Advanced Systems Lab Report

Autumn Semester 2018

Name: Zhifei Yang
Legi: 17-941-998

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

1 System Overview (75 pts)

1.1 System In A Nutshell

In this project we created a middleware-based key-value store system, which consists of three components: the memtier clients, the middleware, and the memcached servers.

The memtier clients set and retrieve key-value pairs by continuously sending requests to the server and port specified in its startup arguments. The request format complies to the memcached protocol [1]. A memtier instance runs with multiple threads, and each thread works as multiple virtual clients: a client sends requests one by one, i.e. it will not send a new request until it has received the response of its last request.

The middleware relays requests and responses between clients to memcached servers, and balances the load. From a client's point of view, it is transparent and behaves in the same way as a memcached server would do. The middleware can receive connections from multiple clients, forward their requests to one or more servers depending on the request type, and send the responses of servers to the corresponding clients. With a multi-threaded architecture, multiple workers can concurrently process multiple requests.

The memcached server listens on a port by both TCP and UDP, and can keep many client connections simultaneously. It can work with multiple threads, but in this project we only use one. It supports a variety of request types, such as storage, retrieval and statistics commands, while in this project we only consider `GET`s and `SET`s.

To deploy the system, a Microsoft Azure cluster with eight virtual machines is used, of which the configurations are shown in Table 1.1. The upload and download bandwidths are measured with `iperf`, each with three repetitions, so both the averages and the standard deviations (in brackets) are listed. To measure the download bandwidth of one virtual machine, it is configured as `iperf` server, and all other machines are setup as clients to send traffic simultaneously to it. Due to various factors of data center VM allocation, and the limit of individual client's upload bandwidth, they together may still not hit the server's true download limit, so the measured value is only the highest lower bound that can be observed. For upload bandwidth, a machine is configured as `iperf` client to send traffic to every server that it needs to communicate with directly in our experiments, with one server each time; the results are consistent among different servers and much lower than any server's download bandwidth, so it is safe to say that the true upload bandwidth is revealed.

VM Name	Type	#Cores	Memory (GB)	Upload (Mbits/s)	Download (Mbits/s)
Client1	Basic A2	2	3.5	201.2 (0.45)	2311.8 (4.39)
Client2	Basic A2	2	3.5	201.2 (0.45)	2310.2 (5.94)
Client3	Basic A2	2	3.5	201.4 (0.55)	2306.1 (1.42)
Middleware1	Basic A4	8	14	801.2 (2.64)	1701.3 (3.21)
Middleware2	Basic A4	8	14	801.2 (1.72)	1705.0 (7.81)
Server1	Basic A1	1	1.75	100.0 (0.00)	1631.6 (20.1)
Server2	Basic A1	1	1.75	100.4 (0.55)	1593.6 (15.7)
Server3	Basic A1	1	1.75	100.6 (0.55)	2391.9 (3.41)

Table 1.1: Azure Virtual Machine Configurations

For running experiments, unless stated otherwise, all repetitions are run for 80 seconds, where the first and last ten seconds are treated as "warm-up" and "cool-down" phases and removed, so the effective data in each repetition has a time length of 60 seconds.

[1] https://github.com/memcached/memcached/blob/master/doc/protocol.txt

1.2 The Middleware

The middleware is implemented in Java 8 with a multi-threaded architecture, as is shown in Figure 1.1, including one main thread (`MainThread`), one network thread (`NetThread`), multiple worker threads (`WorkerThread`), one statistics collector thread (`StatCollector`) and one exit handler thread (`ExitHandler`). In this section, we introduce the functionality and implementation of these threads, as well as the supporting classes around them.

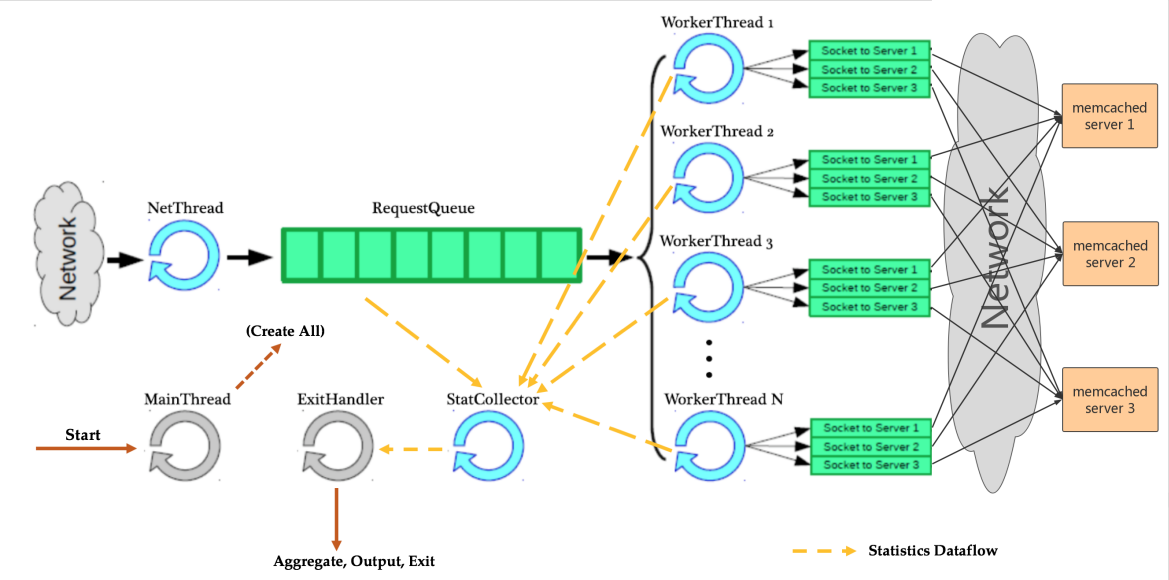

Figure 1.1: Multi-threaded Middleware Architecture

1.2.1 Java Classes Functionality and Implementation

The middleware is implemented in a number of classes, which can be classified into three categories and explained as below:

I. Queueing and Networking

1. `MyMiddleware` - The main class of the middleware, where the `MainThread` starts. The main thread is the bootstrap thread on the middleware's startup, which dies dies after having done the following tasks in order:
   (a) Parse the command line arguments to get the middleware's IP and port to listen on, the number of worker threads, the IP and ports of memcached servers, and whether to process multi-get requests in a sharded or non-sharded manner.
   (b) Create a request queue (a thread-safe blocking queue) to be shared among the network thread and all worker threads.
   (c) Create and start a number of `WorkerThread` threads, a `NetThread` thread, and a `StatCollector` thread, all of which sharing the same request queue created in the last step. The StatCollector thread is scheduled to collect the system's statistics data once a second.
   (d) Create an `ExitHandler` thread, and register it as a ShutdownHook of the JVM. That is, the ExitHandler will be and only be started when the virtual machine is going to shut down (for example, when the user sends a SIGINT signal by executing `kill -2 $PID` to shut down the middleware).

   After the main thread dies, and before the exit handler is executed, the middleware always

has a network thread and a fixed number of worker threads running, plus a statistics collector thread started once a second.

2. `RequestQueue` - A wrapper of `java.util.concurrent.LinkedBlockingQueue`. An object of `RequestQueue` is shared among all worker threads, the network thread and the statistics collector thread. This queue maintains a total number of arrived requests on every `enqueue()`, which helps to compute the arrival rate in the `StatCollector` thread. Every time the method `getArrivalRate(duration)` is called, it returns the arrival rate calculated from the increment of arrivals since the last calculation, divided by `duration` seconds, which is the time elapsed since the last call. The queue also marks the enqueue time and dequeue time for a `Request` when it is enqueued or dequeued.
3. `Request` - A class representing the requests from clients. It keeps the following information of a request:
   - the request content as a `String`,
   - the request type as an `enum RequestType`,
   - the number of keys in the request (only applicable for `GET`s and `MULTI-GET`s),
   - all tokens of the request content split by spaces (only applicable for `MULTI-GET`s),
   - the `SocketChannel` of the client from which this request is sent, which is simply a reference to the `SocketChannel` used in the network thread,
   - six timepoints in microseconds (μs): when this request has completely arrived (`timeArrived`), when it has been enqueued by the network thread (`timeEnqueued`), dequeued by a worker thread (`timeDequeued`), sent to all corresponding servers (`timeSentToServers`), when the worker thread has received response from all servers (`timeServersResponded`), and when the worker has sent a response to the client (`timeSentToClient`).
4. `NetThread` - The network thread (`NetThread`) is the thread that listens on a local port, accepts incoming client connections to this port, reads **and parses** requests from clients and put them into the request queue. As one request may be split into more than one network packets, the network thread only enqueues a request when all packets of this request have been received. In order to recognize a completion of a request, the network thread needs to do parsing after each reading. It also parses a request to get some statistical information necessary for measuring instruments. The detailed parsing procedures are described in 1.2.3.
   The communication functionality is implemented using `java.nio`. A `ServerSocketChannel` is created to listen for incoming connections, and one `SocketChannel` is established for each client connection. To listen for messages on all of these connections in a single network thread, all channels are registered to a `Selector` to perform non-blocking IO, selecting for `Acceptable ServerSocketChannel` or `Readable SocketChannel`s. When registering to the selector, every `SocketChannel` to client is attached with a `ByteBuffer`, which will be reused every time there is some message to be read on this channel. This avoids the overhead of allocating and deallocating `ByteBuffer`s on every message.
   When the `ServerSocketChannel` is acceptable, which means there is a new client trying to connect to the middleware, a new `SocketChannel` is created to the new client. When a `SocketChannel` is readable, which means the client has sent a new message, the current timestamp is marked and `handleRead()` is called to parse the message; if the message forms a complete request, a `Request` is created and enqueued into the request queue with the marked arrival time. The parsing procedure is described in detail in 1.2.3.

II. Request Processing

1. `WorkerThread` - The worker thread, which is responsible for establishing its own socket connections to the memcached servers on start up, and extracting and processing requests one by one from the request queue using its own `RequestProcessor` instance. After a worker thread has successfully taken a request, it will first look at the request's type. If it is a `SET` or a `GET`, the worker will forward the request to one or more servers, wait for response(s), and send a final response back to the client. Otherwise, the worker will log this event and go on taking a next request without responding the client on this request. A worker thread will only take a next request after it has finished processing the current request. The detailed request processing procedures are described in 1.2.4. The thread can be `interrupted` (e.g. when the shutdown hook is triggered), in which case it will stop taking new requests and terminate.
2. `RequestProcessor` - An abstract class providing abstract method definitions and partial implementations for the request processing in a worker thread. There are two classes extending this abstract class: `NonShardedRequestProcessor` and `ShardedRequestProcessor`, between which the only difference is how `MULTI-GET` requests are processed. The methods for processing `SET`s and single-key or non-sharded `GET`s are implemented in `RequestProcessor` itself.
   Depending on the `readSharded` argument specified at startup, each worker thread creates its own `RequestProcessor` instance. The request processor is in charge of sending requests to all servers or in a round-robin manner, possibly re-assembling the request first (for the case of sharded `MULTI-GET`), as well as receiving responses from servers and sending responses to clients. The Socket IO with servers are wrapped by `BufferedReader`s and `PrintWriter`s. Sending to clients is backed by a single `ByteBuffer`, which is reused every time the request processor executes a `sendToClient(client, msg)`. Section 1.2.4 elaborates on how each type of requests is processed.
3. `NonShardedRequestProcessor` - The request processor extending `RequestProcessor` which implements `MULTI-GET` processing in a non-sharded way, which means multi-key `GET`s are tackled in exactly the same way as single-key `GET`s.
4. `ShardedRequestProcessor` - The request processor extending `RequestProcessor` which implements `MULTI-GET` processing in a sharded way, which means multi-key `GET`s are splitted and distributed among more than one memcached servers, and responses are merged again into one before being forwarded to the client. As the number of keys in each shard can be different, it also tries to balance the number of keys distributed to every server, by maintaining a priority queue of servers based on current total load, polling from the queue to get the least-loaded server, and sending the largest shard first. An empirical proof of balance is presented in 1.2.5.

III. Measuring Instruments

1. `Statistics` - Realtime statistics data with regard to all requests completed by one worker thread. Every worker updates its own `Statistics` on completion of every request, including the number of `SET`s, `GET`s, `MULTI-GET`s, total number of keys and misses in `GET`s and `MULTI-GET`s, the sums of waiting time, service time, response time and internal processing time (which equals to `response` − `service` − `waiting`) for each type, and the response time histograms for each type. Besides, after a thread has dequeued a request from the request queue, it will call `markQueueLength(oldLength)` to add to an accumulator the product of the old queue length and the time elapsed since this worker's last dequeuing, from which the average queue length estimated on this worker is derived later according to the definition of average queue length.
   The statistics collector thread will read every worker thread's `Statistics` once a second by calling `getThreadLog(duration)`, which will return a `ThreadLog` object containing the

increments of all the sums, average throughputs of all request types and average queue length in the period between this call and last call, assuming the interval is `duration` seconds.

