Advanced Systems Lab Report

Autumn Semester 2019

Name: Filip Meier - fimeier@student.ethz.ch
Legi: 11-933-405

December 16, 2019

1 System Overview

1.1 System Description

An overview of the middleware implementation is given in the complete System Overview: focus main thread and complete System Overview: focus Worker Thread illustrations. The central components are visualized in the second half of the illustrations, namely the **run()-method** and the **worker thread(s)**. As long as the middleware is running, the **run()-method** iterates through the **connections-queue** and checks if a client is ready to send data. If this is the case, the data structure containing all states for a client (**QueueMetaData**, compare Middleware Data-Structures) is added to the **request-queue**. Otherwise the object gets stored back to the connections-queue. Further, the method is used to load balance requests and to start the instrumentation, once the corresponding flag has been set. The **worker threads** do all the communication/processing needed to handle a client request. This includes the parsing and forwarding of the client's command to a memcached server and the creation of a response in dependency of memcached's answer. Those worker threads pick the data structure from the request-Queue and, once the request has been handled, add the object back to the connections-queue. The number of worker threads is always equal to the specified parameter *numThreadsPTP*. The **run()-method** exists only once (this is the main thread of the middleware and **does "the net-thread's job"**). The system uses **three helper-threads**. One of them implements a **shutdown-Hook**. Another one is there to manage the time during which the **instrumentation** is active (this thread just sets a flag to start/stop measuring). The last one is used to **accept clients** in the first place. This thread creates the QueueMetaData objects and adds them to the connections-queue. This is the only task for this thread. There is no other queue in the system, as the **two queues** mentioned above (connection-queue, request-queue). Both of them contain QueueMetaData objects and are an instance of Java's **LinkedBlockingQueue**. The processing of an object picked from the connection-queue is really fast, as there is no real work to do. The most **important task of the run()-method** is to verify, that the client is ready to send data. Once this has been verified, the object is placed into the request-queue and can safely be read by a worker thread in a blocking manner. There is no benefit in using Java's NIO package (advanced asynchronous networking) in the given setting, as all *the clients are constantly communicating and the worker threads have to process requests one after another (per definition).* Therefore Java's **classic I/O interfaces** have been used. This results in a much simpler and cleaner application. There is no need for other synchronization constructs than the blocking requests-queue. The data flow is trivial, as all the needed meta data is stored in one object and the complete request is parsed inside of one worker thread (no need to transfer commands or data between different threads). **Instrumentation** is done in two places: The **run()-method** keeps track of the clients Inter-Arrival-Time (number of requests, cumulated inter arrival time) and stores the current (system-)time into the QueueMetaData object. All the other statistics are collected directly by the **worker threads**. There is no need for coordination between them, as they do not share any data structures. Everything is stored (per thread) in the **MyMiddlewareStatistics** class. For more details compare Middleware Data-Structures. Before the middleware shuts down, an aggregation step collects all measurements and creates an output like the one shown in finalStats* file.

Listing 1: finalStats* file

```
BEGIN: FINAL STATS
nWindows=82
nGoodWindows=79
nWindowsUsedForAggregation=60 Assert=60 (using windows [15, 74])
runTimeOverall=81500.16052599999 ms (wrong value if... interrupt?)
runTimeWindowCumulatedOverRun=60000000000 (60000.0ms)
```

