Advanced Systems Lab Report

Autumn Semester 2017

Name: Chalumattu Ribin
Legi: 12-930-939

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

Definitions

This section describes all the definitions and abbreviations used throughout the entirety of the report.

Number of servers	S
Number of client machines	CM
Instances of memtier per machine	$NumMemtier$
Threads per memtier instance	CT
Virtual clients per thread	VC
Multi-Get size	K_{mg}
Number of middlewares	$NumMW$
Worker threads per middleware	WT

Time request arrives at MW	T_{ReqIn}
Time response is sent from MW	$T_{RespOut}$
Time request enters queue	$T_{QueueIn}$
Time request leaves queue	$T_{QueueOut}$
Total queue time: $T_{QueueTot} = T_{QueueOut} - T_{QueueIn}$	$T_{QueueTot}$
Time request is dispatched to server	$T_{ServerOut}$
Time response received from server	$T_{ServerIn}$
Total server processing time: $T_{ServerTot} = T_{ServerIn} - T_{ServerOut}$	$T_{ServerTot}$
Total parsing time inside worker-thread = $T_{RespTot} - T_{QueueTot} - T_{ServerTot}$	$T_{WorkerParse}$
Average queue length	$QueueLength$

$$Cache\ miss\ ratio = \frac{\#Empty\ Responses}{Total\ Responses}$$

For a single get or multi-get request an empty response is defined as: n - m, where n = # keys contained in the request and m = # responses returned by the server.

1 System Overview (75 pts)

In this section we first outline the overall architecture of the middleware, followed by subsections specifying the individual components of system in further detail. The most important classes mentioned in this section have a corresponding link given in section 8 of the report.

1.1 Overall architecture

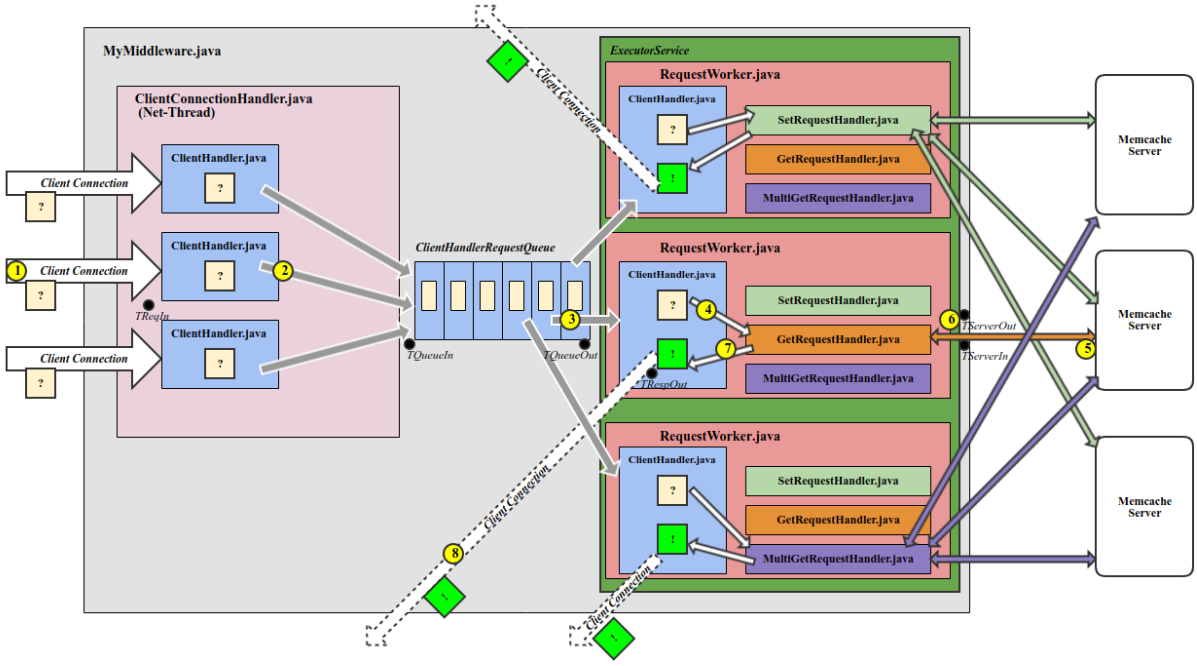


Figure 1: A high-level overview of the load-balancing system. The locations of the measured time-stamps are indicated with black dots with their corresponding names. A sample execution path within the middleware is indicated by the numbered yellow dots.

1.2 MyMiddleware[1]

The *MyMiddleware* is responsible for the following tasks:

- Initialize a request queue to which the net-thread can enqueue pending requests and worker-threads can dequeue from. Since this queue is accessed by multiple threads at the same time, some sort of synchronization has to be performed in order to avoid race conditions. Therefore the queue is implemented using Java's *LinkedBlockingQueue*, due to its thread safety.
- Initialize the thread pool containing the worker-threads. The thread pool is implemented using Java's *ExecutorService*, which provides mechanisms to safely manage multiple threads in the background. We first initialize the *ExecutorService* with a fixed size of T threads, where T represents the number of worker-threads given to the middleware at run-time. Then we initialize each of the workers and finally submit them to the *ExecutorService*.
- Initialize and run the net-thread (call the *run* method of the *ClientConnectionHandler.class*.

- When the middleware is killed, a shut down hook is called from within the MyMiddleware class in a method called *stop*. The shut down hook closes all open connections and logs all the data gathered by the workers into a log file.

1.3 ClientConnectionHandler[2]

This class implements the logic of the net-thread. Since the net-thread would have to handle multiple clients simultaneously, we used Java's *Selector* object from the Java.NIO package. The main reasons for utilizing the *Selector* were the following:

- One to rule them all. We can use a single thread(the net-thread) to manage multiple Java.NIO channels and thus multiple network connections. We avoid therefore the cost of creating and managing separate threads for each new connection.
- The *Selector* automatically recognizes if data becomes available on one or more of its managed channels. Hence we avoid useless polling of the individual connections for incoming data and use the time more efficiently.

The *ClientConnectionHandler* class is mainly responsible for accepting incoming connections, reading incoming data from those connections and enqueueing complete requests into the request queue. The code listing below illustrates a simplified version of the main methods inside the *ClientConnectionHandler* class. Additional tasks performed in this class are described within the comments starting with *Pseudo*.

Listing 1: Main methods of the *ClientConnectionHandler* class

```
//Main run method of the Net-Thread
public void run{
    //---Pseudo:Open ServerSocketChannel on the given Ip and port
    //---Pseudo:Open Selector object
    while(isRunning){
        acceptConnections();
        readFromConnections();
    }
}

private void acceptConnections(){
    //The accept method returns immediately
    SocketChannel socketChannel = serverSocketChannel.accept();
    if(socketChannel != null){
        ClientHandler clientHandler = new ClientHandler(socketChannel);
        //---Pseudo:Register the SocketChannel with the corresponding ClientHandler
    }
}

private void readFromConnections(){
    //The selectNow method returns immediately
    int numOfReadyChannels = selector.selectNow();
    if(numOfReadyChannels > 0){
        //---Pseudo:For each ready channel, get the corresponding ClientHandler class
        //Read the request asynchronously using the clientHandler
        boolean requestComplete = clientHandler.readRequest();
        //---Pseudo:If the complete request has been read, enqueue the clientHandler
            into the ClientHandler request queue
    }
```

}

1.4 ClientHandler[3]

This class is the main interface for communicating with clients. For each new incoming connection, the net-thread instantiates a corresponding *ClientHandler* class. The main two methods of this class are *readRequest* and *sendResponse*. The former is called by the net-thread in the *ClientConnectionHandler.class* in order to read a complete request from the client. The latter method is called by the worker-thread to write the complete response back to the client. The communication between *ClientHandler* and memtier client occurs in an asynchronoues fashion by reading from and writing to a *SocketChannel* object from the Java.NIO package. Instead of directly putting the request onto the queue, the net-thread puts a ClientHandler object on the queue. This allows the worker-thread to send back the response through the corresponding ClientHandler object.

1.5 RequestWorker[4]

This class implements the main logic of the worker-thread. The *RequestWorker* executes the following tasks:

- Create a synchronous network connection to each of the memcache server instances. This connection is implemented by the *SynConnection* class, which holds methods to read from and write to the servers in a synchronous fashion.
- Dequeue a *ClientHandler* from the *ClientHandlerRequestQueue*. Depending on the request type, the worker-thread delegates the request to the corresponding request handler class. The handler class then sends the request to the memcache server, reads the corresponding response and returns it to the worker-thread.
- Call the *sendResponse* method of the *ClientHandler* to respond to the client, by providing the newly received response as a parameter.

1.6 RequestHandlers

All request handler classes implement the method *handleRequest*, which takes a *MemcachedRequest*[5] as a parameter and returns the corresponding *MemcachedResponse*[6] from the server. *MemcachedRequest* and *MemcachedResponse* are wrapper classes containing the data of the request or response. Communication between request handler and memcache server is always done with synchronous networking using the aforementioned *SynConnection* class. *SynConnection* contains a *MemcachedMessageParser*(see section 1.7), which reads incoming data from the server and determines when a complete response has been read. Each request handler first writes the request to all necessary servers, before attempting to read a response. Below we explain the *handleRequest* method for each of the request handlers in more detail.

1.6.1 SetRequestHandler[7]

The *SetRequestHandler* first loops through all servers and sends the current request to each of them in a synchronous fashion. After writing the request to the servers, the *SetRequestHandler* loops again through each of the servers and reads the corresponding responses from them. If one of those responses contains an error message, that response is returned immediately.

Otherwise we return the first response received. In listing 2 we show the implementation of the *handleRequest* in more detail. Additionally we indicated when the time stamps are made (in magenta), in order to measure the waiting time for the server ($T_{ServerTot}$).

Listing 2: Main method of the *SetRequestHandler* class

```
public MemcachedResponse handleRequest(MemcachedRquest clientRequest){
    MemcachedResponse[] serverResponses = new MemcachedResponse[numOfServers];
    //Write the request to all servers
    for(int i = 0; i < numOfServers; i++)
        synConnections[i].write(clientRequest.getMessageInBytes());

     //Measure T_ServerOut
    //Read the request from all servers and send the worst response received, back to
    //the client
    for(int i = 0; i < numOfServers; i++){
        serverResponses[i] = synConnections[i].readResponse();
        if(serverResponses[i].getResponseFlag() == ResponseFlag.ERROR){
             //Measure T_ServerIn
            return serverResponses[i];
        }
    }
    //Measure T_ServerIn
    return serverResponse[0];
}
```

1.6.2 GetRequestHandler[8]

The *GetRequestHandler.class* hashes the key of the current request using the *Hasher.class*(see section 1.8). The hasher then returns the id of the server, to which the *GetRequestHandler* writes the request to and reads the response from. Below we indicate again the implementation of the *handleRequest* method.

Listing 3: Main method of the *GetRequestHandler* class

```
public MemcachedResponse handleRequest(MemcachedRquest clientRequest){
    //Hash the request key to determine which server receives the request
    int serverId = hasher.getServerId(clientRequest.getKey());
    //Send request to the server given by the hash
    synConnections[serverId].write(clientRequest.getMessageInBytes());
     //Measure T_ServerOut
    //Read response from the corresponding server
    MemcachedResponse serverResponse = synConnections[serverId].readResponse();
     //Measure T_ServerIn
    return serverResponse;
}
```

1.6.3 MultiGetRequestHandler[9]

For the nonsharded case this class executes the same code as the *GetRequestHandler*. If the request contains more than a single key, we randomly choose one of the keys and hash it to get the corresponding server. For the sharded setting, the handler determines how to split up the

multi-get request equally among the servers. Then it shards the multi-get request into smaller multi-get requests. The handler loops through all the sharded messages and writes each of them to the corresponding server. The first message is send to the first server, the second message to the second server, etc. After writing, it loops again through all the servers to which it has send a request to and attempts to read a response in a synchronous fashion. Finally it joins all the responses and sends the complete response back to the client. The listing below shows the implementation for the sharded case. Important methods are shown in red and explained in detail further below.

Listing 4: Main method of the *MultiGetRequestHandler* class.

```
public MemcachedResponse handleRequest(MemcachedRquest clientRequest){

    int numOfKeys = clientRequest.getNumOfKeys();

    if(readSharded && numOfServers > 1){
        //determine how many keys each server gets,i.e. server i gets part[i] keys
        int[] parts = getParts(numOfKeys, numOfServers);
        //shard request according to parts
        MemcachedRequest[] shardedRequests = shardRequest(clientRequest.getAllKeys(),
            parts);
        MemcachedResponse[] serverResponses = new
            MemcachedResponse[shardedRequests.length];

        //Send sharded requests to the corresponding servers
        for(int i = 0; i < shardedRequests.length; i++){
            synConnections[i].write(shardedRequests[i].getMessageInBytes());
        }
        //Receive responses
           //Measure T_ServerOut
        int totalMessageLen = 0;
        for(int i = 0; i < shardedRequests.length; i++){
            serverResponses[i] = synConnections[i].readResponse();
            totalMessageLen += serverResponses[i].getMessageContentLength();
        }
         //Measure T_ServerIn
        //assemble responses together
        return joinResponses(serverResponses, totalMessageLen);
    }else{
        //Same as in GetRequestHandler
    }

}
```

In listing 4, 3 methods are of particular interest(indicated in red), namely *getParts, shardRequest, joinResponses*.

- *getParts* takes as input the number of keys contained in the current multi-get request and the number of servers. The method outputs an integer array, indicating how many keys each server is assigned to. This assignment is done by uniformly distributing the number of keys among the servers.
- The *shardRequest* method utilizes the *parts* array to shard the request into multiple smaller multi-get requests. We demonstrate the sharding process using the loop below: We are given an array *parts* of size n and a string array *keys* of size m, containing all the keys of

the multi-get request. Let m = *parts[0] + ... + parts[n-1]*. The output will be a string array *sharded* of size n.

Listing 5: Example of the sharding process.

```
startIndex = 0;
for(int i = 0; i < n; i++){
    endIndex = startIndex + parts[i];
    sharded[i] = "get ";
    for(int j = startIndex; j < endIndex; j++){
        sharded[i] += keys[j]
    }
    startIndex += endIndex;
}
```

An example: We are given *parts=[2,2,1]* and *keys=["one","to","rule","them","all"]*, which would result in *sharded=["get one to", "get rule them", "get all"]*.

- Finally *joinResponses* takes all the responses from the servers as input and returns a joined response. For the merging process, some padding has to be done to each of the responses, such that the merged response conforms with the memcached protocol.

1.7 MemcachedMessageParser[10]

The *MemcachedMessageParser* has 2 main functionalities:

- Parse incoming request messages in the method *parseRequestMessage*. For each type of request, the parser first tries to read the header of the request. This is achieved by reading bytes from the client network connection, until the byte containing the delimiting character $\backslash n$ has been read. In case of a get or multi-get request, the parser has finished reading the request and returns. In case of a set request, the parser reads until $b+2$ bytes have been read, where b is the size of the request body given by the header and an additional 2 bytes for the final delimiting characters $\backslash r$ and $\backslash n$. The *parseRequestMessage* method is called, when the *ClientHandler* calls the *readRequest* method.
- Parse incoming response messages in the method *parseResponseMessage*. In case the corresponding request message was of type get or multi-get, the parser simply reads bytes from the server connection, until the delimiting characters *END* have been read and return the response. In case the request was of type set, the parser reads until the delimiting characters $\backslash r\backslash n$ was read and return the response. The *parseResponseMessage* method is called by the *SynConnection* class, when it tries to read the response from the server in a synchronous fashion.

1.8 Hasher[11]

In order to distribute the incoming requests equally among the servers, each request key is mapped to one server using a load balancing hash function. In order to hash a given key, an MD5 hashing algorithm was used to create the corresponding hash value. This hash value is then converted to a decimal value, which is then divided by the number of servers. By using the remainder of the division, each key is assigned to one of the servers. This approach is simple, fast and works well since we assume no failures from the server side. The load balancing is uniformly distributed, since the MD5 algorithm creates a pseudo random hash, hence each key

has uniform probability of being assigned to a particular server. This ensures us that the load is distributed evenly among the servers. During the multi-get experiments in section 5, we counted the hits on each of the 3 servers for the nonsharded setting with 32 VC per client-machine, 3 repetitions(60s each)and got the following results for a maximum key-size of 1 and 9:

Key-Size	Server 1	Server 2	Server 3
1	548023	506782	510123
9	238078	233246	237013

Table 1: Hits on each of the 3 servers for different maximum key sizes.

