Advanced Systems Lab Report

Autumn Semester 2018

Name: Doruk Çetin
Legi: dcetin
Student Number: 18-947-382

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

1 System Overview

We begin our introduction by first presenting the middleware system and its components (Section 1.1) and then discussing our methodology (Section 1.2) before moving on to the experiments and their results.

1.1 System Design

This section details the inner structure of the middleware. Aim of this middleware is to collect and forward queries from clients to servers, wait for the responses and reply the clients back — all under specified conditions and configurations. Aforementioned servers and clients are the single-threaded instances of Memcached [1] and of memtier_benchmark [2], respectively. The design of the middleware spans several classes and functions that implement different functionalities. The classes operate in a multithreaded fashion so the different functionalities can work together asynchronously. Middleware continuously collect and aggregate statistics so that we can interpret and characterize different workload conditions and experimental configurations.

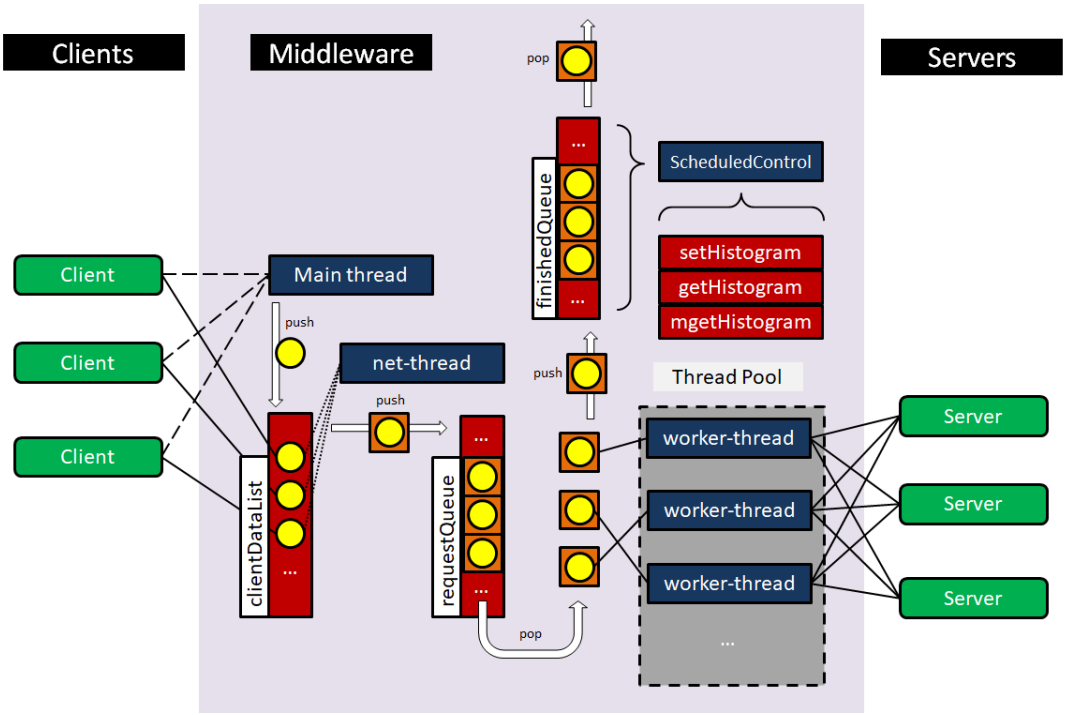


Figure 1: A simplified diagram of the system showing the inner structure of the system and the interaction of its components in an informal manner. Yellow circles represent the clientData objects and the orange squares represent the requestData objects, whose instances encapsulate the ones of the former. Dark blue rectangles represent the threads in the system. Red rectangles represent the shared lists and queues which form the pipeline and help aggregation of the statistics. Green rounded rectangles illustrate the external systems, namely the clients and servers.

1.1.1 Main thread

Main thread of the MyMiddleware class is responsible for setting up the working environment for the whole system. It first initializes the shared data structures that are to be used by other threads; namely, these are clientDataList (which stores information about each one of the connected clients), requestQueue (which is a first-in, first-out queue for queries awaiting execution) and finishedQueue (which temporarily stores the completed requests for the aggregation of their statistics). It then creates a clientHandler (so-called "net-thread", which is responsible for handling communication with clients) and the specified number of serverHandler threads (so-called "worker-threads", which are responsible with for handling communication with the servers). Throughout this report, names clientHandler and net-thread will be used interchangeably. Same also applies for worker-threads and serverHandlers. Lastly, the main thread creates the welcomingSocket and listens for incoming client connections in a loop. As soon as such a connection request is received, main thread creates the respective clientData object and pushes it into the clientDataList. welcomingSocket has a backlog queue for 512 connections, so it is expected to handle at most 512 concurrent incoming connections. This size is more than we will ever need with our experiments and provided as a safety measure.

clientData objects store the information relevant for communicating with the clients. They consist of the net socket, its respective reader and writer streams, an identifier number and lastly a flag that denotes if the client has already sent its request and awaiting for reply.

1.1.2 clientHandler (net-thread)

The clientHandler constantly iterates over the clientDataList, waiting for new requests from clients. It employs busy waiting as it non-blockingly check each client to see if there is an available request. As soon as it receives its first query from a client, clientHandler initializes the ScheduledControl thread, which is responsible for periodically aggregating statistics of the system. clientHandler checks for each client if it has not been waiting for a repy and data is available in its reader stream. That means that client has a new request so the clientHandler creates a new RequestData object and pushes it into the requestQueue for it to be handled by the worker-threads. clientHandler thread does not parse or modify the requests and directly transfer them to the worker-threads.

At the beginning of each iteration over the clientDataList, clientHandler compares the time with the timestamp of the last received query. If it is larger than some safe threshold, that means no other requests will arrive for our working conditions, so it moves on to close the system killing all threads. Before exiting clientHandler prints the response time histograms created by the ScheduledControl thread.

requestData objects contain the necessary relevant information about a request so that it can be executed correctly and its statistics can be aggregated with ease. Aside from information about its respective client, it stores an identifier number, the type of the request (set, get or multi-get), which server the request is sent to (if it is a get or multi-get request), which worker-thread handled the request, how many items were requested and received (if it is a get or multi-get request) and its timestamps.

Timestamps are all obtained in nanoseconds using Java's `System.nanoTime`. ns_netThreadReceived marks the time net-thread received the request from the client. ns_workerThreadReceived marks the time the request has exited the request queue to be handled by a worker-thread. The difference of those two timestamps give the **waiting time in the queue**. Lastly, there is ns_workerThreadFinished that marks the worker-thread received an answer from the servers (either successful or not) and replied to the querying client for the request. The difference of this last two timestamps (namely ns_workerThreadFinished and ns_workerThreadReceived)

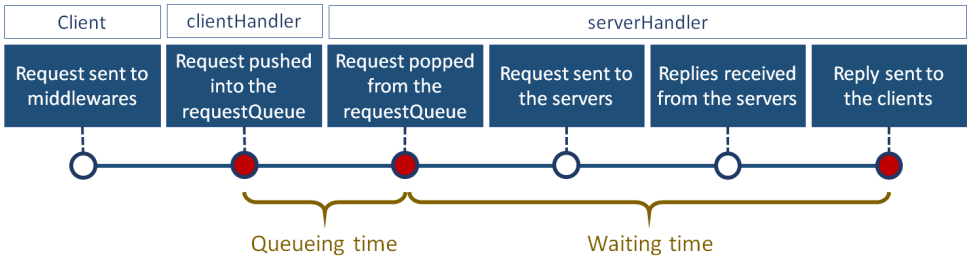


Figure 2: A diagram showing the timestamping of the requests. Line shows the lifetime of a request where the red circles denote the actions where a timestamp is collected. Names above show where the specified actions take place.

gives the **service time of the memcached servers**. The terms queueing time and waiting time in the discussions that take place in the later sections refer respectively to waiting time in the queue and the service time of the memcached servers. Figure 2 illustrates where the timestamps are collected and how these times are calculated.

1.1.3 serverHandlers (worker-threads)

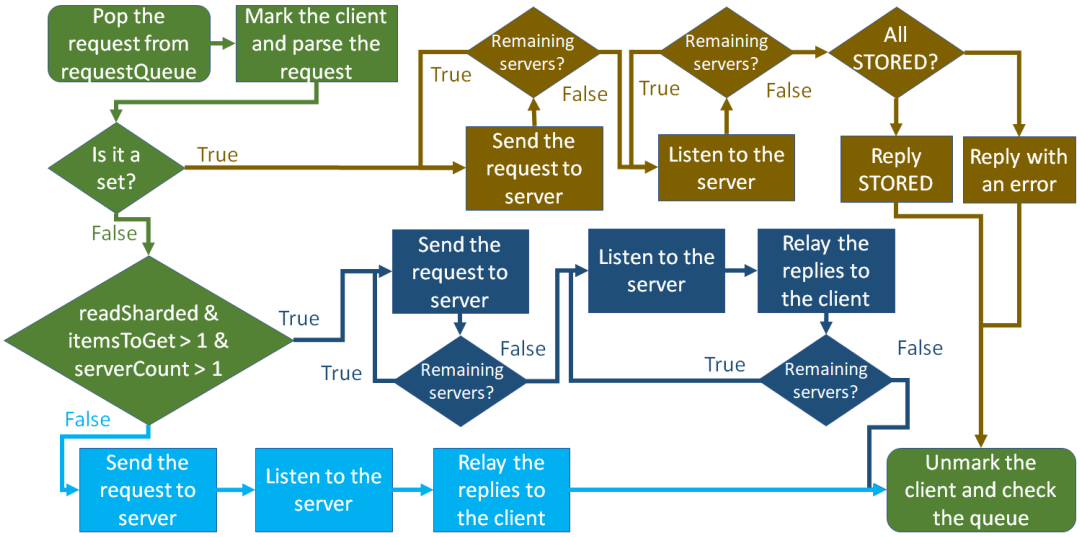


Figure 3: Flowchart showing one iteration of a worker thread in its infinite loop. States are color coded where the gold, light blue and dark blue states are respectively for set, get and multi-get operations. Green states denote states that are common for multiple types of operations. Each thread run its own round-robin over the available servers.

serverHandler threads initially set up dedicated connections to each server with their reader and writer streams. They also operate in an infinite loop, waiting for new request to take from the requestQueue in a blocking manner. It is blocking as a worker thread has no other jobs than handling requests. As soon as there is an available request in the queue the worker-thread wakes up, marks the time and unpacks the requestData structure. **Worker-threads are the ones that parse the messages**, so the certain specifics of the requests are unknown to the system until they exit the queue. After it has been handled accordingly to its type, the resulting requestData is pushed into the finishedQueue.

If it is a set query, the request is sent to all of the servers and reported as successful to the client only if all the responses indicate so. If not, one of the error messages are relayed to the sender client. If it is a get request there are two possibilities. If it is a multi-get request in a multi-server setting with `readSharded` option provided as true, then it is sharded into smaller requests. If the query requests a single object, there is a single server or `readSharded` is provided as false, then the request will be sent to only one server. In either case, middleware employs a simple load balancing scheme through a round-robin scheduling. It iterates over all servers while sending get requests or the sharded get requests. For both set and get requests middleware first sends the requests to servers and then collect the responses. Flowchart illustrating the state flow of the system when handling different types of operations can be viewed in Figure 3. Middleware outputs for the later experiments confirm that the workload is indeed distributed equally to multiple servers, as seen on Figure 4.

Figure 4: Distribution of the requests to three servers over three repetitions. Each of the plots illustrate a different workload condition. Results are taken from the experiments of later sections. First two plot use the results from 2K experiments 6 and the last plot use those from gets and multi gets experiments 5. Experiment with read-only and write-only loads inspect 1 middleware with 8 threads, whereas the sharded read experiment is plotted using aggregated results for the get requests in one of the two middlewares, on 1:6 workload ratio.

1.1.4 ScheduledControl

secs	Second (timestamp)
qlen	Queue length
thru	Throughput
msrt	Miss rate
items	Total number of individual gets
nset	Number of set requests
nget	Number of get requests

nmget	Number of multi-get requests
sqt	Mean set queueing time
gqt	Mean get queueing time
mqt	Mean multi-get queueing time
swt	Mean set waiting time
gwt	Mean get waiting time
mwt	Mean multi-get waiting time

Table 1: Abbreviations for middleware aggregation output.

ScheduledControl is started by the net-thread as it receives its first request. It periodically wakes up, aggregates the statistics over the most recent requests, then sleeps again. It goes

over the entries in finishedQueue in its each run, collecting type-specific statistics. **For all our experiments, the aggregation period is set as 1 second**.

The thread aggregates the number of requests, queue waiting times and server waiting times separately for set, get and multi-get requests. For multi-get and get times, overall miss-rate is also aggregated together. Miss-rates are calculated over the total number of distinct get requests, breaking apart the multi-gets. For example, for two multi-gets with 5 keys if one item in one multi-get is a cache miss, the miss-rate is reported as 10%, not as 50%.

ScheduledControl also keeps track of the distribution of the response times for different types of requests by constructing a separate histogram for each type of request, namely sets, gets and multi-gets. While going over the list of requests the ScheduledControl assigns each of the requests to the appropriate histogram bin with respect to its response time and type. Lastly, length of the requestQueue is saved in each run of the ScheduledControl.

Middleware outputs each second (if aggregation period is set as default) a line summarizing various statistics for that second. The output looks like this:

```
secs,qlen,thru,msrt,items,nset,nget,nmget,sqt,gqt,mqt,swt,gwt,mwt
```

Definition of each abbreviation is given in Table 1.

1.2 Methodology

Three repetitions were seem to be sufficient all the experiments as both our middleware and the external systems it communicated had shown performances that stayed fairly stable under repetitions. Similarly, trimming five seconds of warm-up and cool-down periods, respectively at the beginning and the end of experiments, was found as sufficient through the initial exploratory experiments performed. Clients can lag behind around at most one second per a three repetition run, but it is still negligible with the times cut for warm-up and cool-down periods. Baseline without middleware experiments (Section 2) were run for 100 seconds per each iteration, yielding a one and a half minute period after cutting the warm-up and cool-down times. Experiments involving middleware (Sections 3, 4, 5 and 6 consists of three repetitions of 70-second runs yielding a 60 second stable period for aggregation.

iPerf3 [4] is used to measure the maximum achievable bandwith in between different machines in our environment. The results can be seen in Table 2. For each machine in every experiment, we collect data through dstat [3]. dstat is run in its default mode, so it collects statistics cpu, disk, network, paging and system statistics. It collects data every second after an initial delay of one second — similar to the ScheduledControl in our middleware.

Machine	Bandwidth (Mbps)	Bandwidth (MB/s)	Effective bandwidth* (MB/s)
Client	200	25	~ 24
Middleware	800	100	~ 96
Server	100	12.5	~ 12

*: for read-only and write-only workloads

Table 2: Maximum achievable bandwidth values for all machines, obtained using iperf3. So-called "effective bandwidth" is the maximum bandwidth a machine can utilize only with the set queries or get replies. It is an informal measure that comes as useful while analyzing read-only and write-only workloads in later sections.

We illustrate the dispersion over different repetitions of individual experiments through standard variance. For every plot, error bars indicate a range of one standard variance calculated over the repetitions. At times such error bars may not be visible to the reader, indicating a fairly robust experimental setup. Most of the plots have their x-axis as the number of clients. It

is the effective number of different clients and is simply calculated by multiplying four numbers: number of virtual clients per thread, number of threads per memtier instance, number of memtier instances per virtual machine and the number of client machines. Using the abbreviations in Table 32 the formula could be written as $numClients = vcli \times tcli \times icli \times ncli$. The term latency is used interchangeably with the term response time. Plots specify the source of measurements as middlewares or clients in their subtitles, so it is easy to find for a plot whose perspective it reflects. It should be also noted here that for each separate experiment where the number of virtual clients are varied, we always make sure that at least 6 different values for virtual clients per client thread are explored.

