Advanced Systems Lab Report

Autumn Semester 2018

Name: Stefano Peverelli
Legi: 19-980-396

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

1 System Overview (75 pts)

Following is a description of the class structure of the Middleware:

- `MW` - responsible for accepting incoming socket connections, instantiating the `Writer` threads, and for enqueuing each request.
- `Request` - this class represent a request object, it contains the request's type and measures (presented below).
- `Worker` - each `Worker` establishes a socket connection with the memchached servers, dequeues a request, performs load balancing, and process the request (sends to servers, handle responses and collects some statistics.
- `Statistic` - container for all measured statistics.
- `Writer` - when shutting down the Middleware, it collects all the statistics from each `Worker`, aggregates them and save them to disk.

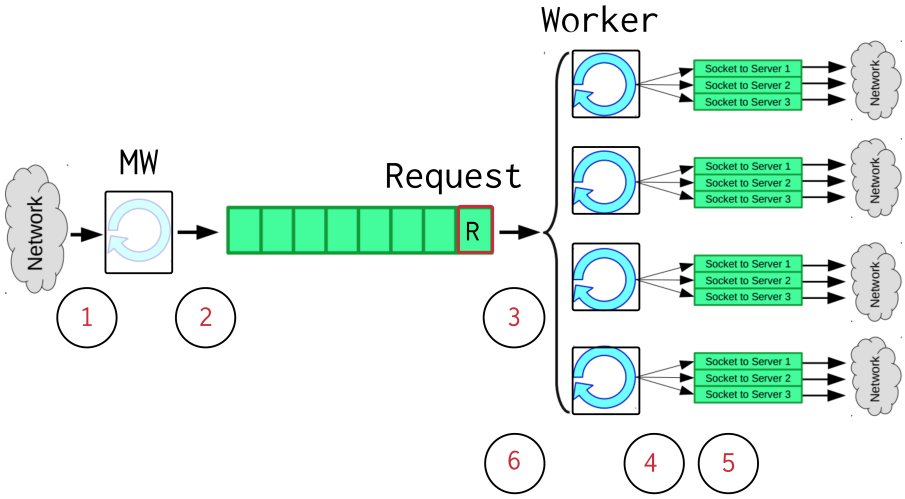


Figure 1: Middleware architecture.

The Middleware is instrumented at six points (shown in Fig.1):

1. R_c - Request created
2. R_e - Request enqueued
3. R_d - Request dequeued
4. R_f - Request forwarded to memcached instances.
5. R_r - Response received from memcached instances.
6. R_a - Request answered to memtier instance.

1.1 Load Balancing

In order to guarantee that each server gets the same amount of jobs, each request's key gets hashed to a specific index that identify a memcached instance.
This is done by method `getServerFromKey` in `Worker` class, and further tested in class `LoadBalancer` of package `test.java.asl`. Following is the result of a test with 1M random strings and 10 servers:

```
Server 0 got 100047 jobs.
Server 1 got 100172 jobs.
Server 2 got 99590 jobs.
Server 3 got 99714 jobs.
Server 4 got 100506 jobs.
Server 5 got 100081 jobs.
Server 6 got 99962 jobs.
Server 7 got 100221 jobs.
Server 8 got 99809 jobs.
Server 9 got 99898 jobs.
```

This indeed shows that the distribution is uniform and that each memcached instance receives the same amount of requests as the others.
NOTE: This is done only for `GET` and *non-sharded* `MULTI-GET`s requests, as `SET`s need to be replicated, and *sharded* `MULTI-GET`s are splitted across memcached instances.

1.2 The system

There are two main components, the `MW` and the `Worker`s. They communicate between each other using a dynamic-sized queue where requests are passed in.

1.3 The queue

The queue is designed to grow as much as needed although in practice it can only grow to the number of clients memtier is using. A fixed-sized queue may have done the job as well, but having a dynamic-sized one has less impact on the memory usage.

1.4 Non-blocking IO

The Middleware communicate both with the clients and the memcached instances via the `java.NIO.SocketChannel`'s library. In `MW`'s constructor `Worker`'s are instantiated and started. Each `Worker`, connects to the memcached instances in a non-blocking fashion.

1.5 Request Protocol

Each request is assumed to be well-formed, and only the first character is checked in order to determine the request type. As a request may be sent into multiple chunks, it is essential to read it without losing any byte. Thisis done by saving the partial content of the `ByteBuffer` the `SocketChannel` has written to, into a `ByteArrayOutputStream`. A request is assumed to be completed when the last bytes are equals to`"\r\n"`. The same assumption is done for responses from the memcached instances, by checking `"STORED\r\n"` or `"END\r\n"`.

1.6 Handling incoming connections

The Middleware listens for incoming connections by memtier clients. This is achieved by using the `java.NIO` package that allows non-blocking IO operations on multiple channels. A `Selector` monitors channels for changes and signal them in a `SelectionKey` object, which contains a set of keys registered with the channel.
Whenever a `SelectionKey`'s interest is set on `ACCEPT`, a `SocketChannel` connection can be established between the client issuing the request and the Middleware. After that, the interest of the `SelectionKey` is set to `READ`, waiting for data from the client.

1.7 Handling incoming requests

When the `SelectionKey`'s interest is set to `READ`, data is read from the `SocketChannel`. From this moment the Middleware starts recording the `responseTime` of the request (Point 1 in Fig.1). When the whole request has been read, it gets enqueued to a `BlockingQueue`, its `queueWaitingTime` is started (Point 2 in Fig.1), and the `SelectionKey`'s interest is set to `WRITE`.

1.8 Forwarding requests

When a new request is ready to be processed by a `Worker`, before sending it to the memcached instances, some operations take place (shown in Fig.1.8):

- The request's `queueWaitingTime` is stopped (Point 3 in Fig.1).
- The request gets copied into a `pendingRequest` object.
- Based on its type, a `multiRequest` object gets created.

Then, when the first entry of the `multiRequest` object gets sent, the `pendingRequest`'s `serviceTime` is started (Point 4 in Fig.1).

1.9 Handling responses

When handling an incoming response each `Worker` does the following:

- Checks if the response is completed
- Increments a counter of the number of responses received, and compares it with the size of the `multiRequest` object created for that `pendingRequest` (expected number of responses).
- Then, in case it has received the expected number of responses:
  - Stops the `pendingRequest`'s `serviceTime` (Point 5 in Fig.1).
  - Creates a `Statistic` object that wraps `pendingRequest` measures.
  - Answers back to the client that issued the request.
  - Stops the `pendingRequests` `responseTime` (Point 6 in Fig.1).