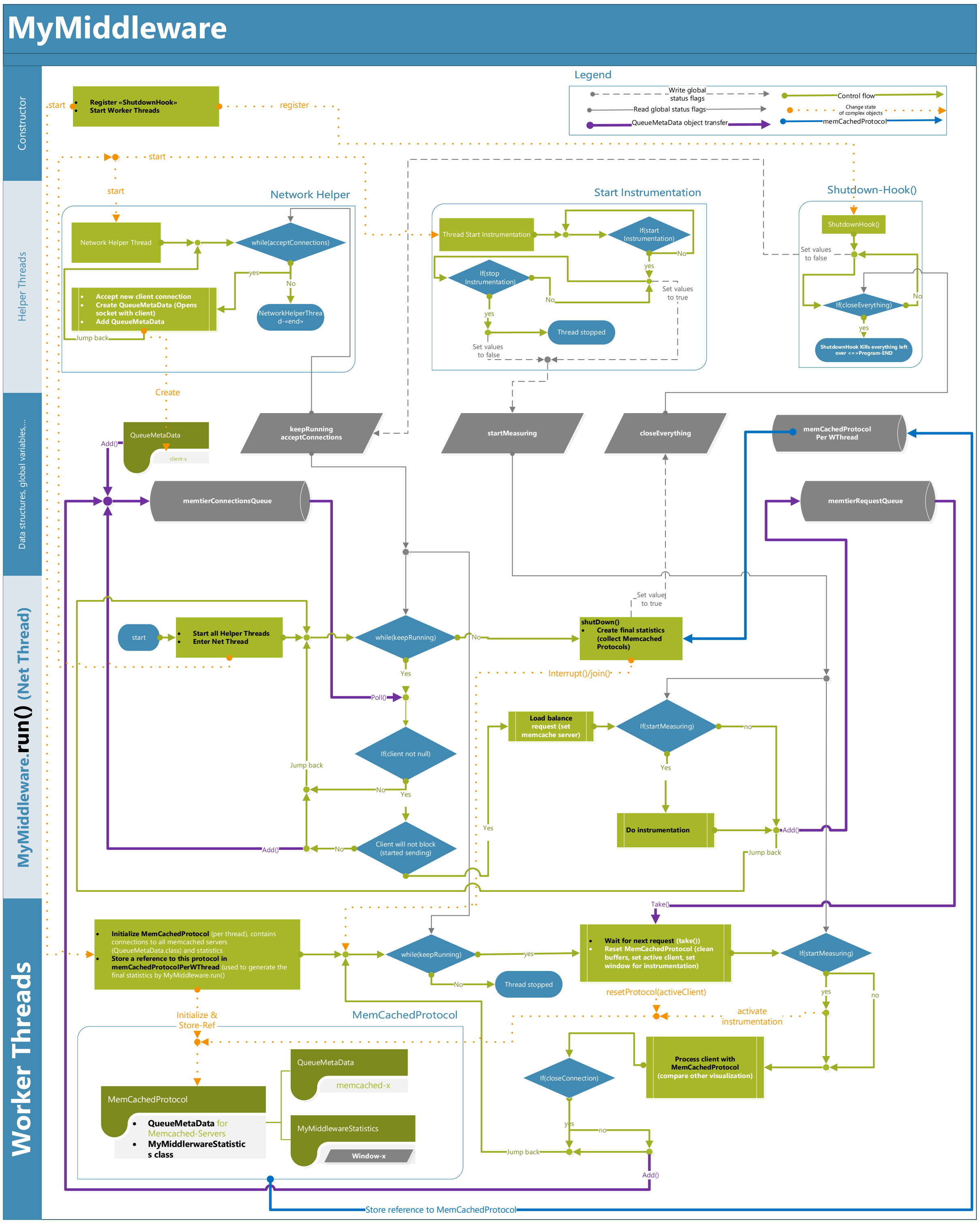

MyMiddleware
Legend
Write global status flags
Read global status flags
QueueMetaData object transfer
Control flow
Change state of complex objects
memCachedProtocol
Constructor
Helper Threads
Data structures, global variables,...
MyMiddleware.run() (Net Thread)
Worker Threads
start
Register «ShutdownHook»
Start Worker Threads
register
start
start
Network Helper
Network Helper Thread
while(acceptConnections)
yes
No
Accept new client connection
Create QueueMetaData (Opens socket with client)
Add QueueMetaData
NetworkHelperThread-«end»
Jump back
Create
Start Instrumentation
Thread Start Instrumentation
If(start Instrumentation)
No
yes
If(stop Instrumentation)
No
Set values to true
yes
Thread stopped
Set values to false
Shutdown-Hook()
ShutdownHook()
Set values to false
If(closeEverything)
No
yes
ShutdownHook Kills everything left over <=>Program-END
QueueMetaData
Add()
client-x
keepRunning acceptConnections
startMeasuring
closeEverything
memCachedProtocol Per WThread
memtierConnectionsQueue
memtierRequestQueue
Set value to true
start
Start all Helper Threads
Enter Net Thread
while(keepRunning)
No
shutDown()
Create final statistics (collect Memcached Protocols)
Interrupt()/join()
Yes
Poll()
If(client not null)
Jump back
No
Yes
Client will not block (started sending)
Add()
No
Yes
Load balance request (set memcache server)
If(startMeasuring)
no
Yes
Do instrumentation
Add()
Jump back
Take()
Initialize MemCachedProtocol (per thread), contains connections to all memcached servers (QueueMetaData.class) and statistics
Store a reference to this protocol in memCachedProtocolPerWThread (used to generate the final statistics by MyMiddleware.run()
while(keepRunning)
yes
No
Thread stopped
