Advanced Systems Lab Report

Autumn Semester 2018

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

Grading

Section	Points
1	
2	
3	
4	
5	
6	
7	
Total	

1 System Overview

Aim of the middleware is to collect and forward queries from clients to servers, wait for the responses and reply the clients back — all under specified conditions and configurations. Aforementioned servers and clients are the instances of Memcached [1] and of memtier_benchmark [2], respectively. The design of the middleware spans several classes and functions that implement different functionalities. The classes operate in a multithreaded fashion so the different functionalities can work together asynchronously.

1.1 System Design

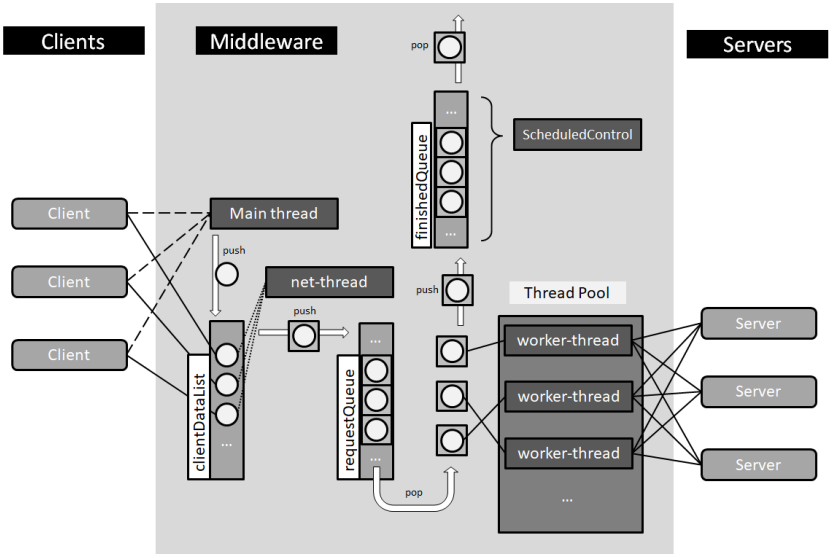


Figure 1: A simplified diagram of the system showing the inner structure of the system and the interaction of its components in an informal manner. Circles represent the clientData objects and the squares represent the requestData objects, whose instances encapsulate the ones of the former. Dark gray rectangles represent the threads in the system, of which there are always three plus the size of the worker thread pool. Light gray rectangles represent the shared lists, of which there are three as shown. Light gray rounded rectangles illustrate the external systems, namely the clients and servers. Dashed line shows the communication with the welcomingSocket and the dotted line is net-thread checking if there is any request available. Solid lines represent the main information flow from clients to servers and again back to the clients.

1.1.1 Main thread

Main thread of the MyMiddleware class is responsible for setting up the working environment for the whole system. It first initializes the shared data structures that are to be used by other threads; namely, these are clientDataList (which stores information about each one of the connected clients), requestQueue (which is a first-in, first-out queue for queries awaiting execution) and finishedQueue (which temporarily stores the completed requests for the aggregation of their statistics). It then creates a clientHandler (so-called "net-thread", which is responsible for handling communication with clients) and the specified number of serverHandler threads (so-called "worker-threads", which are responsible with for handling communication with the servers). Throughout this report, names clientHandler and net-thread will be used interchangeably. Same also applies for worker-threads and serverHandlers. Lastly, the main thread creates

the welcomingSocket and listens for incoming client connections in a loop. As soon as such a connection request is received, main thread creates the respective clientData object and pushes it into the clientDataList.

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

1.1.2 clientHandler (net-thread)

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

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

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

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

1.1.3 serverHandlers (worker-threads)

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

If it is a set query, the request is sent to all of the servers and reported as successful to the client only if all the responses indicate so. If not, one of the error messages are relayed to the sender client. If it is a get request there are two possibilities. If it is a multi-get request

in a multi-server setting with `readSharded` option provided as true, then it is sharded into smaller requests. If the query requests a single object, there is a single server or `readSharded` is provided as false, then the request will be sent to only one server. In either case, middleware employs a simple load balancing scheme through a round-robin scheduling. It iterates over all servers while sending get requests or the sharded get requests.

1.1.4 ScheduledControl

ScheduledControl is started by the net-thread as it receives its first request. It periodically wakes up, aggregates the new information, then sleeps again. It goes over the entries in finishedQueue in its each run, collecting type-specific statistics. While going over the list of requests the ScheduledControl assigns each of the requests to a histogram bin with respect to its response time. Additionally, saves the length of the requestQueue in each run.

1.1.5 Internal parameters

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

- `verboseLogs` specifies if the system should output the work logs or not. Such logs are printed when a request is received, relayed to servers, got answered and replied back to the clients. It also prints out details about which thread handled the request, which client sent the request, to which server the request was sent to and the completely parsed messages and replies. It is a useful option for understanding the inner workings and state transitions of the middleware system. It should be set as false if user is not debugging.
- `verboseAggr` controls the outputting of the aggregated statistics. It is always true for our experiments as it is our main tool for analysing the system.
- `timeoutSecs` specifies the amoun of time the system should wait until new queries before shutting itself down. Altough the middleware can work seamlessly without any interruption, systems needs to be restarted to operate in its most efficient manner in its current implementation. 3 seconds are proved to be enough for our experiments.
- `initDelaySecs` and `periodSecs` respectively specify the initial delay and the period of the ScheduledControl runs. Both are given as 1, that means the ScheduledControl thread begins collecting data after an initial delay of 1 seconds and it wakes itself at the end of every 1 second period after that.
- `sep` sets the column separator to be used by ScheduledControl when outputting aggregated statistics. We output the aggregation table inside the output file in a comma separated format, so we set sep as `','`.

