network overhead, and are statistically insignificant in this analysis.

3 Baseline with Middleware (90 pts)

3.1 One Middleware

In this set of experiments, I use three client memtier virtual machines, and 1 memcached server. These virtual machine instances are connected with exactly one middleware virtual machine in the middle. The three clients connect to the middleware. The middleware connects to the server. For this section, I repeat each experiment for 3 times and plot the standard deviation amongst those trials. I also allow for a 15 second warm-up and 15 second cool-down time, and disregard these measurements when retrieving the logs about the request times from the middleware. I measuring the throughput and response time for different values of number of virtual clients.

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..32]
Workload	Write-only and Read-only
Number of middlewares	1
Worker threads per middleware	[8..64]
Repetitions	3 or more (at least 1 minute each)

The setup is exactly the same as in experiment "Baseline without Middleware and 1 server", with the difference that we inject one middleware between the 3 clients and the server. In addition to the virtual clients per thread, I also investigate how the joint modification of the middleware-threads influences the latency and throughput. I test out the throughput and

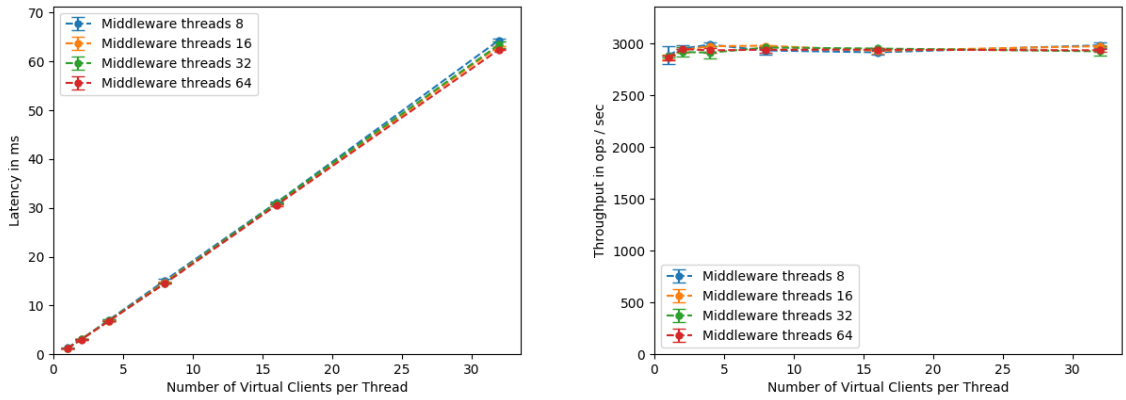
latency for any permutation of virtualthreads=[1, 2, 4, 8, 16, 32] and threads in the middleware=[8, 16, 32, 64].

I first separately investigate **read-only operations** and **write-only** operations, and arrive at a comparison between these four systems in the summar in section 3.3.

Any graph involving the middleware stripped out the operations that happened during the warm-up and cool-down times. **For the sake of conciseness, I had to collapse the multiple plots into one single plot, even though the measurements overlap to a high degree.**

3.1.1 Read-only

I first plot the latency and response as measured on the middleware.



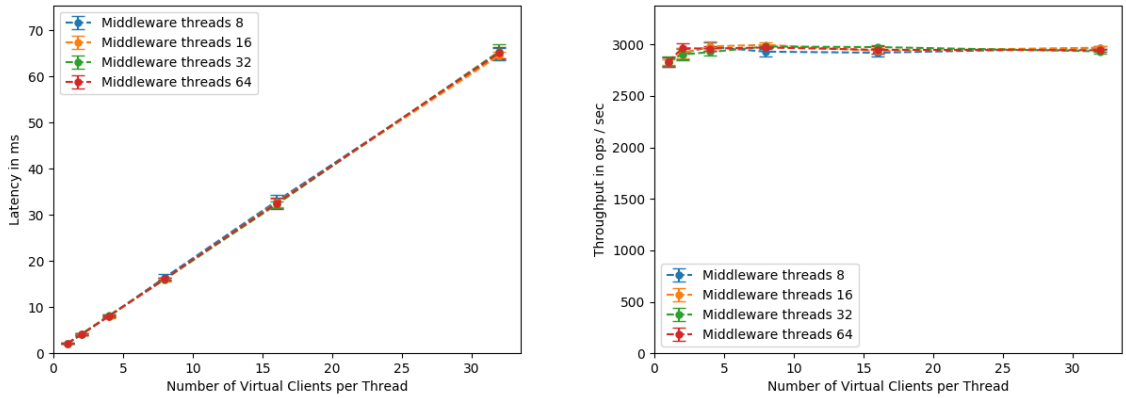
(a) Latency (b) Throughput

Figure 4: Exp3.1: Latency and throughputs for **read-only** as measures by the **middleware** (one middleware)

For read-only operations, the bottleneck is the upload bandwidth of the server, as 3 clients are trying to download a load of 200Mbit/s, and thus must each share appr. 66Mbit/s per machine. This translates to appr. $1000 ops/sec$ per machine, or in other words $3000 ops/s$ total throughput. The throughput graph supports this claim. As a sanity check, this conforms to the observation of the maximum throughput per server in section 1. The middleware is no bottleneck even with only 8 middleware threads, because the results do not deviate from the results in section 2.1 (where the only difference is that we are addding a mdidleware between the clients and the server). Another obvervation supporintg this claim stems from the fact that increasing the number of middleware threads does no increase performance. The graphs support this claim. The middleware virtual machine does not slow this down, as all GET requests are very short simple requests that all fit into the buffer. Also, the IO that serves as input to the middleware is non-blocking, which means that it can accept many such small requests simultaneously. This can be seen as increasing the number of middleware threads does not affect performance, as the server-side is bottlenecked. Each individual request is sent back to the client. But because all threads have designed channels to the respective clients, and because the bandwidth of the server is so low compared to the bandwidth of the middleware virtual machine, this causes no slowdown.

The system is very stable, as can be noticed by the error which is again very small, and thus barely readable around the measured points. As another sanity check, I present the throughput

and latency plots from the **client machines**, which all conform to the throughputs and latencies as caluclated in the middleware. To keep it concise, and because these graphs heavily look like the middleware graphs, I include these in the appendix.



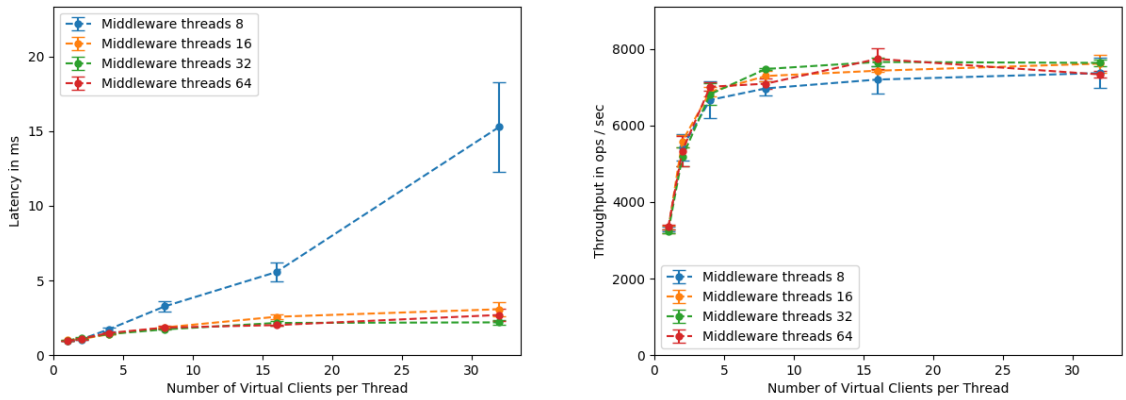
(a) Latency (b) Throughput

Figure 5: Exp3.1: Latency and throughputs for **read-only** as measured by the **clients** (combined throughput)

As the number of threads increases, more requests are able to be generated. Because the network bandwidth stays constant, the round-trip time of individual requests increases. The clients are able to generate enough load with even 1 virtual client per thread, such that with 1 virtual thread the network bandwidth is saturated at a very early stage with 1 virtual client per memtier thread. This implies an increase in latency. The linear increase in the latency graph supports this claim. As a last sanity check, one can confirm that the interative law holds. The throughput flattens out, as the response time increases. This claim is supported by the graphs.

3.1.2 Write-only

I first plot the latency and response as measured on the middleware.



(a) Latency (b) Throughput

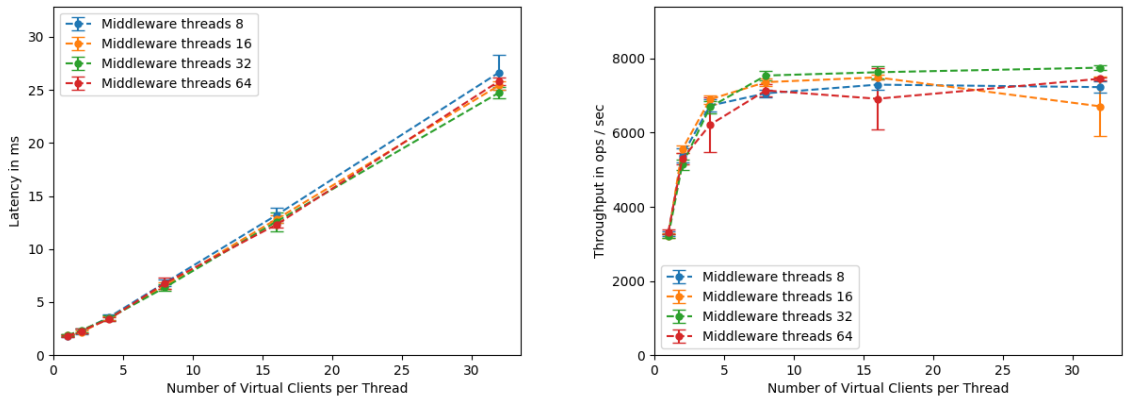
Figure 6: Exp3.1: Latency and throughputs for **write-only** as measures by the **middleware** (one middleware)

The bottleneck is the middleware, as the performance increases until 8 virtual clients per threads, and then sharply drops due to the oversaturation of the incoming requests. This claim is supported by the fact, that 8 middleware-threads has the lowest performance, and increasing the number of middlewarethreads increases the throughput. Especially the response graph supports this claim. Another explanation for this is that the middleware reaches a throughput of approximately $8'000 ops/sec$, which is much lower than the theoretical maximum of $18'000 ops/sec$, as measured in experiment 2.1. The only factor we changed is adding the middleware, so the middleware must be the delimiting factor. Another reason why this is the case is that the network bandwidth is not a delimiting factor for this, as we know from experiment 2.1 that the client maximum throughput for this specific configuration would usually almost be $18'000 ops/sec$, and also because I know that the server only responds with "STORED", which take up almost no bandwidth at all. This claim is supported by the fact that the middleware virtual machines do allow an upload bandwidth of $25'000 ops/sec$, which should easily be able to hand the theoretical maximum total upload throughput of the client machines of $18'000 ops/sec$. In the next experiment (3.2), I will continue to show that this claim is true.