From the results above we conclude that each of the servers handle roughly the same number of requests for get and multi-get messages. Therefore the *Hasher* balances the load equally among all the memcached servers.

1.9 Statistics Gathering in Multithreaded Setting

The following aggregated statistics are gathered per worker-thread, over a 2 second window:

- Average Throughput.
- Average Queue-Length.
- Average Queue-Waiting Time.
- Average Server-Processing Time.
- Number of SET, GET and MULTI-GETS.
- Average number of keys in MULTI-GET requests.
- Average number of keys in MULTI-GET requests.
- Number of requests with unknown type(neither set, get or multi-get).

$T_{QueueIn}$ and T_{ReqIn} are measured in the net-thread, whereas $T_{QueueOut}$ and T_{ReqOut} are measured in the worker-thread. Consequently we store these time stamp values with their corresponding request. Each *MemcachedRequest* object has a *LogDataTracker* instance, that stores all the necessary time stamps. This design decision allows the middleware to compute all the statistics mentioned above in the worker-thread.

In order to avoid performance loss due to excessive logging, we cache the statistics to be logged inside the worker-thread. When the middleware is killed, a shut-down hook will be called, which logs the data of all workers to one log file. Additionally we also log the following statistics, when the middleware is stopped:

- Histogram of the response times in tenth of a millisecond.
- Cache miss ratio.

For aggregating, analyzing and plotting the data gathered by the memtier clients and the middleware, we use an external tool(matlab). We aggregate the data measured by the memtier clients by adding the measured throughput and taking the average of the response time over all client machines. For the middleware measurements we add again the throughput and average the response time over all middleware machines.

1.10 Summary of Design Decisions

In summary the main design decisions affecting the performance of the middleware are the following:

- Asynchronous networking between the clients and the middleware using the JAVA.NIO package. This allowed us to implement one net-thread, which can communicate with multiple client connections simultaneously.
- Synchronous networking between the memcache servers and the middleware. Each worker-thread always writes requests first to all necessary servers, before attempting to read the corresponding responses.
- For each incoming connection a *ClientHandler* class is created. If there is a request to be read from the client connection, the net-thread calls the read method inside the *ClientHandler*. If there is a response to be written to the client connection, the request worker thread calls the write method inside the *ClientHandler*. The request queue inside the middleware consists of *ClientHandler* objects that contain the current request of the client.
- In total there should be k worker threads and one net-thread present in the middleware.

1.11 Structure of the sections

The majority of the sections in this report are structured as follows: First we will pose the key questions to be answered in the section. Following that will be the hypothesis section, where we try to predict the answers to those key questions. Then we show the experimental results and analyze them by comparing the key findings with our predictions. Finally we verify the interactive law by predicting the response time using the throughput data measured by the memtier clients. We chose to check the interactive law using the data measured at the clients instead of the MW. The reason was that the predicted response time was higher than the measured response time at the middleware. This difference stems from the fact that the response time at the MW doesn't take into account the network latency needed to receive requests from the clients and send the responses back to them.

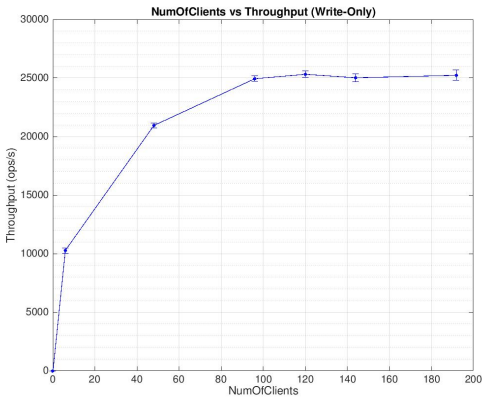
2 Baseline without Middleware (75 pts)

2.1 One Server

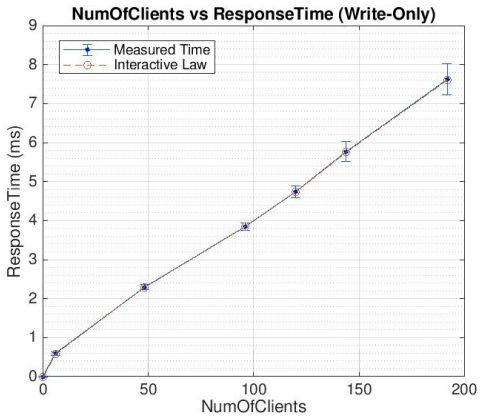
In the following table we summarized all the settings used throughout this experiment. When we aggregate the results, we omit the first and last 5 seconds of data from each repetition, in order to account for the warm-up and cool-down phases.

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[1,8,16,20,24,32]
Workload	Write-only and Read-only
Repetitions	3 (70 seconds each)
Warm-Up/Cool-Down phase	5 seconds

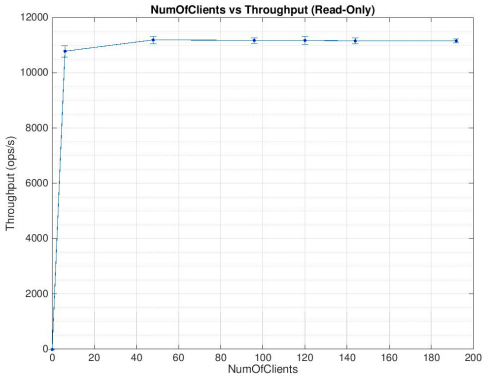
2.1.1 Experimental results



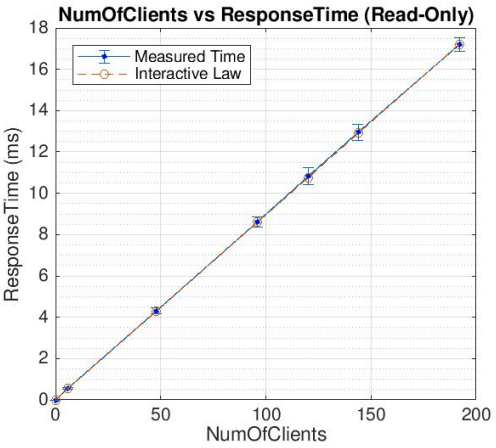
(a) Throughput-write.



(b) Response Time-write.



(c) Throughput-read.



(d) Response Time-read.

Figure 2: Baseline One Server: Experimental Results

2.1.2 Explanation

2.1.2.1 Write-Only Workload

Until 96 clients we can observe from figure 2a that the throughput is rapidly increasing. After 96 clients, the throughput starts to flatten and remains more or less stable up to 192 clients. From these observations we can infer that the memcached server is under-saturated until 96 clients as the server thread still has free resources and can process requests without delay. After 96 clients the server starts to become saturated, since the memcached thread has reached the maximum number of requests it can process in a given amount of time. Therefore the server has to queue some requests and the overall response time increases. This increase is reflected in figure 2b. Initially the slope of the response time curve is small, but after 96 clients the slope becomes larger.

2.1.2.2 Read-Only Workload

From figure 2c we observe that the throughput increases until 48 clients. After that the throughput starts to flatten and remains constant until 192 clients. From figure 2d we see the response time increasing rapidly with increasing number of clients. Furthermore the slope of the response time curve is steeper than for the write-only case. From these findings we conclude that the server is under-saturated until 48 clients. After this point the server thread start to become saturated and therefore the throughput remains stable at 11100 ops/s until 192 clients.

2.1.2.3 Correlation between Throughput and Response Time

In order find the relationship between the throughput and response time, we compute the response time according to the interactive law. We then compare the measured (R_M) and predicted values (R_P). Assuming that the clients need no time to think and send requests almost instantly, we can compute the response time as follows:

$$ResponseTime = \frac{NumClients}{Throughput}$$

The results are shown in the figures 2b and 2d. We see an almost perfect fit between the predicted and measured response time values. From this finding we can conclude that the relation between throughput and response time can be explained using the interactive law: if there is a large increase in response time with each step size, we can expect a decrease in throughput. On the other hand when the throughput is increasing rapidly, we expect a small increase in response time with each step-size.

2.1.2.4 Comparison of Write and Read-Only Workloads

From the experimental results above we can conclude that the server saturates faster for W_{Read} than W_{Write}. One reason for this behaviour could be, that for read requests the responses from the server is larger (1KB) than for a write response(< 10 Bytes). Consequently more time is needed on the network to send the responses back to the client, especially when the number of clients is large. Another reason could be that it takes the server longer to process a single read request than a write request. These hypotheses are supported by the response time figures, where the response time for the same number of clients is higher for W_{Read} than W_{Write}. As a result the server thread processes less read- than write requests in a given amount of time, which leads to faster saturation at the server.

2.2 Two Servers

In the following table we summarized all settings used throughout this experiment.

Number of servers	2
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1,8,16,20,24,32]
Workload	Write-only and Read-only
Repetitions	3 (70 seconds each)
Warm-Up/Cool-Down phase	5 seconds

2.2.1 Key Questions

a. When are the servers under-saturated, saturated or over-saturated?

b. How do response time and throughput correlate?

c. What are the differences between a read-only and write-only workload?

d. How do the results change compared to previous section?

e. What remains the same compared to the previous section?

2.2.2 Hypothesis

a. For a write-only workload we needed 96 clients to saturate one server in the previous experiment. During this experiment each server only has to handle up to 32 clients. Therefore we predict, that neither server will be saturated for a write-only workload and the throughput will be strictly increasing. For a read-only workload, we needed around 48 clients to saturate one server, hence we again predict that neither server will saturate, as the load is balanced among the servers.

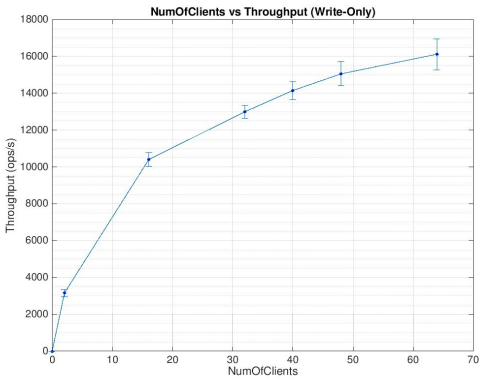
b. As seen previously we can describe the relationship between throughput and response time through the interactive law. We assume that the predicted response time and the measured response time lie close together similar to the previous section.

c. We predict that the throughput values for both workloads will be similar, since no server is saturated. Furthermore from the previous section we assume that the maximum throughput for a read-only workload will be slightly lower than for the write-only case.

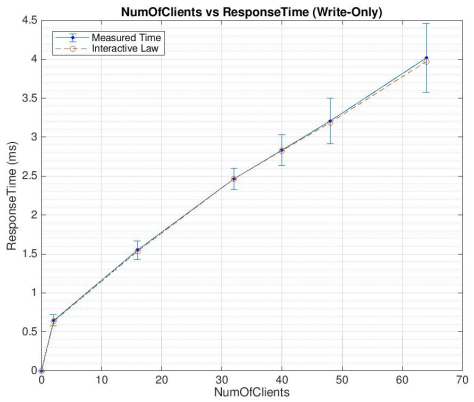
d. **Read-only case**: Since the load is distributed evenly between the 2 servers, we expect that the average response time of both servers will be lower than in the previous experiment. Consequently we estimate that the added throughput of both servers will be higher than with a single server, since neither memcached instance should be saturated.

e. **Write-only case**: We expect the response time and throughput for the same number of clients to be similar to the previous section, since neither server is saturated up to 64 clients(32 per server).

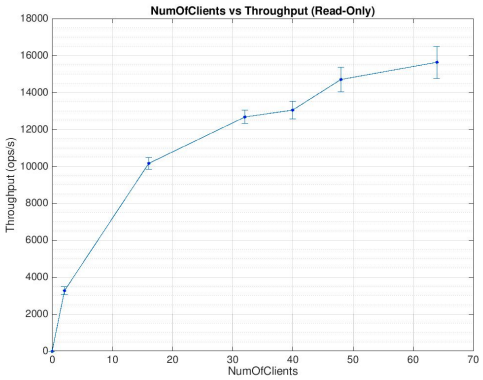
2.2.3 Experimental Results



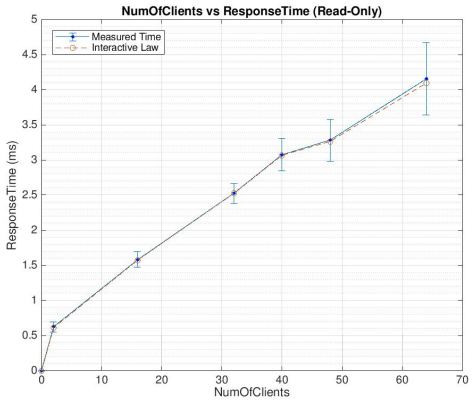
(a) Throughput-write.



(b) Response Time-write.



(c) Throughput-read.



(d) Response Time-read.

Figure 3: Baseline Two Servers: Experimental Results

2.2.4 Analyis of Results

a. **Write-only case**: From figure 3a we observe that the throughput is strictly rising up to 64 total clients. Figure 3b shows that the response time is increasing slowly with each step size and the maximum response time is at around 4 ms. In the previous test, the server started to saturate after 4 ms. Therefore we conclude that with 64 clients the servers are still at the point, where they can process requests without delay but will soon start to saturate after adding more clients.
Read-only case: In figure 3c we observe an increase in throughput with higher number of clients. However this increase slows down toward the end of the experiment. The response time values depicted in figure 3d are also increasing with each step size and the maximum value is at around 4 ms. For the read-only case in the one server setting we observed that saturation occurred after 4 ms. We therefore conclude that the servers are are at the point of saturation and adding more clients will not significantly increase the throughput.

b. We computed the response time values using the interactive law. The results are shown in figures 3b and 3d. We observe that R_M and R_P are almost identical and therefore the interactive law holds..

c. As predicted, we notice that the throughput and response time values lie close together for

both workloads. We can also conclude that the maximum throughput for the read-only case is slightly lower than for the write-only case, as more time is needed to send and process read-requests at the server thread.

d. **Read-only case**: The maximum throughput of both servers combined was at 15600 ops/s for 64 clients. As stated in the hypothesis, the throughput for the 2 server case is significantly higher than for the one server setting, since each server has to handle less load and therefore doesn't get saturated as fast as in the one server setting.

2.2.5 Differences to Hypothesis

e. Write-only case: Contrary to the hypothesis, the throughput for the same number of clients with two servers is lower than with one server. For example the throughput with 2 servers at 48 clients is 15000 ops/s, whereas in the previous case it is at 21000 ops/s. The difference stems from the fact that in one case the server handles 6 client threads simultaneously, whereas in the other a server only has to handle a single client thread. Therefore for write requests the capabilities of the single server are utilized more efficiently with 6 threads rather than 1. As a result the single server can process more requests in a second.

2.3 Summary

Maximum throughput of different VMs.

	Read-only workload	Write-only workload	Configuration gives max. throughput
One memcached server	11200 ops/s	25300 ops/s	VC = 20, Write-Only
One load generating VM	15600 ops/s	16100 ops/s	VC = 64, Write-Only

When comparing the results of both experiments we get the following key take-ways:
Read-only case: The maximum throughput with one server is lower than with 2. For the one server setting, the server-thread saturates after a certain number of clients, which increases the delay for processing requests and flattening the throughput after the saturation point. By adding a second server to the system, the client load could be distributed evenly among the server machines. This lead to an increase in throughput and decrease in response time.
The bottleneck for the read-only case is how fast the server thread can process read requests. The second bottleneck is the network time needed to send the large read-responses back to the client.
Write-only case: The max. throughput is reduced when decreasing the number of client machines from 3 to 1. Since the server only saturates after 96 clients for write-requests, the capabilities of the server thread are much better utilized with 3 client machines rather than a single one. Additionally in the 2 server setting, each server only handles up to 32 total clients. As a result each server-thread is still under-saturated and the aggregated throughput of both servers is lower than with 3 client machines.
The server thread becomes saturated if it can't process all incoming requests without delay and therefore starts to queue them up. As a result, the bottleneck for the write-only case mainly depends on how fast the server can process write-requests.
Key Memtier Take-Away: We sometimes noticed that the performance of the system could suddenly drop due to an anomaly in the network or cloud environment. These anomalies only occurred during specific repetitions and only for a couple of seconds. Therefore the performance drop has nothing to do with the behaviour of our system and the anomalies are simply outliers in data, which needs to handled when analyzing the results.