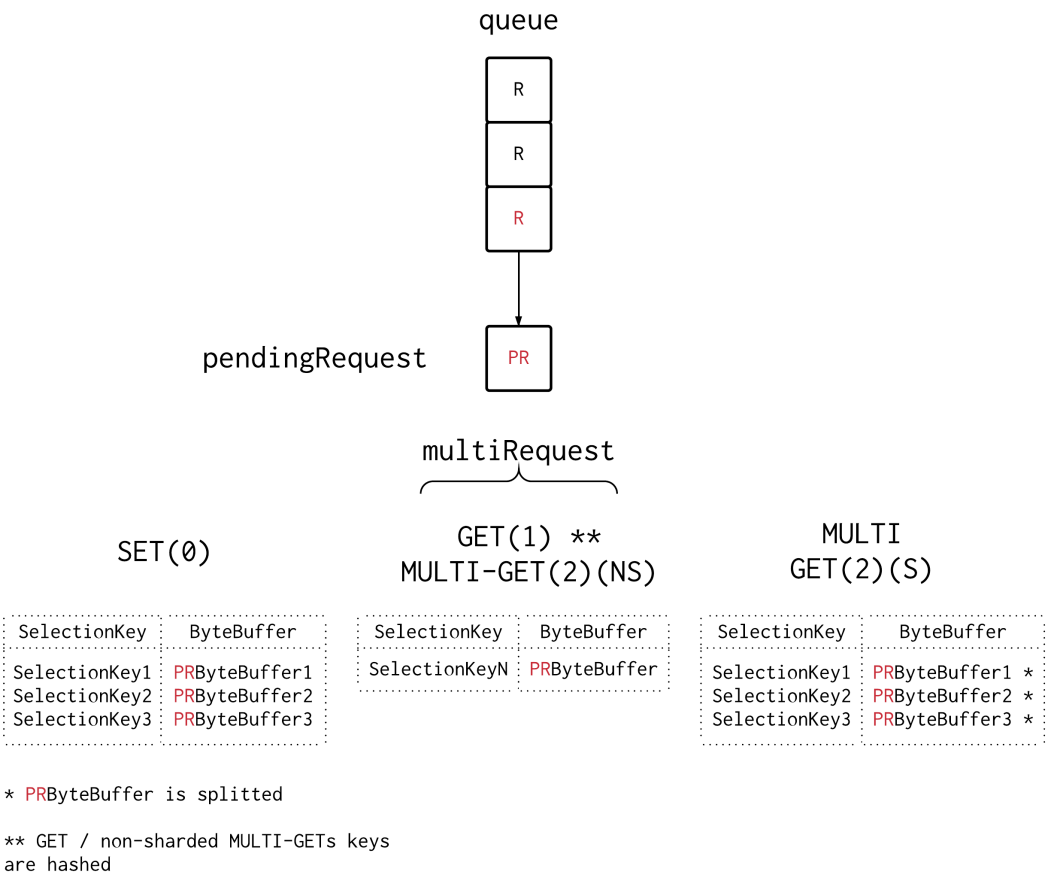


Figure 2: `loadRequest()` behavior.

1.10 Statistics

As already pointed out in 1.9, each `Worker` collects the following statistics:

- The current system's time (`ns`).
- The time the request spent in the queue (`ns`).
- The time elapsed between forwarding the first `multiRequest` and the last response (`ns`).
- The time elapsed between reading the request from the client and the last response (`ns`).
- The current queue size.
- The number of misses of the request.
- The number of keys in the request.

NOTE: All the time-related statistics are then converted in milliseconds (`ms`) when writing them to the logs.

2 Baseline without Middleware (75 pts)

In these experiments we study the performance characteristics of the memtier clients and memcached servers.

2.1 One Server

In this, and in each of the following sections, the number of virtual clients is intended to be the total number of clients in the system, if otherwise, it is emphasized the phrase *per thread*. Bottleneck analysis, and throughput/response time comparsions are shown in each *Explanation* section of each experiment, rather than in the Summary. The focus on the latter is in presenting how the maximum throughput configurations are determined.

2.1.1 Experiment Setting

There are 3 machines that generate write-only and read-only workloads. Each machine runs a single memtier instance with 2 threads, and from a minimum of 1, to 64 virtual clients *per thread* (see table below). The number of virtual clients is the only (tunable) parameter in this setting; it is expected that by raising it, both throughput and response time should grow respectively. The reason behind it is that by raising the number of clients, the server will spend less "idle/free time" between each successive request. The clients machines are connected to a server machine running a single, one-threaded, memcached instance. Each experiment runs for 70 seconds (measures take into account both warm-up and cool-down phases, excluding 10 secs), and is repeated for 3 times under the same exact conditions. By running this benchmark, we hope to find how much load can a single memcached instance sustain.

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[1, 4, 8, 16, 24, 32, 40, 48, 64]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	3 (70 secs, effective 60 secs)
Log Files Path	`experiments/baseline_no_mw_1_server/`
Processed Files Path	`experiments/baseline_no_mw_1_server/out/memtier_data.csv`

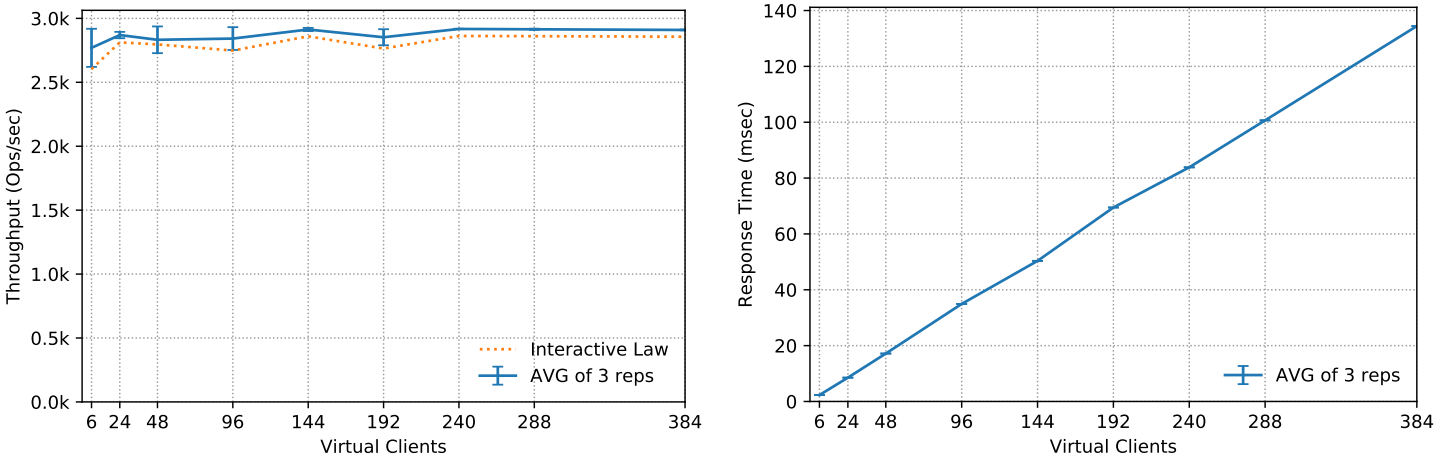
2.1.2 Explanation

The following plot shows how throughput and reponse time behave when increasing the number of virtual clients. The number of virtual clients on the `x-axis` of each plot represents the total number of clients in the system, e.g. the first tick at `VC=6` is the result of `1` virtual client per thread multiplied by the number of threads (`2`), instances(`1`), and client machines(`3`). For each quantity we plot the average measured value of the 3 repetitions, and the standard deviation from the mean as a confidence measure. Additionally as a sanity check, we plot the *Interactive Law* as a dotted line in the troughput plots. In this case it is computed as the invers of the measured response time multiplied by the total number of virtual clients. The throughput shown below is the sum of the throughputs of each memtier instance of each client, while the response time is the average response time of each client's machine.