2. `ThreadLog` - The class of objects returned by `Statistics.getThreadLog(duration)`. It contains the statistics data of a worker thread within a period, as described in section 1.
3. `PeriodLog` - Created in the statistics collector thread to hold the `ThreadLog`s of a one-second period from all worker threads, as well as the arrival rate of the period, which is measured in the `RequestQueue`. Includes a `getMergedLog()` method, which returns a merged `ThreadLog` aggregating all workers' `ThreadLog`s it is holding. This method is called in the exit handler, when it is dumping the final statistics.
4. `StatCollector` - The body of the statistics collector (`StatCollector`) thread, a helper thread scheduled in a fixed rate (one second) by `ScheduledExecutorService`. Every time it is run, a new `PeriodLog` instance marked by the current second is created, and all workers' `ThreadLog`s generated by `Statistics.getThreadLog(period)`, as well as the arrival rate measured on the request queue, are put into the `PeriodLog`. The statistics it gathered are kept in the main memory, not outputted to the standard output until the exit handler thread does so before termination.
5. `ExitHandler` - The shutdown hook thread, registered as the JVM's `ShutdownHook` at startup. It is only run when the middleware is exiting, either by a user-pressed CTRL+C or by a SIGINT signal received by the JVM. It first interrupts the network thread and all worker threads, wait for all of them to stop, and then shutdown the executor of statistics collector. Before exiting, it dumps to the standard output the following statistics:
   (a) All workers' statistics gathered by the statistics collector thread every one second, with a merged total statistics of every second.
   (b) A total statistics within the middleware's whole execution phase, merged from the statistics of all seconds, including the total number, average response time, maximum throughputs, average sizes, hits, misses, and miss rates (if applicable) for each type of requests.
   (c) Three response time histograms, in 0.1 millisecond steps, for `SET`, `GET`, `MULTI-GET` respectively, which are merged from the histograms of all threads.
   (d) All errors and exceptions occurred during the middleware's execution time.

1.2.2 Thread Pool

The network thread and worker threads in the middleware are manually created and started in the main thread, **without** using Java built-in `ThreadPool` or `Executor`. In case of failure on any thread, the whole system simply shuts down and return a `-1`, instead of creating a new thread to replace the failed thread. This keeps the system simple without compromising the required functionality.

The statistics collector thread is scheduled with Java's `ScheduledExecutorService`, which makes it easier to do the statistics collecting at a fixed rate. As the statistics collecting operation is trivial and fast, and we are starting the operation at the beginning of every second, we can safely assume that the interval of collecting operations is exactly one second.

1.2.3 Message Parsing

Message parsing happens at two places in the middleware. The first is when a message from client is received in the network thread, as we need to know if it is a complete request or not, we have to parse the message. Additional information that is useful later, e.g. number of keys, tokenised commands, are also parsed and stored here by the way. The second is by the request processors of worker threads when reading responses, because we want to know if the full response has been received, how many keys are missed, or if any error has occurred.

According to the memcached protocol, a request contains either a command line followed by a data block (for storage commands), or a command line only (for all other commands). Both the command line and the data block are terminated by "\r\n". Similar formats applies for responses: for `GET` responses, there are zero or more items and an "END\r\n" in the end, where an item is a command line followed by a data block, both terminated by "\r\n"; for `SET` responses, there are only four possibilities: "STORED\r\n", "NOT_STORED\r\n", "EXISTS\r\n", "NOT_FOUND\r\n". We do not care about the response formats for other types of requests, since we only relay `SET`s, `GET`s and `GETS`es to memcached servers.

Parsing a request message involves the following steps. The assumption is that one read can get partial, exactly one, or more than one requests (which will not actually happen from memtier clients). So, after the network thread reads the message into the `ByteBuffer` attached to the corresponding `SocketChannel`, it repeatedly checks if the buffer contains a complete request, convert that part into a string, and move the remaining bytes to the beginning of the buffer with `buffer.compact()`. On a next `read()`, the new message will be put in the same buffer after the existing bytes, and the whole buffer is checked again from its beginning for complete requests. After each read, we keep extracting requests from the buffer until it is empty or only contains a partial request. Then, the parsing on this buffer is stopped until a new `read` into this buffer is performed.

To check for a complete request, we decode the whole buffer into a `String`, and look for the first "\r\n" in the string. If it is not found, which indicates a incomplete request, the parsing stops. Otherwise, the string is split into space-separated tokens, and we the first token indicates its type. If it is a storage command[2], we then parse the fifth token as an integer, which indicates the size of the following data block, and we expect (size + 2) more bytes in the string (because of the terminating "\r\n"). Otherwise, we do not expect more bytes, but we still check if the command is a `GET`, and, if it is, count the number of keys in it. Finally, we put the request into the request queue, along with its type, number of keys, `SocketChannel` to client, and, if it is a multi-key `GET`, the split tokens we have already got from parsing, so that we do not have to tokenize the request message again for sharding.

The parsing of server responses is similar, but it does not involve directly manipulating a buffer, because we use `java.io.BufferedReader.readLine()` to get a line string directly. Parsing `SET` responses is trivial, as we only need to read one line. For `GET` or muti-key `GET` responses, first we read one line to see if it is an "END\r\n" (we call it an `end line`). If it is not, it must be the text line of an item. Then we tokenize the line and read the data block size from the fourth token, and call `BufferedReader.read()` for (size + 2) times to get the complete data block and terminating characters. This process is repeated until it has read an end line. For responses to multi-get shards, the only difference is that the end line is discarded. All shard responses are combined with a final end line appended. Note that the line returned by `BufferedReader.readLine()` does not include the carriage return and line feed characters in the end, so they are appended to the end manually.

The above parsing procedure is designed to be correct and robust, but not extremely efficient. It could cause performance bottlenecks, which we will discuss in later chapters.

1.2.4 Request Processing

In the middleware, every request is processed by exactly one worker thread, by calling the corresponding request processing method of the worker's `RequestProcessor`. When `readSharded` is `true`, every worker thread creates a `ShardedRequestProcessor` sharing the worker's `Statistics` and `Socket`s to servers. Otherwise, a `NonShardedRequestProcessor` is created in the same way.

[2]Storage commands in memcached includes `set`, `add`, `replace`, `append` and `prepend`. Although we actaully only process `SET` commands, it is good to know the category in order to apply different parsing procedures.

Both of the two classes extend the `RequestProcessor` class, which implements the processing of `SET`s and `GET`s. The only difference is the implementations of `processMultiGet` method. Thus, below we introduce request processing in four cases: `SET`s, single-key `GET`s, non-sharded multi-key `GET`s and sharded multi-key `GET`s.

For a `SET` request, the `processSet` method of the request processor is called after the request is dequeued and the attribute `timeDequeued` is set in the request. In `processSet`, the following steps are executed:

1. The request is sent to all servers by calling `sendToServer(serverId, requestMessage)` in a `for` loop.
2. The attribute `timeSentToServers` is set in the request.
3. The responses of all servers are read one by one by calling `readers[serverId].readLine()` in a `for` loop, as a `SET` response always consists of only one line.
4. The attribute `timeServerResponded` is set in the request.
5. Every response is checked if its content is "STORED", which means no error has occurred. If so, set the final response to reference the first response, otherwise, set it to reference the first non-STORED response.
6. A "\r\n" is appended to the final response.
7. The final response is sent to the client by calling `sendToClient(channel, response)`.
8. The attribute `timeSentToClient` is set in the request.

For a single-key `GET` request or a non-sharded `MULTI-GET` request, the `processGet` method of the request processor or the `processMultiGet` method of `NonShardedRequestProcessor`, respectively, is called, after the request is dequeued and the attribute `timeDequeued` is set in the request. Both methods in turn calls the `processGetImpl` method, which executes the following steps:

1. The server to send the request to is determined by calling `getAndIncrementNextServerIndex()`, which returns a next server index in a round-robin manner.
2. The request is sent to the determined server by calling `sendToServer(serverId, requestMessage)`.
3. The attribute `timeSentToServers` is set in the request.
4. The response of the server is read by calling `readGetResponse(serverId)`, which follows the parsing procedure as described in 1.2.3. The number of misses is also updated in this process.
5. The attribute `timeServersResponded` is set in the request.
6. The final response is sent to the client by calling `sendToClient(channel, response)`.
7. The attribute `timeSentToClient` is set in the request.

After `processGetImpl` has returned, the number of missed keys is added to the worker's statistics.

For a sharded `MULTI-GET` request, the `processMultiGet` method of `ShardedRequestProcessor` is called, after the request is dequeued and the attribute `timeDequeued` is set in the request. The request is then processed in the following steps:

1. The number and size of shards are determined from the number of keys and servers: $\texttt{numShards} = \min(\texttt{numServers}, \texttt{numKeys})$, $\texttt{shardSize} = \texttt{numKeys}/\texttt{numShards}$.
   If $(\texttt{numKeys} \% \texttt{numShards}) \neq 0$, the sizes of first $(\texttt{numKeys} \% \texttt{numShards})$ shards are added by one.
2. Each shard of request is assembled from tokens into a string, and sent to the lowest-loaded server polled from the priority queue as described in 1.2.1: `ShardedRequestProcessor`.
3. The attribute `timeSentToServers` is set in the request.
4. The response of each involved server is read and appended to a `StringBuilder` for the final response in the same order as in sending, by calling `readMultiGetResponse(serverId)`,

which follows the parsing procedure as described in 1.2.3. The number of misses is also updated in this process.

5. The attribute `timeServersResponded` is set in the request.
6. The final response is sent to the client by calling `sendToClient(channel, response)`.
7. The attribute `timeSentToClient` is set in the request.

1.2.5 Proof of Balanced Load Distribution for GETs and multi-GETs

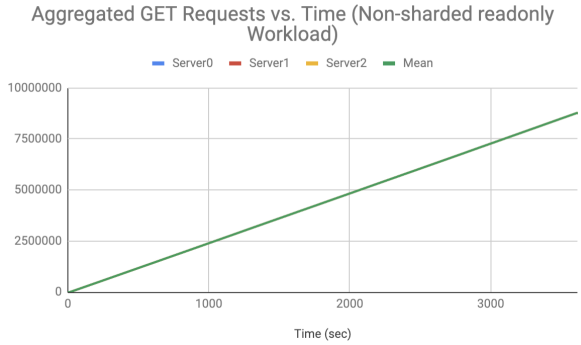

Figure 1.2: Load Distribution of GET
(the lines are too close to be distinguished)

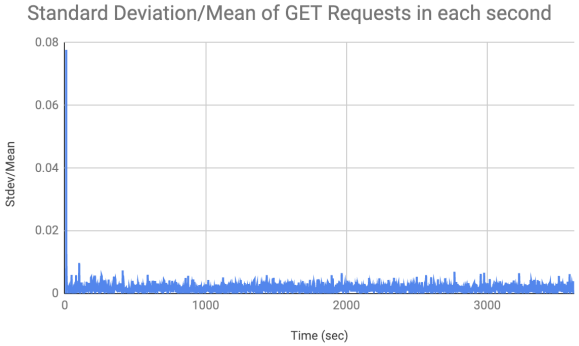

Figure 1.3: Standard Deviations of GET Load Distribution Per Second

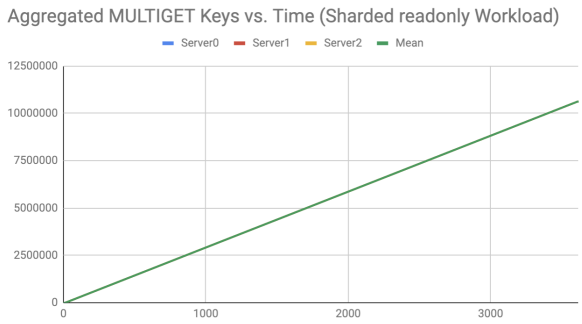

Figure 1.4: Load Distribution of MULTIGET
(the lines are too close to be distinguished)

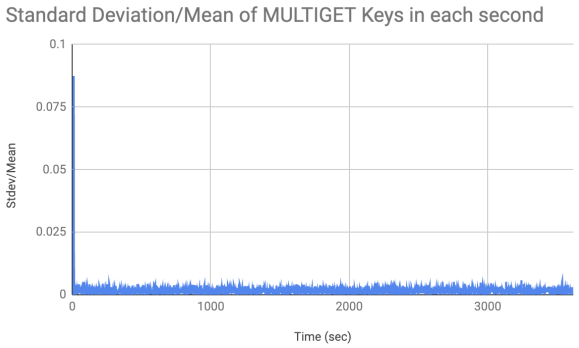

Figure 1.5: Standard Deviations of MULTIGET Load Distribution Per Second

The middleware tries to balance load among servers. This is trivial for `SET`s because they are naturally sent to all servers. For `GET`s, using a simple round-robin mechanism in each worker thread independently is sufficient, assuming that every worker has equal chance of dequeueing a request from the LinkedBlockingQueue. However, it is not easy to tell from the implementation of Java `LinkedBlockingQueue` whether it will release objects evenly among threads. Further, for `multi-GET`s, the shards may not be equal in size, and the request processor always generates larger shard first, therefore, simple round-robin can always distribute the largest shard to some server, resulting in an imbalance. Instead, a priority queue based on current number of keys of each server is used to get the lowest-loaded server every time when deciding each server to send.

To verify the balance distribution for `GET`s and `multi-GET`s, two one-hour readonly experiments was run with the following settings.

Figure 1.2 is the total number of `GET`s sent to each server as a function of total request numbers. The numbers at the end of the experiment are 8,768,473 for server 0, 8,768,443 for server 1, 8,768,424 for server 2, with a total of 45,812,372, an average of 8,768,446.667 and a standard deviation of 24.70. Figure 1.3 is the standard deviations of the three servers'

Number of servers	3
Number of client machines	1
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	32
Workload	Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1
Worker threads per middleware	64
Repetitions	1
Test time (seconds)	3600

Table 1.2: Testing GET Distribution

Number of servers	3
Number of client machines	1
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	32
Workload	Read-only
Multi-Get behavior	Sharded
Multi-Get size	10
Number of middlewares	1
Worker threads per middleware	64
Repetitions	1
Test time (seconds)	3600

Table 1.3: Testing MULTIGET Distribution

load divided by the mean load at each second. Since the standard deviation is no more than 0.1% of the average, we can safely argue that the load for `GET`s is distributed evenly among servers. Similarly, Figures 1.4 and 1.5 are for sharded `MULGIGET`s, with 10,620,038 keys for server 0, 10,620,013 for server 1, 10,619,995 for server 2, on average 10,620,015.33, and standard deviation 21.59, except that the units are keys instead of requests.