The middleware is under-saturated until 4 virtual clients per memtier thread, and is saturated after this point. There is no real over-saturation, as the middleware does not decrease in performance, but keeps stabily this throughput rate. The throughput graph, which increases in throughput until 4 virtual clients per thread, and then (only slowly) increases and almost stays constant supports this claim.

The experiments are still stable, but do have higher variance than all previous experiments, which stems from the fact that a single machine must now be responsible for a very high throughput (instead of distributing this load across multiple virtual machines). This can be seen from the higher errobars (compared to previous experiments).

To apply some sanity checks, I observers the latency and throughput graphs as measured by the client. The client takes into account additional network roundtrip time, which makes the response graph as measure by the client slightly different. The throughput graph confirms the middleware's throughput rates, however. This can be seen in the following graphs.



(a) Latency

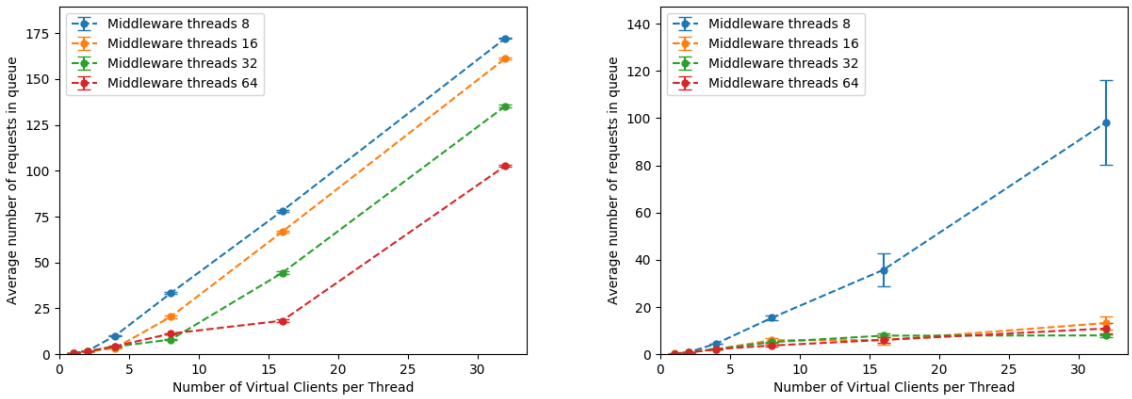
(b) Throughput

Figure 7: Exp3.1: Latency and throughputs for **write-only** as measures by the **clients** (one middleware)

As a final sanity check, the throughput increases and then stays constant, while the throughput constantly increases. This indicates that the interactive law applies. This is also supported by the graphs.

3.1.3 Additional Explanation

I analyse the average size of the queue inside the middleware, and also investigate response times as a function of *middleware threads* and *virtual clients per memtier thread*. The graphs of the response times can be found in the appendix, for conciseness.



(a) Read-only

(b) Write-only

Figure 8: Exp3.1: Average Queuesize in middleware **per** number of middleware threads and **per** virtual cliens per memtier thread

For write-operations, the queue contains the message that is to be written. This means that if the requests cannot be handled fast enough, the queuesize will increase. If the requests can be handled fast enough though, the queue-size will stay at a low constant value. For read-only requests, there is almost no bottleneck, which means that the queuesize amongst all middleware-workers is similar. The queuesize increases with an increasing number of virtual clients per threads, but because the all the queuesizes (across a different number of middleware-threads) increase together, this shows that the middlewareworker is no bottleneck. The read-only average queuesize graph supports this claim.

For write-only requests, however, 8 middleware threads are not as performant as more number of middleware threads. This can easily be seen from the queuesize graph for write-only operations. This graph shows that the queuesize steeply increases for 8 middleware-threads, but stays similar across higher middleware thread-counts.

As the number of virtual clients per threads increases, the latency is shifted more and more from "waiting for the server", to "waiting in the queue", as the wait-time and service-time graphs in the appendix show. This implies that the middleware congests the requests and collects the requests in the queue as it is not able to process all requests fast enough.

3.2 Two Middlewares

In this set of experiments, I use three client memtier virtual machines, and 1 memcached server. These virtual machine instances are connected with exactly two middleware virtual machines in the middle. The three clients connect to the middlewares (two memtier-instances per client machine). The middlewares both connect to the server. For this section, I repeat each experiment for 3 times and plot the standard deviation amongst those trials. I also allow for a 15 second warm-up and 15 second cool-down time, and disregard these measurements when retrieving the logs about the request times from the middleware. I measuring the throughput and response time for different values of number of virtual clients.

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..32]
Workload	Write-only and Read-only
Number of middlewares	2
Worker threads per middleware	[8..64]
Repetitions	3 or more (at least 1 minute each)

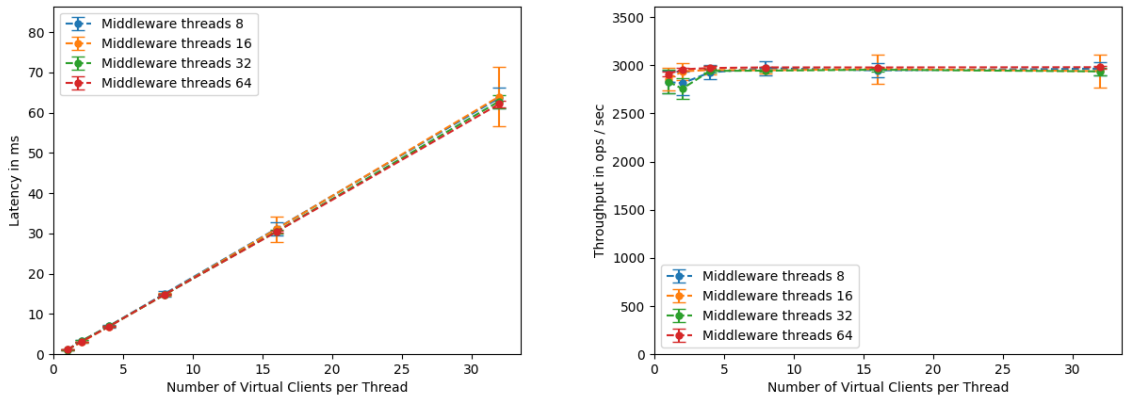
The setup is exactly the same as in experiment "Baseline without Middleware and 1 server", with the difference that we inject two middlewares between the clients, and the server. Another difference is that we allow each individual client to run two memtier instances (each with one thread) to be able to connect to two instances each.

In addition measuring the throughput and response time for different values of number of virtual clients, we also allow to modify the number of middleware threads as another measureable variable. This means that I test out the throughput and latency for any permutation of virtualthreads=[1, 2, 4, 8, 16, 32] and threads in the middleware=[8, 16, 32, 64].

I first talk separately about read-only operations, write-only operations, and then compare them in another section subsubsection "Additional Explanation".

3.2.1 Read-only

I first start with read-only operations.



(a) Latency

(b) Throughput

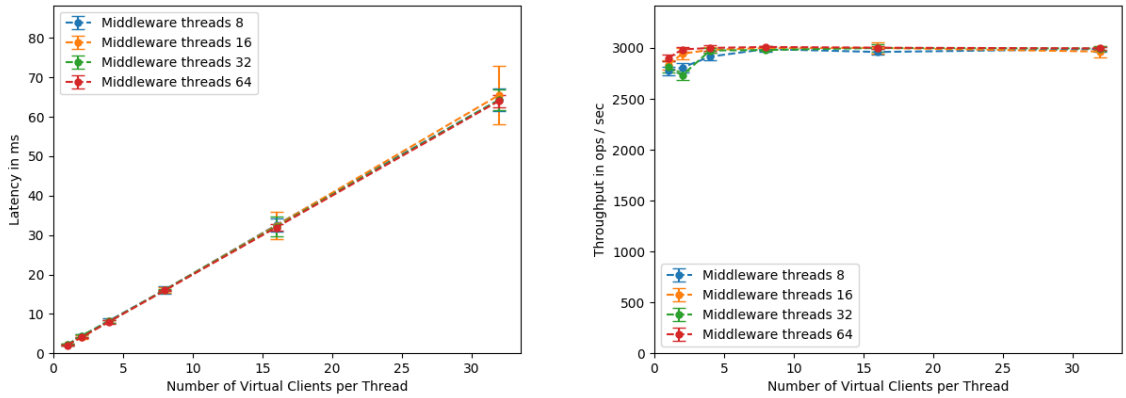
Figure 9: Exp3.2: Latency and throughputs for **write-only** as measures by the **middlewares** (two middlewares)

For read-only workloads, the bottleneck is the server, with the very same explanation as in section 3.1. Please refer to section 3.1, as I will not repeat this explanation here for conciseness. Because we introduce no additional server, this bottleneck does not change, and the maximum number of ops / sec does not increase. This claim is supported by the throughput graph, which shows that the system is already saturated with 2 virtual clients per thread. Introducing the second middleware does not improve this performance, as the number of servers does not change.

As such, the system saturates already with 2 virtual clients per memtier thread - the same as in experiment 2.1 read-only and experiment 3.1 read-only. There is no oversaturation, and

the system is stable, as can be seen from the slight error bars in both the response graph, and the throughput graph.

As a sanity check, one the interactive law applies, as the latency grows although the throughput stays the same. In addition to that, the throughput and response time graphs as measure by the client support the throughputs as observed in the middleware.

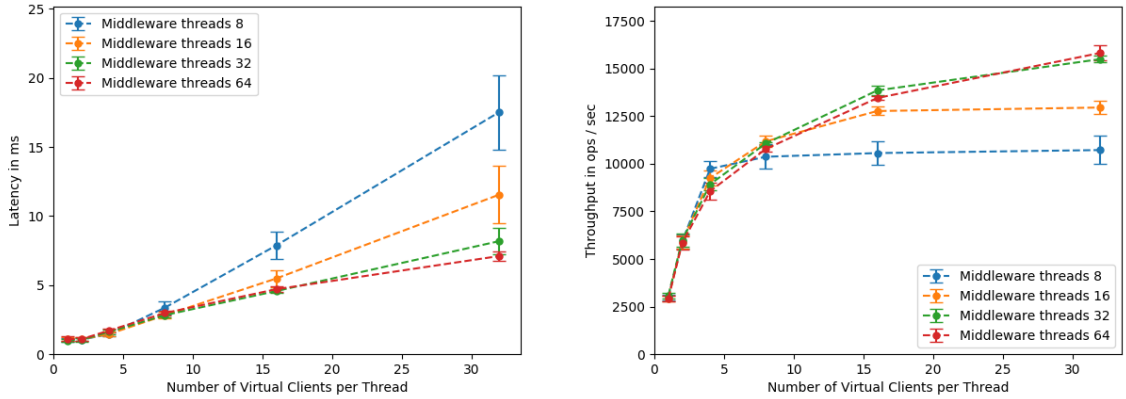


(a) Latency (b) Throughput

Figure 10: Exp3.2: Latency and throughputs for **write-only** as measures by the **clients** (two middleware)

3.2.2 Write-only

I proceed with write-only operations.



