Advanced Systems Lab Report

Autumn Semester 2017

Name: Gokula Krishnan Santhanam
Legi: 16-948-309

Grading

Section	**Points**
1	
2	
3	
4	
5	
6	
7	
Total	

0 Report Overview

This report consists of several sections, going into depth analyzing of the middleware that we have implemented. This sections gives an overview of what to expect in each section and provides a cogent argument as to why the report is structured the way it is structured. Note that this is in no way exhaustive and the corresponding sections will provide more context as well.

We start with describing the system that we built in detail in the *System Overview* section. This goes into the fine-grain details of the design choices we made as well as how the middleware handles the different requests outlined in the project description. After this, we first analyze the case when there is no middleware in the *Baseline without Middleware* section, this shows us the throughput and latency trends when there is no middleware and also gives a sense of the limits of the memtier clients and memcached servers. We also analyze the bottlenecks in this section.

Now, we introduce the Middleware in *Baseline with Middleware* and repeat similar experiments to understand the limits of the middleware as well as how the throughput and response time trends change as a result of introducing the middleware. We further analyze the bottlenecks in this setup.

In the middleware, we handle SETs, GETs and multi-GETs different and would like to know how exactly the trends in throughput and latencies vary for these different kinds of requests. Especially because we replicate in the case of SETs and might do sharding in the case of multi-GETs. We analyze the trends for these different types of requests in the *Throughput for SETs* and *Gets and Multi-gets* sections respectively.

In our experiments with memtier clients, memcached servers and middlewares, we can fundamentally change a few properties. These are the number of virtual clients per thread at the load generating VMs, the number of middlewares in place and the number of worker threads per middleware. We analyze how sensitive the throughput and latency is to these parameters by conducting a 2k analysis in the *2K Analysis* section.

Finally, we model our system in accordance to queueing theory and see how well the things we observe correspond to theory in the final section, *Queuing Model.*

Several appendices are also provided which delve into more details.

1 System Overview

The three main components of the middleware are

- Net Thread / frontend (`MyMiddleware.java`)
- Request Queue and Request object (`Request.java`)
- Worker Threads / backend (`Worker.java`)

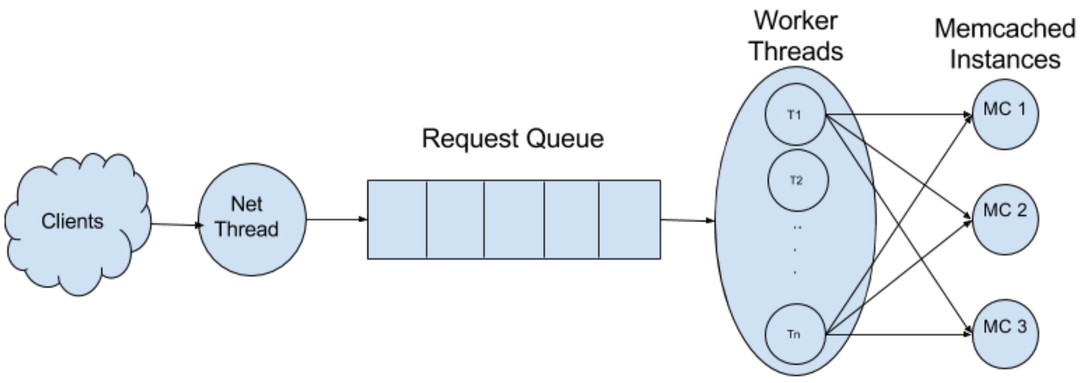
In addition to the above three, there is another class `utils.java` which is used to provide methods to handle parsing and other auxiliary methods which will be discussed in a later section. This enables a modular design and makes the code less error prone. Loggers are added to both the Net and Worker threads in order to log any errors and other information. More details about the instrumentation are discussed in a later section as well. Before processing these logs, the error logs are checked to make sure no errors occurred before proceeding further with the analysis. This ensures that the data used in the subsequent analyses is sane and the results are meaningful.

All the code for the middleware are under the `src` directory in the projects repository. The scripts used to postprocess the logs are in the `scripts` directory. More information about each script can be obtained from the `README` files in the same directory. The logs are stored in the `logs` directory.

All code has been extensively documented and tested with experiments repeated to make sure that no bugs had creeped into the middleware and that variations due to the VM allocation in the cloud did not severely the measurements. All logs like mentioned above were checked before being included in the analysis. This is true of logs generated by both Middleware and Memcached.

A simple graphical representation of the middleware is shown below.

Figure 1: Components of the middleware



1.1 Net Thread

The Net Thread is the entry-point into the middleware and is created by the `RunMW.java` script. In its constructor, the net thread creates the request queue and also creates an appropriate number of worker threads with the arguments it is given. Once these are created, the net thread then creates a non-blocking `ServerSocketChannel` to listen to new connections. This and any new connections that are established are put into a `Selector` and polled. The Selector enables us to handle multiple connections efficiently using just a single thread. Both the `ServerSocketChannel` and `SocketChannel`s corresponding to the established connections are configured to be non-blocking in order to make full use of the Non-Blocking IO facilities provided by Java. The `ServerSocketChannel` is registered with `OP_ACCEPT` and the `SocketChannel`s are registered with `OP_READ` SelectionKeys in order to be notified when there are new connections and when there is data to be read on the connection respectively. This way, we use only one selector to handle a ServerSocket channel and N SocketChannels (where N is the number of clients connected to the middleware). Any time a connection is closed, it is removed from the selector. When a connection is closed, it is selected by the selector and when `read` is called on the socket channel, we read back *-1* bytes indicating that the channel was closed by the remote peer (the Virtual Client on the memcached instance).

The Net thread blocks on calling `select` of the Selector, which returns only when one of the registered channels have a new connection or some data to be read. Thus, the net thread does not do busy waiting. Whenever there's a new connection, we obtain the socket channel corresponding to that connection by calling `accept` on the ServerSocketChannel and add it to the selector. This is handled by the `addNewConnection` method of the Net thread.

Whenever there's incoming data in the established connections, we create a `ByteBuffer` instance of appropriate size and write into it. We then check if the message is complete (using `utils.isCompleteRequest`) by counting the number of new-lines in the received request. The request is complete if there are one and two new-lines in the case of `GET` (also `multi-GET`) and `SET` request respectively.

If the request is not complete, we attach the byte buffer to the SelectionKey (using the `.attach` method)and continue processing other requests. This allows the middleware to not wait for one client to send a complete request before processing other clients.The next time there is new data in the same channel, the attached byte buffer is used instead of creating a new one. We check for completion every time we read from a socket channel.

When the request is complete, we create a new `Request` object with the byte buffer and the corresponding socket channel and add it to the request queue. the above workflow of handling data in established connections is handled by the `readRequest` method.

Every time we add a new request to the request queue, we log the current length of the queue in a separate logger.

1.2 Request Queue

The request queue is an instance of the `LinkedBlockingQueue<Request>` class. It contains instances of the `Request` class which encapsulate all the relevant information about a request as it passes through the middleware. We use this particular class due to several performance and design reasons which we elaborate in this section.

First, LinkedBlockingQueues are unbounded, this means that they can grow indefinitely (subject to total memory available, of course). Due to this unboundedness, we are assured that no request that arrives to the middleware is dropped and thus makes our later analysis simpler.

Second, the LinkedBlockingQueue is thread safe, which means that multiple worker threads can safely dequeue from it and we dont have to implement any complicated synchronization routines to ensure data integrity.

Finally, It also allows worker threads to block on it when there are no requests available for them to process (hence the name `LinkedBlockingQueue`). This prevents busy waiting and the resultant wasting of CPU clock cycles thereby improving the utilization of resources available at the middleware. Moreover, it can wake up exactly one worker thread blocking on it when theres a new request (internally it uses `notify` and not `notifyAll`) thus preventing the thundering herd problem that occurs when multiple threads sleep on a single resource.

As we just saw, the `LinkedBlockingQueue` satisfies all the design and performance requirements we need of a request queue in the middleware and was chosen as a result.

1.3 Worker Threads

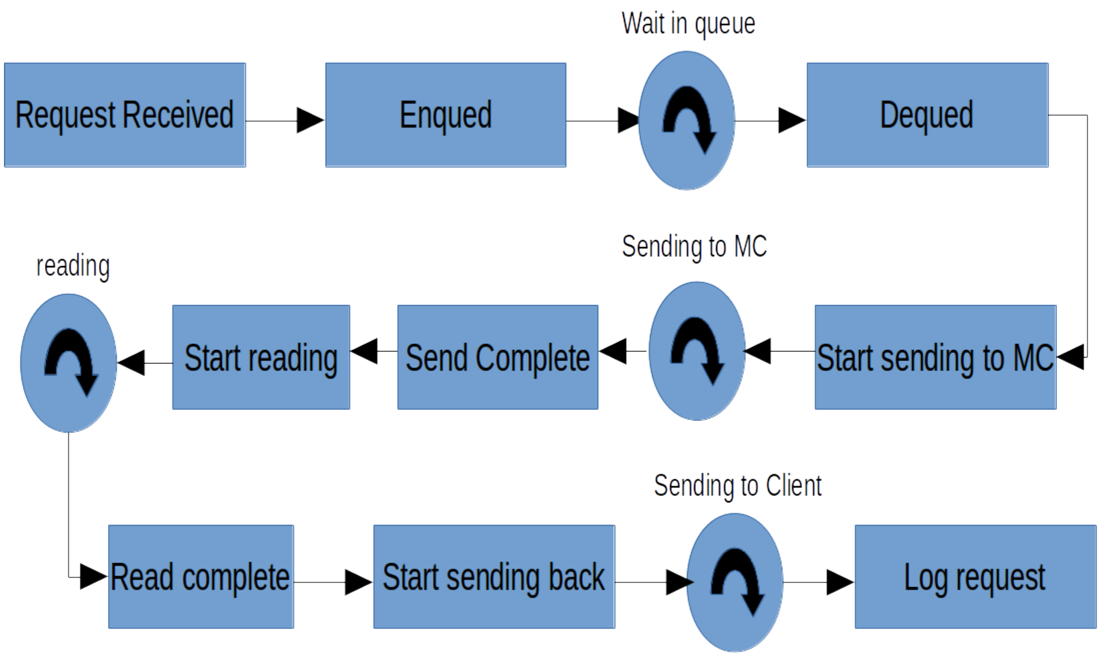
The worker threads do a majority of the tasks inside the middleware. These are started by the Net Thread during initialization. Each worker thread is given the following information as arguments to its constructor.

1. List of Memcached IP and Ports
2. A Reference to the Request Queue
3. Boolean flag (sharded or non-sharded mode)
4. Worker ID and logging directory

The Worker ID is used to uniquely identify a thread among all worker threads. This, along with the logging directory allows the thread to store its information in an uniquely named log file.

Inside the constructor, the worker thread creates a connection to each of the Memcached instances and adds the socket channels to an `ArrayList`. Each worker thread creates a blocking socket channel to each one of the memcached servers. resulting in num_worker_thread * num_memcached_servers connections in total.

Figure 2: Life of a request inside the middleware



These socket channels are configured to be blocking because our worker threads are synchronous and handle only one request at a time. Nevertheless, We still use the socket channel because it provides for an unified interface at both the Net thread and the Worker thread and makes the implementation easier and less error prone as you only need to pass byte buffers around. The worker thread also creates loggers to log information about the requests it processes and any errors that it may encounter during an experiment. Once again, these logs are checked (after the experiment is completed) before running any analysis on them to ensure that we are working with logs which have meaningful data and that our experiments did not run into any errors or unusual behaviours.

In the `run` method, the worker threads block on `requestQueue.take` until is woken up by a new request in the queue. The worker thread dequeues it and calls `processRequest` to handle it. All the information about the current request is logged once the control return backs from the `processRequest` method.

processRequest inspects the request to check if it is a `GET` or `SET` and passes it along to `processGetRequest` and `processSetRequest` accordingly.

The life-time of a request inside the middleware is shown in the graphic below. Here, MC refers to the memcached servers.

1.3.1 Handling SET Requests

`SET` requests are handled by the `processSetRequest` method. This method iterates through the list of socket channels and sends the request to each of the memcached instances sequentially. After sending to a server, byteBuffer.rewind is called to reset the position of the bytebuffer to zero. This allows us to use the same bytebuffer for each one of the memcached servers and no new copies are created. The responses are read back only after sending to all the servers. We log `timeStartSendToMC` when we start sending the requests. We also log `timeSentToMC` when we are done sending the request to each of the servers.

Once the request is forwarded to the memcached servers, we allocate a new byte buffer and iterate through the socket channels again to read in the responses. We log `timeStartRecvdFromMC` before starting this process. If the response is not `STORED\r\n`, then a copy of the response is added to a list of exceptions and the buffer is cleared before moving to the next memcached instance. `timeFinishedRecvdFromMC` is logged once we finish reading from each of the servers.

After reading all the responses, we check if the list of exceptions is empty or not. If it is not empty, the first response is taken and returned back to the client through the request's socket channel. All exceptions are logged even though only one is sent back to the client.

If the exceptions list is empty, the SETs have succeeded. We reuse the contents of the buffer we use to read back the responses (which will be having `STORED \r \n` in this case) and send it back to the client.

1.3.2 Handling GET requests

Handling GET requests is more complicated because of the number of different ways we need to process them. As a result, we have the following program flow. As soon as we detect that the request is a `GET`, we hand it over to `processGetRequest` method. The `processGetRequest` first checks if the request is a single or a multi-`GET` by counting the number of "spaces" in the request. If there is more than one space in the request, it's a multi-GET. The request is then handed off to one of `processSingleGetRequest` or `processMultiGetRequest` respectively.

i. Handling Single GETs The `processSingleGetRequest` method is straight-forward and easy to understand. It simply forwards the request to one of the memcached servers and reads back the response. The timestamps `timeStartSendToMC` is recorded before sending the request to memcached and `timeSentToMC` is recorded after sending the request to the memcached server.

In order to ensure that each of the memcached servers experience equal amounts of load from the single-GET requests, we use a variable called `whosNext` which keeps track of which memcached server to forward the request to. `whosNext` is incremented (and modulo number of memcached servers) after every request. There is one instance of this variable per worker thread and allows us to use a round-robin scheduling to assign single GET requests to memcached servers.

We use the `isCompleteResponse` method from the utils class to check if we have got the entire response back from the memcached server. We keep reading from the server in a while-loop until we have the entire response. This completeness check is especially important in the case of multi-GETs without sharding where the response can become very large and we may not get the entire response in one single `read` call.

The timestamps `timeStartRecvdFromMC` and `timeRecvdFromMC` are recorded before we start reading and after we finish reading respectively.

Once the complete response is with the middleware, the method then checks if the response starts with `VAL` or `END`. These correspond to the case when we found the key in the memcached server and it has returned back the value and the case when we have a GET miss respectively. Note that the latter case will never happen due to the fact that we pre-populate the memcached servers with all the keys we use. Nevertheless, we handle the GET misses case as well. If the response is neither of these, then it is an error and the entire request and response is logged. After this, the worker thread moves on to process the next request available in the queue.

In case that all sanity checks pass, and we have a valid response, the method simply flips the buffer and forwards the response back to the client. The timestamps `timeStartSentBack` and `timeSentBack` are recorded before starting and after completing the `socketChannel.write` call.

ii. Handling Multi-GETs Handling multi-GETs is a bit tricky due to the multiple sub-cases that arise. Inside the `processMultiGetRequest`, we first check if the `isSharded` flag is set. This flag corresponds to whether we need to shard the multi-GET request or not. We also check if the number of memcached servers is greater than one. If either of the above conditions is not true, then the multi-GET request can be treated as a single-GET request and so the `processSingleGetRequest` is called with the request. Therefore, `processMultiGetRequest` is used if and only if the sharded-mode is enabled and there are more than one memcached servers.

If the above conditions are satisfied, we first split the received request into shards. This is done by `utils.getShards` method which gives us as many shards as there are memcached servers. The method extracts the keys from the multi-GET request and creates the shards from the bottom up in a round-robin fashion by looping across the shards and appending one key at a time until there are no keys left in the multi-GET request. This way, we can guarantee that the sharded requests are of almost equal size. They become slightly unequal when the number of keys is not divisible by the number of memcached servers. In such cases, some of the shards have one key extra. No sharding technique can circumvent this issue because it is mathematically impossible to equally divide the load when the number of keys is not a multiple of number of memcached servers.