Therefore, the `get` load is equally distributed among all servers, and the system is stable within a one-hour run.

2 Baseline without Middleware (75 pts)

In this section we study the performance characteristics of the memtier clients and memcached servers.

2.1 One Server

We aim to show how the behavior of one single server changes as we add more clients, with the setup listed below. Figure 2.1 is the response time measured by memtier with regard to the number of clients, for both read-only and write-only workloads. Figure 2.2 is the corresponding throughput, along with the theoretical throughput from the number of clients and measured response time by the interactive law. By comparing it with the measured throughput, we can see that the interactive law holds in these 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	[1, 4, 8, 12, 16, 20, 24, 28, 32] For write [36, 40, 44, 48] in addition
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	3 × 80 seconds each

2.1.1 Explanation

For readonly workload, the system hits a maximum throughput of 2912.41 ops/second and becomes saturated even when there is only one virtual client in each memtier thread. No under-saturated phase is observed; all points after are over-saturating the system since the throughput does not change and the response time grows linearly with the number of clients. This is because the sending bandwidth of the server VM is limited at 100 Mbits/s, or 11.92 MiB/s, when the value size is 4096 Bytes (or full response size 4130 Bytes due to header and

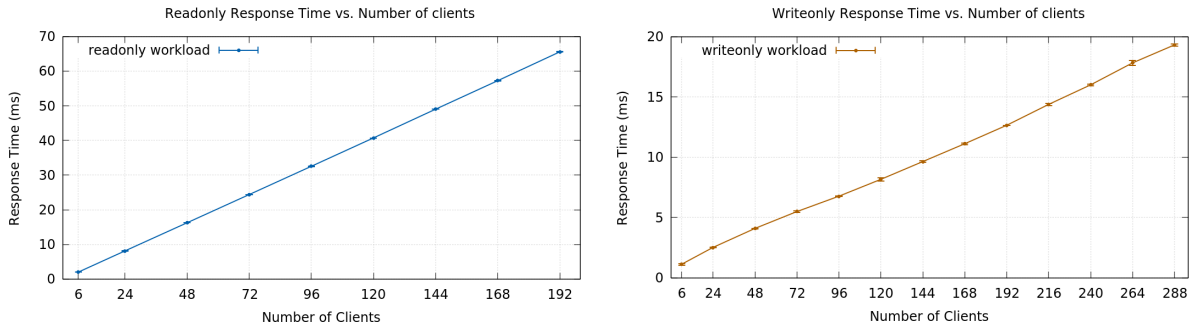

Figure 2.1: Measured Response Time as a function of NumClients for One Server
(Error bars are too small to be distinguished)

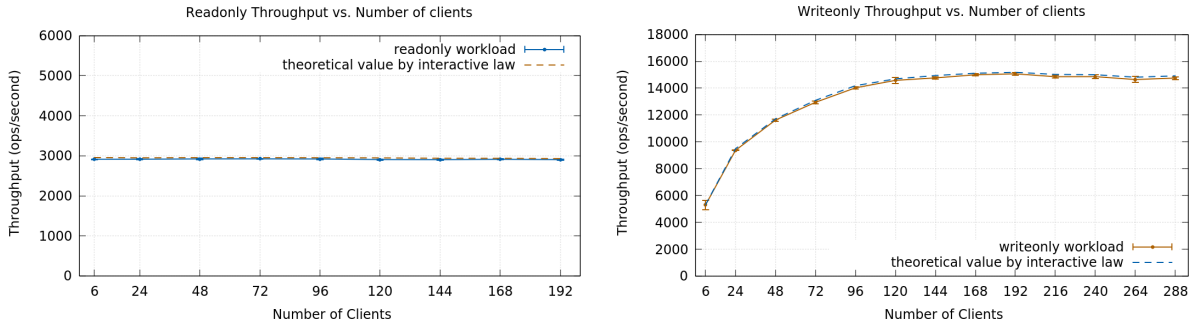

Figure 2.2: Measured Throughput as a function of NumClients for One Server
(Error bars are too small to be distinguished)

termination characters), this yields a maximum sending capacity of 3026.63 ops/second. We do not see 3000 on the plot because the results are actually average throughputs over the whole running period, which is not necessarily the maximum throughput at any second. However, we can still see from the server's `dstat` log that it indeed keeps sending a traffic of 12∼13MB at each second, which proves that the bottleneck is really the server's uploading bandwidth.

For writeonly workload, when increasing clients from one per memtier thread (or 6 clients in total), the system gradually changes from under-saturation to saturation, and it becomes fully saturated at 120 clients with 14578.22 ops/sec, because there is barely no growth in throughput when adding more clients, while the response time grows linearly with the number of clients. The `dstat` log from server also supports this observation: when there are 120 clients, the CPU load becomes nearly 100%, which means that the server has been fully utilized, and it has no more CPU power to spend.

Therefore, we conclude that for readonly workload, a server can support up to about 3000 ops/second, which is limited by its network sending bandwidth. For writeonly workload, a server can support up to about 15000 ops/second, which is limited by its CPU processing capacity.

2.2 Two Servers

Now we use two servers to test the capacity of a single load generating machine, with the experiment setup below. Figure 2.3, left is the response time measured by memtier with regard to the number of clients, for both read-only and write-only workloads. Figure 2.3, right is the corresponding throughput, along with the theoretical throughput from the number of clients and measured response time by the interactive law. By comparing it with the measured throughput, we can see that the interactive law also holds.

Number of servers	2
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1, 2, 3, 4, 5, 6, 8, 16, 32]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	3 × 80 seconds each

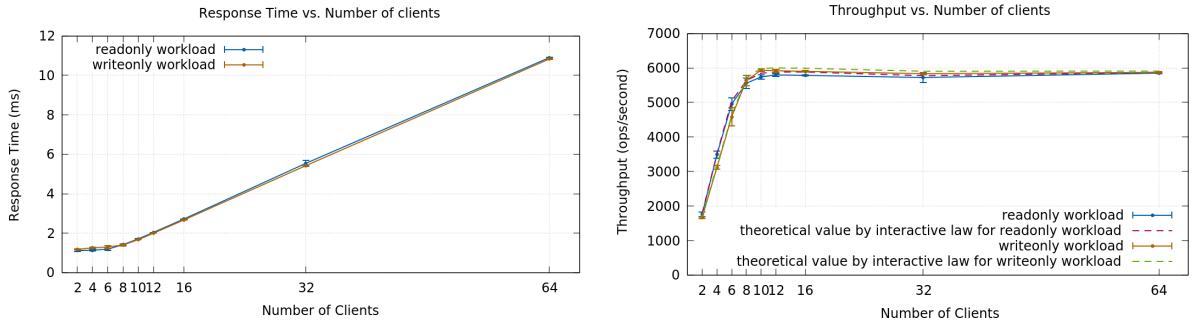

Figure 2.3: Measured Performance as a function of NumClients for Two Servers
(Error bars are too small to be distinguished)

2.2.1 Explanation

The behaviours for readonly and writeonly workloads look very similar, as they both become saturated at 8 clients, reaching a throughput of about 5800 ops/second, and the response time grows linearly afterwards, without considerable growth in the throughput. However, the bottlenecks that cause saturation are really different.

For readonly workload, the maximum throughput is around two times the value in readonly experiments of the previous section, and the key reason is the same: the two servers send 4130 Bytes responses to clients, which makes its 100 Mbits/s sending bandwidth (or 200 Mbit/s in total) a bottleneck, limiting the maximum throughput at 6053 ops/second. The client machine is not a bottleneck, neither in CPU nor in network, which can be seen from its `dstat` logs.

However, for writeonly workload, it is the client that sends 4096 Bytes values to the servers (or 4128 Bytes requests). Since the sending bandwidth of a client machine is ∼200 Mbits/s, or 23.84 MiB/s, it limits the maximum writeonly throughput at 6056 ops/second.

Again, the maximum throughput on the plot does not seem have a small gap with this value, this is because the plotted throughputs are averages over the 60 seconds experimenting period, which are not the maximum throughput at any second. Moreover, the throughput suffers from unstable network when the network bandwidth becomes fully utilized, possibly due to the management events of Azure VM hosts. This also lowers the average throughput over 60 seconds. But we can still prove network saturation from the servers' `dstat` logs that the sending speed is almost always 12∼13 MiB/s on each server (for readonly) or 23∼24 MiB/s on the client (for writeonly).

In conclusion, the readonly maximum throughput in this subsection is still limited by server bandwidth, thus it does not reveal the readonly capacity limit of the client machine. For writeonly workload, we successfully discovered the limit of the client machine, which has been bottlenecked by its sending bandwidth.

2.3 Summary

Maximum throughput (ops/sec) of different VMs.

	Read-only workload	Write-only workload	Configuration gives max. throughput
One memcached server	2912.41, $\sigma = 7.83$	14578.22, $\sigma = 234.82$	VC = 6 for readonly, VC = 120 for writeonly
One load generating VM	5558.06, $\sigma = 152.19$	5637.51, $\sigma = 152.90$	VC = 8 for both

The one server experiments successfully show the limit of one memcached server under both read-only and write-only workloads, by using all three client machines to generate as much load as possible. On the other hand, the two servers (one client) experiments try to show the limit of one load generating machine, with the hope that as many as two servers will not be the bottleneck, but unfortunately the result shows this is not always true.

In one server experiments, the bottleneck is always the server. For read-only workload, it is limited by a network sending bandwidth of 100 Mbits/s. For write-only workload, the network is no longer the bottleneck: it is limited by its CPU capacity. In two servers experiments, the bottleneck is still the servers' sending bandwidth for read-only workload. For write-only workload, instead, it is limited by the client's sending bandwidth of 200 Mbits/s. Although the maximum throughput for both workloads is the same, the bottleneck under the hood is different.

The read-only maximum throughput for one client experiments is roughly two times that of one server (with standard deviation considered), because for read-only experiments the bottleneck is always the server, and we have doubled the number of servers in one client experiments. The write-only maximum throughput for one client experiments is much lower than one server, because only one client machine is not enough to fully utilize the CPU of one server; the upload bandwidth of the client itself has become the bottleneck.

The key take-away messages about the behaviour of a memcached server is that its CPU capacity limits its write-only throughput to at most about 15000 ops/sec, and its sending bandwidth (11.92 MiB/s) limits its read-only throughput to at most about 3000 ops/sec. For a memtier client machine, its sending bandwidth (23.84 MiB/s) limits its write-only workload throughput to a maximum of about 6000 ops/sec; however, its limit on read-only workload is unknown because we did not use enough many servers to reach that.

3 Baseline with Middleware (90 pts)

In this section, we study the performance characteristics of the system with one or two middlewares added between memtier and memcached.

3.1 One Middleware

In this subsection, we study the performance characteristics of one middleware, with three load generating machines and one memcached server. We compare it with baselines in Section 2.1, which has a similar setup except that it has no middlewares involved. Below is the experiment setup.

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[1,2,3,4,6,8,16,32]; For write-only [24] in addition
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1
Worker threads per middleware	[8,16,32,64]
Repetitions	3 × 80 seconds each

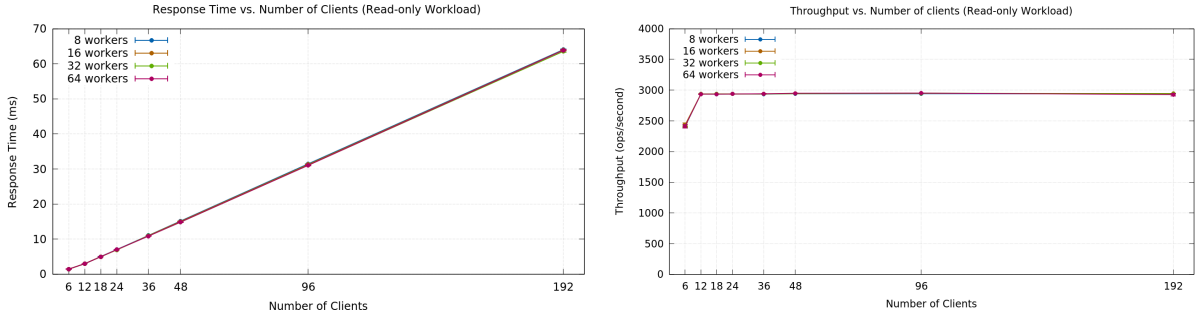

Figure 3.1: One Middleware Read-only Response Time and Throughput Measured on Middleware (Error bars are too small to be distinguished)

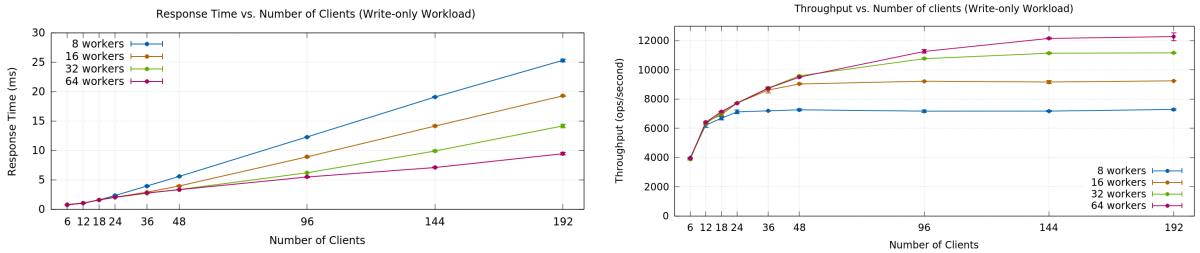

Figure 3.2: One Middleware Write-only Response Time and Throughput Measured on Middleware (Error bars are too small to be distinguished)

Figure 3.1 is the response time and throughput measured by the middleware with regard to the number of clients, for read-only workload. Figure 3.2 is for write-only workload. By calculating the theoretical throughput from the number of clients and the memtier-measured response time, and comparing it with the memtier-measured throughput, we confirm that **the interactive law holds** for memtier clients.