(a) Latency write only (b) Throughput write only

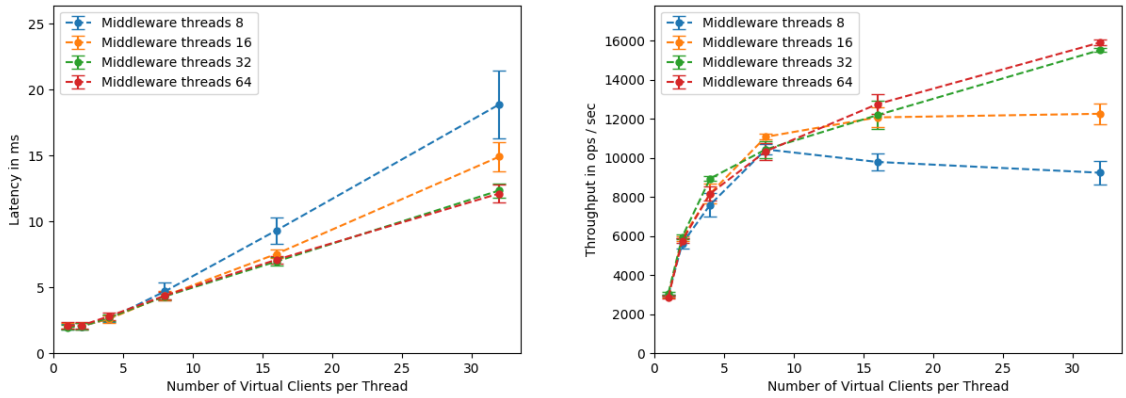
Figure 11: Exp3.2: Latency and throughputs as measures by the middlewares

For read-only workloads, the bottleneck is the middleware instance, as was explained in section 3.1 write-only. Please refer to section 3.1 for an explanation on this. Because the middleware is the bottleneck, one can clearly see that for write-only operations, adding the second middleware almost doubles the throughput of the system. This is because the total load of the system is now distributed amongst two middlewares instead of only a single one. This means

that each in total, we have double the amount of middleware-threads that can handle the requests. This claim is supported by comparing the write-only graphs in this section with the write-only graphs in section 3.1. One can validate, that doubling the number of threads in the middleware (which can be achieved by also by just spawning a second middleware which takes over part of that load), the throughput increases significantly.

The bottleneck is the total number of middlewarethreads in the system. For 8 and 16 middleware threads per middleware instance, the system is under-saturated up to 8, respectively 16 virtual clients per thread. For a higher number of middleware-threads per middleware instance, increasing the number of virtual clients per thread almost allows to reach the system the maximum throughput as is given by the compound throughput of the client machines. This claim is supported by the graph which shows an evening out of the throughput rate after 16 virtual clients per thread for 8 and 16 middleware threads per middleware instance, and an square-root-like increase in the throughput of the system which have 32 and respectively 64 middleware threads per middleware.

As a sanity check, one the interactive law applies, as the latency grows although the throughput stays the same, and this with an ever-increasing rate. In addition to that, the throughput and response time graphs as measure by the client support the throughputs as observed in the middleware as the following graphs show.



(a) Latency write only

(b) Throughput write only

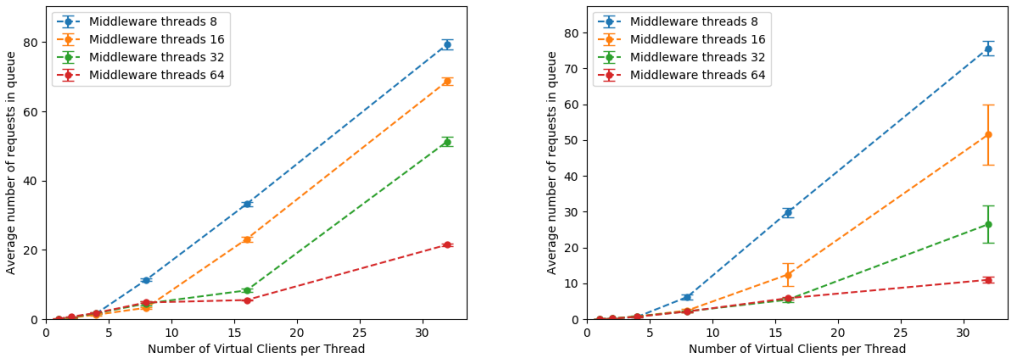
Figure 12: Exp3.2: Latency and throughputs as measures by the clients

3.2.3 Explanation

I will now further elaborate the explanation from the above sections, using the some statistics including the average queuesize per configuration, and the time spent in each possible stage of the lifetime of a thread.

First of all, for read-only requests, the queuesize does increase as we have fewer numbers of middleware threads. However, because saturation (from the server side) is reached quickly, this has no affect on the performance.

For write-only requests, the queue-size is more interesting. The queue is emptied much faster when as we increase the number of available middleware threads. This emptying allows for a higher throughput, as more requests can be handled this way. The right graph below supports this claim. I will analyse the time of a request spent at each stage even further.



(a) Read-only (b) Write-only

Figure 13: Exp3.2: Average Queuesize in middleware **per** number of middleware threads and **per** virtual cliens per memtier thread

The following plots allow for a more throughput analysis, supporting the explanations made in the previous subsection (subsection 3.2). For a concise analysis, I only show the graphs for the two extreme configurations (virtual clients per memtier thread equals 1 , or equals 32). For the other configurations, please view the appendix.
The claim we made before was that - for write-only requests - the number of middleware threads is the bottleneck, which implies that increasing this number lowers the waiting time in the queue. The right figures in the following graph support this claim. As more middleware threads are added, the relative time spent in the queue decreases compared to the waiting time (time spent to wait for servers response). The graph shows that for 32 virtual clients per memtier thread, the ratio is approximately $16/2 = 8$ for 8 middleware threads, whereas this ratio decreases to approximately $2/5 = 2.5$ for 64 middleware threads. This means much less times is spent in the queue itself, and simply waiting for the memcached server to respond. One can also see, that the system is (as claimed before) not saturated with 1 virtual client.

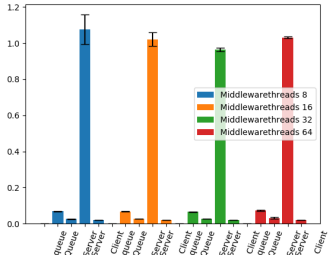


Figure 14: Read-only with 1 virtual client per memtier threads

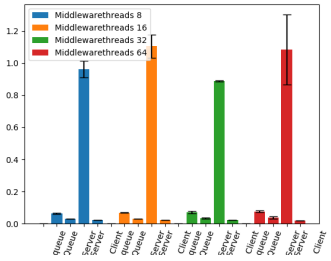


Figure 15: Read-only with 32 virtual client per memtier threads

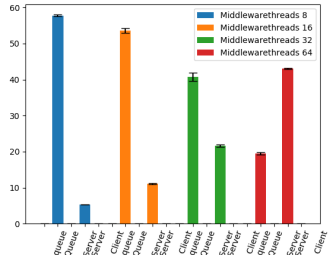


Figure 16: Write-only with 1 virtual client per memtier threads

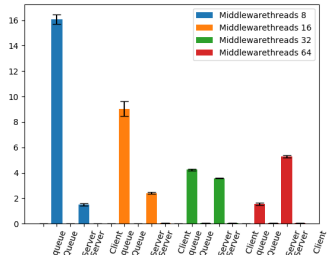


Figure 17: Write-only with 32 virtual client per memtier threads

Figure 18: Exp3.2: Time spent at different stages of the lifetime of a request. Each graph distinguished between read-only and write-only operations, as well as the number of virtual clients per memtier-thread (1 or 32)

3.3 Summary

Maximum throughput for one middleware.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware	2989 ops / sec	7.04 ms	4.280 ms	0
Reads: Measured on clients	2978 ops / sec	8.06 ms	n/a	0
Writes: Measured on middleware	7658 ops / sec	2.16 ms	0.82 ms	n/a
Writes: Measured on clients	7629 ops / sec	12.57 ms	n/a	n/a

Maximum throughput for two middlewares.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware	2978 ops / sec	14.88 ms	0.588 ms	0
Reads: Measured on clients	3010 ops / sec	15.96 ms	n/a	0
Writes: Measured on middleware	15809 ops / sec	7.11 ms	1.561 ms	n/a
Writes: Measured on clients	15901 ops / sec	14.27	n/a	n/a

Notice that the miss rate is always zero, because this is 1. a closed system, and all servers are pre-populated before any experiment starts.

Based on the data provided in these tables, write at least two paragraphs summarizing your findings about the performance of the middleware in the baseline experiments.

Also, when having only one middleware, the middleware is the bottleneck. This can be seen from comparing experiment 2.1 and recognising that for write-only operations, the exact same setup decreases from $18'000 ops/sec$ to $8'000 ops/sec$. As such, adding the second middleware doubles this capacity, from $8'000 ops/sec$ to $16'000 ops/sec$, as we now have independelty two servers that can send out these requests. The graphs of for write-only operations in section 2.1, and section 3.1 resp. section 3.2 underline these statements.

One can see this also from the fact that the middleware may not have enough middleware threads. For 3.1 writes-only, the throughput plateaus at almost $8'000 ops/sec$ when we have 64 middleware-threads. In 3.2 one can observer, that we now increase the number of middleware-threads to a total of $2 \times 64 = 128$ middlewarethreads (in the entire system). As a result, the throughput also almost doubles, and one cannot recognise a saturation phase. This implies that adding more middleware-threads possibly could result at the original, almost $18'000 ops/sec$ calculation.

In addition to that, reads spend much less time in the queue than write operats. The data in the above table supports this claim, as for the experiment with one middle and with two middlewares, the average time in the queue is 10 times higher, respectively 3 times higher for writes compared to read-operations. As a sanity check, the given values coincide with the values found in the preivous experiments, and also comply with the theoretical maximum bandwidth of the clients (of $3'000 ops/sec$), and the servers (of $18'000 ops/sec$). The reduction from this maximum to the observed values stems from network overhead, as well as middleware overhead (as was detailedly explained in section 3.1). Finally, it is important to notice that as the requests move from reads to writes, the load is transferred from the middleware (and thus the network thread) to the server threads. This is because for GET operations, the packets inside the queue do **not** occupy packets of size 4KB, and because the packets need to be handled by the server. For SET operations, the packets inside the queue **do** occupy packets of size 4KB, which means that the average time in the queue is naturally higher (even alone from copying and keeping memory around). These explanations are confirmed by the graphs which show at which stage of the lifetime the request is spending the most time at.