Note that in our experiments, we always have situations where the number of keys in a multi-GET request is divisible by the number of memcached servers so we can assume that the shards are always of equal size in practice. As a result, each multi-GET request puts equal load on each of the memcached servers.

Once we have the shards, we send one shard per memcached server and log the `timeStartSendToMC` and `timeSentToMC` before and after we send the requests.

After this, we read back the responses from each of the memcached servers (checking if we have got the complete response) and store them in an array list. We also log `timeStartRecvdFromMC` and `timeRecvdFromMC` before and after we receive all the responses.

We also check if the responses start with "VAL" or "END" to ensure that we don't get any errors. If there are any, we log them separately and move onto the next request in the queue.

The arraylist of responses is sent to `utils.joinShards` which combines the responses into one. It does this by iterating through the responses, stripping off the "END\r \n" at the end of each response and appending the responses together. It finally appends a "END\r \n" at the end of the combined response to make it a valid response to a multi-GET request.

This is then sent back to the client as the response to its request. We log the `timeStartSentBack` and `timeSentBack` timestamps before and after sending the request back.

Once the logic returns back from either `processSetRequest` or `processGetRequest`, we log to disk all the timestamps corresponding to the request that was processed before moving on to process the next request in the queue (if any).

The flow-chart in the next page illustrates the logic flow of handling requests inside the worker thread.

1.4 Request Object

The request object follows from the Object Oriented Programming (OOP) paradigm and provides encapsulation to keep all the information about a particular request in one place. This includes the socket channel corresponding to the connection from which the request came from as well as the byte buffer that contains the actual contents of the request.

Along with the above two, the request object has timestamps (in nanoseconds, obtained using `System.nanoTime`) for each event in the life of a request in the MW. These include

- `timeCreated` When the request was created
- `timeEnqueued` When the request enqueued
- `timeDequeued` When it was dequeued
- `timeStartSendToMC` When we start sending to memcached servers
- `timeSentToMC` When we finish sending
- `timeStartRecvdFromMC` When we start reading back responses
- `timeRecvdFromMC` When we finish reading responses
- `timeStartSentBack` When we start sending back response to client
- `timeSentBack` When we finish sending back response

It also has which type of request it is (SET or GET) in `reqType` attribute.

Finally, the Request object also has a `GetLoggerLine` method for formatting the timestamps into a line to log in the info logs.

1.5 utils

The utils class contains several auxiliary methods that are used to process the contents received in the byteBuffers. We keep these methods separate from the classes themselves so that their functionality can be used across classes and makes debugging and testing these methods easier. This also provides for a cleaner codebase as well. All the methods in this class are static so that you dont need to instantiate an object of this class to use them. Most of these methods receive a bytebuffer as an input and do not create copies of bytebuffer explicitly or implicitly (by creating new String objects). Instead, they obtain a view into the underlying byte array to handle the processing thereby saving memory and reducing performance overheads.

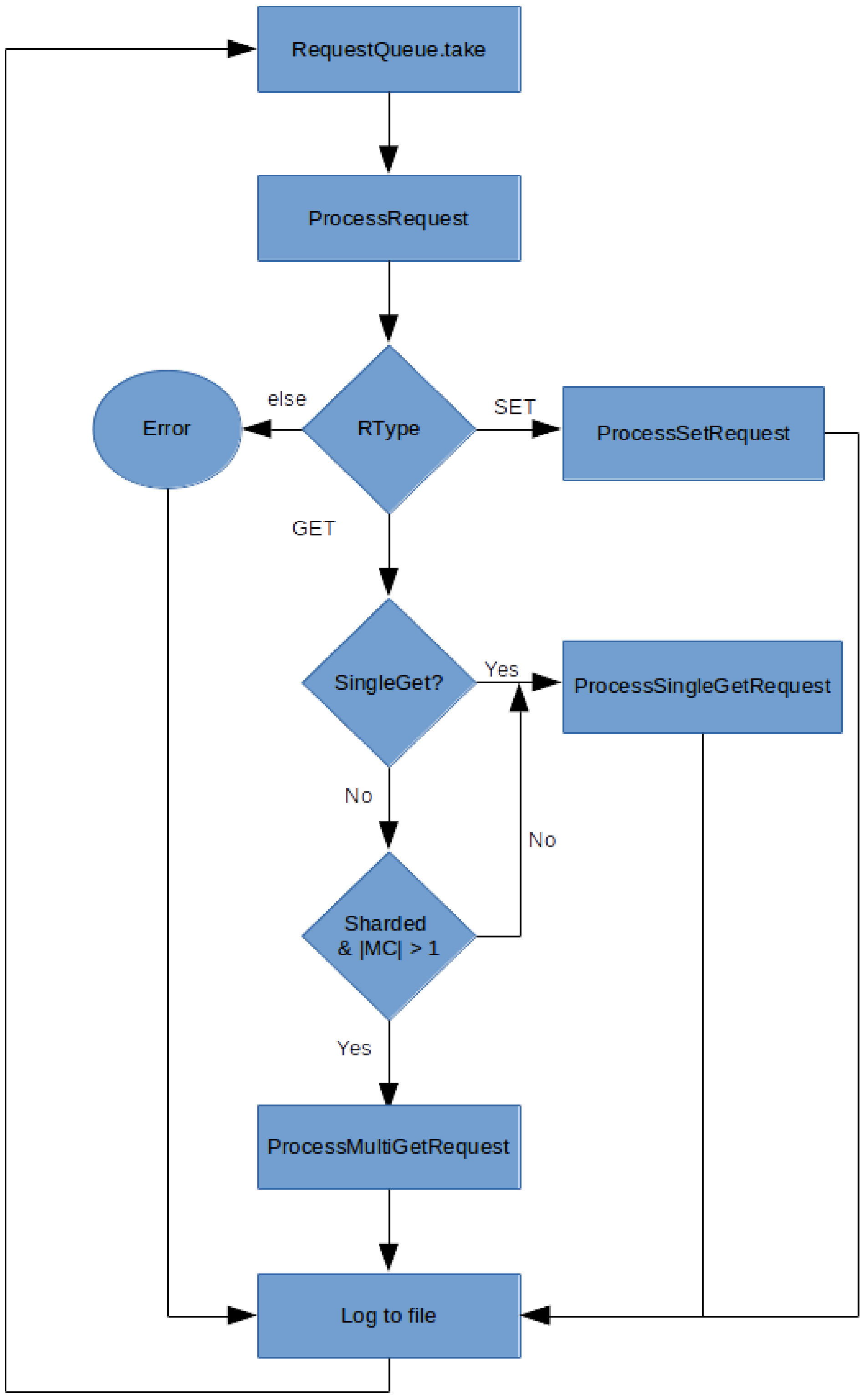
The methods in this class include

- `getRequestType` checks if request is a GET or SET or something else
- `isSingleGet` checks if a GET request is single or multi-GET
- `countNewLines` counts the number of "/r/n" in the byte buffer
- `isCompleteRequest` checks if we have complete request from memtier clients
- `isCompleteResponse` checks if we have complete response from memcached servers
- `getShards` shards a single multi-GET request into multiple smaller multi-GETs when using the middleware in sharded mode
- `joinShards` combines responses for each shard (smaller multi-GET) sent to memcached servers while the middleware is used in sharded mode. joinShards helps combine multiple responses into one which is sent back to the client.

1.6 Logging Infrastructure

Logging plays an important part in the middleware as the logs are important for all the analysis we do downstream. There are multiple log files that are generated by the middleware. All log files that contain errors end with a `.error` suffix and those that information (like that of a request) end with `.info` suffix. Additionally, we log the queue lengths separately in a file with `.length` suffix.

Figure 3: Logic Flow of handling requests in the worker thread



The Net thread and each of the worker threads generate their own log files and have the following naming convention.

`mc-<num_memcached_servers>_workers-<num_worker_threads>_shard-<sharding_flag>_` If the log files are from the frontend, there's a `frontend` appended to the convention above. `id_<worker_thread_id>` is appended if the files are from a worker thread.

An example would be `mc-3_workers-64_shard-false_id-0.info` which corresponds to the configuration when there are three memcached servers, 64 worker threads, sharding disabled and the file is the INFO logs of the first worker thread (id-0). As mentioned earlier, log files are sanity checked before being used in the analysis downstream. This is done using `scripts/raw_mw_sanity_check.py` script. The sanity check involves making sure that all the ERROR files are empty and that none of the INFO files from the workers are empty. These two conditions ensure that the middleware operated nominally during our experiments.

We do similar sanity checks with the memtier generated logs using the `scripts/mt_sanity_check.py` script. Sanity checks here implies checking that the JSON file is parsable, that there are no GET misses and that the response times are not infinity.

We log information about each request (all timestamps and request type mentioned in the section about the Request object), this enables us to have much finer analysis of the middlewares performance and obtain deeper insights about the middlewares performance like if the garbage collector significantly affected performance at some time or not (it did not). We dont write to disk immediately after each request completes but buffer our writes. This way, we save a large amount of unnecessary I/O operations and as a result the middleware performance is not adversely affected. It is also interesting to note that logging every request is not unusual even in production environment. In fact, the Apache webserver logs every single HTTP request it gets.

Finally, we use the `scripts/mw_folder_postprocessor.py` script to aggregate information like throughput and response time across the worker threads for further analysis.

1.7 Load Distribution Among Memcached Servers

Due to the round-robin scheduling we exploit, each worker thread distributes its single-GETs load equally to each memcached server. The SETs are naturally equally distributed because they are forwarded to each of the memcached servers. The only remaining request type is multi-GET and the mechanism by which we equally distribute multi-GET requests among the memcached servers (in both sharded and non-sharded modes) has been discussed in the previous sections. Thus, each worker thread distributes its load equally among the memcached servers, and we can extend this to say that the middleware distributes its load equally among all the memcached servers.

2 Baseline without Middleware

NOTE It is highly recommended to read the appendix A *General Notes on Experimental Setup* to get the common setup which applies across all the experiments.

2.1 One Server

2.1.1 Commands

The following are the commands used to start the memtier instances and memcached instances

memcached instance: `memcached -t 1 -p 6969`

memtier instance:

For Write-Only Workloads: `memtier_benchmark -s <MC_IP> -p 6969 -P memcache_text -t 2 -c <NUM_VC_PER_THREAD> --ratio 1:0 --test-time 80 --json-out-file <LOG_FILE>`

For Read-Only Workloads: `memtier_benchmark -s <MC_IP> -p 6969 -P memcache_text -t 2 -c <NUM_VC_PER_THREAD> --ratio 0:1 --test-time 80 --json-out-file <LOG_FILE>`

The naming conventions used are explained in the relevant sections in the report as well as in the `logs/` directory of the repository. `<MC_IP>` is the private IP of the memcached server (In the 10.0.0.x range)

2.1.2 Experimental Configuration

Experiment Configuration for One Server baseline Experiments

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[5, 15, ..., 145]
Workload	Write-only and Read-only
Multi-Get behaviour	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	3 (1 minute steady state)

2.1.3 Plots

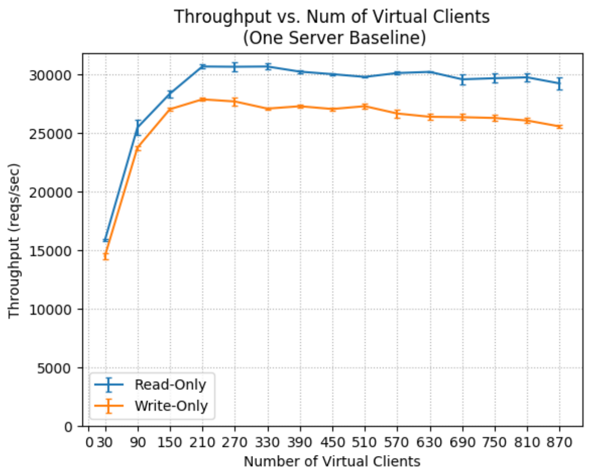


Figure 4: Average Throughput for baseline without middleware,
one memcached server

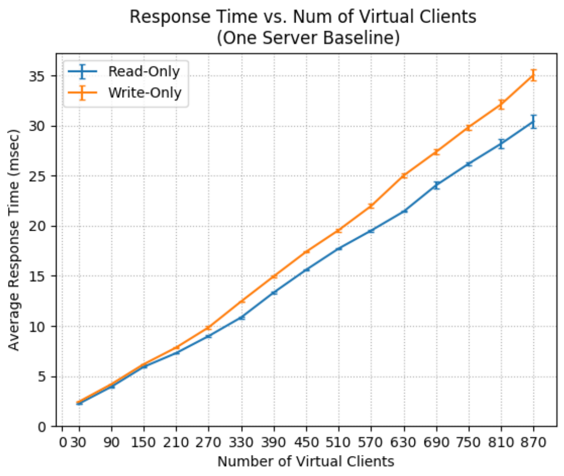


Figure 5: Average Response Time for baseline without middleware,
one memcached server

2.1.4 Explanation

From the throughputs plotted in Figure.4, we can see that for the read-only and write-only loads, the throughput flattens out at around 210 virtual clients. Thus, we can say that the memcached server is under-saturated when the load is less than 210 virtual clients and saturated beyond that. Memtier clients wait for a response before sending the next request, hence the number of requests in the system (memcached + memtier) is bounded. Thus, we can say that the system is closed and as a result, we don't see the oversaturation phase. The errors in measurement are nominal and thus we can conclude that the network, MC servers and MW clients were all stable during our experiments.

From both the plots, we can see that for a given number of virtual clients, the writes have lower throughput and higher response times. From the memcached protocol, we can see that the number of bytes in a SET request are higher than those in a GET request (the SETs have to contain the value to be stored along with the key). This trend is reversed if we consider the response, i.e., responses to GETs are bigger than responses to SETs.

It more instructive to discuss the bottlenecks after we discuss trends with the two server experiments and are able to compare the trends observed in both.

The Interactive law connects the throughput (X) and the response time (X) of a system. This can be written as $R = N/X - Z$. Here, Z is the thinking time for the clients. In most cases, including ours, this is assumed to be zero. Thus, the law simplifies to $R = N/X$. N is the number of jobs in the system, in this case, the number of virtual clients.

We plot the response time vs. throughput plot for the one memcached experiments below. The plot verifies that our observations follow the interactive law (within error bounds).