3 Baseline with Middleware (90 pts)

3.1 One Middleware

The following table summarizes the settings used for this experiment.

Number of servers	1
Number of client machines	1
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[4,8,16,20,24,32]
Workload	Write-only and Read-only
Number of middlewares	1
Worker threads per middleware	[8,16,32,64]
Repetitions	3 (1 minute each)
Warm-Up/Cool-Down phase	4 seconds

3.1.1 Key Questions

a. What is the bottleneck of the system?

b. When does the system saturate?

c. How does the average queue-length change by increasing the clients/worker-threads?

d. What are the differences between a read-only and write-only workload?

3.1.2 Hypothesis

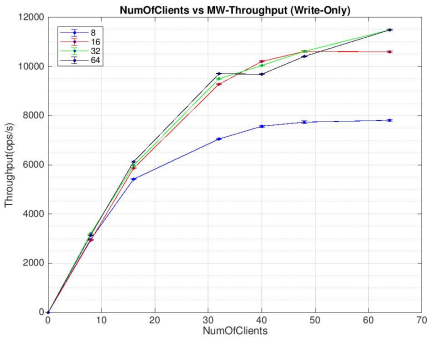
a. When the number of worker-threads(WT) is low, the workers can't keep up with increasing number of requests. This leads to an increase of the average queue length and therefore an increase in the overall response time. We conclude that when not enough worker-threads are used, the MW will be the bottleneck of the system.
When WT is high enough, we expect the server to be the bottleneck as the workers can handle requests without delay and therefore the maximum throughput is limited by the how fast the server thread can process requests and the network roundtrip time needed to communicate with the server.

b. **Server is the bottleneck**: In this case, we expect that the server saturates at similar number of clients as seen in the previous baseline experiment (48 for read-only and 96 for write-only requests).
MW is the bottleneck: Here we assume that saturation occurs before the server saturates. Consequently the maximum throughput measured at the MW will be lower than the throughput values measured during the baseline experiment.

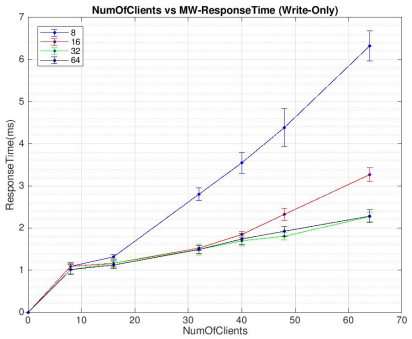
c. As the number of clients is increased, we expect the avg. queue length to increase. This increase will be high if WT is low and vice versa, as depending on the number of worker threads the MW can process more or less requests in certain amount of time.

d. We assume that the differences between both workloads are similar to the baseline experiment. Hence we get a higher throughput for writes than reads, as the server saturates only with 96 clients for writes and 48 clients for reads.

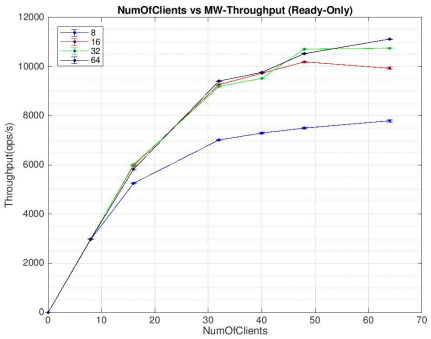
3.1.3 Experimental Results



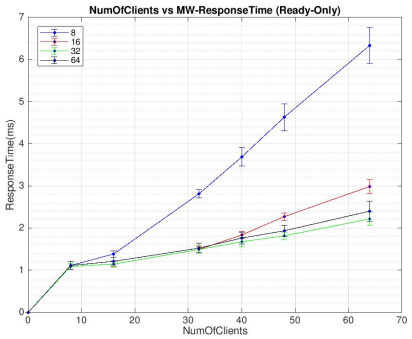
(a) Throughput-write.



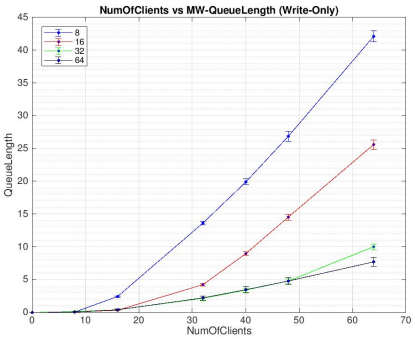
(b) Response Time-write.



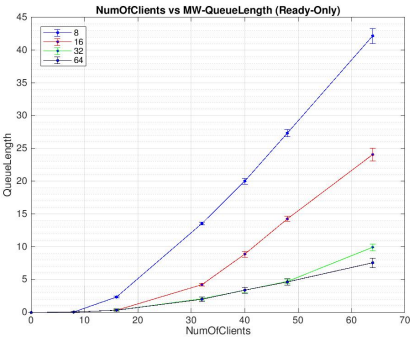
(c) Throughput-read.



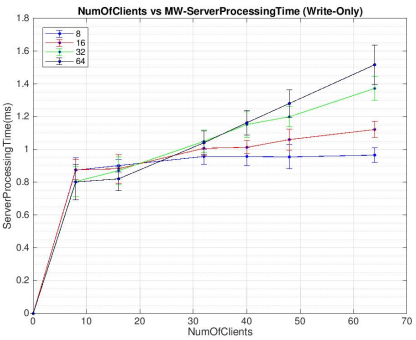
(d) Response Time-read.



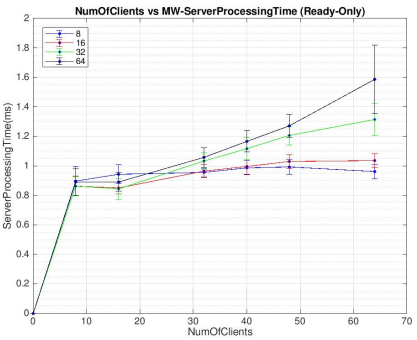
(e) Avg. Queue Length-write.



(f) Avg. Queue Length-read.



(g) Avg. Server Waiting Time-write.



(h) Avg. Server Waiting Time-read.

Figure 4: All measurement made by the MW

Workload	8 WT	16 WT	32 WT	64 WT
Write-only case	16	40	?	?
Read-only case	16	40	48	48

Table 2: Saturation point for each config.

3.1.4 Analysis of Results

a. **Write-only case**: The average response time in figure 4b is composed of the queue waiting time($T_{QueueTot}$), the MW processing time and the average time a worker waits for the response(s) from the server($T_{ServerTot}$), indicated in figure 4g. We observe that $T_{ServerTot}$ remains relatively stable for 8 and 16 WT, as the overall load on the server is low with low number of worker threads. Consequently the majority of the response time is consumed by requests waiting inside the queue to be processed, which can be seen in the average queue length increase in figure 4e. These observations lead us to the conclusion that with 8 and 16 WT, the low number of WT is the bottleneck of system, as the MW can't keep up with incoming requests and therefore the queue length increases rapidly. On the other hand we make the opposite observation with 32 and 64 worker-threads. There is a relatively large increase in $T_{ServerTot}$, whereas the average queue lengths remain rather small. Since the throughput is still increasing for these worker configurations, we need more clients to estimate the maximum achievable throughput and therefore determine the bottleneck.
Read-only case: Upon comparing figures 4d and 4h we make similar observations as in the write-only case. $T_{ServerTot}$ remains low with 8 and 16 workers. Therefore more than half of the response time is consumed by $T_{QueueTot}$ and the low number of WT is the bottleneck of the system. With 32 and 64 workers, the majority of the response time is consumed by $T_{ServerTot}$. As a result, the server processing time is the bottleneck of the system.

b. In order to find when the system saturates, we observed the throughput and average response time figures. The saturation point is then set equal to the number of clients, after which there is a significant increase in response time or the throughput starts to flatten. The results are summarized in table 2. A question mark indicates, that the saturation point has not yet been reached (the saturation point for this settings will be explored in the throughput writes section). In the case where the server is the bottleneck, we observe that the saturation point is at 48 clients for the read-only case, as predicted by the hypothesis. Furthermore we see that in the case where the MW is the bottleneck, the system reaches the saturation earlier than the saturation point of the server. Consequently the throughput curve starts to flatten with a lower number of clients and the maximum throughput achieved is lower than in the baseline experiment.

c. From figures 4e and 4f we observe for all worker configurations an increase in the avg. queue length with increasing number of clients. For 8 and 16 worker threads we observe a large increase in queue length, since the middleware does not have enough worker threads to process all incoming requests. Therefore more and more requests get queued, which increases the overall response time, which then decreases the overall throughput. However with 32 and 64 workers, the MW has enough resources to handle incoming requests without delay, therefore the queue length increase is relatively small.

d. From the results we can gather, that the maximum throughput for the write-only case is higher than for the read-only case for the workers 16, 32 and 64. On the other hand the

throughput with 8 workers is for both workloads almost identical.

3.2 Two Middlewares

Number of servers	1
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[8,16,20,24,32]
Workload	Write-only and Read-only
Number of middlewares	2
Worker threads per middleware	[8,16,32,64]
Repetitions	3 (1 minute each)
Warm-Up/Cool-Down phase	4 seconds

3.2.1 Key Questions

a. What are the differences compared to the one middleware case?

b. What remains the same compared to the one middleware case?

c. What is the bottleneck of the system?

d. When does the system saturate?

e. How does the average queue-length change by increasing the clients/worker-threads?

f. Can the maximum throughput be increased by adding another client machine?

3.2.2 Hypothesis

a. **Case MW is bottleneck**: We predict that the middleware will still be the bottleneck with 8 WT, as the number of workers is still to low to handle all incoming requests with large number of clients. On the other hand with 16 WT, the MW won't be a bottleneck anymore and the maximum throughput will only be limited by the performance of the server. We have seen in the previous section, that a low number of worker-threads was the main reason for the middleware being a bottleneck. By adding another MW machine, we double the number of worker-threads, hence we can handle more requests in a given time frame, which increases the total throughput and decreases the average queue length and response time of the entire system.

b. **Case Server is bottleneck**: We predict that the total throughput and response time values remain similar to the previous experiment. Adding another middleware machine won't make a difference to the overall performance of the system, since the workers of both MW instances need to wait for the responses from the server.

c. From both hypotheses above, we conclude that the main bottleneck of the system with up to 64 clients will be the MW if only 8 WT are used and otherwise the bottleneck will be the processing time of memcache server.

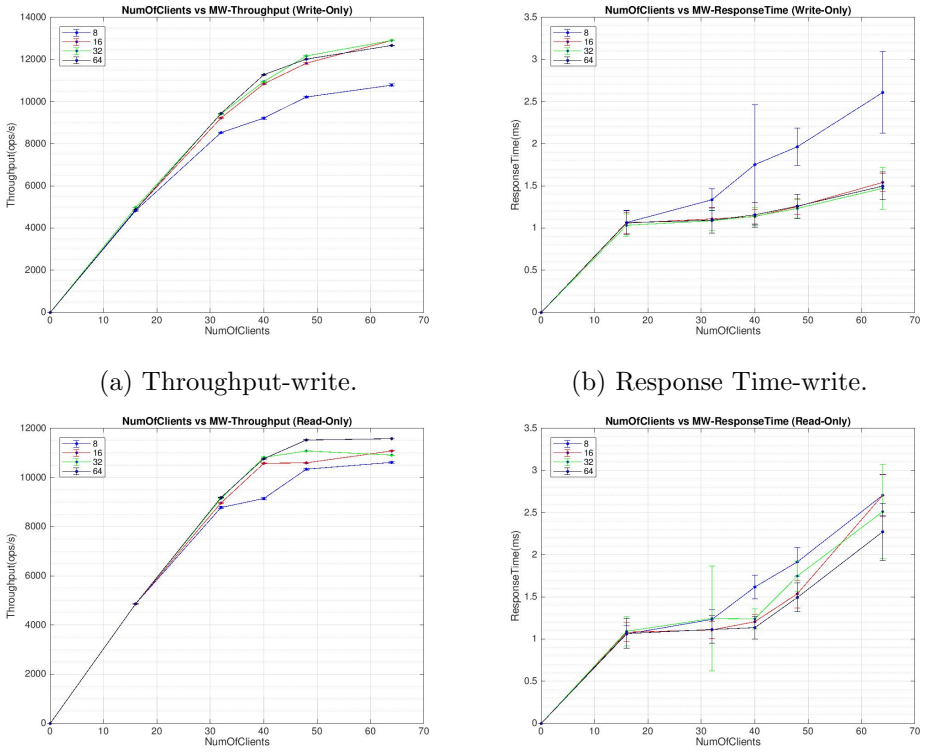
d. We expect that the saturation points coincide again with the saturation points for the server in the baseline experiment when the server is the bottleneck.

e. We assume that as the number of clients is increased, the average queue length of both middleware instances will increase, since the workers can't keep up with all the incoming requests at once. We further predict that as WT is increased the average queue length will decrease, as with more worker-threads we can handle more requests simultaneously.

f. For the write-only workload we predict that the overall throughput can be increased by adding another client machine. Form the first hypothesis above, the middleware won't be the bottleneck, when enough number of worker threads are used. Therefore the maximum throughput is only limited by the server performance.
On the other hand for the read-only performance the maximum throughput won't change much, as adding more clients will only saturate the server faster, as seen by the first baseline experiment.

3.2.3 Experimental Results

All results are depicted in figures 5, 6 and 7.



(a) Throughput-write.

(b) Response Time-write.

(c) Throughput-read.

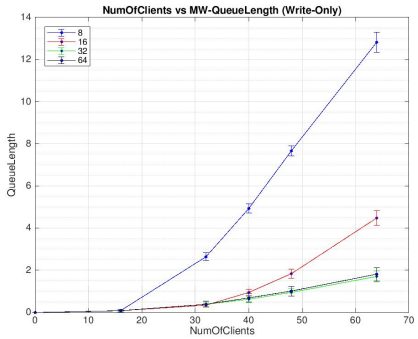
(d) Response Time-read.

Figure 5: Throughput and Response Time as measured by the MW. The measurements are aggregated over both middleware instances.

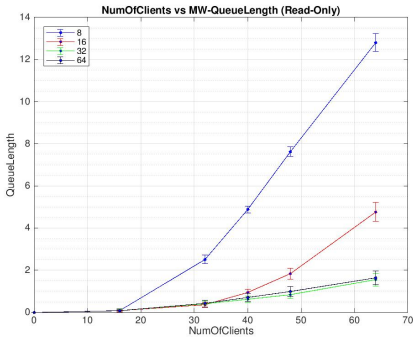
3.2.4 Analysis of the Results

a. **Write-only case**: Compared to the previous experiment, we observe from figure 5a and 5b that the maximum throughput has increased and the average response time has decreased for all worker configurations. Figure 6a further indicates that the average queue length has also decreased for all configurations. **Read-only case**: From figure 5c we observe that the maximum throughput has increased for all configurations compared to the one MW setting. This increase is small for 16,32 and 64 WT and large for 8 WT.

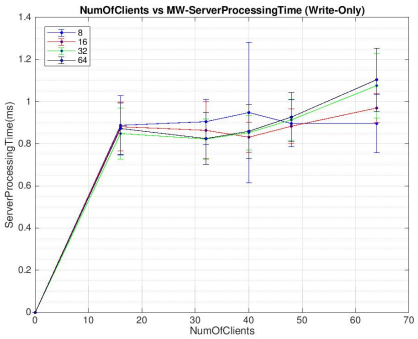
b. As predicted, for the cases where the server was the bottleneck, we only get a small increase in throughput, due to the fact the each worker thread has to wait for the responses from



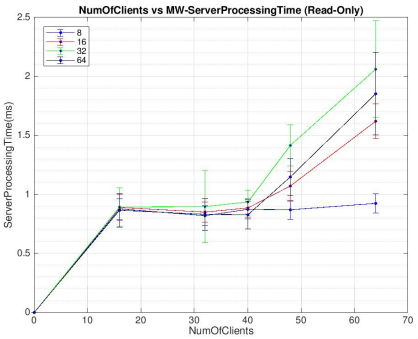
(a) Avg. Queue Length-write.



(b) Avg. Queue Length-read.



(c) Avg. Server Waiting Time-write.



(d) Avg. Server Waiting Time-read.