For every experiment, interactive law is verified by both calculating the throughput from the response times and vice versa. As it would approximately double all the plots we provide here to show that the interactive law indeed holds for all the experiments, we only provide the illustration of this guarantee in Section 2. Figures 5, 6, 7 and 8 show the actual and predicted values coincide perfectly. Reader should feel free to run the provided scripts for other experiments and see the interactive law in action in other experiments as well.

In experiments involving the get operations the servers are always populated beforehand so miss rates are either realistic and negligible. Server population is always done in the same manner: first 99% of the keys that are to be used in experiments were set by the sequential load option of the memtier. Therefore, for all the get requests we expect similar miss rates which are around 1%. Some actual miss rates, from the baseline with middleware experiments, can be viewed in Section 3.3.

2 Baseline without Middleware

In this section, we examine the characteristics of the clients and servers before the introduction of the middleware to the environment. For all different configurations of different experiments we show that the interactive law holds true by calculating the throughput values from the response times and vice versa. Values predicted through the interactive law are always illustrated alongside the actual values obtained by the memtier outputs.

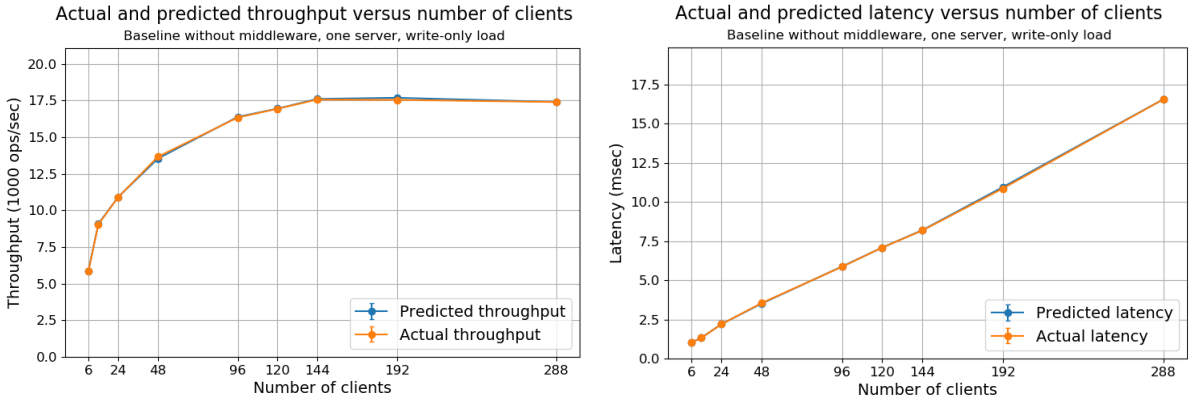
2.1 One Server

First setting contains three memtier clients connecting to one memcached server. Each of the client machines has one memtier instance with two client threads running in them. We vary the number of virtual clients and the type of workload (write-only and read-only) and observe the change in the performance of the system.

2.1.1 Explanation

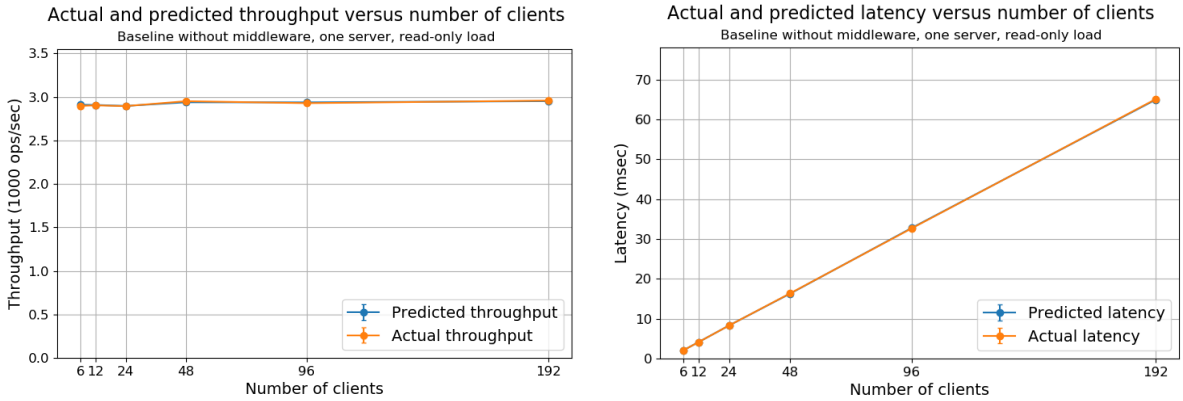
We see the results for the one server configuration in Figures 5 and 6. Plots show that the interactive law holds true for both experiments.

For the write-only part, we can observe the system starts as undersaturated and begins to saturate after 96 clients. Throughput increases steadily until the saturation point, as additional clients are connected to the system. Afterwards we can observe only marginal increases in the throughput until 144 clients, where the system reaches its maximum throughput of 17556.5 requests in average. After 144 clients, the system acts as undersaturated as any additional clients hinder performance, even lowering the throughput values. We see a similar trend in the response time plot as it is slightly curved until around the saturation point but linear



(a) Write-only throughput (b) Write-only latency

Figure 5: Throughput and latency values for the write-only workload for baseline without middleware, for one server case.



(a) Read-only throughput (b) Read-only latency

Figure 6: Throughput and latency values for the read-only workload for baseline without middleware experiment with one server case.

afterwards. After the saturation point, the system cannot handle any more requests and the average response time increases linearly with respect to number of clients while there is no increase in throughput values as the system is already at full capacity. We state 96 clients is the saturation point as it is where the performance yields starts to get more unnoticeable. Going from 96 to 120 or 120 to 144 clients only account for increases around 3% whereas response times continue to increase approximately 20% and 15% respectively.

The reason behind this saturation is meeting the network bandwidth capacity. dstat files show that each client can send at most 24MB per second on average. Looking at the iperf results (Table 2) we can see that client to server communication is limited by 24MB per second for different configurations. As we send 4096B data that means we can send $\sim$ 6000 requests per machine, $\sim$ 18000 requests in total, and this corroborates our results.

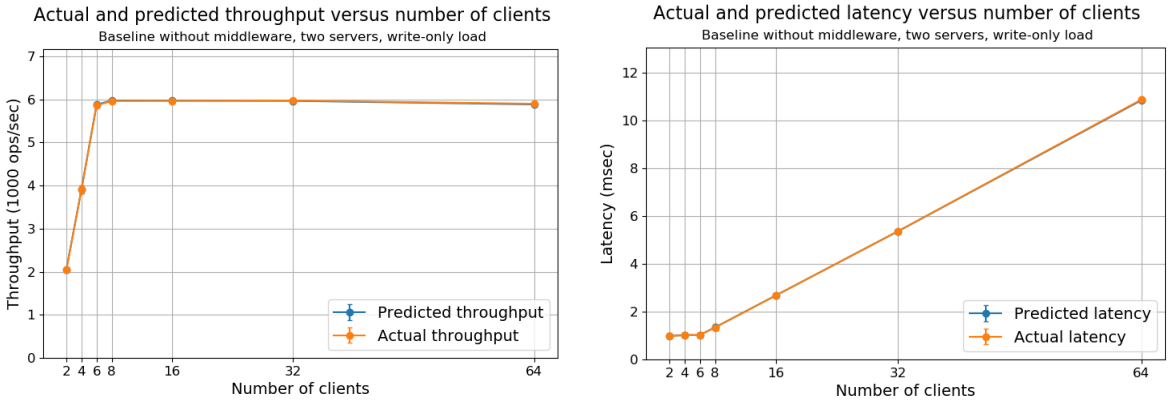
In the read-only part, we see that the system begins operating already in the saturated region for the minimum number of clients we can achieve, which is 6. We conclude this as we cannot observe any noticeable change in throughput when we vary the number of clients and the response time is approximately a linear function of the number of clients everywhere. Although we can see the absolute maximum of the throughput plot is at 32 clients, it is clear that the

system becomes highly inefficient with the addition of each additional client to the system after 6 clients. Only increases in the throughput are quite marginal whereas response times double when we double the number of clients in the system.

Again, dstat and iperf together confirm our assertions and indicates the reason for the saturation. iperf states the server to client connection is limited by 12MB per second (Table 2) and it is what we observe in our server for varying number of clients through all repetitions. This implies the server can reply to at most $\sim$ 3000 requests per second and it is in line with our results.

2.2 Two Servers

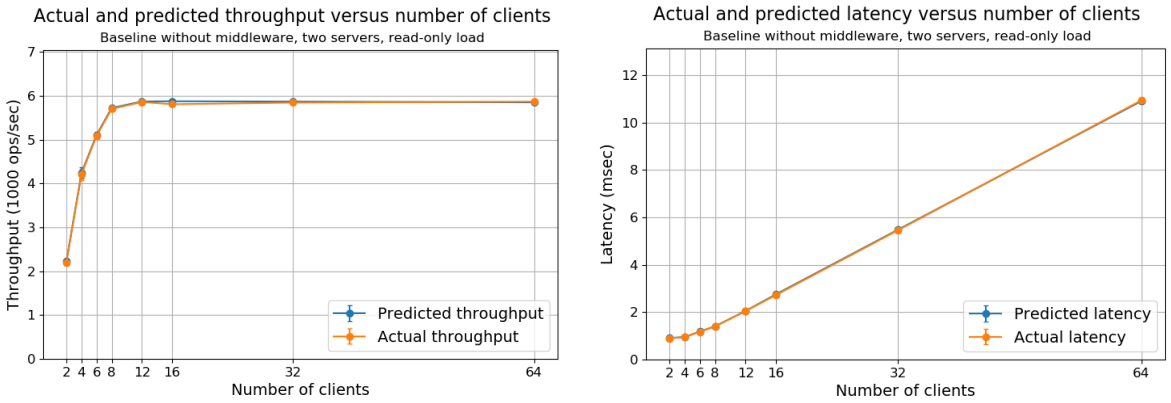
This configuration deals with one client machine sending requests to two servers. This time the client machine has two instances of memtier running (each connected to a different server) with one client thread per instance. We again change the workload (write-only and read-only) and change the total number of clients by varying the virtual number of clients and observe the changes in the performance.



(a) Write-only throughput

(b) Write-only latency

Figure 7: Throughput and latency values for the write-only workload for baseline without middleware experiment with two servers.



(a) Read-only throughput

(b) Read-only latency

Figure 8: Throughput and latency values for the read-only workload for baseline without middleware experiment with two servers.

2.2.1 Explanation

Results are illustrated in Figures 7 and 8. Unlike the first part, we obtain similar trends in plots for both the read-only and the write-only case. For the write-only case, we see the system beings as undersaturated, saturates around 6 clients and reaches its maximum throughput at 8 clients where the throughput plot seems to level off. Figure 7b corroborates our comments as the response times stay stable up to 6 clients, start to increase until 8 clients and become completely linear after it crosses 8 clients.

Similarly, for the read-only case, we conclude the system begins as undersaturated and saturates around 8 clients. Then the system reaches its maximum throughput at 12 clients and again acts as saturated afterwards. We argue that 8 clients is the saturation point as moving from 6 to 8 clients result in comparable changes in both throughput and response times (12% and 19%), moving from 8 to 12 clients bring only marginal increases with respect to throughput (2.7%) but considerable increases in average response times (46%). We can also see that the response times are quite low and almost stable until 8 clients. Similarly to what we have seen before response time plot becomes linear as we move past the saturation point.

The main reason behind the saturation can be found in the network bandwidth limitations for both cases. Although intuitively, the write-only and read-only cases have their bottlenecks in different parts of the network pipeline. Situation is quite similar to the earlier results in Section 2.1. For the write-only case, dstat outputs for the saturated configurations show that the client sends at most 24MB per second to servers, which is its limit confirmed by iperf statistics (Table 2). This corresponds to a total of ~ 6000 requests distributed to two servers, corroborating our results. We also know through iperf that a server can send at most 12MB per second (Table 2) and and we see the same numbers in our dstat files for the read-only part. This equals to 24MB per second received by clients, again amounting to a total of ~ 6000 requests, this time as a result of another limiting factor.

2.3 Summary

Here, we comparatively analyse the different results for the previously discussed set of experiments. The results are explained in more detail in Sections 2.1.1 and 2.2.1, so we will not be going into the specifics of each experiment here.

Maximum throughput of different VMs.

	Read-only workload	Write-only workload	Configuration gives max. throughput
One memcached server	2897.9 ± 16.4	17556.5 ± 50.1	read-only: 6, write-only: 144 clients
One load generating VM	5856.1 ± 12.6	5960.9 ± 3.2	read-only: 12, write-only: 8 clients

Table 3: Summary of baseline without middleware experiments.

Table 3 reports the maximum throughput obtained in both experiments and the configurations that allow such maxima. For both experiments under either read-only or write-only workload conditions, we have concluded that the bottleneck of the system is always the network bandwidth. Intuitively enough, write-only requests have large queries and small responses as they send the data to be stored and only return with a confirmation or an error message. The situation is reversed for the read-only workload as get requests merely contain the keys but their replies carry the data itself (as we get a hit rather than a miss almost everywhere). Therefore, it is in line with our expectations that the throughput values for the experiments with write-only workload are limited by the client to server bandwidth and similarly, throughput values for the read-only counterparts are limited by the server to client bandwidth.

These arguments are clearly evidenced by our results in Table 3. Maximum read-only throughput doubles as we introduce another server to our one server setting. Similarly for the write-only workload, by going from three load generating machines to a one client setting we can see expect the maximum throughput to become one third of what it was before.

As a side note, we observe only the write-only workload with three client machines resulting a appreciable percentage of CPU usage in servers, in the saturated region. Aside from this specific case not one server or client experience any high CPU utilization, idle times are consistently above 90%.

It is also worth noting that for the read-only workload in one server setting (Section 2.1.1) we state the system is already saturated with the presence of 6 clients, even though we do not have any information regarding the fewer number of clients. It is the outcome of a comparison with the read-only experiments with two servers and one client, as we notice that with 6 clients on client machine can help achieve a throughput around 2000 requests. Keeping in mind that the server performance is not a bottleneck anywhere in these experiments we can expect that three clients in one server hypothetically achieving a throughput of 6000 requests whereas the actual value is only the half of it. Then we conclude that system theoretically saturates even earlier, whose effective number of clients we cannot observe with our experimental design.

3 Baseline with Middleware

In this section, we observe the changes in the overall system when the middleware is introduced to the environment, as well as we discuss the characteristics of the middleware that we can observe in the scope of this set of experiments. Interactive law for every experiment is verified using the measurements on clients. More discussion on interactive law can be found at the end of Section 3.3.

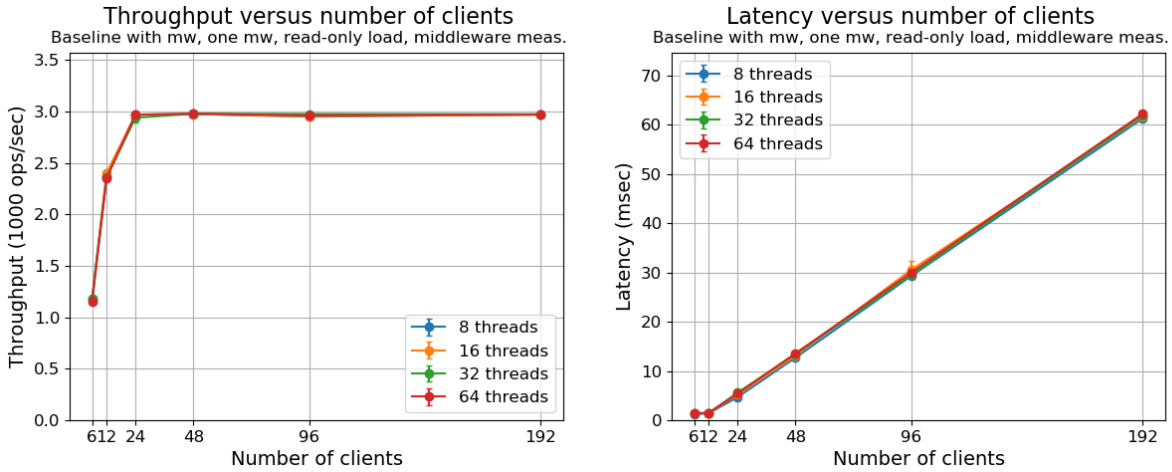
3.1 One Middleware

First experiment for this section is conducted on three client machines connected to one middleware, which in turn connected to one server. Apart from experimenting with two workloads (read-only and write-only) and varying the number of virtual clients we also change the number of worker threads in the middleware.