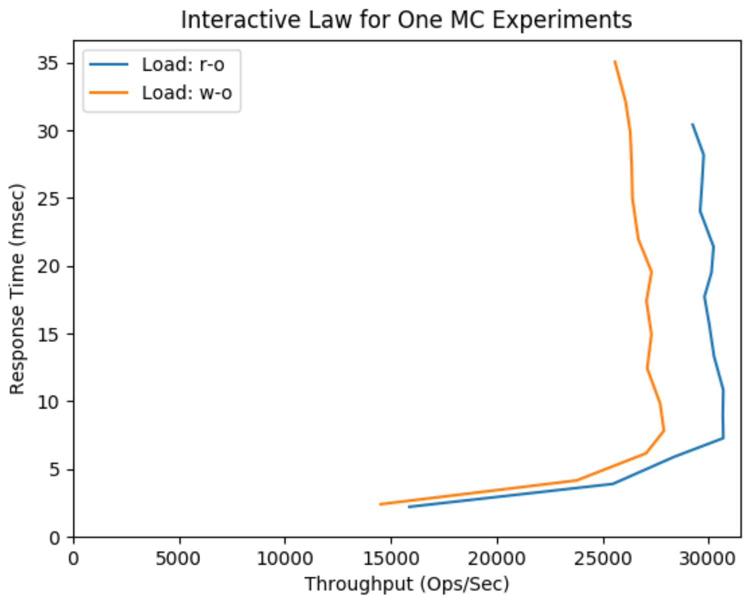


Figure 6: Interactive Law Plot, one memcached server

2.2 Two Servers

2.2.1 Commands

The following are the commands used to start the memtier instances and memcached instances

memcached instance: `memcached -t 1 -p 6969`

memtier instances:

For Write-Only Workloads: `memtier_benchmark -s <MC_IP_1> -p 6969 -P memcache_text -t 1 -c <NUM_VC_PER_THREAD> --ratio 1:0 --test-time 80 --json-out-file <LOG_FILE>`

`memtier_benchmark -s <MC_IP_2> -p 6969 -P memcache_text -t 1 -c <NUM_VC_PER_THREAD> --ratio 1:0 --test-time 80 --json-out-file <LOG_FILE>`

For Read-Only Workloads: `memtier_benchmark -s <MC_IP_1> -p 6969 -P memcache_text -t 1 -c <NUM_VC_PER_THREAD> --ratio 0:1 --test-time 80 --json-out-file <LOG_FILE>`

`memtier_benchmark -s <MC_IP_2> -p 6969 -P memcache_text -t 1 -c <NUM_VC_PER_THREAD> --ratio 0:1 --test-time 80 --json-out-file <LOG_FILE>`

`<MC_IP_1` and `<MC_IP_2` are the private IPs of the two memcached server (In the 10.0.0.x range)

2.2.2 Experimental Configuration

Experiment Configuration for Two Server baseline Experiments

Number of servers	2
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[5, 15, ..., 145]
Workload	Write-only and Read-only
Multi-Get behaviour	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	3 (1 minute steady state)

2.2.3 Plots

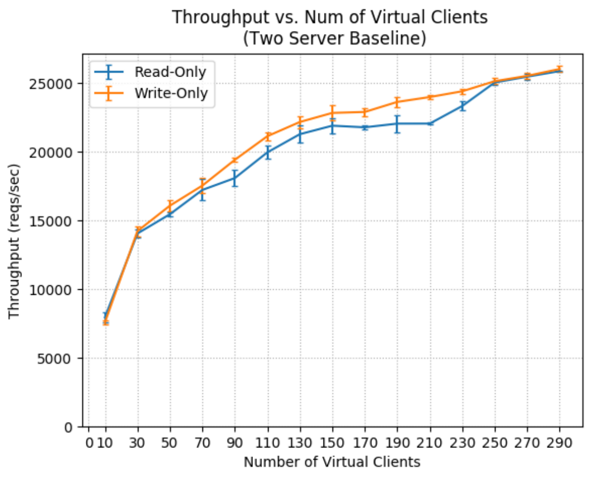


Figure 7: Average Throughput for baseline without middleware,
two memcached servers

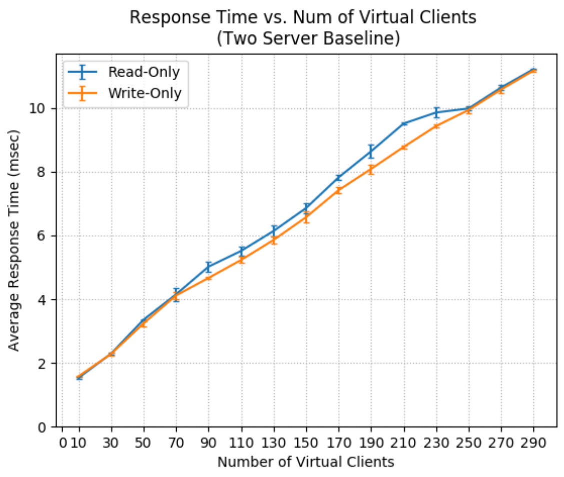


Figure 8: Average Response Time for baseline without middleware,
two memcached servers

2.2.4 Explanation

From the throughputs plotted in Figure. 7, we can see that the throughput is starting to flatten out at the range of 270 to 290 clients.Thus we can say that the memcached server is undersaturated whenever the number of clients is less than 290. Saturation starts from above this number of virtual clients.

When we introduce an extra memcached server, we see that the plots of both throughput and response times for the two kinds of load we use are almost similar (within error bounds, of course). This is in contrast to the one server experiments where clearly saw that the SETs took longer than the GETs. This hints at the fact that memcached could be the bottleneck when there was only one MC server. This hypothesis will be verified in the summary.

Apart from this, two interesting observations can be made. We notice that around 190 to 210 virtual clients, the throughput of reads is noticeably lower than that of writes. The latencies are similarly higher. Above and below this range, the two work loads mirror each other in both throughput and latency. Even after repeating the experiment for multiple times, at different times, this pattern was observed. Thus, this can be attributed to issues with how the virtual machines are networked and noise in the cloud.

The second observation is that for a given number of virtual clients on the X axis, the throughput for two memcached servers is actually lower than for the same number of virtual clients in the one server case. This can be attributed to two factors.

The first is CPU load. In the single server experiment, there were three load generating machines sending requests to a single memcached instance, in the two server server experiment, it is one load generating VM for two memcached instances. Thus, the CPU in this VM has to perform more tasks for the same number of virtual clients. For instance, in read-only experiments with 135 virtual clients per thread, `dstat` reports an average CPU utilization of 5% for the one server experiments and 40% for the two server experiments (these logs are at `logs/2a_one_server_base_line_log` and `logs/2b_two_server_base_line_log`). In other words, the CPU has to perform more work for the same number of virtual clients.

The second is the network latency. Taking a look at the average ping times at `logs/2a_one_server_base_line_log/ping*`) and `logs/2b_two_server_base_line_log/ping*`), we see that some connections can have average ping latencies that are double or triple that of other connections. These factor into reducing the number of requests that are processed as each request takes longer to reach the memcached server. These differences in latencies remained consistent even after allocating and deallocating the VMs multiple times. Nevertheless, the trends observed in the throughput is informative.

2.3 Summary

The experiments done in previous sections can be summarized as below. Here, VC refers to the total number of virtual clients, summed across all memtier threads and load generating VMs.

Maximum throughput of different VMs.

	Read-only workload	Write-only workload	Configuration gives max. throughput
One memcached server	30698	27894	210 VC (both cases)
One load generating VM	25881	26035	290 VC (both cases)

We have already discussed why the observed throughputs are different in the two situations. With all the information collected from the above two sections, we can now discuss the bottleneck in this setup.

In the section with one memcached server, we saw that the SETs took longer than the GETs and in the section with two memcached servers, we see that both SETs and GETs take about similar times. Additionally, for a given number of virtual clients, GETs in both one server and two server setups took about similar amount of time. Thus, adding an extra server helped in reducing the response time for SETs but did not reduce the response times for the GETs.

This shows that memcached server is the bottleneck in single server experiments. To be more specific, the incoming network buffers in the memcached instances are the bottleneck. This can be explained by the fact that GET requests are smaller than SETs and as a result, occupy smaller space in the incoming network buffers at the memcached server. By doubling the number of servers, we are halving the load on each of the servers and effectively doubling the buffer size between the two servers. This means that more SETs could be read into the buffer and processed. This is a plausible reason for the trends we observe in the two previous sections.

It is also interesting to note why we don't see such buffer size issues with GETs because a response to a GET is larger than response to SETs. This could be explained by the fact that we run memcached in a single threaded mode (with flag `-t 1`) and so will process requests one by one, consequently, not putting too much load on the outgoing network buffers when compared to the load on the incoming network buffer due to multiple clients sending requests at the same time.

As already discussed in the previous section, the memtier clients form the bottleneck for the two server experiments.

Some of the takeaway messages are that we should keep an eye out for the difference in latencies across different connections in our setup and consider how they may affect our observations. Another message is to consider the lower level information like kernel buffer sizes because the memcached servers run on A1 VMs which have only 1 CPU core and are the least powerful of all the VMs we use, thus the limits to their capabilities could be a result of them being under-powered

3 Baseline with Middleware

3.1 One Middleware

3.1.1 Commands

The following are commands used to start memtier and middleware instances

memtier instance:

For Write-Only Loads: `memtier_benchmark -s <MW_IP> -p 6969 -P memcache_text -t 2 -c <NUM_VC_PER_THREAD> --ratio 1:0 -d 1024 --test-time 80 --json-out-file <LOG_FILE>`

For Read-Only Loads: `memtier_benchmark -s <MW_IP> -p 6969 -P memcache_text -t 2 -c <NUM_VC_PER_THREAD> --ratio 0:1 -d 1024 --test-time 80 --json-out-file <LOG_FILE>`

memcached instance: `memcached -t 1 -p 6969`

middleware instance: `java -jar /home/sgokula/middleware-sgokula.jar -l <MW_IP> -p 6969 -t <NUM_WORKER_THREAD> -s false -m <MC_IP>:6969`

Here, `<MW_IP>` refers to the private IP of the middleware VM (A A4 Class VM). The log file naming convention can be found in the `logs/README` file of the repository.

3.1.2 Experimental Configuration

Experimental Configuration for One Middleware baseline Experiments

Number of servers	1
Number of client machines	1
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[2, 4, 8, 16, 32, 64, 128]
Workload	Write-only and Read-only
Multi-Get behaviour	N/A
Multi-Get size	N/A
Number of middlewares	1
Worker threads per middleware	[8, 16, 32, 64]
Repetitions	3 (1 minute steady state)

3.1.3 Plots

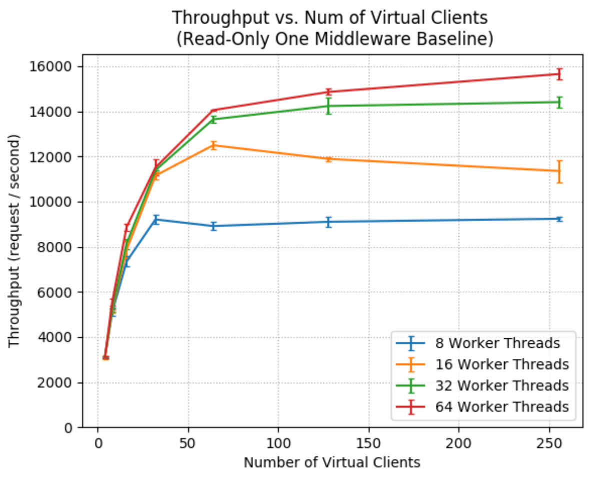


Figure 9: Baseline with One Middleware, Average Throughput (Read-Only Loads)

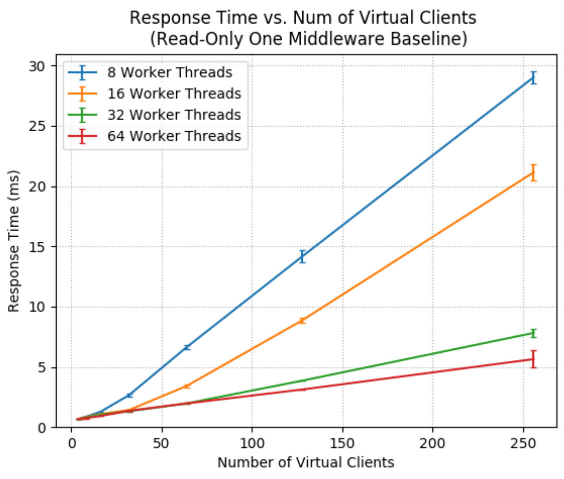


Figure 10: Baseline with One Middleware, Average Response Time (Read-Only Loads)

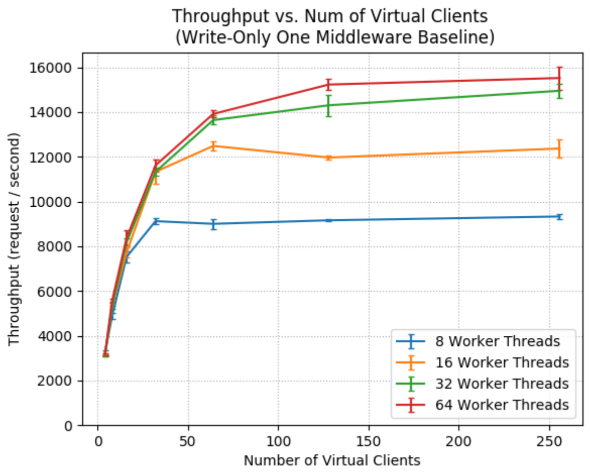


Figure 11: Baseline with One Middleware, Average Throughput (Write-Only Loads)

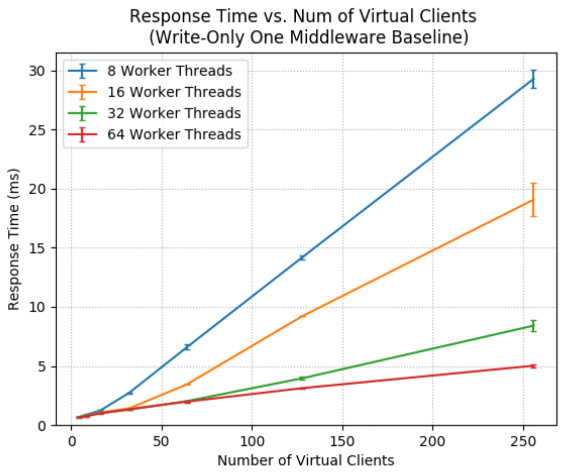


Figure 12: Baseline with One Middleware, Average Response Time (Write-Only Loads)

3.1.4 Explanation

From figures above, we can see that the trends and numbers for GETs and SETs are similar. This holds even though the middleware has to replicate SETs to all the memcached servers because there is only memcached server in our setup. Thus, we are seeing the same behaviour between the two request types just like we saw in the previous section.

From figures 8 and 10, we see that the middleware reaches saturation at 32, 64, 128 and 256 virtual clients when the there are 8, 16, 32 and 64 worker threads respectively. For 8 and 16 worker threads, we also observe the oversaturation phase when there are more than 32 and 64 clients as we observe the throughput dropping for these situations.

To understand why the throughput doesn't increase as we further increase the number of clients in each of the above saturation cases, let's take a look at the queuing time (time spent by a request in the queue) for each of these cases.

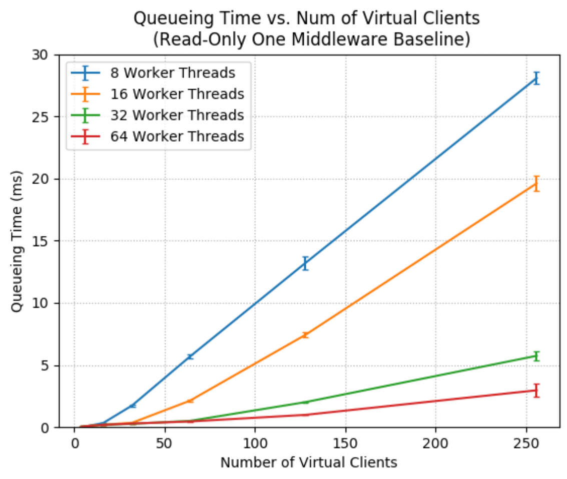


Figure 13: Baseline with One Middleware, Average Queuing Time (Read-Only Loads)

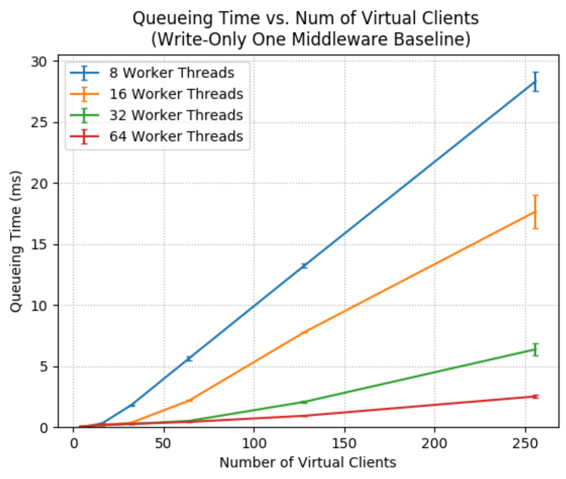


Figure 14: Baseline with One Middleware, Average Queuing Time (Write-Only Loads)

We see that beyond the saturation point we described above, the plots for response time and queuing time are almost similar. This shows that beyond the saturation point, the requests spend significant amount of time in the queue waiting to be processed than actually being serviced. The middleware can process only `|NUM_WORKER_THREAD|` at any given point so any additional request will have to wait for one of the worker threads to become free before it can be processed. These waiting times are marginal when compared to the processing time whenever `|NUM_VIRTUAL_CLIENTS|< 4*|NUM_WORKER_THREAD|` (empirically from the plots) and increase rapidly when more clients are further added, as seen in the plots above. Thus, as we increase clients further, the throughput does not increase because the worker threads are already busy most of the time. The requests, instead spend more time in the queue increasing the response time.

From these arguments, we can conclude that of memtier, middleware and memcached servers, the middleware is the bottleneck because we have seen throughputs in the 20000 - 30000 ranges without the middleware in the previous section. Inside the middleware, the worker threads are the most utilized (as they are busy most of the time) and as a result are the bottleneck in these experiments.