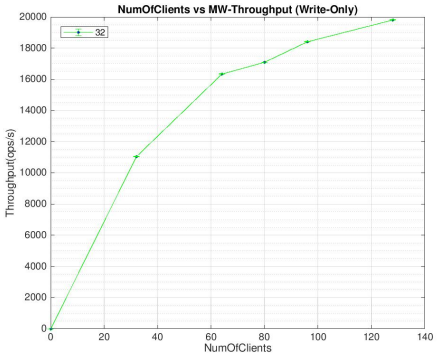
Figure 6: Avg. Queue Length and Server Waiting Time as measured by the MW.

the server. If the server is saturated this waiting time increases, which leads to larger response time values. The sudden increase in $T_{ServerTot}$ with large number of clients($>$ 48) can be observed for 16 WT and higher in figure 6d for the read-only case.

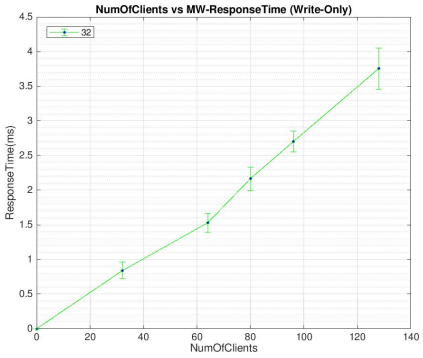
c. **Write-only case**: For 16 WT and higher the average response time in figure 5b is mainly consumed by the average server waiting time ($T_{ServerTot}$) depicted in figure 6c. As a result for 16 and more worker-threads and up to 64 total clients, the bottleneck of the system is the server processing time. However with 8 WT the average response time is mainly consumed by the average queue waiting time and therefore the low number of worker threads is the bottleneck of system. The increase in queue waiting time can be observed in figure 6a, where the average queue length after 16 total clients, increases rapidly for 8 WT.

   Read-only case: From figure 6d we observe a large increase in $T_{ServerTot}$ for 16 WT and higher. With 2 MW instances and therefore more worker-threads, the load on the server is higher and therefore it saturates faster for a read-only workload. In conclusion the average server processing time is the bottleneck of the system for 16 and higher worker-threads. For 8 WT, $T_{ServerTot}$ remains rather small and the queue waiting time consumes the majority of the average response time. Consequently the low number of worker-threads is again the bottleneck of the system.

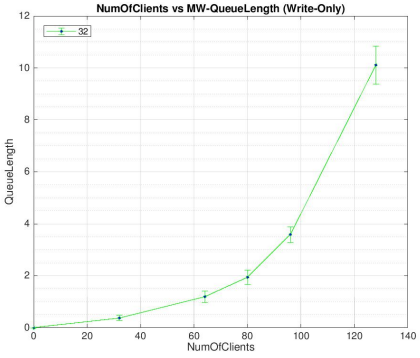
d. **Write-only case**: Figure 5a indicates the with 8 workers, the system already starts to saturate at around 32 clients, as the throughput starts to diverge from the other configurations and the queue length starts to increase rapidly with each step size. For the other configurations, the throughput is still rising and the average response time remains



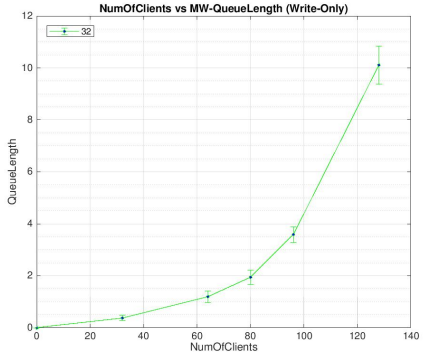
(a) Throughput-write.



(b) Response Time-write.



(c) Avg. Queue Length-write.



(d) Avg. Server Waiting Time-write.

Figure 7: MW measurements for a write-only workload, with 2 client machines and 32 WT

low. Therefore the system is not yet saturated with 16 WT and higher.
Read-only case: Here we observe from figure 5c and 5d that the system starts to saturate at around 32 clients for 8 WT and between 40-48 clients for the other configurations.

e. By inspecting figures 6a and 6b we observe an increase in the average queue length for increasing number of clients for all worker configurations. Additionally we see that the average queue size decreases with increasing number of worker threads. From these finding we can conclude, that with 8 workers per MW, we don't have enough workers to handle all incoming requests, which leads an increase in response time. Therefore at least 16 or more workers are needed in order to handle incoming requests efficiently up to 64 clients.

f. **Write-only case**: We repeated the experiment with 2 client machines and 32 worker threads. The results can be seen in figure 7. As predicted we get an increase in the throughput by adding more clients, therefore the middleware is not the bottleneck for the write-only case with up to 128 clients. Figures 7b and 7c indicate that the response time and average queue length increase rapidly towards 128 clients. Consequently the system is starting to saturate and we expect that the maximum achievable throughput will be lower than the maximum throughput achieved during the baseline experiment (25300 ops/s).
Read-only case: We also repeated the experiment with 2 client machines and 64 worker-threads, but the maximum throughput achieved was more or less identical to the maximum throughput with one client machine (11500 ops/s). This result was to be expected, as the server is already saturated, hence adding additional clients won't help to increase the maximum throughput of the system.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware	11108 ops/s	2.4 ms	0.74 ms	0.04
Reads: Measured on clients	10786 ops/s	5.73 ms	n/a	0.03
Writes: Measured on middleware	11493 ops/s	2.27 ms	0.84 ms	n/a
Writes: Measured on clients	11241 ops/s	5.51 ms/s	n/a	n/a

Table 3: Max. Throughput with one MW instance.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware	11541 ops/s	2.2 ms	0.37 ms	0.0
Reads: Measured on clients	11269 ops/s	5.52 ms	n/a	0.0
Writes: Measured on middleware	19813 ops/s	3.9 ms	1.41 ms/s	n/a
Writes: Measured on clients	19388 ops/s	6.41 ms	n/a	n/a

Table 4: Max. Throughput with two MW instances.

3.3 Summary

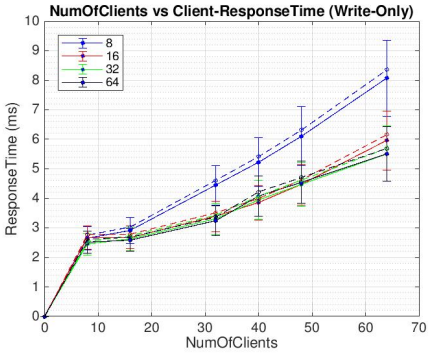
From the results in the tables 3 and 4 we can reach the following conclusions:

- For the case with one middleware, we failed to fill the cache of the memcache server properly and therefore got a cache miss rate of around 0.04. For the case with two MWs we avoided a cache miss, by writing all relevant data to the server before the start of the experiment.
- Adding an another middleware machine benefits the overall performance of the system. Especially in cases where the middleware is the bottleneck, an additional MW machine doubles the total number of worker threads and therefore the system can process a larger number of requests over a given time frame. On the other hand in cases where the server is the bottleneck, an additional MW machine only gives a small increase in the maximum throughput.
- For both experiments we got a bottleneck at the middleware when only using 8 worker-threads. In order to handle all incoming requests efficiently and keep the average queue length as small as possible, we need at least 32 or more worker threads. For the write-only case there was almost no difference in the results if 32 or 64 worker-threads were used for up to 64 total clients. However for the read-only case we got a higher throughput with 64 worker-threads.
- For large number of clients, write-only workloads have a lower response time and hence a higher throughput than read-only requests. The reason is that the server saturates faster(with less clients) with read -than write requests, as seen in the baseline experiment.
- The results from all experiments make sense given the implementation of the middleware. All performance critical code inside the MW, namely de-queuing requests, sending them to the server and sending back the response to the client, is done inside the worker-thread. Therefore adding more worker-threads to the system, increases the overall performance.

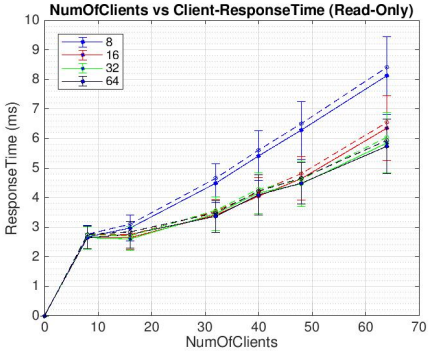
When enough workers are used, the main limitation comes from how long the worker has to wait for the responses from the server.
From all the results gathered by the middleware, we further conclude that the processing time of the net-thread is not a bottleneck of the middleware. Due to the non-blocking nature of the net-thread, it can handle increasing number of clients without delay. When analyzing the average response time, we observed that it is mainly consumed by $T_{QueueTot}$ and $T_{ServerTot}$ and the time for processing inside the net-thread and worker-threads is relatively small.

3.4 Interactive Law Verification

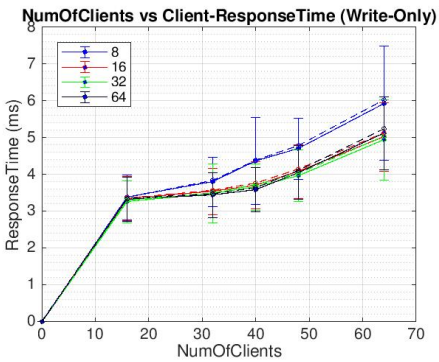
We will verify that the interactive law holds by predicting the response time using the throughput measured by the client. We assume that the think time for the client $Z = 0$ and compute $R_{pred} = \frac{NumClients}{Throughput}$



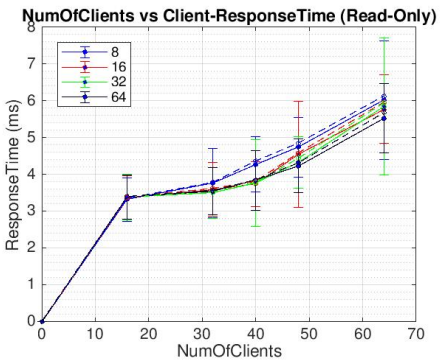
(a) Response Time - One MW - Write



(b) Response Time - One MW - Read



(c) Response Time - Two MW - Write



(d) Response Time - Two MW - Read

Figure 8: Results as measured by the clients. The throughput is added over all client machines. For the response time we take the average of the all client data. In the dashed lines we present the response time predicted by the interactive law.

From the figures above we observe that the predicted response time lies very close to the measured data at the memtier clients. The small discrepancies between prediction and measurement result from variances in the data, when we aggregate the throughput over multiple client machines. Therefore we conclude that the interactive response time law holds for all experiments throughout this section.

4 Throughput for Writes (90 pts)

4.1 Full System

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[8,16,20,24,32]
Workload	Write-only
Number of middlewares	2
Worker threads per middleware	[8,16,32,64]
Repetitions	3 (1 minute each)
Warm-Up/Cool-Down phase	4 seconds

4.1.1 Key Questions

a. What is the bottleneck of the system?

b. When does the system saturate?

c. How does the average queue-length change by increasing the clients/worker-threads?

4.1.2 Hypothesis

a. We predict from the previous experiment that the middleware will be the bottleneck if the number of worker-threads is low. Small number of workers can't handle all incoming requests fast enough and therefore we should see an increase in the average queue length. In contrast if WT is large enough, we expect that the main limiting factor of the system will be the memcache servers. Since the middleware instances replicate incoming requests to all 3 servers, the overall load on all servers will be high with a large number of workers. Therefore we expect that the servers will start to saturate and hence limit total performance of the system.

b. In the first baseline write-only experiment we found that the server saturated at around 96 clients. Due to replication, all servers are processing a large number of requests, therefore we expect that the throughput will start to flatten after 96 clients for the case where the server is the bottleneck.

c. Similar to the previous section we expect an increase in queue length with increasing clients. Furthermore we predict that the average queue length will decrease with increasing number of worker-threads.

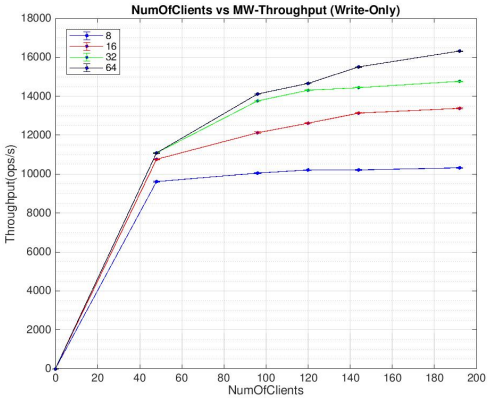
4.1.3 Experimental Results

In figure 9 we present the results, aggregated over all 3 clients and both middleware machines.

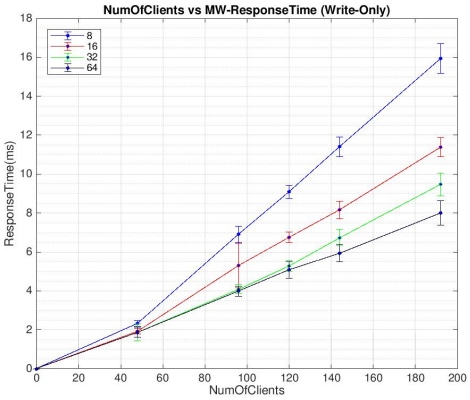
4.1.4 Analysis of Results

a. Our conclusion for the bottleneck analysis of the system is based on the following 3 observations:

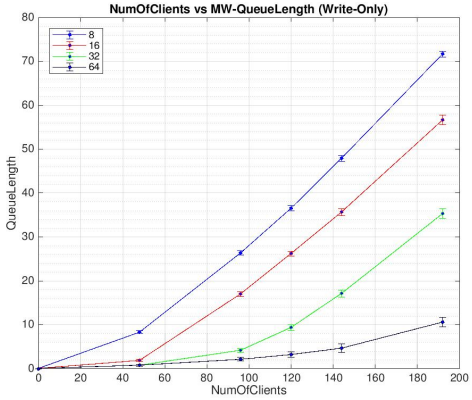
- Observation 1: Figure 9e indicates the average time spent inside the memcache servers ($T_{ServerTot}$). From this data we can gather, that $T_{ServerTot}$ is increasing with increasing number of clients for all configurations. Furthermore we observe an increase in $T_{ServerTot}$ with increasing number of worker-threads. This result is to be



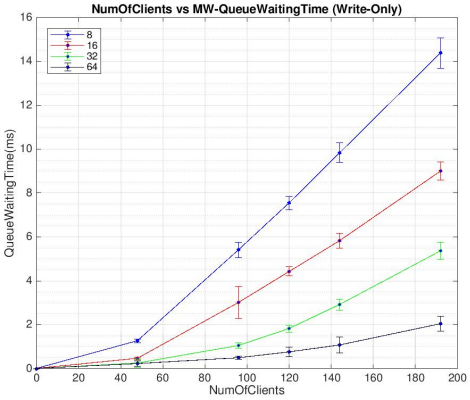
(a) Throughput



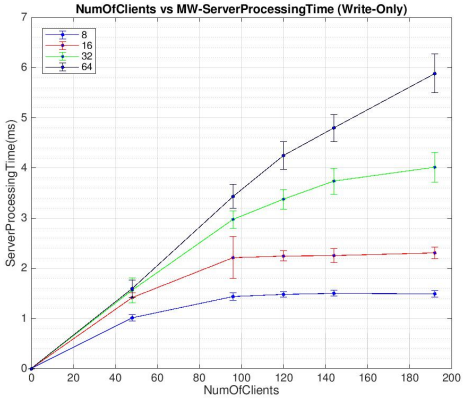
(b) Response-Time



(c) Avg. Queue Length



(d) Avg. Queue Waiting Time



(e) Time spent in Servers

Figure 9: Results as measured by the middleware instances

expected, as with more worker-threads, the middleware handles more requests in a second, which leads to a higher load on each server.

- Observation 2: Figure 9d indicates the average time a request spends inside the network queue ($T_{QueueTot}$). Here we again observe an increase in $T_{QueueTot}$ with increasing clients. We also see a decrease in $T_{QueueTot}$ with increasing number of workers. With more workers, more requests can be handled over a given time frame,

therefore requests spend less time in the queue. This result is also reflected in the average queue length in figure 9c.

- Observation 3: Figure 9b indicates the average response time value, which is the sum of $T_{QueueTot}$, $T_{ServerTot}$ and some additional processing time inside the middleware. From the figure we observe again a rise in the response time with increasing clients. Additionally, with higher number of workers we get lower response times.