1.2 Methodology

All the necessary commands for initializing the experimental setup and conducting the experiments are provided in a companion file called `commands.sh`. It also exemplifies the usage of auxiliary technologies used for further analysing the system and its environment as well as it contains some other helpful commands for copying and plotting the experiment results. `plot` folder contains all scripts (and their eventual outputs) necessary for plotting the results placed under the `res` folder. Every plot provided inside or alongside this report is reproducible with the provided data and the scripts. Lastly, `runner.sh` is the encapsulating script that runs the desired agent (i.e. client, middleware, server or dstat [3] tool) for the configuration provided

Probably should detail which scripts correspond to which experi-

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

Table 1: Abbreviations for experiment parameters.

through the command line arguments. The arguments it expects share the same nomenclature we use in scripts to refer the experiment parameters. Here, in Table 1, we list these abbreviations. Result directory hierarchy also obeys the same nomenclature.

It is important to pay attention the order and timing of executing commands. Servers should be run first and the middlewares (if any) should start waiting connections. Then, both the clients and dstat processes should start simultaneously — we could do this by using Cluster SSH (also called as cssh) [4]. Clients, middlewares and dstat processes ran accordingly for three separate repetitions, afterwards they should be restarted by hand. Another important thing to keep in mind while using the project codebase is to remember that the codes do not cover erroneous cases. For example, middleware does not support any other operations than set or get. Similarly, plotting scripts may not work with data with erroneous content, filenames, directory hierarchy and such.

Three repetitions were seem to be sufficient all the experiments as both our middleware and the external systems it communicated had shown performances stable under repetitions. Similarly, trimming five seconds of warm-up and cool-down periods, respectively at the beginning and the end of experiments, was found as sufficient through the initial exploratory experiments performed. Clients can lag behind around at most one second per a three repetition run, but it is still negligible with the times cut for warm-up and cool-down periods.

iPerf3 [5] is used to measure the maximum achievable bandwith in between different machines in our environment. For each machine in every experiment, we collect data through dstat. dstat is run in its default mode, so it collects statistics cpu, disk, network, paging and system statistics. It collects data every second after an initial delay of one second — similar to the ScheduledControl in our middleware.

Error bars in plots show a range of one standard variance, unless indicated otherwise. Most of the plots have their x-axis as the number of clients. It is the effective number of different clients and is simply calculated by multiplying four numbers: number of virtual clients per thread, number of threads per memtier instance, number of memtier instances per virtual machine and the number of client machines. Using the abbreviations in Talbe 1 the formula could be written as $numClients = vcli \times tcli \times icli \times ncli$. The term latency is used interchangeably with the term response time. Plots specify the source of measurements as middlewares or clients in their subtitles, so it is easy to find for a plot whose perspective it reflects.

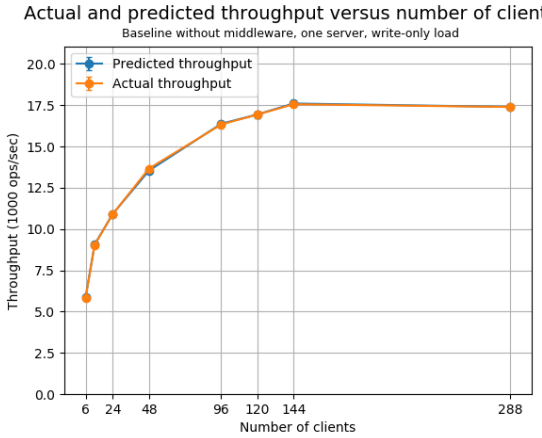
2 Baseline without Middleware

In this section, we examine the characteristics of the clients and servers before the introduction of the middleware to the environment. For all different configurations of different experiments we show that the interactive law holds true by calculating the throughput values from the response

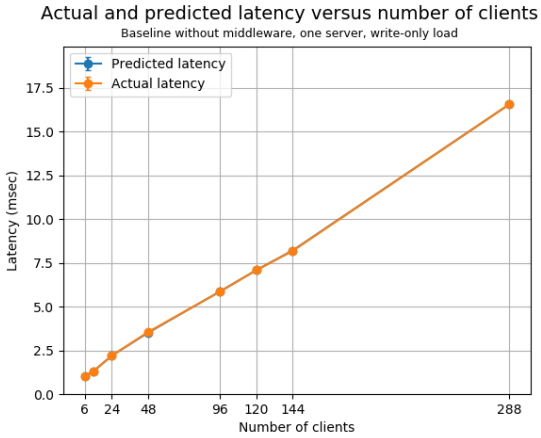
times and vice versa. Values predicted through the interactive law is always illustrated alongside the actual values outputted by the memtier client instances.

2.1 One Server

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

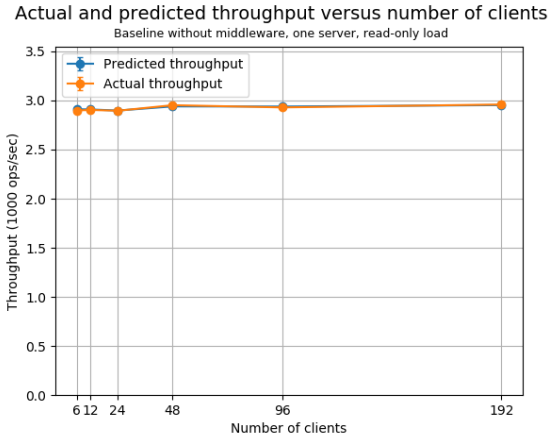


(a) Write-only throughput

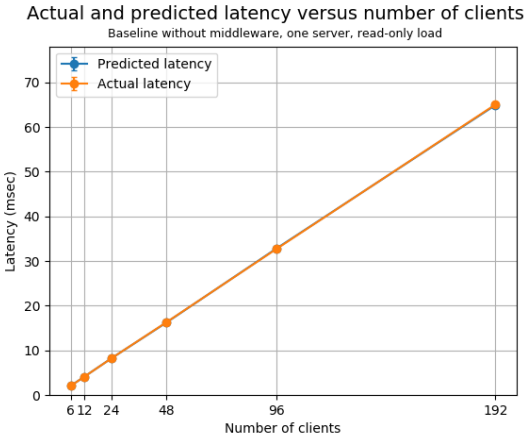


(b) Write-only latency

Figure 2: Throughput and latency values for the write-only workload for baseline without middleware, for one server case.