Wait for next request (take())
Reset MemCachedProtocol (clean buffers, set active client, set window for instrumentation)
If(startMeasuring)
resetProtocol(activeClient)
yes
no
activate instrumentation
Initialize & Store-Ref
MemCachedProtocol
QueueMetaData
memcached-x
MemCachedProtocol
QueueMetaData for Memcached-Servers
MyMiddlerwareStatistics class
MyMiddlewareStatistics
Window-x
Process client with MemCachedProtocol (compare other visualization)
If(closeConnection)
yes
no
Jump back
Add()
Store reference to MemCachedProtocol

Figure 1: complete System Overview: focus main thread

```
cacheMissRatioFinal=0.0 Assert=0.0
keysSentToServerFinal=251204.0
keysReturnedFromServerFinal=251204.0
keysNotReturnedFinal=0.0 Assert=0.0
numberGetCommands=251204.0
throughputGet=4186.733333333334
throughputAllCommands=4186.733333333334
throughputGetAllKeys=4186.733333333334
avgQueueWaitingTime=0.1205128886920590З
avgQueueLength=2.3063127975669175
avgMemcachedResponseTimeGet=0.6058466925884938
avgWorkerThreadResponseTimeGet=0.6285168977086352
avgMiddlewareResponseTimeGet=0.7490297864006942
nCommandsAddedToQueueOverRun=251204
nCommandsProcessedByWorkersOverRun=251204
Total-set-get-multiget=0 Assert=0
ClientThinkTime_/10.0.0.13=2.04868 ms
ClientThinkTime_/10.0.0.12=2.0058599999999998 ms
ClientThinkTime_/10.0.0.11=2.3181469999999997 ms
ClientThinkTime_AllClients=2.117664 ms
InterArrivalTime=0.24204599999999998 ms
ArrivalRate=4.131446088759988 1/ms and not per sec!!!
WorkBalance server1=33.333536227208675% (109527 out of 328579.0 for all windows!)
WorkBalance server2=33.33323188639566% (109526 out of 328579.0 for all windows!)
WorkBalance server3=33.33323188639566% (109526 out of 328579.0 for all windows!)
END: FINAL STATS
```