To sum up (for explanation, please refer to the above subsections), the total number of middlewarethreads operating in the system as a whole, has a high impact on the throughput of

the system. However, the system will be bottlenecked by the server-upload bandwidth for read-only operations, as the server constantly needs to upload packets of size 4KB. For write-only operations, this bottleneck would usually arise from the upload bandwidth of clients, which would be $18'000 ops/sec$ in total. However, in section 3.1 we obsevere that the middleware decreases the throughput rate compared to expeirment 2.1. In section 3.2, we observe that adding another middleware instance heavily increases this performance. This makes us think that the number of total middlewarethreads in the entire system heavily affects the overall throughput of the system.

4 Throughput for Writes (90 pts)

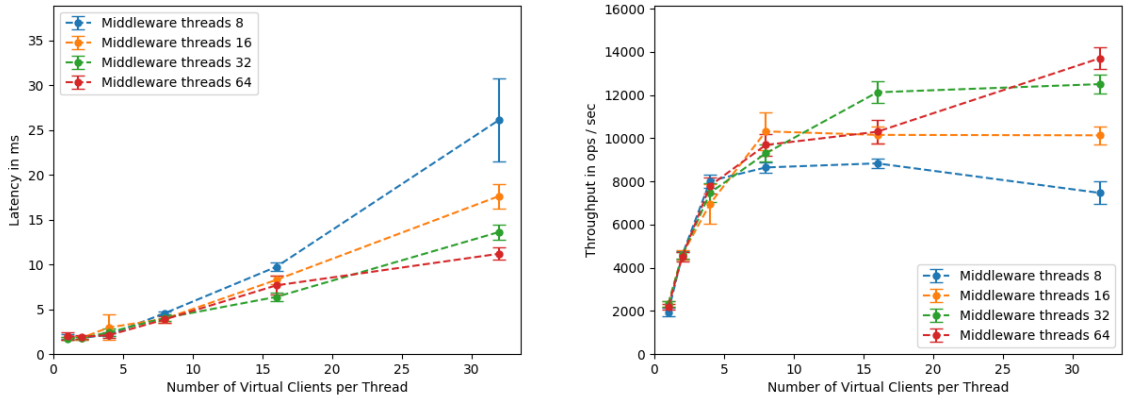
4.1 Full System

I am connecting each of three load generating client VMs to two middlewares. These middlewares are connected to three memcached server VMs each. I have the following setup.

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..32]
Workload	Write-only
Number of middlewares	2
Worker threads per middleware	[8..64]
Repetitions	3 or more (at least 1 minute each)

During this experiment, I iterate over all possible permutations of virtual clients per thread (in the range of [1, 2, 4, 8, 16, 32]), and worker threads per middlewares (in the range of [8, 16, 32, 64]). I run each experiment for 90 seconds (which includes a 15 second warm-up and a 15 second cool-down time). Each experiment again consists of three trials from which we measure the mean and the standard deviation. This section only covers write-only experiments. I cover response time (latency) in milliseconds, and throughput in ops/sec.

The following are graphs from the middleware.



(a) Latency write only

(b) Throughput write only

Figure 19: Exp3.2: Latency and throughputs as measures by the middlewares

The bottleneck of this experiment is the middleware. This is because even with a single server (instead of three servers), we know from section 2.1 that the maximum throughput

reaches almost $18'000 ops/sec$ with only 16 virtual clients per memtier thread. Compared to that experiment, we cap out at lower throughput rates (namely approximately $10000 to 12000 ops/sec$) for the same number of virtual clients per thread. The above graphs support this claim. It shows that for 8 middlewarethreads is a hard bottleneck, which saturates between 4 and 16 virtual clients per thread, and even starts oversaturating after 16 virtual clients per memtier thread. This over-saturation occurs because the middleware cannot effectively handle all the requests fast enough, and the proportion of wait-time (for the memcached servers) to queue-time (within the middleware) increases drastically with an increasing number of virtual clients per thread. The following graph supports this claim, where one can see that. This ratio increases from approximate $0.1/2.0 = 0.2$ to approximately $23/3 \approx 7.1$, which means that more time is spent in the queue as we increase the number of virtual clients per thread.

For 16 middlewarethreads, the system is undersaturated until 16 virtual clients per threads, and after this can successfully handle an increasing number of requests per second. This claim is supported by the above graph which shows a flattning out of the throughput per second, and by the graphs below, which show a much lower ratio (from approximately 0.2 to 5).

For 32 and 64 middlewarethreads, the system is under-saturated until 16 virtual clients per threads, and slowly reaches saturation as we increase to 32 virtual clients per thread. Reaching this saturation can be seen by the graph which starts flattening out more and more. This is also supported by the graphs below, which show the time spent at different stages of the lifetime of a request. The ratio from time spent in the queue, and time waiting for server increases much slower than for the 8 or 16 middlewarethreads (i.e. increases and decreases from approximately 0.2 to 2 (32) and 0.5 (64) respectively).

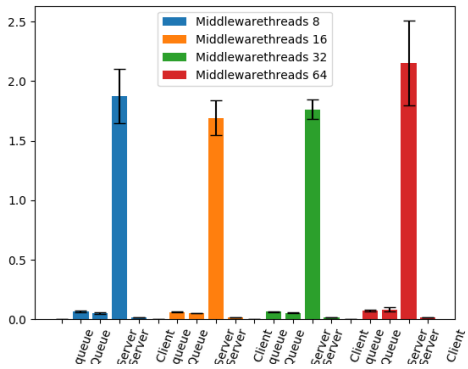


Figure 20: Write-only with 1 virtual client per memtier threads

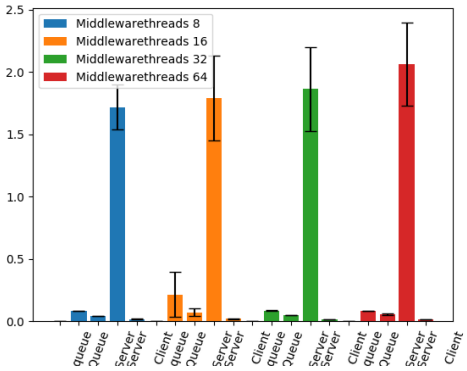


Figure 21: Write-only with 2 virtual client per memtier threads

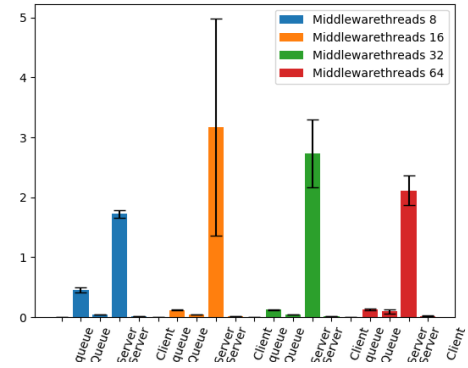


Figure 22: Write-only with 4 virtual client per memtier threads

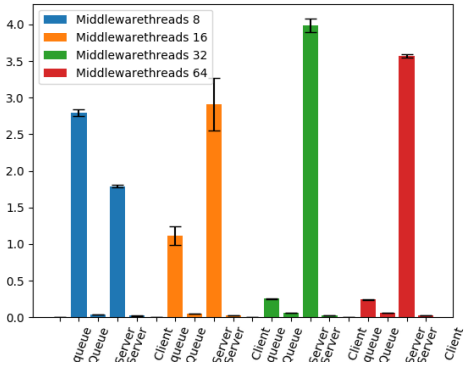


Figure 23: Write-only with 18 virtual client per memtier threads

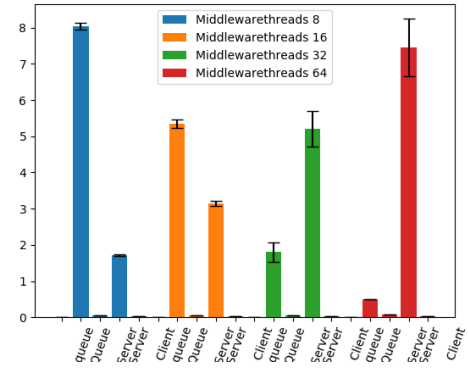


Figure 24: Write-only with 16 virtual client per memtier threads

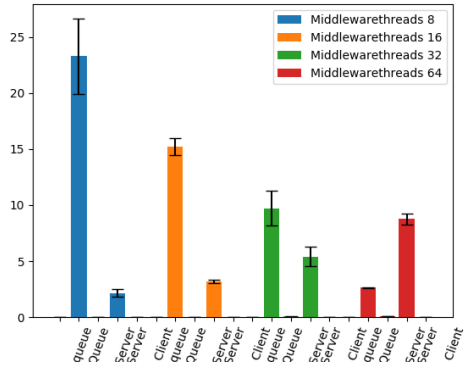
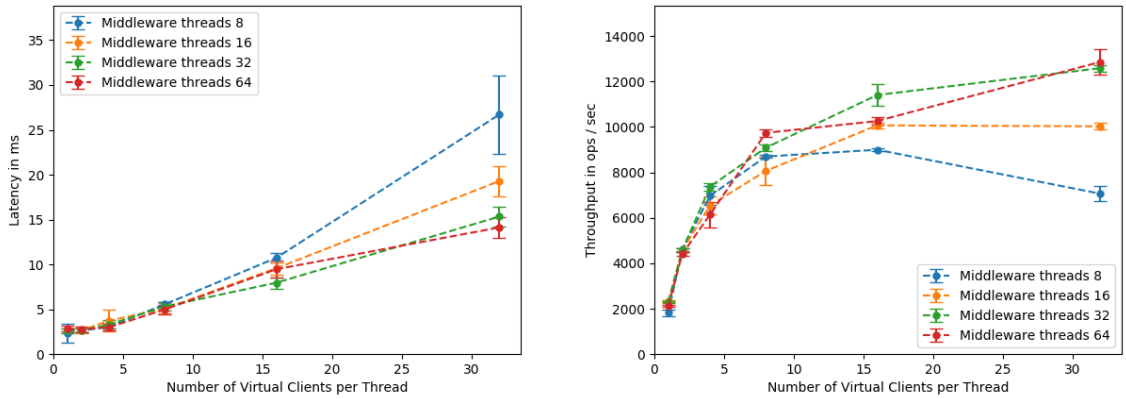


Figure 25: Write-only with 32 virtual client per memtier threads

Figure 26: Exp4.1: Time spent at different stages of the lifetime of a request. Each graph distinguished between read-only and write-only operations, as well as the number of virtual clients per memtier-thread (1 or 32)

Once can see an outlier point for at virtual number of clients = 16 and for 64 middlewarethreads. I have repeatedly run this experiment and get the same results each time. The middleware congests for this case, as the requests are sent to the server, but are waiting for the server rather than being congested in the queue. This waiting time is higher than the case with 32 middlewarethreads. However, one can double check that for both cases (32 and 64 middlewarethreads), this point does not provide a saturation point, can be seen by the below figure on the average queue-length (which is lowes for 64). This would mean that the requests are "in-sync" with the requests sent to the servers for 32 middlewarethreads, and starts to get congested with 64 middlewarethreads.