Now we can conclude the following:

- With 8, 16 and 32 workers at 192 total clients, the middleware is the bottleneck of the system, as the majority of the response time is consumed by $T_{QueueTot}$. Since the middleware doesn't have enough workers to efficiently handle incoming requests with high number of clients, the requests start to queue up, which leads to an increase in the average queue length and $T_{QueueTot}$.
- With 64 workers, the server is the bottleneck of the system, as the majority of the response time is dictated by $T_{ServerTot}$. We give an example for the case with 192 clients: the response time with 64 workers is at around 8 ms. 5.9 ms are spent waiting for the memcache server, while only 2 ms are spent for requests waiting inside the queue.

b. For 8 and 16 WT saturation occurs after 48 clients. After 48 clients we observe a sudden increase in $T_{QueueTot}$ and response time, whereas $T_{ServerTot}$ remains almost constant. From figure 9a we observe a steep increase in throughput up to 96 clients for 32 and 64 WT. After this point the throughput curve starts to flatten. This is also reflected in the response time data in figure 9b, where we see a large increase in response time with each step-size after 96 clients.

c. As predicted, from figure 9c we observe a similar behaviour for the average queue lengths as in the previous sections. This result is also reflected in figure 9d, where with less worker-threads, the average queue waiting time per request increases.

4.2 Summary

	WT=8	WT=16	WT=32	WT=64
Throughput (Middleware) in ops/s	10316	13383	14756	16316
Throughput (Derived from MW response time)	12052	16871	19200	21333
Throughput (Client) in ops/s	10112	13070	14446	15901
Average time in queue in ms	14.37	9.01	5.36	2.04
Average length of queue	71	57	35	11
Average time waiting for memcached in ms	1.49	2.3	4.01	5.88

Table 5: Maximum throughput for the full system. The maximum throughput is achieved for 192 total clients for all WT configurations.

From these results we can conclude the following:

- As the number of workers are increased, we can increase the maximum throughput achieved by the system. More workers allow the middleware to process more requests simultaneously and therefore the overall throughput increases. Maximum performance for a write-only workload is achieved with 64 WT per middleware.

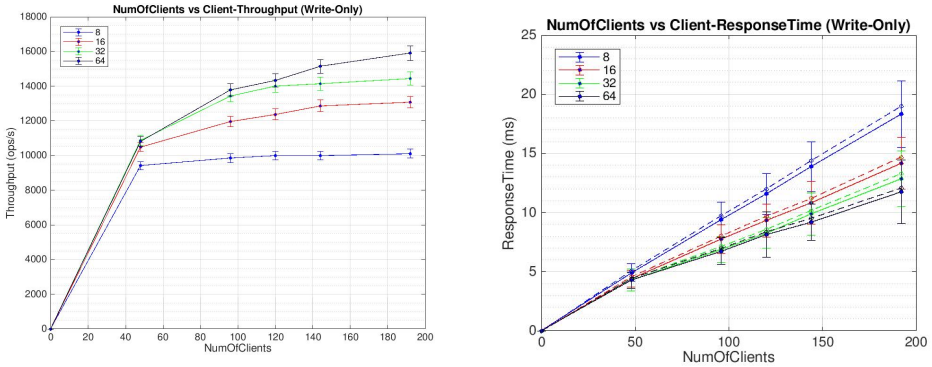
- For 8, 16 and 32 worker-threads the middleware is the bottleneck since most of the requests are waiting inside the queue to be processed and the queue lengths become very large at 192 clients. On the other with 64 workers, the servers seem to be the bottleneck, as most of the time the workers are waiting for the response from the memcached servers.

- There is a discrepancy between the measured and predicted MW throughput. This difference in throughput mainly arises from the fact, that the response time inside the middleware does not take into account the network latency needed to communicate with the clients. As a result the measured throughput values are smaller than the predicted ones.

- The results again correlate well with the implementation of the middleware. As explained in section 3, most of the performance critical code is done inside the worker-threads, therefore more workers help to increase the overall throughput and decrease the average response time of the system. The net-thread does not limit the performance of the system, as it can handle multiple incoming connections and requests in a non-blocking fashion. Additionally due to replication to all servers, the MW has to wait for all answers before sending the corresponding response(s) back. Due to our implementation inside the *SetRequestHandler.java* class, the responses of the servers are read sequentially in a loop (as explained in section 1.6.1), which can increase the overall response time of the MW. But since the answers of SET-requests are relatively small (< 10 bytes), we expect the impact of this overhead to be small on the overall response time.

4.3 Interactive Law Verification

We will verify that the interactive law hold by predicting the response time using the throughput measured by the client. We assume that the think time for the client $Z = 0$ and compute $R_{pred} = \frac{NumClients}{Throughput}$



(a) Throughput (b) Response-Time

Figure 10: Results as measured by the clients. The throughput is added over all client machines. For the response time we take the average out of all the client data. In the dashed line we present the response time predicted by the interactive law.

From figure 10b we observe that the predicted response is almost identical to the measured response time. The small discrepancies between prediction and measurement result from variances in the data, when we aggregate the throughput over multiple client machines. Therefore we conclude that the interactive response time law holds.

5 Gets and Multi-gets (90 pts)

5.1 Sharded Case

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	1:Key-Size
Multi-Get behavior	Sharded
Multi-Get size	[1,3,6,9]
Number of middlewares	2
Worker threads per middleware	64
Repetitions	3 (1 minute each)
Warm-Up/Cool-Down phase	4 seconds

5.1.1 Key Questions

a. What is the bottleneck of the system?

b. How does the average queue-length change with increasing key sizes?

c. How does the system behave with increasing multi-get key size(K_{mg})?

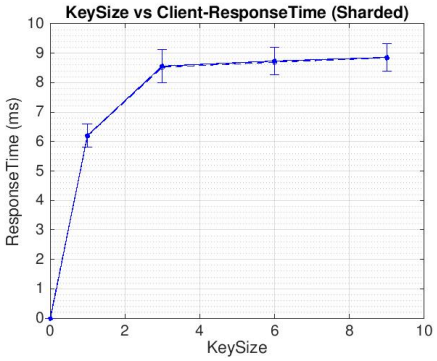
5.1.2 Hypothesis

a. We expect that the server processing time won't be the bottleneck of the system with 12 clients. Each server handles at most 36 get-requests(12 clients * 3 gets) simultaneously and from the baseline we know that saturation occurred after 48 clients with one get request each. Therefore the servers won't saturate for our current experiment settings.
We assume that the main bottleneck of the system is the middleware, in particular the processing time (P_{mg}) needed to handle large multi-get requests. P_{mg} is composed of sharding, sending requests and receiving responses and finally joining the responses. As the middleware has enough workers to handle all incoming requests and there is no saturation at the server, the performance for multi-get requests will depend on how efficiently the worker threads can process them. Whether sharding will help or hurt the system, will therefore depend on how short resp. how long P_{mg} is.

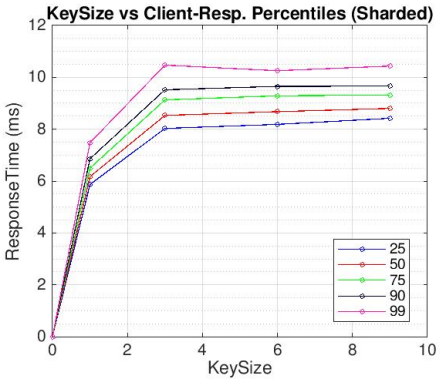
b. With larger key sizes, the workers will need slighty more time to process the multi-get requests. However, since there are only 12 clients and each middleware instance has 64 worker threads, we expect only few requests to wait inside the queue. Consequently we predict that the average queue length will be very small for all key sizes.

c. With increased K_{mg}, we expect the average response time of the system to slightly increase and the throughput to slightly decrease. With larger K_{mg}, the responses from the servers will be larger and therefore more time will be needed to send them over the network. However, at any given time the maximum amount of data sent from the server to the middleware is around 37 KB (12 clients * 3 keys(each server gets at most 3 keys due to sharding) * 1024 B). Therefore we expect the impact of increased K_{mg} to be small on our system.

5.1.3 Experimental Results

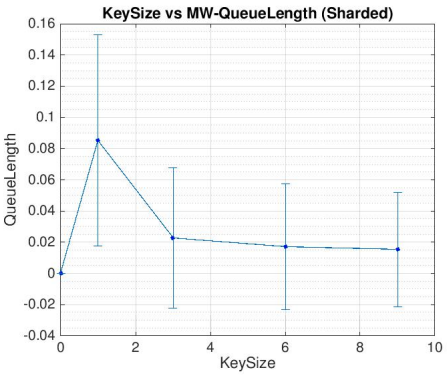


(a) Client-Resp. Sharded and Interactive Law(dashed line)

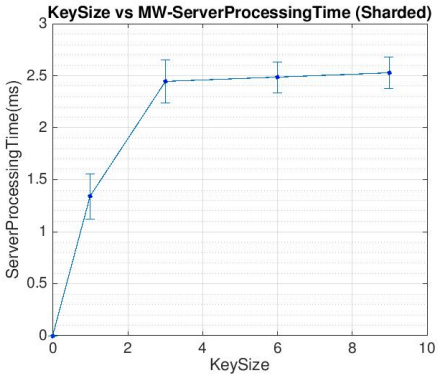


(b) Percentiles

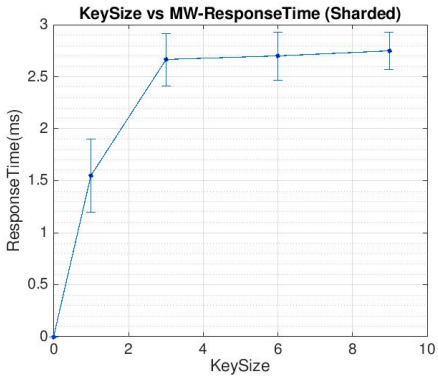
Figure 11: Response time and corresponding percentiles as measured by the clients.



(a) MW Queue-length



(b) MW Server Waiting Time



(c) MW Response Time

Figure 12: Middleware data for the sharded case. This data corresponds only to get and multi-get requests, the set-response time data is excluded.

5.1.4 Analysis of Results

a. From figure 12c we see that the average response time is mainly dominated by $T_{ServerTot}$, depicted in figure 12b. $T_{ServerTot}$ indicates the average time the middleware waits for the responses of the server, without including sharding or joining time. Why $T_{ServerTot}$ is the bottleneck of the system will be explained below.

b. Figure 12a indicates a queue length close to 0 for all key sizes. Additionally we observe a small decrease in queue length with increasing K_{mg}. With larger K_{mg} the client has to wait longer for the reply, as it takes more time to send the response over the network. Therefore the frequency at which the client sends requests is reduced and there are fewer requests waiting inside the queue.

c. From figure 11a we observe that the average response time is increasing with larger K_{mg}. This increase is large when going from a key size of 1 to 3 and small when increasing K_{mg} further. These findings are confirmed by the percentiles plot in figure 11b. Here we also observe a large increase for all percentile values from 1 to 3. After that the percentiles increase only to a small degree up to $K_{mg} = 9$.
In order to understand these results we take a closer look at the middleware results and the implementation of the *MultiGetRequestHandler.class*. When looking at the implementation of the *MultiGetRequestHandler.class*(see section 1, listing 4), we see the reason for this behaviour: With more than one 1 key in a multi-get request, the middleware sends each of the sharded requests sequentially in a loop to each server. Then the sharded responses are read by looping again through all servers, to which the worker has send a request to. Therefore we have the cost of sending and receiving larger messages on separate network connections and these costs are added up, due to the sequential reading of the responses. With one key the middleware sends and receives the entire message on one network connection, the same as the nonsharded case. Therefore the large increase in response time from 1 to 3 keys, is due to the sequential reading of the responses and the additional overhead of sending and receiving larger messages on separate networks. Joining and sharding time is also a reason for the increase in response time, but the impact of both is much smaller than $T_{ServerTot}$.

5.2 Non-sharded Case

As the system executes exactly the same code for the sharded and non-sharded case when $K_{mg} = 1$, the non-sharded experiment was only repeated for the other key sizes.

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	1:Key-Size
Multi-Get behavior	Non-Sharded
Multi-Get size	[3,6,9]
Number of middlewares	2
Worker threads per middleware	64
Repetitions	3 (1 minute each)
Warm-Up/Cool-Down phase	4 seconds

5.2.1 Key Questions

a. How does the system behave with increasing multi-get key size(K_{mg})?

b. What is the bottleneck of the system?

c. How does the average queue-length change with increasing key sizes?

d. How does the sharded and nonsharded system compare?

5.2.2 Hypothesis

a. The average response time will increase slightly, as more time is needed to send larger responses over the network from the server to the middleware. As at any given time the maximum amount of data on the network between server and middleware is at 9 KB(9 keys * 1024 Byte per response), we expect the impact of increased K_{mg} to be small on the system.

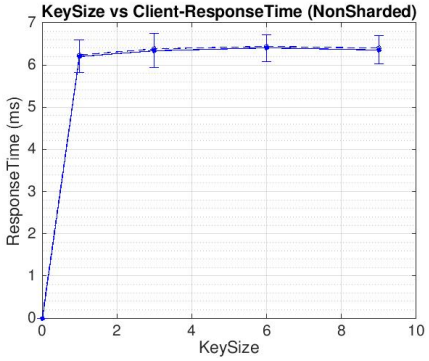
b. We predict that the middleware won't be the bottleneck of the system, as there is no overhead of sharding, joining and sending messages over multiple networks for the nonsharded case.
We expect the bottleneck of the system to be the processing time of the server thread and the additional network time needed to send large responses. In the worst case, a single server has to handle up to 108 (12 clients * 9 keys) get-requests simultaneously, if the middleware sends all requests to the same server. Since there is no sharding, the responses sent from the server to the middleware will be larger and as a consequence the time to send them over the network will be slightly larger. In conclusion, the main limiting factor for the performance will be the time to send larger messages over the network and how fast the server thread can process multi-get requests.

c. We expect the average queue length again to be very small, as with 64 workers, each request can be processed almost immediately for 12 clients.

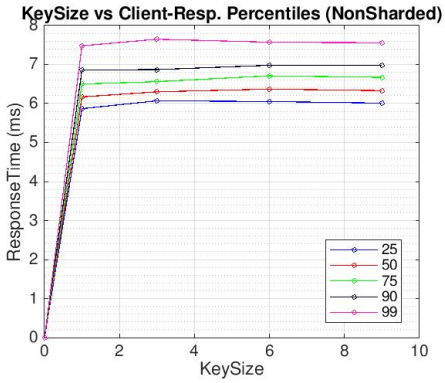
d. From the previous section we infer that the nonsharded version will be faster if the key size is small, like 3 or 6. For large K_{mg} more time is needed to send the responses from the server to the middleware and the load on the server is higher in the nonsharded setting. Therefore we predict that sharding will only outperform the nonsharded case for $K_{mg} = 9$.

5.2.3 Experimental Results

We note again that all results for $K_{mg} = 1$ have been taken from the sharded case.

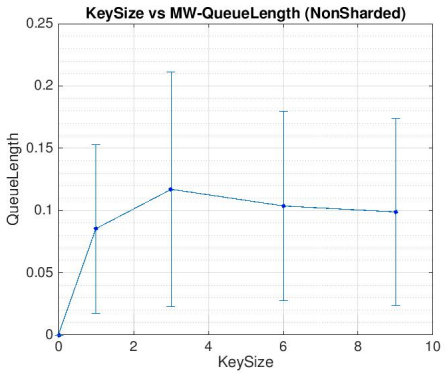


(a) Client-Resp. NonSharded and Interactive Law(dashed line)

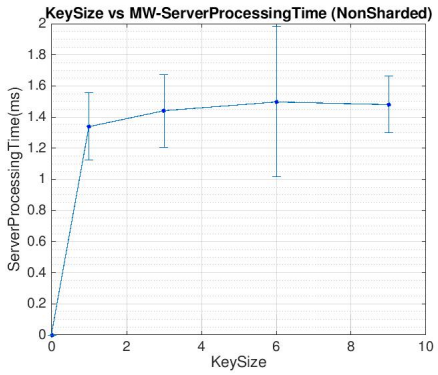


(b) Percentiles

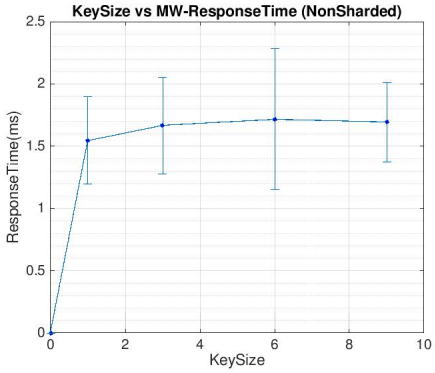
Figure 13: Response time and corresponding percentiles as measured by the clients.



