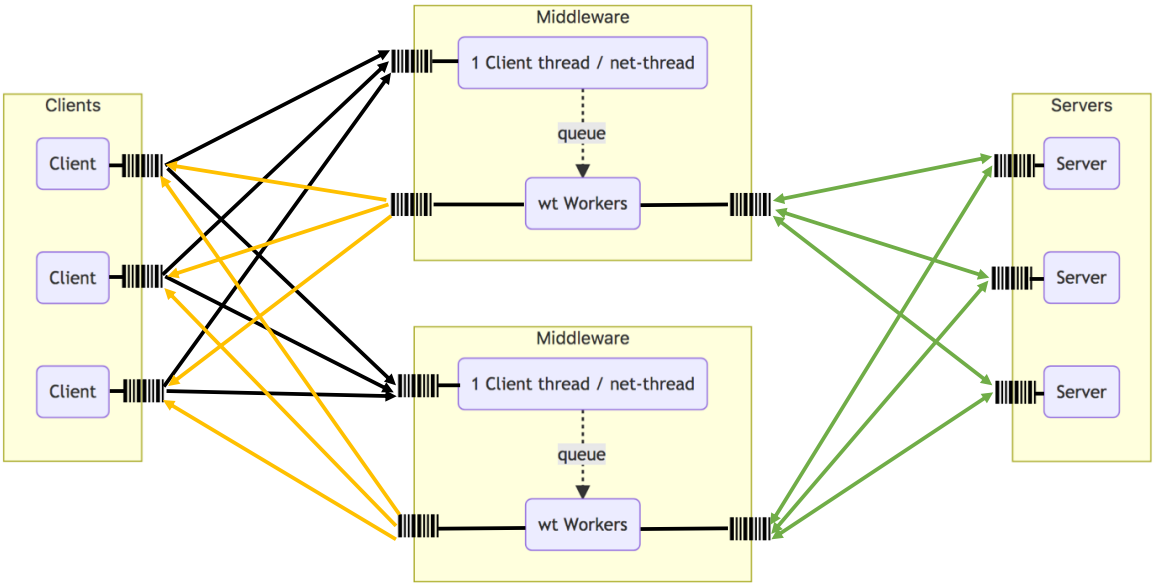
Advanced Systems Lab Report

Autumn Semester 2018

Name: Pirmin Schmid



1 System Overview (75 pts)

In this ASL project a distributed system is studied (Figure 1). It consists of key-value stores (memcached servers), load generating clients (memtier_benchmark), and a self-written Java middleware following the specification in the project description. The system is evaluated when deployed on virtual machines (VMs) in the Microsoft Azure cloud running Ubuntu 16.04 LTS.

The defined machines have different properties (Table 1), mainly with respect to computing power (vCPU count) and network bandwidth limitations for sending network traffic, which is crucial as several of the experiments have revealed. During all experiments, disk I/O does not matter because all processing runs in RAM; logging has been designed to make only negligible use of disk I/O (see results). RAM of the VMs has been sufficient for all experiments and all instrumentation activity to have no paging and no process kills by the kernel.

All experiments are run as *closed system*, i.e. the number of clients is fixed, and clients do not send new requests before they have received the reply to the former request. Thus, the system is self-regulating and cannot "explode" such as an *open system*.

Key design decisions and implementation information relevant to understand the data of the experiments are explained here. Additionally, a very detailed documentation (including e.g. exact definition of each used variable) is provided in the document `DESIGN_AND_TECHNICAL_NOTES.md`[1].

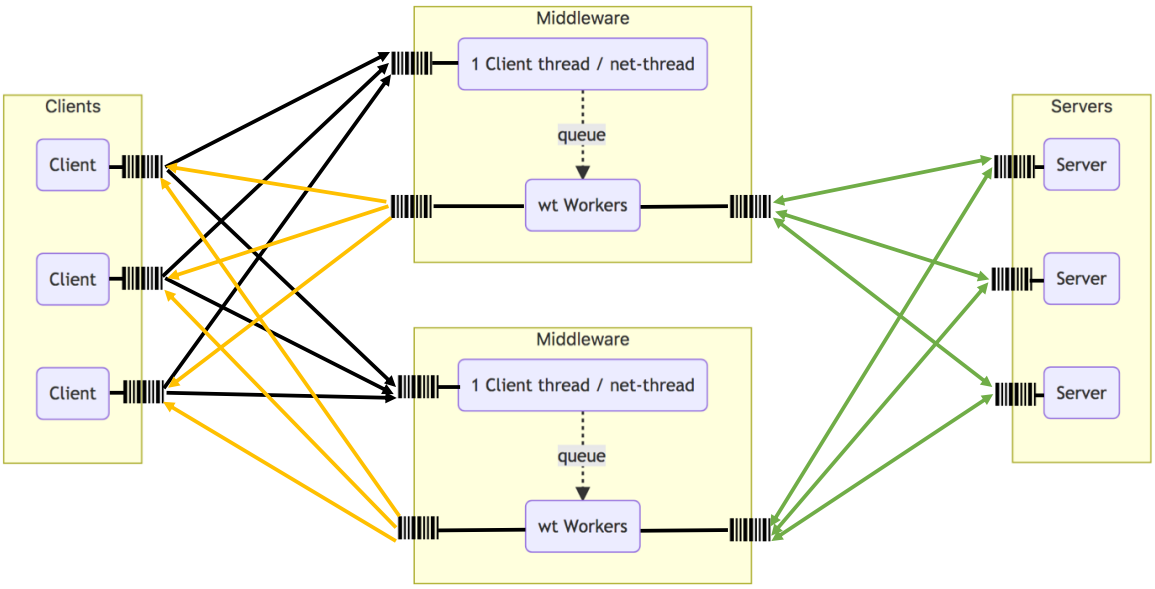


Figure 1: **System overview**: 3 client, 2 middleware, and 3 server VMs; up to 288 virtual clients send requests to the client thread (net-thread) of the middleware instance(s) (black); parsed jobs are enqueued; workers send requests to servers (green), process the replies, and send a reply back to clients (orange); the barcode symbols represent additional queues of the network stack (hardware, virtualization, kernel, runtime system) that exist besides the queue implemented in the middleware; wt: number of worker threads per middleware instance; only partial systems are used in subsections 2.1 (no middleware, only 1 server), 2.2 (no middleware, 1 client VM, 2 servers), 3.1 (1 middleware, 1 server), and 3.2 (1 server).

[1] also provided in html and pdf formats

Table 1: Azure virtual machines, data from [3, 4]

VM	usage	vCPU	RAM[GB]	Disks [a]	Network *send* bandwidth [b] [Mbit/s]
Basic A1	Server, Admin	1	1.75	2	100
Basic A2	Client	2	3.5	4	200
Basic A4	Middleware	8	14.0	16	800

[a] Each disk is allowed 300 I/O operations per second (IOPS). Standard disks have a listed bandwidth limit of 60 MB/s. Only one 30 GiB standard HDD is used per VM.
[b] This bandwidth limit has been measured with `iperf`. Current documentation does not list any network bandwidth limit for the Basic VMs [3, 4, 5]. Please note: network *receive* is not limited by this limit but is CPU bound (see results and discussion).

1.1 Servers: memcached

Memcached v1.4.25 [1], a high-performance in-memory key-value store, is used as server. Each memcached instance – configured to use 1 thread – runs on a Basic A1 VM (Table 1).

1.2 Clients: memtier_benchmark

Memtier_benchmark v1.2.14 [2], a traffic generation and banchmarking tool by RedisLabs, is used as load generator. Instances of memtier_benchmark – configured to simulate specific numbers of virtual clients as explained in the experiments – run on Basic A2 VMs (Table 1).

Payload size has been fixed to 4096 bytes. Number and lifetime of keys has been set as indicated in the project description. The virtual clients are configured to use different random seeds, thus reflecting truly different clients and not just following all the same key access pattern. For read-only and mixed-workload experiments, memcached is pre-filled with data to have a get-miss-rate of 0.0 %.

1.3 Middleware

The self-written middleware (v2.1.3) was used as middleware between clients and servers. It offers all the functionality (replication, load-balancing with round-robin, sharded/non-sharded gets, etc.) as specified in the project description. It uses the Java 1.8 API, builds with `ant` to `middleware.jar`, and starts with the provided *RunMW* class. No external libraries are used. Each middleware instance runs on a Basic A4 VM (Table 1).

1.3.1 Threads

ClientThread The *ClientThread* (net-thread) runs in the main thread of the Java program after launching all worker threads. This thread accepts new client connections that are kept alive during the entire run: a separate *ClientConnectionHandler* is used for each client connection. *ClientThread* reads and parses all the requests from all clients, and enqueues completely parsed requests as jobs into a multi-threading-safe *LinkedTransferQueue*.

WorkerThread n *WorkerThread* instances are launched as defined in the `-t n` option using a *Executors.newFixedThreadPool(n)*. Each worker connects to all defined memcached servers. Each server connection is handled by a separate *ServerConnectionHandler*. All workers and all these connections are kept alive during the entire runtime. As specified, each worker handles only one job at the time in a sequential order:

1. dequeue job (`LinkedTransferQueue.take()` allows waiting without busy polling/idling if the queue is empty)
2. request(s) are sent to one or all connected servers dependent on job type and middleware configuration (sharded)
3. round-robin counter is incremented accordingly to use all servers equally
4. worker waits for server reply/replies and parses it/them
5. worker sends reply to client
6. embed collected instrumentation data into stats (`Stats.processJobTimestamps(Job job)`)
7. reset internal state and loop to beginning of this sequence

1.3.2 Data flow

All requests are completely parsed by the *ClientThread* and enqueued. All *WorkerThread*s can handle all request types and handle them as defined here. The worker is sending the reply to the client.

set Set requests are forwarded to all servers, all replies are awaited and the reply to the client reflects the replies of the servers (typically `STORED\r\n`; one of the error messages in case of error).

non-sharded get The entire get request is forwarded to the server with current round-robin number. The server reply is forwarded to the client.

sharded get Requested keys are distributed among the servers as required following the round-robin number. The algorithm assures that each server receives a continuous sequence of keys, which allows simple combination of the received replies also in case of keys not found by the memcached instance. Replies of all involved servers (typically all 3; may be 1 for 1 key or 2 for 2 keys) are awaited, the replies combined for the final reply to the client. In case of an error message from a server, one of the server error messages is forwarded instead.

Load balancing in the middleware As suggested, a round-robin system was implemented for load balancing of the requests among the memcached servers. Workers start using different servers (initial `roundRobinNr = threadNr % mcAddresses.size()`) leading to a well-balanced start with all servers. Server usage is instrumented in detail (number of uses to handle requests, number of keys, data throughput, server response time listed as RTT) and thus available for analysis. The implementation guarantees that an equal number of requests and keys are sent to each server in all experimental settings.[2]

Error handling The middleware implements error handling as described in the project description. Details associated with the fact that memtier cannot actually process error messages are discussed in section 10 of the technical notes.

1.3.3 ShutdownHook

As suggested in the project description, a *ShutdownHook* thread is registered in the Java runtime to handle proper shutdown, summarizing of statistics and logging of all data when the Java

[2] Additional discussion of here not relevant corner cases due to the simple round-robin system can be found in sub-section 4.2 in the technical notes

runtime receives SIGINT (user CTRL-C) or SIGTERM ("kill" with default signal). Please note: the middleware does *NOT* listen to any input from stdin.

Instrumentation data are collected during the experiment and summarized (stable / all windows) during the shutdown procedure by each worker independently. However, the ShutdownHook thread aggregates the data of all workers for final output. The implementation uses proper mechanisms to assure sequential consistency and memory visibility for this switch (Java memory model). See sub-sections 4.4 and 4.5 in the technical notes for details.

1.3.4 Networking

Networking is implemented in 2 layers.

- upper layer: *ClientThread* (net-thread) and *WorkerThread*
- lower layer: *ClientConnectionHandler* and *ServerConnectionHandler*

Lower layer *ClientConnectionHandler* and *ServerConnectionHandler* handle a connection with one client (or one server) each. They use Java's NIO interface to handle network I/O. This is a change compared to my initial middleware implementation that used plain Java sockets. I changed the implementation to avoid several of the problems observed then.

The current implementation with its event-triggered design and clean state-machines for parsing client requests and server replies is much easier to reason about and shows to be working much better than my old implementation. Additionally, using the lower-level NIO interface of Java instead of the wrapping stream interface allows the middleware to instrument several details of the network communication that are happening in the background also when the stream interface is used. But they cannot be measured then.

However, I went to great lengths to assure that the behavior of the client and server fully matches the sequential network usage pattern as described in the project description and emphasized in the exercise session. This is *not* a simple "08:15" NIO implementation that just lets the async/non-blocking interface run as a black box. Thus, the measured values are fully usable for comparison with the queueing models used in this project.

As a consequence, the implemented sequential execution of the jobs does not unleash the full performance potential that could come from a NIO implementation where each worker thread might handle even several requests in parallel to have less waiting time. This restriction has been made on purpose to remain fully compliant with the requirements of the project description and to have an execution model that is easier to reason/model (see applied queueing theory) than a fully async/non-blocking networking I/O system.

Upper layer *ClientThread* (net-thread) and *WorkerThread* use the objects of the lower layer to organize all connections with clients and servers and to implement the actual "business logic" of the middleware to handle all requests properly as defined in the project description and explicitly guaranteed in the next paragraph. Additionally, each worker tracks instrumentation data in its own *Stats* object. Blocking functions are used (`selector.select()` for network I/O and `queue.take()` to dequeue a job from the queue). Therefore, there is no busy polling/idling in the middleware.

Details of the implementation (organization in packages, interfaces, etc.) are explained in the design notes.

Implementation guarantees Explicitly, the middleware implementation assures the *sequential execution* of all steps in the correct designated thread (see details in 1.3.1; more details in appendix) as defined in the project description and required for modeling.

1.4 Admin VM

In addition to the described client, server and middleware VMs, one Basic A1 VM was created as admin / login node. This allowed controlling all experiments from within the Azure cloud, which was considered to be more reliable than control from the outside.

1.5 Instrumentation

In the middleware alone, more than 50 variables are tracked and used for analysis. Please see section 7 in the technical notes for details. Here, the key information are summarized.

Middleware Many variables are based on timestamps collected in the *Job* object while the request is being processed. The worker embeds the calculated time intervals into the *Stats* object. All data is kept in RAM that was pre-allocated before the first client connection was made and kept until end of shutdown procedure.