1.2 Middleware Data-Structures

The section System Description introduced the **connection-queue** and the **request-queue**. This section outlines some of the used data structures. Compare the illustrations complete System Overview: focus main thread and complete System Overview: focus Worker Thread. Both queues are instances of Java's LinkedBlockingQueue and contain instances of the **QueueMetaData class**. For each new client connection, an instance of this class is created. This object contains a Java **Socket**, a **PrintWriter** and a **BufferedReader**. This is everything that is needed to communicate with a client, respective to exchange messages. To optimize the performance, some space is pre-reserved for the BufferedReader. This class also serves as a communication channel between the run()-method and the worker threads. An important field is the *preferredMemcachedServer field*, which contains the server to be used (set by the **load balancer**). And there are two **timestamps**: The first indicates the time the client has been added to the queue and the second one contains the time when the last response to the client has been sent. This is used to calculate the **queue waiting time** and the **Inter-Arrival-Time**. The QueueMetaData class is further used to open connections between each pair of worker thread and memcached server. While the QueueMetaData objects representing clients are stored and forwarded in the queues, the QueueMetaData objects representing memcached server's are encapsulated together with the MyMiddlewareStatistics in a class called **MemCachedProtocol**. There exists an instance for each worker thread. Compare 1 Worker Threads, MemCachedProtocol and 2 data structures. Most of the statistics are stored in a **MyMiddlewareStatistics** object (except for the ones mentioned above). The used data structures are of a primitive type (long, double, int,...). Some of the numbers are a simple counter (e.g. *numberOfGets++*), others are a combination of a counter (e.g. *numberOfGets++*) and a cumulative value (like *getResponseTime += timeForThisGet*). In the final aggregation (during the shutdown of the middleware) the counters of different worker threads can simply be summed. If there is a counter and a cumulative value, the sum of all counters respective cumulative values can be used to calculate an average (e.g. average response time). As a result of this design, the instrumentation is lightweight and there is no need to store data onto the disk during runtime. Further, triggered by the run()-method, the statistics are stored in different MyMiddlewareStatistics objects. The number of **windows** is pre-known, and the corresponding MyMiddlewareStatistics objects are already present in each worker thread, allowing a fast switch to the next window. The different windows are used to create statistics containing a shorter interval and to skip some windows at the beginning and at the end (cold start and shut down phase for instrumentation). The **network thread** has already been described in the section System Description. The needed functionality is implemented in the run()-method. As pointed out, there is one helper thread that creates the QueueMetaData objects and therefore accepts client connections (creates sockets). The main task of the network thread (run()-method) is to iterate through the connection-queue and forward QueueMetaData objects into the request-queue once a client is ready to send data. The network thread is not aware of the type of data the client is sending.

1.3 Middleware Request Handling

The sections System Description and Middleware Data-Structures outline how a client request is forwarded to a worker thread (connections-queue to requests-queue), how the response is sent back to the client and how the client object gets prepared for the next request (pushed back into connections-queue). The missing details, namely the internals of the worker threads, will be described in the next two subsections.

1.3.1 Request Parsing

The network thread (run()-method) is not aware of the type of data the client is sending. All the parsing is done inside of a worker thread. The **MemCachedProtocol class** has a method parseInput() which does all the work (compare 1 and 2). Given a QueueMetaData object (client Socket, PrintWriter, BufferedReader), the worker thread reads-in a complete line of the client's input and uses regular expressions to decide what to do next. In the given setting (are requests are of type get), there is not much to do, as we only have get requests. Once decided what the type of the request is, the corresponding state machine (control flow) will be entered.

1.3.2 Requests Processing

After the Request Parsing, the parseInput()-method has entered the corresponding state machine to handle the specific command. The communication with the memcached servers is organized in the same way, as the communication with the clients. Figure 2 illustrates how requests are processed. Each worker thread holds an open connection with each memcached server. Those connections are wrapped in an QueueMetaData object (like the client connections) and are stored in the MemCachedProtocol. The state machine simply forwards a command from a client to the corresponding server. The target server is marked inside of the client's QueueMetaData object (*preferredMemcachedServer*). This (marking) has been done by the network thread, while adding the client to the request-queue. The response from a memcached server will be checked and then forwarded to the client. For performance reasons, only the commands in server responses (or client requests) will be parsed and stored in internal buffers. The data part of a command is directly forwarded from one socket to the other (BufferedReader to PrintWriter).