However, for the middleware, the interactive law does not automatically hold by simply using the number of memtier virtual clients and the middleware-measured response time, because the middleware-measured response time is not the of the whole system, but of the sub-system where the middleware works as clients and memcached works as server. There exist two ways to fix this: either to adapt the number of clients, which is smaller than the number of memtier clients, or to add a think time, because between one request is completed and the next request is ready, there is a time gap, which could be contributed by various factors, including network latency and the waiting time when the network thread is too busy.

For the first approach, the real number of clients (requests) in the sub-system should be the number of waiting-in-queue requests (N_q) plus the number of requests being served (N_s). N_s is equal to the number of "active" threads, which can be less than the number of worker threads, possibly due to middleware saturation or insufficient number of requests. If we take the active worker threads and the server as another sub-system, by definition the number of active workers

is the product of throughput and service time measured at the middleware. Therefore, using $N_q + N_s$ as the number of clients, the interactive law is verified to hold for the middleware.

For the second approach, we take the response time difference between memtier and middleware measurements, which is exactly the network round trip time plus any waiting time before a request is read into the network thread, as the think time Z, and the interactive law is again verified to hold for the middleware.

3.1.1 Explanation

The results are consistent with the baseline measurements in Section 2.1, in that the maximum throughputs are always lower than the case without middleware, and response times are a bit higher due to the added network and processing latency.

For read-only workload, the saturation point, which is 12 clients, comes later than the 6 clients in Section 2.1, because the network latency is roughly doubled as we now have an additional node in the middle, making it not enough to be hidden with just 6 clients. But as soon as we have enough parallelism as 12 clients, the system is again saturated, with the response time growing linearly while throughput staying unchanged, due to the bottleneck at the memcached server's sending bandwidth. The middleware never becomes the bottleneck, as we can see from Figure 3.1, even 8 workers are able to achieve the maximum throughput of 2937.96 ops/sec.

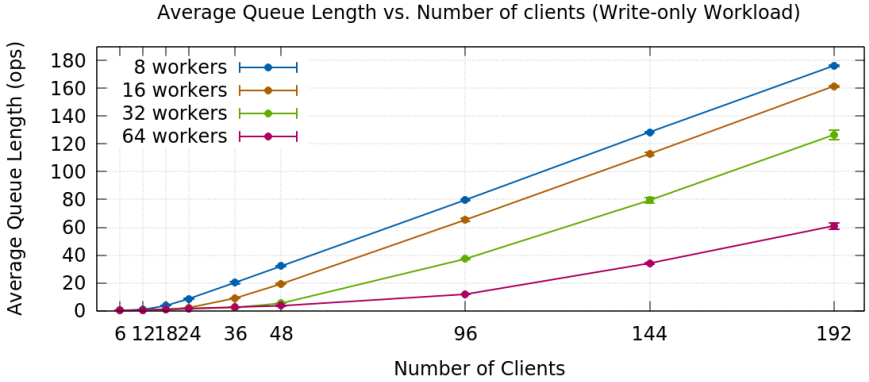

Figure 3.3: Average Queue Length for One Middleware Write-only Workload
(Error bars are too small to be distinguished)

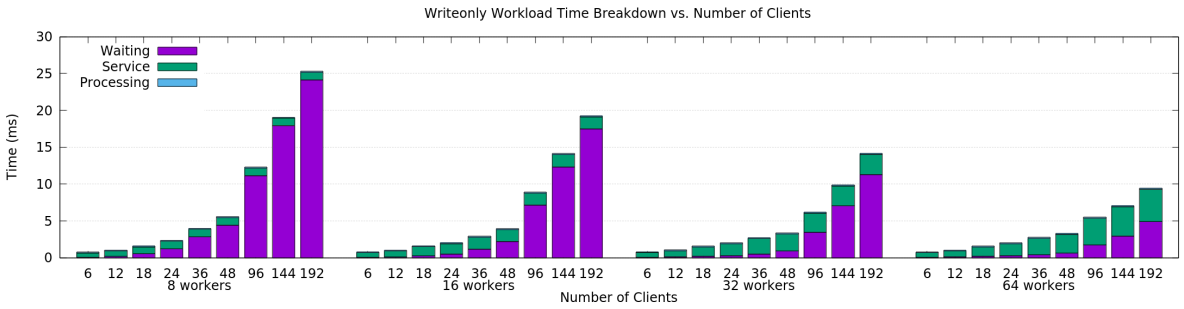

Figure 3.4: Middleware Time Breakdown for One Middleware Write-only Workload
(Errors are too small to be seen)

For write-only workload, as we increase the number of worker threads, it requires more and more clients to saturate the middleware, because more and more requests can be processed at the same time. Also, we can see that as we add more workers, the performance gain becomes less, because the machine's parallel processing capacity is limited, and adding more threads also

introduce more scheduling overhead and resource contention. With the support of Figures 3.3 and 3.4, we can see the worker threads begins to become saturated at 12, 24, 48, 96 clients respectively for 8, 16, 32, 64 worker threads, as both the average queue length and the in-queue waiting time start to scale super-linearly; the middleware becomes fully saturated at 24, 48, 96, 144 clients respectively for 8, 16, 32, 64 threads, as the maximum throughput (7126.27 ops/sec, 9036.63 ops/sec, 10767.83 ops/sec, 12154.73 ops/sec, respectively) has been reached while response time grows linearly. In total, the system reaches the maximum throughput at 144 clients, 64 threads, with 12154.73 ops/sec. The maximum processing capacity of the memcached server is never reached, because now the bottleneck is in the middleware.

It is worth noting that the bottlenecking component of the middleware can be not only the worker threads, but also the network thread. An important phenomenon we have observed is the response time discrepancy between memtier and middleware measurements. When the middleware is not saturated, we find the difference value is simply equal to the round-trip time between memtier and middleware machines measured by `ping`, just like the case in Figure 3.5 left, which is trivial to understand. However, further observation, as plotted in Figure 3.5 right, shows that when the middleware is saturated, the discrepancy has a positive correlation with the number of workers, and when the number of worker threads is high, the discrepancy even grows linearly with regard to the number of clients.

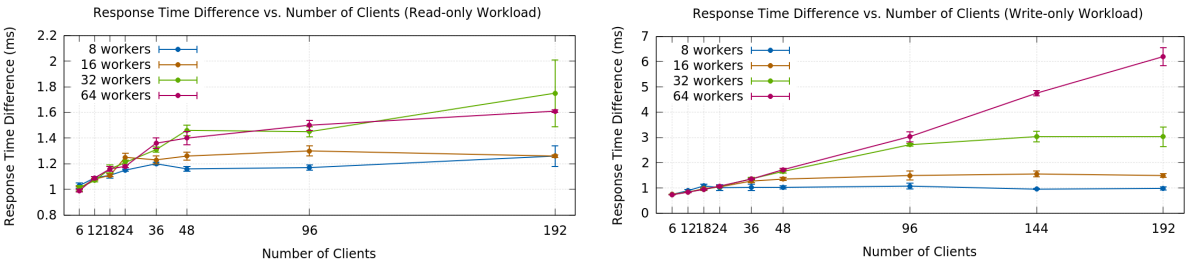

Figure 3.5: Response Time Discrepancy Between Memtier and Middleware Measurements

When the middleware is saturated, the discrepancy is not only the RTT. First, adding worker threads increases the throughput, which is also the arrival rate, due to the nature of a closed system. As the arrival rate reaches the processing capacity (service rate) of the network thread, new requests will fill up the operating system network queue, and the network thread cannot immediately handle the `isReadable` event and read it into the middleware because it may be still parsing the last request. Also, when there are more worker threads, it is more likely to have resource contention since there are only eight cores in the middleware VM, and the network thread could become less responsive due to thread scheduling. So, the requests will have to spend more time waiting in the OS network queue, and this part of waiting time cannot be recorded by the measuring instruments of the middleware, resulting in a larger discrepancy between the measured response times of memtier and the middleware.

3.2 Two Middlewares

In this subsection, we double the number of middlewares from the previous subsection and study the performance change. The experiment setup is listed below. Figure 3.6 is the response time and throughput measured by the middleware with regard to the number of clients, for read-only workload. Figure 3.7 is for write-only workload. The throughput and response time measured by memtier (not shown here) follows the interactive law. Since we have two middlewares now, the pressure on the network thread is relieved, and it does not become the bottleneck, because by simply introducing the network RTT as think time, we confirm that the interactive law also

holds for the middleware.

Number of servers	1
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	For read-only: [1,2,3,4,6,8,16,32] For write-only: [1,2,4,6,8,16,24,32,36,40]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8,16,32,64]
Repetitions	3 × 80 seconds each

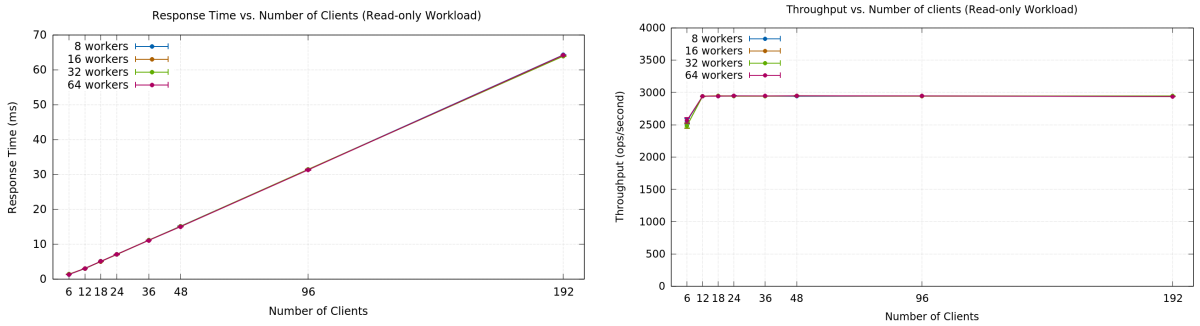

Figure 3.6: Two Middlewares Read-only Response Time and Throughput Measured on Middleware (Error bars are too small to be distinguished)

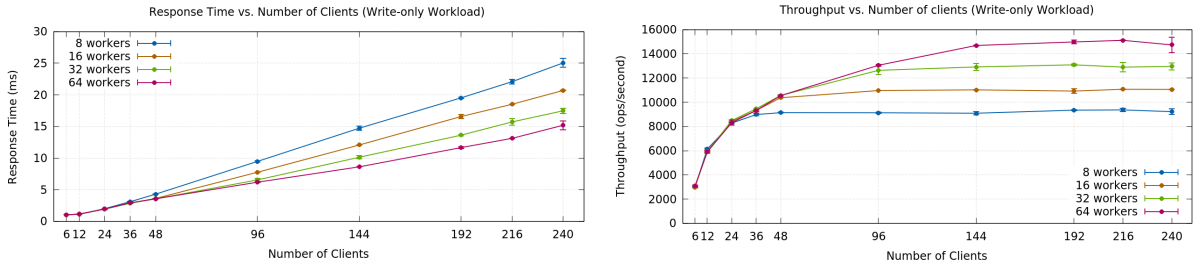

Figure 3.7: Two Middlewares Write-only Response Time and Throughput Measured on Middleware (Error bars are too small to be distinguished)

3.2.1 Explanation

For read-only workload, the number of worker threads makes no difference, again this is because the throughput is limited by the sending bandwidth of memcached server, and even 8 worker threads are enough to deal with it. For the same reason as explained in Section 3.1.1, the system is saturated at 12 clients, reaching a maximum throughput of 2941.21 ops/sec. The throughput does not increase when adding more clients, while the response time grows linearly.

For write-only workload, the system becomes saturated at 36, 48, 96, 144 clients for 8, 16, 32, 64 worker threads, respectively, because maximum throughput (8981.92 ops/sec, 10368.90 ops/sec, 12630.75 ops/sec, 14688.05 ops/sec, respectively) has been reached and adding clients only results in linear growth of response time. This conclusion is also supported by the middleware time breakdown and average queue lengths as in Figures 3.8 and 3.9, as from these points upwards, the average queue lengths begin to increase dramatically, so do the waiting times in the request queue. The service times are also increased with regard to the number worker threads, indicating the saturation of the memcached server when the load becomes higher. Here

we can argue that the bottleneck is not within the middlewares, because the pressure on a single middleware in the previous section has been shared by two middlewares, and with 64 worker threads and 144 clients, the system is able to reach a maximum throughput of about 15000 ops/sec, which is the limit of a memcached server as we have measured in Section 2.1. However, for 8, 16 and 32 threads, the maximum throughput is still limited by insufficient request processing capacity of the middleware.

In the previous subsection, the network thread became saturated when the arrival rate was too high, resulting in large response time discrepancies which is unmeasured by the middleware. However, in this subsection, this does not become a bottleneck because the load is distributed to two middlewares. The fact that the response time discrepancy between memtier and middleware measurements is equal to the `ping` latencies also supports this conclusion, showing that the waiting time in the OS network queue is about zero.