Each *WorkerThread* has its own *Stats* object that keeps this data in 2 different data container types: *StatsWindow* and *HistogramSet*. Each window lasts for 1 s. Of the default 100 windows allocated at the beginning, the windows 20 (inclusive) to 80 (exclusive) are aggregated as stable / steady state average during shutdown procedure. Time intervals for ResponseTime, QueueingTime, ServiceTime, and RTT of each server are tracked in histogram bins (0.1 ms each up to max. 500 ms).

Additionally, all instrumentation data is collected in detail for different types of operation – set, get with each key count – and then summarized into average of all gets and average of all requests during shutdown. Thus, *both*, detailed and aggregated data, are available for the analysis.

Queue length is measured in the *ClientThread* with a frequency of 10 Hz and added to the current job for data collection. Measuring queue length in the *ClientThread* comes with the advantage that queue length corresponds well with the queueing time for this particular job.

System data available to the Java program (CPU usage, memory allocation, Garbage collector runs) are collected once per window by worker thread 0.

Memtier Memtier output (json, stdout, stderr) are stored in files for each run and then analyzed in data processing. Memtier offers request and data throughput, response times as average and average per 1 s windows. Additionally, histograms of response time are available. However, some data are only available in aggregated form (set and get requests combined). No information is available about the "service time" that is needed to create/process requests. S_{client} and D_{client} are approximated by upper bounds as explained in subsection 2.1.

System tools Bandwidth of the network connections is measured before each experiment type using `iperf`. Connections are measured sequentially and parallelly simulating the write-only and read-only workloads of the middleware. Network RTT are measured by `ping` running on all network connections used in the experiment using default payload 64 bytes. System data (CPU, memory, disk and network I/O) are collected by `dstat` running on each VM. `ping` and `dstat` are collecting data with 1 Hz during the entire run of the experiments. At this frequency, the contributions of both programs to CPU and network I/O are negligible.

1.6 Data collection

Data collection has been designed to be self-documenting. Thus, in addition to all data files, the entire set of used scripts and the entire run log is stored in the data folder. Each run folder

contains various experiments that have all been run during the same deployment of VMs in the cloud. This includes 5 runs (see data folder):

- Experiments of sections 2.1 and 2.2 (3 clients, 2 servers)
- Experiments of sections 3.1 and 3.2 (3 clients, 2 middlewares, 1 server)
- Experiments of sections 4.1 to 6.1 (3 clients, 2 middlewares, 3 servers)
- Additional experiment 810 (2 hours continuous run; 3 clients, 2 middlewares, 3 servers)
- Additional experiment 820 (4.1 with more workers; 3 clients, 2 middlewares, 3 servers)

Each run of experiments comes in its own zipped folder that contains collected data, processed data and figures (PDF format). Details of the workflow, the files and all scripts are explained in the `README.md` file that is in each of these data folders. Noteworthy is the file naming system that embeds all experiment information for easier data processing (see details in subsection 8.1 in the design notes).

1.7 Data processing

The python scripts in `scripts/data_processing` process all data files, aggregate data, create summary statistics, and the figures. Basically a database is created for each experiment containing all raw and aggregated values. Proper statistics methods are used. For documentation, this database can be stored as json file for each experiment.

1.7.1 Statistics

Due to few missing data files during testing, the decision was made to run 4 iterations / repetitions for each experiment to increase the chance to have at least 3 everywhere. The actual experiments went very well in general. Thus, data are shown as mean ± SD of these 4 iterations in this report unless mentioned otherwise.[3]

1.7.2 Operational laws

The operational laws (OL), such as interactive response time law (IL), Little's law, forced flow law, utilization law and asymptotic bounds are used as defined in chapter 33 of the recommended book [7] to validate the measured experimental data and to gain additional insight into the system, in particular also for bottleneck analysis.

The utilization and bottleneck analysis used in sections 2 to 6 are based on forced flow law and do not depend on any concrete model (M/M/1, M/M/m, MVA with network of queues), which may give additional insights in section 7. The used standard variables are explained in Table 1 of the appendix. I refer to the book chapter 33 [7], in particular also box 33.1, for full definitions and explanations how they correlate with each other.

With the configurations of the systems under test (with $\forall i.V_i \in (0.0, 1.0]$), $D \leq S \leq min(R)$ holds for each configuration[4], which is used to derive some bounds.

Reported maximum throughput in the tables: usable capacity There are various throughput values that may be of interest: knee capacity, usable capacity, nominal capacity (see chapter 3 in the book [7]). Following the explanations in the tutorial/exercise sessions, the maximum throughput is used that can be achieved with a *reasonable* response time, i.e.

[3] Assuming a normal distribution of the random variable, the 95% confidence interval can be estimated from shown mean ± SD with n=4 as $[mean - 1.6 \cdot SD, mean + 1.6 \cdot SD]$ using SEM and two-tailed t-distribution.

[4] e.g. number of client/middleware/server VMs, network connections between them, number of worker threads in the middleware, workload type, same deployment of the VMs

the usable capacity. The throughput vs response time plots (see experiments) are used to determine this data point: follow the data points with increasing number of virtual clients until the throughput does not increase significantly. This setting with N_{uc} virtual clients (jobs in the system) is used as usable capacity. Being the start of the throughput plateau, it is often also the knee capacity (as well as it can be found with the experiment resolution). The response time associated with this data point was considered to be reasonable. With more than N_{uc} clients, only response time increases (queueing time) without significant increase of the throughput.

System saturation N_{uc}, as determined above, was used to define the start of the saturation phase of the system. It typically coincides with the knee capacity in the system under test (SUT), where relevant queueing starts (creating the knee in the response time plot). Finding the knee with asymptotic bounds ($N^* = \frac{D+Z}{D_{max}}$) [7] worked quite well in section 2 without middleware but not in the more complex systems with middleware, where we are not interested when *any* queueing starts in the system but the relevant queueing in the middleware indicating that all workers are busy (see appendix).

Thus, the system state is classified as saturated with number of jobs (= number of virtual clients) $N \geq N_{uc}$, and under-saturated with $N < N_{uc}$. No over-saturated state has been detected in the experiments: the system under test (SUT) is a *closed* system.[5]

Bottleneck analysis I: device utilization Utilization U_i is calculated for each device i. The device with highest utilization is the bottleneck device.

$D_{middleware}$ is available by instrumentation of the middleware and defined as $D_{middleware} = (PreprocessingTime + ProcessingTime) * V_{middleware}$ with $V_{middleware} = 1/8$ for one middleware and $V_{middleware} = 1/16$ for two middleware instances.[6]

D_{server} and D_{client} are not available directly by instrumentation of memcached and memtier, respectively. These two key variables are derived in the baseline experiments (see section 2). This procedure allows defining separate "devices", with their own service demand as explained in detail in the result/discussion sections 3 to 6: D_{cs}, D_{sc}, D_{cm}, D_{mc}, D_{ms}, D_{sm} for connections (from, to) between client, middleware and server instances.

Bottleneck analysis II: system configuration Independent of the utilization calculation above, a device (in particular middleware) becomes a bottleneck if its configuration leads to a system state that prevents a subsystem to reach its usable capacity, e.g. too few worker threads

[5] Closed systems have a self-regulatory circuit based on utilization - response time relationship: when utilization goes towards 1, then response time increases, up to infinity [7]. Longer response time leads to fewer requests that can be sent by the clients. Thus, the closed system regulates itself into a steady state of balanced utilization and response time.

[6] There are multiple views that can be expressed with the V_i values for the devices i (number of VM, number of vCPU, number of threads) each with unique weight and interpretation. After consideration, the number of used vCPU has shown to be the best denominator for the bottleneck analysis here, in particular for cross-comparison between different devices. See Table 1 for the list of vCPUs per VM type. Conveniently, send network bandwidth limit is 100 Mbit/s per vCPU. Thus, the following definitions have been used: $V_{client} = 1/2$ for one client VM; and $V_{client} = 1/6$ for three client VMs. $V_{server} = 1$ for one server VM; $V_{server} = 1/2$ for two server VMs without middleware; due to replication mechanism in the middleware this value differs for different request types and 3 server VMs: $V_{server} = 1$ for set and sharded get, $V_{server} = 1/3$ for non-sharded get. $V_{middleware} = 1/8$ for one middleware VM; and $V_{middleware} = 1/16$ two middleware VMs. For more detailed view, $V_{middleware,workerthread} = V_{middleware}$, but $V_{middleware,clientthread} = 1$ for one middleware and $V_{middleware,clientthread} = 1/2$ for two middleware VMs. These and other model parameters are set in the configuration of my analysis software (`tools/config.py`) or derived during its runtime.

Please note: these V variables relating to the vCPU count of each device are used for operational laws in sections 2-6 and are not the same as the ones defined for the model in subsection 7.3.

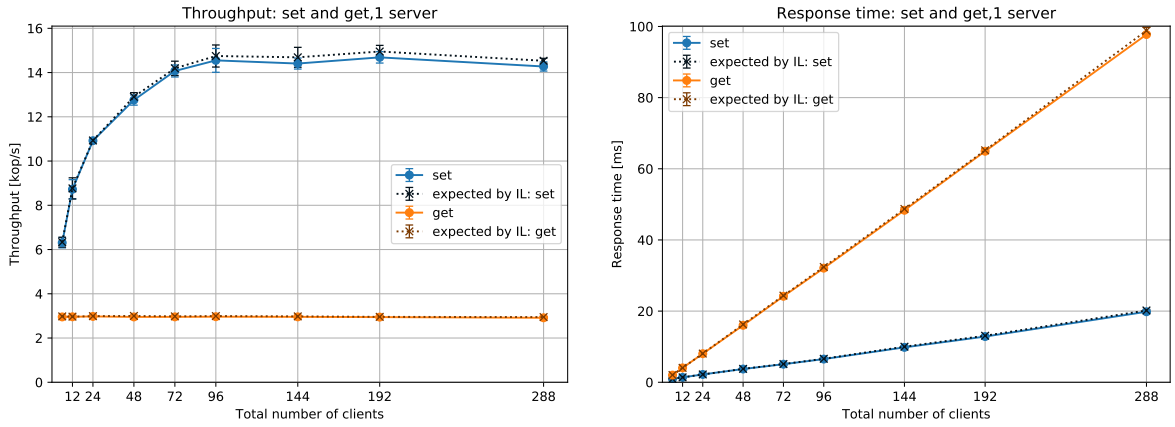
leading to too few middleware to server connections. The complete system (clients, middleware instances, servers) is a system of multiple subsystems.

2 Baseline without Middleware (75 pts)

2.1 One Server

For this baseline, 3 client VMs are connected with one memcached server. The client VMs generate a workload of 6 to 288 virtual clients. One goal of this experiment is to determine the maximum possible throughput of a single memcached server in the defined settings in the Azure cloud. Write-only workload leads to large amount of data (see 4 KiB payload) sent from clients to the server; read-only workload leads to large amount of data sent from server to clients.

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	1, 2, 4, 8, 12, 16, 24, 32, 48
Total number of clients	**6, 12, 24, 48, 72, 96, 144, 192, 288**
Workload	write-only (set) and read-only (get)
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	4 repetitions; 60 s steady-state each



(a) Throughput

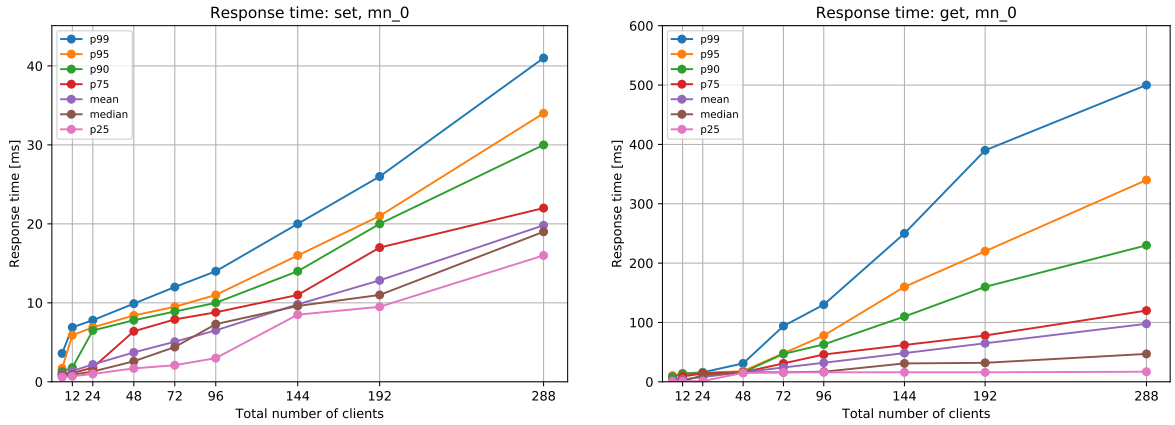
(b) Response time

Figure 2: Baseline with one memcached server

2.1.1 Explanation

Throughput and response time for write-only (set) and read-only (get) workloads are shown in Figure 2 as measured in the memtier clients (mean ± SD, $n = 4$). Both figures include the expected throughput and expected response time, respectively, as calculated by the interactive response time law (IL) assuming a thinking time $Z = 0$ ms. The measured data match well with the expected values of the IL. The small difference[7] is in part explained due to the fact

[7] approx. 4 to 360 μs for write-only workload and approx. 16 to 1000 μs for read-only workload dependent on the number of virtual clients that need to be simulated by memtier thread; this is only an approximation because



(a) Write-only workload

(b) Read-only workload

Figure 3: Response time percentiles, baseline with one server (note different y scales)

that processing in memtier is not instantaneous. This difference can be used as an upper bound for S_{client} and consecutively D_{client} (no detailed instrumentation data available for memtier).

The linear increase of the average response time (Figure 2) is caused by the fact that the memcached server uses only one thread to fulfill all requests. Therefore, queueing starts very early as can also be seen by the low N^* numbers (≤ 13.3 for write-only, ≤ 6.5 for read-only, see appendix).

However, the "smoothness" of the mean response time reveals only a simplified view of the system. Percentiles (Figure 5) indicate better how the system responds to the users revealing long tails in the distribution of response times.

With the lack of detailed instrumentation data of memcached, D_{server} and following utilization U_{server} are derived (or at least bounded) from measured data using operational laws (Table 2 in the appendix). Write-only and read-only workloads must be distinguished.

write-only As defined in subsection 1.7, the system is saturated at $N_{uc} = 72$ and under-saturated with fewer clients. No over-saturated phase could be detected in the used parameter envelope (even when using 384 clients during testing). While closed systems are self-regulating and do not "explode" like open systems (see explanation in 1.7).[8] Throughput and response time correlate with each other in the inverse relation as defined by the interactive law, which holds for all experiments.