3.1.1 Explanation

Plots in Figures 9 and 11 illustrate the throughput and response time measured on the middleware, respectively for read-only and write-only loads. With the read-only load, we observe a trend quite similar to the read-only results in baselines without middleware. Until we reach around 24 clients we can observe quite an increase in the throughput which is only accompanied by relatively smaller increases in the response time, regardless of the size of the thread pool in middleware. Afterwards, the trend is again the same for different number of threads: an increase in the number of clients results in a linear increase in the response times and does not change the average throughput obtained. Therefore, we state that the system saturates and reaches its maximum throughput when there are around 24 clients in the system, regardless of the size of the worker thread pool in the middleware.

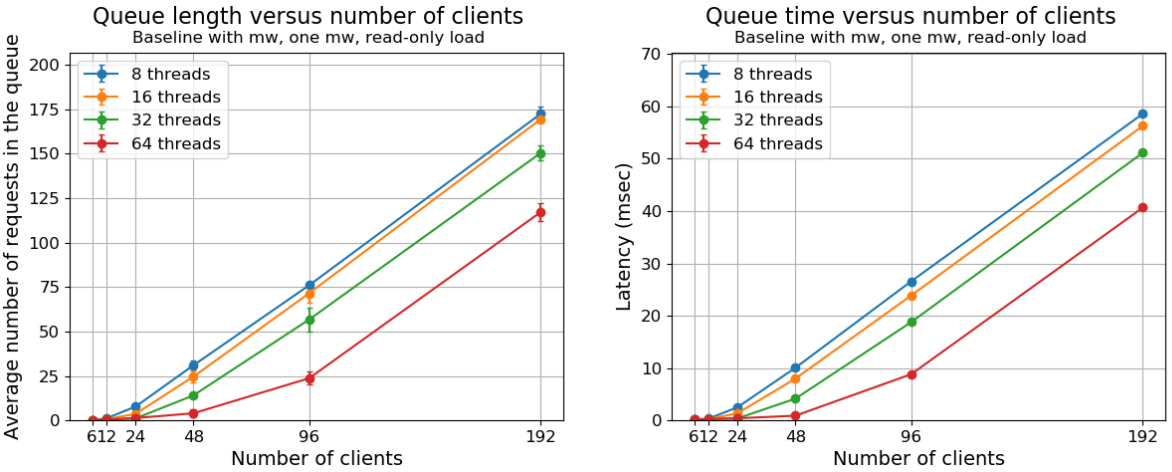
The reason behind this saturation is familiar to us from the previous sections: server to middleware connection is limited by a bandwidth of 12MB per second (Table 2), so the server cannot send more replies because of the network limitations. This saturation happens so early in



(a) Read-only throughput

(b) Read-only latency

Figure 9: Throughput and latency values for the read-only workload for baseline with middleware experiment with one middleware.



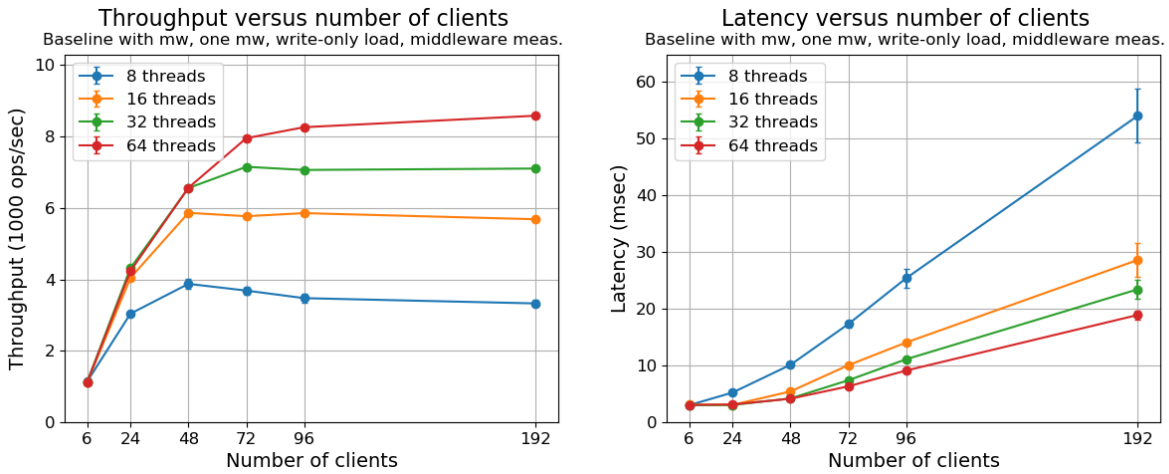
(a) Avg. queue length

(b) Avg. queue time

Figure 10: Average queue lengths and queue times for the read-only workload for baseline with middleware experiment with one middleware.

the system that we do not get to observe the effects of different threads in middleware. In other words, the middleware works efficiently even with 8 worker threads until the system saturates, so any number of worker threads could hypothetically handle more work if the server could be able respond more clients. It should also be noted that we cannot observe any network-wise limitations on top of the server-side send bottleneck and the CPU usage is fairly low for all the machines in the system (80% idle time at worst), as per dstat and iperf.

Figure 10a illustrates the average queue lengths for various configurations. We can observe the queue lengths (and correspondingly queueing times) are being insignificant for all configurations until around 24 clients where we observe the saturation. As the saturated system is network bound for this workload, the server can only reply a portion of the requests while the other worker threads wait to send their requests. As seen in Figure 10b, a greater number of threads mean that requests spend less amount of time in the queue. However, from Figure 11b,



(a) Write-only throughput

(b) Write-only latency

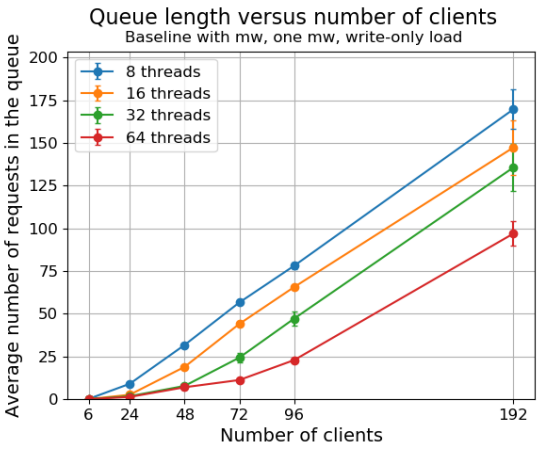
Figure 11: Throughput and latency values for the write-only workload for baseline with middleware experiment with one middleware.

we know that each configuration has the same average response time, meaning even a request exits the queue early it should still wait for the server to be available. Even so, greater number of threads mean a higher contention for the server usage. As the number of worker threads change the times spent in queue and waiting for the server also changes while the bottleneck stays the same. Lastly, we can see by comparing the values in Figures 10a and 10b that the queue times and the queue lengths have an approximately linear correlation as expected. The reason behind such relation is that as the system is network bound it is not the middleware but the server side that affects the queue times in the saturated region so changing the number of worker threads does not make a difference.

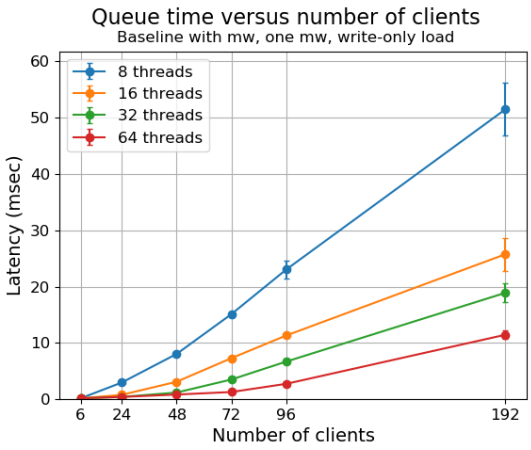
In the write-only case, we can observe a trend which is completely different from the read-only case, as the network is no longer our bottleneck. The trend until reaching 24 clients is similar with the read-only case as differences in the number of worker threads do not correspond to a difference in terms of the performance of the system, except 8 thread configuration which saturates around 24 clients. This is because the system is under-saturated for 16, 32 and 64 threads so we can expect their throughput and response time statistics to go hand in hand as the number of worker threads do not result in a noticeable difference in these metrics under light workload conditions. As we introduce more and more clients to the system we observe more divergence with respect to performance for different number of threads in the middleware.

We can see that there are saturation points with different values in both axes for different worker thread configurations. For the parameters explored, we say the 8, 16, 32 and 64 thread configurations have their saturation points respectively around 24, 48, 48 and 72 clients in total. Due to the limitations of the experiment, the number of clients were not explored in depth for us to pinpoint exact saturation points for all worker thread configurations. We believe a comparably more fine-grained study might show slightly different saturation points.

This time, the comparison of queue lengths and queue times (Figures 12a and 12b, respectively) yield a different interpretation. We can clearly see that, for the same queue lengths, different number of worker threads result in different slopes for the queue times, i.e., queue length to queue time ratios increase with the number of worker threads. What that means is that as the size of the thread pool increases, the system could process a longer queue for the



(a) Avg. queue length



(b) Avg. queue time

Figure 12: Average queue lengths, queue times and waiting times for the write-only workload for baseline with middleware experiment with one middleware.

same queueing times in average, since there will be more worker threads available to handle the requests.

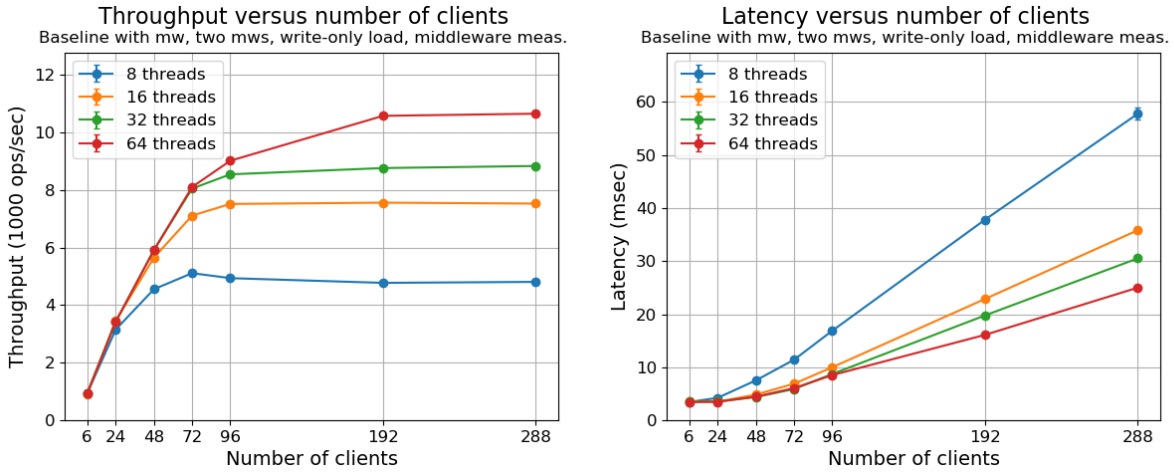
After interpreting all the relevant plots we can conclude that for the write-only workload the middleware is the bottleneck of the system. dstat measurements are also cross-checked and we do not report any exceptional usage in network or CPU for the write-only configurations. We say that the number of worker threads is the limiting parameter for our experiments for the write-only load and more worker threads mean better system performances. Moreover, if we were to compare the results obtained here to the ones we have discussed in Section 2.1, we could see that we cannot achieve the levels of throughput reached without middleware, as the middleware acts as a bottleneck between the clients and the server for the parameters we explored. Although the middleware introduces overhead in the system especially with bookkeeping and additional network delays, we believe the difference between middleware and client performances would diminish more with additional worker threads in the middleware.

3.2 Two Middlewares

Second experiment for this section again has an environment with three clients and one server, but two middlewares instead of one. Again, we vary the workloads, number of clients and number of worker threads in the middleware and discuss the results.

3.2.1 Explanation

Plots in Figures 13 illustrate throughput and response time statistics for the write-only load. We see results similar to the one middleware setup. System saturates at different number of clients for different number of worker threads. For the same average queue lengths, the queue times shorten as the number of worker threads increases (not shown in the report due to space limitations). Again the number of working threads is our limiting factor for the system performance. Thus, we conclude the middleware being the bottleneck for the write-only case with two middlewares. Again, we find it intuitive that the statistics for the one middleware setup matches those of the two middleware setup with the half amount of worker threads, as it was the case with the read-only workload.



(a) Write-only throughput

(b) Write-only latency

Figure 13: Throughput and latency values for the write-only workload for baseline with middleware experiment with two middlewares.

An intuitive correlation is as follows: statistics for the 8, 16 and 32 thread configurations for two middleware setup matches respectively with the 16, 32 and 64 thread settings for the one middleware. In other words, doubling the number of threads for one midleware has the same effects as introducing another middleware with the same number of threads on the system. Although, we can only say that for our experimental setup and configurations and this equivalence may not hold for factors much greater than two.

Results for the read-only workload is almost exactly the same as those from the setting with one middleware, as the bottleneck of the system is stays the same, which was detailed in the previous sections. Both the throughput and latency values (Figure 14) with respect to number of clients is quite similar to those of the configuration with one server, stemming from the same limitations.

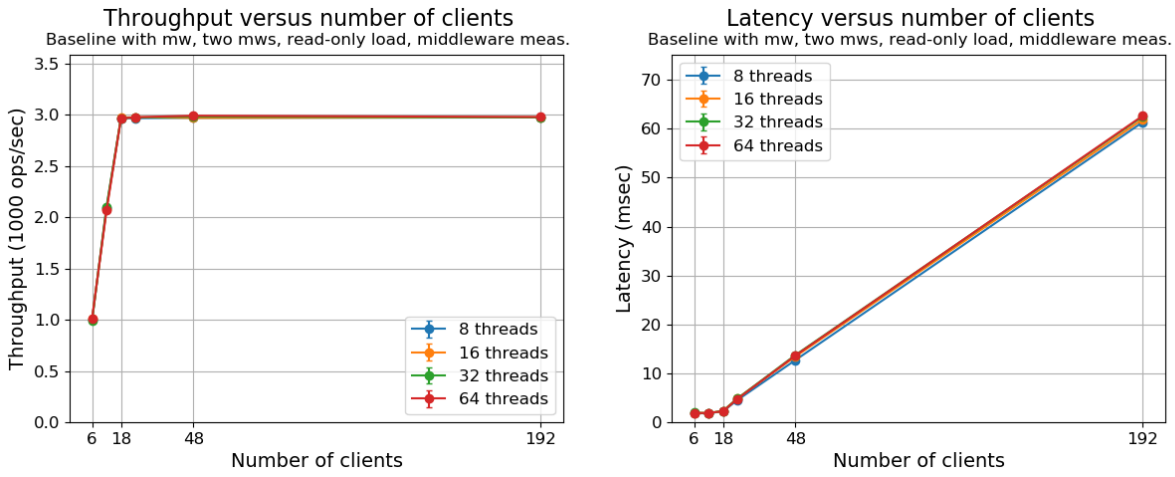
Figure 15 illustrates the average queue length and queue time statistics for this experiment on read-only load. Here, we warn the reader to be careful when comparing the queue lengths for two and one middleware setups directly as the reported queue length is averaged over the middlewares. That is, to approximate the total number of requests waiting in the middleware queues one should multiply the averaged value by the number of middlewares in the environment. Lastly, we note the relation between queue lengths and queue times is the same as in the one middleware setup.

3.3 Summary

Tables 4 and 5 reports the results for the middleware experiments, respectively the one middleware and the two middleware setups. All of the results shown in the tables are from the 64 worker thread configuration of the respective experiments.

First thing we can do is to compare the values aggregated by the clients with those obtained by the middlewares. Throughput and miss rate statistics show that they are approximately, but not absolutely, equal. As we aggregate data after cutting the warm-up and cool-down periods on both client and middleware outputs, this is a natural result. It is verified through additional experiments that the statistics match completely if the aggregations were to be done on the entirety of the data collected by both sides.