Read-Only

Consider the throughput below: we can see pretty clearly that, for the whole virtual clients range, apart from `VCs=6`, (in which the memcached server is still **under-saturated**), it essentially remains constant at $\approx$ 3000ops/s. Suprisingly, increasing the number of virtual clients doesn't affect the throughput at all; the server is indeed saturated after 6 virtual clients. Although we have extended the range of virtual clients per thread to 64, we still cannot observe over-saturation of the server.
The same exact behavior can be double checked by looking at the response time plot. It grows almost linearly, confirming the fact that the server is saturated.
As for now, (without additional information both on the arrival rate of requests and the service rate at which they get processed), we cannot determine the bottleneck of the sytem. Either outbound load is limited by the client's bandwidth, or the server's peak performance in reading is 3k ops/s. In Section 2.2 we investigate how much workload a client machine can generate, that is the missing piece of the puzzle to conclude the bottleneck analysis.

Figure 3: Plots for baseline with one server (read-only)



Write-Only

When testing for write-only workloads we observe a completely different behavior with respect to what seen above. Here the throughput follow our initial assumption and grows as the number of virtual clients increases. We can see that at `VCs=240` we reach saturation. We can explain the fact that we reach saturation at a much higher number of VCs compared to the read-only workload, simply because the service rate at which the server process write-only requests, is higher than when reading. This is investigated and further confirmed when measuring service time for different workloads inside the middleware in Section 3. With no surprise, the response time plot reflects the measured throughput: we have an initial under-saturated phase until `VCs=240`, for which the rate $\frac{r_t}{\texttt{VC}}$ monotonically decreases. After that point, the rate stays almost constant, suggesting a flattening:

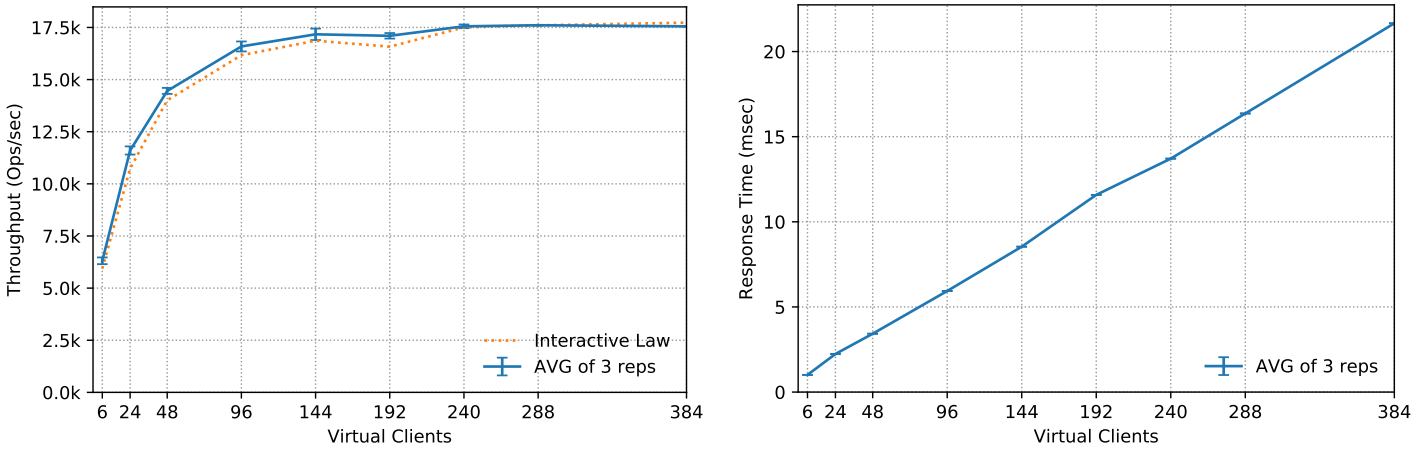
$$\frac{2.5}{24} =\approx 0.1$$

$$\frac{4}{48} =\approx 0.08$$

$$\frac{6}{96} =\approx 0.06$$
$$\frac{12}{192} =\approx 0.06$$
$$\frac{14}{240} =\approx 0.058$$
$$\frac{22}{384} =\approx 0.057$$

In addition, by looking at the interactive law, we can observe that it follows the throughput line precisely, concluding the analysis for the experiment.

Figure 4: Plots for baseline with one server (write-only)



2.2 Two Servers

2.2.1 Experiment Setting

There is a single client machine that generates write-only and read-only workloads. The machine runs two different memtier instances with 1 thread, and from a minimum of 1, to 32 virtual clients (see table below). The number of virtual clients is again, the only (tunable) parameter in this setting; it is expected that by raising it, both throughput and response time should grow accordingly. We connect each client's instance to 2 different server machines running a single, one-threaded, memcached instance. The same exact conditions on the number of repetions, duration and measurement stated before still hold. By running this benchmark, we hope to find how much load can a single memtier client produce before saturation.

Number of servers	2
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1, 4, 8, 16, 24, 32]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	3 (70 secs, effective 60 secs)
Log Files Path	`experiments/baseline_no_mw_2_server/`
Processed Files Path	`experiments/baseline_no_mw_2_server/out/memtier_data.csv`

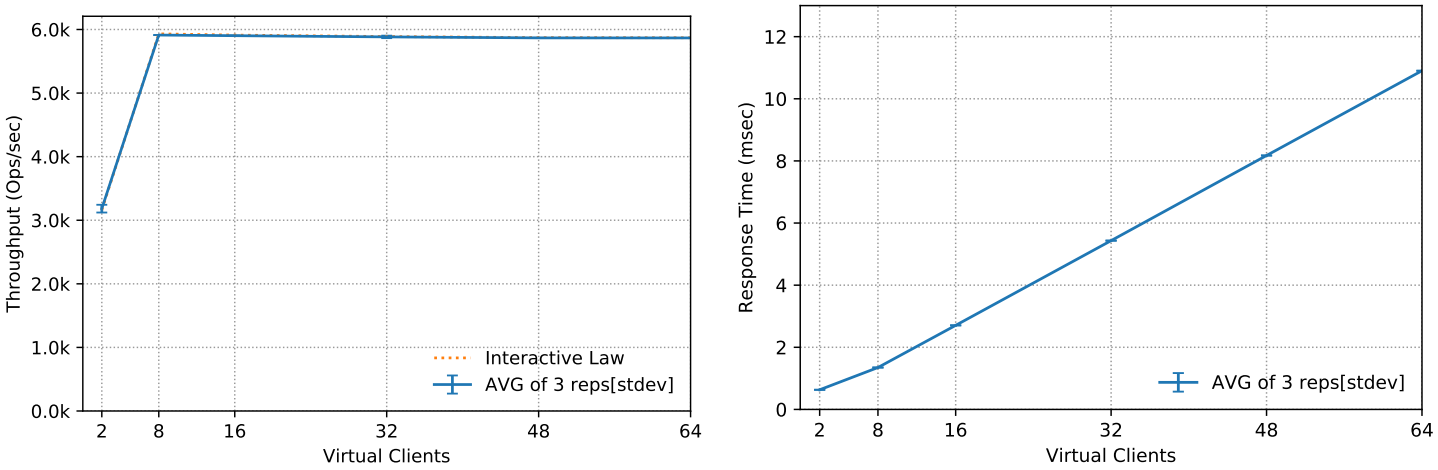
2.2.2 Explanation

Once again we plot both throughput and response time as a function of the total number of clients. It is interesting to note how these quantities are measured. The response time is the mean response time of each memtier instance, for each client machine, for each repetition. The throughput is the sum of the throughputs measured in each instance, summed over the number of client machines, and averaged between each repetition. The error measure introduced is again the standard deviation from the mean value in all repetitions.