3.2 Two Middlewares

3.2.1 Commands

The following are the commands used to start the memtier, middleware and memcached instances

memcached instance: `memcached -t 1 -p 6969`

middleware instances:

```
java -jar /home/sgokula/middleware-sgokula.jar -l <MW_IP_1> -p 6969 -t <NUM_WORKER_THREAD> -s false -m <MC_IP>:6969
java -jar /home/sgokula/middleware-sgokula.jar -l <MW_IP_2> -p 6969 -t <NUM_WORKER_THREAD> -s false -m <MC_IP>:6969
```

memtier instance:

For Write-Only Loads:

```
memtier_benchmark -s <MW_IP_1> -p 6969 -P memcache_text -t 1 -c <NUM_VC_PER_THREAD> --ratio 1:0 -d 1024
--test-time 80 --json-out-file <LOG_FILE>
memtier_benchmark -s <MW_IP_2> -p 6969 -P memcache_text -t 1 -c <NUM_VC_PER_THREAD> --ratio 1:0 -d 1024
--test-time 80 --json-out-file <LOG_FILE>
```

For Read-Only Loads:

```
memtier_benchmark -s <MW_IP_1> -p 6969 -P memcache_text -t 1 -c <NUM_VC_PER_THREAD> --ratio 0:1 -d 1024
--test-time 80 --json-out-file <LOG_FILE>
memtier_benchmark -s <MW_IP_2> -p 6969 -P memcache_text -t 1 -c <NUM_VC_PER_THREAD> --ratio 0:1 -d 1024
--test-time 80 --json-out-file <LOG_FILE>
```

3.2.2 Experimental Configuration

Experimental Configuration for Two Middleware baseline Experiments

Number of servers	1
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[2, 4, 8, 16, 32, 64, 128]
Workload	Write-only and Read-only
Multi-Get behaviour	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8, 16, 32, 64]
Repetitions	3 (1 minute steady state)

3.2.3 Plots

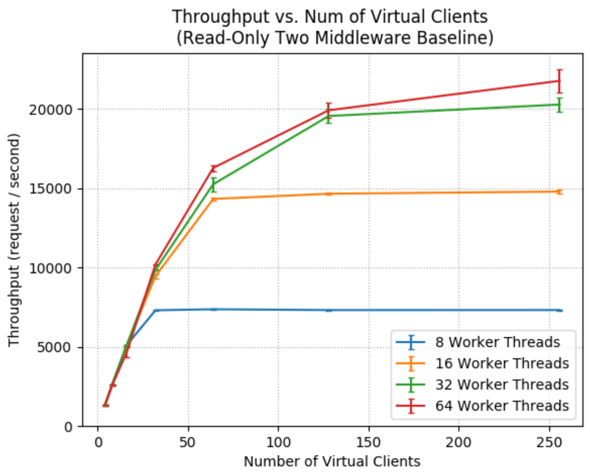


Figure 15: Baseline with Two Middlewares, Average Throughput (Read-Only Loads)

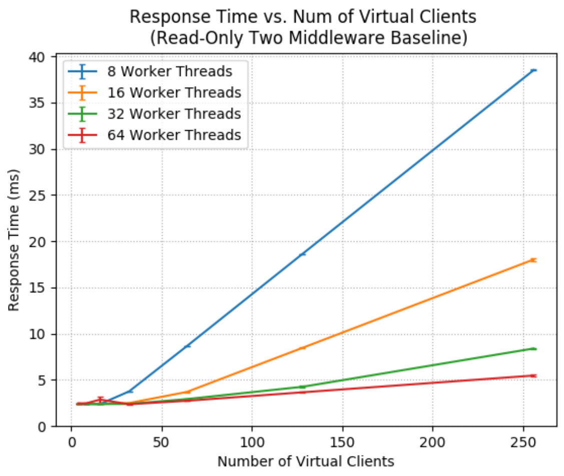


Figure 16: Baseline with Two Middlewares, Average Response Time (Read-Only Loads)

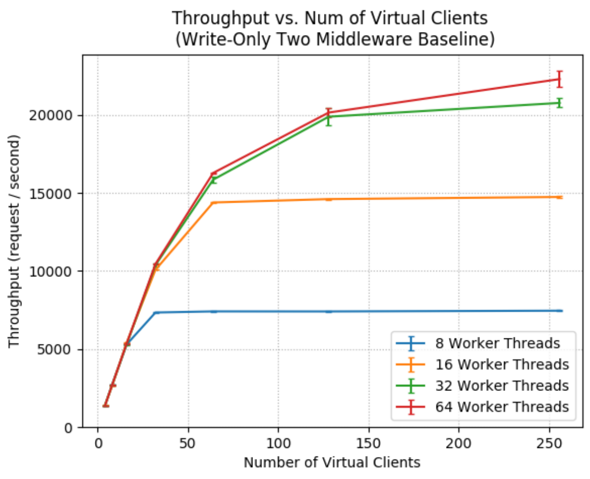


Figure 17: Baseline with Two Middlewares, Average Throughput (Write-Only Loads)

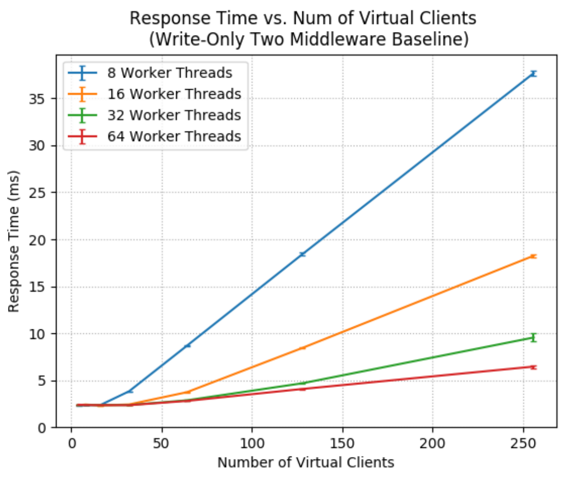


Figure 18: Baseline with Two Middlewares, Average Response Time (Write-Only Loads)

3.2.4 Explanation

Just like the section with one middleware, we see that the trends for SETs and GETs mirror each other. This shows that, even in the two middleware setup, the middleware is able to handle both types of requests in a similar way. Interestingly, the middleware saturates at the same number of virtual clients as we see in the one middleware case i.e., the middleware saturates at 32, 64, 128 and 256 clients when there are 8, 16, 32 and 64 worker threads per middleware respectively. However, the peak throughput achieved in each of these cases is higher than observed with only one middleware. The subtlety here is that 8, 16, 32 and 64 are the worker threads per middleware and in fact there are 16, 32, 64 and 128 in these cases when combining both the middlewares. Thus, we see that the individual middlewares behave just like a single middleware behaved in the previous section. Combining the two middlewares simply increased the throughput while maintaining the trends observed before. Thus, we can use the same arguments as before about the bottlenecks and utilizations of the different components of the middlewares.

The queuing time plots are shown below.

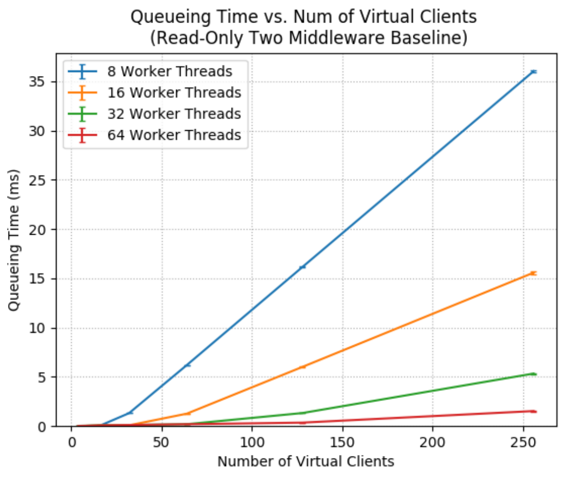


Figure 19: Baseline with Two Middleware, Average Queuing Time (Read-Only Loads)

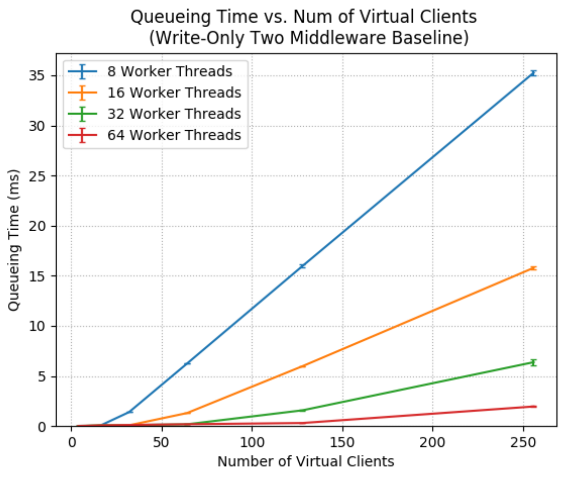


Figure 20: Baseline with Two Middleware, Average Queuing Time (Write-Only Loads)

To ensure that the load generating VM is not the bottleneck, we can check the `dstat` logs, found at `logs/3b_two_mw_base_line/mt_log`. We see that the CPU utilization is barely 5% at the saturation points, this shows that that the load generating VM is not being stressed at the saturation point. Thus, we can conclude that, like before, the middleware is the bottleneck. More specifically, the worker threads of the middleware is the bottleneck and increasing them by doubling the number of middlewares (keeping the number of threads per middleware constant) increases the throughput.

3.3 Summary

The experiments in the above sections can be summarized as below, these numbers are averaged across repetitions to reduce the effect of random noise in the system.

Maximum throughput for one middleware.

	Throughput	Response time(msec)	Average time in queue(msec)	Miss rate
Reads: Measured on middleware	15652.98	5.64	2.97	0.00
Reads: Measured on clients	10112.42	19.3	n/a	0.00
Writes: Measured on middleware	15521.47	5.02	2.52	n/a
Writes: Measured on clients	13580.12	18.86	n/a	n/a

Maximum throughput for two middlewares.

	Throughput	Response time(msec)	Average time in queue(msec)	Miss rate
Reads: Measured on middleware	21758.16	5.47	1.53	0.00
Reads: Measured on clients	19037.3	13.48	n/a	0.00
Writes: Measured on middleware	22291.02	6.45	1.96	n/a
Writes: Measured on clients	19502.47	13.13	n/a	n/a

From the above two tables we see that the response times as measured on the clients is higher than that observed on the middleware. This is because the response time at clients involve the network latency between the clients and middleware thus adding to the value. Also, we see that the throughput observed at clients is lower than that observed in middleware. This seemingly contradictory observation does not mean that there is something odd going on. This is merely because of the fact that to compute throughput, memtier considers the entire 80 seconds we run experiments. By doing so, it includes the warm-up and cool-downs phases where there are fewer requests per second. While reporting throughput for the middleware, we remove the first 5 and last 5 seconds as warm-up and cool-down phase. This explains the observed disparity. Moreover, multiplying the throughput at client by 80 and multiplying the throughput at middleware by 70 give similar numbers as expected.

We also see that, inside the middleware, the requests spend more than 30% of the time waiting in queue, clearly indicating that the worker threads are the bottleneck. Further, throughput for both the workloads in the two middleware setup is more than the corresponding values in one middleware setup further strengthening the claim that the middleware is the bottleneck as adding an extra middleware helped increase the throughput.

4 Throughput for Writes

4.1 Full System

4.1.1 Commands

The following are the commands used to start the memtier, middleware and memcached instances

memtier instance (on each load generating VM):

```
memtier_benchmark -s <MW_IP_1> -p 6969 -P memcache_text -t 1 -c <NUM_VIRTUAL_CLIENTS> -d 1024 --ratio 1:0
--test-time 80 --json-out-file <LOG_FILE>
memtier_benchmark -s <MW_IP_2> -p 6969 -P memcache_text -t 1 -c <NUM_VIRTUAL_CLIENTS> -d 1024 --ratio 1:0
--test-time 80 --json-out-file <LOG_FILE>
```

memcached instances: `memcached -t 1 -p 6969`

middleware instances:

```
java -jar middleware-sgokula.jar -l <MW_IP_1> -p 6969 -t <NUM_MW_THREAD> -s false
-m <MC_IP_1>:6969 <MC_IP_2>:6969 <MC_IP_3>:6969
java -jar middleware-sgokula.jar -l <MW_IP_2> -p 6969 -t <NUM_MW_THREAD> -s false
-m <MC_IP_1>:6969 <MC_IP_2>:6969 <MC_IP_3>:6969
```

The naming convention for the log files are mentioned in the appendix and in the `logs/README` file in the repository.

4.1.2 Experimental Configuration

Experimental Configuration for ”Throughput for Writes” Experiment

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1, 6, 12, 18, 24, 30, 32]
Workload	Write-only
Multi-Get behaviour	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8, 16, 32, 64]
Repetitions	3 (1 minute steady state)

4.1.3 Plots

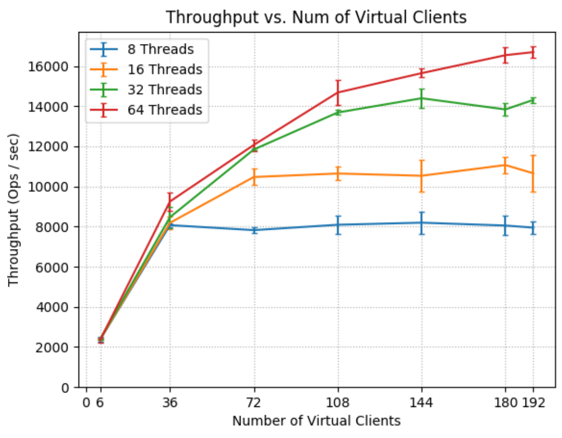


Figure 21: Average Throughput for Sets Throughput Experiments

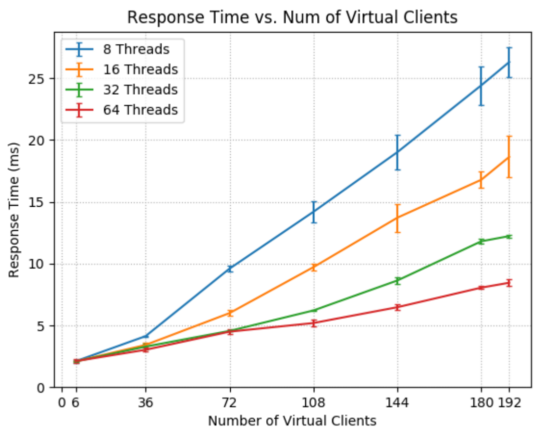


Figure 22: Average Response Time for Sets Throughput Experiments

4.1.4 Explanation

From the plots above, we see the middleware getting saturated at 36, 72, 108 and 192 virtual clients when there are 8, 16, 32 and 64 worker threads per middleware respectively. We also see some signs of over-saturation and system instability to the right of these points as the throughput drops and there is higher variance in the observed throughput, indicating that the system is exhibiting erratic behaviour. We also see that the throughput at saturation increases when we increase the number of threads indicating that the middlewares do not become a bottleneck because of lack of compute or memory resources. They become a bottleneck due to a lack of parallelism brought by having a larger number of worker threads.

To further back up our claim about the saturation point, let's look at the plots of queuing time and average queue length for these situations.

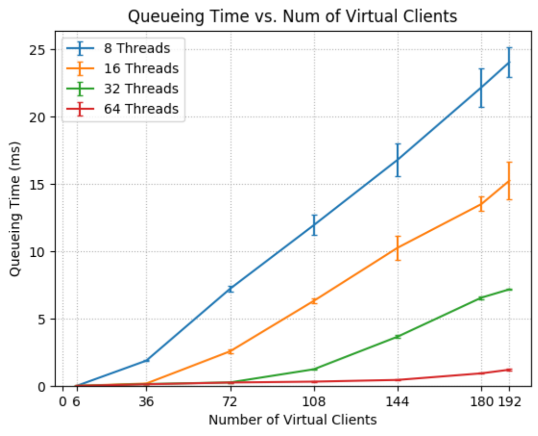


Figure 23: Average Queuing Time for Sets Throughput Experiments

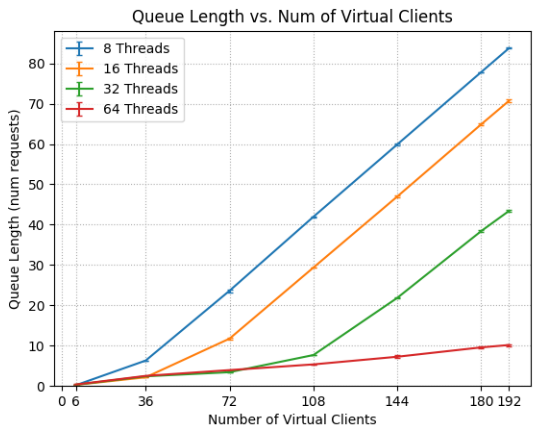


Figure 24: Average Queue Length for Sets Throughput Experiments

we see that the average queue length and queuing time explode when we increase the number of cents above the saturation point. This means that as we increase clients further, the worker threads and hence the middleware itself is unable to process the request fast enough and the incoming requests are put in a queue. Since the queue gets logger and longer, the queuing get bigger and bigger as well.