(a) Read-only throughput



(b) Read-only latency

Figure 3: Throughput and latency values for the read-only workload for baseline without middleware experiment with one server case.

2.1.1 Explanation

We see the results for the one server configuration in Figures 2 and 3. Plots show that the interactive law holds true for both experiments.

For the write-only part, we can observe the systems starts as undersaturated and saturates around 144 clients by looking at the throughput plot. Throughput increases until the saturation point, as additional clients are connected to the system. Afterwards we cannot observe changes in the throughput and state the system is saturated. We see a similar trend in the response time plot as it is slightly curved until around teh saturation point but linear afterwards. After the saturation point, the systems cannot handle any more requests and the average response time would increase linearly with respect to number of clients while there is not any change in the throughput as the system is at full capacity.

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

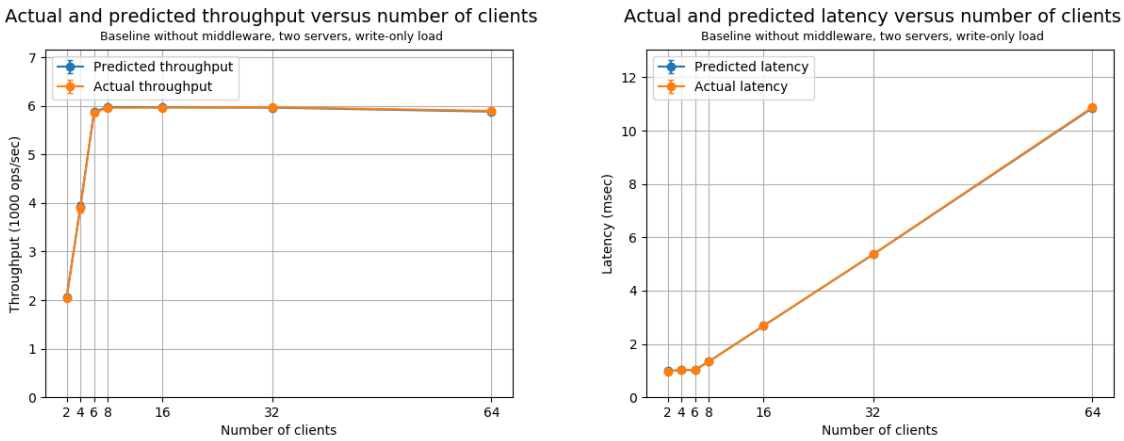
In the read-only part, we see that the system begins operating in the saturated region for the minimum number of clients we can achieve, which is 6. We conclude this as we cannot observe any change in throughput when we vary the number of clients and the response time is approximately a linear function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

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

is usage limited by correct, both here and below?

2.2 Two Servers

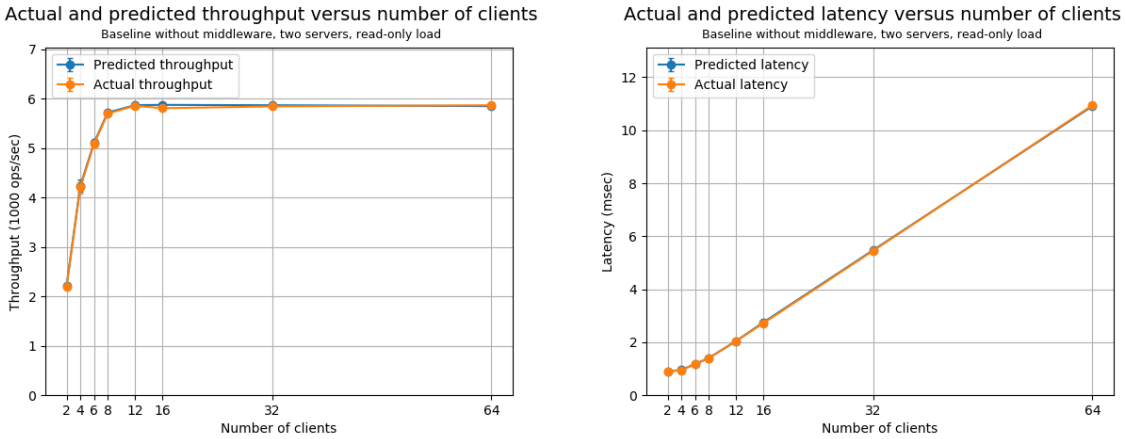
This configuration deals with one client machine sending requests to two servers. This time the client machine has two instances of memtier running (each connected to a different server) with one client thread per instance. We again change the workload (write-only and read-only) and change the total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



(a) Write-only throughput

(b) Write-only latency

Figure 4: Throughput and latency values for the write-only workload for baseline without middleware experiment with two servers.



(a) Read-only throughput

(b) Read-only latency

Figure 5: Throughput and latency values for the read-only workload for baseline without middleware experiment with two servers.

2.2.1 Explanation

Results are illustrated in Figures 4 and 5. Unlike the first part, we obtain similar plots for both types of workloads, namely the read-only and the write-only case. We say that the system saturates when there is 8 clients in the system for the read-only case as it is elbow point for the throughput plots and the response time plots continue increasing linearly after crossing 8 clients. It is the same response in the write-only case when number of clients equals 12, although it could be argued that the actual saturation point is somewhere slightly earlier in the plot, e.g., 9 or 10 clients. We can clearly say that the system is under saturated before those points as the throughput rises quickly we only see marginal changes in the response times in both workloads.

why 12 and 8, why not equal?