For response times, there is a significant but consistent difference between the averaged



(a) Read-only throughput

(b) Read-only latency

Figure 14: Throughput and latency values for the read-only workload for baseline with middleware experiment with two middlewares.

Maximum throughput for one middleware.

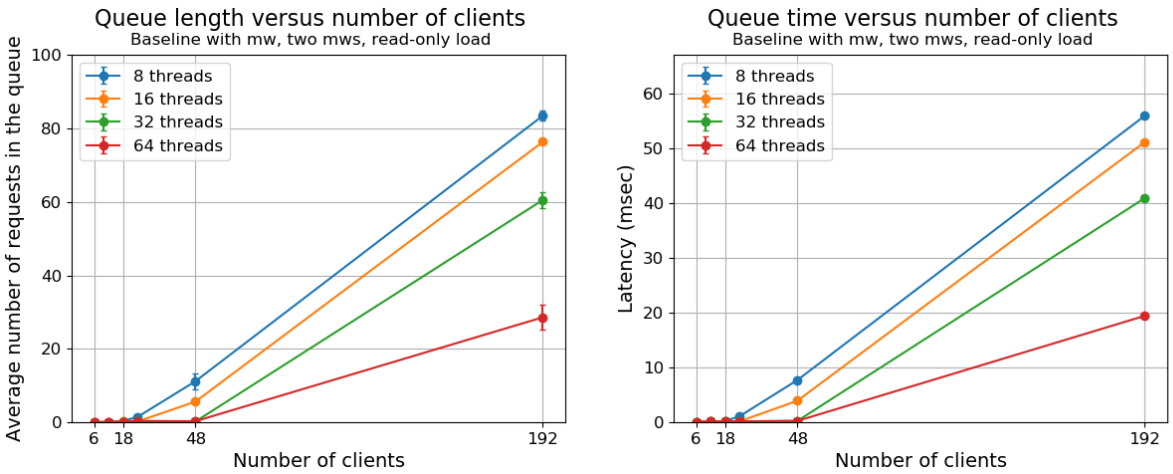
	Throughput	Response time	Avg. queue time	Miss rate
Reads: Meas. on mw	2969.3 ± 1.5	5.413 ± 0.021	0.406 ± 0.011	0.0095404
Reads: Meas. on clients	2969.3 ± 1.8	8.083 ± 0.004	n/a	0.0095287
Writes: Meas. on mw	8264.1 ± 17.5	9.125 ± 0.017	2.740 ± 0.015	n/a
Writes: Meas. on clients	8263.5 ± 17.3	11.628 ± 0.026	n/a	n/a

Table 4: Summary of the baseline with one middleware.

values reported by the middlewares and the clients. Client response times also include the time a request requires to travel from a client to a middleware and its reply from the middleware to the client, similarly. The difference between the aforementioned response times are equal to the round-trip time (RTT) of the clients and serves for the respective experiments. We can also observe that this RTT is not always constant. It is again natural as the network characteristics may fluctuate in between (and even during) experiments and we also employ multiple clients on different physical machines, we expect not a constant difference but a RTT that samples a probability distribution.

As discussed in Section 1.2, we populate the server with the 99% of the keys we will use in the read-only experiments. This, in turn, (together with the fact that we do not allow any expiry) means that for each read-only experiment we should observe a miss rate around 1% when looked from both middleware and client perspectives. Looking at the reported miss rates we can see that the values observed is in line with our expectations and client results corroborate the middleware aggregations. The marginal difference between values reported by clients and middlewares are due to times cut, as it is the case with throughput values.

As explained in the preceding sections, maximum read-only throughput is just under 3000 requests per second, for both experiments. As the system is network bound under read-only load, changes in the number of middleware and the number of worker threads per middleware change neither where the system saturates, nor the maximum throughput achieved. Write-only experiments present us with a different picture. As the bottleneck of the system is middleware for both configurations, we see an increase in the maximum throughput obtained when we add



(a) Avg. queue length

(b) Avg. queue time

Figure 15: Average queue lengths and queue times for the read-only workload for baseline with middleware experiment with two middlewares.

Maximum throughput for two middlewares.

	Throughput	Response time	Avg. queue time	Miss rate
Reads: Meas. on mw	2964.3 ± 0.3	2.288 ± 0.061	0.181 ± 0.001	0.0093647
Reads: Meas. on clients	2964.4 ± 0.8	6.077 ± 0.002	n/a	0.0095260
Writes: Meas. on mw	10566.1 ± 59.6	16.121 ± 0.118	4.557 ± 0.039	n/a
Writes: Meas. on clients	10566.4 ± 59.6	18.206 ± 0.105	n/a	n/a

Table 5: Summary of the baseline with two middlewares.

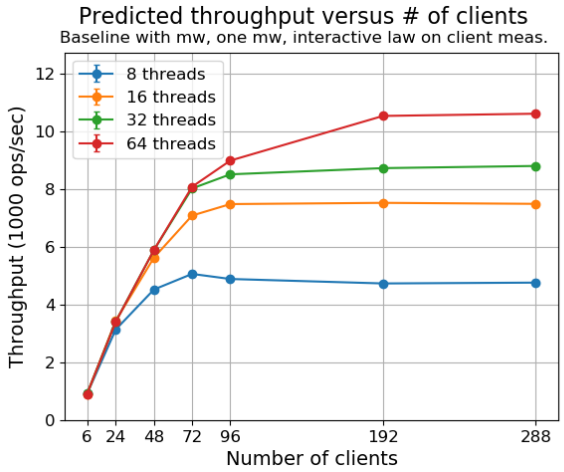
a new middleware to the environment. This holds true not only for the 64 thread configuration which gives the best performance but for all sizes of the worker-thread pool. Looking once again at the Figures 11 and 13 we can see the saturation points for all worker-thread configurations go up with respect to clients. To illustrate, one middleware with 16 worker threads is already saturated with 48 clients but with two middlewares of 16 threads we see a continuous increase lasting between 72 and 96 clients. As discussed before, doubling the the number of middlewares has approximately the same effects on the system performance as doubling the size of the worker-thread pools in middlewares.

The reader may notice the performance of the system measured in average throughput does not double when we double the number of middleware threads, either by doubling directly in one machine or adding another middleware. Our system is neither an ideal nor a simple system, so we do not expect such strong correlations and there are a couple of reasons that hinder the performance gain with a high number of worker threads. We always configure our servers to work with one thread, so an increase in the number of worker threads translate into an increase in the number of server connections, which might not scale very well in the presence with one thread. Additional handicaps could be due to the limitations of the middleware machines as the higher number of threads will result in more overhead and context switches, which diverges the system from the ideal performance.

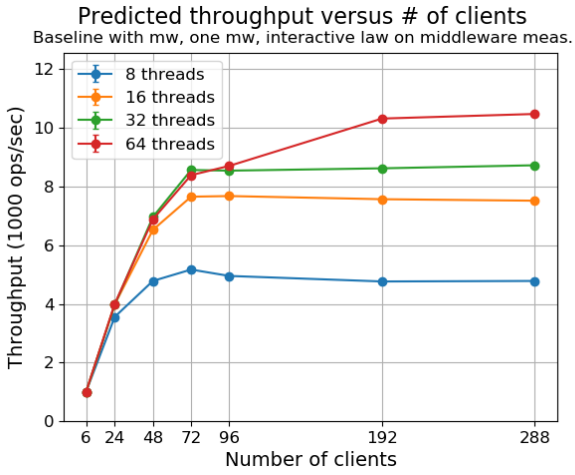
Similarly to what we have discussed earlier in the Section 3.1.1, again in the two middleware setting we can observe the server waiting times increasing when there are more worker threads present in the system. The trend is similar with both workloads and in the presence of one and

two middlewares. One such trend can also be observed in the next section, where we have almost an identical experimental setup. Figure 19a illustrates one such example for the throughput for writes experiments. It serves as another method of verification of system saturation as a saturated system that operates with maximum utilization is expected to have stable waiting times as increases in the number of clients should only affect the queueing times.

Lastly, we note that even with two middlewares of 64 threads we cannot reach the maximum throughput achieved without middleware in the environment. We say the parameter values explored through experiments of this section is not enough to obtain performances comparable to the ones in Section 2.1. For the same number of clients, additional network delays incurred and the overhead of the middlewares result in lower performances with the systems with middlewares.



(a) Predictions on client measurements



(b) Predictions on adapted middleware measurements

Figure 16: Application of the interactive law illustrated for predictions made using client and midleware measurements. Each response time observed in middleware is increased by a constant amount to simulate the effects of RTT between clients and the middlewares. Results not being perfect indicate the RTT values are changing during experiments. These results show that RTT is not a constant value as in the ideal case, but it is a random variable.

Figure 16a provides an example illustration of the interactive law, reader should compare this plot with the actual values in Figure 14a. We only provide one exemplary plot here and the reader should feel free to use provided scripts and auxiliary plots to see the interactive law in action on other configurations that include middleware.

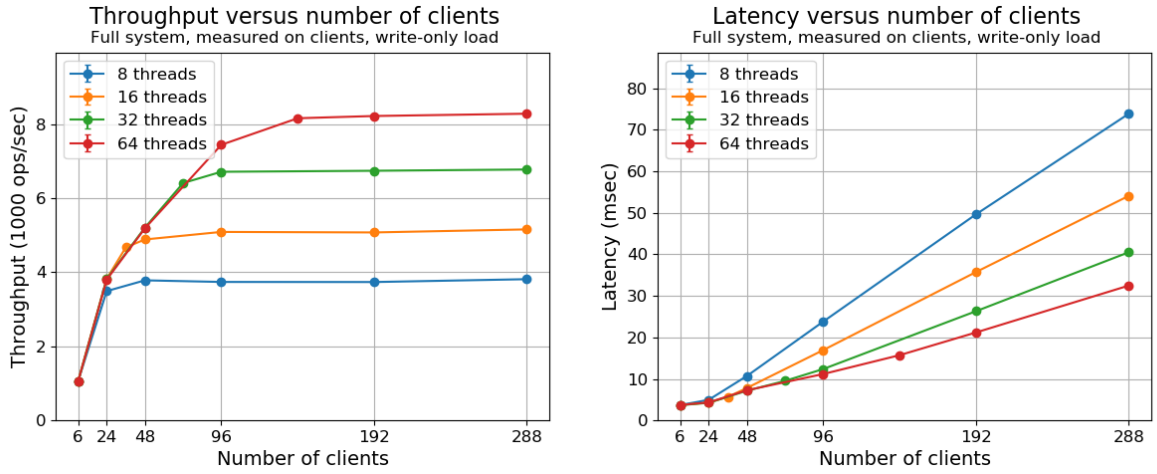
If we were to use the middleware measurements to predict the corresponding values, we would need to adapt the observed values. Our approach is to take the RTT between client and middleware machines as the think time of the clients when viewed from the perspective of the middleware. If we add the observed RTT times to the middleware response times for each experiment we would simply obtained the exact same results as ones with the client measurements. Instead, we also try modifying the middleware response times with an approximate RTT value for all experiments and the results can be still satisfactory. Figure 16b illustrates such a case (actual values again in Figure 14a). Reader can see some discrepancies between actual and the predicted values, which indicates the end-to-end delays not being constant for the duration of one experimental configuration, even using the same physical setup. Quite intuitively, we cannot perfectly estimate the actual throughput and response time values from the middleware

observations without knowing the network conditions for that specific experiment.

4 Throughput for Writes

This section inspects the system with three clients, two middlewares and three servers, under write-only load. Number of worker-threads and the number of clients are varied in order to understand the characteristic of the system through different configurations.

4.1 Full System



(a) Throughput

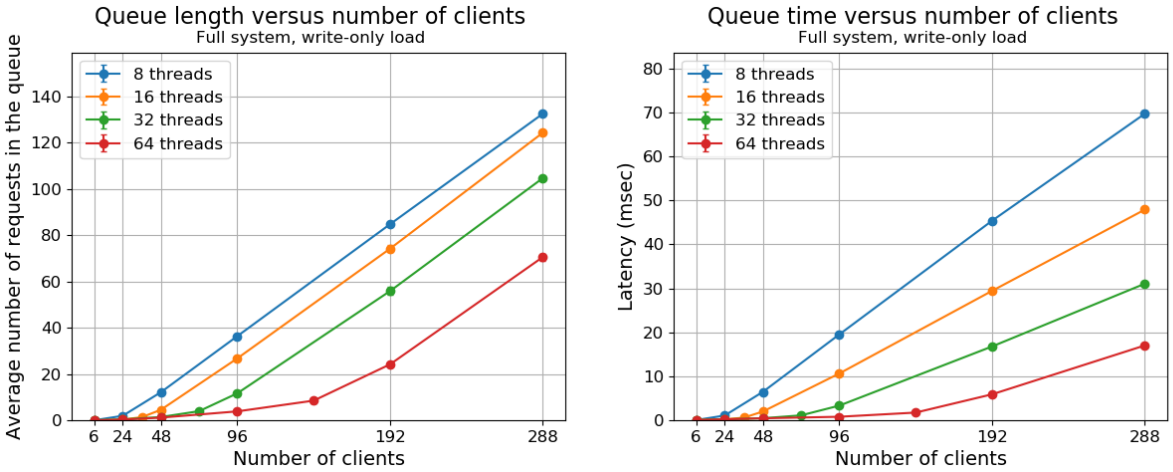
(b) Latency

Figure 17: Throughput and latency values for the throughput for writes experiment, write-only workload.

Figures 17a and 17b illustrate the average throughputs and response times of the system under different configurations. Each line represents a setup with a different number of threads for both plots. Results are similar to those analysed in Section 3.2, as the setup stays the same except the increase in the number of servers the system has. We see each configuration saturates at different points, since the middleware is the bottleneck of the system. Each of them conforms to the same trend: throughput gains become minimal and response times start to form a linear pattern as we move closer to the number of clients which saturates a given configuration.

4.1.1 Explanation

For 8, 16, 32 and 64 worker-threads we state the saturation points are around 24, 36, 72 and 144 clients. With the exception of the 8 thread setup, we can see that the number of clients a saturated system can handle doubles as we double the worker-threads in the middleware. It is in line with our expectations as this ratio of number of worker-threads to the number of clients at the point of saturation should hold true for different number of worker threads with same configuration. Since middlewares are the bottlenecks of this setup, we except there should be enough requests in the systems waiting to be handled. This, in turn, means an increase in the number of worker-threads should amount to an approximately linear increase in the average number of requests that can be handled for a given period, which is indeed corroborated by the aforementioned results. Reader should note that the 8 worker thread setting has not the same level of granularity as the other settings and experiences a much sharper increase before



(a) Avg. queue length

(b) Avg. queue time

Figure 18: Average queue lengths and queue times for the throughput for writes experiment, write-only workload.

leveling off. This points to the need for a finer grained search in the 6-24 client range, as the saturation of the 8 thread configuration is believed to be expected earlier than 24 clients, much likely around 18 clients considering the pattern described above.

Figures 18a and 18b respectively illustrate the average queue length and queueing time statistics for the throughput for writes experiment. As the experimental setup of this section is quite similar to the one in baseline experiments with two middlewares in Section 3.2, it is of no surprise that we obtain similar trends related to request queue. We can verify the saturation points for the different thread configurations as the points where lines in both plots become completely linear. We can again observe that corresponding lines in queue length and queue time plots have different slopes. For the system to reach a chosen queue length, we would need more clients with configurations with more threads. What that indicates is an increase in the number of threads in the middleware increases performance by reducing the queueing times, so the number of middleware threads is our limiting factor for this experiment.