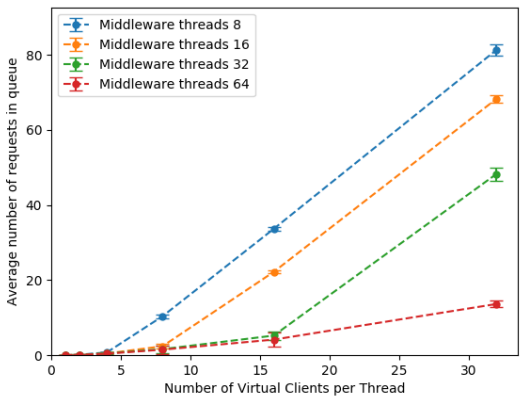


(a) Latency write only

(b) Throughput write only

Figure 27: Exp4.1: Latency and throughputs as measures by the clients

As a sanity check, the throughput and latency graphs by the client (the figure above) directly matches the values as measured by the middleware. Furthermore, the interactive law holds, as the throughput saturated and flattens out, as the latency starts growing super-linearly. The system is stable, as the error bars indicate small deviation amongst individual trials.



(a) Write only

Figure 28: Exp4.1: Average Queuesize in middleware **per** number of middleware threads and **per** virtual cliens per memtier thread

4.2 Summary

Based on the experiments above, fill out the following table with the data corresponding to the maximum throughput point for all four worker-thread scenarios.

Maximum throughput for the full system (in ops/sec, unless a different unit is specified)

	WT=8	WT=16	WT=32	WT=64
Throughput (Middleware)	8833	10153	12503	13706
Throughput (Derived from MW response time)	9858	11591	14096	17143
Throughput (Client)	8833	10068	12586	12858
Average time in queue	8.042 ms	5.346 ms	8.533 ms	2.631 ms
Average length of queue	33	22	48	13
Average time waiting for memcached	1.722 ms	3.136 ms	4.977 ms	8.760ms

For WT=8 and WT=16, the maximum throughput is achieved when the number of virtual clients per memtier threads are 16. For WT=32 and WT=64, the maximum throughput is achieved when the number of virutal clients per memtier threads are 32.

The results are very similar to the ones achieved in experiment 3.2, write-only. This is because exactly two servers are added to the system configuration from experiment 3.2. However, because the number of servers is not a bottleneck for write-only operations (as was explained before in section 2.1), adding more servers does not significantly affect performance (expect possibly delaying it a bit, because we now have to wait for more servers before we can proceed with the response, which may causes a slightly higher latency as can be seen by comparing the response time graph in this section with the response time graph in section 2.1).

Following the above logic, the results in this experiment should be very similar to the experiment 3.2, wirte-onle. The similarities with experiment 3.2 are that 8 middlewarethreads starts saturating with 8 virtual client threards (just as in this experiment), and then slightly oversaturates. This can be seen from the graph which starts to slightly lose some throughputs per second after 8 virtual client threads. Furthermore, 16 middlewarethreads saturates stabily, just like in this experiment. This can be seen by the graph stays flat after 16 virtual clients. For 32 and 64 middlewarethreads per middleware, the saturation point agrees with the saturation point in this experiment. The only markable difference is the additional latency caused by contacting multiple servers, which imply a slight increase in response times, as can be seen by higher slopes in this experiment)

The calculation for the throughput derived from the MW response time is not fully accurate, as the interactive law only when we have also defined a constant term Z that we take into consideration. Because the derived throughput is always up to 15% accurate, I arrive at the conclusion that these values do provide a sanity check.

Based on the data provided in these tables, draw conclusions on the state of your system for a variable number of worker threads. Overall, the number of total middleware threads are the bottleneck for the throughput of the system. This means that the number of total middleware threads is the most important parameter for the performance, whereas adding additional servers does not have any significant effect at all, as this was never a bottleneck for write-only experiments (as explained in section 3.2 and 2.1 write-only). This claim is supported by the fact, that the mean throughput of the maximum throughput configuration increases as we increase the number of middlewarethreads. The system is able to complete requests faster with an increasing number of middlewarethreads. In addition to the observed throughput, this can be seen by the average length in the queue, and average time in the queue. Whenever the number of virtual clients is constant (i.e. for WT=8 and WT=16 it is 16, and for WT=32 and

WT=64 it is 32), the average length of the queue decreases, as well as the average time in the queue (from 8ms to 5.4ms, and 8ms to 2.6ms) respectively.

The throughput on the clients match the throughputs on the middlewares, which finally support a sound analysis of my system as a sanity check.

5 Gets and Multi-gets (90 pts)

I use three load generating machines, two middlewares and three memcached servers. Each memtier instance has 2 virtual clients in total and the number of middleware worker threads is 64, as determined by previous experiments, providing the highest throughput.

For multi-GET workloads, I use the `--ratio` parameter to specify the exact ratio between SETs and GETs. I measure response time on the client as a function of multi-get size, with and without sharding on the middlewares.

5.1 Sharded Case

I run multi-gets with 1, 3, 6 and 9 keys (memtier configuration) with sharding enabled (multi-gets are broken up into smaller multi-gets and spread across servers). The following describes the detailed experiment setup.

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..9]
Number of middlewares	2
Worker threads per middleware	max. throughput config.
Repetitions	3 or more (at least 1 minute each)

The following are average response time as measured on the client, as well as the 25th, 50th, 75th, 90th and 99th percentiles.

To double-check that the above graph is correct (mainly the averages of the individual keysizes), I analyse the response time and throughput graphs. I will only include graphs from the middleware as these values don't include the warm-up and the cool-down times. However, the values measured as per client do conform with the trends found in the graph.

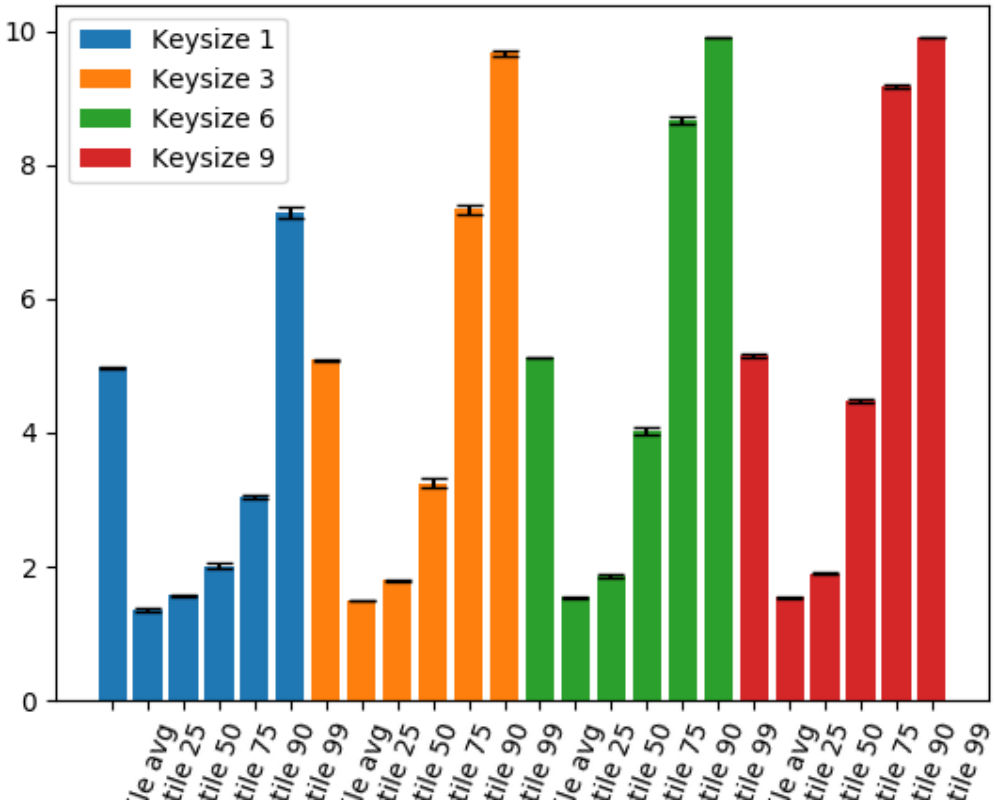
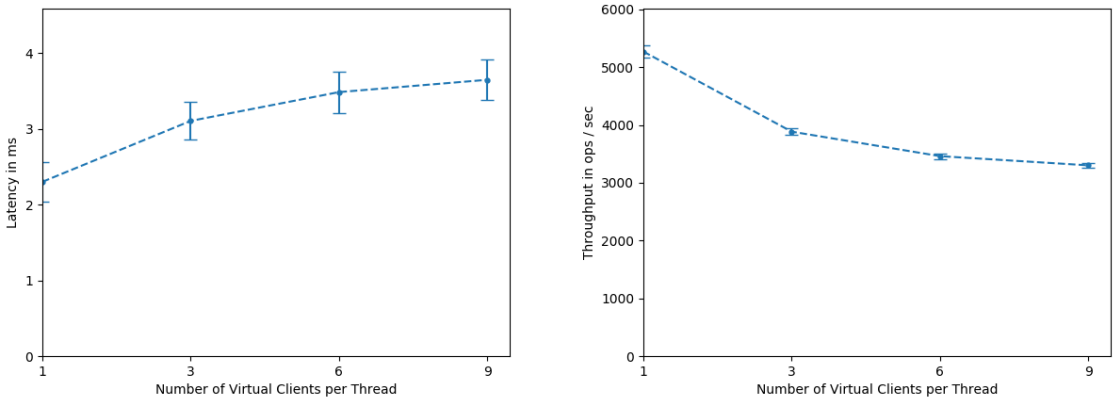


Figure 29: Exp3.2: Latency and throughputs as measures by the middlewares

The multiegets all have a similar response time. This is because the middleware does not internally distinguish between sending only a single packet, and multiple packets to the servers, as long as these packets fully fit into the *byteBuffer*. As my byteBuffer is of size $20 * 1024 * 4$, and a single "full" datapacket for message is of size 4096, this easily fits into the byteBuffer. As such, there is no significant slowdown in the response time. However, because of the higher memory requirements, bigger multi-gets tend to be a bit more unstable (as this requires more operations, and more operations can cause delays). This can be seen in the graph, where the latency percentiles (especially the 90th percentiles) increase with a higher number of keysizes.