1.4 Work Balancing

Work balancing is done inside of the run()-method, by setting the ***preferredMemcachedServer field***. Please compare all the sections above. The load balancing is optimal. The maximum difference between requests sent to all memcached servers is one. The Work Balance listing shows a collection of finalStats* files for baseline 34. All the numbers have the same **suffix 33.333**. This has been verified for all experiments.

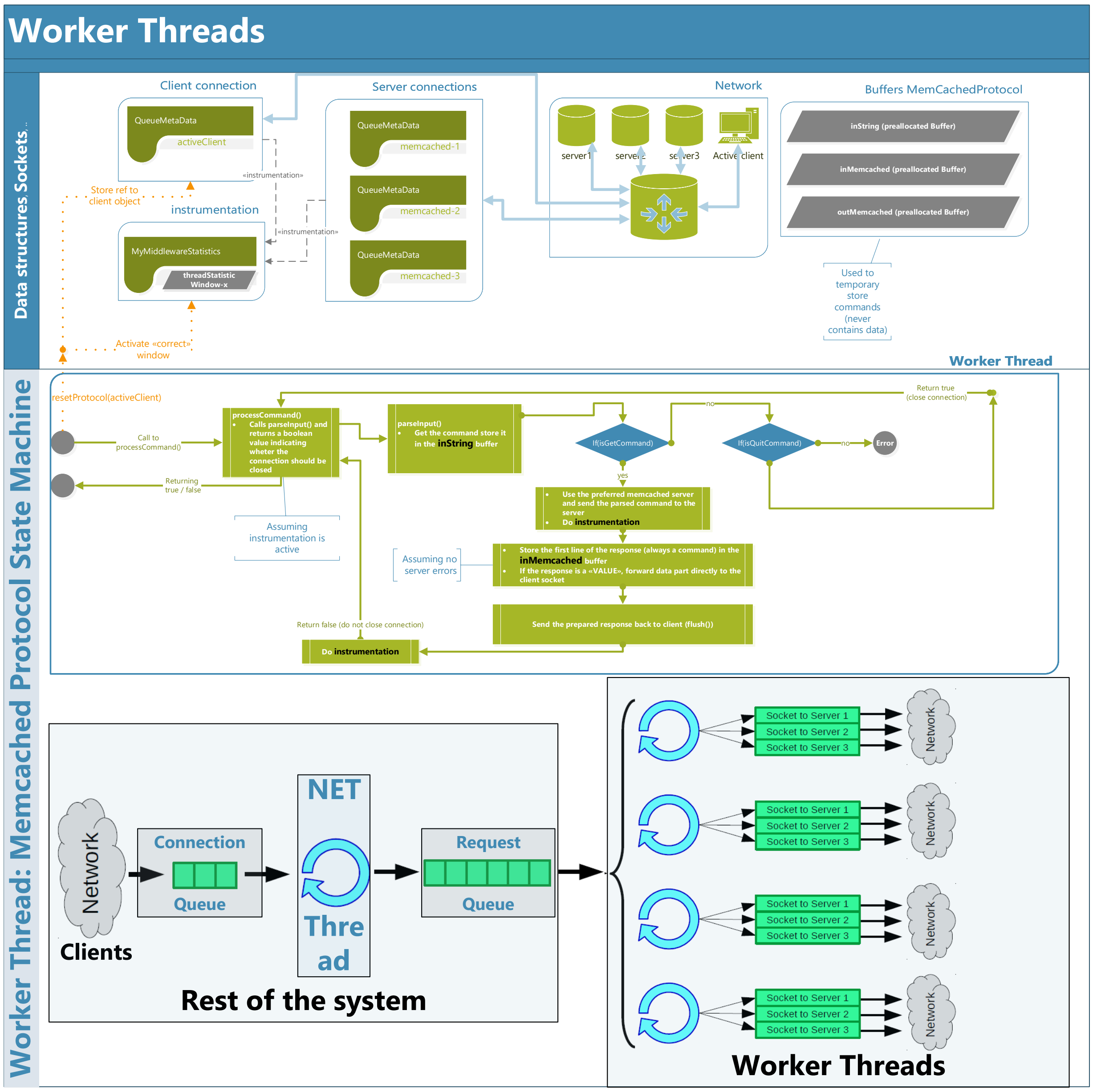

Worker Threads
Data structures, Sockets, ..
Client connection
QueueMetaData
activeClient
Server connections
QueueMetaData
memcached-1
QueueMetaData
memcached-2
QueueMetaData
memcached-3
«instrumentation»
«instrumentation»
Store ref to client object
instrumentation
MyMiddlewareStatistics
threadStatistic Window-x
Activate «correct» window
Network
server1
server2
server3
Active client
Buffers MemCachedProtocol
inString (preallocated Buffer)
inMemcached (preallocated Buffer)
outMemcached (preallocated Buffer)
Used to temporary store commands (never contains data)
Worker Thread
Worker Thread: Memcached Protocol State Machine
resetProtocol(activeClient)
Call to processCommand()
processCommand()
Calls parseInput() and returns a boolean value indicating wheter the connection should be closed
Returning true / false
parseInput()
Get the command store it in the inString buffer
If(isGetCommand)
If(isQuitCommand)
no
no
Error
yes
Return true (close connection)
Use the preferred memcached server and send the parsed command to the server
Do instrumentation
Assuming instrumentation is active
Assuming no server errors
Store the first line of the response (always a command) in the inMemcached buffer
If the response is a «VALUE», forward data part directly to the client socket
Send the prepared response back to client (flush())
Return false (do not close connection)
Do instrumentation
Network
Clients
Connection
Queue
NET Thread
Request
Queue
Rest of the system
Socket to Server 1
Socket to Server 2
Socket to Server 3
Network
Worker Threads

Figure 2: complete System Overview: focus Worker Thread

Listing 2: Work Balance

```
%get the list
cat `ls | grep final` | grep WorkBalance
WorkBalance server1=33.33329499315628% (289803 out of 869410.0 for all windows!)
WorkBalance server2=33.33341001368744% (289804 out of 869410.0 for all windows!)
WorkBalance server3=33.33329499315628% (289803 out of 869410.0 for all windows!)
[..]
%get the numbers
cat `ls | grep final` | grep WorkBalance | cut -f2 -d= | cut -f1 -d%
33.33329499315628
33.33341001368744
33.33329499315628
[..]
```