(a) MW Queue-length



(b) MW Server Waiting Time



(c) MW Response Time

Figure 14: Middleware data for the nonsharded case. This data corresponds only to get and multi-get requests, the set-response time data is excluded.

5.2.4 Analysis of the Results

a. When observing the response time values at the client in figure 13a, we see that the response time increases slightly from $K_{mg} = 1$ to $K_{mg} = 3$. For the higher key-sizes the data remains more or less constant with only small variations. This behaviour is also mirrored in the percentiles data in figure 13b. The reason for this slight increase in response time can be found when inspecting the middleware data. From 14c and 14b we deduce, that the average response time of the middleware is almost completely consumed by $T_{ServerTot}$. As there is no overhead of sharding and joining, $T_{ServerTot}$ is determined by the server processing time and the network round trip time to send and receive the messages. Therefore the increase in $T_{ServerTot}$ is due to the larger messages, which takes the server longer to process and longer to send the responses back to the middleware. However, as the number of clients is low and the message sizes are only up to 9 KB, the impact of increasing K_{mg} is very small on the response time data.

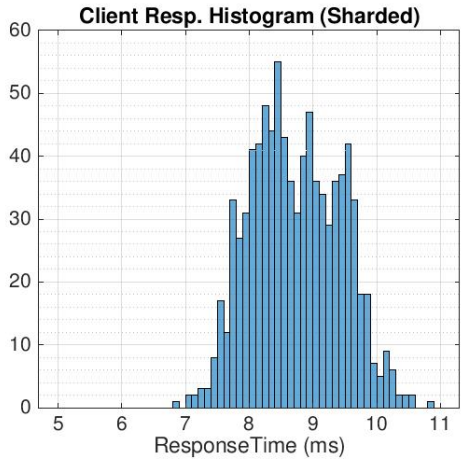
b. As explained above, the bottleneck of the system is the server processing time and the time needed to send large response from the server back to the middleware. This bottleneck is captured in the average server waiting time data, i.e. $T_{ServerTot}$.

c. From figure 14a we observe that the average queue lengths are close to 0, similar to the sharded case. Furthermore we observe a slight decrease in queue length with increasing K_{mg}. The reason for this is the same as in the sharded case.

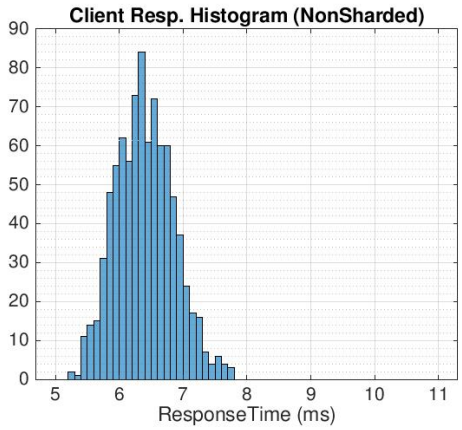
d. Contrary to the hypothesis, when comparing the sharded and nonsharded data for $K_{mg} >$

1, we observe that the response time values for the nonsharded case are always smaller than for the sharded one. When analyzing the percentile data for both cases this difference become even more evident: In the sharded setting the 25^{th} percentile is at 8.2 ms, i.e. 75% of the data is above 8.2 ms. On the other hand in the nonsharded settings, the 99^{th} percentile is at 7.8 ms or 99% of the data is below 7.8 ms. In conclusion, the nonsharded setting outperforms the sharded one with our current implementation. The reason for this difference will be explained in detail in section 5.4.

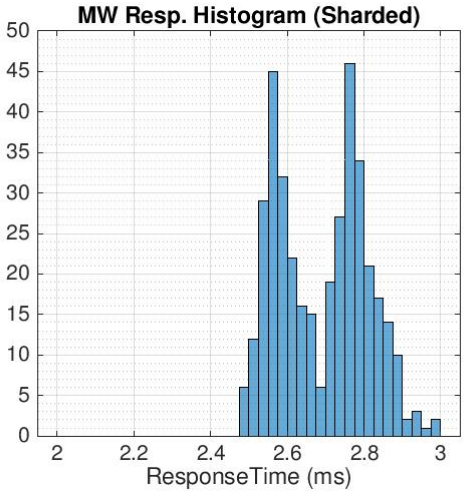
5.3 Histogram



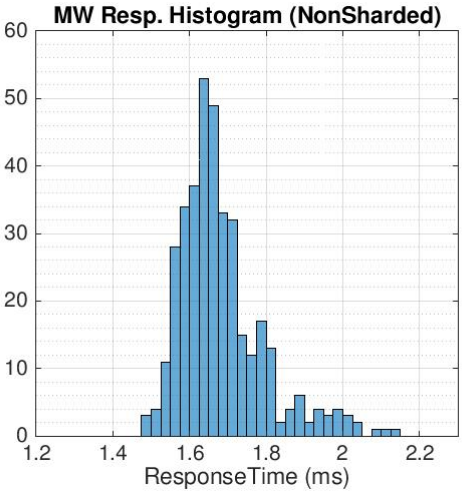
(a) Client-Resp. Sharded



(b) Client-Resp. NonSharded



(c) MW-Resp. Sharded



(d) MW-Resp. NonSharded

Figure 15: Histograms of response times as measured by the clients and the middleware instances. The bucket sizes from a to d are: 0.1, 0.1, 0.025, 0.025

On analyzing the histogram data, we observe the same behaviour as already seen in the previous sections: The average nonsharded response time is smaller than the sharded data. In figure 15a the sharded response time values are represented in buckets of 0.1 ms. The average response time is between 8 and 9 ms. In contrast, from figure 15b we see that average response time values for the nonsharded case are between 6 and 7 ms. Similar observations can be made when comparing the middleware histograms.

5.4 Summary

From all the observations made above we want to answer this central question:
Why is the nonsharded version outperforming the sharded one?

The answer can be found upon inspecting the middleware data for both settings. When comparing the average waiting time for the server($T_{ServerTot}$) for $K_{mg} > 1$ in figures 12b and 14b, $T_{ServerTot}$ is at 2.5 ms for the sharded and around 1.5 ms in the nonsharded version. There are 2 reasons for this difference:

- As there are only 12 clients, the load on the servers is low and therefore the server threads can process requests without delays. Even with the highest key size of 9, the average response time in the nonsharded setting remained relatively stable and therefore well below the average response time of the sharded case.
- The main reason for the difference lies in the implementation of the *MultiGetRequestHandler.class*. As explained in section 5.1.4 c, the sharded responses from each of the servers are read sequentially in a loop, therefore adding up the cost of receiving the responses from different networks. Due to the sequential read, the middleware looses the advantage it gained from sharding. On the other hand, in the nonsharded implementation, the middleware only receives the response on one network without the need for a loop. As a result, the nonsharded implementation outperforms the sharded version. In order to make sharding a viable option for our system, we should parallelize the reading of the responses in the sharded code. Finally we note that the sharding and joining time introduces a slight overhead in the response time, but this overhead is neglible when comparing them to $T_{ServerTot}$.

In conclusion, due to an inefficient implementation of the sharded code, the nonsharded version is the preferred option for our system for a low number of clients. However, when the number of clients is increased and the servers start to saturate, the sharded option might be the preferred one.

5.5 Interactive Law Verification

We plotted the response time predicted by the interactive law in figure 11a for the sharded case and figure 13a for the nonsharded case. We observe that the predicted and measured values are almost identical for both cases and conclude that the interactive response time law holds.

6 2K Analysis (90 pts)

The following table contains all the settings used throughout this experiment.

Number of servers	2 and 3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	32
Workload	Write-only, Read-only, and 50-50-read-write
Number of middlewares	1 and 2
Worker threads per middleware	8 and 32
Repetitions	3 (1 minute each)
Warm-Up/Cool-Down phase	4 seconds

6.1 Key Questions

a. Which of the 3 factors have the most impact on the performance of the system?

b. Which configuration gives the maximum throughput?

c. How do the results differ for each workload?

6.2 Hypothesis

a. **Number of workers**: Will have a high impact on the overall performance of the system. As seen in previous sections, having few workers, lead to a large increase in the average queue length which therefore increases the average response time of the system. More workers allow the system to process more requests per second. Consequently the overall throughput of the system is increased.
Number of MW machines: Will also have a high impact, as this is coupled with the first factor. More MW machines means more workers, which leads again to a higher throughput in the system.
Number of Servers: Will a have low impact on the system, compared to the factors above. Due to replication, we expect that more servers lead to a slightly larger response time for write-only workloads. As seen in the first baseline experiment, balancing the load among the servers can benefit the performance of the system for a read-only throughput. But we expect that the differences in throughput and response time between 2 and 3 servers to be minimal.

b. From hypothesis a., we predict that the maximum throughput will be achieved with 2 middleware machines and 32 worker-threads for all three workload configurations. Furthermore we expect that for a write-only workload the maximum throughput will be higher with 2 servers instead of 3, due to replication inside the middleware. For the read-only workload we expect that the throughput is slightly higher with 3 servers since the client load can be balanced better among the memcached instances. For the read-write workload we expect that the maximum throughput will be an average of the maximum throughput of the write-only and read-only case.

c. **Write-only**: We expect, that the impact of the number of servers, is much higher than for the read-only case. Increasing the amount of servers, increases the time to replicate all incoming requests to all servers. Consequently the average response time goes up and the average throughput goes down.
Read-only: We assume that the impact of S will be small and that the other factors play a much bigger role. Read-Write: Here we expect behaviour, which is a mixture of the write-only and read-only case, as the read-write ratio is 50:50.

6.3 Experimental Results

Since plotting the results of the 2k experiment can be ambiguous, we summarized the experimental results in the following tables. The tables contain the throughput and response time as measured by the client. Additionally in brackets beside the response time value, we give the response time values computed by the interactive law.

1 MW			2 MW		
	8 Workers	32 Workers		8 Workers	32 Workers
2 Servers	3547	8991	2 Servers	11350	16981
3 Servers	3576	8301	3 Servers	7430	14200

Table 6: Write-Only Throughput (ops/s)

1 MW			2 MW		
	8 Workers	32 Workers		8 Workers	32 Workers
2 Servers	53.97 (54.1)	21.45 (21.3)	2 Servers	19.26 (17)	11.63 (11.3)
3 Servers	53.73 (53.7)	23.22 (23.1)	3 Servers	25.8 (25.8)	13.61 (13.5)

Table 7: Write-Only Response Time (ms)

1 MW			2 MW		
	8 Workers	32 Workers		8 Workers	32 Workers
2 Servers	4288	11231	2 Servers	14178	22049
3 Servers	5516	11554	3 Servers	13328	22563

Table 8: Read-Only Throughput (ops/s)

1 MW			2 MW		
	8 Workers	32 Workers		8 Workers	32 Workers
2 Servers	44.8 (44.7)	17.2 (17.1)	2 Servers	15.8 (13.5)	9 (8.7)
3 Servers	34.9 (34.8)	16.7 (16.6)	3 Servers	14.7 (14.4)	8.7 (8.5)

Table 9: Read-Only Response Time (ms)

1 MW			2 MW		
	8 Workers	32 Workers		8 Workers	32 Workers
2 Servers	4109	9555	2 Servers	12594	19533
3 Servers	4499	9766	3 Servers	9627	17408

Table 10: Read-Write Throughput (ops/s)

1 MW			2 MW		
	8 Workers	32 Workers		8 Workers	32 Workers
2 Servers	46.7 (46.7)	20.2 (20.1)	2 Servers	17.2 (15.2)	10.1 (9.8)
3 Servers	42.7 (42.7)	19.7 (19.6)	3 Servers	19.9 (19.9)	11.1 (11)

Table 11: Read-Write Response Time (ms)

6.4 2k-Analysis

The analysis was done with 3 factors (*S=#servers, M=#MWs, W=#workers*) and 3 replications. We now explain in detail how the results of the analysis were computed:

a. Variables:

$$x_S = \begin{cases} -1 & ,S = 2 \\ 1 & ,S = 3 \end{cases}, \qquad x_M = \begin{cases} -1 & ,M = 1 \\ 1 & ,M = 2 \end{cases}, \qquad x_W = \begin{cases} -1 & ,W = 8 \\ 1 & ,W = 32 \end{cases}$$

b. Model:

$$y = q_0 + q_S x_S + q_M x_M + q_W x_W + q_{SM} x_S x_M + q_{SW} x_S x_W + q_{MW} x_M x_W + q_{SMW} x_S x_M x_W,$$

We chose the additive over of the logarithmic model. The main problem with the logarithmic model was, that the variation due to the S factor was slightly smaller and the variation due to the other 2 factors were also slightly reduced. As seen in the previous experiments, the number of middleware instances and workers are the most important factors for the performance of our middleware and therefore we choose the additive model.

c. Estimate the effect-parameters q by using the sign table method.

d. Estimate the allocation of variation using the sum of squares:

$$\begin{aligned} c =& 2^k r = 2^3 * 3 \\ SST =& SSS + SSM + SSW + ... + SSSMW + SSE \\ =& c * q_S^2 + c * q_M^2 + c * q_W^2 + ... + c * q_{SMW}^2 + \sum_{i,j} e_{ij}^2 \\ VarSSS =& SSS/SST, ..., VarSSSMW = SSSMW/SST \text{ (Variation of each effect)} \\ S_{Err} =& \sqrt{SSE/2^k(r-1)} \text{ (Standard deviation of errors)} \\ S_{Eff} =& S_{Err}/\sqrt{2^k r} \text{ (Standard deviation of effects)} \end{aligned}$$

6.4.1 Results

We summarized the results in the following tables:

Parameters	q_0	q_S	q_M	q_W	q_{SM}	q_{SW}	q_{MW}	q_{SMW}	S_{Err}	S_{Eff}
Mean Estimate (ops/s)	9297	-920	3193	2821	-754	52.6	279	232	138.6	28.3
Variation %	n/a	0.043	0.517	0.4	0.029	1.4e−4	0.0039	0.0027	n/a	n/a

Table 12: Write: Results for the throughput analysis

Parameters	q_0	q_S	q_M	q_W	q_{SM}	q_{SW}	q_{MW}	q_{SMW}	S_{Err}	S_{Eff}
Mean Estimate (ms)	27.8	1.26	-10.3	-10.4	0.87	-0.32	5.4	-0.82	0.4	0.08
Variation %	n/a	0.0065	0.4294	0.4384	0.0032	4e−4	0.12	0.003	n/a	n/a

Table 13: Write: Results for the response time analysis

Parameters	q_0	q_S	q_M	q_W	q_{SM}	q_{SW}	q_{MW}	q_{SMW}	S_{Err}	S_{Eff}
Mean Estimate (ops/s)	13088	152	4941	3761	-235	57.3	516	284	363	74.2
Variation %	n/a	5.9e−4	0.62	0.36	0.0014	8.4e−5	0.007	0.002	n/a	n/a

Table 14: Read: Results for the throughput analysis

Parameters	q_0	q_S	q_M	q_W	q_{SM}	q_{SW}	q_{MW}	q_{SMW}	S_{Err}	S_{Eff}
Mean Estimate (ms)	20.2	-1.48	-8.2	-7.3	1.13	1.27	4.1	-1.09	0.46	0.09
Variation %	n/a	0.015	0.46	0.37	0.009	0.011	0.12	0.008	n/a	n/a

Table 15: Read: Results for the response time analysis

Parameters	q_0	q_S	q_M	q_W	q_{SM}	q_{SW}	q_{MW}	q_{SMW}	S_{Err}	S_{Eff}
Mean Estimate (ops/s)	10886	-561	3904	3179	-711	82.9	500.8	127.7	262.2	53.5
Variation %	n/a	0.012	0.57	0.38	0.02	2.5e−4	0.01	6.2e−4	n/a	n/a

Table 16: Read-Write: Results for the throughput analysis

Parameters	q_0	q_S	q_M	q_W	q_{SM}	q_{SW}	q_{MW}	q_{SMW}	S_{Err}	S_{Eff}
Mean Estimate (ms)	23.4	-0.08	-8.8	-8.2	1.04	0.23	4.2	-0.67	0.52	0.1
Variation %	n/a	4.3e−5	0.47	0.4	0.006	3.3e−4	0.1	0.003	n/a	n/a