To do a sanity check, I include the total latency and throughputs as measured by the clients (because the above histogram was derived from client values). One can clearly see that the trend presented in the above barchart is supported by the response times measured by the clients. In additional to that, the inverse law for throughput and latency holds, as the throughput and latency both have an overall almost constant tendency (with the exception of the keysize "9" decreasing throughput slightly, and increasing the latency slightly).



(a) Latency write only

(b) Throughput write only

Figure 30: Exp3.2: Latency and throughputs as measures by the clients

As a sanity check, the interactive applies, which means that the latency increases when the throughput decreases. As one can see, the graphs support this claim of the inverse correlation between response time and throughput.

5.1.1 Explanation

The bottleneck is either the server which has to respond to all the get-requests (and can do some compoundedly in the case of multi-gets with keysize ¿ 1), or the middleware which does not allow the client to push more requests towards the server. However, because the bandwidth of the middleware is so high, and theoretically supports almost $35'000 ops/sec$, it is highly likely that it is again the servers which cannot utilize more than the existing bandwidth of $3'000 ops/sec$. This relates to experiment 2.2 (baseline no middleware, read-only), where one single client instance is able to max out the servers fairly quickly. The latency caused by contacting multiple servers instead of just one can be seen by comparing these values to the non-sharded case. However, this network overhead is negligible as can be seen by comparing the graphs for the sharded case, vs. for the non-sharded case.

However, it could also be the middleware which is not able to pass on all requests quick enough. This would also explain the drop from $6'000 ops/sec$ in experiment 2.2, to $3'500 ops/sec$ in this experiment. After doing a profiling of the code, I recognize that the java function **String.split** takes up most of the performance, and heavily increases system utilization.

5.2 Non-sharded Case

I run multi-gets with 1, 3, 6 and 9 keys (memtier configuration) with sharding disabled. The following provides a more detailed view of the configuration that I used.

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..9]
Number of middlewares	2
Worker threads per middleware	max. throughput config.
Repetitions	3 or more (at least 1 minute each)

I plot average response time as measured on the client, as well as the 25th, 50th, 75th, 90th and 99th percentiles.

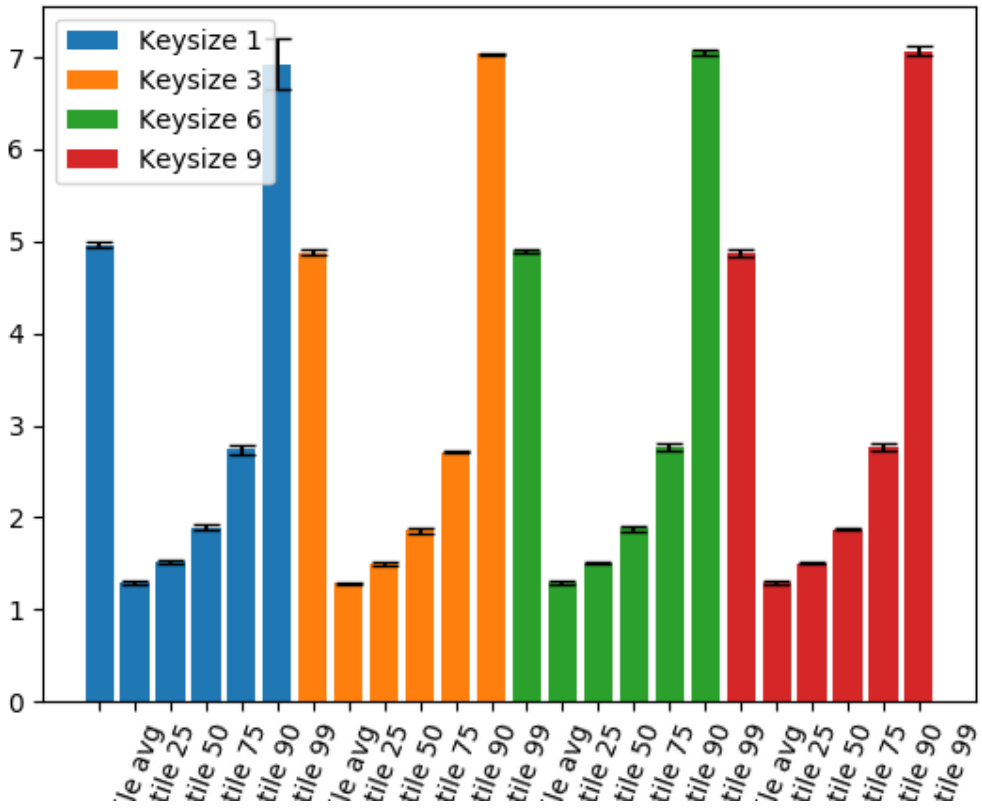
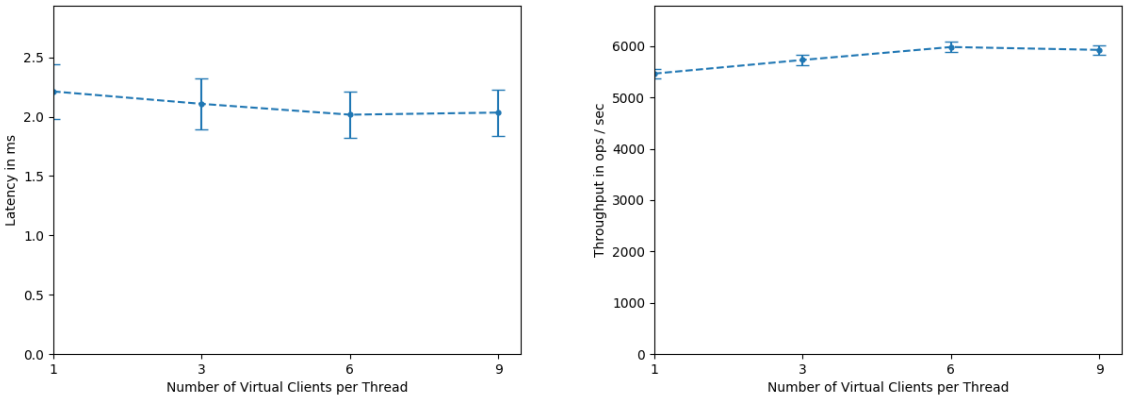


Figure 31: Exp3.2: Latency and throughputs as measures by the middlewares

Internally and inside the middleware, the non-sharded multiegets are treated exactly like multigets with keysize 1. This means - if the middleware operates well - that the response time and throughput should be very similar for any of the presented multiget keysize values. And this is indeed the case. The average and percentiles match up in response time as can be seen from the graph. There are some deviations, but these deviations are all within the error bounds of the other multikey-get sizes. The reason behind the very similar multi-key-getsizes is also the *byteBuffer* that I use, which allows each multikey-get request to fit into the buffer. As my byteBuffer is of size $20 * 1024 * 4$, and a single "full" datapacket for message is of size 4096, this easily fits into the byteBuffer. As such, there is no statistically significant decrease or increase in the response time or throughput.

To do a sanity check, I include the total latency and throughputs as measured by the clients (because the above histogram was derived from client values). One can clearly see that the trend presented in the above barchart is supported by the response times measured by the clients. In additional to that, the inverse law for throughput and latency holds, as the throughput and latency both have an overall almost constant tendency (with the exception of the keysize "9" decreasing throughput slightly, and increasing the latency slightly).

To double-check that the above graph is correct (mainly the averages of the individual keysizes), I analyse the response time and throughput graphs. I will only include graphs from the middleware as these values don't include the warm-up and the cool-down times. However, the values measured as per client do conform with the trends found in the graph.



(a) Latency write only

(b) Throughput write only

Figure 32: Exp3.2: Latency and throughputs as measures by the clients

As a sanity check, the interactive applies, which means that the latency increases when the throughput decreases. As one can see, the graphs support this claim of the inverse correlation between response time and throughput.

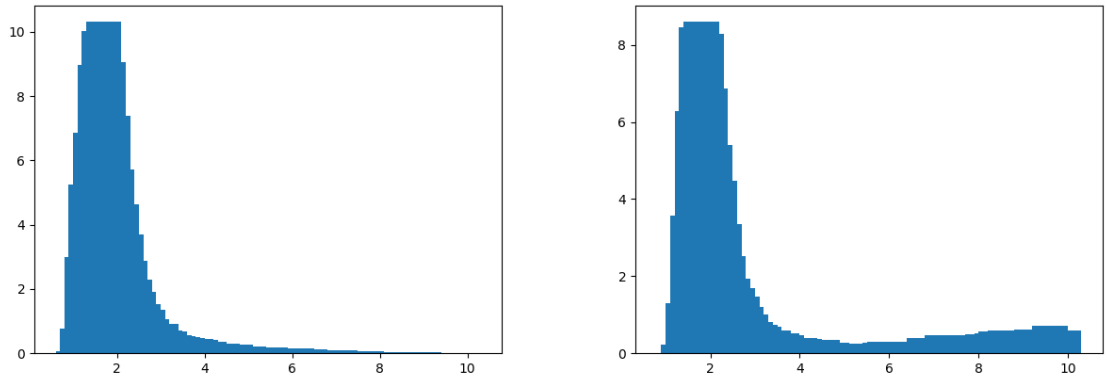
5.2.1 Explanation

Provide a detailed analysis of the results (e.g., bottleneck analysis, component utilizations, average queue lengths, system saturation). Add any additional figures and experiments that help you illustrate your point and support your claims.

5.3 Histogram

For the case with 6 keys inside the multi-get, I now display four histograms representing the sharded and non-sharded response time distribution, both as measured on the client, and inside the middleware. I chose the bucket sizes such that the represent intervals of at least 100 micro-seconds (i.e. 1/10th of a millisecond). After considering taking the mean of multiple items, I arrived at the conclusion of only picking a single memtier-instance's output and plotting this. This is for two reasons. 1. This gives us a more true distribution, and not a flattened mean. Furthermore this is a good approximation, because all latency-histograms (across all memtier-instances) are very similarly distributed. 2. Adding multiple histograms together which are slightly deviated on the x-axis will increase the common latencies (around 2 milliseconds), and

proportionally decrease the flatter regions, which also provide information to the reader. By just picking one instance, I solve this problem and allow for a clearer display of these flat regions (also because the spikes are clearly visible in either case).



(a) Non-Sharded Latency Histogram

(b) Sharded Latency Histogram

Figure 33: Exp3.2: Latency and throughputs as measures by the clients

More network operations come from the fact that more individual servers need to be contact to finish a request (instead of one server, all servers need to be communicated with). This can be seen in the histogram because the sharded case has a longer tail, as this requires 1. more operations (i.e. more computation effort), as was measured using the tool dstat, and 2. requires more network operations.

The client includes the network round-trip between sending it off from the client machine, and arriving at the middleware. In contrast, the middleware only views latency as the round-trip time from incoming request (right before entering the queue), and right before sending the response back to the client. This can be seen in the graphs, as the middleware histograms are shifter to the left w.r.t. the client histograms.