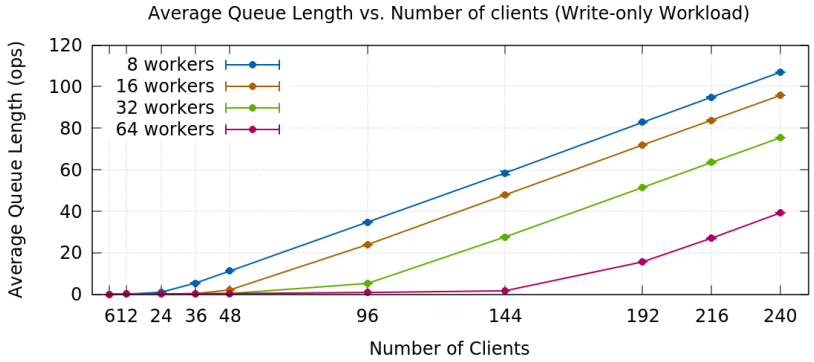

Figure 3.8: Average Queue Length for Two Middlewares Write-only Workload
(Error bars are too small to be distinguished)

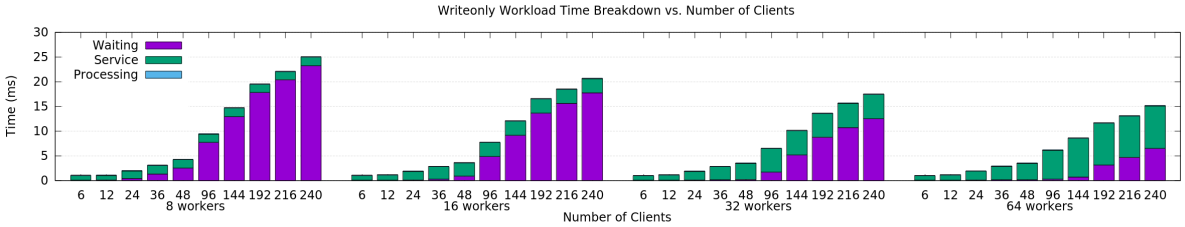

Figure 3.9: Middleware Time Breakdown for Two Middlewares Write-only Workload
(Errors are too small to be seen)

3.3 Summary

In the following tables, the maximum throughput experiment is selected so that it is the one that gives the maximum throughput with the least worker threads: 8 threads, 12 clients for read-only workloads in both tables; 64 threads, 144 clients for write-only workloads for one middleware; and 64 threads, 144 clients for write-only workloads for two middlewares.

Maximum throughput for one middleware.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware	2937.96	2.98	0.70	0
Reads: Measured on clients	2903.11	4.07	n/a	0
Writes: Measured on middleware	12154.73	7.09	2.95	n/a
Writes: Measured on clients	12023.66	11.84	n/a	n/a

Maximum throughput for two middlewares.

	Throughput (ops/sec)	Response time (ms)	Average time in queue (ms)	Miss rate
Reads: Measured on middleware	2941.21	3.03	0.15	0
Reads: Measured on clients	2908.47	4.07	n/a	0
Writes: Measured on middleware	14688.05	8.62	0.73	n/a
Writes: Measured on clients	14524.03	9.81	n/a	n/a

For read-only workload, as we can see from the result, both configurations reach the same maximum throughput with the same number of clients and same response time (with tolerable error). This is because the bottleneck is only at the server side: it cannot send at a higher bandwidth than 100 Mbits/s. It is not even able to saturate a single middleware with only 8 worker threads. A noticable difference is that the average time in queue is much lower in the two middlewares case, and the reason is that the load is distributed to two middlewares, relieving the pressure on a single middleware. In each middleware, the arrival rate is halved, there are less requests waiting in the queue, thus the waiting time is lower.

For write-only workload, two middlewares reach a higher maximum throughput than one. This is because in the one middleware case, the middleware really becomes the bottleneck. Both its network thread and worker threads are compromised: the large response time discrepancy indicates a saturated network thread, which cannot learn the arrival of a request immediately; the large average time in queue indicates saturated worker threads: a request has to wait longer until some worker thread become available to process it. However, with two middlewares, the load is distributed, and the bottleneck again moves back to the memcached server, because the throughput has reached the measured limit of one server in Section 2.1, the response time discrepancy is just the network RTT, and the average waiting time in queue is low.

The experiments in this section also showed us that increasing the number of worker threads, or in its essence, increasing parallelism in request processing, could improve the system's performance. In Section 3.2, we are not able to hit the limit of the memcached server with insufficient number of worker threads, even though we have two middlewares. Before the middleware is saturated, the more worker threads, the more clients we need to saturate the system, and the higher throughput we can get.

Furthermore, the additional plots on average queue length and middleware time breakdowns for work-only workload imply that when the system becomes saturated, its average queue length and waiting time will experience linear or superlinear growth with more clients, which can be used as a indicator of saturation.

4 Throughput for Writes (90 pts)

4.1 Full System

In this section, we investigate the write-only workload performance characteristics of the full system, with the setup listed below. Figure 4.1 is the response time and throughput measured by the middlewares with regard to the number of clients. The throughput and response time measured by memtier (not shown) is verified to follow the interactive law. Similar to the case in Section 3.2, the pressure on the network thread is relieved by doubling the middleware, so that it does not become the bottleneck, and we can confirm this by solely introducing the network RTT as think time, which turns out it indeed fits the interactive law at the middleware.

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1, 2, 4, 6, 8, 16, 24, 32, 36, 40, 44]
Workload	Write-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8, 16, 32, 64]
Repetitions	3 × 80 seconds each

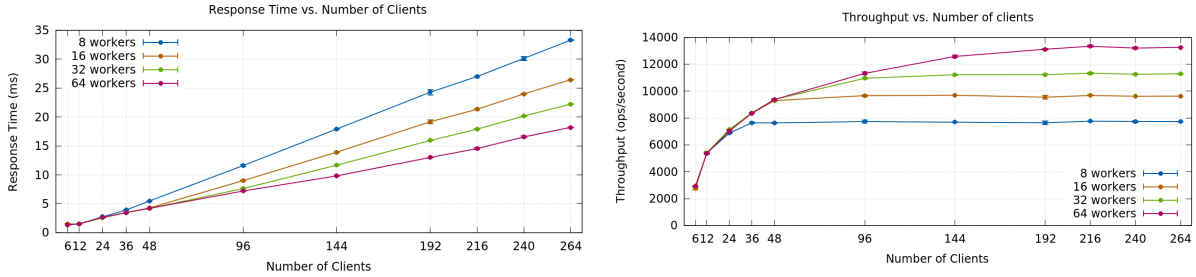

Figure 4.1: Full System Write-only Performance Measured on Two Middlewares
(Error bars are too small to be distinguished)

4.1.1 Explanation

The difference between this section and Section 3.2 is that the number of servers is increased to three. For write-only workload, we expect a throughput decrease introduced by this, because the middlewares will have to replicate the `SET` operation to every server, resulting in a increase in the time spent communicating with servers (the service time). The experiment results indeed match our prediction: the maximum throughputs (listed in the table of Section 4.2), which are hit at 36, 48, 96, 192 clients respectively for 8, 16, 32, 64 workers, are always lower than the maximum throughput with the same number of workers in Section 3.2; and the service times are always higher than those of Section 3.2, as shown by the all-positive values in Figure 4.2. To make it clear, the saturation points are chosen in such a way that further increasing clients will never introduce a throughput gain larger than 5%, while the response time will grow almost linearly. The steep scaling of average queue length when going beyond the chosen saturation points, as plotted in Figure 4.3, also supports our choice.

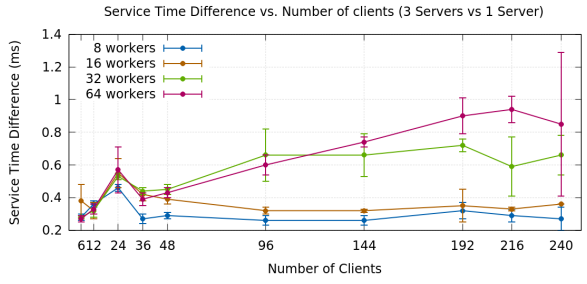

Figure 4.2: Service Time Differences
Between Sections 4 and 3.2

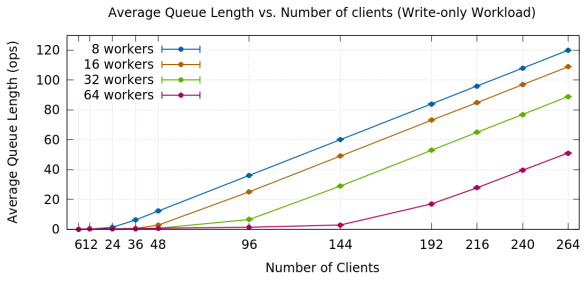

Figure 4.3: Average Queue Length
(Error bars are too small to be distinguished)

As the maximum throughput in this section never reaches the maximum write-only throughput that a server can handle (which was measured to be about 15000 ops/sec by Section 2.1 One Server baselines), the bottleneck is again not in the server side, but in the middleware itself.

This can be proved by the time breakdown as in Figure 4.4: when there are only 8 workers, most time is spent in waiting in the request queue; as the number of worker threads increases, the waiting time decreases because more requests can be processed in parallel, and the memcached service time increases because more load is put on the servers, resulting in longer processing time. If the bottleneck was the memcached servers, when the number of worker threads is fixed and the number of clients increases, the service time should also be increased due to server saturation, but it is not the case, because the bottleneck is the number of worker threads of the middlewares, which determines how many clients (requests) the memcached servers should deal with. We also find that the response time discrepancy between middleware and memtier measurements is equal to the network RTT, indicating that the waiting time in the OS network queue is about zero, thus the network thread does not become a bottleneck.

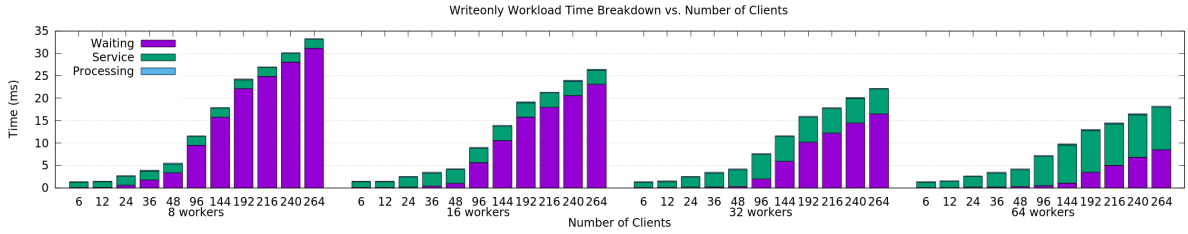

Figure 4.4: Middleware Time Breakdown in Two Middlewares (Write-only Workload)
(Errors are too small to be seen)

4.2 Summary

Maximum throughput for the full system

	WT=8	WT=16	WT=32	WT=64
Throughput (Middleware)	7645.00	9291.08	10954.73	13117.29
Throughput (Derived from MW response time)	7501.04	9275.36	11198.45	13587.20
Throughput (Client)	7562.64	9214.13	10842.41	12932.08
Average time in queue	1.79	1.02	1.97	3.45
Average length of queue	6.21	2.71	6.51	16.95
Average time waiting for memcached	1.97	3.08	5.46	9.32

To derive throughput from MW response time correctly with the interactive law, the mean `ping` latency is considered as the think time. Note that our gathered `ping` logs show that network latencies are relatively unstable, and the standard deviations in each single repetition are often 50%∼100% of the average latency, so using the average RTT as the think time is just a best-effort estimation, which can explain why the derived throughputs, while fitting the measured values fairly well, do have an error of less than 5% from the direct measurements.

For the 64 worker threads case, the average waiting time, service time and queue length is noticeably higher than the other cases, this should be because at this state the system is already saturated, and even an adequately smaller number of clients between 144 and 192 is also able to saturate the system, but our testing points are limited and did not include that precise number of clients. Increasing from that number to our chosen number of clients (192) have resulted in the increase of the aforementioned metrics. Similar argument holds for the 32 workers case, only that it is not as severe as 64 workers.

From the table and the data from previous experiments, we can see that the maximum throughput in each column is about 2000 ops/sec less than the maximum throughput in Section 3.2 with the same number of worker threads. Also, by comparing the average time waiting for memcached with Section 3.2 (i.e. the service time difference as compared in Figure 4.2),

it shows that adding more servers indeed introduces more overhead in the service time, thus lowering the throughput the system can reach. Further, we observe that the service time also has a positive correlation with the number of worker threads, because with more workers the middleware is able to push more load to the servers, and the servers will yield higher service time under heavier load.

By looking at the average length of queue and the average time in queue, we find that when the system reaches the maximum throughput, the averge queue length will be much larger than in a under-saturated state, because more requests will have to wait longer in the queue, as all workers are busy. On the other hand, Figure 4.3 shows that for the same number of clients, the more worker threads we have, the lower average queue length we get, despite that the throughput also increases. This is because with more workers, more requests will be under processing by the workers, instead of being waiting in the queue. With more workers, it will be more difficult to make the average queue length grow explosively when adding more clients.

5 Gets and Multi-gets (90 pts)

In this section, we measure the performance characteristics of the full system under mixed `GET-SET` workload. We experiment with 1, 3, 6, 9 keys for `get` requests, using memtier's `--multi-key-get=<Multi-Get size>` argument.

Meanwhile, we control the workload ratio by `--ratio=1:<Multi-Get size>`, so each memtier client will send interleaving `set` and `get` requests one after another, so that the total numbers of keys in `set`s and in `get`s matches the specified ratio, while the total numbers of requests for each type are the same.

5.1 Sharded Case

In this subsection, we run multi-gets with 1, 3, 6 and 9 keys with sharding enabled. The setup is listed in the table below. For this set of experiments, we choose 64 as the number of middleware worker threads, because it always provides the highest throughput in the system, as shown in Sections 3 and 4. Although in the previous sections when there are only 2 clients in each memtier instance, the throughputs were the same among different numbers of worker threads, in this section the large `multi-get` size may possibly put higher pressure on the workers, so it is also a safe choice to have 64 worker threads.