4.2 Summary

Based on the experiments above, we can extract the following information about the maximum write throughput achieved for different setups.

Maximum throughput for the full system

	WT=8	WT=16	WT=32	WT=64
Throughput (Middleware)	8072	10470.12	13682.36	16691.45
Throughput (Derived from MW response time)	8720	12020	17419.35	22711.42
Throughput (Client)	7062.83	9175.8	11998.7	14624.21
Average time in queue	1.89	2.58	1.25	1.22
Average length of queue	6.3	11.74	7.67	10.16
Average time waiting for memcached	2.21	3.37	4.92	7.2
Response Time at Middleware	4.128	5.99	6.2	8.45
Total Number of clients at max. TPS	36	72	108	192

Here too, like in the middleware baseline experiments, the throughput measured at the clients is consistently lower than that measured on the middleware. The same reasoning used in the previous sections apply here as well. Across the columns, as we increase the number of worker threads, the middleware is able to make better use of the compute resources, increasing the throughput significantly for marginal increase in response time. This again shows that the lack of parallelism is the bottleneck.

We also see that the average time waiting for memcached increases. This is because the middleware replicates SETs across all memcached servers. As a result, as you increase the number of worker threads, the load on a single memcached server increases. Since these are under-powered, A1 VMs, they become the bottleneck as you increase the number of worker threads further.

In fact, the start of this trend is already visible at T=64 worker threads, where requests spend a large amount of time waiting for the memcached server's response. In these cases, further increasing the number of worker threads will not help.

In summary, at lower number of worker threads, the number of worker threads in the middleware is the bottleneck. As you increase the number of worker threads beyond 64, the memcached servers become the bottleneck due to the middleware replicating SETs across all the servers, overloading them.

It is also interesting to note that the interactive law consistently predicts higher throughput than what we measure at the middleware. This is because while computing the throughput from the interactive law, we assume that the think time (Z) is zero to simplify our calculations. This is not the case when it comes to the middleware because there are delays introduced by the network connections between the MT instances and the MW instances. Thus, this explains why predict higher throughput. The thinking time (Z) can be approximated as the difference between the response time measured at the clients and at the middleware. We shall incorporate this into our analysis and get a better estimate for throughput in the section on network of queues.

5 Gets and Multi-gets

5.1 Sharded Case

5.1.1 Commands

The following are the commands used to start the memtier, middleware and memcached instances memtier instance (for each VM)

```
memtier_benchmark -s <MC_IP_1> -p 6969 -P memcache_text -t 1 -c 2 --ratio 1:<MULTI_GET_KEY> --multi-key-get=<MULTI_GET_KEY>
--test-time 80 --key-maximum 10000 -d 1024 --json-out-file <LOG_FILE>
memtier_benchmark -s <MC_IP_2> -p 6969 -P memcache_text -t 1 -c 2 --ratio 1:<MULTI_GET_KEY> --multi-key-get=<MULTI_GET_KEY>
--test-time 80 --key-maximum 10000 -d 1024 --json-out-file <LOG_FILE>
```

memcached instance: `memcached -t 1 -p 6969`

middleware instances:

```
java -jar middleware-sgokula.jar -l <MW_IP_1> -p 6969 -t <NUM_WORKER_THREAD> -s True
-m <MC_IP_1>:6969 <MC_IP_2>:6969 <MC_IP_3>:6969
java -jar middleware-sgokula.jar -l <MW_IP_2> -p 6969 -t <NUM_WORKER_THREAD> -s True
-m <MC_IP_1>:6969 <MC_IP_2>:6969 <MC_IP_3>:6969
```

Here, `<MC_IP_1>` and `<MC_IP_2>` refer to the private IPs of the two middlewares. `<MULTI_GET_KEY>` refers to the number of keys in a multi-GET request. This could be 1, 3, 6 or 9. `<NUM_WORKER_THREAD>` refers to the number of worker threads per middleware.

5.1.2 Experimental Configuration

Experimental Configuration for Sharded Gets Experiments

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	2
Workload	memtier-default (with `--ratio 1:Multi-GET-size`)
Multi-Get behaviour	Sharded
Multi-Get size	[1, 3, 6, 9]
Number of middlewares	2
Worker threads per middleware	max. throughput config.(64 Worker Threads)
Repetitions	3 (1 minute steady state)

5.1.3 Plots

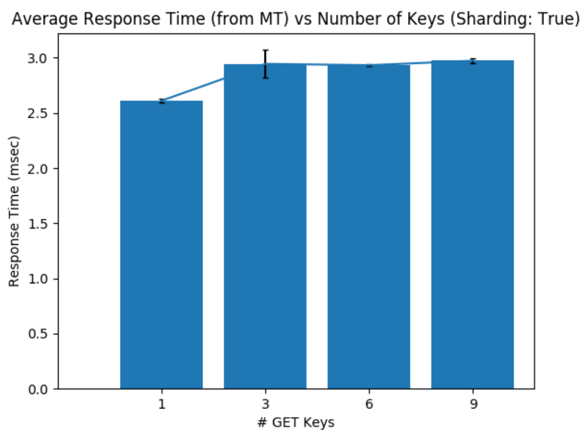


Figure 25: Average Response Time at Client (Sharded Mode)

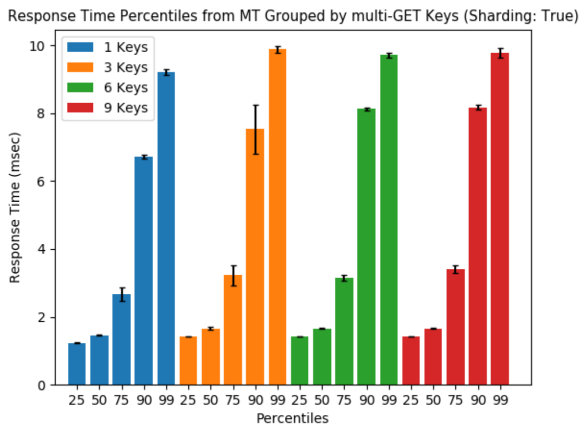


Figure 26: Response Time at client Percentiles Group by num keys (Sharded Mode)

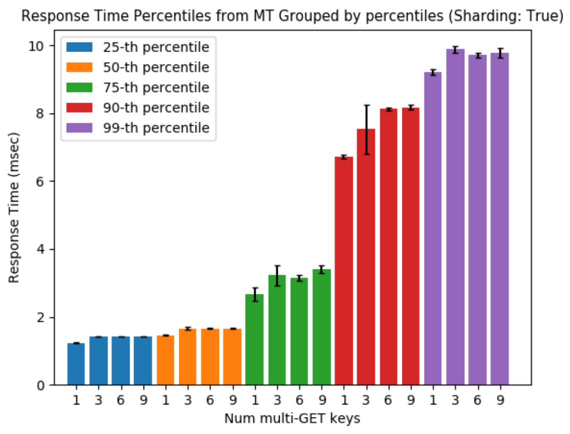


Figure 27: Response Time at client Percentiles Group by percentile (Sharded Mode)

5.1.4 Explanation

In the sharded case, the middleware breaks down the multi-GET into individual keys and assigns keys to multiple smaller multi-GETs in a round robin fashion until all keys have been assigned.

These are sent to the memcached servers one after the other, reading the responses only after sending all the requests. The responses are combined at the middleware and sent back to the client. Due to the `-ratio 1:<MULTI_GET_KEY>` flag. Our workloads consist entirely of equal number of SETs and multi-GETs with `<MULTI_GET_KEY>` keys. From the table above, we can see that this can take values 1, 3, 6, 9 only. When the value is 1 ,the mult-gets are treated like single GETs and are assigned to a single server in a round-robin fashion. When it is higher, each of the 3 memcached server gets a multi-GET request with exactly `<MULTI_GET_KEY> / 3` key. This implies that thanks to our sharding policy, each of the memcached server is equally loaded in each of our sharded experiments.

This further explains the average response time plot. The response times for 3, 6 and 9 keys are almost similar are higher than response time for 1 key. This is because of the overheads of sharding, sending to and reading from multiple servers (3x the network I/O) as well as combining the response. All these combine to give us a higher response time. Since the response time doesn't change much inbetweeen 3, 6 and 9 keys. We can conclude that in the sharded case, the actual size of data being written into the network doesn't matter as much as the number of times we write into it.

This allows us to conclude that the sharding process is the bottleneck in our middleware. More specifically, the higher number of network I/Os created by the sharding process is the limiting factor to the throughput and response time we observer in our middleware.

Additionally, we see the exact trends with percentiles as well. From figure 27, we see that even the 25, 50, 75, 90 and 99 percentiles of response time for the multi-get with 3, 6 and 9 keys are similar (and greater than that for 1 key), further providing evidence to the statement above.

Finally, from figure 26, we see that the response times at different percentiles for a given number of keys increases in an exponential fashion as we increase the percentile to 99. This suggests that the response times are exponentially distributed and might be similar to a long-tailed distribution. We see this trend across all the different number keys we conduct experiments on.

5.2 Non-sharded Case

5.2.1 Commands

The following are the commands used to start the memtier, middleware and memcached instances

memtier instance (for each VM)

```
memtier_benchmark -s <MC_IP_1> -p 6969 -P memcache_text -t 1 -c 2 --ratio 1:<MULTI_GET_KEY> --multi-key-get=<MULTI_GET_KEY>
--test-time 80 --key-maximum 10000 -d 1024 --json-out-file <LOG_FILE>
memtier_benchmark -s <MC_IP_2> -p 6969 -P memcache_text -t 1 -c 2 --ratio 1:<MULTI_GET_KEY> --multi-key-get=<MULTI_GET_KEY>
--test-time 80 --key-maximum 10000 -d 1024 --json-out-file <LOG_FILE>
```

memcached instance: `memcached -t 1 -p 6969`

middleware instances:

```
java -jar middleware-sgokula.jar -l <MW_IP_1> -p 6969 -t <NUM_WORKER_THREAD> -s False
-m <MC_IP_1>:6969 <MC_IP_2>:6969 <MC_IP_3>:6969
java -jar middleware-sgokula.jar -l <MW_IP_2> -p 6969 -t <NUM_WORKER_THREAD> -s False
-m <MC_IP_1>:6969 <MC_IP_2>:6969 <MC_IP_3>:6969
```

Here, `<MC_IP_1>` and `<MC_IP_2>` refer to the private IPs of the two middlewares. `<MULTI_GET_KEY>` refers to the number of keys in a multi-GET request. This could be 1, 3, 6 or 9. `<NUM_WORKER_THREAD>` refers to the number of worker threads per middleware.

5.2.2 Experimental Configuration

Experimental Configuration for Non-Sharded Gets Experiments

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	2
Workload	memtier-default (with `--ratio 1:Multi-GET-size`)
Multi-Get behaviour	Non-Sharded
Multi-Get size	[1, 3, 6, 9]
Number of middlewares	2
Worker threads per middleware	max. throughput config. (64 Worker Threads)
Repetitions	3 (1 minute steady state)

5.2.3 Plots

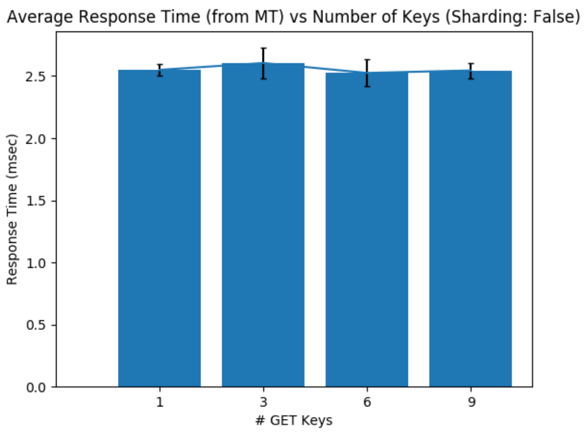


Figure 28: Average Response Time at Client (Non-Sharded Mode)

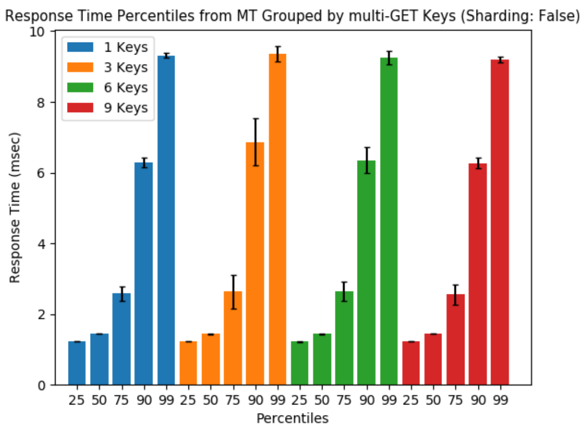


Figure 29: Response Time at client Percentiles Group by num keys (Non-Sharded Mode)

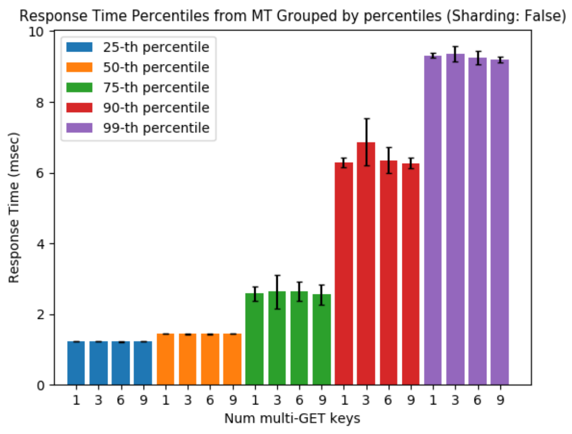


Figure 30: Response Time at client Percentiles Group by percentile (Non-Sharded Mode)

5.2.4 Explanation

Unlike the sharded multi-gets, from figure 28, we see that the average response time is similar across all the number of keys we test on. This further provides evidence to the argument that the number of network I/O is more important than the amount of data sent during a single network I/O operation. This trend is visible in Figure 30, response time percentiles, grouped by percentiles as well. The rest of the argument is the same as the one from the sharded case.

We further see that the response times are lower when compared to the sharded case. This can be attributed to the absence of computations to shard the multi-GET request and join the responses from multiple servers, i.e., the response time is lower because we don't have the overheads of sharding or joining the responses.

5.3 Histogram

We choose bin sizes of 0.1 milliseconds and as many bins as to include as many of the requests in the plot as possible. Note that all of the following plots are for the case when there are 6 keys in our mulit-GET requests and we plot only the GETs at the middleware and memtier. The discussions about the histogram are included in the summary section.