The reason behind the saturation for both cases is th same: the network bandwith limitations. Situtation is quite similar to the earlier results in Section 2.1. For the write-only case, dstat outputs for the saturated configurations show that the client sends at most 24MB per second to servers, which is its limit confirmed by iperf statistics. This corresponds to a total of $\sim$ 6000 requests distributed to two servers, corroborating our results. We also know through iperf that a server can send at most 12MB per second and and we see the same numbers in our dstat files for the read-only part. This equals to 24MB per second received by clients, again amounting to a total of $\sim$ 6000 requests, this time as a result of another limiting factor.

2.3 Summary

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

Table 2 reports the maximum throughput obtained in both experiments and the configurations that allow such maxima. For both experiments under either read-only or write-only workload conditions, we have concluded that the bottleneck of the system is always the network bandwidth. Intuitively, the throughput values for the experiments with write-only workload are limited by the client to server bandwidth and similarly, throughput values for the read-only counterparts are limited by the server to client bandwidth. These arguments apparently reflect on our results in Table 2.

Maximum throughput of different VMs.

	Read-only workload	Write-only workload	Configuration gives max. throughput
One memcached server	2896.4 ± 21.6	17592.5 ± 25.4	numClients = 144 for both
One load generating VM	5855.5 ± 15.6	5959.9 ± 3.2	numClients = 12 and 8, resp.

Table 2: Summary of baseline without middleware experiments.

Read-only throughput doubles as we introduce another server to our one server setting. Similarly for the write-only workload, by going from three load generating machines to a one client setting we can see expect the throughput to become one third of what it was before.

As a side note, we observe only the write-only workload with three client machines resulting a high percentage of CPU usage in servers, in the saturated region. The idle time of the server for this setting is under 20% most of the time. Aside from this specific case not one server or client experience any high CPU utilization, idle times are consistently above 90%.

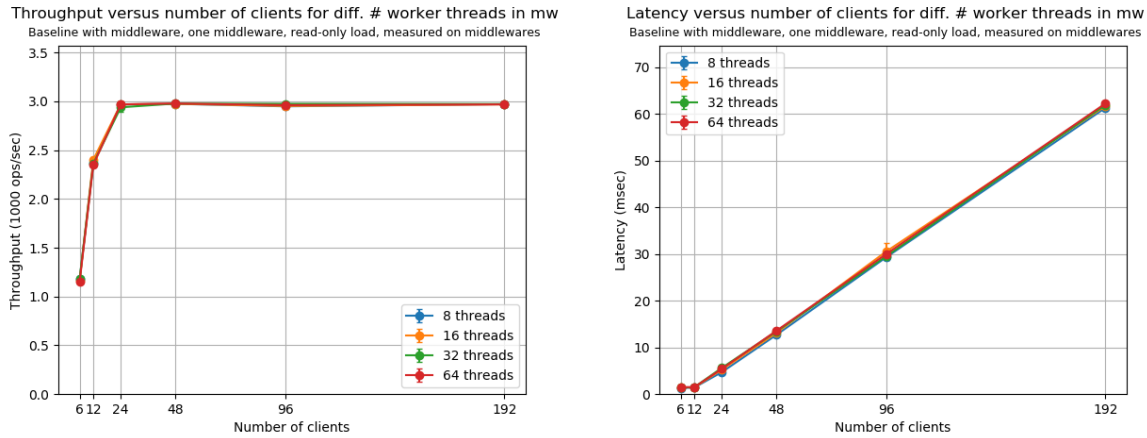
3 Baseline with Middleware

In this section, we observe the changes in the overall system when the middleware is introduced to the environment, as well as we discuss the characteristics of the middleware that we can observe in the scope of this set of experiments.

Comment about interactive law.

3.1 One Middleware

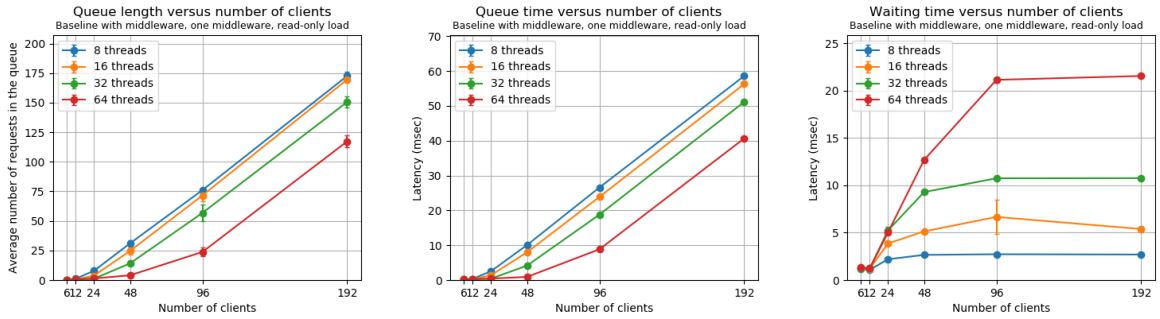
First experiment for this section is conducted on three client machines connected to one middleware, which in turn connected to one server. Apart from experimenting with two workloads (read-only and write-only) and varying the number of virtual clients we also change the number of worker threads in the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