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	2
Workload	ratio=1:<Multi-Get size>
Multi-Get behavior	Sharded
Multi-Get size	[1, 3, 6, 9]
Number of middlewares	2
Worker threads per middleware	64
Repetitions	3 × 80 seconds each

Figure 5.1 is the response time measured by memtier on its average, as well as 25^{th}, 50^{th}, 75^{th}, 90^{th} and 99^{th} percentile, with regard to multi-get size. The average response times fit the interactive law, both on the middleware side (with RTT added as think time) and on the memtier side.

5.1.1 Explanation

From the response time plots we can see that for every percentile (and the average), the response time always grows with multi-get size, which is reasonable because the more keys would require

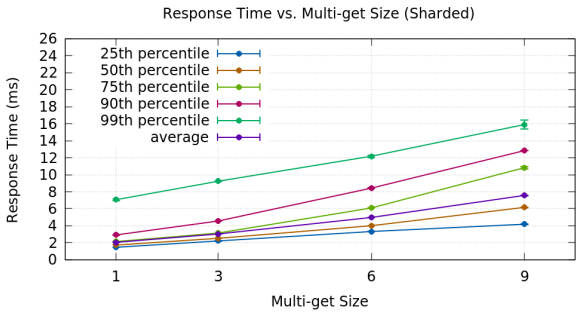

Figure 5.1: Memtier Response Time (Sharded)
(Error bars are too small to be distinguished)

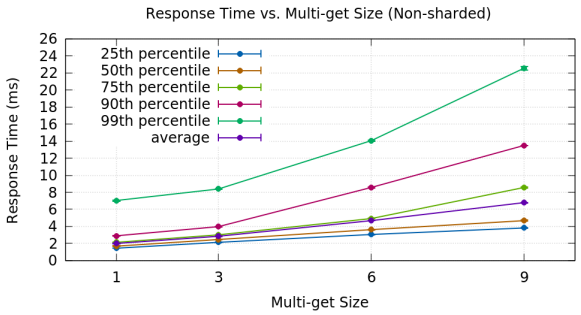

Figure 5.2: Memtier Response Time (Non-sharded)
(Error bars are too small to be distinguished)

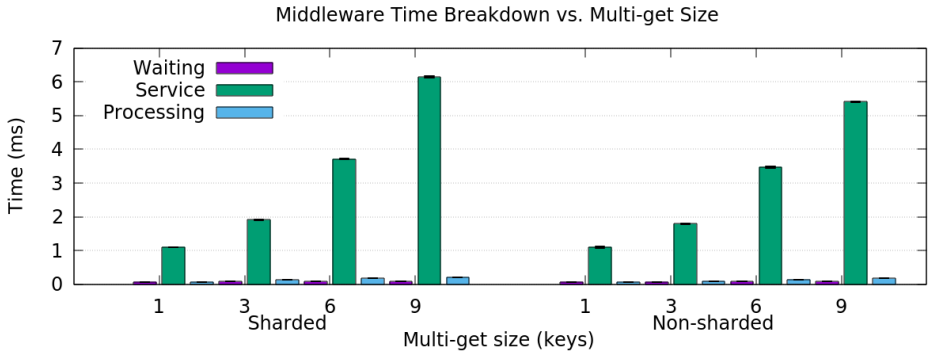

Figure 5.3: Middleware Time Breakdown (Sharded vs Non-sharded)
(Error bars are too small to be seen clearly)

more service time. This is indeed supported by Figure 5.3, left part: the memcached service time for 6-key and 9-key multi-gets is respectively about two and three times that of 3-key multi-gets. Note that although for 1-key and 3-key gets, the middleware always sends 1-key gets to servers, the average service time of 3-key is still larger than 1-key, because for 1-key gets, each request is sent to one server with uniform distribution, so the finally aggregated average service time is the average of three servers, while for 3-key gets, each request is split into three, one 1-key get is sent to each server and the middleware waits for all three responses, so the average service time is determined by the slowest server's service time, plus some parsing overhead (the request processor is implemented in such a way that it will parse the response in the mean time of receiving it).

The `get` (`multi-get`) throughputs in 1,3,6,9-key experiments are 2705.54 ops/sec, 2181.43 ops/sec, 1445.69 ops/sec and 986.36 ops/sec respectively, or 2705.54 keys/sec, 6544.29 keys/sec, 8674.14 keys/sec and 8877.24 keys/sec respectively, as measured by memtier. Middleware measurements are very close, within 1% error bound. For 1 and 3 keys, the bottleneck is the small number of clients, because no component is observed as saturated. For 6 and 9 keys, however, the bottleneck is again at the sending bandwidth of memcached servers, because this many keys per second (plus the additional `set` responses) have required the servers to send at a total bandwidth of 36 MiB/s. The servers' `dstat` logs indeed support this claim. 9-key experiments have resulted slightly more keys per second than 6-key, because the ratio of `set`s and `get`s is $1:1$, so 9-key experiments have less `set`s involved, leaving more traffic available for `get`s. The middleware is never a bottleneck, and consistently, our middleware logs show that the average queue length and the queue-waiting time has always been about zero.

5.2 Non-sharded Case

In this subsection, we run multi-gets with 1, 3, 6 and 9 keys with sharding disabled. As the maximum throughput is always reached by 64 worker threads, and also for the purpose of controlling variates, we still choose 64 as the number of worker threads.

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	2
Workload	ratio=1:<Multi-Get size>
Multi-Get behavior	Non-Sharded
Multi-Get size	[1, 3, 6, 9]
Number of middlewares	2
Worker threads per middleware	64
Repetitions	3 × 80 seconds each

Figure 5.2 is the response time measured by memtier on its average, as well as 25^{th}, 50^{th}, 75^{th}, 90^{th} and 99^{th} percentile, with regard to multi-get size. The average response times fit the interactive law, both on the middleware side (with RTT added as think time) and on the memtier side.

5.2.1 Explanation

Similar to the sharded case, the average response time, as well as the percentiles of different levels, grows with regard to the number of keys, as shown in Figure 5.2. Further, we can conclude from Figure 5.3 that the growth is mostly due to the increase in memcached service time, and the memcached service time for 6-key and 9-key multi-gets is respectively about two and three times that of 3-key multi-gets. However, for one-key get, the service time is 63.5% of the 3-key case (instead of only $\frac{1}{3}$), because it is dominated by the network latency.

The bottlenecks are also the same as in the sharded case. For 1-key and 3-key `gets`, the throughput (measured by memtier; middleware measurements are within 1% error bound, similarly hereinafter) is 2754.25 ops/sec and 2230.97 ops/sec respectively, which is only limited by the low number of clients. For 6-key and 9-key `gets`, the throughput is 1464.44 ops/sec and 993.21 ops/sec respectively, or 8786.64 keys/sec and 8938.89 keys/sec, with the overhead of `set` requests added, both has reached the sending bandwidth limit of memcached servers, which can also be proved from the `dstat` logs of three servers, so the bottleneck is again the server bandwidth. The middleware never becomes a bottleneck and gets saturated, as the queue waiting time and average queue length are always close to zero.

5.3 Histogram

Figures 5.4 $\sim$ 5.7 are four histograms for the case of 6-key-`multigets` representing the sharded and non-sharded response time distribution, both as measured on the client, and inside the middleware. Every histogram has a bucket size of one millisecond, and twenty buckets in total. Due to the long-tail effect because the server is sometimes a little bit unstable, there are rare instances of requests whose response time is higher than 20.0 ms, but such requests are fewer than 0.2% of the population, so they can be safely ignored.

The first observation is that the response time distribution has a long tail, with a majority hit in 2.0 $\sim$ 5.0 ms (memtier side) and a maximum hit at 14 ms for non-sharded 12 ms for sharded. The long tail effect is less significant in the sharded case than in non-sharded, because utilizing all three servers mitigates the randomness of the service time of a single server, making the overall response time more stable and the distribution more focused below 10 ms. This conclusion could also be drawn from the response time comparisons (Figures 5.1 and 5.2): the 99^{th} percentile response time of sharded case is significantly lower than non-sharded.

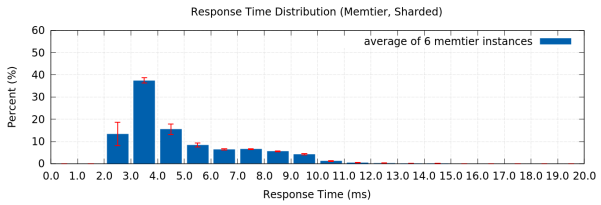

Figure 5.4: Memtier Histogram (Sharded)

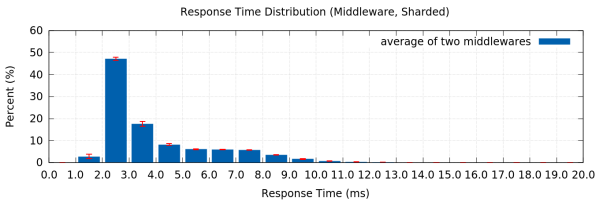

Figure 5.5: Middleware Histogram (Sharded)

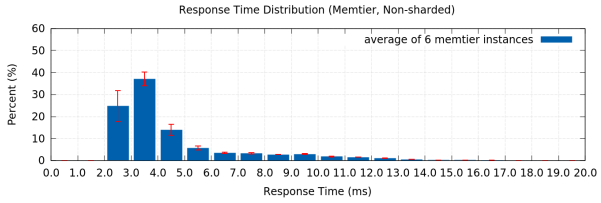

Figure 5.6: Memtier Histogram (Non-sharded)

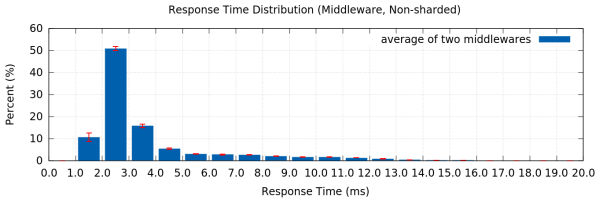

Figure 5.7: Middleware Histogram (Non-sharded)

Also worth noting is that the distribution measured by the middleware is entirely 1 ms lower than on memtier, and the $2.0 \sim 3.0$ bucket holds significantly more requests than the $1.0 \sim 2.0$ bucket. The main reason is that the network latency between clients and middlewares, which is on average 1 ms as well, is not included in the middleware measurements. Also, the network latency is also fairly unstable, with a mean standard deviation of 0.72 ms, so the response time difference is rather variable round 1 ms, making the middleware histogram not strictly 1-ms-shifted from the other side.

5.4 Summary

The table below lists our performance measurements with standard deviations in brackets.

Keys	Throughput (ops/sec)		Throughput (keys/sec)		Mean Response Time (ms)	
	Sharded	Non-sharded	Sharded	Non-sharded	Sharded	Non-sharded
1	2705.54 (35.76)	2754.25 (34.97)	2705.54 (35.76)	2754.25 (34.97)	2.05 (0.03)	2.01 (0.02)
3	2181.43 (8.97)	2230.97 (16.71)	6544.29 (26.91)	6692.91 (50.13)	3.03 (0.01)	2.85 (0.01)
6	1445.69 (0.18)	1464.44 (2.19)	8674.14 (1.08)	8786.64 (13.14)	4.97 (0.01)	4.68 (0.01)
9	986.36 (0.83)	993.21 (0.97)	8877.24 (7.47)	8938.89 (8.73)	7.56 (0.03)	6.80 (0.03)

From the table we can see clearly that sharding multi-key `gets` and distributing the shards to all servers always makes the response time slightly larger than simply relaying the original request to one server, despite that the request size has been reduced. For 1-key `gets`, whether sharding is enabled makes no difference because single-key `gets` are always processed by the same function in the middleware implementation, so the throughput and response time keeps the same (within reasonable experimental error bound). For 3-, 6- and 9-key `multi-gets`, Figure 5.3 shows that the memcached service time with sharding enabled is slightly higher than the non-sharded case. This is understandable because in the `ShardedRequestProcessor` of the middleware, receiving from all three servers is implemented in a synchronous way, i.e. there is a loop to receive from the servers one by one, so overhead of receiving from two more servers, as well as the response-assembling overhead, could override the the time saved by smaller number of keys, especially when there exists a server whose network RTT with the middleware is considerably higher than others, this server will be a drag in the whole process. The middleware processing time for sharding is also slightly larger, because new requests need to be assembled before being sent to servers, and this part is not counted into the memcached service time.

Comparing the response time means and percentiles, we find despite that the mean, 25^{th},

50^{th} and 75^{th} percentile response time of non-sharded is always lower than sharded, however, for the 99^{th} percentile, response time for sharded of any `multi-get` size is lower than non-sharded, because as explained already, performing sharding to utilize all three servers with less keys for each server could potentially reduce the randomness of multi-key service time of simply sending all keys to one server, thus mitigating the long-tail effect.

Finally, the response time histograms from memtier and middleware all feature a peak around $2.0 \sim 5.0$ ms, while middleware response time is entirely lower than memtier by about 1 ms, because it does not include the communication latency between memtier and middleware. For throughputs, for 1- and 3-key the maximum throughput is limited by the number of clients, as no component is found to be fully utilized; for 6-key and 9-key the system do hit the maximum throughput limited by the servers' sending bandwidth. This bottleneck analysis applies for both sharded and non-sharded. Although the throughput numbers for non-sharded experiments are slightly higher than sharded (taking standard deviations into account), the differences are actually very small, because the overhead introduced to the middlewares and the workload reduced for the servers are very close to each other. However, we do observe that as we increase the `multi-get` size, the sharded throughput becomes closer to non-sharded, so we do have a reason to believe that for even larger sizes of `multi-get`s, sharding may become a better option because the dominating part in the response time would be each server's service time, instead of the synchronised network communication overhead and request/response assembling overhead, thus the decrease on service time for larger sizes will make more sense.