Read-Only

The throughput presents an interesting behavior: as before, we observe that it flattens early on, at `VCs=8`, reaching saturation at $\approx$ 6k *ops/s*. From that point on, the system seems to begin to over-saturate, as the throughput slightly decrases.
In Section 2.1 we have 3 clients connected to one server, reaching peak performance at $\approx$ 3k *ops/s*, this suggests that now that we have a single client that can read 6k *ops/s* from two servers, that the bottleneck when reading is effectively the service rate at which a single memcached instance process incoming requests. The response time, after a sub-linear growth between the first and second virtual clients grows linearly, confirming saturation.

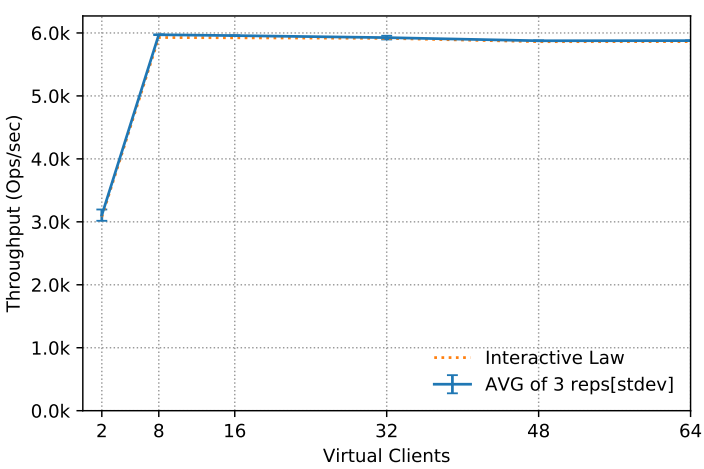
Figure 5: Plots for baseline with two servers (read-only)

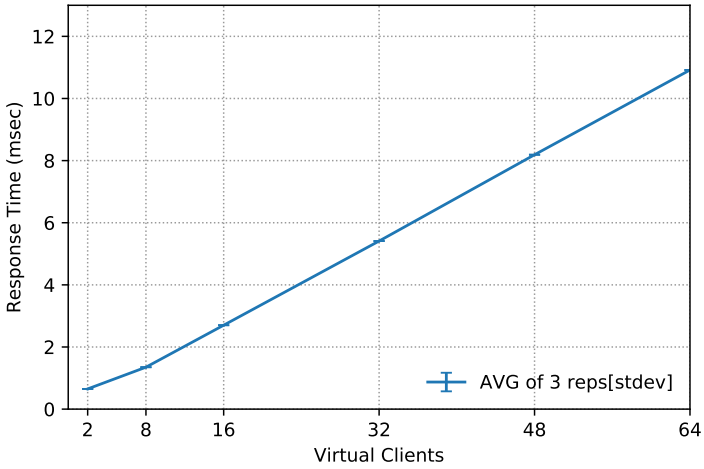


Write-Only

The throughput is almost identical to the read-only workload. Again a pretty fast flattening at `VCs=8`. Let's compare it with the one in Section 2.1: there, we have 3 clients writing to a single server, reaching an $\approx$ 18k *ops/s* peak performance), meaning an approximate throughput per client of 6k *ops/s*. Now by doubling the number of servers, a single client still produces 6k *ops/s*. Thus, we can conclude that the bottleneck is the maximum arrival rate of incoming write-only request from a single client machine, limited at 6k *ops/s*.

Figure 6: Plots for baseline with two servers (write-only)

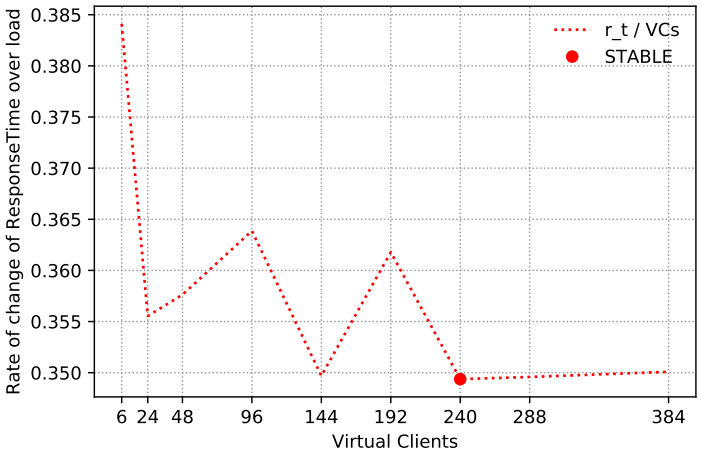




2.3 Summary

Following, the maximum throughput of each experiment for both write-only and read-only workload is shown. In order to determine what is the maximum throughput configuration, we additionally plot the rate of change of the response time over the generated load and pick the configuration from which the rate reaches an horizontal asyntote. Before that point the response time can grow sub-linearly, even oscillating (in the read-only case). So, we choose the first stable virtual client which rate stays almost flat for the entire range of virtual clients to represent the maximum throughput configuration in our system.

Figure 7: Plots for baseline with one server (Rate of change of response time over load, left: read-only, right: write-only)



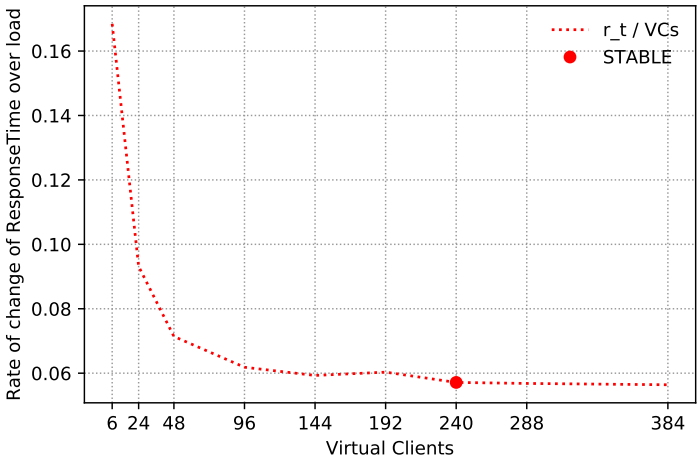
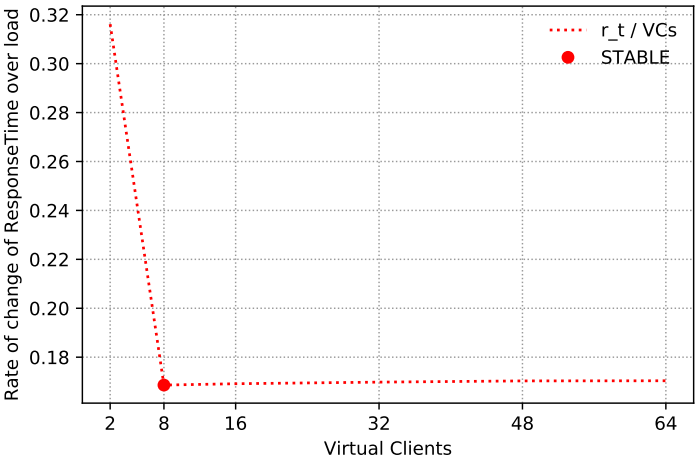
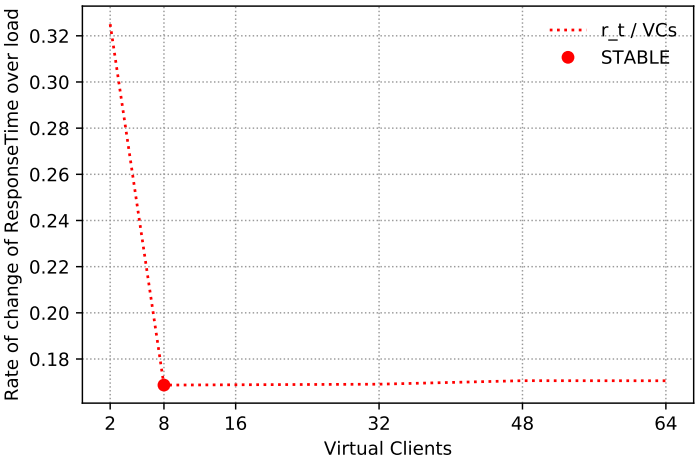