Usable capacity was 14'071 $\pm$ 270 op/s detected at $N_{uc} = 72$ (see Figure 6 for this determination[9]). The server is then the bottleneck (87 % utilization), which is confirmed by CPU usage of 87.5 $\pm$ 1.9 % (`dstat`). This number includes all aspects of the server (receiving network stack in kernel and standard library[10], memcached service, sending network stack including limited send bandwidth as listed in Table 1).

there is also some noise in the measurements of throughput and response time; thus the values may only be used as an upper bound but not as a precise estimate

[8] As a thought: running a closed system with an extreme abundance of clients might also lead to thrashing.

[9] Throughputs for 72 clients 14'071 op/s, 95 % CI [13'641, 14'501] and 96 clients 14'550 op/s, 95 % CI [13'692, 15'408] are not significantly different. The 33 % increase of clients from 72 to 96 does not lead to a significant increase of throughput but a wider SD and is associated with an increase of the response time by 28 % (non-overlapping 95 % CI)

[10] there are other aspects of the network stack, of course, that are not mentioned here because they cannot be measured directly, such as hardware with queues, virtualization of the machines, etc.

However, detailed analysis of the server CPU usage (total 87.5 %, user 20.4 %, system/kernel 51.3 %, softirq 15.8 %) confirms what is speculated from knowledge of the system and in particular network functionality: not the memcached service is the bottleneck but the receiving network stack (see system/kernel time, softirq handling). `dstat` does not list hardware interrupts, probably due to the virtualization of the machine. In contrast to the hard limit for send network bandwidth, each VM can receive as much data as its CPU can handle. Thus, write-only workload is not limited by the network bandwidth policy (would allow up to 18.3 kop/s).

This kernel CPU usage cannot be explained by memory requests to the kernel because the number of keys used in the tests is limited. Even for replacing keys with new set requests, this can be handled in the heap by the standard library without syscalls to map more memory. Disk I/O is negligible during the experiments.

read-only In contrast to write-only workload, the read-only workload is limited by the network send bandwidth policy in the Azure cloud. As measured with `iperf`, the Basic A1 VM of the server can only send 100 Mbit/s (Table 1). This bandwidth limitation is the clear bottleneck (utilization 97 %).

The calculated upper bound of server utilization ($\leq$ 85 %) is a clear over-estimation influenced by this bandwidth limitation (see method). Using D_{server} from write-only, U_{server} is probably closer to 18 %, which matches better the measured CPU usage (10 %, `dstat`).

The system is saturated for all measured number of clients. Also here, no over-saturation. The knee at N^* (based on OL) is ≤ 6.5. Based on Figure 6, $N_{uc} = 6$ was used to determine the usable capacity of the system with 2'954 $\pm$ 8 op/s. Please note: get miss rate was 0.0 % for all experiments.

2.2 Two Servers

For this baseline, one client VM is connected with 2 memcached servers. The client VM generates a workload of 2 to 64 virtual clients. One goal of this experiment is to determine the maximum possible load that can be generated by a single client VM in the defined settings in the Azure cloud.

Number of servers	2
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	1, 2, 3, 4, 8, 12, 16, 24, 32
Total number of clients	**2, 4, 6, 8, 16, 24, 32, 48, 64**
Workload	write-only (set) and read-only (get)
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	4 repetitions; 60 s steady-state each

2.2.1 Explanation

Throughput and response time for write-only (set) and read-only (get) workloads are shown in Figure 4 as measured in the memtier clients (mean $\pm$ SD, $n = 4$). Also here, and fully expected, the measured data match well the expected values as calculated using the IL.

The table with calculations for operational laws, including utilization and bottleneck analysis is in the appendix.

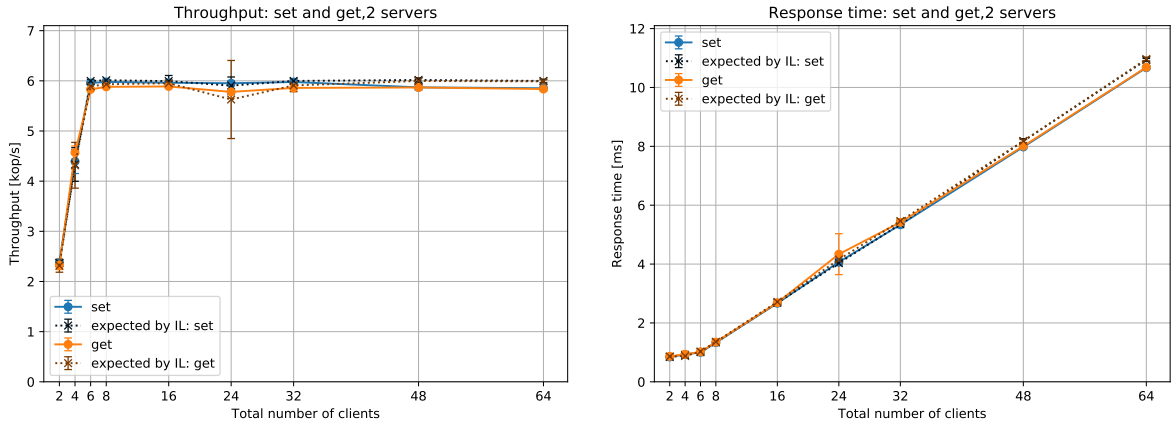
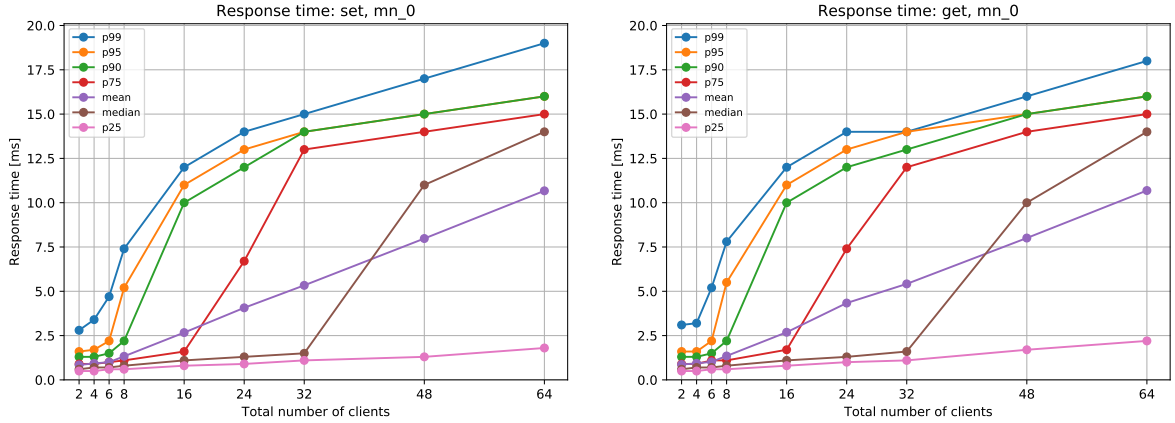


Figure 4: Baseline with one client VM



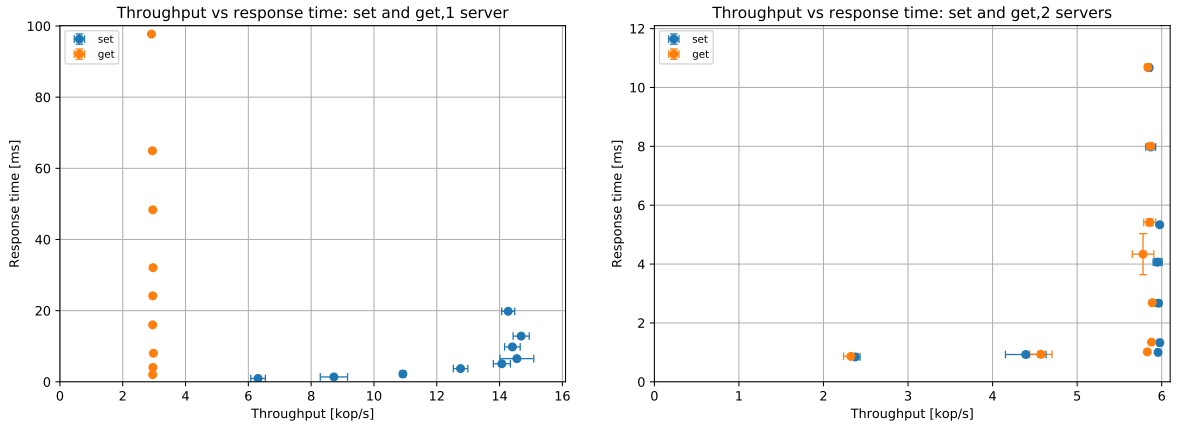
(a) Write-only workload

(b) Read-only workload

Figure 5: Response time percentiles, baseline with one client VM; the observation of several percentiles collapsing together (e.g. p90 and p95 for ≥ 32 clients for write-only workload) is an artifact due to limited resolution in the histogram/CDF output of memtier.

write-only Starting at about $N_{uc} = 6$ clients, the system is saturated (knee by OL at $N^* = 5.9$). It is under-saturated below. No over-saturation phase could be observed with the same argument as in subsection 2.1. The observed usable capacity 5'956 $\pm$ 24 op/s for one client VM ($N_{uc} = 6$) confirms that the limitation in subsection 2.1 has been the server: 3x 5'956 = 17'868 op/s $\geq$ 14'071 op/s. Neither client nor server VM are really challenged in this experiment (see CPU usage < 22 % for all workloads in subsection 2.2). The send bandwidth limitation of the Basic A2 VM for the client is the clear bottleneck (utilization 98 %, see appendix).

read-only With the doubling of the server send bandwidth from 100 Mbit/s (one server; subsection 2.1) to 200 Mbit/s for both servers (subsection 2.2), the observed usable capacity also doubled from 2'954 $\pm$ 8 op/s to 5'830 $\pm$ 17 op/s ($N_{uc} = 6$) confirming again the bandwidth limitation of the A1 VMs as the bottleneck for read-only workload (utilization 96 %, see appendix). At measured $N_{uc} = 6$ clients, the system is saturated (saturation may start at $N^* = 4.3$ by OL). It is under-saturated below. No over-saturation phase could be observed (same argument as in 2.1).



(a) one memcached server (subsection 2.1)

(b) one client VM (subsection 2.2)

Figure 6: Throughput vs Responsetime

2.3 Summary

Table 2: Maximum throughput of different VMs (usable capacity; mean ± SD, $n = 4$).

	Read-only (r/o) workload	Write-only (w/o) workload	Configuration gives max. throughput
One memcached server	2'954 ± 8 op/s	14'071 ± 270 op/s	r/o: 6 clients (VC=1); w/o: 72 clients (VC=12)
One load generating VM	5'830 ± 17 op/s	5'956 ± 24 op/s	both: 6 clients (VC=3)

The network send bandwidth limitation of the Azure VMs (Table 1) has been the clear bottleneck in 3 of the 4 experiment settings. Only for write-only workload with one server (subsection 2.1) the CPU of the server has been the bottleneck, mainly due to kernel operation while handling incoming traffic. Neither memtier nor memcached themselves have ever been close to their throughput limitations.

It has been one of my goals in these baseline experiments to also deduce S_{client} and S_{server} well for use in subsection 7.3 based on the operational laws. Due to the given network bandwidth limitations and unavailable more detailed instrumentation options in memtier and memcached, only coarse upper bounds could be deduced (see subsections above). CPU usage (a surrogate for utilization of the VMs) confirmed the utilization calculations with operational laws.

Take-away messages The server VMs (Basic A1) have been the critical bottleneck for both, receiving (CPU bound) and sending (bandwidth limitation by policy) network traffic. Changing these VMs to more powerful machines would increase throughput for both workload types even if only one memcached service with one thread would be kept using on these more powerful machines. Conceptually the same limitation applies to the client VMs (Basic A2) if only one is used. However, typically 3 of them are used in the experiment settings, which removes them from being the bottleneck.

3 Baseline with Middleware (90 pts)

3.1 One Middleware

For this baseline, 3 client VMs are connected with one middleware that is then connected with one memcached server. The client VMs generate a workload of 6 to 288 virtual clients; the middleware uses 8 to 128 worker threads to handle the requests. This baseline allows comparing the effect of an added middleware to the system as tested in subsection 2.1.

Due to an additional network connection, it has to be expected that response time is longer and, thus, throughput is lower than in 2.1 even if the middleware could handle requests instantaneously.