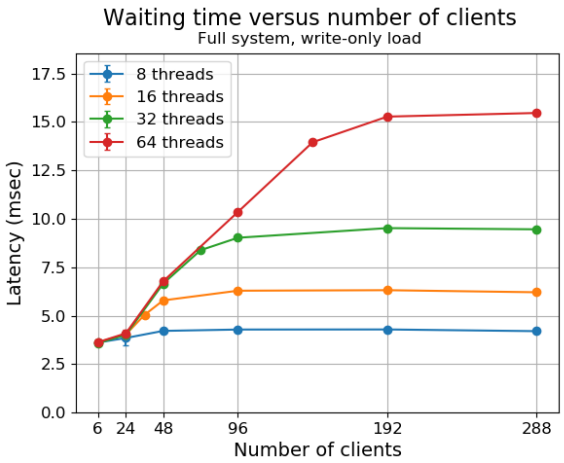
4.2 Summary

Maximum throughput for the full system

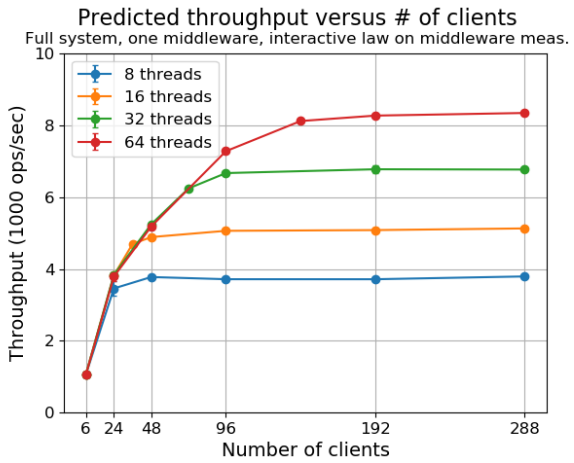
	WT=8	WT=16	WT=32	WT=64
Throughput (Middleware)	3492.9 ± 49.1	4673.6±23.5	6411.0 ± 5.0	8161.1±24.9
Throughput (Derived from MW response time)	3459.6±205.4	4698.3±40.1	6246.1 ± 5.8	8124.1±27.3
Throughput (Client)	3492.8 ± 49.1	4674.3±23.4	6412.2 ± 6.2	8161.5±25.2
Average time in queue	1.112 ± 0.062	0.622±0.006	1.148±0.010	1.779±0.017
Average length of queue	2.0 ± 0.0	1.4 ± 0.1	4.0 ± 0.2	8.6 ± 1.2
Average time waiting for memcached	3.9 ± 0.4	5.0 ± 0.1	8.4 ± 0.0	13.9 ± 0.0

Table 6: Summary of the throughput for writes experiments.

Table 6 presents a summary of the experiments of this section. Through the help of this table and the plots in Figure 20 and 19, we will quickly go over the salient points of this experiment, many of which we have discussed before.



(a) Server waiting times



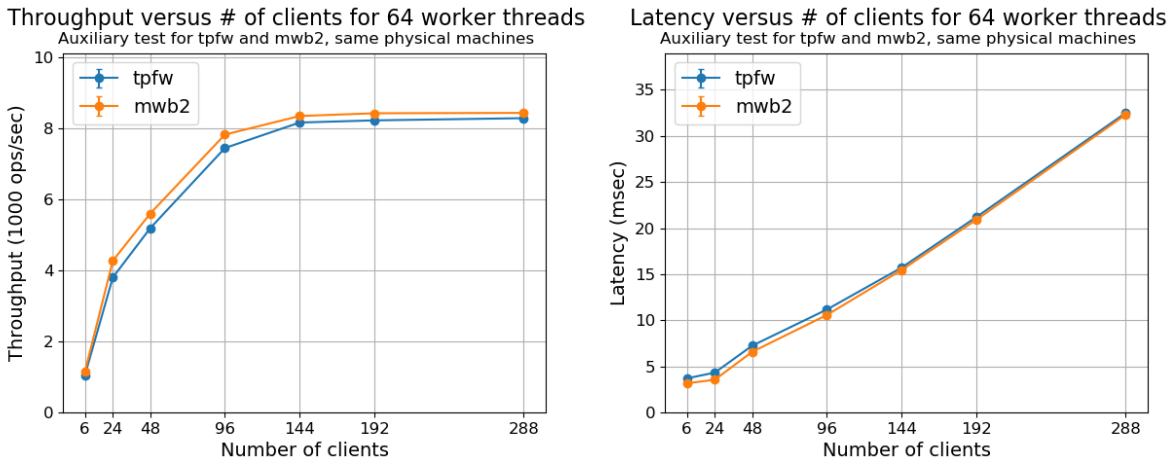
(b) Interactive law on middleware measurements

Figure 19: Server waiting times and the application of the interactive law on the altered middleware measurements.

When we look into the reported maximum throughput values, we first verify the values reported by the clients and middlewares corroborate each other. The occasional differences are smaller than 0.1% of the reported values and due to times cut for warm-up and cool-down periods, as discussed before in Section 3.3. The difference between the actual values and the ones derived using the interactive law is more apparent. Again, as discussed before in the aforementioned section, we first alter the middleware measurements to adapt them to the actual experimental setup. For that reason we take the average RTT between the clients and the middlewares to model the think time of the clients from the middleware perspective. Figure 19b shows the overall results of this derivation, user should compare this plot to 17a. It is of no surprise that the predicted values do not match our observations as we approximate the RTT between clients and middlewares as a constant, yet the end-to-end delay between different client and middleware machine pairs are different and they may even change as time passes.

Last thing regarding maximum throughput values we can mention is that improvement in the performance with the larger thread pools as the middlewares can serve more clients. However, one should note that the doubling the resources (e.g. number of middleware threads) does not always result in system performance (e.g. with respect to throughput) doubling. We argued about some possible underlying reasons and it is sensible that the change in the setup would not validate such claims. In fact, additional number of servers arises as another reason of why we cannot view the system as a black box and expect the changes in the system performance to be simple functions of its parameters.

We can see the average queue lengths for the points that give maximum throughput are minimal, resulting in negligible queueing times. It is also easy to see the higher number of threads can process the same queues faster by observing the queue time to queue length ratio decreasing with higher number of threads. Server waiting times presents us with another picture. We can see in Figure 19a the waiting times for different number of clients. Although it may not be as critical as the queueing times for the most cases, waiting times can be problematic for the configurations with many threads. As also discussed in Section 3.3, middlewares do their best to pop items from the queue and send them to the servers at the saturated regions. After saturation an increase in the number of clients will only result in an increase in the queue lengths as waiting times will not change as long as the middleware configuration is kept the same. Waiting times also serve as another tool for verifying our comments about saturation as



(a) Throughput

(b) Latency

Figure 20: Throughput and latency values for an auxiliary experiment, write-only workload with 64 worker threads on middlewares. tpfw stands for the throughput for writes experiment, whereas the mwb2 denotes the middleware baseline experiment with two baselines.

we can see the saturation of the system as discussed earlier from another perspective.

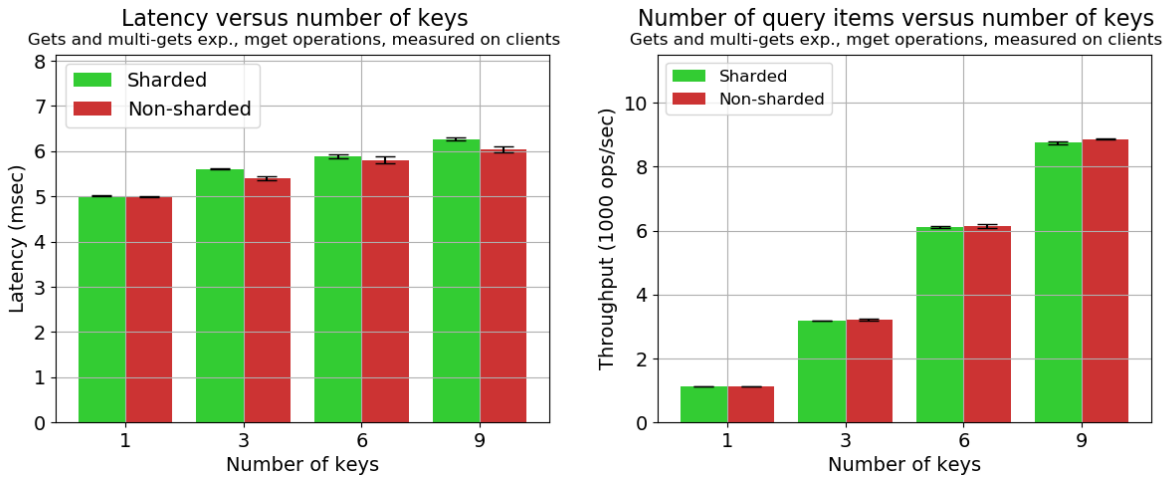
As the experiments of this section and ones in the Section 3.2 were conducted in different physical setups, it might not be satisfactory to compare them directly as we only expect minute differences in results. For this reason, we have conducted an auxiliary experiment which compares the configurations with same number of worker threads for these two experiments. Results can be viewed in Figure 20. The only difference between the configurations in this plot is the number of servers being 1 and 3. We expect the case with more servers to have a slightly lower performance than the other as sending requests to servers and waiting for their replies would incur some delays, even when it is being done in an asynchronous manner.

5 Gets and Multi-gets

This experiment again deals with the full system having 3 clients, 2 middlewares and 3 servers. Each of the two middleware has 8 worker threads and each client instance has one thread with two virtual clients (a total of 12 clients in the system). Clients send multi-get requests together with set requests and set to multi-get ratio for each experiment is 1:X where X is the number of keys in the multi-get requests. We change this key size and see how the system reacts. A comparative analysis is made on effects of the sharded and non-sharded aproaches for multi-get handling. Reader should feel free to take a look on Figure 3, which describes the operational flows for sharded and non-sharded cases.

5.1 Sharded Case

In sharded case we split the multi-get requests send all splits to respective servers and proceed to listen for replies in the order we sent the requests. Illustrated in Figures 21a and 21b we see the response times and number of requested items for experiments with different key sizes. We define the number of requested items as total number of get queries sent through multi-get queries, that is, it is simply throughput times key size.



(a) Response times

(b) Number of requested items

Figure 21: Response time and number of requested item statistics for sharded and non-sharded reads in different key size experiments

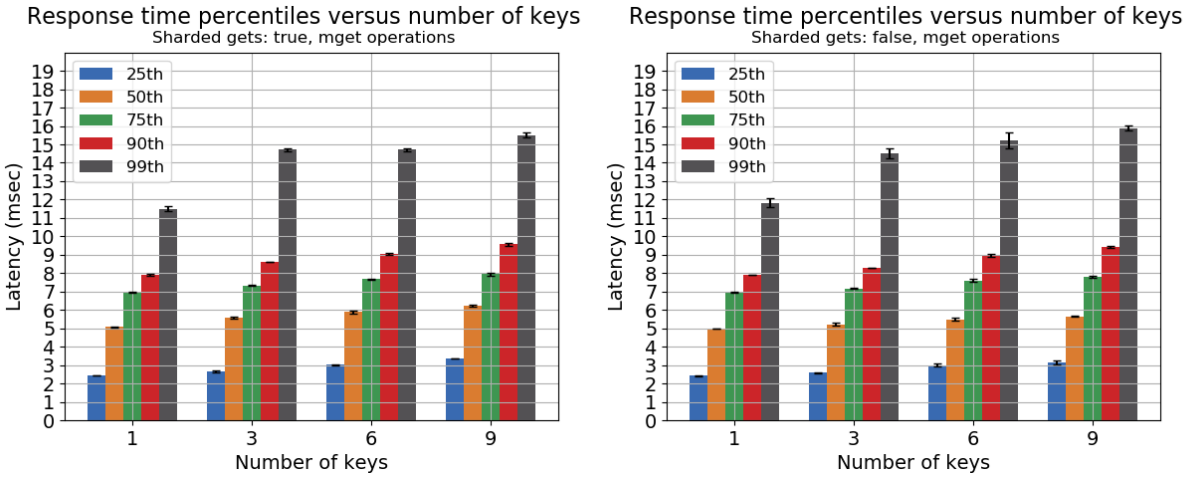
5.1.1 Explanation

If we were to compare the response times observed with the ones in Section 3, we can see that we get comparably larger response times for the same number of clients (12) as we now deal with a greater data flow throughout the system. Response times increase as the key sizes increase since larger requests result in more overhead in both middleware and server machines and accounts for larger end-to-end delays over the network.

We can verify in Figure 21b the number of individual get requests do not get close to 9000 requests, which was shown to be the upper limit due to bandwidth limitations in Sections 2 and 3. We say that we do not utilize the network in full efficiency, maybe except the 9 key case. If we were to continue increasing key cases in a similar fashion after the 9 key case, we believe we would not see any increase with respect to throughput as we would be already sending ~ 9000 requests back and forth. We also see the total number of requests are not perfectly proportional to the key sizes as larger requests take more time in the system, resulting in smaller throughput per multi-get request.

We see for all key sizes that the average queue lengths and queueing times being approximately zero, although those statistics are not illustrated here due to space limitations. Then we can conclude the response time are only the results of the server waiting times as there are always worker threads available for the request handling. This shows the number of middleware threads is not a problem for any configuration as the queueing times are negligible everywhere.

We see the middleware worker threads being idle and network bandwidth not being a problem up to 9 key case, so we conclude the number of incoming requests are the limiting factor for us with the explored configurations. If the clients were sending more requests middleware would be able to relay them to the servers, achieving more throughput without causing long delays. 9 key case presents us with a different picture as it is where the system bottleneck changes: worker threads are still under saturated, but the system would not be able to handle any additional requests of same type as network is already fully utilized. It is intuitive that we would reach such a limit if we increase the request sizes without changing system parameters. Number of requests are the same and the additional overhead a larger multi-get causes is comparably small. Therefore, changing the key sizes create only small differences in middleware utilization but greatly effects the network usage. Figure 22a shows the percentiles of the response time



(a) Sharded case

(b) Non-sharded case

Figure 22: 25th, 50th, 75th, 90th and 99th percentiles of sharded and non-sharded reads, for 1, 3, 6 and 9 key cases. Instead of their respective sections each, plots are provided side by side together to help the reader when comparing two cases.

distribution for the sharded reads. The results our in line with our expectations since we would expect most percentiles to go up as the key sizes grow larger. Assuming the network conditions are more or less the same in average and the overall system operates in a predictable and robust manner, each distinct request would have a bigger size, which in turn incur more network delays and both middleware and server overheads as discussed above. We cannot assert such a trend would be clearly visible in all percentiles as the extremal percentiles (99th for our case) only accumulate the extreme outliers which might not be representative of the general distribution of the response times. Such outliers indicate the extremal changes in the system states (be it internal or external, rather than the mean conditions. However, discrepancies between extremal percentiles would diminish more with more repetitions.

5.2 Non-sharded Case

Non-sharded case is rather straightforward as the worker threads of the middleware send the whole multi-get requests to only one server. The reader should remind the non-sharded case still utilizes a simple load-balancing scheme through round-robin as we have discussed earlier.

5.2.1 Explanation

Figures 21a and 21b show the response time and requested number of items statistics for the non-sharded case along with the sharded one. The trend in performance is quite similar to the one observed in sharded reads, which we have discussed earlier. Larger key sizes mean larger requests, which brings additional overhead in machines and latency over the network. Queue lengths are approximately zero and queueing times are negligible, again practically making the response times only a function of server waiting times (which incorporates both the service time of the servers and the RTT of the middleware-server connections). We again state the bottleneck is number of clients with respect to throughput and network bandwidth for the 9 key case only, with respect to total number of requested items.