Table 17: Read-Write: Results for the response time analysis

6.4.2 Analysis of the results

a. In the tables above, we indicated the factors with the biggest impact in blue.
Number of workers: For all 3 workloads, the W factor has the second highest impact on all workloads, except for the response time data in table 13, where it has a slightly higher impact than M. Due to the implementation of the middleware, most of the performance critical code is executed inside the worker-threads. Consequently increasing the W factor increases the average throughput and decreases the average response time.
Number of MWs: For all workloads, the M factor has the highest impact on for all workloads, except for the response time in table 13. Only increasing the number of worker-threads on a single machine gives only a limited performance boost, as the number of threads that can be run simultaneously are limited by the resources of the machine. Therefore adding an additional middleware instance, almost doubles the number of worker-threads that can be run in parallel and gives a significant increase in throughput and decrease in response time. Furthermore increasing the M factor is directly correlated with increasing the W factor. As a result, our system can process more requests in parallel in a certain amount of time.
Number of Servers: The S factor has the third highest impact on throughput for the write-only workload. Adding another server to the system, increases the overall processing time of a worker-thread for the write-only case. The worker needs more time to replicate the request to all servers and more time to receive all corresponding responses. Therefore increasing S decreases throughput and increases response time for the write-only workload.
On the other hand for the read-only workload, adding another server slightly reduces the response time and increases the throughput of the system. As seen in the baseline experiment in section 2.2, an additional server allows for better load-balancing and therefore the

servers don't saturate as fast. For the read-write case, S has a large impact on throughput but less than the write-only case, as only half of the requests are set-requests.
Effect of Middleware and Worker(MW:) For the read-only setting and the response time of the write-only and read-write settings, MW has the third highest impact on the system. As M and W have already the highest impact and are directly correlated, it makes sense and that the combined effect of increasing both factors would increase the overall throughput of the system. Especially for the read-only case, where the impact of S is lower, the impact of MW is higher.
Effect of Server and Middleware (SM): For the read-write case SM is the third highest factor for throughput. This finding results from the fact that S has a large impact on throughput for the write-portion of the requests, whereas M has a large impact on the throughput of read requests. The read-write ratio is 50:50, hence the combined effect of S and M creates a large impact on throughput.
Effect of other factors: As changing the number of servers is mainly independent of changing the number of workers (when we fix W and M and only change S, the difference in throughput and response time is relatively small compared to changing M or W), SW and SMW have a relatively small impact on the system, compared to the other factors..

b. **Write-only**:Maximum throughput is achieved with 2 servers, 2 MWs and 32 workers.
Read-only:Maximum throughput is achieved with 3 servers, 2 MWs and 32 workers.
Read-Write:Maximum throughput is achieved with 2 servers, 2 MWs and 32 workers.

c. **Impact of S**: The impact of S is much smaller on the read-only case than the write-only setting, since there is no overhead of replication for read-requests.
Consequently we see a higher mean throughput q_0 for the read-only workload. This difference stems from the fact that q_S's negative impact on the write-only case, reduces the overall throughput. As expected, for the read-write case, we observe a mixture of the write-only and read-only case.

6.5 Interactive Law Verification

In the experimental results section, we observe that for the most part the predicted and measured response time data lie close to each other. Therefore we conclude that the interactive law holds for all experiments conducted throughout this section.

7 Queuing Model (90 pts)

For each of the following sections, we first define the input parameters λ (mean arrival rate) and μ (mean service rate) of our queuing model. We then proceed to compute the output of the model based on Little's Law and other formulas presented in the book. Finally we compare the predicted values to the measured ones and explain the difference.

7.1 M/M/1

In this section we model our entire system as an M/M/1 queue. Therefore we assume that the interarrival times and service times are exponentially distributed and only one server(i.e our system) operates on the queue. The following calculations are performed for each worker-thread configuration.

7.1.1 Calculations

- **Input Parameters:** We set the mean arrival rate λ equal to the average throughput measured on the client side. We define the mean service rate μ as the highest measured throughput at the middleware instances. Selecting the parameters this way, ensures that the system utilization $\rho < 1$, hence our system is stable. The values for both parameters haven been directly taken from table 5 in section 4.
- **Output Parameters:** We compute the traffic intensity/utilization ρ, the average service time per job $E[s]$, average queue length $E[n_q]$, average queue waiting time $E[w_q]$, average response time $E[w]$ and the mean number of jobs in the system $E[n]$. These output parameters allow a direct comparison with the measured data in section 4.
- **Results**:

WT	8	16	32	64
λ	10112	13070	14446	15901
μ	10316	13383	14756	16316
$\rho = \lambda/\mu$ (%)	98	97.66	97.9	97.46
$E[s] = 1/\mu$ (ms)	0.097	0.075	0.068	0.062
$E[n_q] = \rho^2/(1-\rho)$	48.58	40.78	45.62	37.34
$E[w_q] = E[n_q]/\lambda$ (ms)	4.8	3.12	3.15	2.34
$E[w] = E[w_q] + E[s]$ (ms)	4.9	3.2	3.22	2.4
$E[n] = \rho/(1-\rho)$	49.56	41.75	46.6	38.31

Table 18: Results of the M/M/1 model

7.1.2 Differences

When observing the predicted and observed data, several differences can be found between them. These discrepancies arise from the fact that we try model a parallel system(our middleware) as a sequential system(the M/M/1 model). By treating the entire system as one service, the M/M/1 model hides the parallelism in our middleware, therefore the predicted data doesn't match the observed data from section 4. The following discrepancies are all caused by forcing a parallel system into a sequential model:

- If we look at the mean service time $E[s]$, we observe something completely different from the measured service time(which we get by subtracting the queue waiting time from the total response time in table 5). $E[s]$ is below 1 ms in the model, which is not the case in the experiment, as each thread processes a request several milliseconds. Furthermore the mean service time seems to be decreasing with increasing number of workers. This trend is not true in our system, as with increased workers, the waiting time for the server goes up. Therefore $E[s]$ should increase, with increasing number of workers.
- We also observe a difference between the predicted and average queue length. With 8 and 16 workers, the measured values are significantly higher (71 vs. 49 and 57 vs. 42). On the other hand, the measured values with 32 and 64 workers are lower(35 vs 46 and 11 vs. 37). Since $E[n_q]$ is computed using only ρ, this model doesn't predict the behaviour of our multi-threaded system, which results in the discrepancy.
- Due to the same reason as above, we again get a difference between the measured and predicted queue waiting time. However the trend of decreasing queue length with increasing number of workers is preserved in the model.
- Another observation is that most of the total waiting time $E[w]$ is completely dominated by the average queue waiting time $E[w_q]$. This should not be the case, as we have seen in section 4, the time waiting for memcache server plays a significant role for 32 and 64 workers.
- Lastly, the number of requests present in the system $E[n]$ should be higher than the values predicted by the model. The M/M/1 model only takes into account the requests waiting inside the queue and not the jobs currently in service $E[n_s]$ at any given time. Therefore with m worker threads, there should be at least m jobs in service, assuming that each worker-thread can be run in parallel.

7.2 M/M/m

In this section we model our entire system as an M/M/m queue. Each middleware worker-thread is treated as one service. As we have two middleware instances running simultaneously, we are running 16, 32, 64 and 128 worker-threads in total. We expect this model to capture the multi-threaded nature of our system better than the M/M/1 model.

7.2.1 Calculations

- **Input Parameters:** Since we have two middleware instances running in parallel, we set $m = 2 * WT$. We set the mean arrival rate λ again equal to the maximum throughput measured on the client side. We set the mean service time $E[s]$ equal to the average response time measured on the middleware minus the average queue time for the case with 192 total clients. From $E[s]$ we can compute μ and the other necessary output parameters. Selecting the parameters this way, ensures again that the system utilization $\rho = \lambda/(m\mu) < 1$, hence our system is stable.
- **Output Parameters:** We compute the same output values as in the previous section. Additionally we compute the probability that an arriving job has to wait in the queue ϱ, as this probability is needed to compute some of the other output parameters.
- **Results**:

WT	8	16	32	64
λ	10112	13070	14446	15901
$\mu = 1/E[s]$	667	435	244	167
m	16	32	64	128
$\rho = \lambda/m\mu$ (%)	94.75	93.9	92.5	74.4
$\varrho = \frac{(m\rho)^m}{m!(1-\rho)} p_0$ (%)	77	63.75	43.4	0.08
$E[s]$ (ms)	1.5	2.3	4.1	6
$E[n_q] = \rho\varrho/(1-\rho)$	14	10	5	0
$E[w_q] = E[n_q]/\lambda$ (ms)	1.37	0.75	0.37	0.0002
$E[w] = E[w_q] + E[s]$ (ms)	2.87	3	4.47	6
$E[n] = m\rho + \rho\varrho/(1-\rho)$	30	40	65	95

Table 19: Results of the M/M/m model

7.2.2 Differences

We again notice differences between the predicted values of the M/M/m model and the observed values in the middleware. The main cause for this differences are due to the fact, that the model is a high level abstraction of the real system, therefore we miss several factors. The model assumes that each worker has unlimited resources and do not have to share the network connection with other threads. Furthermore the model neglects the fact, that in real systems, threads have to synchronize on the cpu and they can't always run simultaneously.

- The queue length $E[n_q]$ is again lower than the observed values for all worker configurations. The main reason for this discrepancy stems from the fact that the formula for $E[n_q]$ can't capture the complex behaviour of multiple threads on a machine with limited number of cpus. As seen in section 4, with 8,16 and 32 workers per middleware, our system begins to saturate after a certain number of clients, as the workers can't process all incoming requests simultaneously. As a result the requests start to pile up in the queue and the queue length increases rapidly. The reason for a low $E[n_q]$ becomes clear when looking at the formula to compute p_0. As m goes to infinity, p_0 goes to 0. Even with finite values for m, p_0 is on the order of 10^{-6}. Since the probability of queuing ϱ also depends on p_0, this causes the average queue length and average queue waiting time to be much smaller than the measured values.
- The average queue waiting time $E[w_q]$ does not match the observed values, due to the same reason as for $E[n_q]$. $E[w_q]$ should be significantly higher for all worker configurations.
- We also observe that the average response time is increasing with increasing number of workers. This is the exact opposite of what happens in our middleware. Since $E[w] = E[w_q] + E[s]$, the main reason for the difference is that $E[w_q]$ is too low and therefore the queue waiting time is not accurately represented in the overall response time.

7.2.3 Similarities

With the M/M/m model we can directly map the number of services m, to the number of worker-threads in our system. We notice the following similarities between observed and predicted data:

- The model preservers again the tendency of the middleware, that with more workers, the average queue length and queue waiting time decreases.

- We also notice that the predicted number of requests in the system ($E[n]$) at any given time is closer to what we would expect in the middleware. With more workers more requests are processed simultaneously. However, because not all workers can be run in parallel, we would expect $E[n]$ to be close to the number of requests waiting inside queue plus the requests being currently handled by the workers.

In summary the M/M/m model does a better job at capturing the multi-threaded behaviour in our system, but its main drawback - the underestimation of queue length and queue waiting time - renders it useless for practical purposes.

7.3 Network of Queues

In this section we model our system as a closed network of queues. We will first present our queuing network and proceed with explaining the configurations used. Finally we will perform an extended mean value analysis and present its results.

7.3.1 Queuing Network

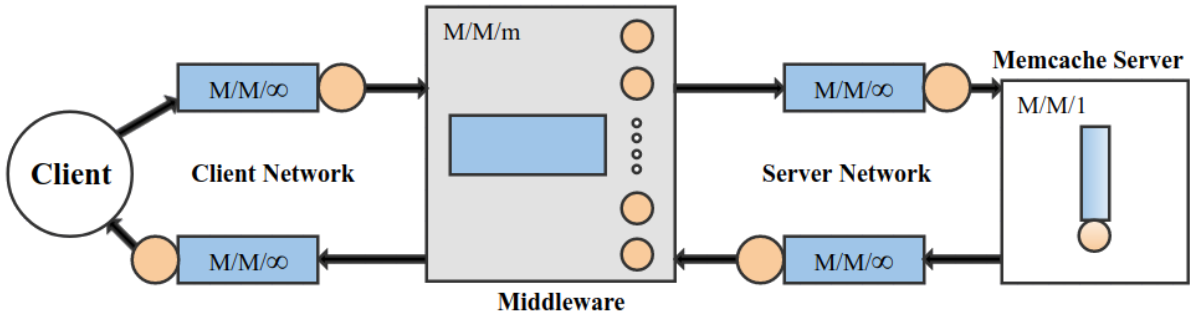


Figure 16: An overview of the closed queuing network used to model our system. The blue rectangles represent queues and the orange circle the corresponding service working on the queue.

Figure 16 shows an overview of our queuing network. In total there are 4 different devices in our model: Client Network, Middleware, Server Network and the Memcache Server. In the following sections we will describe each of the devices in more detail:

7.3.1.1 Client Network This device represents the network latency between the client and the middleware. Each direction (i.e. sending and receiving) of the network is modeled as a delay center, since we assume that jobs never have to wait inside the queue and the response time is equal to the service time. We assume the service time for both delay centers to be equal. Since we based our model on section 3, we have one client machine in total.

7.3.1.2 Middleware We represent our middleware as a load-dependent service center,i.e. as an M/M/m model. We choose the M/M/m model, as it correlates well with the internal architecture of our middleware: we have one queue present and m worker-threads processing the requests simultaneously from the queue. We simplify the case when two middleware instances are present in the system: We model them both with one M/M/m device and simply double the number of worker threads, i.e. double the m value of our model.

7.3.1.3 Server Network This network represents the network connection between the middleware and the server. We modeled these connections again as delay centers and furthermore we assume that both sending and receiving models have the same service time.

7.3.1.4 Memcache Server We model the memcache server as a fixed capacity(M/M/1) center. Since we based our model on section 3, we have one server machine in total. Therefore the M/M/1 model fits our server well, as the server will have one queue and one thread processing incoming requests and sending out responses.

7.3.2 Configurations

In order to evaluate our queuing network, we will perform an extended mean value analysis as described in the book. Since we have 4 devices in the system, we will need 4 different service time values (S_i) as input for the analysis. Additionally we also need the visit count (V_i) to each device i and the service rate $\mu(j)$ of our middleware, where j represents the number of jobs in the middleware device. For the MVA, we focus on the case with 1 and 2 middleware instances present in the system with 8 and 64 worker threads each. As we have a write and read workload per configuration, the MVA will be repeated a total of 8 times. It should also be noted, that all the measurements used throughout this section are taken from section 3.1 and 3.2 for the case with 64 total clients present in the system.

7.3.2.1 Visit Counts The following table presents how many time one job visits a device. The visit count values remains the same throughout each configuration:

Device	V_i
Client Network	2
Middleware	1
Server Network	2
Memcache Server	1

Table 20: Visit Counts.

7.3.2.2 Service Rate Since we model our middleware as a M/M/m device, we define the service rate $\mu(j)$, with j jobs inside the middleware as defined in the book. Let S_{MW} be the mean service time per request(defined below). We then define:

$$\mu(j) = \begin{cases} j/S_{MW} & j < m \\ m/S_{MW} & j \geq m \end{cases}$$

7.3.2.3 Service Time

- **Client Network**: We define the service time for both delay centers as follows: $S_{CN} = (R_C - R_{MW})/2$, where R_C is the response time measured by the client and R_{MW} the response time measured by the middleware. In order to better correlate the results of the MVA with the observed values at the client, we set S_{CN} for the one MW case equal to the S_{CN} of the two MW setting. The values for S_{CN} are depicted in table 21a.
- **Middleware**: We define the service time for the M/M/m model as follows: $S_{MW} = (R_{MW} - Q_{MW} - S_{Server})/m$, where Q_{MW} is the queue waiting time and S_{Server} is the

Workload	8 WT	64 WT
write	1.66	1.8
read	1.65	1.65

(a) S_{CN}

Workload	1-8WT	1-64WT	2-8WT	2-64WT
write	0.0048	0.0009	0.0018	0.0003
read	0.0054	0.0011	0.0019	0.0003

(b) S_{MW}

Workload	Service Time
write	0.0397
read	0.0876

(c) S_S

Workload	8 WT	64 WT
write	0.2039	0.2560
read	0.1869	0.4050

(d) S_{SN}

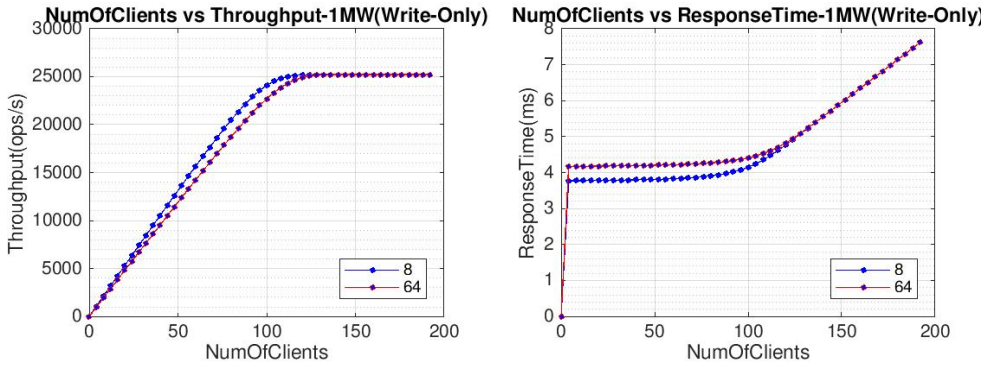
Table 21: Depiction of all service time values(ms) for each device. The same S_{CN} and S_{SN} values are used for both the 1 MW and 2MW setting.

memcached waiting time as measured by the middleware($T_{ServerTot}$). In detail, S_{MW} contains the time a requests spends inside the middleware, without accounting for the queue waiting time and the memcached waiting time divided by the number of workers. We have 8 different values for each configuration shown in table 21b.