This is the expected behaviour, as the run()-method assigns a server modulo 3 to a client connection that is ready to send. If the client is not ready, the object is stored back to the connections-queue and the server modulo 3 count doesn't change. A worker thread uses the assigned server (***preferredMemcachedServer field***) and therefore does not change the work balancing. To conclude: The run()-method distributes the load evenly under all the servers (**round robin load balancing**). Experimental results are provided by the worker threads, as they are measure the distribution. The aggregated distribution is calculated and stored while generating the finalStats* file.

1.5 Data Processing

I created the java program **ASLJobControlling** to control all the experiments. In this section an overview of the **DataProcessing** functionality is given. In the Experimental Setup the program's job controlling functionality will be outlined. The **DataProcessing** class aggregates different source files and calculates statistics. The plots are created with gnuplot. The program generates the needed gnuplot-config files and calls gnuplot. The program itself uses config files as shown in the listing plot configuration: this listing shows a configuration to create all plots for baseline 31 (value size plots). Everything that is needed to create the plots is defined in it. **Hint:** the *filename blueprint* comment (first line in the listing) shows how the data files are named, the ones we will be using for this specific plot. The config defines the needed filters to select all files for a specific plot (parameter space). To create plots out of middleware data, this are files like the finalStats* file. The second part of the config file defines what graphs should be created out of the source files. There are always multiple options, as different values are collected and stored in each file (throughput, latency, queue length, ...). Here two **middlewareDataKeys** are defined (throughput, response time). To highlight one option, compare the **aggregationType** field. The throughput is aggregated as a sum while the latency will be aggregated as an average value. This means that the program sums up throughput values (e.g. if there are two middlewares) or calculates an average in the case of a latency plot. **The plots generated for this report show the average value of three runs together with the standard deviation. The values for each run are calculated as described above.** A run (for a specific experiment) contains the data collected from a middleware (for 60s) or from a memtier-client (collected for 80s). Compare Experimental Setup for details.