We can see the non-sharded case consistently beating the sharded case in terms of throughput for the 1,3,6 and 9 key cases. Non-sharded case sends the whole data at once through one

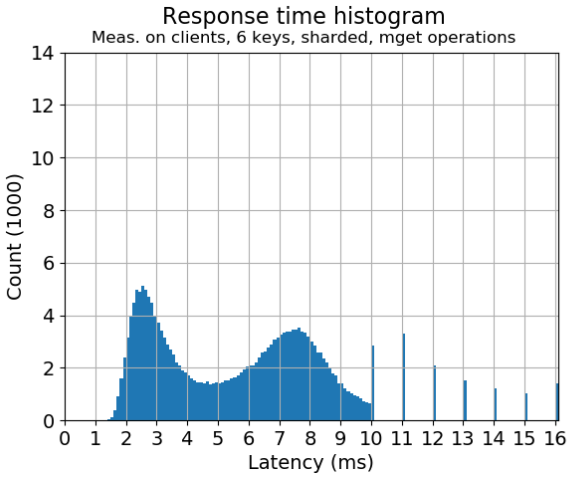
connection whereas the sharded case splits the data into three parts and sends it over three connections. Although the overall data sent through the network is same (excluding metadata of the network packages) utilizing multiple connections incur network overheads, both while sending the requests and receiving the replies. Therefore, it is no surprise that sharded reads have higher response times in average. Figure 22b illustrates the percentiles of the response time distribution for the non-sharded case. Similarly to the sharded case, percentiles go up as key size increases and we can even see this trend in the 99th percentiles of the non-sharded case. Non-sharded case almost consistently have smaller response times in all percentiles than the sharded one. This could be regarded as intuitive, since sharded reads cost additional latency per each request. We will not go into the details of the percentile plots here as it is hard to compare the trends for sharded and non-sharded percentiles in this range and we will discuss them alongside other topics in Section 5.4, where the results from additional experiments are explored together with the earlier ones.

5.3 Histogram

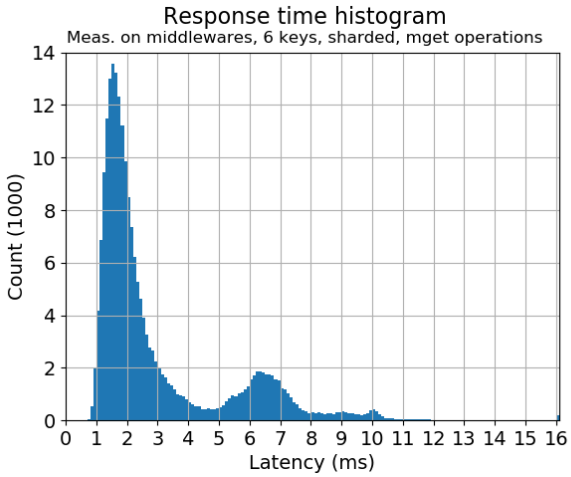
Plots in Figure 23 show the response time histograms for the 6 keys case. These four plots illustrate the distribution of the response times for client and middleware measurements and under sharded and non-sharded read conditions. Histograms were constructed by aggregating all the response time in consecutive repeats, instead of averaging over them. This method of presentation was chosen in order to remove the clutter of error bars over histograms. Each histogram is inspected before to verify if the errors over repeats were negligible on each histogram constructed.

The most apparent characteristic of these pairs of plots is arguably the difference between the client and middleware histograms. Addition of the client-side latencies change the shape of the histograms by shifting the mass of response times to higher values. We attribute this difference mainly to the end-to-end delays between client and the middleware machines, although connection overheads in the middleware net-thread may also have some contribution to this additional latency. The reader should note that the response histograms do not just shift, but instead change their distribution. We may want to model the network as a delay center, but our observations indicate that the RTT between clients and middlewares are not constant as in the ideal case, but instead acts also as another queueing system. This is quite natural and we have also discussed this phenomenon in the earlier sections, especially when interpreting the predictions of the interactive law on altered middleware measurements (Figure 16b in Section 3.3 and Figure 19b in Section 4.2). We suspect the reason we get a bimodal distribution in the middleware histograms in the first place is tied to the fact the requests similarly experience network delays in middleware-server communications. Although we do not have the necessary data to argue further as our measurements do not discern the times spent in middlewares, servers or in-between.

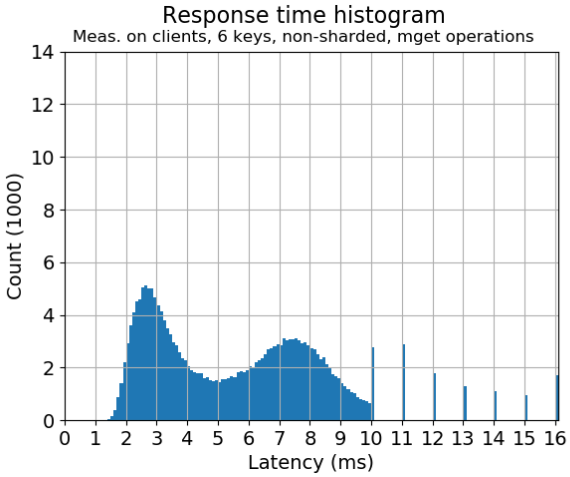
Now, we compare the pairs of histograms for sharded and non-sharded read conditions. We can see in both pairs that the response time histograms for the sharded case are more sharply peaked than the non-sharded ones, and the non-sharded ones having fatter tails. The difference is especially evident in middleware histograms. As we discussed above, network conditions are not always the same and this result in end-to-end delays between machines fluctuating over time. By aggregating our observations we may diminish the effects of such fluctuations in our statistics but individually there could be significant differences between different requests. We also hinted at this issue while talking about the percentiles of the response time distribution (Figures 22a and 22b), stating such fluctuations were more visible in 99th percentiles than the



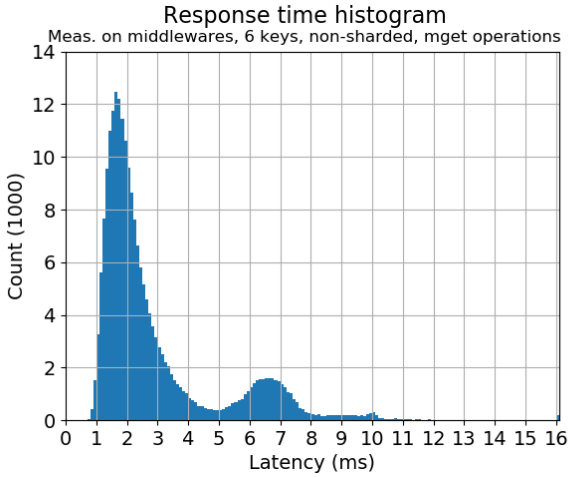
(a) Sharded case, client measurements



(b) Sharded case, middleware measurements



(c) Non-sharded case, client measurements



(d) Non-sharded case, middleware measurements

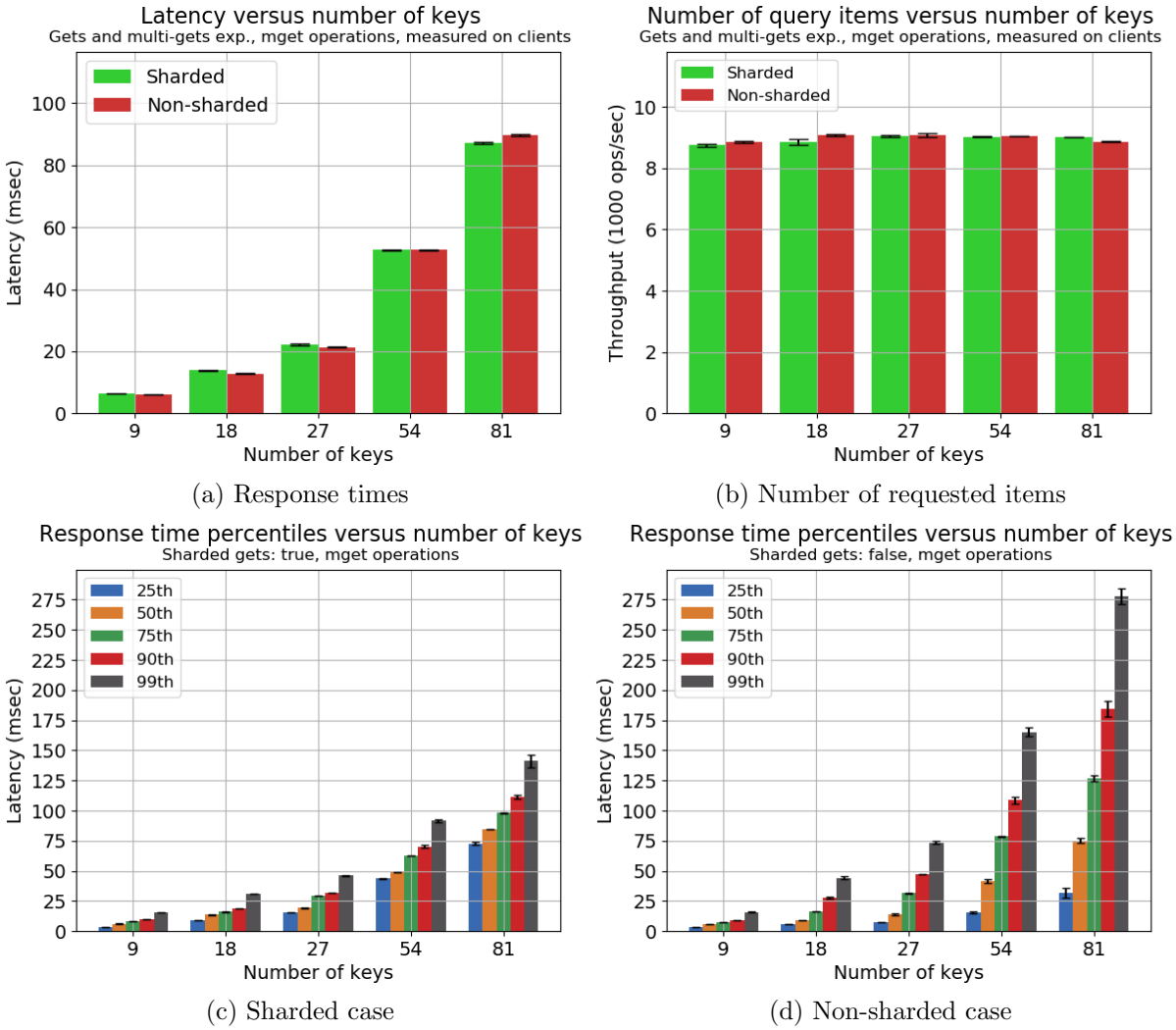
Figure 23: Response time distribution histograms for 6 key case. Bucket size is chosen as $100\mu s$ for all histograms, except for their last bins which contains the number of outlier response times that are greater than $16ms$. When choosing this cut-off value, it is ensured that less than 1% of the occurrences are collected in the last bin for each individual result used in aggregation.

others as they contain outlier values of a small sample size. This is essentially the reason why we see such a difference between the sharded and non-sharded reads.

Non-sharded ones send the complete request to one server, so it only experiences the end-to-end delay between a middleware and server, and only once. Network conditions may or may not be favorable when the data transfer happens, so the response time for that specific request should be expected to vary from time to time. Sharded read, on the other hand, splits the request into smaller parts and send them to different servers. Obviously this means we will have to experience more network delays in total, as we have noted earlier. However, by splitting the requests we diminish the effects of the network conditions between the middleware and the respective servers at the time the request is send and received. The way the sharded read sends smaller requests over the network further helps this, as the network conditions would be more observable in larger requests. Such conditions may or may not be favorable, so sharded case does not help requests to achieve a fast transfer —in fact, we are bound to wait the slowest connections to send and receive their respective split. We can observe this as

the sharded histograms are slightly shifted to the right (higher response times) both location-wise and weight-wise. As the source of this difference is the middleware-server connections and how we choose to use the utilize these connections, difference between histograms being more visible in middleware histograms make sense. Client histograms lessen the effects of different read strategies by adding additional delays to both. Lastly, we claim the sharded reads tend to output more stable response time distribution but with the cost of additional delays.

5.4 Summary



(a) Response times

(b) Number of requested items

(c) Sharded case

(d) Non-sharded case

Figure 24: An auxiliary experiment for comparing sharded and non-sharded performances on multi-get requests with bigger key sets. Figures 24a and 24b compare the average response time and number of requested items statistics for the sharded and non-sharded reads with 9, 18, 27, 54 and 81 key cases. Figures 24c and 24d illustrates 25th, 50th, 75th, 90th and 99th percentiles of sharded and non-sharded reads, for the experiments with aforementioned set of key sizes.

So far, we have discussed several aspects of the multi-get requests and the effects of different key sizes on sharded on non-sharded read strategies. Here, we inspect the experimental setup under larger key sizes to observe the trends in more detail. We test both strategies using 18, 27, 54 and 81 key sizes, on top of the 9 key case we have seen before. Results show that the our

comments still hold true for bigger multi-get requests and we see other trends emerge. Plots in Figure 24d show a summary of our auxiliary experiments with larger multi-get requests. These experiments are conducted on the same physical machines as the ones before, so reader should see them as natural extentions of the plots in Figures 21 and 22.

Figure 24a compares the mean response times for the sharded and non-sharded experiments for the large key sizes. Response time differences become negligible in 54 keys and non-sharded response times become larger afterwards. We can see this trend from another perspective in Figure 24b, where we see the 81 key case as the only configuration that has greater number of items requested for the sharded case than the non-sharded option. Bottleneck stays the same as in the 9 key case as we cannot see an increase in throughput (in terms of total get requests) as the system has already reached the network bandwidth. Queue lengths and queueing times are still negligible and the middleware is not fully utilized. We again turn our attention to the changing trend in the performance of sharded versus non-sharded reads in the light of this network limitation. As key sizes grow larger it becomes harder for the non-sharded case to allocate enough bandwidth to send big chunks of data across the network, whereas the sharded case can utilize server bandwidths more efficiently by requesting smaller replies from each server. For our case, this only becomes noticeable with comparably large key sizes, i.e. after 54 keyed multi-get requests. In a different setting, effects of network bandwidths could be more limiting and we can observe such changes with smaller key sizes (or vice versa with larger bandwidths).

Figures 24c 24d illustrate the percentiles of the response time distribution for the sharded and non-sharded configurations on larger key sizes. Unlike the smaller key sizes, we can see the different behaviours for two strategies more clearly where different percentiles scale much differently depending on the read strategy. 25th and 50th percentiles of the sharded reads are consistently greater than their non-sharded counterparts. This is not the case for the other ones as we see 75th percentiles favoring the sharded reads after 27 keys and 90th percentiles are always greater for the non-sharded reads. We talked about the inherent differences of two approaches in Section 5.3, where we have asserted that the sharded read approach results in more predictable response times but costs additional delays by doing so. This is corroborated by our percentile plots in this section as the underlying reasons are the same. Percentiles show that median response times are always smaller for the non-sharded read, confirming the sharded reads adds the requests more delay. Differences in other values show that the range of the response times are larger in the non-sharded reads. That is, fast replies are faster but the slow replies are slower in non-sharded reads. As the network conditions can fluctuate, non-sharded reads are effected (for better or for worse) more as they only use one connection at a time. We can see the mean and median differences in non-sharded case are larger and grow even more with larger key sizes. Overall, we see sharded reads generating more stable response times.

All in all, the preference of using one strategy over another may depend on the conditions and requirements of the system. It is harder to give reliable (or preferable) guarantees using non-sharded reads as they are affected more by the different conditions. We can give upper bounds for response times in sharded reads more easily, but this means we also report higher latencies. We suggest using sharded reads when dealing with large key sizes, limited bandwidths (e.g. in presence of network contention) and volatile or inconsistent network (or server) conditions. For our setting, it seems sensible to use sharding for multi-get requests that contain more than 54 keys, but the choice again depends on the tradeoff of having either smaller or more stable response times.

6 2K Analysis

In this section, we perform a 2^k factorial analysis on three parameters: number of middlewares, number of servers and the number of worker threads per middleware. We believe the effects of these parameters are unidirectional, so 2^k factorial analysis suits our interests best. We solve the linear equation systems for both throughput and response times on both read-only and write-only workloads. Our aim is to assess the impacts of different parameters for different configurations, as well as further investigating and whether an additive or a multiplicative model would capture the interactions and variations in our system best.

Parameter	Factor	Variable	Value at min. (-1)	Value at max. (+1)
Number of middlewares	A	xA	1	2
Number of servers	B	xB	1	3
Worker threads per middleware	C	xC	8	32

Table 7: 2^k experimental design used in this section, specifying the factors and treatments.