(a) Read-only throughput

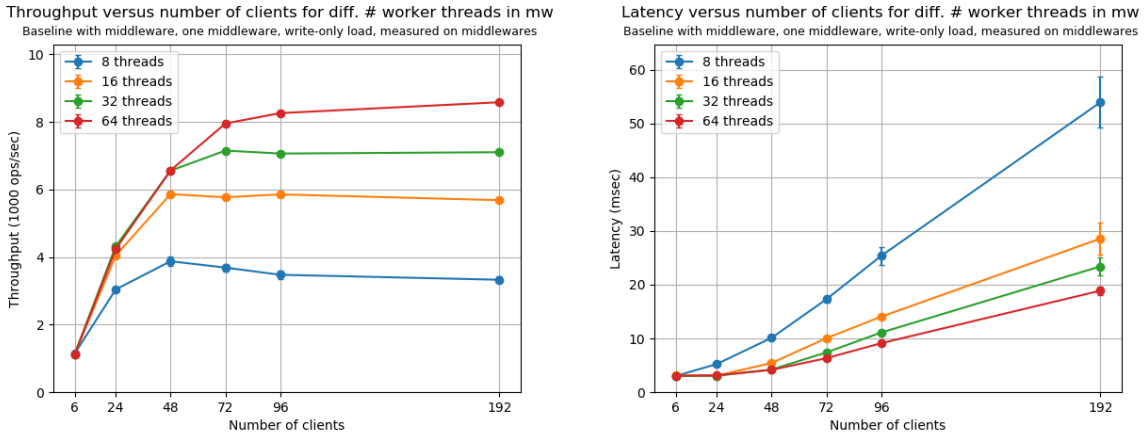
(b) Read-only latency

Figure 6: Throughput and latency values for the read-only workload for baseline with middleware experiment with one middleware.



(a) Avg. queue length (b) Avg. queue time (c) Avg. waiting time

Figure 7: Average queue lengths, queue times and waiting times for the read-only workload for baseline with middleware experiment with one middleware.



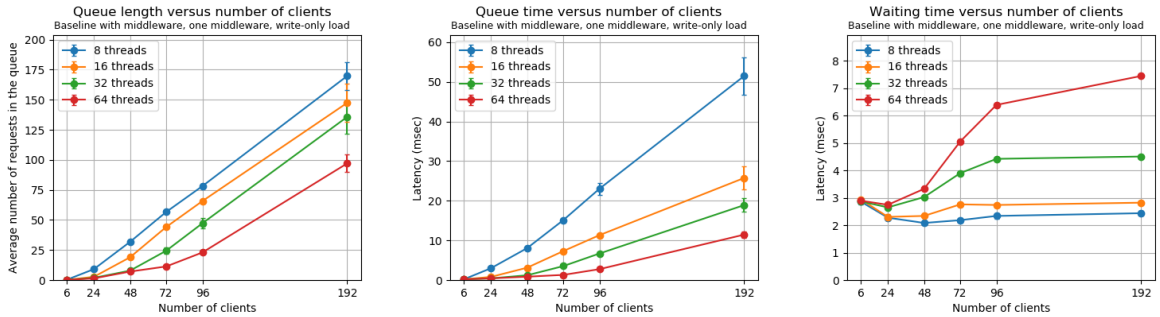
(a) Write-only throughput (b) Write-only latency

Figure 8: Throughput and latency values for the write-only workload for baseline with middleware experiment with one middleware.

3.1.1 Explanation

Plots in Figures 6 and 8 illustrate the throughput and response time measured on the middleware, respectively for read-only and write-only loads. With the read-only load, we observe a trend quite similar to the read-only results in baselines without middleware. Until we reach around 24 clients we can observe quite an increase in the throughput which is only accompanied by relatively smaller increases in the response time, regardless of the size of the thread pool in middleware. Afterwards, the trend is again the same for different number of threads: an increase in the number of clients results in a linear increase in the response times and does not change the average throughput obtained. Therefore, we state that the system saturates when there are around 24 clients in the system.

The reason behind this saturation is familiar to us from the previous sections: server to middleware connection is limited by a bandwidth of 12MB per second, so the server cannot send more replies because of the network limitations. This saturation happens so early in the system that we do not get to observe the effects of different threads in middleware. In other words, the middleware works efficiently even with 8 worker threads until the system saturates, so any number of worker threads could hypothetically handle more work if the server could be able respond more clients. It should also be noted that we cannot observe any network-wise



(a) Avg. queue length (b) Avg. queue time (c) Avg. waiting time

Figure 9: Average queue lengths, queue times and waiting times for the write-only workload for baseline with middleware experiment with one middleware.

limitations on top of the server-side send bottleneck and the CPU usage is fairly low for all the machines in the system (80% idle time at worst), as per dstat and iperf.

marginal change for tmw's

Figure 7a illustrates the average queue lengths for various configurations. As the system is network bound for this workload, the server can only reply a portion of the requests while the other thread wait to send their requests. We expect these number of waiting threads to be very small, if the number of clients are smaller than the number of middleware threads. For the range the number of clients are greater than the worker of threads every additional client should increase the average queue length linearly. The plot confirms both our intuitions as for every different number of worker threads the queue length stay close to zero until the number of clients exceed the number of worker threads and the difference in queue lengths for the linear region is proportional to the number of clients. Lastly, we can see by comparing the values in Figures 7a and 7b that the queue times and the queue lengths have an approximately linear correlation as expected. The reason behind such relation is that as the system is network bound it is not the middleware but the server side that affects the queue times in the saturated region and since the server has only one thread we expect the queue times increasing linearly with the increasing queue length.

In the write-only case, we can observe a trend completely different from the read-only case, as the network is no longer our bottleneck. The trend until reaching 24 clients is similar with the read-only case as differences in the number of worker threads do not correspond to a difference in terms of the performance of the system, except the relatively smaller performance gain for the 8 worker threads. As we introduce more and more clients to the system we observe more divergence with respect to performance for different number of threads in the middleware. We can see that there are saturation points with different values in both axes for different worker thread configurations. We can also see that we cannot achieve the levels of throughput reached in the first part of the baseline without middleware, as the middleware introduces a considerable amount of overhead between the clients and the server.