Listing 3: plot configuration

```
#filename blueprint finalStats_e=baseline31_nS=1_nC=3_nInst=1_nCT=2_nVC=2_wl=ReadOnly...
..._dataSize=64_nMW=1_nWT=8_rep=1_Middleware=1
mode:=middleware
#Files
experimentFolder:=baseline31/
experimentSubfolder:=valueSize
experimentName:=Baseline31ValueSize_WorkerThreads_8
filterInnerLoopPrefix:=finalStats_e=baseline31
```

```
expectedNumberOfFiles:=1
experimentDetail:=WT=8
defaultxLabel:=Value Size
#Loop-Stuff
xAxisDataPointsInput:=64,128,256,384,512,768,1024
ratioXaxis:=1
xFilterPrefix:=_dataSize=
linesToPlotInput:=2,4,8,12,16,24,32
linesFilterPrefix:=_nVC=
numberOfRepetitions:=1,2,3
repetitionFilterPrefix:=_nWT=8_rep=
##read-only case
middlewareDataKeys:=throughputGet,avgMiddlewareResponseTimeGet,[...]
#optional title for all keys in this file
defaultTitle:=Datasource Middleware: ../baseline31/valueSize (8 Worker Threads)
##throughputGet
throughputGet:=throughputGet
aggregationType:=sum
mainType:=tp
operation:=get
colorsToUse:=defaultColorsThroughputRead
lineTitles:=Num-Clients=12;Num-Clients=24;Num-Clients=48;Num-Clients=72;[...]
##avgMiddlewareResponseTimeGet
avgMiddlewareResponseTimeGet:=avgMiddlewareResponseTimeGet
aggregationType:=avg
mainType:=latency
operation:=get
colorsToUse:=defaultColorsLatencyRead
lineTitles:=Num-Clients=12;Num-Clients=24;Num-Clients=48;Num-Clients=72;[...]
```

1.6 Experimental Setup

My Java program **ASLJobControlling** coordinates everything that is needed to collect data. The **Benchmarks** class implements the needed functionality. To execute commands on (remote) systems Java uses system calls to execute ssh. To copy files scp is used. Each experiment is defined as a control flow: loading the memcached servers with the correct data (size), starting middleware(s) and client(s) with the needed parameters and copy back the generated output files. This process is repeated for all needed configurations (multiple loops that iterate through the parameter space). The control loop is idle while the system is running, waiting for a counter to reach zero before it collects the generated outputs. The start of a middleware (memtier or memcached) is a simple call to a local script (over ssh). The script returns immediately. There is no open connection while the system is under test. Outputs are stored locally on the nodes. After the middlewares have started, the clients begin to generate **load for 80s**. A middleware is collecting measurements for 79s, but uses only **60s** for aggregation in the intervall 15-74s (slow start, cool down). To monitor the overall system performance, **NMON**-instances [2] are started in such a way that they can easily be linked to the corresponding experiment (one NMON file per node for all repetitions of the same configuration). The ASLJobControlling program has been placed onto the node middleware1 (inside of a screen session). The console output (starting, stopping, copying protocols) is stored on the middleware1 node. The progress can be monitored with tail -f. To get a better understanding of the network performance, a series of ping and iperf [1] experiments have been run between the different nodes. This is a one time task. The data has been collected right before the measuring has been started.