Figure 8: Plots for baseline with two servers (Rate of change of response time over load, left: read-only, right: write-only)





Maximum throughput of different VMs.

	Read-only workload	Write-only workload	Configuration gives max. throughput
One memcached server	2.88k ops/s	17.5k ops/s	40 VCs per thread (240 total clients)
One load generating VM	5.87k ops/s	5.98k ops/s	4 VCs per thread (8 total clients)

3 Baseline with Middleware (90 pts)

In this set of experiments we test the performance of the Middleware/s. We measure and display the results both at the client, and inside the middleware, and compare them. Additionally, in order to gain a better insight on the internal performance breakdown of the middleware, and run analysis on the components, we present the statistics introduced in Section 1.10.

3.1 One Middleware

3.1.1 Experiment Setting

There are 3 clients machine generating both write-only and read-only workload. Each client machine runs a single memtier instance with 1 thread, and from a minimum of 1, to 64 virtual clients (see table below). Each client machine is connected to the the middleware, which is connected to a single, one-threaded, memcached machine. Essentially we are tuning the load the clients produce, and the service rate at which the middleware dispatches request between client and server machines.

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[1, 4, 8, 16, 24, 32, 64]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1
Worker threads per middleware	[8, 16, 32, 64]
Repetitions	3 (70 secs, effective 60 secs)
Log Files Path	`experiments/baseline_mw_1_mw/logs`
Processed Files Path	`experiments/baseline_mw_1_mw/out/memtier_data.csv`

3.1.2 Explanation

We plot throughput and response time as measured at the client and inside the middleware, for an increasing number of both virtual clients and worker threads.
The measurements conducted in the middleware are grouped in five-seconds windows for a total of 14 per experiment ($70.0/5.0 = 14$). Everytime a request has been processed, and a response has been sent back to the client machines, the measurements for the specific request, and a snapshot of the system components, gets saved and later, processed and logged.
Measurements taken inside the middleware, do not include the latency between the client machines and the middleware, so there's a gap between the two, that remains constant and negligible ($\approx 2ms$).

Read-Only

As in Section 2.1, the throughput plot shows a flattening at $\approx$ 3k op/s starting from `VCs=24`. The presence of the middleware doesn't seem to affect the performance at all, moreover we cannot spot any differences when changing the number of worker threads inside the middleware. The unique possible explanation, (as we can exclude the fact that the client machines are the bottleneck, from what we have concluded in **??**), is again. the fact that a single memcached machine cannot process more than $\approx$ 3k ops/s incoming read-only requests.

Figure 9: Plots for baseline with one Middleware (read-only CLIENT)

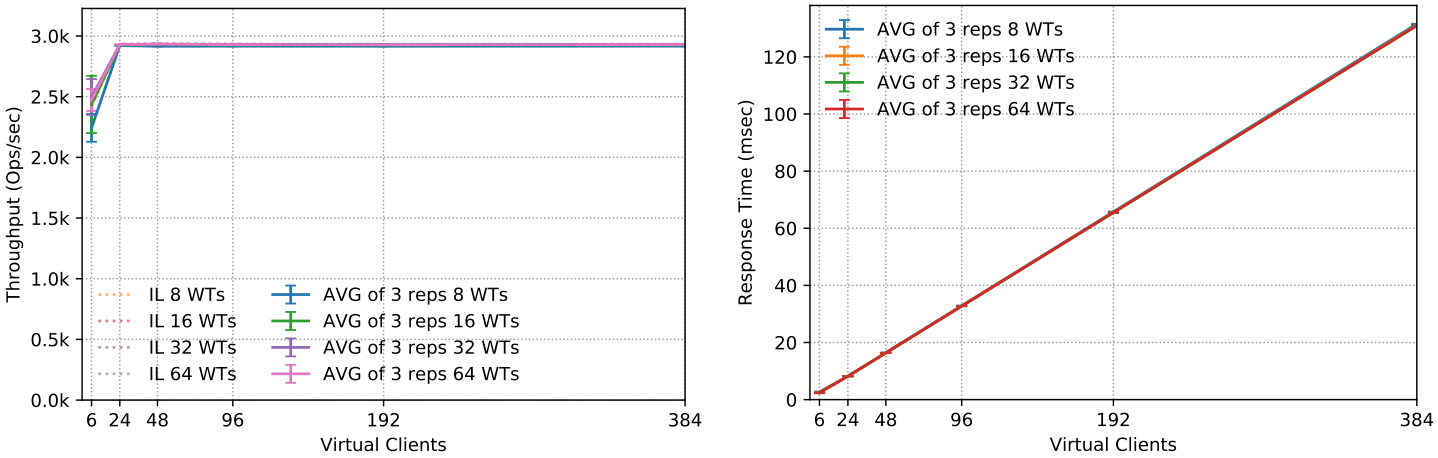
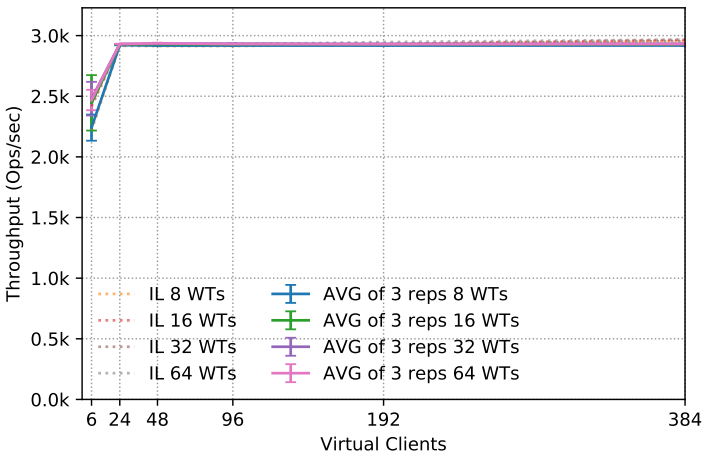