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	1, 2, 4, 8, 12, 16, 24, 32, 48
Total number of clients	**6, 12, 24, 48, 72, 96, 144, 192, 288**
Workload	write-only (set) and read-only (get)
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1
Worker threads per middleware (WT)	8, 16, 32, 64, 128
Repetitions	4 repetitions; 60 s steady-state each

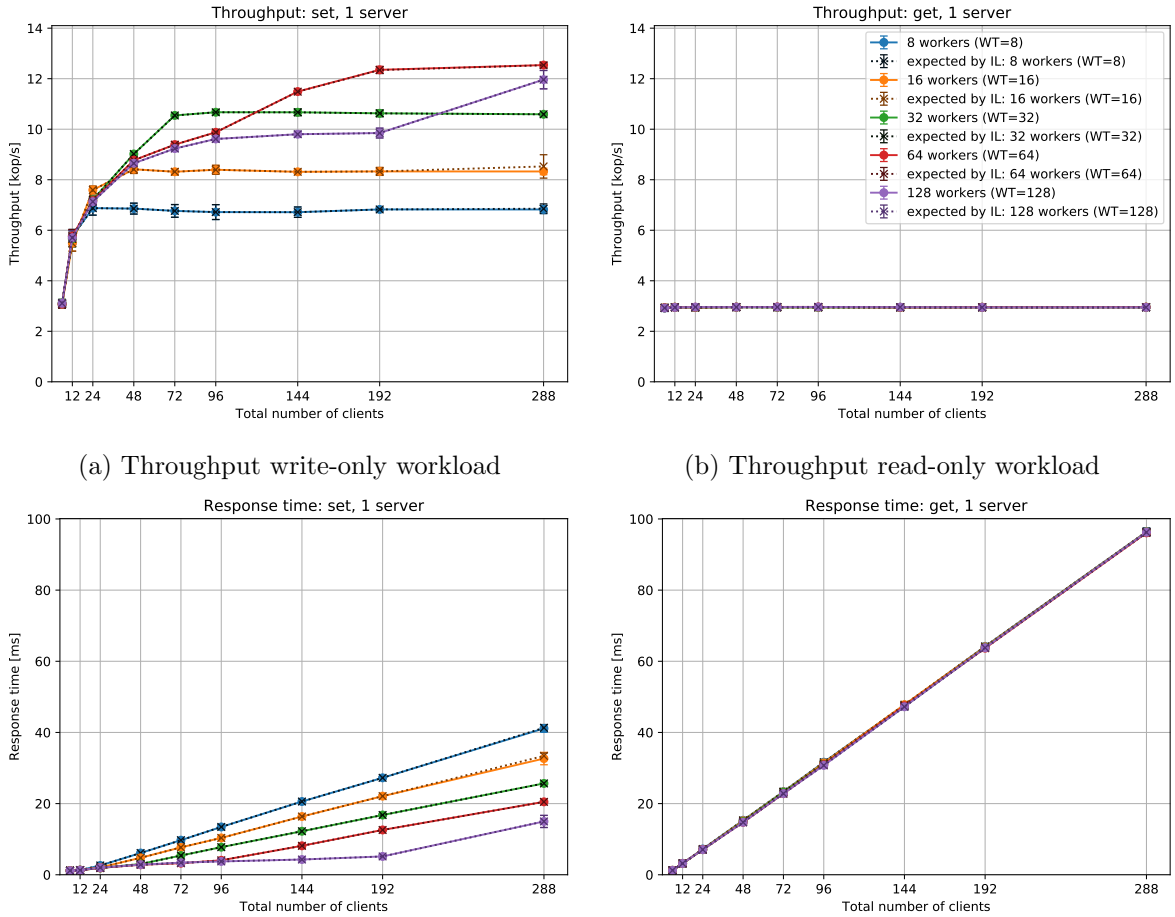
3.1.1 Explanation

Data have been collected at both, the memtier clients and the middleware. Figures show the values (mean ± SD) as measured at the middleware. Standard deviations are typically so small that the plotted error bars are barely visible. The measured time `ClientRTTAndProcessingTime` is used as thinking time Z to plot the expected response time and expected throughput based on IL. The expected and measured values match so well that both markers are typically on top of each other. Thus, IL holds. No errors occurred during the experiments; all get requests were found.

Write-only: best configuration Using 64 worker threads in the middleware achieves highest throughput (Figure 7 (a) red curve). Based on asymptotic bounds (OL) some queueing starts with $N^* = 24.2$ jobs in the system. The system becomes saturated (as defined in 1.7) with $N_{uc} = 192$ and is under-saturated with fewer clients. Also here, no over-saturated phase (see earlier explanations).

An increase in response time (Figure 7 (c) red curve) can be observed after 96 clients indicating that queueing starts then in the middleware (basically a knee). This is associated with an increase in throughput again, which seems to be paradoxical at first. However, with only few clients, i.e. few jobs in the system, most of the workers are actually sleeping while waiting to dequeue a new job. Only when the queue starts to be non-empty all the time, all workers can work. Indeed, with 72 virtual clients average queue length in the middleware is 0.1 jobs, and 40 of the 64 workers are waiting to dequeue a job in average. In comparison, with 96 clients average queue length in the middleware starts growing with 2.2 jobs and waiting workers decreasing (33 workers); with 144 clients further increasing queue length (32 jobs) and decreasing number of waiting workers (13).

In the same context, the queueing time of 7.6 ± 0.1 ms with $N_{uc} = 192$ (Table 4) seems to be long at first (approx. half of the response time). On the other hand, this corresponds to an average queue length of 81.5 ± 2.6 jobs, barely more than one job per worker. Accordingly,



(a) Throughput write-only workload

(b) Throughput read-only workload

(c) Response time write-only workload

(d) Response time read-only workload

Figure 7: Baseline with one middleware and one memcached server; legend idem for (a)-(d)

the average number of waiting workers to dequeue jobs has decreased to 3.9 ± 1.6.[11] Only a non-empty queue prevents workers from waiting.

The server is the bottleneck (utilization 87 %) in this configuration middleware configuration with 64 workers (value from $N_{uc} = 192$). In comparison, utilization of client (29 %) and middleware (22 %)[12] are much lower. These calculations by OL are supported by the measured average CPU usage for client (13.9 ± 0.3 %), middleware (24.8 ± 0.4 %), and server (72.8 ± 3.0 %; mainly system/kernel/softirq with 51.5 %; user with 21.3 %) VMs for this experiment setting.

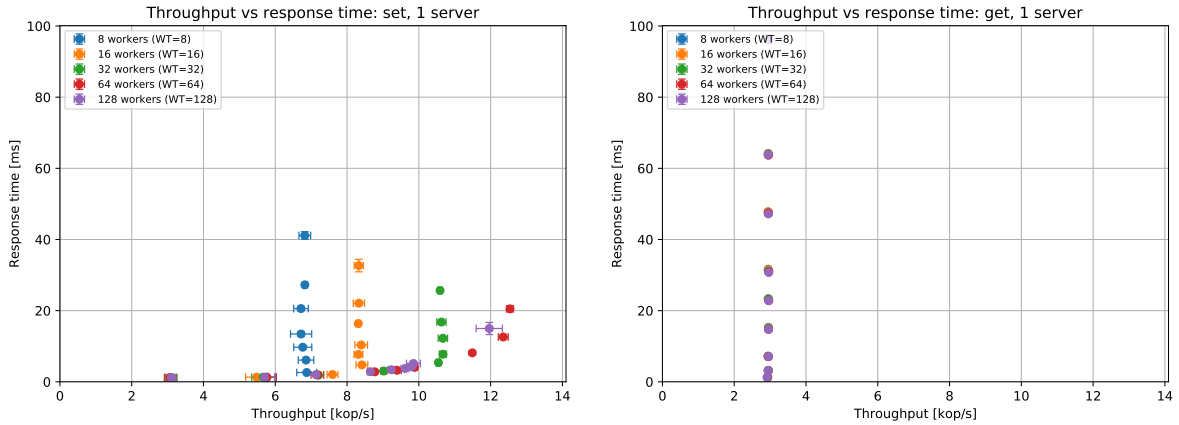
As seen in subsection 2.1 (3 clients, 1 server), the network bandwidth is not the bottleneck but the server vCPU handling incoming network traffic.

Usable capacity with the middleware (12'282 ± 183 op/s) is lower than observed in the baseline without middleware (14'071 ± 270 op/s) as expected because of the increased network latency of 2 network connections (client - middleware; middleware - server) compared to direct connection (client - server).[13]

[11] number of waiting workers decreases further to 0.4 with 288 clients

[12] detailed component analysis for middleware: client thread 38 %, worker threads 17 %

[13] `ping` RTT data available for all connections for the entire duration of the experiments



(a) write-only (set)

(b) read-only (get)

Figure 8: Throughput vs response time with one middleware and one memcached server

Write-only: fewer worker threads Middleware and server build a subsystem that corresponds to the baseline system analyzed in 2.1. In contrast to the 3 client VMs the middleware VM is more powerful (8 instead of 6 vCPUs; 800 Mbit/s instead of 600 Mbit/s) thus not introducing a new bottleneck. Due to the specific design of the workers (sequential execution of requests; subsection 1.3), the number of worker threads can be seen as the number of clients.[14]

Table 3: Usable capacity for various middleware configurations for write-only workloads

Workers	N_{uc}	$X(N_{uc})$ [op/s] [a] (mean ± SD, n=4)
8	24	6'876 ± 277
16	48	8'413 ± 164
32	72	10'546 ± 67
64	192	12'351 ± 137
128	288	11'963 ± 365 [b]

[a] measured at the middleware

[b] more could be achieved with more load generated by more clients

And indeed, achieved usable capacity for 8, 16 and 32 workers (Table 3, Figure 8 for N_{uc}) follows quite well measurements/interpolations in section 2.1 (Figure 2 (a) blue curve) for 8, 16 and 32 clients, respectively. There is no perfect match, of course, due to different VMs used for clients and middleware. But it is sufficiently close to illustrate that middleware configurations with 8, 16, or 32 worker threads prevent the middleware-server subsystem from achieving its maximum throughput (usable capacity).

This bottleneck does not show up in utilization analysis as done above. Also with 8, 16 and 32 threads, the server utilization is larger than client and middleware utilization.

Write-only: more worker threads To test the middleware, also 128 worker threads (requested maximum) was tested. Up to max. tested workload with 288 clients, the system did not reach the best usable capacity of 64 worker threads. However, similar to the described observation with 64 worker threads above, response time does not start to increase relevantly before 192 clients (see knee in Figure 7 (c)) having most worker threads waiting with fewer jobs

[14] `ServersNettoResponseTime` measured in the middleware can be used as response time for this subsystem

in the system. This is not surprising. With 128 workers, 128 jobs can be in the system without any queueing in the middleware at all.

Similarly to the system with 64 worker threads, the throughput increases when more workers are actually busy working. See the increase in throughput with 288 clients after a phase that looked like a plateau (Figure 7 (a)) Thus, it can be assumed that a similar usable capacity could be reached for this configuration with more clients generating load. Utilization calculations (bottleneck $U_{server} = 85$ %) and average CPU usage (client 13 %, middleware 28 %, server 68 %) do not indicate that this could not be reached.

Read-only Throughput and response time remain unchanged for all tested configurations (Figure 7 (b)). Usable capacity (Table 4) is already reached and, therefore, the system saturated with 6 clients. With more clients, queueing time increases without achieving more throughput (Figure 7 (d)). As discussed in detail in subsection 2.1 (baseline with 1 server), the network send bandwidth limitation of the Azure Basic A1 VMs (100 Mbit/s) is the bottleneck of the system. Thus, usable capacity is identical to the measured usable capacity in this baseline experiment (Table 2) Low average CPU usage (all VMs < 10 %) indicates that client, middleware and server could achieve much higher read-only throughput without this bandwidth bottleneck.

3.2 Two Middlewares

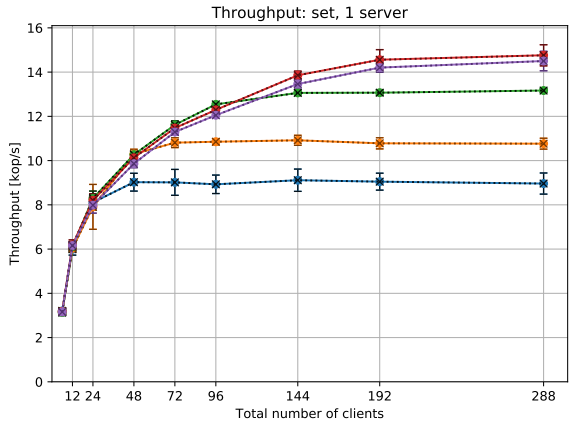
In this experiment, the configuration is modified to use 2 middlewares instead of one (section 3.1) using 16 to 256 worker threads in total. One of the questions is whether this second middleware can compensate for the observed longer response time and lower throughput when one middleware was added to the system. The experiment is run identically as 3.1.

Number of servers	1
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	1, 2, 4, 8, 12, 16, 24, 32, 48
Total number of clients	**6, 12, 24, 48, 72, 96, 144, 192, 288**
Workload	write-only (set) and read-only (get)
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware (WT)	8, 16, 32, 64, 128
Repetitions	4 repetitions; 60 s steady-state each

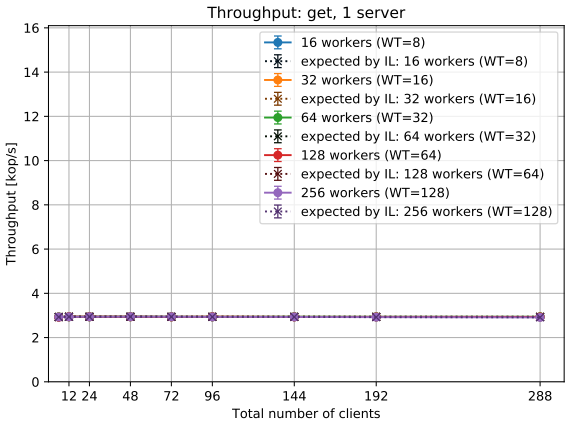
3.2.1 Explanation

Data have been collected at both, the memtier clients and at the middleware. Figures here are shown as measured at the middleware. Interactive law holds.

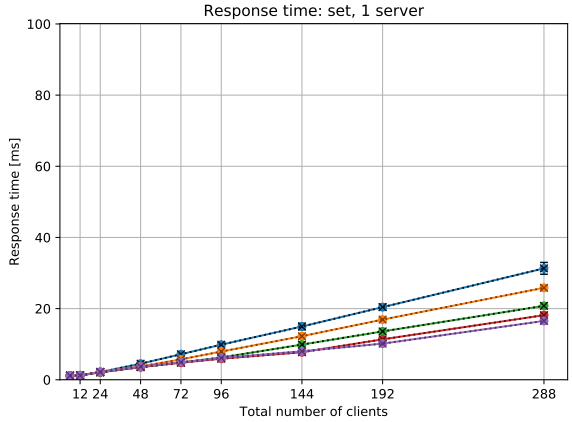
Write-only: best configuration The configuration with 128 worker threads (WT=64) achieves the largest usable capacity of all configurations: 14'376 $\pm$ 434 op/s with $N_{uc} = 192$ (see Figures 9 and 10). N_{uc} also defines the start of the saturated phase of this system configuration. This throughput is significantly larger than the usable best usable capacity with one middleware but is not significantly different from the usable capacity achieved in the baseline experiment without middleware in section 2.1 (overlapping 95 % CI). Therefore, a second middleware can compensate for the introduced delay of the added middleware (see explanation in 3.1).