In summary, in our system, for 1-,3-,6-,9-key `multi-get`s, sharding is never a better option because of the processing and network communication overhead introduced cannot be hidden by the reduced request size.

6 2K Analysis (90 pts)

In this section, for write-only and read-only workload respectively, we perform statistical analysis about the impact on throughput and response time by three factors with two levels each, namely Middlewares (M): 1 and 2, Memcached servers (S): 1 and 3, Worker threads per MW (W): 8 and 32. We use a $2^k r$ factorial experimental design, where $k = 3$ and $r = 3$, with the following setup:

Number of servers	1 and 3
Number of client machines	3
Instances of memtier per machine	1 (1 middleware) or 2 (2 middlewares)
Threads per memtier instance	2 (1 middleware) or 1 (2 middlewares)
Virtual clients per thread	32
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1 and 2
Worker threads per middleware	8 and 32
Repetitions	3×80 seconds each

The experiment results are gathered in the table below. It is verified that the interactive law holds both for the middleware (with the adaptions done as described in Section 3.1) and on the memtier side.

Next we will discuss the effects of the three factors for each workload, based on the 2k analysis results acquired with the following method:

1. For each factor M, S or W, the corresponding variable x_i is -1 for the lower level, 1 for the higher level.
2. The effects q_i are calculated with the sign table method for each response variable (*throughput* and *response time*) respectively.

Original Data			Readonly		Writeonly	
			8 workers	32 workers	8 workers	32 workers
TP	1 MW	1 S	(2930.50, 2900.02, 2916.42)	(2903.40, 2919.75, 2930.52)	(6602.47, 6957.25, 6985.67)	(10223.78, 10246.70, 10026.18)
		3 S	(5666.55, 5661.40, 5655.28)	(8471.73, 8549.28, 8541.87)	(4556.73, 4602.15, 4616.67)	(7920.83, 7875.78, 7847.98)
	2 MW	1 S	(2898.20, 2921.13, 2921.95)	(2901.75, 2929.00, 2895.27)	(8449.55, 8620.18, 8631.85)	(12301.38, 12406.87, 12439.07)
		3 S	(8677.92, 8736.78, 8500.63)	(8729.42, 8725.97, 8725.12)	(6905.23, 7444.00, 7234.07)	(10527.00, 10638.98, 10587.75)
RT	1 MW	1 S	(65.45, 65.40, 65.40)	(65.33, 65.32, 65.09)	(28.70, 27.28, 27.32)	(18.67, 18.63, 18.93)
		3 S	(33.86, 33.89, 33.93)	(22.65, 22.32, 22.34)	(41.88, 41.46, 41.33)	(24.09, 24.10, 24.32)
	2 MW	1 S	(65.19, 65.26, 65.22)	(65.29, 65.06, 65.26)	(22.40, 22.02, 21.99)	(15.47, 15.34, 15.43)
		3 S	(21.87, 21.85, 22.38)	(21.79, 21.80, 21.80)	(27.49, 25.51, 26.24)	(17.98, 17.94, 18.02)

TP: Throughput (ops/sec), RT: Response Time (ms), MW: Middleware, S: Server

3. Using 3 repetitions enables us to separate out the unexplained variation attributed to the experimental errors (factor e) and account for its effect. The sum of the squared errors (SSE) is computed by subtraction: $SSE = SSY - 2^3 \cdot 3(q_0^2 + q_M^2 + q_S^2 + q_W^2 + q_{MS}^2 + q_{MW}^2 + q_{SW}^2 + q_{MSW}^2)$, where $SSY = \sum_{i,j} y_{ij}^2$.
4. Then, the portions of variation are calculated with $SSi/SST = 2^3 \cdot 3q_i^2/(SSY - 2^3 \cdot 3q_0)$. 90% confidence intervals are calculated with $q_i \mp t_{[0.95;2^3\cdot(3-1)]} s_{q_i}$, where t is the t-distribution and $s_{q_i} = s_e/\sqrt{2^3 \cdot 3} = \sqrt{SSE/16}/\sqrt{24}$.
5. All calculations are omitted in the report for brevity[3].

6.1 Write-only Workload

For computation of effects, the additive model is chosen because 1) the range of values covered by the response variables is not large, as the maximum is no more than three times of the minimum, 2) although the effects of middlewares and workers seems to multiply, as their product is the total number of workers in the whole system, previous bottleneck analysis has shown that the throughput and response time do not always change with total number of workers, as the system is often bottlenecked by the server's bandwidth, CPU, or the middleware's network thread, 3) the effects of all other interactions are additive by intuition. So we can assume that different components affect the response variables additively and try to verify the assumption afterwards.

	Throughput			Response Time		
Factor	Effect	Variation	Confidence Interval	Effect	Variation	Confidence Interval
I	8527.005		(8477.38, 8576.64)	24.27125		(24.12, 24.43)
M	1155.155	**25.19%**	(1105.53, 1204.79)	-3.78875	**24.43%**	(-3.94, -3.63)
S	-963.9075	**17.54%**	(-1013.53, -914.27)	3.25875	**18.08%**	(3.11, 3.42)
W	1726.5225	**56.27%**	(1676.9, 1776.16)	-5.19625	**45.96%**	(-5.34, -5.04)
MS	171.2525	0.55%	(121.63, 220.89)	-1.54625	4.07%	(-1.69, -1.39)
MW	74.8275	0.11%	(25.2, 124.46)	1.40875	3.38%	(1.26, 1.57)
SW	-56.565	0.06%	(-106.19, -6.93)	-1.25875	2.70%	(-1.41, -1.1)
MSW	-49.71	0.05%	(-99.34, -0.07)	0.83125	1.18%	(0.68, 0.99)
e		0.24%			0.21%	

Table 6.1: Effects and Variation Explained in Write-only Workload

All effects are statistically significant with 90% confidence as no confidence intervals include 0. Table 6.1 shows clearly in boldface that the three largest portions of variation are explained by the number of worker threads per middleware (W), number of middlewares (M), number of servers (S) respectively. The same order applies for both throughput and response time, which is reasonable because with other parameters in the interactive law fixed, throughput and response time are just inversely proportional to each other. The contribution of interactions are less than 1% for throughput and 11.33% for response time, indicating that the interactions of factors do not affect much. The unexplained variation attributed to error is only 0.24% and

[3]The complete procedure can be re-run with Google Sheets from `logs/6/2k.xlsx` under the project repository.

0.21%. The overall result is reasonable and explainable, and the predictions using this regression model have been verified to match the responses with small residuals and no trend of spreading, plus that the quantile-quantile plot shows a satisfactorily normal distribution[4], it is safe to say the additive assumption holds.

For both throughput and response time, the number of worker threads per middleware explains the largest porition of variation (56.27% and 45.96%). This means increasing worker number from 8 to 32 gives the greatest performance boost (i.e. throughput increase and response time decrease), which is consistent with findings in Sections 3.1, 3.2 and 4.1, as the number of workers limits the number of requests to be processed in parallel, and the middleware is far from reaching the maximum throughput using only 8 workers, even with two middlewares.

The second largest portion is explained by the number of middlewares, with about half the portion of worker numbers: 25.19% for throughput, 24.43% for response time. Similarly, increasing from one middleware to two can boost the performance, by increasing the total number of worker threads in the system and by mitigating the burden on network queues with doubled network threads. However, the network thread is not the major bottleneck in this experiment, and the workers added are not as many as increasing W from 8 to 32, therefore it is less effective. The third largest portion is explained by servers, with 17.54% for throughput and 18.08% for response time. It has a negative effect on throughput and positive effect on response time, because adding more servers for write-only workload means that the middleware will have to replicate each request to three servers instead of only one, thus increasing the service time. This is also consistent with the differences of experiment results in Section 3.2 and 4.1.

In this set of experiments, the system reaches a maximum throughput of 12382.44 ops/sec with 2 middlewares, 1 server and 32 workers per middleware, which matches the prediction of this model.

6.2 Read-only Workload

Again in this subsection, we use additive model for regression, because the range of response values is not large, and a quick looking from the original data tells that the results are mainly determined by the number of servers itself. So we assume the effects of the three factors are additive and try to verify the assumption afterwards.

	Throughput			Response Time		
Factor	Effect	Variation	Confidence Interval	Effect	Variation	Confidence Interval
I	5400.40875		(5383.44, 5417.39)	45.15625		(45.06, 45.26)
M	396.51625	2.24%	(379.54, 413.5)	-1.59375	0.61%	(-1.69, -1.49)
S	2486.41875	**88.17**%	(2469.45, 2503.4)	-20.11625	**96.85**%	(-20.21, -20.01)
W	368.17875	1.93%	(351.21, 385.16)	-1.48375	0.53%	(-1.58, -1.38)
MS	399.29125	2.27%	(382.32, 416.27)	-1.53125	0.56%	(-1.63, -1.42)
MW	-347.35375	1.72%	(-364.33, -330.37)	1.42125	0.48%	(1.32, 1.53)
SW	368.88875	1.94%	(351.92, 385.87)	-1.43625	0.49%	(-1.53, -1.33)
MSW	-345.51875	1.70%	(-362.49, -328.54)	1.38375	0.46%	(1.29, 1.49)
e		0.02%			0.01%	

Table 6.2: Effects and Variation Explained in Read-only Workload

All effects are statistically significant with 90% confidence as no confidence intervals include zero. Table 6.2 shows that for both throughput and response time is mainly affected by the number of servers, with contributions of 88.17% and 96.85% respectively. The portions of variation explained by other factors and interactions are close to each other, and all significantly smaller

[4]The quantitle-quantitle plots and residual plots for 6.1 and 6.2 are available in `logs/6/process.ipynb`.

than the contribution by the number of servers. The residuals are orders of magnitude smaller than predictions with no trend of spreading, and the quantile-quantile plots show satisfactory normal distribution, so the choice of additive model is suitable.

The reason why the number of servers account for almost all variation is that the bottleneck in read-only workload is almost always the server, as we have found out in Sections 2 and 3. As we can see from the original throughput data, except the 1 middleware - 8 workers case, the system always reached the maximum throughput allowed by the corresponding servers' total sending bandwidth. In the 1 middleware - 8 workers case, the system is limited by the number of worker threads, and this bottleneck can be removed by increasing either the number of middlewares or workers, so there is a small portion of variation explained by M, W and the involved interactions.

7 Queuing Model (90 pts)

In this section, we model the system with different queuing models and analyze the similarities and differences between model predictions and actual measurements. Common notations that are used in all subsections include: traffic intensity (ρ: %) mean arrival rate (λ: ops/sec), service time per job (s: ms), mean service rate per server (μ: ops/sec), number of jobs in the system (n: ops), number of jobs waiting in the queue (n_q: ops), number of jobs receiving service (n_s: ops), response time or time in the system (r: ms) and waiting time between arrival and service start (w: ms).

7.1 M/M/1

For each worker-thread configuration in Section 4, we model the entire closed system with one M/M/1 queue, and compare the output parameters with actual measurements. As input, the arrival rate λ is set to the average throughput measured by memtier clients, instead of the average arrival rate measured by the network thread of the middleware, because when modelling the entire system as a whole, the best-effort estimation of arrival rate should have been done by the load generator. The service rate μ is set to the maximum memtier-measured throughput that has been observed under the same work-thread configuration, regardless of the number of clients, because this from the observations gives a best lower bound of the real service rate.

Because an M/M/1 queue is too much abstraction for the system, there is no "correct" way to map it to our system's actual components. Here we define the waiting/service separation point as the moment a request starts to be served by the memcached server, therefore, $E[s]$ is the middleware-measured memcached service time, $E[n_q]$ is the average length of the middleware request queue, $E[r]$ is the memtier-measured response time, and $E[w]$ is $E[r] - E[s]$.

	8 Workers		16 Workers		32 Workers		64 Workers	
	Predicted	Measured	Predicted	Measured	Predicted	Measured	Predicted	Measured
λ	-	7562.64	-	9214.13	-	10842.41	-	12932.08
μ	-	7694.54	-	9604.29	-	11240.95	-	13234.23
ρ	98.29%	-	95.94%	-	96.45%	-	97.72%	-
$E[s]$	0.13	1.97	0.10	3.08	0.09	5.46	0.08	9.32
$E[n_q]$	56.35	6.21	22.66	2.71	26.24	6.51	41.82	16.95
$E[w]$	7.45	2.73	2.46	2.08	2.42	3.30	3.23	5.09
$E[r]$	7.58	4.70	2.56	5.16	2.51	8.76	3.31	14.64

Table 7.1: Results of M/M/1 Model

For each worker-thread configuration, Table 7.1 is acquired based on the formulae in Box 31.1 from the textbook. Note that the experiments have been repeated for different number of

clients. For brevity, only results for the saturation points (as summarized in Section 4.1.1 and 4.2) are listed, namely 36, 48, 96, 192 clients for 8, 16, 32, 64 worker threads respectively. All ρ values are below 100%, so the system is stable.