Figure 10: Plots for baseline with one Middleware (read-only MIDDLEWARE)



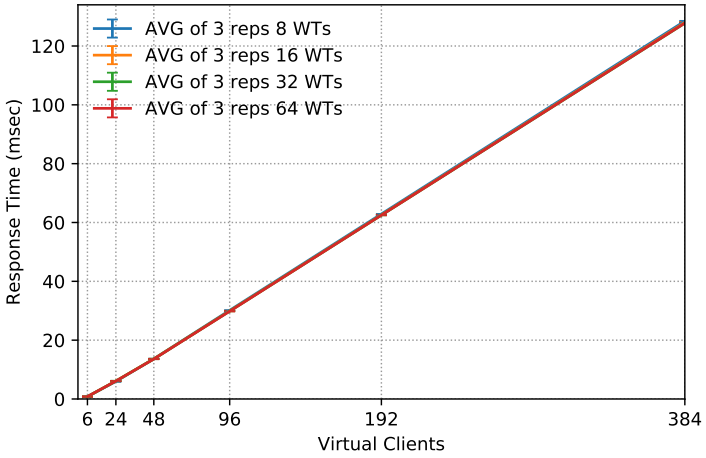
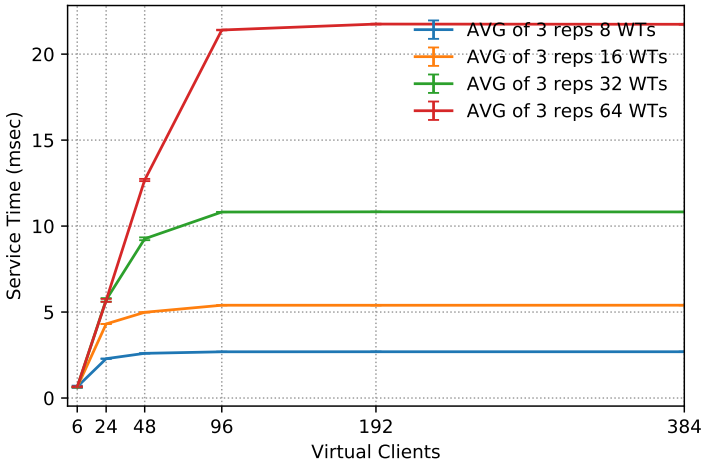
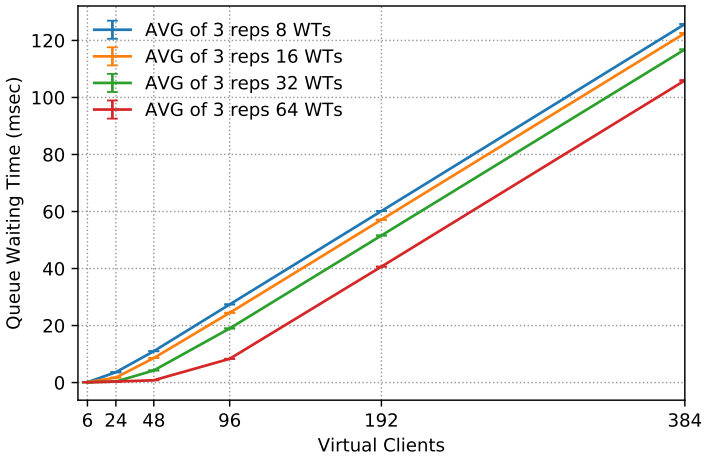


Figure 11: Plots for baseline with one Middleware, Service Time and Queue Waiting Time (read-only MIDDLEWARE)





Write-Only

As expected, increasing the number of worker threads in the middleware, results in an increase of the throughput. Even though, for every distinct worker thread configuration, we observe a distinct throughput behavior, there's a common treat that each configuration seems to manifest. In fact we can clearly identify and split the plot in two phases: one that goes from 6 to 192 `VCs`, and another one, that goes from 192 to 384 `VCs`. In the former (under-saturated phase), the configurations with 16, 32 and 64 workers, behave almost identically (apart from when we reach `VCs=96`, where we observe a difference of $\approx$ 500-600 ops/s). The latter phase shows a much larger difference in performance for different worker threads configuration. We observe a clear distinction between 8 and 16 worker threads, and between 16 and 32, but not as much between 32 and 64 worker threads.

Figure 12: Plots for baseline with one Middleware (read-only MIDDLEWARE)

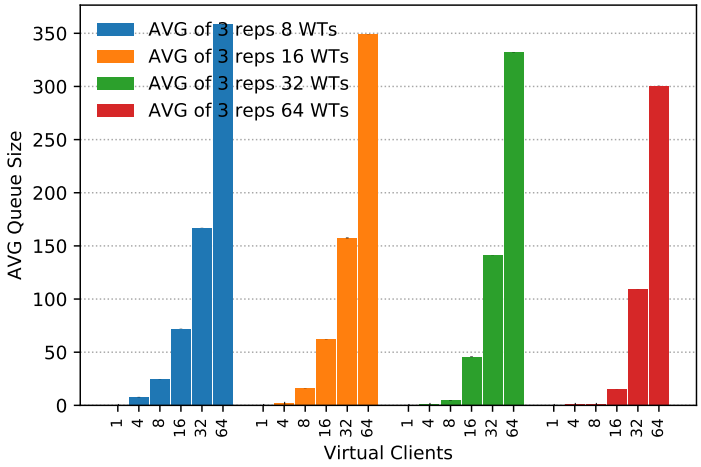


Figure 13: Plots for baseline with one Middleware (write-only CLIENT)

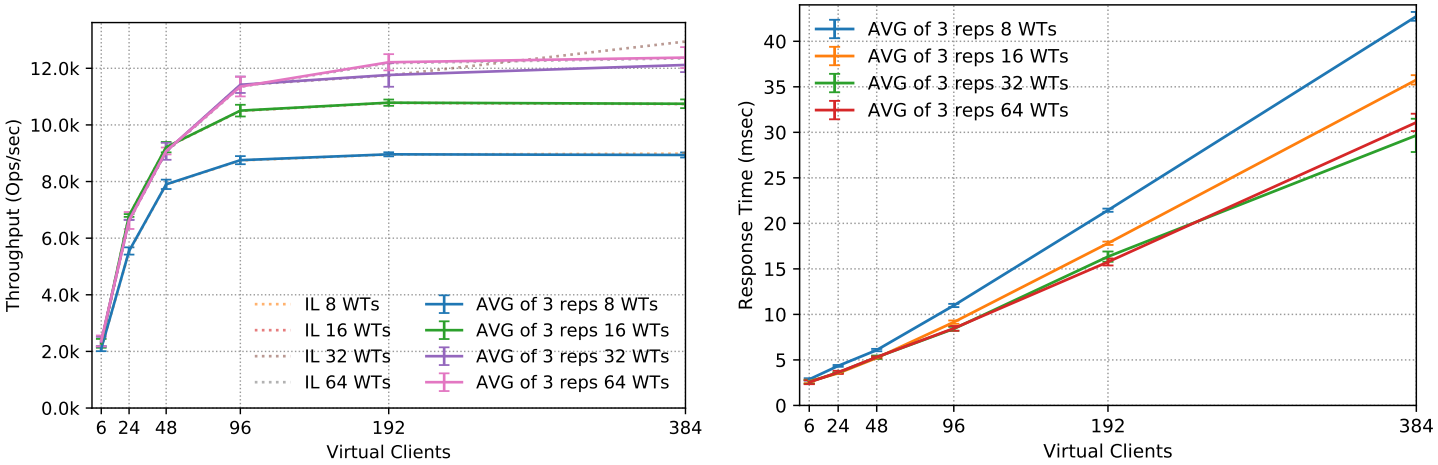


Figure 14: Plots for baseline with one Middleware (write-only MIDDLEWARE)