This time, the comparison of queue lengths and queue times (Figures 9a and 9b, respectively) yield a different interpretation. We can clearly see that, for the same queue lengths, different number of worker threads result in different scaling factors for the queue times. Moreover, queue length to queue time ratio increases with the number of worker threads. What that means is that the systems could process a longer queue for the same queueing times in average, since there will be more worker threads available to handle the requests.

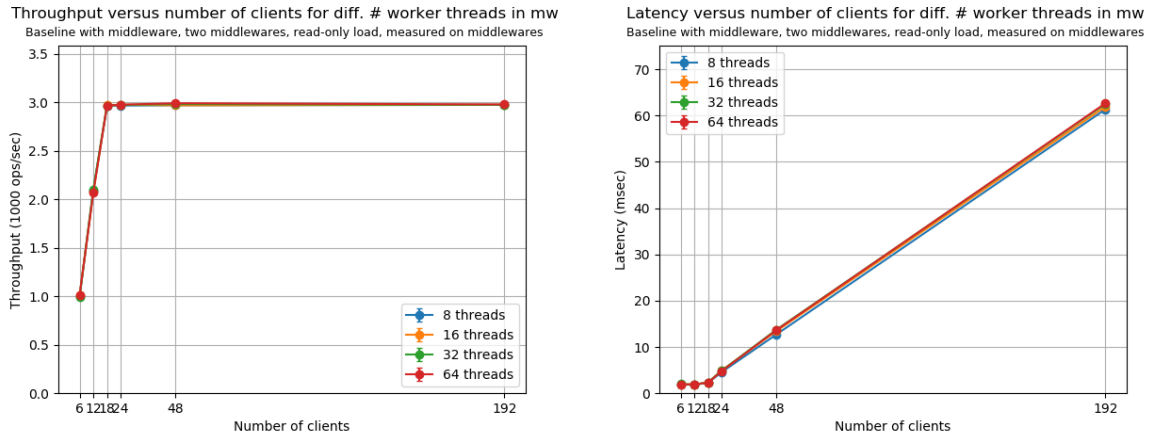
After interpreting all the relevant plots we can conclude that for the write-only workload the middleware is the bottleneck. dstat measurements are also cross-checked and we do not report

any network or CPU usage limit for the write-only configurations. We say that the number of worker threads is the limiting parameter for our experiments for the write-only load and more worker threads mean better system performances. However, we cannot assert the system performance would continuously increase as we introduce more and more worker threads into the middleware. Looking at the Figures 7c and 9c we can see that waiting times and the number of working thread in the middleware are inversely proportional and we can expect the waiting times to increase more as we keep increasing the worker threads. For our experimental setup this is an expected result as we configure our servers to only use one thread.

BE WARY, rethink that, compare against csb values

3.2 Two Middlewares

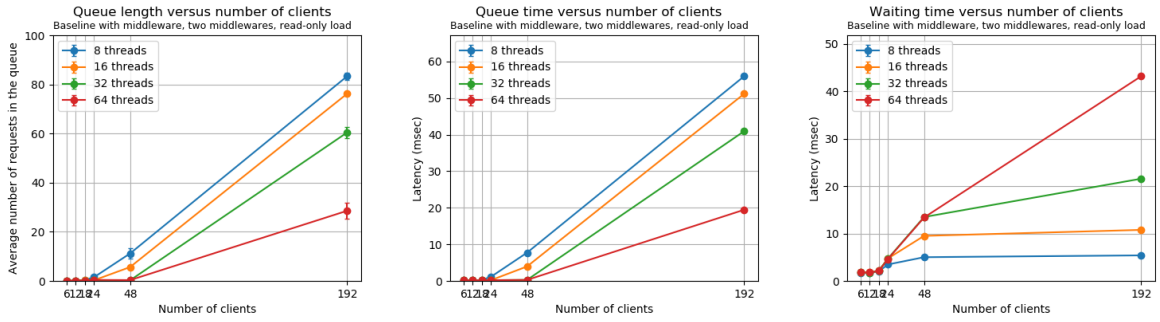
Second experiment for this section again has an environment with three clients and one server, but two middlewares instead of one. Again, we vary the workloads, number of clients and number of worker threads in the middleware and discuss the results.



(a) Read-only throughput

(b) Read-only latency

Figure 10: Throughput and latency values for the read-only workload for baseline with middleware experiment with two middlewares.