(a) Throughput write-only workload



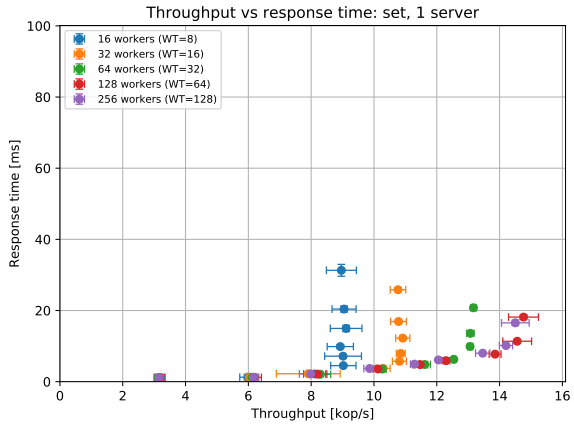
(b) Throughput read-only workload



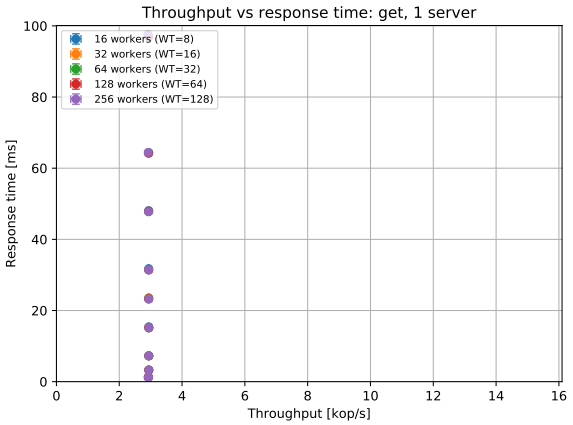
(c) Response time write-only workload

(d) Response time read-only workload

Figure 9: Baseline with two middlewares and one memcached server



(a) write-only(set)



(b) read-only (get)

Figure 10: Throughput vs response time with two middlewares and one memcached server

The server is the bottleneck (CPU handling incoming network traffic; same explanation as in subsection 3.1). As expected, middleware utilization is lower when two middleware instances

are used (14 % compared to 22 % in 3.1).[15]

Write-only: fewer worker threads The same considerations apply as discussed in subsection 3.1.

Write-only: more worker threads Throughput and response time are not significantly different between 128 (WT=64) and 256 (WT=128) workers (overlapping 95 % CI). This indicates that each middleware instance can handle 128 worker threads on 8 vCPU cores without thrashing under the given workload. However, as seen in the baseline experiments, bottleneck is the server vCPU handling receiving network messages. Advantages / disadvantages of using 64 or 128 worker threads per middleware instance cannot be concluded definitively with current data. Both settings have an average CPU usage of approx. 20 % (overlapping 95 % CI) indicating that the middleware itself is far from being the bottleneck and could handle much larger throughput in both configurations.

Read-only Throughput and response time are shown in Figure 9. The system is saturated already with 6 clients. Due to the network send bandwidth bottleneck of the server Basic A1 VM – which has not been changed by adding a the second middleware instance – bottleneck analysis and discussion are identical to the results in subsection 3.1.

3.3 Summary

Table 4: Maximum throughput for *one* middleware (usable capacity of best configuration with 64 worker threads (WT=64); $N_{uc,write} = 192$; $N_{uc,read} = 6$; mean ± SD, $n = 4$).

	Throughput [op/s]	Response time [ms]	Average time in queue [ms]	Miss rate
Reads: Measured on middleware	2'933 ± 8	1.2 ± 0.02	0.06 ± 0.001	0.0 %
Reads: Measured on clients	2'930 ± 7	2.0 ± 0.005	n/a	0.0 %
Writes: Measured on middleware	12'351 ± 137	12.6 ± 0.2	7.6 ± 0.1	n/a
Writes: Measured on clients	12'282 ± 183	15.5 ± 0.2	n/a	n/a

Bottleneck The server VM (Basic A1) is again the critical bottleneck for both, receiving (CPU bound) and sending (bandwidth limitation by policy) network traffic (see recommendation in summary of section 2).

Middleware The middleware in the best configuration setting has never come close to become the bottleneck based on utilization. We do not know what would be the actual limitations of memtier, memcached or the Java middleware.

As explained above, queue lengths in the middleware are rather short at the N_{uc} (with associated short queueing time) due to quite fine-granular experiment configurations. In an

[15] detailed with 2 middleware instances: client-threads 29 %, worker-threads 10 %, both lower than in 3.1.

Table 5: Maximum throughput for *two* middlewares (usable capacity of best configuration with 128 worker threads (WT=64); $N_{uc,write} = 192$; $N_{uc,read} = 6$; mean ± SD, $n = 4$).

	Throughput [op/s]	Response time [ms]	Average time in queue [ms]	Miss rate
Reads: Measured on middleware	2'919 ± 29	1.3 ± 0.03	0.06 ± 0.001	0.0 %
Reads: Measured on clients	2'905 ± 29	2.1 ± 0.6	n/a	0.0 %
Writes: Measured on middleware	14'560 ± 458	11.4 ± 0.4	2.8 ± 0.2	n/a
Writes: Measured on clients	14'376 ± 434	13.1 ± 0.6	n/a	n/a

approximation, the queue length follows the function $max(N - N_{uc}, 0)$. More detailed discussion about average queue lengths is in subsection 4.2.

As expected and explained in 3.1 in detail, usable capacity of one middleware is lower than of the base system (section 2.1) for write-only workloads. Two middlewares achieve the same usable capacity as the base system.

Because the network bandwidth bottleneck for read-only workloads is unaffected by any of these system changes (0, 1 or 2 middlewares) usable capacity is always the same.

Additional observations Analysis of the response time percentiles revealed similar long tails as discussed in section 2. Most of the time (detailed plots of variables available over 100 s of all windows) instrumentation data remained very stable during the steady-state phase. Examples of dynamic behavior occurred also here (see discussed example in section 4). Due to the design of the memory management in the middleware, the garbage collector typically only run for < 50 ms in total during the stable windows of 1 minute.

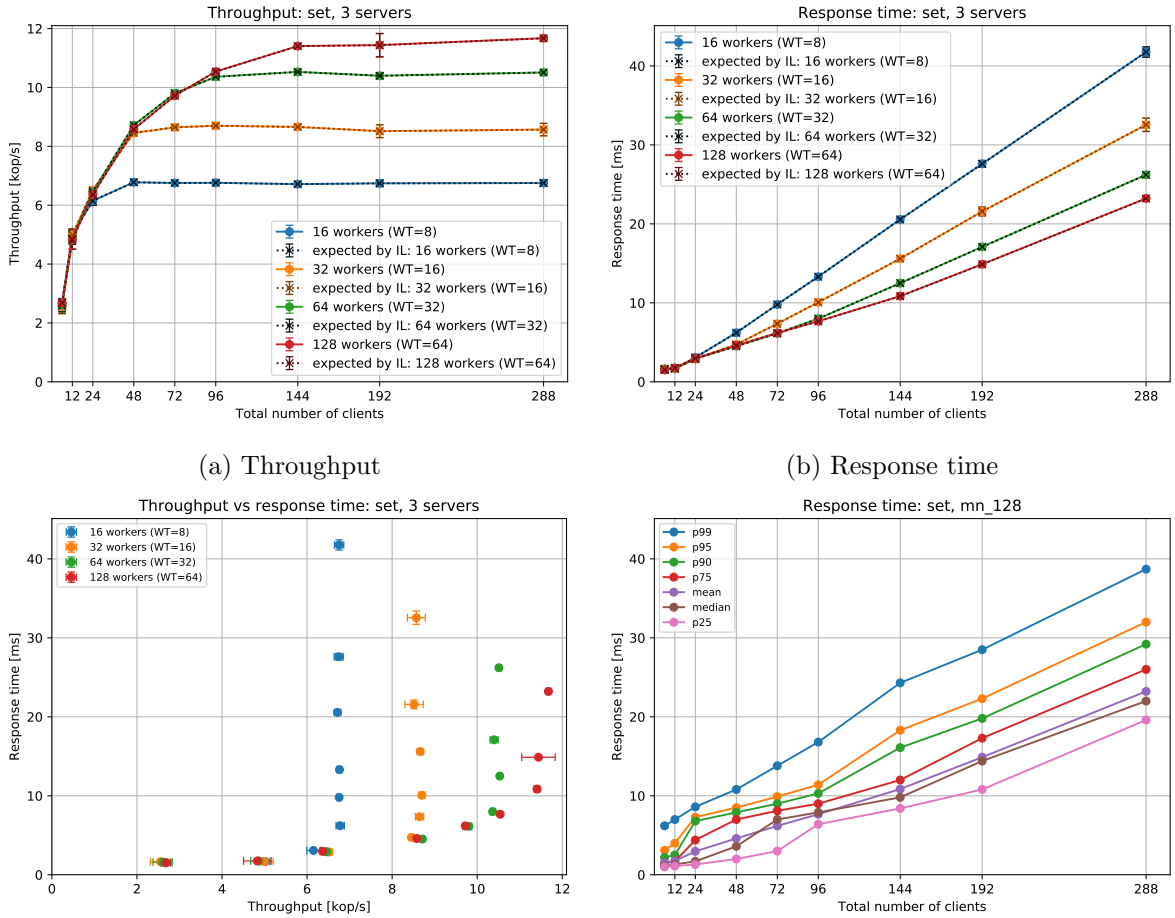
Key take-away message The middleware should not be run in a configuration that prevents a subsystem (here middleware-server) from running with its maximum usable capacity. This creates an artificial bottleneck that cannot be detected by OL based utilization analysis alone.

4 Throughput for Writes (90 pts)

4.1 Full System

Write-only workload is tested on the full system with 3 client, 2 middleware and 3 server VMs (Figure 1). The middleware replicates the set requests to all 3 servers.

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	1, 2, 4, 8, 12, 16, 24, 32, 48
Total number of clients	**6, 12, 24, 48, 72, 96, 144, 192, 288**
Workload	write-only (set)
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware (WT)	8, 16, 32, 64
Repetitions	4 repetitions; 60 s steady-state each



(a) Throughput

(b) Response time

(c) Throughput vs response time

(d) Response time percentiles (WT=64)

Figure 11: Write-only workload: 2 middleware and 3 memcached server instances

4.1.1 Explanation

Saturation, utilization, and bottleneck analysis: best configuration Using 128 worker threads (WT=64) revealed the largest usable capacity (Table 6), which is lower than what could be achieved in the system with only 1 server (subsection 3.2) or in the baseline without middleware (subsection 2.1). This is not surprising based on the overhead caused by replication of the request to all 3 servers, increasing the `ServiceTime` of the middleware that needs to wait for the reply of all servers.[16] The system is saturated with $\geq Nuc = 144$ jobs in the system; it is under-saturated with fewer jobs; no over-saturation phase (as explained in earlier sections).

Utilization at $Nuc = 144$ based on OL: servers (93 %, clear bottleneck), clients (33 %), middlewares (14 %).[17] This analysis is supported by the average CPU usage: servers (54.7 ± 0.5 %),[18] clients (15.3 ± 0.4 %), and middlewares (21.3 ± 0.2 %). Server CPU usage here is

[16] The different VM deployments of experiments for sections 2, 3 and 4 could be another factor causing lower usable capacity than in the former sections (similar to explanation for difference between subsections 2.1 and 3.1; compensated by second middleware in subsection 3.2). However, `ping` data show that the latencies between the relevant VMs are not that different between deployment for experiments of 3.2 and 4 leaving the overhead introduced by replication as main explanation.

[17] detailed component analysis for middleware: client-threads (17 %), worker-threads (12 %)

[18] mainly kernel (system, softirq) 41.1 %, little user (mainly memcached; other applications negligible) 13.6 %, as seen in sections 2 and 3

lower than seen in section 3 because of lower overall throughput of the system due to the delays in waiting for replies of all servers.

Saturation, utilization, and bottleneck analysis: other configurations Usable capacity was lower for middleware configurations with fewer worker threads (Table 6, including N_{uc} that indicates the start of the saturation phase for these configurations). The middleware introduces a bottleneck in these configurations by preventing the middleware-server subsystem from running with its full usable capacity as explained in detail in 3.1.

Utilization analysis with OL and measured CPU usages still show highest utilization / CPU usage for servers compared to clients and middlewares also in these cases and would not allow detection of this bottleneck per se.

4.2 Summary

Table 6: Maximum throughput for the full system (usable capacity; mean ± SD, $n = 4$)

	WT=8	WT=16	WT=32	WT=64
Throughput (Middleware) [op/s]	6'779 ± 98	8'455 ± 53	10'363 ± 55	11'406 ± 62
Throughput (Derived from MW response time) [op/s] [a]	6'779 ± 98	8'455 ± 53	10'361 ± 55	11'403 ± 62
Throughput (Client) [op/s]	6'766 ± 96	8'413 ± 58	10'302 ± 57	11'323 ± 64
Average time in queue [ms]	3.8 ± 0.05	1.1 ± 0.01	2.0 ± 0.03	0.8 ± 0.01
Average length of queue [jobs] [b]	19.2 ± 0.1	2.1 ± 0.05	9.6 ± 0.3	1.8 ± 0.1
Average time waiting for memcached [ms] [c]	2.3 ± 0.03	3.5 ± 0.01	5.8 ± 0.04	9.9 ± 0.06
N_{uc} used, see Figure 11 (c)	48	48	96	144

[a] `ClientRTTAndProcessingTime` measured in the middleware is used as thinking time Z to estimate the throughput using the IL. As can be seen already in the plots of this report, these estimates match the actual measurements very well for the middleware. Therefore, very similar or identical values are not an error.

[b] in preparation for section 7, this is the sum of both middleware queues; divide by 2 to have average queue length per middleware; queue lengths are available for each instance, of course; they did not differ relevantly

[c] `ServersNettoResponseTime` measured in the middleware is used

Middleware configuration A middleware configuration with not enough worker threads to fully use the usable capacity in the middleware-server subsystem introduces a bottleneck. As always when the best observation is found at the end of a configuration space (here WT=64), it cannot be excluded that more workers (e.g. WT=80, 96, or 128) could not achieve higher usable capacity. Looking at the available network bandwidth limitation (600 Mbit/s from client VMs) and bottleneck CPU usage at servers (54.7 %), it seems likely that such an increase could be achieved.

Indeed, an additional experiment (`e820`, see appendix) with WT=32 and 64 as comparison and additional WT=96 and 128 (i.e. max. 256 workers in total) confirmed this hypothesis. A maximum usable capacity could be achieved there with 192 workers (WT=96): 13'816 ± 149 op/s ($N_{uc} = 288$) with a only a bit higher usable capacity for 128 workers (WT=64; 12'068 ± 531 op/s) and closer matching usable capacity for 64 workers. Based on request replication to all 3 servers, maximum usable capacity of the configuration in 3.2 cannot be reached, of course.

Average queue length Due to quite fine-grained experiment settings with multiple different client numbers, N_{uc} could be picked with short queues associated with short queueing times (Table 6). More than N_{uc} clients lead to an increase of the queue length (approx. by this amount of jobs), with longer queueing time but unchanged service time.