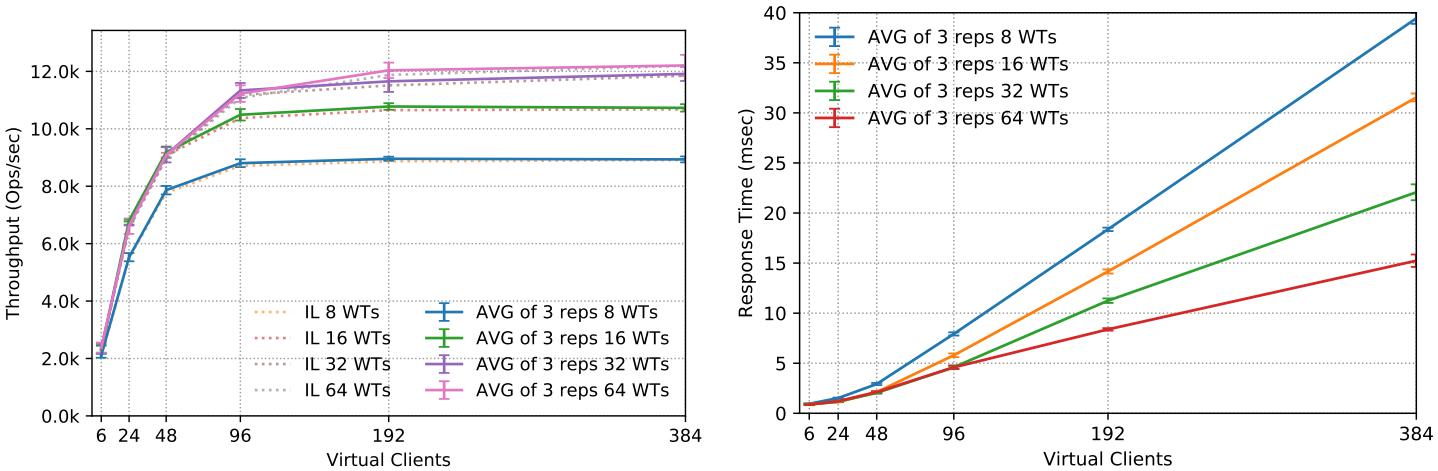


Figure 15: Plots for baseline with one Middleware, Service Time and Queue Waiting Time (write-only MIDDLEWARE)

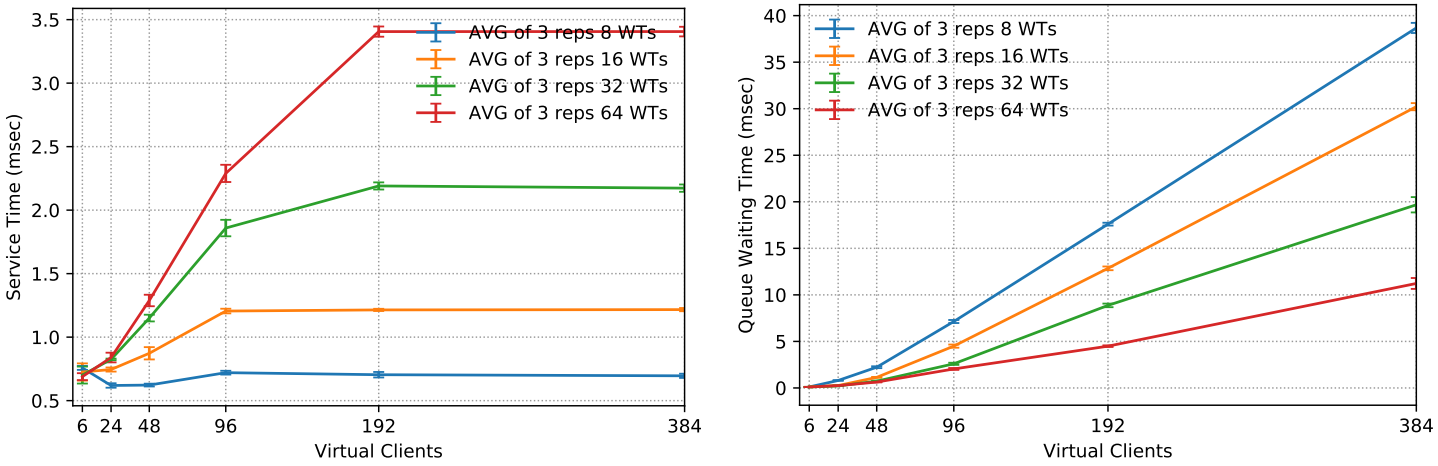
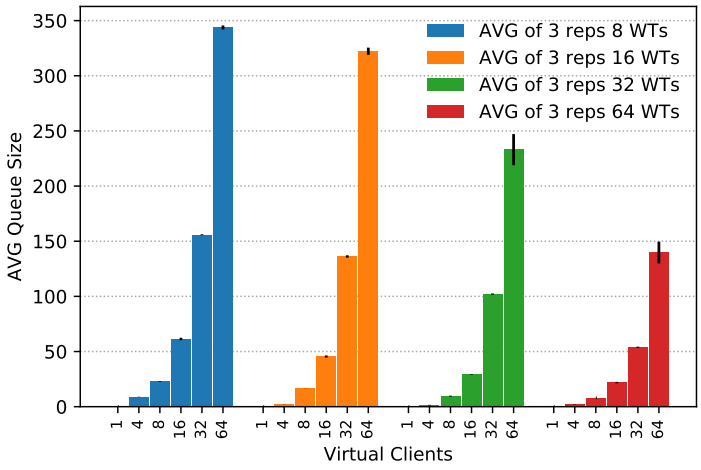


Figure 16: Plots for baseline with one Middleware, Average Queue Size (write-only MIDDLEWARE)



3.2 Two Middlewares

Connect three load generator machines (two instances of memtier with CT=1) to two middlewares and use 1 memcached server. Run a read-only and a write-only workload with increasing number of clients (between 2 and 64) and measure response time *both at the client and at the middleware*, and plot the throughput and response time as measured in the middleware.

Repeat this experiment for different number of worker threads inside the middleware: 8, 16, 32, 64.

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
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8..64]
Repetitions	3 or more (at least 1 minute each)

3.2.1 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.

3.3 Summary

Based on the experiments above, fill out the following table. For both of them use the numbers from a single experiment to fill out all lines. Miss rate represents the percentage of GET requests that return no data. Time in the queue refers to the time spent in the queue between the net-thread and the worker threads.

Maximum throughput for one middleware.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware				
Reads: Measured on clients			n/a	
Writes: Measured on middleware				n/a
Writes: Measured on clients			n/a	n/a

Maximum throughput for two middlewares.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware				
Reads: Measured on clients			n/a	
Writes: Measured on middleware				n/a
Writes: Measured on clients			n/a	n/a

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.