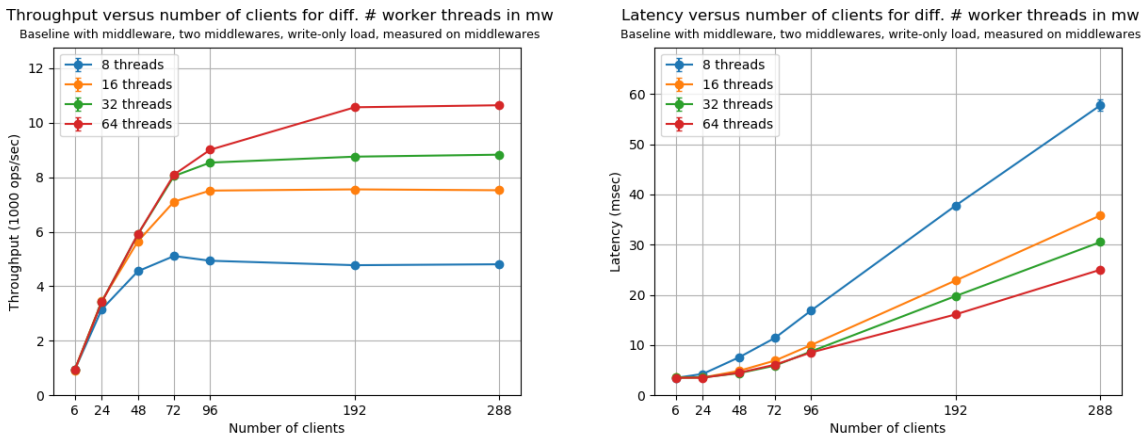


(a) Avg. queue length

(b) Avg. queue time

(c) Avg. waiting time

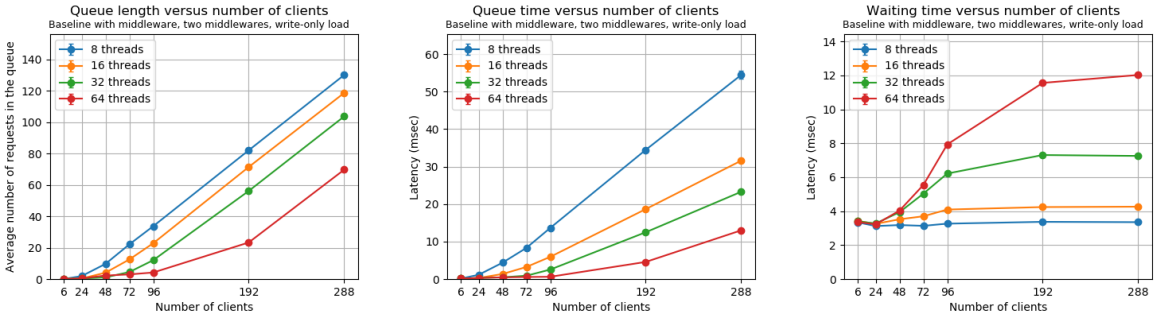
Figure 11: Average queue lengths, queue times and waiting times for the read-only workload for baseline with middleware experiment with two middlewares.



(a) Write-only throughput

(b) Write-only latency

Figure 12: Throughput and latency values for the write-only workload for baseline with middleware experiment with two middlewares.



(a) Avg. queue length

(b) Avg. queue time

(c) Avg. waiting time

Figure 13: Average queue lengths, queue times and waiting times for the write-only workload for baseline with middleware experiment with two middlewares.

3.2.1 Explanation

Results for the read-only workload is almost exactly the same as those from the setting with one middleware, as the bottleneck of the system is stays the same, which was detailed in the previous sections. Both the throughput and latency values (Figure 10) with respect to number of clients is quite similar to those of the configuration with one server. Figure 11 illustrates the average queue length, queue time and waiting time statistics for this experiment on read-only load. An intuitive correlation is as follows: statistics for the 8, 16 and 32 thread configurations for two middleware setup matches respectively with the 16, 32 and 64 thread settings for the one middleware. In other words, doubling the number of threads for one midleware has the same effects as introducing another middleware with the same number of threads on the system. Although, we can only say that for our experimental setup and configurations and this equivalence may not hold for factors much greater than two. Lastly, we warn the reader to be careful when comparing the queue lengths for two and one middleware setups directly as the reported queue length is averaged over the middlewares. That is, to approximate the total number of requests waiting in the middleware queues one should multiply the averaged value by the number of middlewares in the environment. Lastly, we note the relation between queue lengths and queue times is the same as in the one middleware setup.

waiting times

Plots in Figures 12 and 13 illustrate various statistics for the write-only load. We see results similar to the one middleware setup. System saturates at different number of clients for different number of worker threads. For the same average queue lengths, the queue times shorten as the number of worker threads increases. Again the number of working threads is our limiting factor for the system performance. Thus, we conclude the middleware being the bottleneck for the write-only case with two middlewares. Again, we find it intuitive that the statistics for the one middleware setup matches those of the two middleware setup with the half amount of worker threads, as it was the case with the read-only workload.

3.3 Summary

write a summary, explain the miss rates

Maximum throughput for one middleware.

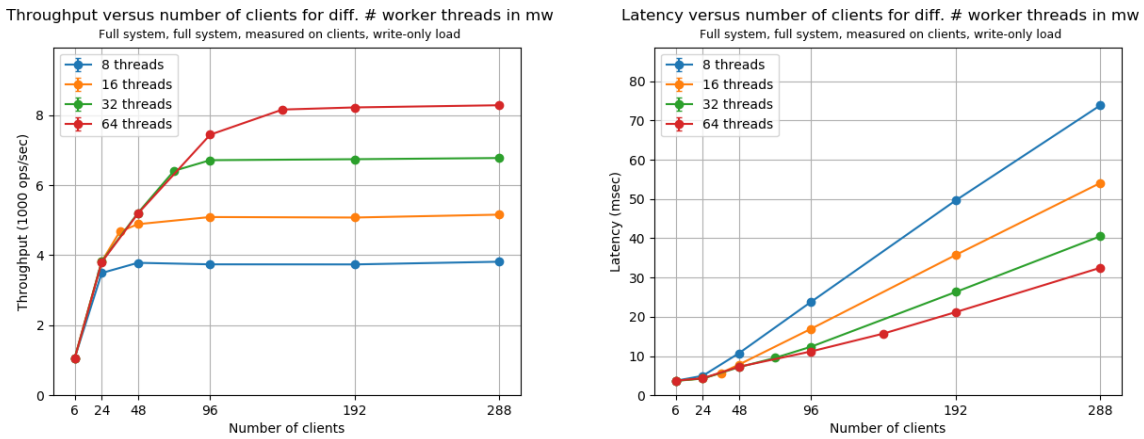
	Throughput	Response time	Avg. time in queue	Miss rate
Reads: Meas. on mw	2969.3 ± 1.5	5.413 ± 0.021	0.406 ± 0.011	0.0095404
Reads: Meas. on clients	2969.3 ± 1.8	8.083 ± 0.004	n/a	0.0095287
Writes: Meas. on mw	8264.1 ± 17.5	9.125 ± 0.017	2.740 ± 0.015	n/a
Writes: Meas. on clients	8263.5 ± 17.3	11.628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

Table 3: Summary of the baseline with one middleware.