5.4 Summary

Provide a detailed comparison of the sharded and non-shareded modes. For which multi-GET size is sharding the preferred option? Provide a detailed analysis of your system. Add any additional figures and experiments that help you illustrate your point and support your claims.

6 2K Analysis (90 pts)

This 2k analysis includes the analysis of any correlations between the following factors: 1. number of memcached servers, 2. the number of middleware vm's and finally 3. the number of worker threads per middleware. The following table shows which possible values we cross-reference, such that we can later on analyse which factors have the most impact on throughput and response time. I place labels for each configuration, as these will be needed for later analysis

	Option -1	Option 1	Variable in analysis
Middlewares	1	2	X
Memcached servers	1	3	Y
Worker threads per MW	8	32	Z

Any configuration is run 3 times (3 repetitions) for 90 seconds, which implies a 15 second warm-up and a 15 second cool-down time. The following table shows a detailed configuration of my setup.

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

I apply this analysis once for read-only workloads, and then separately for write-only workloads, as both procedures are fundamentally different. I don't use any multi-get behavior (i.e. keysize is always 1 for read-only workloads). The 2k-analysis for the response time, and the throughput will be separate. For all the measured values, I will use values as observed by the middleware and not by the client. I use the **2k-r**, which implicitly calculates the errors as *implicitly* as part of the analysis.

For each of the experiments I calculate the q-values (which I are the coefficients before each random variable X, Y, Z), the squared sums (which I will abbreviate as **SS** from here on), and the fractional effects created by that specific factor. A table of all the values that was used to calculate the q-values, SS values and fractional effects can be found in the appendix. I will solve the equation to get the responsible coefficients per experiment:

$$y = a_X \times X + a_Y \times Y + a_Z \times Z + a_{X,Y} \times XY + a_{X,Z} \times XZ + a_{Y,Z} \times YZ + a_{X,Y,Z} \times XYZ + a_0 + e \quad (3)$$

where a_0 is a constant value, and y is the variable which we want to solve for (either throughput or response time), with e being the constant error term. I use this model, as this captures any possible linear combination between variables. Because any possible combination between variables is taken into consideration, I can then say which factors have the biggest impact (by looking at which fractional effect is the biggest), without sacrificing accuracy (by trying to separate variables, and by using an error term).

6.0.1 Read-only

I will first start with read-only operations. These are depending on the allocation of the configuration and are measured through my experiments. The cells contain the values that the respective variable attains.

For read-only operations, I get the following vector for the vector $\mathbf{a} = [a_0, a_X, a_Y, a_Z, a_{X,Y}, a_{X,Z}, a_{Y,Z}, a_{X,Y,Z}]$ and the error term e:

Factor	q-value	Sum-Squared	Effect (in fractions)
a_0	40.18	38748.79	-
a_X	-0.13	0.42	0.0000
a_Y	-19.11	8768.05	0.9939
a_Z	-1.08	27.80	0.0032
$a_{X,Y}$	-0.60	8.70	0.0010
$a_{X,Z}$	0.33	2.67	0.0003
$a_{Y,Z}$	0.01	0.00	0.0000
$a_{X,Y,Z}$	0.62	9.19	0.0010

Table 1: Exp6.1: 2k analysis graph for **Response Time and Read-only** operations

The analysis results in the heaviest coefficient (with a weight of 0.9939) to be Y, which corresponds to the number of memcached servers. This means that changing the number of memcached servers has the most effect on the response time of the system. Comparing this result to the difference in response time of experiment 3.1 and experiment 3.2, this result is accurate. More servers can handle more read-requests at the same time, which means that the latency will naturally go down.

Factor	q-value	Sum-Squared	Effect (in fractions)
a_0	5570.54	744742427.04	-
a_X	56.21	75825.04	0.0005
a_Y	2596.54	161808687.04	0.9967
a_Z	35.04	29470.04	0.0005
$a_{X,Y}$	55.71	74482.04	0.0005
$a_{X,Z}$	-55.79	74705.04	0.0005
$a_{Y,Z}$	60.38	87483.38	0.0005
$a_{X,Y,Z}$	-72.63	126585.38	0.0008

Table 2: Exp6.1: 2k analysis graph for **Throughput in ops/sec and Read-only** operations

Because the interactive law holds, almost the same result applies for the throughput, where the number of servers carry the highest effect (with a parameter-weight of 99%). Again, this is because we have identified repeatedly, that for read-requests, the server-upload-bandwidth is the bottleneck, as the upload-bandwidth is only $100 Mbit/s$, whereas the clients can pull for much more. As the 2k analysis shows, (and all previous experiments read-only experiments), changing the number of servers also has the highest effect amongst all variables on the throughput.

6.0.2 Write-only

Now I discuss the 2k-r analysis on the write-only workloads.

The write-only workloads substantially differ from the read-only workloads. The read-only workloads have always been bottlenecked by the servers, which was also clear from the analysis in any previous section involving read-only operations

The write-only workloads however have multiple factors affecting the experiments.

Factor	q-value	Sum-Squared	Effect (in fractions)
a_0	14.44	5002.59	-
a_X	0.36	3.10	0.0014
a_Y	4.97	592.82	0.2593
a_Z	-6.72	1082.46	0.4735
$a_{X,Y}$	-2.67	171.09	0.0748
$a_{X,Z}$	2.10	106.09	0.0464
$a_{Y,Z}$	-2.17	112.49	0.0492
$a_{X,Y,Z}$	2.31	128.25	0.0561

Table 3: Exp6.1: 2k analysis graph for **Response Time and Write-only** operations

From the calculation, the response-2k-r-analysis table shows that the most significant two factors are 1. the number of middlewarethreads in the system, and 2. the number of servers in the system (with approximately 47% and respectively 26% of the entire parameter-weights). This is simple, as I have shown in section 4.1, that for write-only operations, the total number of middleware-threads in the system has the most impact on the write-only times. Naturally, the less number of servers we have, the less the middleware needs to wait for each individual server (but can simply fetch the requests from one server), which means that adding a second server just increases latency. This is captured in the 2k-r analysis, being the second-highest factor having effect.

Factor	q-value	Sum-Squared	Effect (in fractions)
a_0	9182.71	2023731176	-
a_X	2334.71	130820712.04	0.5827
a_Y	-1290.21	39951301.04	0.1779
a_Z	1194.04	34217652.04	0.1524
$a_{X,Y}$	-210.21	1060501.04	0.0047
$a_{X,Z}$	703.54	11879301.04	0.0529
$a_{Y,Z}$	188.63	853905.37	0.0038
$a_{X,Y,Z}$	-373.21	3342827.04	0.0149

Table 4: Exp6.1: 2k analysis graph for **Throughput in ops/sec and Write-only** operations

Similar result applies for the 2-k-r analysis on the throughput, as the interactive law applies as was shown in all previous sections involving write-only operations. However, for the throughput, the parameter weights are much more dominant in the number of middlewares (with approximately 58% of the parameter weights), followed by the number of servers with 18% of the parameter weights, and finally with the number of middlewarethreads with 15% of the parameter-weights. Adding a second middleware literally doubles the number of threads, and allows for the load generated by the client to be distributed to two different virtual machines. This naturally allows for more requests per second, as the load for each individual middleware is almost halved compared to having only one middleware. This naturally means that the numeber of middlewares has the highest parameter weight (with about 58%) compared to the other factors. However, the number of servers affects the throughput, as this is coupled to the latency (by the interactive law). For the same reasons as mentioned above, the throughput for write-only experiments increases when we introduce less servers, for the simple reason that less servers need to be contacted (and also because the servers never provide a bottleneck for

write-only operations). For this reason, and because the number of servers is rather insignificant to splitting the client-load amongst two virtual machines, the parameter weights is also high with 18%. Finally, because the increasing the number of middlewarethreads on one machine also means increasing the capacity of a single machine (and for the similar reasons as mentioned in experiment 4.1 and in the above paragraph), the parameter-weight of this factor is also fairly large with 15%.

7 Queuing Model (90 pts)

In this section I model the workerqueue of the middleware (after the requests come in) using queueing theory to model how the system behaves with an asymptotically increasing number of threads. In both subsection I will go use the different number of middleware (specifically, one of [8, 16, 32, 64]) threads to apply this analysis.

I choose the following input parameters to model the system:

1. Mean arrival rate λ: I choose the mean arrival rate by the number of virtual clients per memtier thread. This rate is given by the **throughput as measured by the client-side in ops/sec**.
2. Mean service rate μ: For the M/M/1 case, I choose the mean service rate to be the **maximum throughput which can possibly be handled by the middleware** within the configuration in section 4. This means that I will choose the maximum possible throughput that the middleware experiences.

The following table provides a quick reference to these values. as given by the maximum throughput of the middleware (ops/sec) as given by the maximum throughput of the clients (ops/sec)

Threads in the middleware	μ Service rate	λ Arrival rate
8	8278	8120
16	10491	9209
32	11775	11789
64	12984	12150

7.1 M/M/1

In this subsection I model the behavior of the middleware and it's workerqueue using a M/M/1 queueing model.

I will predict the traffic intensity, the queue length, the latency and the wait time. I will then compare these to the observed values from section 4.1 to arrive at a conclusion if the comparison makes sense. The system is stable, as is specified by all the rho values which are all smaller than 1.

Table 5: M/M/1 Predicted vs. Observed values for 8 Middlewarethreads

			Predicted	Predicted	Predicted	Observed	Observed	Observed	
Middelware threads μ	λ	ρ	Queue length	Latency	Wait time	Queue length	Latency	Wait time	
8	8278	8120	0.9809023	50.38	0.00632536	0.00620456	14365.88		
16	10491	9209	0.87780868	6.31	0.00078007	0.00068475	14365.88		
32	11810	11789	0.99885762	1796.46	0.15272627	0.15264135	14365.88		
64	12984	12150	0.93577961	13.63	0.00119927	0.00112226	14365.88		

For each configuration of workerthreads, I will proceed with this calculation.

your entire system. Motivate your choice of input parameters to the model. Explain for which experiments the predictions of the model match and for which they do not.

7.2 M/M/m

Table 6: M/M/1 Predicted vs. Observed values for 8 Middlewarethreads

			Predicted	Predicted	Predicted	Observed	Observed	Observed	
Middelware threads μ	λ	ρ	Queue length	Latency	Wait time	Queue length	Latency	Wait time	
8	1034	8120	0.9809023	50.38	0.00632536	0.00620456	14365.88		
16	655	9209	0.87780868	6.31	0.00078007	0.00068475	14365.88		
32	367	11789	0.99944397	1796.46	0.15272627	0.15264135	14365.88		
64	202	12150	0.93577961	13.63	0.00119927	0.00112226	14365.88		

Build an M/M/m model based on Section 4, where each middleware worker thread is represented as one service. Motivate your choice of input parameters to the model. Explain for which experiments the predictions of the model match and for which they do not.