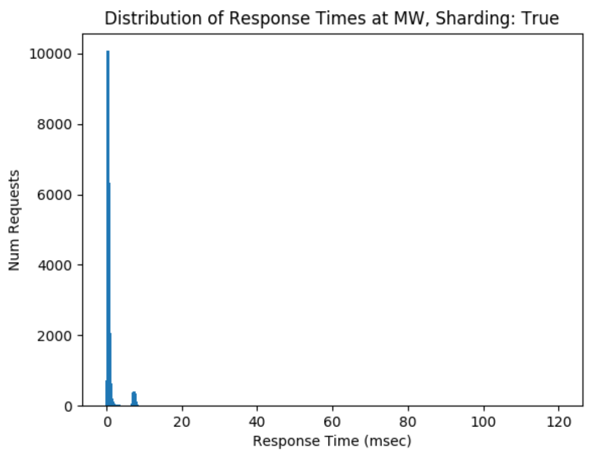


Figure 31: Response Times at Middleware, Sharding: True

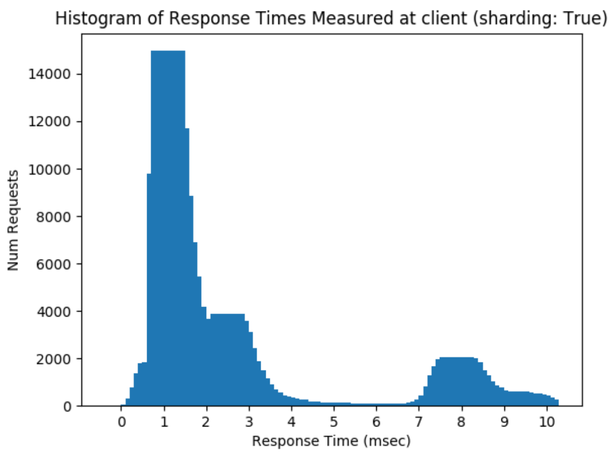


Figure 32: Response Times at Client, Sharding: True

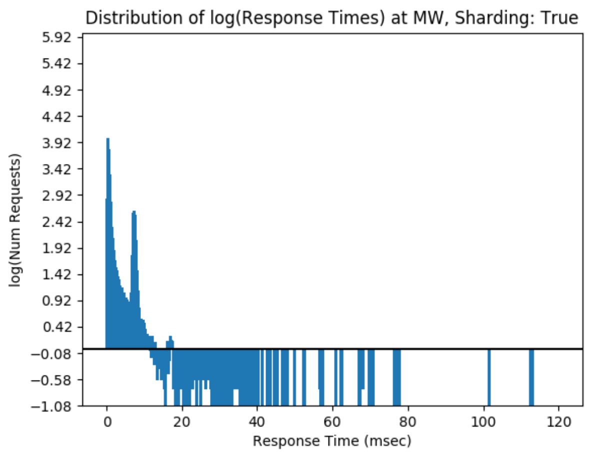


Figure 33: log(Response Times) at Middleware, Sharding: True

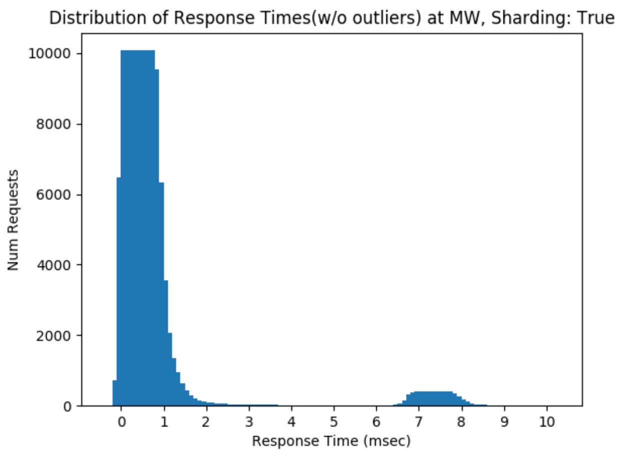


Figure 34: Response Times at Middleware w/o Outliers, Sharding: True

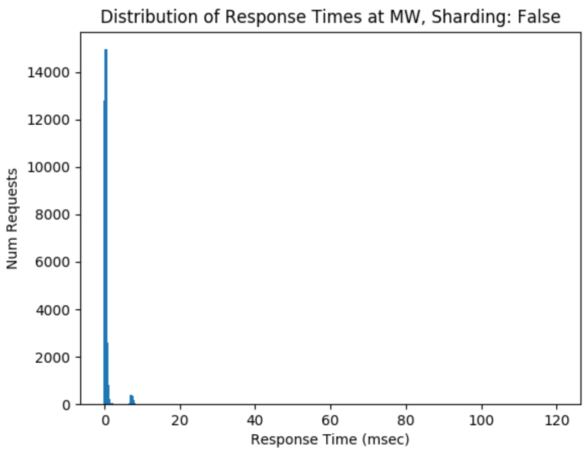


Figure 35: Response Times at Middleware, Sharding: False

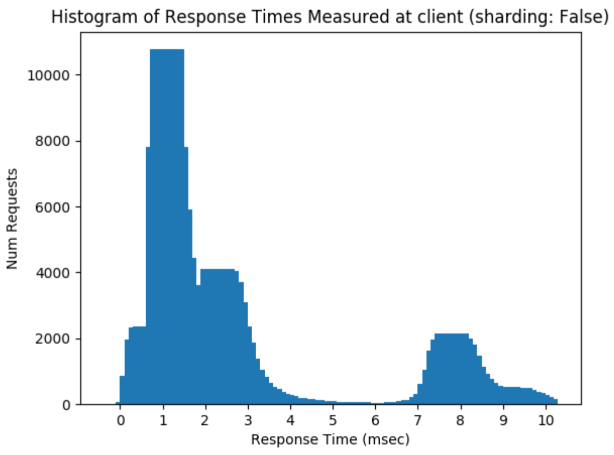


Figure 36: Response Times at Client, Sharding: False

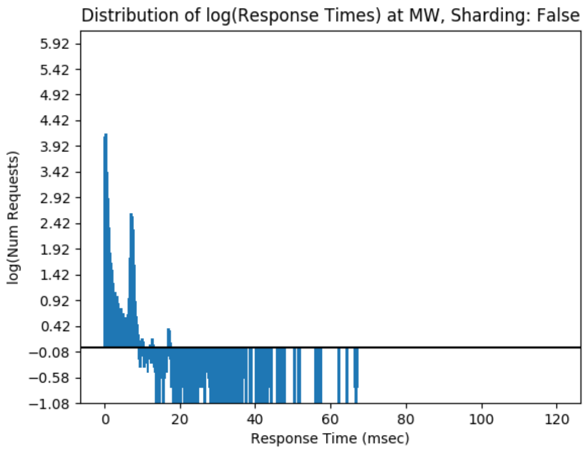


Figure 37: log(Response Times) at Middleware, Sharding: False

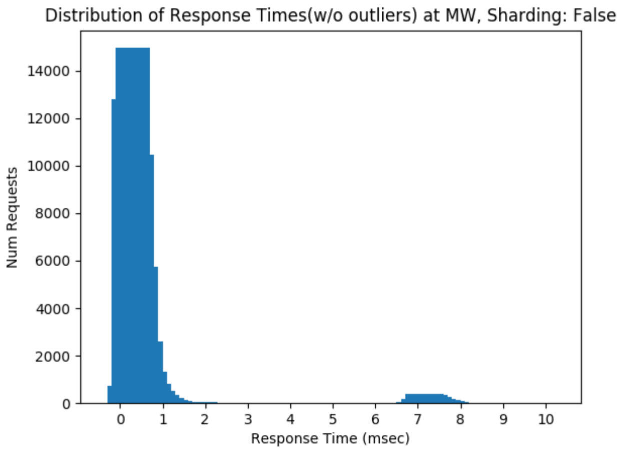


Figure 38: Response Times at Middleware w/o Outliers, Sharding: False

5.4 Summary

From the histograms in figures 31 and 35, we notice that they are extremely sharp and concentrated at a very small area. To check if there are any outlier, we plot the histograms in a log scale in figures 33 and 37 and these reveal that there are indeed small numbers of requests (¡ 10) in the higher bins. These could be due to many reasons like sudden increase in network latencies, garbage collector running, stragglers in the servers which are taking a long time to process .etc. These outliers explain the exponential behaviour in the percentile plots we saw before. We then remove the outliers (response time ¿ 10 ms) and plot the histograms in figures 34 and 38. To plot the histogram from middleware, we consider only those requests that have ¡ 10 ms response time. In all the cases, this covers 98+ % of the requests.

Comparing these plots, we observe that the middleware has more or less the same response time across all the requests in both sharded and non-sharded case (except for a small bump near 7 to 8 msec). We see higher spread in the response times measured at the client indicating that there is a larger variance in the network latencies between the clients and the middleware than inbetween the middleware and the servers. We see these trends for both sharded and non-sharded cases.

Finally, while comparing between the sharded and non-sharded case, we see that the response times of the sharded case is higher because of the sharding and multiple network I/O overheads. For multi-GET sizes ¿ 6, the size of the requests starts to play a more important role and thus sharding will start to be preferred than the non-sharded case when we have more than 6 keys in our multi-GETs.

6 2K Analysis

6.1 Experiment Overview

6.1.1 Variables

For our 2K analysis, we observe the changes in throughput and response time of the middleware while we vary the following parameters. All the GET requests have only one key and there are 3 client machines (with 64 total virtual clients per client VM).

Table 1: Factors used in the 2k-r analysis

Symbol	**Factor**	**Level -1**	**Level +1**
A	Memcached servers	2	3
B	Middlewares	1	2
C	Worker Threads per Middleware	8	32

We repeat the experiment for write-only, read-only, and a 50-50-read-write workloads as well. We use the memtier logs from the clients to get the numbers in the subsequent sections. The logs are at `logs/6_2k_exp_log` directory in the project's repository. The experimental configuration is as shown below

6.1.2 Experimental Configuration

Number of servers	2 and 3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	32
Workload	Write-only, Read-only, and 50-50-read-write
Multi-Get behaviour	N/A
Multi-Get size	N/A
Number of middlewares	1 and 2
Worker threads per middleware	8 and 32
Repetitions	3 (1 minute steady state)

The exact commands used depends on the exact configuration in question, for example, one vs two memcached instances. Thus, it is suggested to take a look at the commands from `scripts/1_exp_runners/5_run_2k.py` file.

6.2 Write-Only Loads

In this section, we perform a 2k-r experiment on the system when the work-load is write-only (i.e., consists only of `SETs`). The next two subsections display the tables from the analysis and a discussion follows afterwards.

6.2.1 Throughput

Table 2: Average Throughput for each configuration

I	A	B	C	AB	AC	BC	ABC	Mean Throughput
1	-1	-1	-1	1	1	1	-1	6110.47
1	-1	-1	1	1	-1	-1	1	10818.73
1	-1	1	-1	-1	1	-1	1	9275.35
1	-1	1	1	-1	-1	1	-1	17781.02
1	1	-1	-1	-1	-1	1	1	3142.77
1	1	-1	1	-1	1	-1	-1	6355.51
1	1	1	-1	1	-1	-1	-1	6609.54
1	1	1	1	1	1	1	1	12858.60

Table 3: Observed Throughput and errors across repetitions

I	A	B	C	AB	AC	BC	ABC	Rep_1	Rep_2	Rep_3	ERR_1	ERR_2	ERR_3
1	-1	-1	-1	1	1	1	-1	6583.99	5872.06	5875.35	473.52	-238.41	-235.12
1	-1	-1	1	1	-1	-1	1	10784.87	10559.31	11112	-33.86	-259.42	293.27
1	-1	1	-1	-1	1	-1	1	9237.88	9269.57	9318.59	-37.47	-5.78	43.24
1	-1	1	1	-1	-1	1	-1	18600.28	17272.73	17470.06	819.26	-508.29	-310.96
1	1	-1	-1	-1	-1	1	1	3115.38	3198.37	3114.57	-27.39	55.60	-28.20
1	1	-1	1	-1	1	-1	-1	6247.53	6427.93	6391.08	-107.98	72.42	35.57
1	1	1	-1	1	-1	-1	-1	6461.39	6646.43	6720.81	-148.15	36.89	111.27
1	1	1	1	1	1	1	1	12841.69	13124.02	12610.09	-16.91	265.42	-248.51

Table 4: Table of q values

	Mean
q0	9119.00
qA	-1877.39
qB	2512.13
qC	2834.47
qAB	-19.66
qAC	-469.02
qBC	854.22
qABC	-95.14

Table 5: Sum of Squares & Variation Contributions

Component	Sum of Squares	Contribution
SSY	2449347512.38	
SS0	1995747499.24	439.98
SST	453600013.14	100.00
SSA	84590387.28	18.65
SSB	151459030.80	**33.39**
SSC	192820830.83	**42.51**
SSAB	9281.09	0.00
SSAC	5279457.97	1.16
SSBC	17512466.73	3.86
SSABC	217227.45	0.05
SSE	1711330.99	0.38

Table 6: Average Latency(ms) for each configuration

I	A	B	C	AB	AC	BC	ABC	Mean Latency
1	-1	-1	-1	1	1	1	-1	31.55
1	-1	-1	1	1	-1	-1	1	17.69
1	-1	1	-1	-1	1	-1	1	21.89
1	-1	1	1	-1	-1	1	-1	10.91
1	1	-1	-1	-1	-1	1	1	60.93
1	1	-1	1	-1	1	-1	-1	30.21
1	1	1	-1	1	-1	-1	-1	29.03
1	1	1	1	1	1	1	1	14.96

6.2.2 Latency

Table 7: Observed Latency(ms) and errors across repetitions

I	A	B	C	AB	AC	BC	ABC	LAT_1	LAT_2	LAT_3	ERR_1	ERR_2	ERR_3
1	-1	-1	-1	1	1	1	-1	29.16	32.83	32.67	-2.40	1.28	1.12
1	-1	-1	1	1	-1	-1	1	17.80	17.96	17.31	0.11	0.27	-0.38
1	-1	1	-1	-1	1	-1	1	22.08	21.83	21.75	0.19	-0.06	-0.14
1	-1	1	1	-1	-1	1	-1	10.33	11.28	11.11	-0.58	0.37	0.20
1	1	-1	-1	-1	-1	1	1	61.62	60.04	61.13	0.69	-0.89	0.20
1	1	-1	1	-1	1	-1	-1	30.73	29.87	30.04	0.52	-0.34	-0.17
1	1	1	-1	1	-1	-1	-1	29.46	28.97	28.66	0.43	-0.06	-0.37
1	1	1	1	1	1	1	1	14.99	14.65	15.24	0.03	-0.31	0.28

Table 8: Table of q values

	Mean
q0	27.15
qA	6.64
qB	-7.95
qC	-8.70
qAB	-3.84
qAC	-2.49
qBC	2.44
qABC	1.72

Table 9: Sum of Squares and Variation Contribution Analysis

Component	Sum of Squares	Contribution
SSY	22806.38	
SS0	17685.51	345.36
SST	5120.87	100.00
SSA	1057.09	20.64
SSB	1516.86	**29.62**
SSC	1818.47	**35.51**
SSAB	353.54	6.90
SSAC	149.15	2.91
SSBC	143.03	2.79
SSABC	71.06	1.39
SSE	11.66	0.23

6.2.3 Discussion

In both throughput and latency, we see that the number of Middlewares and the number of workers threads in each of the middleware contributes significantly to the observed throughput and lantency. Together they explain around 70% of the observed variations in throughput and latency. Moreover, the interaction terms like AB, AC, AB, ABC have very small contributions to the variance indicating that the factors are not correlated.

More specifically, the number of worker threads contributes more to the variance than the number of middlewares. To understand this, let's take a look at the definitions of throughput and latency. Throughput is the average number of jobs completed per unit time. Now, to increase this term, we either need to reduce the time taken to complete one job or increase the level of parallelism so that more jobs can be processed at the same time. In our systems, the network latency dominates the time taken to complete a request (this can be seen from any of the middleware logs in any of the sections). We cannot alter the network latency by changing any of the three factors, hence we cannot significantly reduce the amount of time a single request takes. Therefore, increasing the parallelism is the only way ahead.

Now, let's take a look at response time, it is the average amount of time a request takes to complete. As we have just seen, the network latency is the biggest contributor to the response time. Since the worker threads are synchronous, they have to wait until the response from the memcached servers comes back before processing the next request. This means that the worker threads are idle most of the time and the CPU is underutilized. Adding another middleware does not increase the utilization of a single core as you are doubling the number of cores and effectively reducing the load per core (as the requests are now split across two middlewares) when you are adding another middleware. However, by increasing the number of worker threads, you are increasing the utilization of the CPU and are effectively reducing the amount of time the middleware takes to forward the request to the memcached servers.

From the above two paragraphs, it is clear that increasing the level of parallelism will be the way to higher throughput and lower service times. Parallelism in the middleware(s) can be understood by looking at the total number of worker threads. Let's take the case when there's only one middleware with 8 worker threads (Both B and C are -1). If we were to change factor B, the number of middlewares, the total number of worker threads would be 16. Instead, if we change factor C, the number of worker threads, the total number of worker threads goes up to 32. Thus, the latter has double the effect of the former option and as a result has more effect on both the latency and throughput due to the effective increase in parallelism it introduces.

6.3 Read-Only Loads

In this section, we perform a 2k-r experiment on the system when the work-load is read-only (i.e., consists only of `GET`s). There are no multi-`GET`s and the memcached servers are pre-populated to avoid any GET misses.