Fewer jobs lead to a queue that is empty during long periods of time leading to more workers waiting to dequeue a job. As an example, in the configuration with 128 workers (WT=64) and 6 clients in average 126.3 of these 128 workers are waiting to dequeue a new job. With 96 clients (measurement below N_{uc}) still average 59.9 are waiting for jobs at any time; with 144 clients (N_{uc}) 19.1; with 192 clients 3.9 workers waiting with increase of average queue length to 32 jobs.

This mix of both, average queue length > 0 and average waiting workers > 0, in the same setting is not puzzling but reflects that the averages show an incomplete picture: There are times with empty queue and waiting workers and there are times with all workers busy and growing queue. Additionally, 32 jobs in the queue are only about 1/4 of the workers available in this WT=64 setting.

stable system As an observation, the experiments run very stable over all 4 repetitions. There is only very little variance in the measurements (Figure 11 and Table 6). This is in part luck (stable deployment in the cloud) but also part very well controlled and tested experiment setup with synchronization helpers, long warmup and cool-down phases to have very stable steady-state windows, and data parsing matching memtier windows as closely as possible with the middleware stable windows. The interactive law holds (Figure 11 (a) and (b)).

Response time percentiles (Figure 11 (d)) illustrate that the typically shown mean response time is a simplification (long tail). With the derived $N_{uc} = 144$ for WT=64, median response time (shown here for measurements at the middleware) is 9.8 ms, 95 percentile is 18.3 ms, but few requests take longer (99 percentile 24.3 ms; max 235.2 ms).

How does the system react to dynamic changes? While the description of the stable steady-state situation has its merits, it has been very insightful for me to observe how the system reacts to short-term dynamic changes.

First, it has been puzzling to see a quick large increase in middleware `ResponseTime` that was much larger than the relatively small change in `ServersNettoResponseTime` that initiated this dynamic event by leading to a similar small change of the middleware `ServiceTime`. Then I realized how the subsystems are coupled with each other. The associated reduction in throughput in this middleware-server subsystem leads to an immediate reduction of the service rate of the middleware, which leads to an immediate increase of the queue length, which then leads to the observed larger increase in the middleware response time due to longer `QueueingTime`.

Steady state is then achieved in this example by `ServersNettoResponseTime`, middleware `ServiceTime`, and `ResponseTime` returning to the former values of the steady state. However, if the change reflected a longer-lasting effect (e.g. network problem in the cloud), a new steady state would be established due to self-regulatory effect of the closed system (see section 1.7 for detailed explanation).[19] Plots are in the appendix.

Key take-away messages Replication comes with the cost of reduced usable capacity of the system in write-only workloads. It is crucial for system performance to configure the middleware

[19] Such examples have not only been observed in this experiment, of course. Thus, the explanation applies also to such observed cases in the other sections of the report, even if I have not mentioned them explicitly. Please note: such not too frequent examples do not reduce the quality of the overall very stable measurements (see also small SD) that was achieved due to the mentioned strict control of the experiments.

that it does not introduce artificial bottlenecks (see also subsection 3.1 for details).

5 Gets and Multi-gets (90 pts)

The middleware offers two modes to handle get requests with multiple keys (multi-gets): sharded and non-sharded (both explained in subsection 1.3). The workload definition with `--ratio` generates a mixed workload of 50 % set requests and 50 % get requests with the defined number of keys. The system is configured with a low number of clients on purpose (see configuration table) because usable capacity, and therefore saturation, is already reached there for the known bandwidth bottleneck for get operations (Table 1, results sections 2 and 3) even if they consist of only 50 % of the requests.

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
Total number of clients	**12**
Workload	mixed, ratio=1:<Multi-Get size>
Multi-Get behavior	subsection 5.1: **sharded** subsection 5.2: **non-sharded**
Multi-Get size	1, 3, 6, 9
Number of middlewares	2
Worker threads per middleware	64
Repetitions	4 repetitions; 60 s steady-state each

5.1 Sharded Case

5.1.1 Explanation

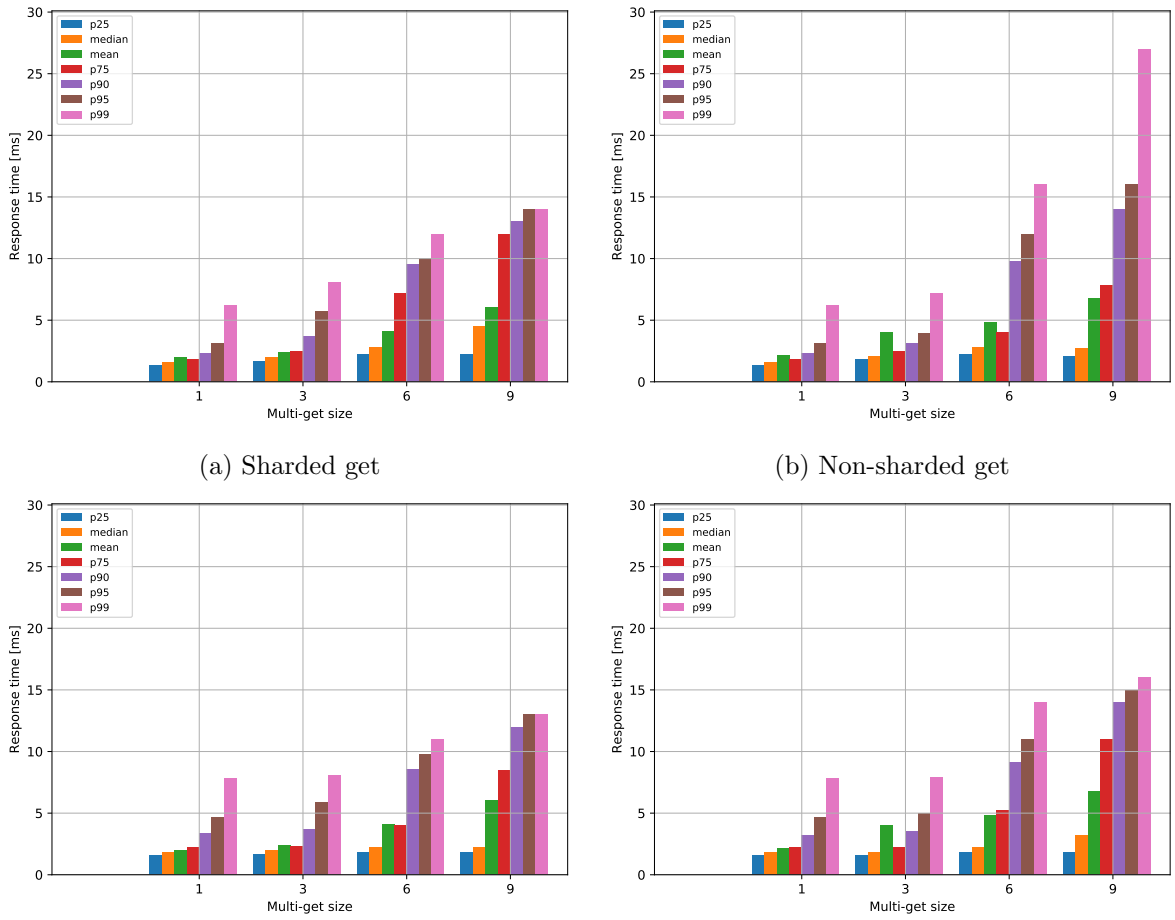
Average response time and key percentiles, measured at memtier, are shown in Figure 12 (a) and (c). Get miss rate was 0.0 %. IL was checked and holds. Thus, with increasing average response time, average throughput decreases accordingly (plot not shown). Due to the fixed ratio of set and get requests, throughput of both request types is of course equal (confirmed with detailed middleware data) and limited by the bottleneck of the get requests (send bandwidth limitation of server Basic A1 VM). Both, middleware instances and server CPU, have a low utilization.

sharded get The increase of the response time with increasing number of keys from 3 to 9 in each get request (Figure 12 (a)) is caused by symmetrically more processing overhead in the middleware and servers, and more network bandwidth usage. In contrast, there is a conceptual difference for get requests with only 1 key: only one server is accessed in this case which has the advantage that average RTT of each server (all servers are accessed in round-robin) is always shorter than always maximum RTT of all servers.

In addition to increasing average, also variance of the response time increases: wider inter-quartile range (IQR), longer tail (see 99 % percentiles) indicating that the system is getting closer to its *nominal capacity*.[20] Thus, longer tails here.

set (sharded get mode) The distribution of response times for set requests associated with the multi-get sizes (Figure 12 (c)) is almost identical to the distribution of these get requests. This is no surprise: both request types share the same queue in the middleware and are handled by the same workers.

[20] when utilization of a system goes towards 1.0, response time goes towards infinity [7]



(a) Sharded get

(b) Non-sharded get

(c) Set (sharded mode for get)

(d) Set (non-sharded for get)

Figure 12: Response time as measured on the client (mean and percentiles); sharded and non-sharded are next to each other on purpose for easier visual comparison.

5.2 Non-sharded Case

5.2.1 Explanation

Average response time and key percentiles, measured at memtier, are shown in Figure 12 (b) and (d). Get miss rate was 0.0 %. IL was checked and holds. Thus, with increasing average response time, average throughput decreases accordingly (plot not shown). Explanations about throughput coupling of set and get requests, bottleneck and utilization are the same as in 5.1.

non-sharded get The increase of average and median response time with increasing number of keys from 1 to 9 in each get request (Figure 12 (b)) is caused by a similar explanation as in 5.1. Differences: each server handles only 1/3 of the requests, but each reply is 3x larger than in the sharded case.[21]

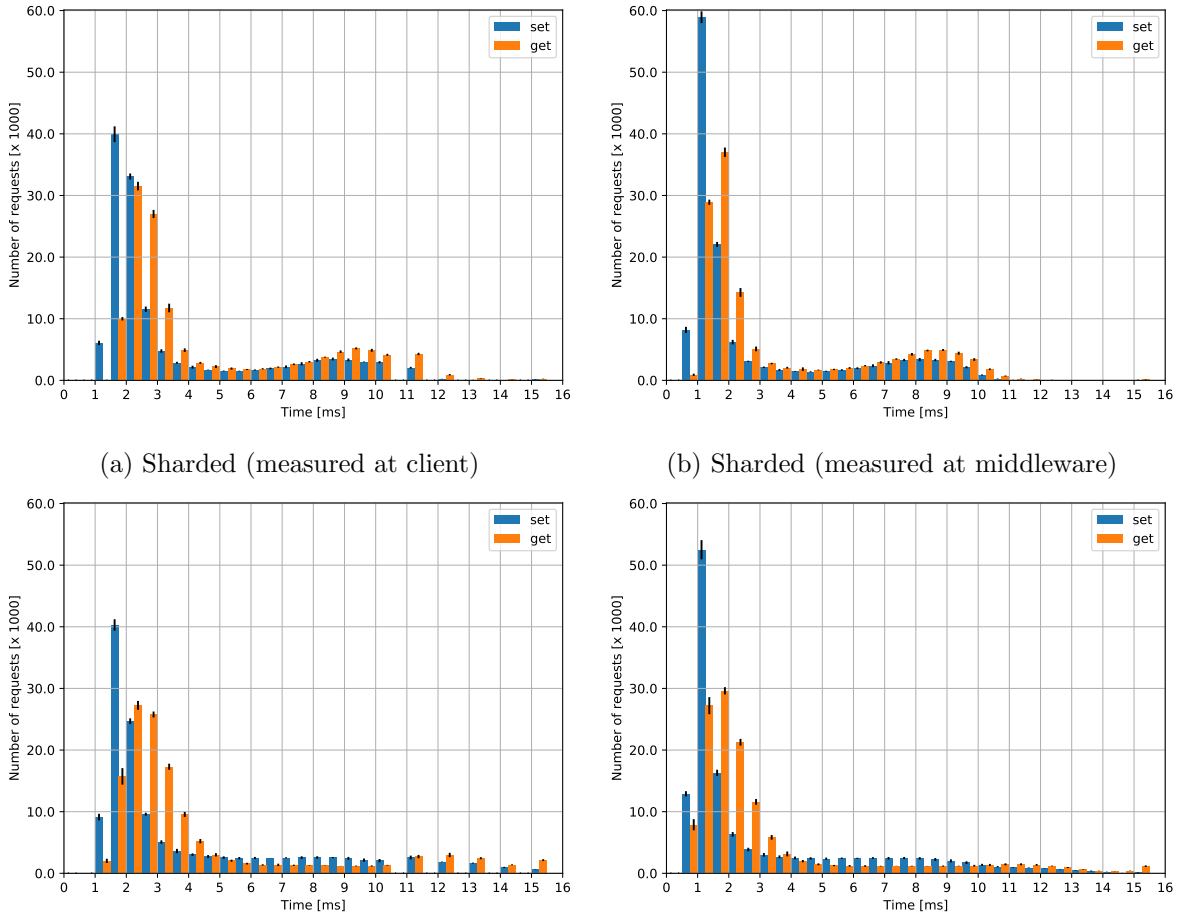
This explains also the observed data. Median response times are quite similar between sharded and non-sharded gets. However, the long tail of response times becomes much longer in

[21] again exception: requests with 1 key look identical for the servers despite having been processed in different code paths in the middleware (sharded vs non-sharded middleware worker functions), which matches the response time distribution (Figure 12).

non-sharded get because the round-robin system in the middleware has a conceptual limitation in load-balancing the requests to the servers (see also explanations in 5.4). This is in particular pronounced for 9 keys/request, where utilization of the bottleneck device "send bandwidth limitation of server VM" becomes the highest.

set (non-sharded get mode) Except 99 % percentile, the distribution of the set operations (Figure 12 (d)) is almost identical to the get requests with the same explanation as in 5.1. The 99 % percentile is lower for the set requests because the payload (needs much bandwidth) is not limited by this asymmetric bottleneck of the Azure VM policy.

5.3 Histogram



(a) Sharded (measured at client)

(b) Sharded (measured at middleware)

(c) Non-sharded (measured at client)

(d) Non-sharded (measured at middleware)

Figure 13: Response time histograms for mixed workload with 6 keys/get request (bins 0.5 ms; cutoff 15 ms; counts ± SD); the last bucket in each histogram includes also all response times > 15 ms; memtier resolution is limited for response times ≥ 10 ms; set and get bins next to each other for easer comparison; sharded and non-sharded vertically aligned to allow easier comparison; shorter response times for middleware than clients is self-evident

The response time histograms measured at the client and the middleware (Figure 13; layout chosen specifically for easier visual comparisons) illustrates the explained aspects (5.1 and 5.2) well. Response times do not follow a normal distribution. After an early peak (earlier for set

than get), there is a long tail that is longer for non-sharded requests than sharded requests (already seen in the percentile distributions). Of course, response time at middleware is lower than measured at the clients.