- **Memcache Server**: The service time for the server(S_S) is computed by performing a M/M/1 analysis on the data from the baseline experiments in section 2.1. Using the throughput and response time measured by the client, we can directly deduce the service time using the formulas in the book. We used the results from the first baseline experiment for the write-only(24955 ops/s, 3.8473 ms) and read-only(11184 ops/s, 4.3 ms) workloads as input for the analysis. From the analysis we get a service time for read and write requests. The results are summarized in table 21c.
- **Server Network**: Finally the service time for both delay centers in the server network is computed as follows: $S_{SN} = (S_{Server} - S_S)/2$, where S_{Server} is the memcached waiting time as measured by the middleware and S_S the previously computed service time for the server. We again set S_{SN} for the 1 MW case equal to the values of the 2 MW case for better correlation. S_{SN} is depicted in table 21d.

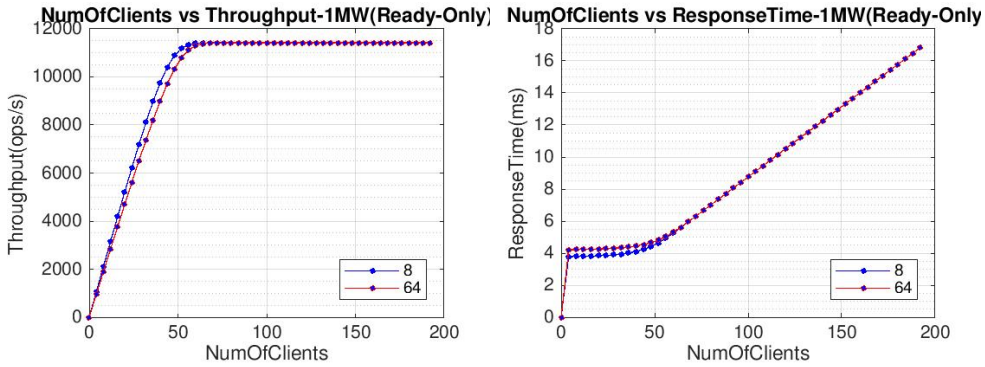
7.3.3 Results of extended Mean value analysis

The results of the extended MVA are presented in figure 17. For analyzing the measured data, we compare throughput and response time of the MVA with the throughput and response time measured at the client machine in section 3. These client measurements are depicted in figure 18.



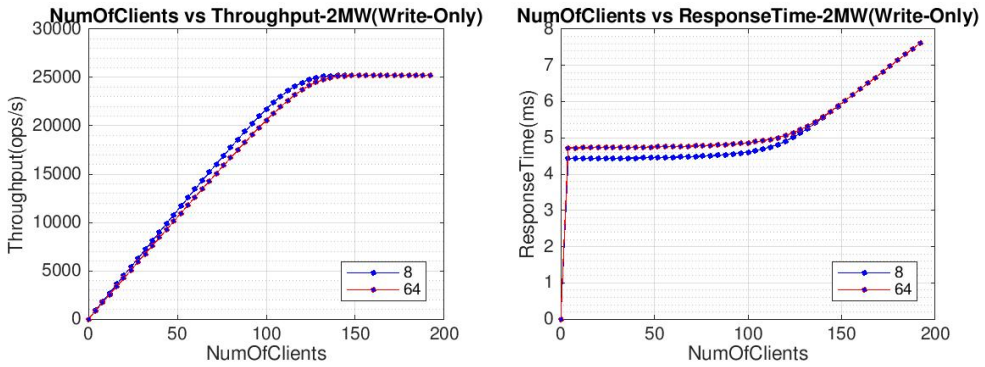
(a) 1MW Thrp. Write

(b) 1MW Resp. Write



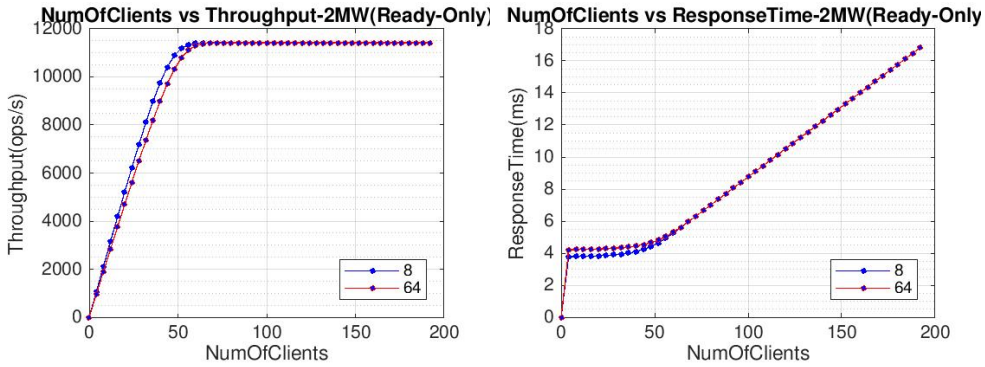
(c) 1MW Thrp. Read

(d) 1MW Resp. Read



(e) 2MW Thrp. Write

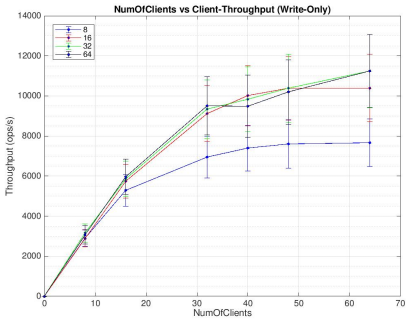
(f) 2MW Resp. Write



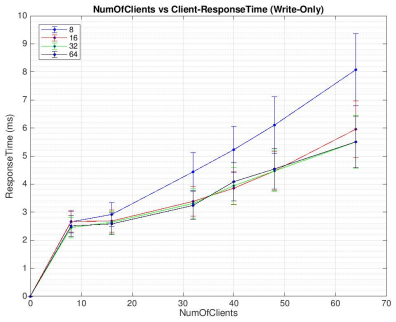
(g) 2MW Thrp. Read

(h) 2MW Resp. Read

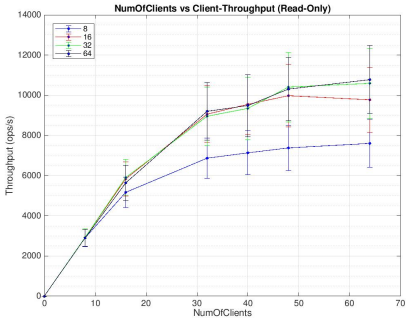
Figure 17: Results of the extended MVA.



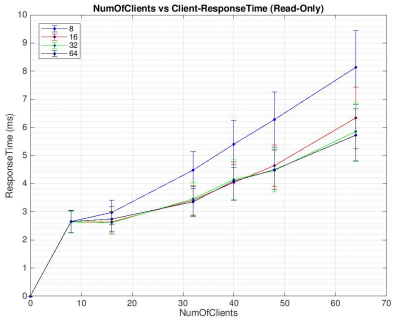
(a) 1MW Thrp. Write



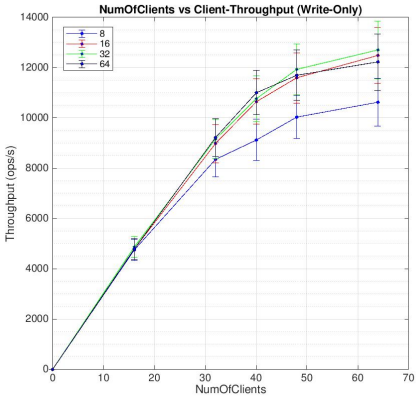
(b) 1MW Resp. Write



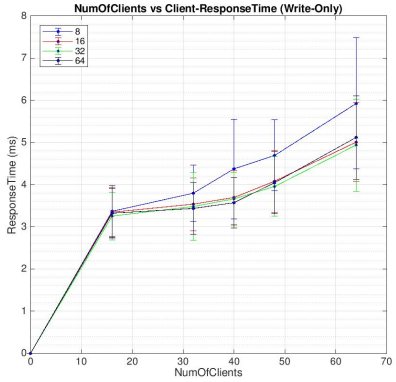
(c) 1MW Thrp. Read



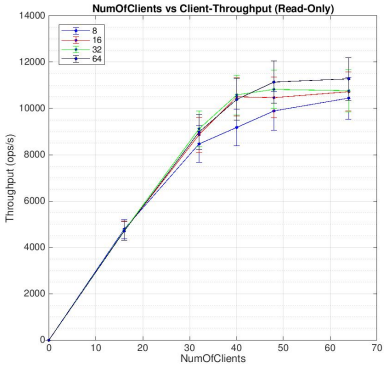
(d) 1MW Resp. Read



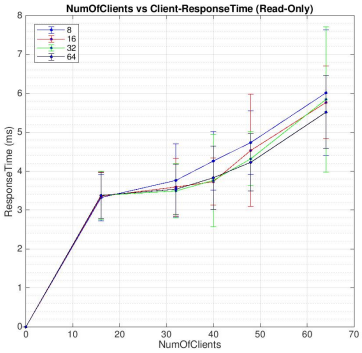
(e) 2MW Thrp. Write



(f) 2MW Resp. Write



(g) 2MW Thrp. Read



(h) 2MW Resp. Read

Figure 18: Results of section 3 measured at the client side

7.3.4 Bottleneck Analysis

The utilization for each configuration was the following as computed by the extended MVA for the case with 192 total clients in the system: 37.82 (Client Network), 19.5 (Server Network), 1 (Memcached Server) and 0.0035 (Middleware).

We observe that the utilization for the delay centers(client and server network) are larger than 1. From the book we infer that delay centers can have higher utilization than 1 without stability problems and therefore cannot be the bottleneck device of our system. When we disregard the delay centers, the memcached server device(M/M/1) has the next highest utilization. Consequently the server device is the bottleneck of the system. Contrary to our expectation, the utilization of our middleware is dwarfed by the server utilization. We would have expected the worker-processing time to have a much higher impact on the system. The reason becomes clear when inspecting how the service time was computed for the middleware: The service time doesn't contain the queue waiting time nor the memcached server waiting time, therefore the overall effect of the middleware is lower than expected. When neglecting the queue waiting time from the MW response time, it makes sense that the server processing time is the main limiting factor of the system, as seen in section 3.

7.3.5 Differences

$$R = \sum_{i=1}^{M} R_i V_i$$

$$X = \frac{N}{Z + R}$$

Figure 19: Small part of the extended MVA Algorithm as presented in the book.

The main difference between the measured data in figure 18 and the predicted data in figure 17 is, that the performance with 8 WT is almost exactly the same as the performance with 64 WT for both MW settings. Similarly, when comparing throughput and response time between the 1 MW and 2 MW case in figure 17, there is almost no difference between them. The reason for this behaviour are the following:

- Due to the definition of the service rate for each worker, we would have expected a difference in throughput between 8 and 64 WT. The reason why this is not the case lies in our choice for the service time values for each device and the definition of the extended MVA algorithm: Figure 19 shows a small part of the extended MVA, which computes the average response time of the system as the weighted sum of the response time of all devices. The throughput is then computed using the interactive law. As the client and server network have the highest utilization in the system, both devices have large service time values and therefore a large response time compared to the middleware device. Therefore the benefit of having more workers in the middleware is overshadowed by S_{CN} and S_{SN}. As a result the throughput with 8 WT is at certain points larger than the throughput with 64 WT since S_{CN} and S_{SN} are slightly smaller with 8 WTs.
- The service time at the middleware doesn't include the time a request waits inside the queue. Therefore the additional queue waiting time needed with lower number of worker-threads, is not properly reflected in our model. Consequently the performance with 8 WT is almost the same as with 64 WT.

Another difference is that for the write-only case the MVA throughput rises steadily up to 125-130 clients and then starts to flatten. This is not the case for the measured data at the client, as the throughput slope already starts to flatten sooner. The reason for this behaviour is similar to the one discussed in section 7.2.2. The M/M/m model assumes that the workers-threads can be run completely in parallel and have unlimited resources and don't have to share CPU time. Therefore the throughput increases without considering the real-world limitations of the system and the predicted saturation point occurs after the measured saturation point.

7.3.6 Similarities

As the results of the 1 MW and 2 MW are relatively similar, we will mainly discuss the similarities of the 2 MW case with 64 WT each to the client measurements in figure 18:

- When comparing the throughput values with 64 worker-threads in figures 17e and 18e, we observe that the predicted and measured data up to 64 clients lie close to each other. The predicted data is at 13434 ops/s for 64 clients, while the measured value lies at 12219 ops/s.
  The same behaviour can be seen when comparing the throughput for the read-only workload in figures 17g and 18g. The throughput starts to flatten in the predicted data and remains at 11407 ops/s after 64 clients. This flattening of the data can also be observed in the throughput measured by the client, where the data is at 11269 ops/s for 64 total clients.

- In figure 17f we observe that the average response time value with 64 WT is slightly larger than the measured values in figure 18f. This is also true for the read only case. This discrepancy can be accounted to the variances in the input data to the MVA. Especially for the service time values for each device, approximations have been made and due to this we loose some precision in the end result. For the read-only workload the increase in response time after 48 clients is well captured by the MVA output.

- Figures 17f and 17h we observe that the response time starts to rise at around 96 clients for the write-only and 48 clients for read-only case. This correlates well with the results from the baseline experiment, as saturation occurred after 96 resp. 48 clients.

7.3.7 Summary

In conclusion the network of queues model captures the mannerisms of our real world system better than the previous models. Furthermore the predicted throughput and response time values are very close to the measured ones. The main issue of this model is that the performance with low number worker-threads is almost exactly the same as with 64 WT, as it doesn't take into consideration the additional queuing time with only 8 WT. Furthermore since the service time of client and server network devices are relatively large compared to the middleware device, the benefit of having more workers is overshadowed by the large response time values of the delay center devices.

8 References

Below we indicated the link to all the java classes referenced in section 1.

1 https://gitlab.ethz.ch/cribin/asl-fall17-project/blob/master/src/MyMiddleware.java

2 https://gitlab.ethz.ch/cribin/asl-fall17-project/blob/master/src/ClientConnectionHandler.java

3 https://gitlab.ethz.ch/cribin/asl-fall17-project/blob/master/src/ClientHandler.java

4 https://gitlab.ethz.ch/cribin/asl-fall17-project/blob/master/src/RequestWorker.java

5 https://gitlab.ethz.ch/cribin/asl-fall17-project/blob/master/src/message/MemcachedRequest.java

6 https://gitlab.ethz.ch/cribin/asl-fall17-project/blob/master/src/message/MemcachedResponse.java

7 https://gitlab.ethz.ch/cribin/asl-fall17-project/blob/master/src/requestHandlers/SetRequestHandler.java

8 https://gitlab.ethz.ch/cribin/asl-fall17-project/blob/master/src/requestHandlers/GetRequestHandler.java

9 https://gitlab.ethz.ch/cribin/asl-fall17-project/blob/master/src/requestHandlers/MultiGet--RequestHandler.java

10 https://gitlab.ethz.ch/cribin/asl-fall17-project/blob/master/src/message/MemcachedMessageParser.java

11 https://gitlab.ethz.ch/cribin/asl-fall17-project/blob/master/src/utils/Hasher.java