6.3.1 Throughput

Table 10: Average Throughput for each configuration

I	A	B	C	AB	AC	BC	ABC	Mean Throughput
1	-1	-1	-1	1	1	1	-1	7141.31
1	-1	-1	1	1	-1	-1	1	12813.33
1	-1	1	-1	-1	1	-1	1	10894.01
1	-1	1	1	-1	-1	1	-1	21132.15
1	1	-1	-1	-1	-1	1	1	5537.52
1	1	-1	1	-1	1	-1	-1	11035.39
1	1	1	-1	1	-1	-1	-1	10639.02
1	1	1	1	1	1	1	1	19984.16

Table 11: Observed Throughput and errors across repetitions

I	A	B	C	AB	AC	BC	ABC	Rep_1	Rep_2	Rep_3	ERR_1	ERR_2	ERR_3
1	-1	-1	-1	1	1	1	-1	6894.96	6907.95	7621.01	-246.35	-233.36	479.70
1	-1	-1	1	1	-1	-1	1	12602.77	13113.52	12723.7	-210.56	300.19	-89.63
1	-1	1	-1	-1	1	-1	1	11112.42	10709.1	10860.5	218.41	-184.91	-33.51
1	-1	1	1	-1	-1	1	-1	20920.12	21609.2	20867.14	-212.03	477.05	-265.01
1	1	-1	-1	-1	-1	1	1	5270.24	5655.58	5686.73	-267.28	118.06	149.21
1	1	-1	1	-1	1	-1	-1	10950.75	10764.43	11391	-84.64	-270.96	355.61
1	1	1	-1	1	-1	-1	-1	10710.6	10579.93	10626.53	71.58	-59.09	-12.49
1	1	1	1	1	1	1	1	20216.69	19582.97	20152.81	232.53	-401.19	168.65

Table 12: Table of q values

	Mean
q0	12397.11
qA	-598.09
qB	3265.22
qC	3844.15
qAB	247.34
qAC	-133.39
qBC	1051.67
qABC	-89.86

Table 13: Table of Sum of Squares and Variation Contribution Analysis

Component	Sum of Squares	Contribution
SSY	4337759141.88	
SS0	3688520320.39	568.13
SST	649238821.49	100.00
SSA	8585043.67	1.32
SSB	255880467.30	**39.41**
SSC	354659356.93	**54.63**
SSAB	1468284.44	0.23
SSAC	427058.76	0.07
SSBC	26544382.17	4.09
SSABC	193786.68	0.03
SSE	1480441.54	0.23

6.3.2 Latency

Table 14: Average Latency(ms) for each configuration

I	A	B	C	AB	AC	BC	ABC	Mean Latency
1	-1	-1	-1	1	1	1	-1	26.87
1	-1	-1	1	1	-1	-1	1	14.98
1	-1	1	-1	-1	1	-1	1	19.36
1	-1	1	1	-1	-1	1	-1	9.14
1	1	-1	-1	-1	-1	1	1	34.71
1	1	-1	1	-1	1	-1	-1	17.43
1	1	1	-1	1	-1	-1	-1	18.19
1	1	1	1	1	1	1	1	9.59

Table 15: Observed Latency(ms) and errors across repetitions

I	A	B	C	AB	AC	BC	ABC	Rep_1	Rep_2	Rep_3	ERR_1	ERR_2	ERR_3
1	-1	-1	-1	1	1	1	-1	27.84	27.79	24.98	0.97	0.92	-1.89
1	-1	-1	1	1	-1	-1	1	15.23	14.64	15.08	0.25	-0.35	0.10
1	-1	1	-1	-1	1	-1	1	19.08	19.61	19.40	-0.28	0.24	0.04
1	-1	1	1	-1	-1	1	-1	9.27	8.90	9.24	0.13	-0.24	0.10
1	1	-1	-1	-1	-1	1	1	36.43	33.94	33.76	1.72	-0.77	-0.95
1	1	-1	1	-1	1	-1	-1	17.53	17.83	16.92	0.10	0.41	-0.51
1	1	1	-1	1	-1	-1	-1	18.11	18.29	18.18	-0.09	0.10	-0.01
1	1	1	1	1	1	1	1	9.51	9.74	9.52	-0.08	0.15	-0.07

Table 16: Table of q values

	Mean
q0	18.78
qA	1.20
qB	-4.71
qC	-6.00
qAB	-1.38
qAC	-0.47
qBC	1.29
qABC	0.88

Table 17: Table of Sum of Squares and Variation Contribution Analysis

Component	Sum of Squares	Contribution
SSY	10019.89	
SS0	8468.90	546.03
SST	1550.98	100.00
SSA	34.31	2.21
SSB	532.95	**34.36**
SSC	863.68	**55.69**
SSAB	45.40	2.93
SSAC	5.33	0.34
SSBC	40.11	2.59
SSABC	18.48	1.19
SSE	10.71	0.69

6.3.3 Discussion

We see similar trends in the contributions to variations like seen in the write-only workload section. These can be explained with similar arguments as made before and hence will not be repeated here. An interesting observation to make is the wider gap between SSC and SSB in both throughput and latency as compared to the same quantities in the write-only work loads. In order to understand this, we need to recollect how the middleware handles SETs and Single-GETs.

In the case of SETs, the middleware replicates the request to each of the memcached servers, for a total of $|memcached\_servers|$ network writes. For Single-GETs, the middleware simply forwards the request to just one of the memcached servers and load-balances among st the servers by using a round-robin strategy. Even though the multiple network I/Os in case of SETs are not done sequentially, it is important to remember that all worker threads are forwarding requests to all the memcached servers, increasing the memcached servers' load and consequently increasing the time the single-threaded memcached servers take to process these requests and send a response back.

Thus, the network latencies play a slighty smaller role here when compared to the write-only workload and as a result, the effective parallelism plays a more significant role. Thus, increasing the number of worker threads per middleware has more pronounced effect than increasing the number of middlewares.

6.4 Read-Write Loads

In this section, we perform a 2k-r experiment on the system when the work-load is read-write (50-50) (i.e., we pass `--ratio 1:1` flag to memtier to have equal number of SETs and GETs).

6.4.1 Throughput

Table 18: Average Throughput for each configuration

I	A	B	C	AB	AC	BC	ABC	Mean Throughput
1	-1	-1	-1	1	1	1	-1	6356.06
1	-1	-1	1	1	-1	-1	1	11769.73
1	-1	1	-1	-1	1	-1	1	10533.18
1	-1	1	1	-1	-1	1	-1	18866.50
1	1	-1	-1	-1	-1	1	1	4007.03
1	1	-1	1	-1	1	-1	-1	7307.03
1	1	1	-1	1	-1	-1	-1	8428.42
1	1	1	1	1	1	1	1	15992.66

Table 19: Observed Throughput and errors across repetitions

I	A	B	C	AB	AC	BC	ABC	Rep_1	Rep_2	Rep_3	ERR_1	ERR_2	ERR_3
1	-1	-1	-1	1	1	1	-1	6232.82	6180.9	6654.47	-123.24	-175.16	298.41
1	-1	-1	1	1	-1	-1	1	12054.08	11417.01	11838.1	284.35	-352.72	68.37
1	-1	1	-1	-1	1	-1	1	10542.57	10579.65	10477.33	9.39	46.47	-55.85
1	-1	1	1	-1	-1	1	-1	19360.33	18999.67	18239.5	493.83	133.17	-627.00
1	1	-1	-1	-1	-1	1	1	3987.2	4015.18	4018.7	-19.83	8.15	11.67
1	1	-1	1	-1	1	-1	-1	6129.36	7755.02	8036.72	-1177.67	447.99	729.69
1	1	1	-1	1	-1	-1	-1	8354.41	8368.4	8562.45	-74.01	-60.02	134.03
1	1	1	1	1	1	1	1	16266.49	16348.99	15362.5	273.83	356.33	-630.16

Table 20: Table of q values

	Mean
q0	10407.58
qA	-1473.79
qB	3047.61
qC	3076.40
qAB	229.14
qAC	-360.34
qBC	897.99
qABC	168.07

Table 21: Table of Sum of Squares and Variation Contribution Analysis

Component	Sum of Squares	Contribution
SSY	3129965586.60	
SS0	2599623857.89	490.18
SST	530341728.70	100.00
SSA	52129514.52	9.83
SSB	222910789.66	**42.03**
SSC	227142240.79	**42.83**
SSAB	1260137.10	0.24
SSAC	3116314.01	0.59
SSBC	19353067.41	3.65
SSABC	677964.13	0.13
SSE	3751701.09	0.71

6.4.2 Latency

Table 22: Average Latency(ms) for each configuration

I	A	B	C	AB	AC	BC	ABC	Mean Latency
1	-1	-1	-1	1	1	1	-1	30.23
1	-1	-1	1	1	-1	-1	1	16.32
1	-1	1	-1	-1	1	-1	1	19.34
1	-1	1	1	-1	-1	1	-1	10.31
1	1	-1	-1	-1	-1	1	1	47.91
1	1	-1	1	-1	1	-1	-1	26.65
1	1	1	-1	1	-1	-1	-1	22.80
1	1	1	1	1	1	1	1	12.02

Table 23: Observed Latency(ms) and errors across repetitions

I	A	B	C	AB	AC	BC	ABC	Rep_1	Rep_2	Rep_3	ERR_1	ERR_2	ERR_3
1	-1	-1	-1	1	1	1	-1	30.80	31.06	28.85	0.56	0.83	-1.39
1	-1	-1	1	1	-1	-1	1	15.93	16.82	16.22	-0.39	0.50	-0.10
1	-1	1	-1	-1	1	-1	1	19.40	19.24	19.39	0.06	-0.11	0.05
1	-1	1	1	-1	-1	1	-1	10.08	10.20	10.66	-0.23	-0.12	0.35
1	1	-1	-1	-1	-1	1	1	48.16	47.81	47.77	0.25	-0.10	-0.14
1	1	-1	1	-1	1	-1	-1	31.32	24.75	23.88	4.67	-1.90	-2.77
1	1	1	-1	1	-1	-1	-1	22.99	22.94	22.47	0.19	0.14	-0.33
1	1	1	1	1	1	1	1	11.81	11.76	12.50	-0.21	-0.26	0.47

Table 24: Table of q values

	Mean
q0	23.20
qA	4.15
qB	-7.08
qC	-6.87
qAB	-2.86
qAC	-1.14
qBC	1.92
qABC	0.70

Table 25: Table of Sum of Squares and Variation Contribution Analysis

Component	Sum of Squares	Contribution
SSY	16030.63	
SS0	12916.52	414.78
SST	3114.10	100.00
SSA	412.76	13.25
SSB	1203.41	**38.64**
SSC	1133.69	**36.40**
SSAB	195.64	6.28
SSAC	31.02	1.00
SSBC	88.61	2.85
SSABC	11.79	0.38
SSE	37.18	1.19

6.4.3 Discussion

In this case, just like the two discussed before, the number of worker threads and the number of middlewares contribute the most to the variation in throughput and latency. From the tables 21 and 25, it seems like the number of middlewares contributes slightly more than the number of worker threads. On a closer inspection and checking the computations in `scripts/2k_analysis.ods`, it is clear that contributions of factor B and C are same (within error bounds)

To understand why this happens, we need to consider the fact that the read-write load has two kinds of requests, the slower SETs and the faster GETs. The potential improvements in throughput and latency of GETs is diminished because on average, the GETs are blocked by a slower SET request 50% of the time (because there are equal number of SETs and GETs and we assume that the system at any given time has equal numbers of SETs and GETs) and have to wait for the SETs to complete first. As a result of this, we see that factors B and C end up contributing almost similar amounts to variations in throughput and latency.

7 Queuing Model

7.1 M/M/1

In this section, we will model our middleware as an M/M/1 system when there are 8, 16, 32 and 64 worker threads per middleware (or 16, 32, 64, 128 total worker threads). We will compute things like Traffic Intensity, Mean number of jobs in the system, number of jobs in the queue, response time and the waiting time. We will also compare these with the observed values from section 4 and comment on the plausible reasons for the observed deviations (if any)

The following are the maximum throughputs observed for a given number of worker threads in a single repetition. These will function as the service rate to our models.

Table 26: Maximum Observed Throughput / Service Rate

Worker Threads	Max TPS
8	8877.54
16	11719.44
32	15070.91
64	17069.94

Since we have a closed system and do not lose any requests, the average throughput in each of these cases will be the arrival rate. Once we have these two parameters, we can compute the rest using results from queuing theory as below. The exact calculations and numbers can be found in `scripts/MM1.ods` in the repository.

Table 27: M/M/1 Queueing Model, λ arrival rate, μ service rate (both jobs/sec)

Parameter	Formula
Traffic Intensity (ρ)	λ/μ
Mean num jobs in system	$\rho/(1-\rho)$
Mean num jobs in queue	$\rho^2/(1-\rho)$
Mean response time	$(1/\mu)/(1-\rho)$
Mean waiting time	$\rho\frac{1/\mu}{(1-\rho)}$

7.1.1 WT 8

For the under-saturation region, let's take Num VC = 6 and for the saturation region, Num VC = 36. These have average throughputs of 2368.2 and 8072 respectively.

Table 28: M/M/1 model for WT 8, undersaturation (VC = 6)

Statistics	Model	Measured	difference
Arrival rate	2368.2000		
Service rate	8877.5429		
Traffic intensity (System utilization)	0.2668		
Probability of zero jobs in system	0.7332		
Mean number of jobs in system	0.3638	16.1898	15.8260
Mean number of jobs in queue	0.0971	0.1898	0.0927
Variance of number of jobs in queue	0.1583	0.0009	0.1573
Mean response time	0.0002	2.0974	2.0973
Variance of response time	0.0000000	0.0170	0.0170
Mean waiting time	0.00004	0.0181	0.0181
Variance of waiting time	0.00	0.0000	0.0000

Table 29: M/M/1 model for WT 8, saturation (VC = 36)

Statistics	Model	Measured	difference
Arrival rate	8072.0000		
Service rate	8877.5429		
Traffic intensity (System utilization)	0.9093		
Probability of zero jobs in system	0.0907		
Mean number of jobs in system	10.0206		10.0206
Variance of number of jobs in system	110.4324		110.4324
Mean number of jobs in queue	9.1113	6.3080	2.8033
Variance of number of jobs in queue	108.6964	0.0001	108.6963
Mean response time	0.0012	4.1283	4.1271
Variance of response time	0.0000015	0.0021	0.0021
Mean waiting time	0.00113	1.8952	1.8941
Variance of waiting time	0.00	0.0004	0.0004

7.1.2 WT 16

For the under-saturation region, let's take Num VC = 36 and for the saturation region, Num VC = 72. These have average throughputs of 8186 and 10470.12 respectively.

Table 30: M/M/1 model for WT 16, undersaturation (VC = 36)

Statistics	Model	Measured	difference
Arrival rate	8186.0000		
Service rate	11719.4429		
Traffic intensity (System utilization)	0.6985		
Probability of zero jobs in system	0.3015		
Mean number of jobs in system	2.3167	34.1553	31.8386
Mean number of jobs in queue	1.6182	2.1553	0.5371
Variance of number of jobs in queue	6.4975	0.0022	6.4953
Mean response time	0.0003	3.4170	3.4167
Variance of response time	0.0000001	0.0238	0.0238
Mean waiting time	0.00020	0.1956	0.1954
Variance of waiting time	0.00	0.0001	0.0001

Table 31: M/M/1 model for WT 16, saturation (VC = 72)

Statistics	Model	Measured	difference
Arrival rate	10470.1200		
Service rate	11719.4429		
Traffic intensity (System utilization)	0.8934		
Probability of zero jobs in system	0.1066		
Mean number of jobs in system	8.3806		8.3806
Variance of number of jobs in system	78.6157		78.6157
Mean number of jobs in queue	7.4872	11.7454	4.2582
Variance of number of jobs in queue	76.9241	0.7512	76.1729
Mean response time	0.0008	5.9937	5.9929
Variance of response time	0.0000006	0.0609	0.0609
Mean waiting time	0.00072	2.5890	2.5883
Variance of waiting time	0.00	0.0170	0.0170

7.1.3 WT 32

For the under-saturation region, let's take Num VC = 72 and for the saturation region, Num VC = 144. These have average throughputs of 11840.6 and 14393.32 respectively.