Interestingly, there is a second smaller peak for sharded gets at around 9 ms (and sets a bit earlier) at memtier (and a bit earlier at the middleware). This second peak is caused by the mentioned cyclic load on the servers caused by conceptual weakness of the round-robin balancing (see 5.4).

5.4 Summary

This experiment revealed load-balancing limitations of the simple round-robin system. It guarantees that all servers are used for the same number of requests (verified with instrumentation data) but does not give any guarantee with respect of timing. Despite starting with a best effort approach (workers start with different servers as first server, see section 1.3), there is no synchronization among the workers that may adjust the request distribution based on effective load on the middleware - server subsystems, whose utilization dynamically changes. Additionally, there is no load balancing for static differences between different middleware - server connections (observable with the `ping` RTT) based on specific deployment of the system in the Azure cloud.

These limitations have a larger effect (in particular long response time tails) in the setting with many keys/request and in particular with non-sharded gets (see detailed explanations in 5.1 to 5.3). Based on the percentile distributions (Figure 12), sharded mode should be used for 6 or more keys/request due to relevantly lower 95 and 99 % percentiles. Non-sharded mode has a bit shorter response time for 3 keys/request, which is caused by the fact that average RTT for one server is shorter than average $max(RTT)$ of 3 servers. Mode does not matter for 1 key/request because server and network usage is identical in both modes. Please note: in comparison with the waiting times for the servers, the small differences in computing overhead in the middleware for sharded get code path is negligible.

The observed behavior is caused by the tight network send bandwidth bottleneck for server VMs. Advantages and disadvantages of sharded/non-sharded get functionality of the middleware for specific request sizes would have to be reevaluated in a system without this bottleneck (if e.g. the middleware was the bottleneck, or different payload sizes).

Queue length It is an interesting observation for both experiment settings that the middleware queues of the 2 instances are empty. Thus, also the typical queueing time of approx. 60 μs that reflects time needed to enqueue and dequeue a job. This observation matches well with the observation that median response time does not increase so much with more keys per request but the long tails do. The prolonged response time at the client reflects directly the delay in the middleware-server subsystem. With a long queue, also the median response time would be increased relevantly.

6 2K Analysis (90 pts)

Number of servers	1 and 3 (factor A)
Number of client machines	3
Instances of memtier per machine	1 (1 middleware) or 2 (2 middlewares)
Threads per memtier instance	2 (1 middleware) or 1 (2 middlewares)
Virtual clients per thread	32
Total number of clients	**192**
Workload	write-only and read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1 and 2 (factor B)
Worker threads per middleware	8 and 32 (factor C)
Repetitions	4 repetitions; 60 s steady-state each

A $2^k r$ factorial experiment design with 3 factors ($k = 3$) and 4 repetitions ($r = 4$) was used to study the effect of 3 factors (A, B, C) on throughput and response time in the setting of (a) write-only and (b) read-only workloads using 192 virtual clients for each test.[22]

- A: 1 or 3 memcached servers; $x_A = -1$ for 1 server; $x_A = 1$ for 3 servers
- B: 1 or 2 middleware instances; $x_B = -1$ for 1 instance; $x_B = 1$ for 2 instances
- C: 8 or 32 worker threads per middleware; $x_C = -1$ for 8 threads; $x_C = 1$ for 32 threads

Methodology was used as described on pages 283-313 of the book [7]. The full calculation tables are implemented in four Excel worksheets in `data/e610_2kr_analysis/` (read-only/write-only; throughput/response time) solving the associated linear equations. The worksheets allow solving both models, additive and multiplicative $2^3 4$ factorial design models. The detailed calculation tables are not shown here due to space constraints. Percentage attribution and effect – including 95% CI – are reported as main results (Tables 7 and 8).

The *additive* $2^3 4$ model was chosen for the reported values here because the ratio of $\frac{y_{max}}{y_{min}}$ was small: 2.5 for write-only; 3.1 for read-only.[23] And indeed, looking at the results of the multiplicative model calculations, the conclusions remained the same with only small differences in the detail numbers.

How to read the result tables Explanations are given with throughput examples from write-only workload (Table 7). Mean "baseline" throughput was 7'903 op/s with 95 % CI [7'867, 7'938]. The effect corresponds to the q value in the model. Noise was low in the results of all experiments (error ≤ 0.3 % in all models).

The number of worker threads (factor C) had the largest effect (69.5 % attribution). Using 32 threads per middleware instance ($x_C = 1$) *increases* the mean throughput by 1'724 op/s with 95 % CI [1'688, 1'760]. Using 8 threads per middleware instance ($x_C = -1$), however, *decreases* the mean throughput by the same amount. Thus, the actual difference between using 8 or 32 threads is approx. 3'448 op/s.

Negative effects in the table have the inverse result. Using 3 memcached servers ($x_A = 1$) *decreases* the mean throughput by 502 op/s with 95 % CI [466, 538]. Using 1 memcached server ($x_A = -1$) *increases* the mean throughput by this amount. Thus, the actual difference betwen using 1 or 3 memcached servers is approx. 1004 op/s.

[22] Using 192 clients assures all system configurations being tested in their saturated phase because $192 > N_{uc}$ for all configurations (see sections 3-5).

[23] see discussion by Raj Jain: "...the logarithmic transformation is useful only if the ratio $\frac{y_{max}}{y_{min}}$ is large. For a small range the log function is almost linear, and so the analysis with the multiplicative model will produce results similar to that with the additive model." (page 308 [7])

The same interpretations apply also to the response time and the values of the read-only table. Naturally, an increase of throughput is associated with a decrease of the response time. The IL was checked and held for all experiments.

6.1 Write-only workload

Table 7: Write-only workload

	Throughput			Response time		
Factor	% attrib.	effect [op/s]	95% CI	% attrib.	effect [ms]	95% CI
I	—	7'903	[7'867, 7'938]	—	25.9	[25.8, 26.1]
A	5.9 %	−502	[−538, −466]	7.9 %	2.1	[2.0, 2.3]
B	23.2 %	996	[960, 1'032]	24.5 %	−3.8	[−3.9, −3.6]
C	69.5 %	1'724	[1'688, 1'760]	59.5 %	−5.9	[−6.0, −5.7]
AB	1.0 %	206	[170, 242]	2.9 %	−1.3	[−1.5, −1.1]
AC	0.0 %	6	[−29, 42] [a]	1.5 %	−0.9	[−1.1, −0.7]
BC	0.2 %	94	[58, 130]	2.9 %	1.3	[1.1, 1.5]
ABC	0.0 %	29	[−7, 65] [a]	0.6 %	0.6	[0.4, 0.8]
error	0.2 %	—	—	0.3 %	—	—

[a] not significant

Throughput Best configuration: 32 worker threads (C), 2 middlewares (B), 1 memcached server (-A). As mentioned above, the number of worker threads (factor C) had the largest effect (69.5 % attribution). Adding an additional middleware instance (factor B; effectively doubling the number of worker threads) had the second largest effect (23.2 % attribution). This effect is smaller than factor C because it "only" doubles the number of worker threads in contrast to factor C that quadruples this number. Indeed, the effect of factor C (1'724 op/s) is approx. twice as large as the effect of factor B (996 op/s). In the used range of total worker threads (8 to 64), these two factors are practically independent of each other. There is no relevant external bound that limits them (see combined factor BC practically negligible). Using 3 instead of 1 servers (factor A) reduces the the throughput relevantly because of the replication mechanism implemented in the middleware for write operations.

Response time The response time model calculations confirm the data from the throughput model. As expected, an increase of throughput is associated with a decrease of response time. Here the factors are not fully independent of each other: e.g. combining factors B and C, i.e. comparing 8 worker threads from 1 middleware with total 64 worker threads from 2 middleware instances, leads to a small increase of the response time (1.3 ms) compared to the larger decreases (-3.8 ms + -5.9 ms = -9.7 ms) seen from each individual factor. The utilization of the server(s) increases relevantly when the configuration is changed from 8 to 64 worker threads,[24] increasing response time.

Bottleneck analysis and comparison with results of other parts of the report The "price" of replication to 3 instead of 1 memcached servers can be quantified. The results match the observations of earlier sections 3 and 4. The clean factorial experiment design quantifies the effect of these observations.

[24] CPU usage as surrogate increases from 8 to 28 % with one server and from 15 to 48 % with 3 servers in the configuration

For detailed utilization and bottleneck analysis, I refer to section 3 (one server), section 4 (3 servers). In the configurations used in this experiments, not the servers are the actual main bottleneck but the number of worker threads that lead to the situation that the middleware-server subsystem cannot provide its maximum usable capacity.

6.2 Read-only workload

Throughput Best configuration: 3 memcached servers (A), number of middlewares (B) and worker threads (C) do not matter. As seen in earlier sections 2, 3, 5, the send network bandwidth limitation of the servers' A1 VMs is the bottleneck for read-only workload. Not surprisingly, factor A (number of servers and thus associated network connections) is almost the unique factor with an effect in this experiment. Based on the effects of I and A, average throughput differs from 2913 op/s with one server to 8363 op/s in the model, both close to the known bandwidth limits of 100 Mbit/s per server.

Response time The findings above are confirmed by the analysis of the response time. Based on effects of I and A, average response time differs from 23.1 ms for 3 servers to 65.1 ms for one server. Response time are long due to the large number of virtual clients (192). The same throughput could already be achieved with much fewer clients (6 or 12), see results from sections 3 and 5.

Table 8: Read-only workload

	Throughput			Response time		
Factor	% attrib.	effect [op/s]	95% CI	% attrib.	effect [ms]	95% CI
I	—	5'638	[5'616, 5'660]	—	44.1	[43.8, 44.3]
A	97.3 %	2'725	[2702, 2748]	99.4 %	−21.0	[−21.2, −20.8]
B	0.4 %	176	[154, 199]	0.1 %	−0.6	[−0.8, −0.4]
C	0.5 %	189	[166, 211]	0.1 %	−0.5	[−0.7, −0.3]
AB	0.5 %	193	[171, 216]	0.1 %	−0.5	[−0.7, −0.3]
AC	0.5 %	189	[166, 211]	0.1 %	−0.7	[−0.9, −0.5]
BC	0.4 %	−182	[−205, −159]	0.1 %	0.6	[0.3, 0.8]
ABC	0.4 %	−180	[−203, −158]	0.0 %	0.4	[0.2, 0.6]
error	0.0 %	—	—	0.1 %	—	—

Bottleneck analysis and comparison with results of other parts of the report As already seen in the earlier sections 2, 3, 5, the send network bandwidth for the A1 VMs used for the memcached servers is the clear bottleneck for all read-only workloads. This has been confirmed again in these experiments here.

7 Queuing Model (90 pts)

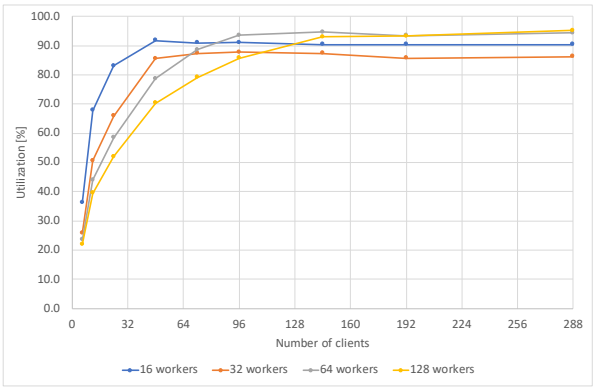
M/M/1, M/M/m and queueing network (QN) models were created with the data from earlier sections. Please note: the models here are not related to the utilization and bottleneck analysis in the other sections that relied only on operational laws (explication in subsection 1.7).

7.1 M/M/1

Using the experimental data from section 4, a separate single-station M/M/1 model was built for each configuration of the middleware (number of worker threads) following chapter 31 of the book [7]. This is a substantial simplification of the real system (Figure 1).

Model input Maximum throughput observed in a 1 s window (X_{max}) was used as service rate μ for each configuration. Average throughput (X) was used as arrival rate λ for each configuration and number of clients tested in the experiment.

Model output Expected value ($\mathbb{E}$) and variance were calculated as described in box 31.1 [7]. Traffic intensity $\rho < 1$ for all measurements, as expected for a closed system.[25] Standard deviations (calculated from variance; not shown in the figures) were very wide, i.e. in the size of the expected values, limiting the predictive "power" of the model.



(a) Utilization

(b) Response time

(c) Number of jobs in system

(d) Queue length

Figure 14: M/M/1 model: expected averages of the experiments in section 4

Utilization Traffic intensity ρ (shown as utilization; Figure 14 (a)) reaches the expected plateau approx. with the number of clients used to determine usable capacity (see section 4), which fits how this N_{uc} has been determined. It is expected that the system does not reach an utilization of 100 %. With increasing utilization, response time increases to infinity. In a closed system, longer response time will reduce the arrival rate of requests (self-regulatory circuit).

[25] and as designed by given selection for input μ and λ

Response time ($\mathbb{E}[r]$) The model (Figure 14 (b)) underestimates the observed response time (Figure 11 (b)) because it assumes only one worker processing all requests. Service rate μ is given as input and not service time. Mapping μ to service time for one worker ($= 1/\mu$) is much less than the actual service time ($= WT/\mu$ for WT workers per instance) leading to the underestimate of the response time.

The plateau with more than N_{uc} clients is an artifact (see Table 6 for N_{uc} of each configuration). Response time is underestimated there additionally because queueing time is not considered for these jobs that the model is not aware of (see $\mathbb{E}[n_q]$ below).

Queue length ($\mathbb{E}[n_q]$) The model prediction (Figure 14 (d)) does not match the measurements of the data where the middleware queue does not start to grow relevantly before N_{uc} has been reached (Table 6). The model assumes that only one job can be processed at a time; all other jobs are queued. This is a significant difference to the effective system, where each of the 16 to 128 workers needs to be working on a job before the middleware queue starts growing relevantly.

Also the prediction of a plateau is wrong. $\mathbb{E}[n_q]$ is calculated using ρ that has a plateau (see utilization above). There is no input to the model that could indicate more jobs in the system than a maximum number of jobs derived from $max(\rho)$.