Comparing between model predictions and experiment results, we find that a simple M/M/1 model is far from reporting the real life behaviour of the system. (1) The predicted mean service time $E[s]$ is much lower than reality, because M/M/1 cannot model the parallel request processing pattern of the system: the high arrival rate is actually not a result of low service time, but due to the fact that there are multiple workers processing multiple requests at the same time, but with M/M/1 the service time is just the inverse of arrival rate. (2) The predicted mean number of jobs $E[n_q]$ in the queue is much larger than reality, because there are more than one queues in the whole system (as we will show in Section 7.3), but M/M/1 has abstracted everything into one queue, and the length of middleware request queue is only a part of the queue length reported by this model. (3) The predicted mean response time $E[r]$ also has a large discrepancy from the reality, and does not follow the increasing trend of real response time with regard to the number of workers, because the model is not aware of the parallel processing of workers. (4) The mean waiting time in the queue $E[w]$ seems to be a little bit close to reality, but this is just a coincidence made by the choice of client numbers, for other numbers of clients with similar arrival rates, the predicted $E[w]$ will be similar but the measured waiting time could completely change, e.g. due to over-saturation.

Therefore, an M/M/1 model is over-simplified for our system, and indeed it cannot model the characteristics of the entire system satisfactorily.

7.2 M/M/m

In order to better model the multi-processing nature of the system, next we build an M/M/m model based on Section 4, where each middleware worker thread is represented as one service, therefore m = the total number of worker threads in two middlewares. The arrival rate λ is again set to the average throughput measured by memtier as it is the load generator and provides the best-effort estimation. The service rate μ of one service is set to the same μ in M/M/1 but divided by m, assuming that each worker has the same service rate. For the measured values of $E[n_q]$, $E[r]$ and $E[w]$, we use the same definitions as in Section 7.1; for $E[s]$, we set the measured value to the middleware-measured response time minus the middleware-measured queue waiting time, which is a best estimation of the mean time spent in worker threads.

	8 Workers (m=16)		16 Workers (m=32)		32 Workers (m=64)		64 Workers (m=128)	
	Predicted	Measured	Predicted	Measured	Predicted	Measured	Predicted	Measured
λ	-	7562.64	-	9214.13	-	10842.41	-	12932.08
μ	-	480.91	-	300.13	-	175.64	-	103.39
ρ	98.29%	-	95.94%	-	96.45%	-	97.72%	-
$E[s]$	2.08	2.12	3.33	3.25	5.69	5.65	9.67	9.55
$E[n_q]$	52.81	6.21	17.66	2.71	18.86	6.51	30.65	16.95
$E[w]$	6.98	2.58	1.92	1.91	1.74	3.11	2.37	5.09
$E[r]$	9.06	4.70	5.25	5.16	7.43	8.76	12.04	14.64

Table 7.2: Results of M/M/m Model

Based on the formulae from Box 31.2 in the textbook, Table 7.2 is computed with each worker-thread configuration, again with only results of saturation points as in 7.1. All ρ values are below 100%, so the system is stable. Comparing the predictions with realities, we do find similarity in $E[s]$ with errors $<= 0.12$: with the number of parallel workers considered, an M/M/m model is successful in predicting the mean service time of worker threads. However, for predicting the average queue length $E[n_q]$, the discrepancy is again large, because one M/M/m

model is still insufficient to describe the complex queuing structure of the system, for example it does not distinguish between the network queues and the middleware request queue. For the mean queue waiting time $E[w]$, it matches for 16 workers by coincidence, but in general it does not match at all due to the same reason as in $E[n_q]$. As a result, the predicted mean response time $E[r] = E[s] + E[w]$ does not match the reality either, because despite that the $E[s]$ part is correct, the $E[w]$ part is wrong.

In conclusion, an M/M/m model is better than M/M/1 for predicting the mean service time, because the parallelism of processing is captured; however, it is still insufficient to predict the queuing properties and system response time correctly, which calls for a queuing network specifically designed according to the system architecture.

7.3 Network of Queues

Now based on Section 3, we build two networks of queues to simulate the one middleware case and two middlewares case, respectively. Below is listed the commonalities of both networks. We use Java Modelling Tools[5]. for model design and simulation, because it provides handy graphical interface to organize queuing networks and run simulations and analysis algorithms including Mean Value Analysis (MVA).

The network between clients and middlewares is modeled as delay centers, which have infinite services with fixed delay (service time). The delay is set as half of the RTT measured between the two end, because it is a single journey. Note that in reality the number of services of our network may be limited by the bandwidth divided by request size, as we have seen in previous sections, but for simplicity we still model it as delay centers, and see if this can account for any mismatch in the end.

A middleware is modelled as an M/M/1 queue for the network thread, followed by an M/M/m queue for all worker threads, where m is the number of workers in the middleware. For two middlewares, we double this whole structure, so there will be two M/M/1 queues and two M/M/m queues, which is the only difference between the two queuing networks for one and two middlewares. The service time of the M/M/1 queue is the processing time of the network thread. The service time of the M/M/m queues is described in the next paragraph.

The memcached server, which may at first thought seem to be an M/M/1 queue, is not explicitly added as a component, but absorbed into the middleware M/M/m queues together with the network between middleware and server. This is because the service of memcached is actually a part of the service of a worker thread. By definition, if we model the memcached server with a separate M/M/1 queue, then when a request leaves a worker thread for the memcached server, the worker thread should immediately take a next request, which is not the case in our implementation: the worker thread needs to wait for the request (response) to come back, and send back the response to a client, before it becomes available for a next request. Therefore, the service time of a worker service is from "a request is dequeued" to "a response is sent to client", which consists of worker pre-processing time, memcached service time and worker post-processing time.

The measuring instruments of our middleware log down the mean middleware-processing time of all requests, which is the sum of network thread processing time, worker thread pre-processing time and worker thread post-processing time, but unfortunately the three parts are not logged separately. For best-effort estimation, we take one third of the mean middleware-processing time as the value for each part, because they essentially do similar things: network thread processing is mainly doing a network-receive of the request, worker pre-processing is mostly doing a network-send to the single server, and worker post-processing is mostly doing

[5]http://jmt.sourceforge.net. The corresponding model files can be found in `logs/7/` of the project repository.

another network-send to the corresponding client. We assume that network operations dominate the processing time and these operations take the same time on average.

As we have carefully split the system into different components, described almost all important characteristics of each component of the system in detail in the model, and employed best-effort estimation for every input parameter, the model is expected to match the measured values very well, and should be useful in predicting the bottleneck of the system through component utilization. For brevity, we only list the model prediction and reality comparison for maximum throughput configurations, which are described in Section 3.3, namely 64 threads/144 clients for write-only workload, 8 workers/12 clients for read-only workload, for both one middleware and two middlewares. In both parts, for device i, we list the number of services m_i, service time S_i (ms), visit ratio V_i, throughput X_i (ops/sec), response time R_i (ms), and utilization U_i (%); for M/M/m devices, the service rate with n jobs in the system is μ_i is $\min(n, m)/S_i$ (ops/sec). Based on this we identify the bottleneck component and compare the model with reality.

7.3.1 One Middleware

The network of queue for one middleware is depicted in Figure 7.1. The parameters for each component is listed in Table 7.3, where m_i, S_i, μ_i and V_i are inputs, $U_i(\%), X_i, R_i$ are outputs. The input parameters are chosen according to the aforementioned method.

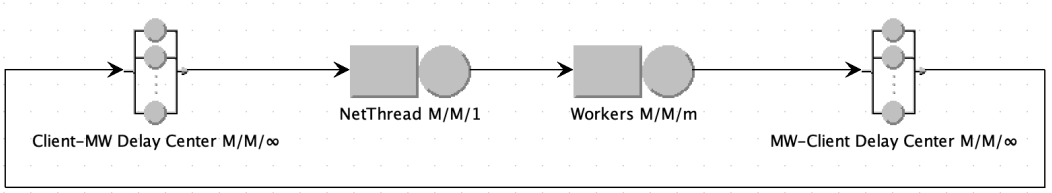

Figure 7.1: Queuing Network for One Middleware

Device	Readonly (8 workers/12 clients)							Writeonly (64 workers/144 clients)						
	m_i	S_i	μ_i	V_i	U_i	X_i	R_i	m_i	S_i	μ_i	V_i	U_i	X_i	R_i
Client-MW	∞	0.67	0.67	1	-	-	0.67	∞	0.57	0.57	1	-	-	0.57
NetThread	1	0.03	0.03	1	9.46	3160.43	0.03	1	0.05	0.05	1	78.66	15529.01	0.23
Workers	8	2.24	17.92	1	89.46	3155.11	2.41	64	4.08	260.91	1	100	15700.87	7.80
MW-Client	∞	0.67	0.67	1	-	-	0.66	∞	0.57	0.57	1	-	-	0.57

Table 7.3: Network of Queues for One Middleware

For readonly workload, the system throughput prediction is 3167.67 ops/sec, and response time prediction 3.79 ms, with the measured values 2903.11 ops/sec and 4.07 ms. The utilization of worker threads is as high as 89.46%, indicating that the workers are about saturated, which is exactly the case, as from Section 3.1 we know the bottleneck is on the bandwidth of the memcached server, and in this model the server has been absorbed into the workers M/M/m queue. The predictions match the realities closely, but the reported throughput is still a little higher than the measured values, because the model is not aware of the fact that the server's sending bandwidth has limited the maximum throughput.

For writeonly workload, the system throughput and response time is predicted as 15733.99 ops/sec and 9.22 ms, while the measured values are 12023.66 ops/sec and 11.84 ms. This does not match so well, and it is understandable: as we can see from the utilization, network threads are utilized by 78.83% and workers by 100%, indicating that the bottleneck is on the middleware worker threads; however, the resources of a middleware machine is limited, and not all worker

threads are active at the same time, as we have analysed in Section 3.1; therefore even the full utilization of all worker threads is not able to actually yield a theoretical maximum throughput of 64 workers. But the model is not aware of this, it simply assumes that the processing capacity is directly propotional to the number of workers, thus predicting a higher throughput and lower response time than reality.

Therefore, the model captures the system characteristic fairly well, as we have included most of the important system components in the model. It is also useful in identifying the bottleneck, as the utilization values align well with our findings in Section 3.1. However, it does not take the resource limitations into consideration, resulting in some considerable discrepancy between the prediction and reality.

7.3.2 Two Middlewares

Similarly, the network of queue for two middlewares is depicted in Figure 7.2. The values of input parameters are chosen as described in the beginning. The result for each component is listed in Table 7.4, where the last row represents for the entire system and therefore only cares about system throughput and response time.

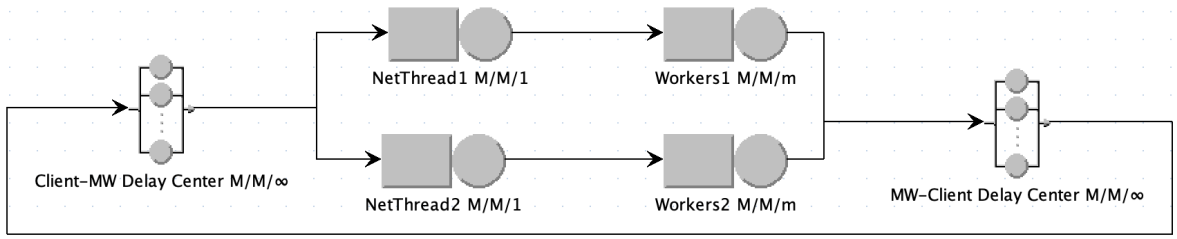

Figure 7.2: Network of Queues for Two Middlewares

Device	Readonly (8 workers/12 clients)							Writeonly (64 workers/144 clients)						
	m_i	S_i	μ_i	V_i	U_i	X_i	R_i	m_i	S_i	μ_i	V_i	U_i	X_i	R_i
Client-MW	∞	0.61	0.61	1	-	2910.86	0.62	∞	0.58	0.58	1	-	15503.87	0.57
NetThread1	1	0.03	0.03	0.5	4.44	1458.88	0.03	1	0.03	0.03	0.5	23.01	7664.37	0.39
NetThread2	1	0.03	0.03	0.5	4.35	1469.5	0.03	1	0.03	0.03	0.5	23.18	7654.36	0.39
Workers1	8	2.85	22.80	0.5	51.24	1453.43	2.83	64	7.85	520.40	0.5	95.16	7797.62	8.21
Workers2	8	2.85	22.80	0.5	52.24	1470.11	2.90	64	7.85	520.40	0.5	94.83	7665.26	8.19
MW-Client	∞	0.61	0.61	1	-	2907.74	0.61	∞	0.58	0.58	1	-	15499.25	0.58
System	-	-	-	-	-	2907.74	4.13	-	-	-	-	-	15499.25	9.35

Table 7.4: Results of Network of Queues for Two Middlewares

As comparison, the measured throughput and response time is 14524.03 ops/sec and 9.81 ms for write-only workload, or 2908.67 ops/sec and 4.07 ms for read-only workload. We can see that the predictions almost perfectly match the measurements, especially for read-only workload. The component utilization in readonly workload is about 4.4% for both M/M/1 network threads and about 51% for both M/M/m workers, showing that the load is equally distributed and the burden on workers from One Middleware has been relieved, and the middleware is still not the bottleneck. The utilization of both worker groups sums up to 100%, which matches the fact that the bottleneck is from the server bandwidth, which has been absorbed into workers in our model.

For write-only workload, the discrepancy between prediction and reality is also smaller than in the One Middleware model, because the resource scarcity is mitigated with doubled middleware, therefore the disadvantage of the model due to unawareness of the resource limitation is

also relieved. The utilization of network threads has decreased greatly to only 23% compared with One Middleware, but the utilization for workers is still as high as 95%, because the bottleneck is still the memcached server (as we have found in Section 3.2), and the server has been absorbed into the workers M/M/m queues.

To conclude, our network of queues is even better to model the two middlewares case than the one middleware case. It is successful to predict the system throughput and response time, and utilization values do align with the real bottlenecks we have found in Section 3.2. The discrepancy for write-only workload is also relieved, because the effect of resource limitation has been mitigated by doubled number of middlewares.