Table 32: M/M/1 model for WT 32, undersaturation (VC = 72)

Arrival rate	11840.6000		
Service rate	15070.9143		
Traffic intensity (System utilization)	0.7857		
Probability of zero jobs in system	0.2143		
Mean number of jobs in system	3.6655	67.4207	63.7553
Mean number of jobs in queue	2.8798	3.4207	0.5409
Variance of number of jobs in queue	15.6982	0.0385	15.6596
Mean response time	0.0003	4.5511	4.5508
Variance of response time	0.0000001	0.0165	0.0165
Mean waiting time	0.00024	0.2780	0.2777
Variance of waiting time	0.00	0.0151	0.0151

Table 33: M/M/1 model for WT 32, saturation (VC = 144)

Arrival rate	14393.3200		
Service rate	15070.9143		
Traffic intensity (System utilization)	0.9550		
Probability of zero jobs in system	0.0450		
Mean number of jobs in system	21.2418	85.8504	64.6086
Mean number of jobs in queue	20.2868	21.8504	1.5637
Variance of number of jobs in queue	470.5886	0.0951	470.4935
Mean response time	0.0015	8.6359	8.6344
Variance of response time	0.0000022	0.2852	0.2852
Mean waiting time	0.00141	3.6772	3.6758
Variance of waiting time	0.00	0.0987	0.0987

7.1.4 WT 64

For the under-saturation region, let's take Num VC = 144 and for the saturation region, Num VC = 192. These have average throughputs of 15642.9 and 16691.45 respectively.

Table 34: M/M/1 model for WT 64, undersaturation (VC = 144)

Statistics	Model	Measured	difference
Arrival rate	15642.9000		
Service rate	17069.9400		
Traffic intensity (System utilization)	0.9164		
Probability of zero jobs in system	0.0836		
Mean number of jobs in system	10.9618	135.2539	124.2921
Mean number of jobs in queue	10.0454	7.2539	2.7915
Variance of number of jobs in queue	129.3662	0.3296	129.0366
Mean response time	0.0007	6.4727	6.4720
Variance of response time	0.0000005	0.2591	0.2591
Mean waiting time	0.00064	0.4637	0.4631
Variance of waiting time	0.00	0.0154	0.0154

Table 35: M/M/1 model for WT 64, saturation (VC = 192)

Statistics	Model	Measured	difference
Arrival rate	16691.4500		
Service rate	17069.9400		
Traffic intensity (System utilization)	0.9778		
Probability of zero jobs in system	0.0222		
Mean number of jobs in system	44.1001	138.1615	94.0614
Mean number of jobs in queue	43.1223	10.1615	32.9608
Variance of number of jobs in queue	1986.9857	0.2209	1986.7648
Mean response time	0.0026	8.4539	8.4513
Variance of response time	0.0000070	0.2915	8.4539
Mean waiting time	0.00258	1.2236	0.2889
Variance of waiting time	0.00	0.0946	0.0945

7.1.5 Discussions

From the above tables, we see that all the parameters we predict are way off from the actual parameters we observe. This is not surprising because in our model, we have two middlewares (hence two queues) with more than one service center (worker thread) per middleware. These two features are significant in increasing the parallelism and thus responsible for increasing throughput and reducing response times in our system. Modelling it as a M/M/1 removes both of these features and hence the predicted and observed values do not match at all across all parameters. We can thus conclude that M/M/1 s are too simple to model our system because it does not take into account the multiple worker threads and queues as well as asynchronous behaviour of the net thread.

7.2 M/M/m

In this section, we model our middleware as an M/M/m model which implies that there is only one queue and there are m servers / worker threads. Like before, we will compare the observed and predicted parameters of the model. The following table lists these parameters and equations to compute them from queuing theory.

Table 36: M/M/m Queueing Model, m service centers λ arrival rate, μ service rate of one server (both jobs/sec). ϱ is the probability of more than m jobs in system

Parameter	Formula
Traffic Intensity (ρ)	$\lambda/m\mu$
Mean num jobs in system	$m\rho + \rho\varrho/(1-\rho)$
Mean num jobs in queue	$\rho\varrho/(1-\rho)$
Mean response time	$\frac{1}{\mu}(1 + \frac{\varrho}{m(1-\rho)})$
Mean waiting time	$\varrho/[m\mu(1-\rho)]$

We keep things simple and model only the case when there are 16 worker threads per middleware and consider two cases, when it is undersaturated and when it is saturated. Note that we don't consider the oversaturation region in either this or the previous sections because of two reasons. The first is that, in a closed system, the number of jobs is bounded and it is impossible to have utilization ρ greater than 1 (oversaturation). Secondly, even if we are to see similar situations, the system is not stable and is unpredictable in this region. Hence, trying to model the system during oversaturation does not provide us any meaningful insights.

The motivations of the input parameters are the same as the ones in the previous section. One thing in particular is that we divide the service rate by m, the number of worker threads in order to get the per-server service rate. The exact calculations and numbers can be found in `scripts/MMm.ods` in the repository.

7.2.1 WT 16, Num VC = 36, Under-saturation

Table 37: M/M/m with WT = 16, Num VC = 36, at Undersaturation

Statistics	Model	Observed	Difference
Arrival rate	8186.0000		8186.0000
Service rate per worker thread	366.2326		366.2326
Number of Services	32.0000		32.0000
Traffic intensity (System utilization)	0.6985		0.6985
Probability of zero jobs in system	0.0000		0.0000
Probability of queuing	0.0371		0.0371
Mean number of jobs in the system	22.4378	34.1553	11.7175
Mean number of jobs in the queue	0.0859	2.1553	2.0695
Variance of number of jobs in the queue	0.4763	0.0022	0.4741
Mean response time	0.0027	3.4170	3.4142
Variance of response time	0.0000	0.0238	0.0238
Mean waiting time	0.0000	0.1956	0.1956
Variance of the waiting time	0.0000	0.0001	0.0001

7.2.2 WT 16, Num VC = 72, Saturation

Table 38: M/M/m with WT = 16, Num VC = 72, at Saturation

Statistics	Model	Observed	Difference
Arrival rate	10470.1200		10470.1200
Service rate per worker thread	366.2326		366.2326
Number of Services	32.0000		32.0000
Traffic intensity (System utilization)	0.8934		0.8934
Probability of zero jobs in system	0.0000		0.0000
Probability of queuing	0.4316		0.4316
Mean number of jobs in the system	32.2057	34.1553	1.9496
Mean number of jobs in the queue	3.6170	2.1553	1.4616
Variance of number of jobs in the queue	51.1595	0.0022	51.1573
Mean response time	0.0031	3.4170	3.4139
Variance of response time	0.0000	0.0238	0.0238
Mean waiting time	0.0003	0.1956	0.1952
Variance of the waiting time	0.0000	0.0001	0.0001

7.2.3 Discussions

Unlike the M/M/1 models, the predictions for mean number of jobs in the system by the M/M/m model are quite close to the actual observed values. The jobs in queue predictions are slightly off because the model does not incorporate the fact that we two queues in our model. The other predictions, like those for response time are orders of magnitude off. This probably hints at the fact that having multiple queues in two different middlewares helps to effectively distribute the load between two middlewares and thus drastically reduce the response time. We can conclude that even though the M/M/m model is able to capture some features of our model, it fails to capture others. We will model the middleware and load generating VMs as a network of queues in the next section.

7.3 Network of Queues

In this section, we look specifically into the configuration of two middlewares from Section 3. We model the system (middlewares, memcached server and the memtier clients) as a closed network of queues for both read-only and write-only loads. In the middleware, the net thread is modelled as an M/M/1 center. This is because even though we don't specifically put a queue in the net thread, the incoming requests are nevertheless queued at a lower level and the net thread handles them sequentially. The service time of this center is the time between we get the full request in the buffer and the time we enqueue it into the request queue.

We model the worker thread thread and the memcached server together because each worker thread waits for the memcached to reply back before servicing the next request Hence, the time to send to memcached and to get back the response from the MC server has to be included into the service time of the worker threads. Moreover, even though the single-threaded memcached server would be internally using a queue, we cannot get the queuing time at memcached server separately and hence cannot model it separately. The service time for this component is the time between dequeueing the request from the request queue and sending the response back to the client.

Finally, we model the network connection between the clients and middlewares as delay centers (M/M/inf) with service times equal to half the difference between the response time measured at the clients and the response time measured at the middleware.

The network is graphically represented in Figure 39.

Here, NW refers to the network connections, NT 1 and NT 2 are net threads in the two middlewares. WT1 ... WTn are the *n* worker thread in each of the middlewares. Finally, MC refers to the single memcached server that all the worker threads in both the middleware connect to. It is shown multiple times for easier visualization but in reality there is only one instance of memcached in our setup. An enlarged version of the diagram can be found in `images/7_network_of_queues.png` in the repository. We use the `queuing` package from Octave to model the network of queues. The script is at `scripts/network.m`.

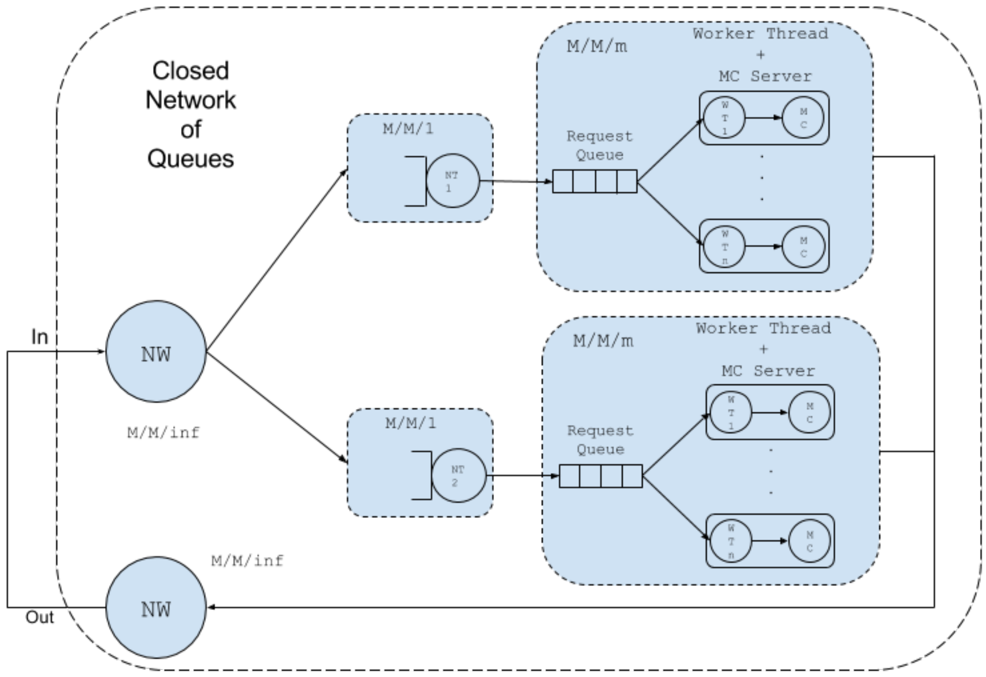


Figure 39: Network of Queues Model of the system

7.3.1 Parameters of the Model

A General Notes on Experimental Setup

Before we delve into the experimental details and analysis of each of the following sections, there are a few common things to keep in mind about how the experiments are conducted. These are presented here in order to avoid repeating them in each subsequent section and to keep them in mind while explaining the outcomes of our experiments. The scripts used to run the experiments can be found at `scripts/1_exp_runners` directory. We have one script per section (of analysis in the report). Note that, unless stated otherwise, each configuration is repeated three time to remove random fluctuations of the observed values. Finally, all middleware and memcached instances listen on port 6969

Abbreviations For the sake of brevity, we use several abbreviations throughout the report. Most can be inferred from context, nevertheless, here are a few abbreviations we use.

- MT Memtier Instance if more than one memtier per VM, else load generating VM itself
- LG Load Generating VMs, these are A2 VMs
- MW Middleware, these are A4 VMs
- MC Memcached servers / VMs, these are A1 VMs
- TPS Throughput
- LAT / RT Latency / Response Time (used interchangeably)

A.1 VM Allocation and Deallocation

We have split up the automated running of experiments for each of the sections into separate scripts. Each of the scripts contains calls to the Azure Command Line Utility which allocates the Virtual Machines needed for the experiments(we use the command `az vm allocate` to allocate and start VMs). Once this is done, memcached instances are started on the A1 VMs, appropriate directories to store the logs are created on the A2 and A4 VMs (the load generating and middleware VMs). Finally, the memcached servers are pre-populated as discussed in a subsequent section. And before running the different configurations, ping tests are run to ensure stable network conditions, this is discussed in the next section. Finally, once all the experiments are done, the scripts deallocate the VMs (using `az vm deallocate`) so that we don't waste credits keeping the VMs running when not in use.

This way, we ensure that all the numbers we report in a given section are from similar if not same network conditions and thus the numbers can be compared and system behaviours can be inferred from it.

A.2 Network Latencies

In order to make sure that the network connections between the different components of a setup are similar, we run pings between each individual connections between components of our setup the before starting our experiments. Depending on the experiment in question, the setup can vary. For example, 1, 2 or 3 Load generating VMs with one or two `memtier_benchmark` instances per VM connected to one or two middlewares each in turn can be connected one, two or three memcached servers. The results of these ping tests can be found in `logs/<EXP_NAME>/mt_log/` directories each with the prefix `ping`. It can be argued that the network latencies and general behaviour can vary depending on the amount of load you put on the network due to the TCP Flow Control mechanisms kicking in to avoid congestion. This means that raw mean latencies from these pings should be considered with the grain of salt. We observe that the mean latencies, as measured by ping (the exact command used is `ping`

`-c 5 <private_IP_of_destination>`) have the same order of magnitude (a few milliseconds) across all experiments. On a closer look at the exact numbers, some of the links can have almost double the mean latency of another link. For example, see `ping_foraslvms1.txt` and `ping_foraslvms2.txt` at `logs/2a_one_server_base_line_log`. The former has more than double the average latency of the latter. Though, this may not cause significant issues as we usually average the latencies across the load generating VMs, it is important to note this observation as a plausible source of deviations from the norm.

A.3 Pre-populating memcached servers

After starting the memcached instances, they are empty. This means that any GET request will return an empty response (just a `END\r\n`). These are cheap to construct and send back and thus can have significant effects in the numbers we measure if we have "GET Misses". In order to avoid these, we always pre-populated the memcached servers using memtier instances to ensure that there are no misses. We do this even when there are only write-only workloads, in order to ensure that we are comparing similarly loaded memcached servers across different workloads (read-only, write-only, 50-50). We use the following command to prepopulate the MC servers. We use very long expiry range in order to make sure that no keys are evicted before the experiments are completed.

```
   memtier_benchmark -s <PRIVATE_MC_IP> -p <MC_PORT> -P memcache_text -n allkeys
--ratio 1:0 --key-maximum 10000 --hide-histogram --expiry-range 99999-100000
-d 1024 --key-pattern=S:S> /dev/null 2>&1
```

A.4 Performance Monitoring (`dstat`)

In many of the analysis that we subsequently do, two key questions we try to answers are what is the bottleneck of the system and what are the utilizations of the different components. We hope to get these purely from the MT and MW logs. However, since the memtier instances are running on mediocrely powerful VMs, we can get more insights by looking at how much the CPU, RAM and Network interface are used. This will allow us to better explain why a component is a bottleneck or not. We use `dstat` on the LG VMs for this purpose and save the results in a CSV file in the same directory as the memtier logs. These files start with the prefix `dstat_`. The exact command we use is `dstat -cmn --nocolor --output <out_file> 5 18 >/dev/null </dev/null 2>&1`. Here the readings are taken every 5 seconds and are done 18 times (roughly the same amount of time a single repetition of a configuration runs)

A.5 Experimental Startup

All experiments for a section are done together and the VMs and deallocated only after all the experiments belonging to one section are completed. In each of the experiments and each of the repetitions we run, we do the following steps.

- Start the MC instances
- create log directories in MT and MW instances
- Pre-populate the MC servers
- Start Middlewares (if any are needed)
- Wait 20 seconds for Middleware to connect to all the memcached servers to avoids errors
- Start `dstat`
- Run the `memtier_benchmark`(s)
- Kill the Middleware

- Wait for 5 seconds before starting next repetition / configuration

A.6 Common Configurations

A.6.1 Memcached

All instances of memcached across all experiments are started with this command `memcached -t 1 -p 6969`. We always run the servers in single threaded mode because the A1 VMs have only one core and also because it is advisable to use memcached in single-threaded mode.

A.6.2 Memtier

Many of the flags that we use with memtier depend on the experiment in question but there are a few that are common across all experiments. An incomplete set is `memtier_benchmark -s <Dest_IP> -p 6969 -P memcache_text --test-time 80 --key-maximum 10000 -d 1024 --json-out-file <JSON_FILE_NAME>`. The additional flags used in each report will be described in their corresponding sections.