4 Throughput for Writes (90 pts)

4.1 Full System

Connect three load generating VMs to two middlewares and three memchached servers. Run a write-only experiment. You need to plot throughput and response time measured on the middleware as a function of number of clients. The measurements have to be performed for 8, 16, 32 and 64 worker threads inside each middleware.

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1..32]
Workload	Write-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8..64]
Repetitions	3 or more (at least 1 minute each)

4.1.1 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.

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

	WT=8	WT=16	WT=32	WT=64
Throughput (Middleware)				
Throughput (Derived from MW response time)				
Throughput (Client)				
Average time in queue				
Average length of queue				
Average time waiting for memcached				

Based on the data provided in these tables, draw conclusions on the state of your system for a variable number of worker threads.

5 Gets and Multi-gets (90 pts)

For this set of experiments you will use three load generating machines, two middlewares and three memcached servers. Each memtier instance should have 2 virtual clients in total and the number of middleware worker threads is 64, or the one that provides the highest throughput in your system (whichever number of threads is smaller).

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

5.1 Sharded Case

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

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)

5.1.1 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.

5.2 Non-sharded Case

Run multi-gets with 1, 3, 6 and 9 keys (memtier configuration) with sharding disabled. Plot average response time as measured on the client, as well as the 25th, 50th, 75th, 90th and 99th percentiles.

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)

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, display four histograms representing the sharded and non-sharded response time distribution, both as measured on the client, and inside the middleware. Choose the bucket size in the same way for all four, and such that there are at least 10 buckets on each of the graphs.

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)

We perform a $2^k r$ experimental analysis to investigate the effect of ($k = 3$) **factors**, namely:

- The number of memcached servers
- The number of middlewares
- The number of worker threads per middleware

and study the effects of each of them on throughput and response time respectively (**response variables**).

6.1 Experiment setting

There are 3 machines that generate workloads. Then based on the number of middlewares (1 or 2), each client machine runs (1 or 2) instances of memtier, each with 32 virtual clients and (2 or 1) threads respectively. In the case of two memtier instances per machine, each of them connects to a different middleware. Each middleware runs either (8 or 32) worker threads, and is connected to either (2 or 3) mecached machines. We measure both throughput and response time for each workload (read-only and write-only). We investigate experimental errors by replicating each experiment 3 times ($r = 3$), so the response variable, is simply the average between each repetition. Based on results on experiments in section 3, we do expect the number of middlewares and the number of worker threads, to have relevant impact on the response variables. Furthermore, we do assume that the effects of the factor are additive and that measurements error are independent and follow a normal distribution.

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

6.2 Model

We model the response variable y as

$$y = q_0 + q_A x_A + q_B x_B + q_C x_C + q_{AB} x_A a_B + q_{AC} x_A x_C + q_{BC} x_B x_C + q_{ABC} x_A x_B x_C + \epsilon \quad (1)$$

where x_A, x_B, x_C are defined as

$$x_A = \begin{cases} -1 & \text{if 1 server} \\ 1 & \text{if 3 servers} \end{cases}$$

$$x_B = \begin{cases} -1 & \text{if 1 middleware} \\ 1 & \text{if 2 middlewares} \end{cases}$$

$$x_C = \begin{cases} -1 & \text{if 8 worker threads} \\ 1 & \text{if 32 worker threads} \end{cases}$$

and $q_0, q_A, \ldots q_{ABC}$, and ϵ (**experimental error**) are the parameters to be computed with the *Sign Table Method*.

6.3 Results

6.3.1 Throughput

6.3.1.1 Read-Only

i	I	A	B	C	AB	AC	BC	ABC	y_1	y_2	y_3	$\hat{y}$
1	1	-1	-1	-1	1	1	1	-1	0.0	0.0	0.0	0.0
2	1	-1	-1	1	1	-1	-1	1	0.0	0.0	0.0	0.0
3	1	-1	1	-1	-1	1	-1	1	0.0	0.0	0.0	0.0
4	1	-1	1	1	-1	-1	1	-1	0.0	0.0	0.0	0.0
5	1	1	-1	-1	-1	-1	1	1	0.0	0.0	0.0	0.0
6	1	1	-1	1	-1	1	-1	-1	0.0	0.0	0.0	0.0
7	1	1	1	-1	1	-1	-1	-1	0.0	0.0	0.0	0.0
8	1	1	1	1	1	1	1	1	0.0	0.0	0.0	0.0

6.3.1.2 Write-Only

i	I	A	B	C	AB	AC	BC	ABC	y_1	y_2	y_3	$\hat{y}$
1	1	-1	-1	-1	1	1	1	-1	0.0	0.0	0.0	0.0
2	1	-1	-1	1	1	-1	-1	1	0.0	0.0	0.0	0.0
3	1	-1	1	-1	-1	1	-1	1	0.0	0.0	0.0	0.0
4	1	-1	1	1	-1	-1	1	-1	0.0	0.0	0.0	0.0
5	1	1	-1	-1	-1	-1	1	1	0.0	0.0	0.0	0.0
6	1	1	-1	1	-1	1	-1	-1	0.0	0.0	0.0	0.0
7	1	1	1	-1	1	-1	-1	-1	0.0	0.0	0.0	0.0
8	1	1	1	1	1	1	1	1	0.0	0.0	0.0	0.0

6.3.2 ResponseTime

6.3.2.1 Read-Only

i	I	A	B	C	AB	AC	BC	ABC	y_1	y_2	y_3	$\hat{y}$
1	1	-1	-1	-1	1	1	1	-1	0.0	0.0	0.0	0.0
2	1	-1	-1	1	1	-1	-1	1	0.0	0.0	0.0	0.0
3	1	-1	1	-1	-1	1	-1	1	0.0	0.0	0.0	0.0
4	1	-1	1	1	-1	-1	1	-1	0.0	0.0	0.0	0.0
5	1	1	-1	-1	-1	-1	1	1	0.0	0.0	0.0	0.0
6	1	1	-1	1	-1	1	-1	-1	0.0	0.0	0.0	0.0
7	1	1	1	-1	1	-1	-1	-1	0.0	0.0	0.0	0.0
8	1	1	1	1	1	1	1	1	0.0	0.0	0.0	0.0

6.3.2.2 Write-Only

i	I	A	B	C	AB	AC	BC	ABC	y_1	y_2	y_3	$\hat{y}$
1	1	-1	-1	-1	1	1	1	-1	0.0	0.0	0.0	0.0
2	1	-1	-1	1	1	-1	-1	1	0.0	0.0	0.0	0.0
3	1	-1	1	-1	-1	1	-1	1	0.0	0.0	0.0	0.0
4	1	-1	1	1	-1	-1	1	-1	0.0	0.0	0.0	0.0
5	1	1	-1	-1	-1	-1	1	1	0.0	0.0	0.0	0.0
6	1	1	-1	1	-1	1	-1	-1	0.0	0.0	0.0	0.0
7	1	1	1	-1	1	-1	-1	-1	0.0	0.0	0.0	0.0
8	1	1	1	1	1	1	1	1	0.0	0.0	0.0	0.0

7 Queuing Model (90 pts)

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

7.1 M/M/1

Build queuing model based on Section 4 (write-only throughput) for each worker-thread configuration of the middleware. Use one M/M/1 queue to model 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

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.