Table 7 shows the system parameters and their corresponding variables and values chosen for the analyses. For each configurationon each workload, we solve two linear systems, which are illustrated in Equations 1 and 2. The former uses the assumption that the effects of the factors are additive, whereas the latter one assumes multiplicative effects. Equation for the multiplicative model is only different from the additive one in the sense that the take the response variables as logarithms of the observed values. In the scope of this section e denotes error whereas r denotes the number of repetitions. We use the response time and throughput values measured in client machines.

$$y = q0+qA*xA+qB*xB+qC*xC+qAB*xA*xB+qBC*xB*xC+qAC*xA*xC+qABC*xA*xB*xC+e \quad (1)$$

$$log(y) = q0+qA*xA+qB*xB+qC*xC+qAB*xA*xB+qBC*xB*xC+qAC*xA*xC+qABC*xA*xB*xC+e \quad (2)$$

Before going into the results of the analysis, it is good to know how the linear equations are formed and solved. Table 27 (in the appendix) shows an example of such equation systems, in this case the response variable being the throughput of the system on write-only load. Left part of the table is a sign table where -1's refer to the minimum treatment and $+1$'s refer to the maximum treatment for a variable. We always conduct our factorial analyses with replications (so-called $2^k r$ design) so we solve our equations for the mean response (y_mean) and estimate the errors in response values as e1, e2 and e3 over three repetitions. After solving the equation system we obtain the effects of the variables (e.g. qA for variable xA or qBC for variable interaction xB*xC) and calculate allocation of variation for each factor through sum of squares values. Equations 3, 4 and 5 show how these calculations were done. The SSE accounts for the unexplained variation attributed to the experimental errors.

$$SSA = 2^2 qA^2, \text{ Fraction of variation explained (\%) by factor A} = SSA/SST \quad (3)$$

$$\text{Sum of Squares for Error: } SSE = \sum_{i=1}^{2^k} \sum_{j=1}^{r} e_{ij}^2, \text{ Fraction of unexplained variation} = SSE/SST \quad (4)$$

$$\text{Total Sum of Squares: } SST = SSA + SSB + SSC + SSAB + SSBC + SSAC + SSABC + SSE \quad (5)$$

Our principle reference for this chapter is Raj Jain's book [5] on system performance analysis. It is argued that the "knowledge about the system behavior should always take precedence over statistical considerations". As we are already familiar with the system through the different experimental setups in the previous sections, we do not view the system as a black box and rather choose the models that suits our parameter interactions best. For write-only workload we expect the factors xA (number of middlewares) and xC (number of threads per middleware) to

be the important parameters, since our limiting factor is the total number of middleware threads in the system. As the total number of threads is the multiplication of these two factors and we expect the variation caused by the number of servers and the errors to be small, we choose multiplicative model for the write-only load. Similarly, with the read-only load we expect only the facor xB (number of servers) to be effective on the final results, so we choose the additive model as it also assumes the errors of the system are additive, not multiplicative.

Although we choose the models that fit the inner workings of our system, it is a good practice to model the system using both additive and multiplicative models (for both workloads and response variables) to see if there are any critical differences between two approaches. Respective results for the not chosen models are presented in the appendix part of this report, in Tables 28, 29, 30 and 31. We see that the differences between additive and multiplicative models for all our analyses were not quite significant as both models essentially indicate the same results. There are several reasons behind this outcome. System is robust across repetitions and the residuals are low, thus modeling the errors one way or another does not affect the unexplained variation greatly. As it is also indicated in the book, "the logarithmic transformation is useful only if the ratio y_{max}/y_{min} is large. For a small range the log function is almost linear, and so the analysis with the multiplicative model will produce results similar to that with the additive model", which is exactly our case. Going from the additive to multiplicative model does not create a big difference in terms of the modelling power.

Results of the additive model on read-only load

Source	Mean Estimate	Variation Expl. (%)
qA	265.8600	0.009
qB	2678.2889	0.944
qC	263.8299	0.009
qAB	262.0704	0.009
qBC	260.5430	0.009
qAC	−257.6810	0.009
qABC	−260.9624	0.009

Table 8: Throughput analysis

Source	Mean Estimate	Variation Expl. (%)
qA	−0.0009	0.002
qB	−0.0208	0.988
qC	−0.0009	0.002
qAB	−0.0009	0.002
qBC	−0.0009	0.002
qAC	0.0009	0.002
qABC	0.0009	0.002

Table 9: Latency analysis

Now we turn our attention to the findings of the models. We present the mean estimate of the parameters and the percentage of variation in the response variable explained by the variation of those parameters for analyses on throughput and latency over read-only and write-only workloads. Tables 8 and 9 show the results for the read-only load. Clearly, the only important factor is the number of servers in the system while the other factors and factor interactions are equally unimportant. Analyses on both throughput and latency corroborate each other: number of servers have a strong positive influence on the throughput and a strong negative influence on the response time values. As we have discussed this issue on Sections 2 and 3, network bandwidth is the bottleneck for the read-only load most of the time. Our choice of minimum and maximum treatments for each factor (Table 7) results in number of servers being the bottleneck in every experiment except the 8 thread total with 3 servers case. This is in line with our expectations and findings in previous sections. Percentage of unexplained variation due to errors amount to 0.002 and 0.001, respectively for throughput and response time analyses. Therefore, we rule out the effects of sources other than the number of servers, including the error, as their variation does not really change values of the response variable.

For the write-only workload we expect the total number of worker threads to have a strong effect on system performance such that more worker threads should correspond to higher throughput and lower response time values. Similarly, we believe the number of servers for write-only

Results of the multiplicative model on write-only load

Source	Mean Estimate	Variation Expl. (%)
qA	0.1457	0.194
qB	−0.0822	0.062
qC	0.2832	0.733
qAB	0.0264	0.006
qBC	0.0137	0.002
qAC	−0.0017	0.000
qABC	0.0039	0.000

Table 10: Throughput analysis

Source	Mean Estimate	Variation Expl. (%)
qA	−0.1384	0.165
qB	0.0963	0.080
qC	−0.2730	0.642
qAB	−0.0192	0.003
qBC	−0.0031	0.000
qAC	0.0124	0.001
qABC	0.0065	0.000

Table 11: Latency analysis

workloads should hinder the system performance, perhaps not as noticeably as with the total number of threads. We first send the queries to all servers and then wait for their responses. That way we do not experience the RTT for all the servers, although the time a worker thread waits increases linearly with the number of servers. In the optimistic case, reply from the first server would arrive as we finish sending the query to last server.

In Tables 10 and 11 we can see the factorial analysis results for the write-only workload. Number of middlewares and the number of threads per middleware together account for the vast majority of the variation for both throughput and latency as expected. The reason that the variation due to number of threads per middleware (xC) is greater than that of the number of middlewares (xA) should not be surprising as the min-max values for xC covers a greater change than the xA, with respect to total number of threads (Table 7). We can also see xB having a slightly negative influence on throughput (vice versa for response times) since the number of servers only affect the performance marginally, as we have demonstrates in Section 4.2 with the help of findings shown in Figure 20. Unexplained variations (0.003 and 0.108 for throughput and response times, respectively) for the write-only load is greater than those for the read-only load, since the system conditions (e.g. end-to-end delays between physical machine pairs) have an effect on the performance of the system in the observed value ranges. It was not the case with the read-only workload as the network bandwidth limited the performance such that changing the number of threads in our range did not change the bottleneck being the bandwidth.

7 Queuing Model

In this section, we try to model our system through various approaches. We begin modeling our system with M/M/1 and M/M/m queues, respectively in Sections 7.1 and 7.2. Table 12 explains the notation used through these two sections. Lastly, in Section 7.3, we construct networks of queues that better matches the underlying structures of the actual system. All time values used in the subsections of this sectino are given in milliseconds.

7.1 M/M/1

We isolate system from the middleware perspective and try to model the system using middleware measurements. It is clear that we should not be able to model our system using only one queue, so we use middleware statistics instead of the client ones to at least get rid of the client-middleware communication latencies while modelling our system. Yet, by constructing an M/M/1 queue we are making the assumption that our all the middlewares and servers in our system acn be seen as one black-box system.

We determine a service rate for each number of worker threads and construct a different M/M/1 model for that specific configuration. Tables 13, 14, 15 and 16 show the results for

ρ	Traffic intensity
$E[n]$	Mean num. of jobs in the system
$E[n_q]$	Mean num. of jobs in the queue
$E[r]$	Mean response time
$E[w]$	Mean waiting time

λ	Arrival rate
qlen	Avg. queue length per middleware
2×qlen	Avg. queue length of the system
qtime	Avg. queueing time
wtime	Avg. server waiting time

Table 12: Notation used throughout in Sections 7.1 and 7.2. Traffic intensity shows the utilization of the queues for both models. Mean waiting time stands for time waited in the queue and corresponds to qtime statistic we measure. Similarly, mean response time denotes the total amount of time, which should ideally match $qtime + wtime$ of ours. Since qlen only denotes the average queue length over two middlewares we report 2×qlen as well, so we can better compare the model output $E[n_q]$, which should account for the total number of requests waiting to be serviced.

num	Model outputs						Middleware meas.			
Cli	λ	ρ	$E[n]$	$E[n_q]$	$E[r]$	$E[w]$	qlen	2×qlen	qtime	wtime
6	1060.2	0.278	0.385	0.107	0.36	0.10	0.0	0.0	0.078	3.6
24	3492.9	0.916	10.8	9.9	3.11	2.84	2.0	4.0	1.11	3.9
48	3783.7	0.992	121.3	120.3	32.05	31.79	12.2	24.4	6.49	4.2
96	3740.3	0.980	50.1	49.2	13.41	13.14	36.4	72.8	19.54	4.3
192	3737.8	0.979	48.5	47.5	12.97	12.71	84.8	169.6	45.38	4.3

Table 13: Results of the M/M/1 model for the 8 worker thread configuration. Service rate is $\mu = 3814.9$, which is the absolute maximum throughput this configuration can achieve.

num	Model outputs						Middleware meas.			
Cli	λ	ρ	$E[n]$	$E[n_q]$	$E[r]$	$E[w]$	qlen	2×qlen	qtime	wtime
6	1056.0	0.205	0.257	0.0527	0.24	5e-02	0.1	0.2	0.079	3.6
24	3823.2	0.741	2.86	2.12	0.748	0.555	0.4	0.8	0.246	4.0
36	4673.6	0.906	9.62	8.71	2.06	1.86	1.4	2.8	0.622	5.0
48	4887.3	0.947	17.96	17.01	3.67	3.48	4.6	9.2	2.02	5.8
96	5091.2	0.987	74.65	73.66	14.66	14.47	26.8	53.6	10.66	6.3
192	5077.2	0.984	61.77	60.78	12.16	11.97	74.2	148.4	29.44	6.3

Table 14: Results of the M/M/1 model for the 16 worker thread configuration. Service rate is $\mu = 5159.4$, which is the absolute maximum throughput this configuration can achieve.

the models constructed for 8, 16, 32 and 64 threads, respectively. We consider the maximum throughput achieved by a specific configuration indicates its maximal capacity and choose that value as the service rate for that configuration. By constructing and testing our models we use the data obtained through the experiments in Section 4.

First thing to notice is that M/M/1 models estimate constant service time, which does not fit the reality. There is queueing at both network and the servers and M/M/1 models inherently cannot account for those waiting times in between. The actual system is load-dependent, and the reported server waiting times show that a constant service time assumption would not fit the real system at all. Similarly, the models assume the number of requests in service can at most be 1 and predict overly optimistic service rates for every number of clients. Our system is a closed one, meaning a client does not send any new requests under its current request has been replied. We cannot model a closed system using only a M/M/1 queue, so the number of requests inside the system will always be off as the model will output numbers proportionate to the predicted utilization of the system.

num Cli	Model outputs						Middleware meas.			
	λ	ρ	$E[n]$	$E[n_q]$	$E[r]$	$E[w]$	qlen	2×qlen	qtime	wtime
6	1046.8	0.154	0.183	0.028	0.174	2.7e-02	0.0	0.0	0.083	3.6
24	3823.6	0.564	1.29	0.73	0.339	0.191	0.4	0.8	0.237	4.0
48	5206.5	0.768	3.32	2.55	0.636	0.489	1.5	3.0	0.494	6.7
72	6411.0	0.946	17.53	16.58	2.73	2.59	4.0	8.0	1.148	8.4
96	6715.1	0.991	109.0	108.0	16.23	16.09	11.7	23.4	3.363	9.0
192	6742.6	0.995	197.7	196.7	29.32	29.18	55.9	111.8	16.806	9.5

Table 15: Results of the M/M/1 model for the 32 worker thread configuration. Service rate is $\mu = 6776.7$, which is the absolute maximum throughput this configuration can achieve.

num Cli	Model outputs						Middleware meas.			
	λ	ρ	$E[n]$	$E[n_q]$	$E[r]$	$E[w]$	qlen	2×qlen	qtime	wtime
6	1045.4	0.126	0.144	0.018	0.138	1.7e-02	0.0	0.0	0.084	3.6
24	3803.6	0.459	0.849	0.389	0.223	0.102	0.3	0.6	0.250	4.1
48	5187.8	0.626	1.675	1.05	0.323	0.202	1.2	2.4	0.484	6.8
96	7443.5	0.898	8.857	7.96	1.19	1.07	3.9	7.8	0.829	10.4
144	8161.1	0.985	66.46	65.47	8.14	8.02	8.6	17.2	1.779	13.9
192	8221.8	0.992	132.4	131.4	16.10	15.98	24.2	48.4	5.932	15.3

Table 16: Results of the M/M/1 model for the 64 worker thread configuration. Service rate is $\mu = 8283.9$, which is the absolute maximum throughput this configuration can achieve.

Even under these constraints, M/M/1 models occasionally make meaningful predictions regarding queue lengths, although very rarely. 32 and 64 configuration with small number of clients could be given as examplary better predictions, where the trend in the model outputs correlate the actual trend observed in the system. However, even predictions about queue lose their accuracy very fast when we examine the saturated configurations of our system.

7.2 M/M/m

num Cli	Model outputs						Middleware meas.			
	λ	ρ	$E[n]$	$E[n_q]$	$E[r]$	$E[w]$	qlen	2×qlen	qtime	wtime
6	1056.0	0.510	16.32	0.00	15.45	0.00	0.1	0.2	0.079	3.6

Table 17: Results of the M/M/m model for the 16 worker thread configuration. Service rate is $\mu = 64.718$, which is the absolute maximum throughput per thread of the 64 thread configuration.

num Cli	Model outputs						Middleware meas.			
	λ	ρ	$E[n]$	$E[n_q]$	$E[r]$	$E[w]$	qlen	2×qlen	qtime	wtime
6	1046.8	0.253	16.17	0.00	15.45	0.00	0.0	0.0	0.083	3.6
24	3823.6	0.923	64.17	5.09	16.78	1.33	0.4	0.8	0.237	4.0

Table 18: Results of the M/M/m model for the 32 worker thread configuration. Service rate is $\mu = 64.718$, which is the absolute maximum throughput per thread of the 64 thread configuration.

Now we try modelling our system through M/M/m queues using the experiment results in Section 4. Here, we again use middleware statistics instead of client ones as per the reasons

num Cli	Model outputs						Middleware meas.			
	λ	ρ	$E[n]$	$E[n_q]$	$E[r]$	$E[w]$	qlen	2×qlen	qtime	wtime
6	1045.4	0.126	16.15	0.00	15.45	0.00	0.0	0.0	0.084	3.6
24	3803.6	0.459	58.77	0.00	15.45	0.00	0.3	0.6	0.250	4.1
48	5187.8	0.626	80.16	0.00	15.45	0.00	1.2	2.4	0.484	6.8
96	7443.5	0.899	116.46	1.45	15.65	0.19	3.9	7.8	0.829	10.4
144	8161.1	0.985	179.83	53.73	22.04	6.58	8.6	17.2	1.779	13.9
192	8221.8	0.993	246.16	119.12	29.94	14.49	24.2	48.4	5.932	15.3

Table 19: Results of the M/M/m model for the 64 worker thread configuration. Service rate is $\mu = 64.718$, which is the absolute maximum throughput per thread of the 64 thread configuration.

num Cli	Model outputs						Middleware meas.			
	λ	ρ	$E[n]$	$E[n_q]$	$E[r]$	$E[w]$	qlen	2×qlen	qtime	wtime
6	1046.8	0.154	9.89	0.00	9.44	0.00	0.0	0.0	0.083	3.6
24	3823.6	0.564	36.11	0.00	9.44	0.00	0.4	0.8	0.237	4.0
48	5206.5	0.768	49.26	0.09	9.46	0.02	1.5	3.0	0.494	6.7
72	6411.0	0.946	70.39	9.84	10.98	1.54	4.0	8.0	1.148	8.4
96	6715.1	0.991	163.08	99.66	24.29	14.84	11.7	23.4	3.363	9.0
192	6742.6	0.995	251.90	188.22	37.36	27.92	55.9	111.8	16.806	9.5

Table 20: Results of the M/M/m model for the 32 worker thread configuration. Service rate is $\mu = 105.886$, which is the absolute maximum throughput per thread of the 32 thread configuration.

discussed in the previous section. To model an M/M/m queue, we need to determine a service rate for individual threads. Once more, we make the assumption that the absolute maximum throughput we observe in our experiments stand for the capacity of our middleware for that specific configuration. We also assume our middlewares are ideal, that is, each worker thread is identical to one another —be it in the same middleware or not. Under both this assumptions, we divide the maximum observed throughput by the total number of middleware threads to obtain the service rate of one individual thread.