Number of jobs in the system ($\mathbb{E}[n]$) The model underestimates the true number of jobs ($\approx$ number of clients in a closed system) in the system (Figure 14 (c)). Up to N_{uc} of each configuration, it follows the trend but underestimates the number of jobs because it overestimates the queue length (see above) associated with relatively longer response times (= service time + queueing time; queueing time increasing linearly with queue length), and thus fewer jobs that can be associated with observed arrival rate λ (see interactive law).[26]

The observed plateau with more than N_{uc} is an artifact (see explanation above).

7.2 M/M/m

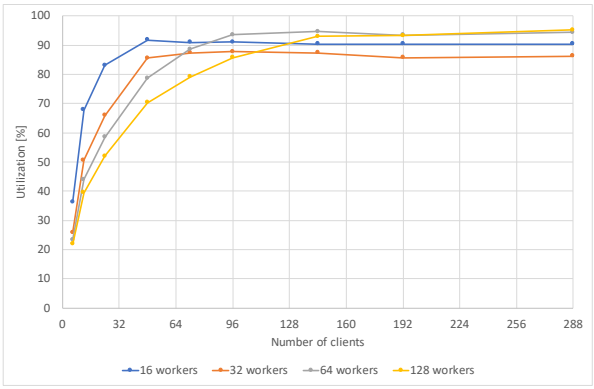
A separate M/M/m model was built for each configuration of the middleware (number of worker threads) with the Octave queueing package [8] using the same experimental data as above. Experimental data from section 4 as described in 7.1.

Model input The number of worker threads of both middleware instances was used as m. Maximum throughput observed in a 1 s window per worker (X_{max}/m) was used as worker service rate μ for each configuration.[27] Average throughput (X) was used as arrival rate λ for each configuration and number of clients tested in the experiment.

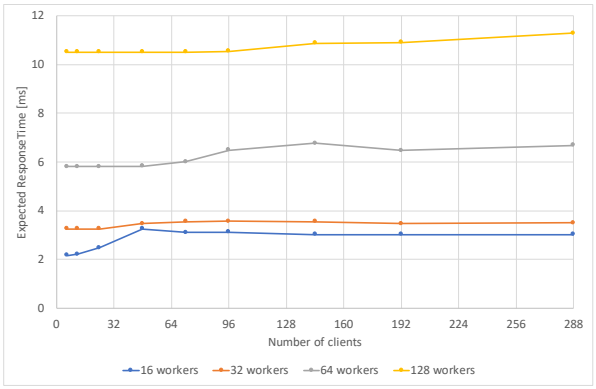
Utilization For the traffic intensity ρ (shown as utilization; Figure 15 (a)) the same discussion applies as in subsection 7.1.

[26] This is *not* in contradiction to the observed overall underestimation of the response times by the model.

[27] This assumes that the worker service rate depends on the number of workers, which makes sense of course and is close to observed data. In an alternative model, it could be assumed that the service rate of a worker does not change (using $max(X_{max}/m)$ of all configurations for modeling all of them). Plots in the `e720_MMm_model_b.xlsx` Excel sheet show that this model is not a good fit for the system under test (SUT) because worker service rate *indeed* depends on the number of workers in the SUT because all of them connect to the same servers that are the bottleneck.

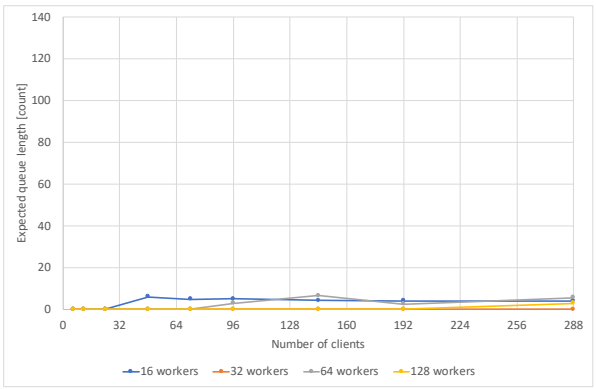


(a) Utilization



(b) Response time

(c) Number of jobs in system (Q)



(d) Queue length, derived from $max(Q - m, 0)$

Figure 15: M/M/m model: expected averages of the experiments in section 4

Response time ($\mathbb{E}[r]$) The model (Figure 15 (b)) gives rather good estimates for few workers up to N_{uc} and overestimates for many workers up to N_{uc}. This overestimation is caused by the fact that μ was derived from configuration specific X_{max} that is influenced by the bottleneck (servers) that is not dependent on the number of workers. Response time is underestimated (model artifact) for client counts $> N_{uc}$ as discussed in subsection 7.1.

Number of jobs in the system ($\mathbb{E}[n]$) The model estimates the number of jobs in the system (Figure 15 (c)) quite well (within factor of 1.2) up to N_{uc}. After reaching the usable capacity of the system, $\mathbb{E}[n]$ lacks the jobs in the queue that are not available to the model that also bases on traffic intensity (see explanation in 7.1).

Queue length ($\mathbb{E}[n_q]$) This value is not available directly from the Octave queueing package and was calculated by $max(\mathbb{E}[n] - m, 0)$. Thus, it reflects well the situation before reaching N_{uc} (practically empty queue) but lacks the queueing in the middleware that relevantly starts after reaching N_{uc} due to explained underestimation of $\mathbb{E}[n]$.

7.3 Network of Queues

A closed system single-class queueing network (QN) model [8] was used for modeling the system of sections 3.1 (3 client, 1 middleware, and 1 server VMs) and 3.2 (3 client, 2 middleware, and 1 server VMs). These models do not allow modeling of components waiting for another component (middleware worker waiting for server). Thus, the system (see Figure 1 but only 1 server; 1 or

2 middlewares) was unrolled (Figure 16). Instead of modeling the network connections twice (each with S value set to half the RTT measured by `ping`) and having the two parts of the worker (processing before sending request to server; processing after receiving reply from server again with S for each being a fraction of measured `ProcessingTime`), a simplified model was deduced for analysis.

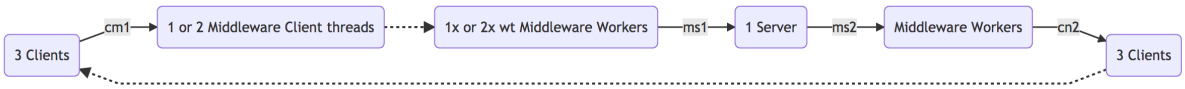


Figure 16: Unrolling of the systems (wt: number of middleware workers per instance)

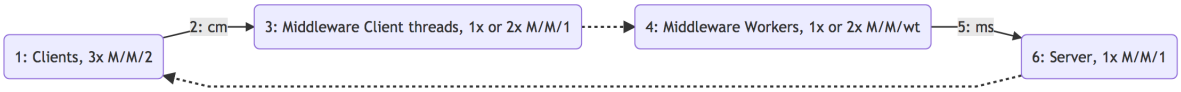


Figure 17: Network of queues: 6 components (wt: number of middleware workers per instance)

Model The base QN model consists of 6 components (Figure 17), which was used in 4 variants to model the system of section 3. Modifications involve: (1) number of middleware instances, modeled using $V_{middleware} = 1$ (one instance, subsection 7.3.1) or $V_{middleware} = 1/2$ (two instances, subsection 7.3.2); (2) write-only workload (bandwidth limitation on `cn` network connection) or read-only workload (bandwidth limitation on `ms`).

The aforementioned simplification can be done because the model is not aware of any order, in which the components are used. The model receives S_i, V_i and m for every component i, of which the component demand D_i can be calculated but no indicator of order.

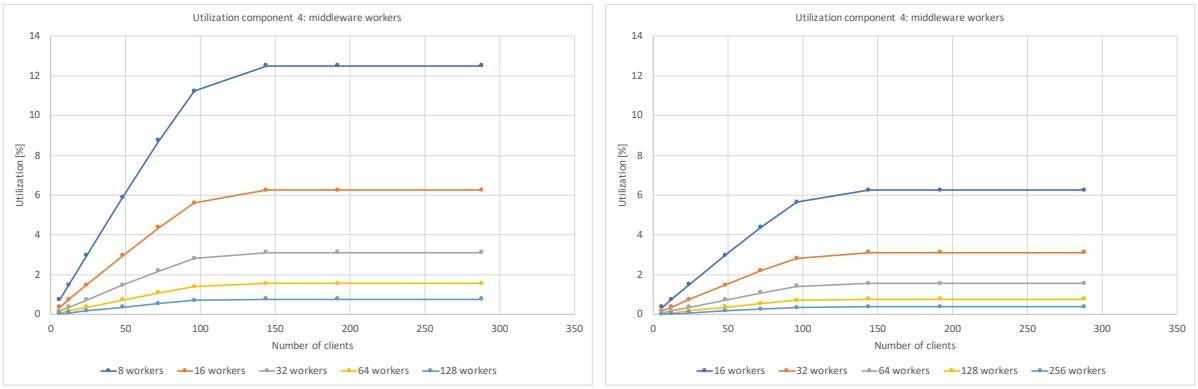
Network connections are typically modelled as delay centers (M/M/∞). Having realized the relevant bottleneck effect caused by the network send bandwidth limitations (Table 1) in particular for the read-only workload (Figures 7 and 9), I tried to accommodate for this by modeling the network connections as M/M/m centers, too. For write-only workload, m of the `cn` component was set to the throughput corresponding to 600 Mbit/s (18 kop/s); for read-only, m of the `ms` component to the limit corresponding to 100 Mbit/s (3 kop/s); the other network component was set to an unreachable limit in each configuration.

Of course, this modeling does not fully match the behavior of these network bandwidth limitations. Additionally, the models were also run with the networks modeled as plain delay centers without relevant differences for the other components, in particular not for utilization.

Model input Average service times S were used from raw data from section 3 (middleware `PreprocessingTime` and `ProcessingTime`, `ping` RTT for network connections), and from section 2 (upper bounds for S_{client} and S_{server}). Please see `scripts/octave/m730_QN.m` for details. Extended mean value analysis (MVA)[28] was applied on all models (4 model types mentioned above for all 5 worker thread settings) and all configurations (number of clients).

Model output Output of this analysis is quite extensive. Thus, only key results can be reported with given space, mainly utilization of each component and estimated throughput. The queue of component 4 (middleware workers) corresponds to the queue in the middleware implementation.

[28] the implementation in the Octave package has some numerical instabilities (negative values in results)



(a) 1 Middleware instance

(b) 2 Middleware instances

Figure 18: QN model: Load and configuration dependent utilization of component 4 (middleware workers)

7.3.1 One middleware

Write-only workload The model predicts a system throughput from 1 to 16 kop/s that matches quite well with measured throughput in this middleware configuration (some overestimation). Relative throughput of components is correct.

The model predicts utilization for all components that match quite well the calculated utilizations using OL in section 3. Only the middleware worker thread (component 4) shows a configuration specific utilization (Figure 18 (a)) that matches with the calculations by OL in section 3. An utilization plateau is reached. Utilization decreases with more worker threads.

Utilization of the other components increases with increasing number of clients. But there is no relevant difference with different middleware worker thread counts. This is related to the fact that the middleware worker utilization is already low (< 15 %) with 8 workers. Thus, relative change with respect to utilization is not very big for the other components.

Client utilization increases from 1 to 11 %, middleware net-thread (client thread) from 2 to 33 %, and server from 5 to 100 % with 6 to 288 jobs in the system. The server is the clear bottleneck, which matches the calculations based on OL in section 3.

Read-only workload The model predicts a system throughput from 1 to 6 kop/s (reached with 72 clients). Afterwards, the model predictions cannot be used anymore because of the numerical instabilities (negative values). In comparison with the measured data, it takes more clients in the model to reach this plateau.

Utilization looks almost identical to the utilization shown for the write-only workload. In particular, still the server is the bottleneck. The approach in modeling the network connection as an M/M/m device (see above) could not create a strong enough bound like the bandwidth limitation seen in the real system.

7.3.2 Two middlewares

Write-only workload The model predicts also here a system throughput from 1 to 16 kop/s that matches quite well with measured throughput in this middleware configuration (some overestimation). Relative throughput of components is correct, in particular middleware throughput (components 3 and 4) have only half of this system throughput.

Middlware net-thread and worker utilization (components 3 and 4) are only approx. half of the utilization predicted with 1 middleware (see above and Figure 18). This was expected and

also matches the observations made in the utilization values by OL in section 3 when comparing both settings. The other utilization predictions remained approx. the same, mainly again with the bottleneck in the server (100 %).

Read-only workload Throughput prediction matches the prediction for one middleware (see above). Utilization prediction of the model looks almost identical to the write-only prediction with 2 middlewares again with numerical instabilities above 72 clients.

7.3.3 Limitations

The model works quite well, in particular for utilization analysis in write-only workload. Also when trying various changes to the settings, I could not create a model setting that reproduces the true bottleneck for read-only workloads: the bandwidth limitation of component 5 (server-middleware connection). This may be possible when using different modeling algorithms that use complete service time profiles to model true load-dependent services. Such profiles could be created from measured instrumentation data by interpolation of the measured data points. However, I deliberately did not go into this direction that would also become something like a black-box modeling ("just make a model fit the data"). I preferred – also for the purpose of learning this MVA QN modeling system – to explore the parameter space of setting the inputs to the model manually, as explained above and documented in the octave program in detail (see `m730_QN.m`).

Additionally, some needed input parameters for the QN model could not be deduced exactly enough due to limitations of the setting in the cloud (see e.g. explanations in section 2 and associated tables in the appendix for section 2).

This model did not bring so much new insight into the system because my own modeling implemented in my analysis software, that is based on operational laws alone, worked so well.

References

[1] Memcached. https://memcached.org/ (last accessed 2018-11-27)

[2] Memtier benchmark. https://github.com/RedisLabs/memtier_benchmark (last accessed 2018-11-27)

[3] Azure portal. https://portal.azure.com (last accessed 2018-11-27)

[4] Azure VM documentation. https://docs.microsoft.com/en-us/azure/virtual-machines/linux/sizes (version 2018-11-14, last accessed 2018-11-27)

[5] Azure Github documentation. https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/azure-stack/user/azure-stack-vm-sizes.md (version 2018-10-23, last accessed 2018-11-27)

[6] Hoefler T. and Belli R. Scientific Benchmarking of Parallel Computing Systems. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15) 73:1-12, 2015

[7] Jain R. The Art of Computer Systems Performance Analysis. 1st ed. Wiley Professional Computing, 1991

[8] Marzola M. The qnetworks Toolbox: A software package for queueing networks analysis. doi:10.1007/978-3-642-13568-2_8 (version 1.2.6 from https://www.moreno.marzolla.name/software/queueing/)