Maximum throughput for two middlewares.

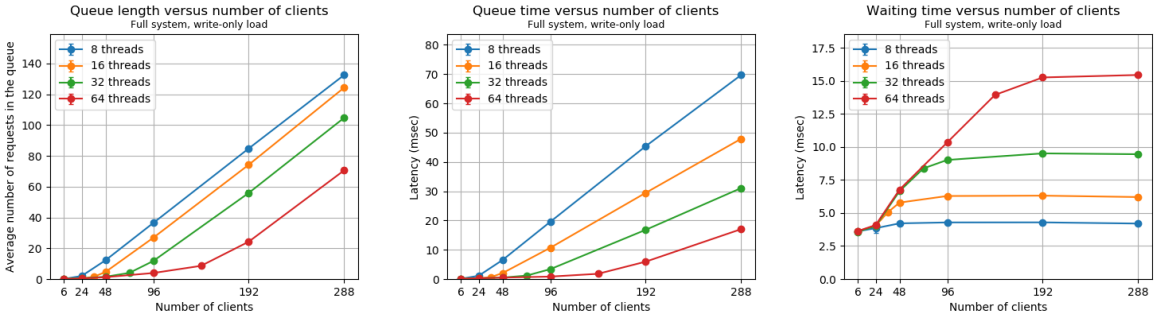
	Throughput	Response time	Avg. time in queue	Miss rate
Reads: Meas. on mw	2964.3 ± 0.3	2.288 ± 0.061	0.181 ± 0.001	0.0093647
Reads: Meas. on clients	2964.4 ± 0.8	6.077 ± 0.002	n/a	0.0095260
Writes: Meas. on mw	10566.1 ± 59.6	16.121 ± 0.118	4.557 ± 0.039	n/a
Writes: Meas. on clients	10566.4 ± 59.6	18.206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

Table 4: Summary of the baseline with two middlewares.



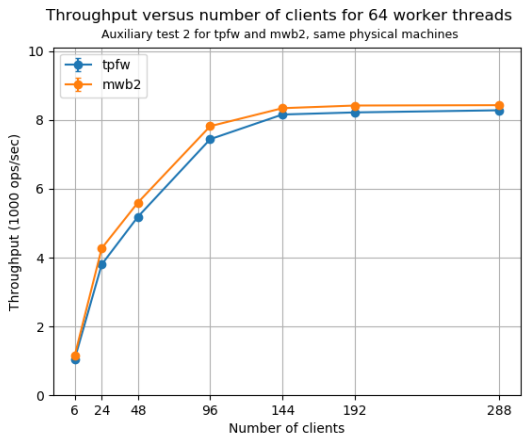
(a) Throughput (b) Latency

Figure 14: Throughput and latency values for the throughput for writes experiment, write-only workload.

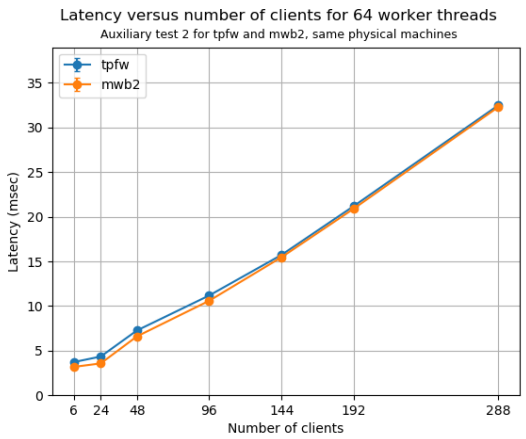


(a) Avg. queue length (b) Avg. queue time (c) Avg. waiting time

Figure 15: Average queue lengths, queue times and waiting times for the throughput for writes experiment, write-only workload.



(a) Throughput



(b) Latency

Figure 16: Throughput and latency values for the auxiliary experiment, write-only workload with 64 worker threads on middlewares. tpfw stands for the throughput for writes experiment, whereas the mwb2 denotes the middleware baseline experiment with two baselines.

Maximum throughput for the full system

	WT=8	WT=16	WT=32	WT=64
Throughput (Middleware)	3492.9 ± 49.1	4673.6 ± 23.5	6411.0 ± 5.0	161.1 ± 24.9
Throughput (Derived from MW response time)	3459.6 ± 205.4	4698.3 ± 40.1	6246.1 ± 5.8	8124.1 ± 27.3
Throughput (Client)	3492.8 ± 49.1	4674.3 ± 23.4	6412.2 ± 6.2	8161.5 ± 25.2
Average time in queue	1.112 ± 0.062	0.622 ± 0.006	1.148 ± 0.010	1.779 ± 0.017
Average length of queue	2.0 ± 0.0	1.4 ± 0.1	4.0 ± 0.2	8.6 ± 1.2
Average time waiting for memcached	3.9 ± 0.4	5.0 ± 0.1	8.4 ± 0.0	13.9 ± 0.0

Table 5: Summary of the throughput for writes experiments.

4 Throughput for Writes

4.1 Full System

4.1.1 Explanation

4.2 Summary

5 Gets and Multi-gets

5.1 Sharded Case

5.1.1 Explanation

5.2 Non-sharded Case

5.2.1 Explanation

5.3 Histogram

5.4 Summary

6 2K Analysis

7 Queuing Model

7.1 M/M/1

7.2 M/M/m

7.3 Network of Queues

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

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

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

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