Figures 17, 18 and 19 show the results for the M/M/m queues constructed using the service rate obtained through the 64 thread configuration. 8 thread configuration and most of the 16 and 32 thread configurations cannot be modeled through the service rates obtained from the 64 threads, as the resulting system would be unstable, so their results cannot be shown here. The reason is that small number of threads utilize the server more efficiently, if viewed thread-wise. Since the servers would be able to handle fewer connections, actual service rates through those connections would be higher. Input service rates obtained through the 64 thread configuration would not be suitable for aforementioned conditions as the actual experimental setup would be impossible to achieve for the constructed models. For that reason, we also construct M/M/m models for each thread configuration using the values obtained through their own experiments. Results of such a model is shown in Table 20 for 32 threads and the model results for 8 and 16 threads can be found in appendix, respectively in Tables 25 and 26.

M/M/m models are different from the M/M/1 queues in the sense that they model a pool of workers that uses a common queue. This is exactly the situtation inside the middleware machines, although by using only one M/M/m queue we will be trying to model more than one middleware as one. Aside from this multiple worker approach M/M/m can not be helpful for our modelling purposes as we still cannot capture many aspects of the real world system, e.g. the real setup being a closed system.

Although we get comparably more realistic queueing and service times, M/M/m proves to be insufficient as well as it cannot model a load dependent service. Server waiting times of the actual system increase along with the number of clients, whereas the M/M/m model always predicts a constant service time for all client configurations. Since we calculate the service rates using the values from the configuration that achieved maximum throughput, it is sensible for that configuration to be the one with most clients in the system. That means our service times converge to the actual vlaues as we increase the number of clients. However, modelling the whole system as an M/M/m queue ignores the two separate queues (respectively for each middleware) and fails to produce accurate queueing times, even with large number of clients.

7.3 Network of Queues

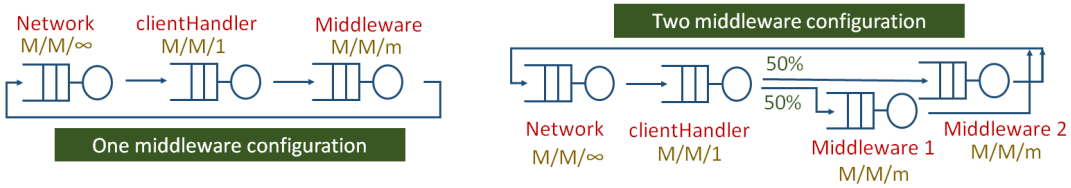


Figure 25: Diagram showing the network of queues models for one and two middleware configurations.

num Cli	Model outputs				Middleware meas.			
	MW Util	TP	Lat	qlen	TP	Lat	qlen	qtime
6	0.025	929.3	6.46	1.58	922.4	6.522	0.0	0.118
24	0.089	3564.1	6.73	5.70	3418.2	7.031	0.4	0.282
48	0.196	6263.1	7.66	12.53	5907.5	8.132	2.2	0.444
72	0.372	8656.8	8.32	23.81	8098.6	8.896	2.9	0.540
96	0.590	9559.6	10.04	37.76	9007.3	10.668	4.2	0.599
192	0.434	4787.6	40.10	90.99	10566.4	18.206	23.3	4.557
288	0.756	8065.4	35.71	135.51	10639.9	27.110	69.7	12.983

Table 21: Results of the network of queues model for two middlewares with 64 worker threads, on write-only workload

num Cli	Model outputs				Middleware meas.			
	MW Util	TP	Lat	qlen	TP	Lat	qlen	qtime
6	0.109	1024.3	5.86	0.87	1001.6	6.008	0.0	0.112
12	0.228	2145.3	5.59	1.82	2093.3	5.743	0.1	0.146
18	0.387	3094.8	5.82	3.10	2967.0	6.074	0.4	0.252
24	0.725	3316.5	7.24	6.14	2966.0	8.107	1.5	1.114
48	0.965	3086.9	15.55	18.60	2968.7	16.223	11.2	7.729

Table 22: Results of the network of queues model for two middlewares with 8 worker threads, on read-only workload

Now, we move on from modelling the whole system using only one queue to instead using a network of queues. Using more complex models does not only allow more parameters to fit the model, but also allows more a priori knowledge to be transferred into the model —for example, we were simply not able to model the client-middleware RTTs alongside the rest of the system,

num Cli	Model outputs				Middleware meas.			
	MW Util	TP	Lat	qlen	TP	Lat	qlen	qtime
6	0.052	1151.1	5.21	3.34	1118.5	5.369	0.1	0.174
24	0.194	4606.7	5.21	12.44	4240.1	5.662	1.4	0.418
48	0.383	7430.3	6.46	24.52	6567.8	7.316	7.0	0.841
72	0.733	9197.5	7.83	46.91	7957.7	9.062	11.3	1.291

Table 23: Results of the network of queues model for one middleware with 64 worker threads, on write-only workload

num Cli	Model outputs				Middleware meas.			
	MW Util	TP	Lat	qlen	TP	Lat	qlen	qtime
6	0.024	1189.2	5.05	1.55	1151.2	5.254	0.2	0.170
12	0.047	2494.5	4.81	2.99	2350.5	5.106	0.5	0.260
24	0.244	3125.5	7.68	15.63	2969.3	8.083	1.3	0.406
48	0.625	3148.6	15.24	39.99	2978.8	16.124	4.0	0.901

Table 24: Results of the network of queues model for one middleware with 64 worker threads, on read-only workload

using M/M/1 and M/M/m approaches. We construct two separate models for the one and two middleware setups. These models can be viewed in Figure 25. Middlewares are modeled as separate M/M/m queues as they consist of a set number of workers using a common queue. RTT between client and middleware machines are assumed ideal and -modeled as M/M/∞ (i.e. delay centers or infinite servers), although we have discussed in Section 5.4 and its preceding sections that this is not the case. Since we can now model this RTT with the network of queues approach, we are now able to test our models using client-side statistics. We model the net-thread of the middleware as an M/M/1 queue as the clientHandler consists of only one thread reserved for client communication. We use some additional measurements to determine the service time of the clientHandler for this part. We continue to use one component to model the net-thread in the two middleware setup as well, as its service time is found to be negligible. We conduct mean-value analysis on our networks of queues, with the hep of the queueing package [7] for Octave.

Using a network of queues, we overcome much of the difficulties we met in the earlier sections. We can now model our setup as a closed system and we can configure our workers to output load-dependent service times, which we could not do using primitive models. As described above, we model can model two middleware-server pipelines separately and account for delays due to client-middleware connections as well as net-threads. Still, the models we inspect here are quite simplistic and they leave much room for improvement. To illustrate, we again do not distinguish middlewares and servers from each other. We cannot input the number of servers or their number of threads into our models, neither can we separate the middleware-server RTTs from the operational overheads. That means definitions of queue length for the network of queues model and the actual system still does not quite match.

References

[1] Memcached: a distributed memory object caching system https://memcached.org/

[2] memtier_benchmark: a high-throughput benchmarking tool for Redis & Memcached https://github.com/RedisLabs/memtier_benchmark

[3] Dstat: a versatile resource statistics tool http://dag.wiee.rs/home-made/dstat/

[4] iPerf: the TCP, UDP and SCTP network bandwidth measurement tool https://iperf.fr/

[5] Jain, R. (1990). The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling. John Wiley & Sons.

[6] Cluster SSH: cluster administration tool via SSH https://github.com/duncs/clusterssh

[7] A Queueing Package for GNU Octave https://www.moreno.marzolla.name/software/queueing/

Appendix

M/M/m: additional tables

num Cli	Model outputs						Middleware meas.			
	λ	ρ	$E[n]$	$E[n_q]$	$E[r]$	$E[w]$	qlen	2×qlen	qtime	wtime
6	1060.2	0.278	4.45	0.00	4.19	0.00	0.0	0.0	0.078	3.6
24	3492.9	0.916	21.67	7.02	6.20	2.01	2.0	4.0	1.11	3.9
48	3783.7	0.992	132.53	116.66	35.03	30.83	12.2	24.4	6.49	4.2
96	3740.3	0.980	61.33	45.64	16.40	12.20	36.4	72.8	19.54	4.3
192	3737.8	0.980	59.67	43.99	15.96	11.77	84.8	169.6	45.38	4.3

Table 25: Results of the M/M/m model for the 8 worker thread configuration. Service rate is $\mu = 238.431$, which is the absolute maximum throughput per thread of the 8 thread configuration.

num Cli	Model outputs						Middleware meas.			
	λ	ρ	$E[n]$	$E[n_q]$	$E[r]$	$E[w]$	qlen	2×qlen	qtime	wtime
6	1056.0	0.205	6.55	0.00	6.20	0.00	0.1	0.2	0.079	3.6
24	3823.2	0.741	23.92	0.21	6.26	0.05	0.4	0.8	0.246	4.0
36	4673.6	0.906	33.63	4.64	7.20	0.99	1.4	2.8	0.622	5.0
48	4887.3	0.947	42.54	12.23	8.71	2.50	4.6	9.2	2.02	5.8
96	5091.2	0.987	99.72	68.15	19.59	13.39	26.8	53.6	10.66	6.3
192	5077.2	0.984	86.81	55.32	17.10	10.90	74.2	148.4	29.44	6.3

Table 26: Results of the M/M/m model for the 16 worker thread configuration. Service rate is $\mu = 161.231$, which is the absolute maximum throughput per thread of the 16 thread configuration.

2K Analysis: additional tables

I	xA	xB	xC	xA*xB	xB*xC	xA*xC	xA*xB*xC	y_mean	e1	e2	e3
1	−1	−1	−1	+1	+1	+1	−1	3364.3	45.56	-15.33	-30.23
1	+1	−1	−1	−1	+1	−1	+1	4318.9	-19.85	-1.77	21.63
1	−1	+1	−1	−1	−1	+1	+1	2655.2	23.47	37.06	-60.54
1	+1	+1	−1	+1	−1	−1	−1	3729.6	9.09	-39.41	30.32
1	−1	−1	+1	+1	−1	−1	+1	5832.8	6.49	-17.43	10.93
1	+1	−1	+1	−1	−1	+1	−1	7321.3	17.68	-21.73	4.04
1	−1	+1	+1	−1	+1	−1	−1	4786.4	27.09	-18.71	-8.38
1	+1	+1	+1	+1	+1	+1	+1	6782.0	-16.40	-2.95	19.35

Table 27: An example system of equations for the 2^k factorial analyses. Specifically, this set of equations with the corresponding y values model the throughput for the write-only load, assuming additive variable effects. Variable xA, xB and xC denote respectively the effects of number of middlewares, number of servers and the number of worker threads in our model.

Running the code

All the necessary commands for initializing the experimental setup and conducting the experiments are provided in a companion file called `commands.sh`. It also exemplifies the usage of

Results of the additive model on write-only load

Source	Mean Estimate	Variation Expl. (%)
qA	689.1529	0.196
qB	−360.4905	0.054
qC	1331.8182	0.731
qAB	78.3606	0.003
qBC	−35.9031	0.001
qAC	181.8770	0.014
qABC	48.4125	0.001

Table 28: Throughput analysis

Source	Mean Estimate	Variation Expl. (%)
qA	−0.0064	0.175
qB	0.0045	0.087
qC	−0.0120	0.620
qAB	−0.0015	0.010
qBC	−0.0013	0.008
qAC	0.0023	0.022
qABC	0.0008	0.003

Table 29: Latency analysis

Results of the multiplicative model on read-only load

Source	Mean Estimate	Variation Expl. (%)
qA	0.0348	0.004
qB	0.5144	0.974
qC	0.0345	0.004
qAB	0.0335	0.004
qBC	0.0334	0.004
qAC	−0.0330	0.004
qABC	−0.0341	0.004

Table 30: Throughput analysis

Source	Mean Estimate	Variation Expl. (%)
qA	−0.0351	0.005
qB	−0.5130	0.970
qC	−0.0353	0.005
qAB	−0.0348	0.004
qBC	−0.0346	0.004
qAC	0.0348	0.004
qABC	0.0356	0.005

Table 31: Latency analysis

auxiliary technologies used for further analysing the system and its environment as well as it contains some other helpful commands for copying and plotting the experiment results. `plot` folder contains all scripts (and their eventual outputs) necessary for plotting the results placed under the `res` folder. Every plot provided inside or alongside this report is reproducible with the provided data and the scripts. Lastly, `runner.sh` is the encapsulating script that runs the desired agent (i.e. client, middleware, server or dstat tool) for the configuration provided through the command line arguments. The arguments it expects share the same nomenclature we use in scripts to refer the experiment parameters. Here, in Table 32, we list these abbreviations. Result directory hierarchy also obeys the same nomenclature.

It is important to pay attention the order and timing of executing commands. Servers should be run first and the middlewares (if any) should start waiting connections. Then, both the clients and dstat processes should start simultaneously — we could do this by using Cluster SSH (also called as cssh) [6]. Clients, middlewares and dstat processes ran accordingly for three separate repetitions, afterwards they should be restarted by hand. Another important thing to keep in mind while using the project codebase is to remember that the codes do not cover erroneous cases. For example, middleware does not support any other operations than set or get. Similarly, plotting scripts may not work with data with erroneous content, filenames, directory hierarchy and such. One really important thing to note is that in its current state the scripts create and use folder names that contain the colon (`:`) character. Readers should keep in mind that such folders work fine on Linux and MAC operating systems but would be problematic for Windows versions.

Internal parameters

The system has several options hardcoded as parameters of MyMiddleware class:

- `verboseLogs` specifies if the system should output the work logs or not. Such logs are printed when a request is received, relayed to servers, got answered and replied back to

Number of servers	nsvr	Multi-Get behavior	mgshrd
Number of client machines	ncli	Multi-Get size	mgsize
Instances of memtier per machine	icli	Number of middlewares	nmw
Threads per memtier instance	tcli	Worker threads per middleware	tmw
Virtual clients per thread	vcli	Repetitions	reps
Workload	wrkld	Test time	ttime

Table 32: Abbreviations for experiment parameters.

the clients. It also prints out details about which thread handled the request, which client sent the request, to which server the request was sent to and the completely parsed messages and replies. It is a useful option for understanding the inner workings and state transitions of the middleware system. It should be set as false if user is not debugging.

- `verboseAggr` controls the outputting of the aggregated statistics. It is always true for our experiments as it is our main tool for analysing the system.
- `timeoutSecs` specifies the amount of time the system should wait until new queries before shutting itself down. Although the middleware can work seamlessly without any interruption, systems needs to be restarted to operate in its most efficient manner in its current implementation. 3 seconds are proved to be enough for our experiments.
- `initDelaySecs` and `periodSecs` respectively specify the initial delay and the period of the ScheduledControl runs. Both are given as 1, that means the ScheduledControl thread begins collecting data after an initial delay of 1 seconds and it wakes itself at the end of every 1 second period after that.
- `sep` sets the column separator to be used by ScheduledControl when outputting aggregated statistics. We output the aggregation table inside the output file in a comma separated format, so we set sep as `','`.