7.3 Network of Queues

Based on Section 3, build a network of queues which simulates your system. Motivate the design of your network of queues and relate it wherever possible to a component of your system. Motivate your choice of input parameters for the different queues inside the network. Perform a detailed analysis of the utilization of each component and clearly state what the bottleneck of your system is. Explain for which experiments the predictions of the model match and for which they do not.

8 Appendix

8.1 Experiment 3.1 - Baseline with one middleware

For read-only experiments, I have the following throughput and response graph derivated from the values of the client:

8.2 Experiment 3.2

Here is the extensive bar-plot on where most time is spent:

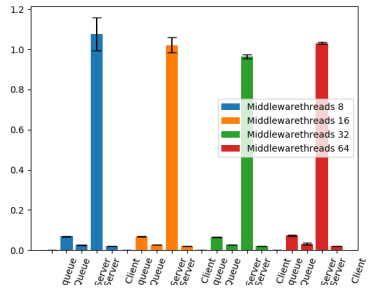


Figure 46: Initial condition

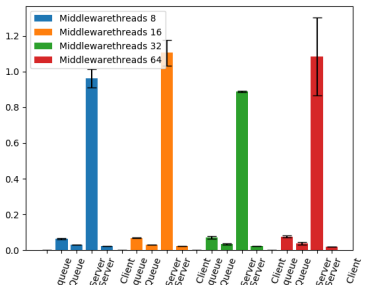


Figure 47: Rupture

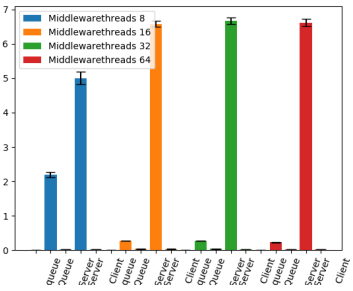


Figure 48: DFT, Initial condition

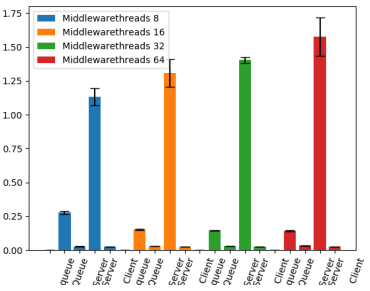


Figure 49: DFT, rupture

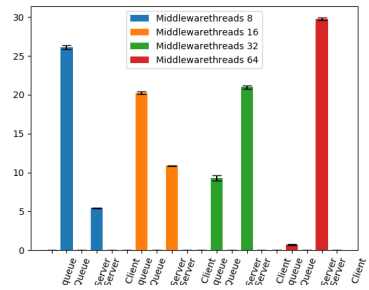


Figure 50: DFT, Initial condition

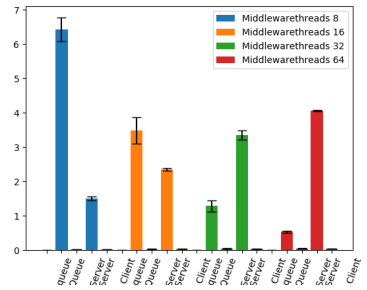


Figure 51: DFT, rupture

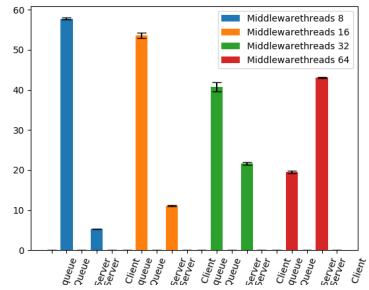


Figure 52: DFT, Initial condition

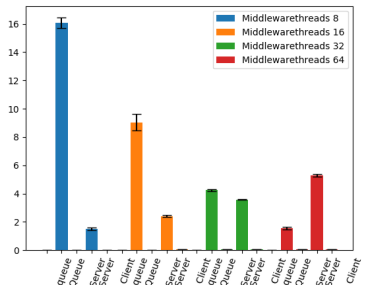
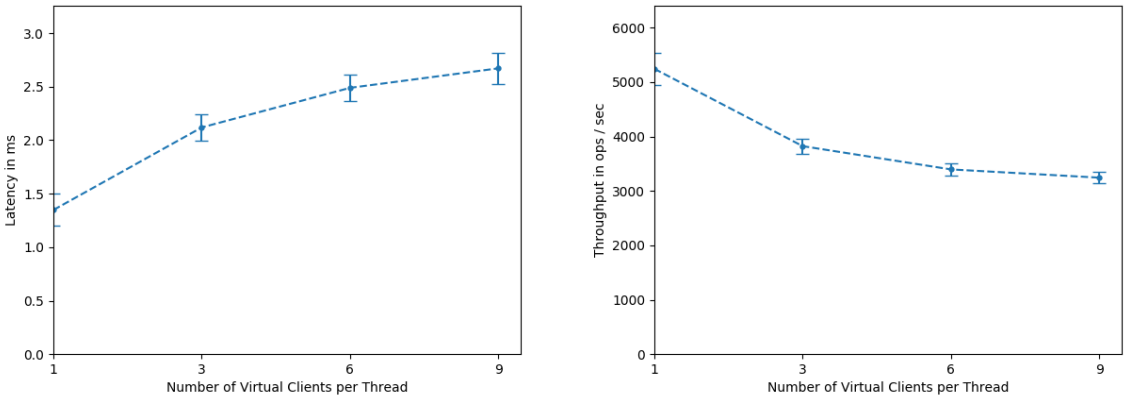


Figure 53: DFT, rupture

8.3 Experiment 5.1 - Multi-key GETs

5.1. throughput and latency as measured by the clients (as opposed to middlewares). This is the sharded case.

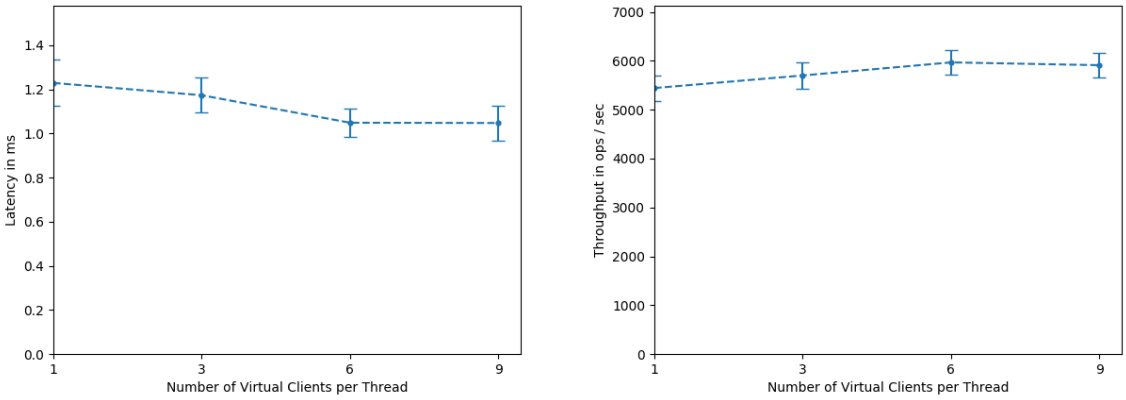


(a) Latency write only

(b) Throughput write only

Figure 54: Exp3.2: Latency and throughputs as measures by the middlewares

5.2 throughput and latency as measured by the clients (as opposed to the middlewares). This is the non-sharded case.



(a) Latency write only

(b) Throughput write only

Figure 55: Exp3.2: Latency and throughputs as measures by the middlewares

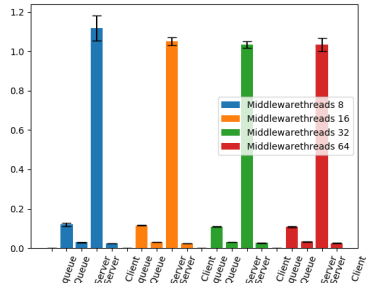


Figure 34: Initial condition

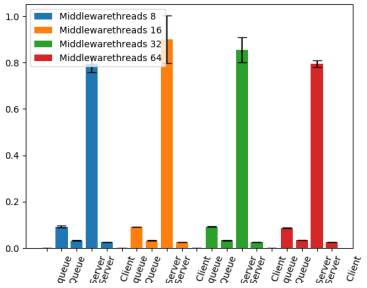


Figure 35: Rupture

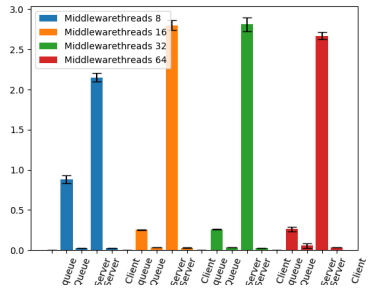


Figure 36: Initial condition

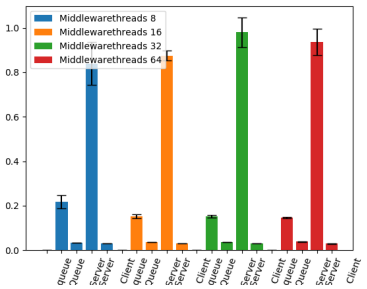


Figure 37: Rupture

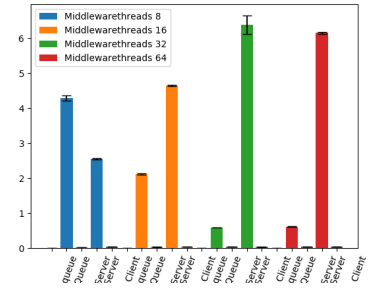


Figure 38: DFT, Initial condition

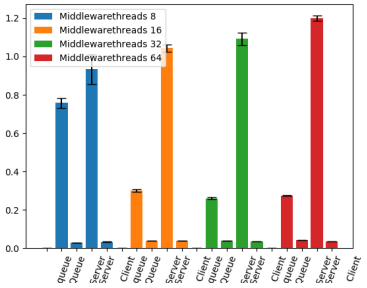


Figure 39: DFT, rupture

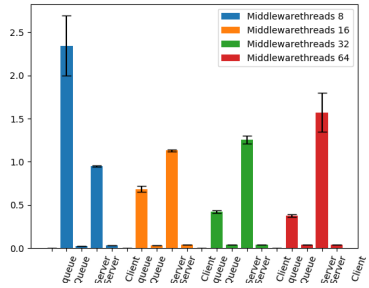


Figure 40: DFT, rupture

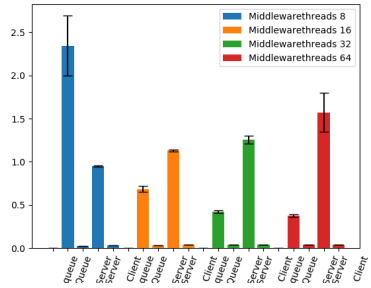


Figure 41: DFT, rupture