1.7 Additional Remarks: Network and Azure

I use this section to give an overview of the measured throughput and latency of the used azure infrastructure. The following data has been collected with iperf and the ping utility.

sender\receiver	client	Middleware	Memcached
client	180 Mbits/sec	180 Mbits/sec	180 Mbits/sec
middleware	710 Mbits/sec	710 Mbits/sec	710 Mbits/sec
memcached	88 Mbits/sec	88 Mbits/sec	88 Mbits/sec

The round trip time between the systems is around 0.9ms (+-0.2ms). From the above table one can conclude the following performance characteristics for the network cards (relevant upper bounds (TX)/lower bounds (RX)):

	TX Mbits/sec	RX Mbits/sec	TX KB/sec	RX KB/sec
client	180	710	22'500	88'700
middleware	710	710	88'700	88'700
memcached	88	710	11'000	88'700

The presented results in this report were all collected with the network characteristics described in the tables above. The infrastructure has never been stopped. Remark: Data has been collected around the first of November and multiple times in December. In the first experiment the network was much more performant. Bandwidth +12% (200/800/100 Mbits/sec TX for client/middleware/memcached) and latency -25% .

2 Baseline without Middleware

2.1 One Server

2.1.1 Setup

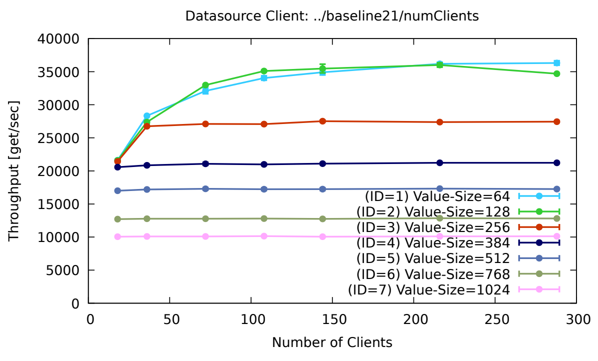
Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance (CT)	3
Virtual clients per threats (VC)	2, 4, 8, 12, 16, 24, 32
Total client connections	**18, 36, 72, 108, 144, 216, 288**
Workload	Read-only
Value Size (Bytes)	**64, 128, 256, 384, 512, 768, 1024**
Number of middlewares	0
Worker threads per middleware	N/A
Repetitions	3 (each 80s)

The setup is as follows: There are three clients running a memtier instance. Each client is connected to the same memcached server. The clients use a fixed number of 3 threads per memtier instance and a varying number of virtual clients per thread. There is no middleware used in the experiment. The data is collected in 3 runs (each 80s). Compare Experimental Setup for details on how the experiments have been run. All experiments in this report use the same deployment (compare Additional Remarks: Network and Azure). The network bottleneck in this experiment (as observed with iperf):

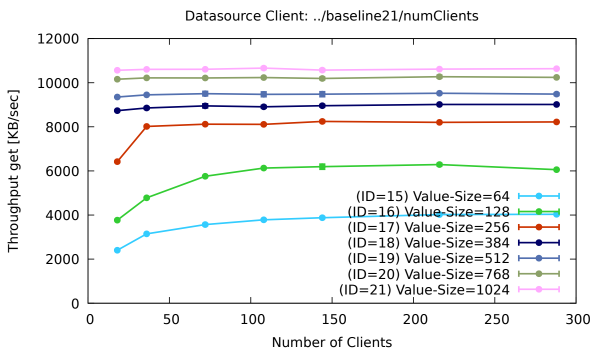
Round trip time: 0.9ms (+-0.2ms). And **Bottleneck Link Capacity:** Memcached server throughput TX 88 Mbits/s = 11'000 KByte/s

2.1.2 Throughput

The plots in this section show the results collected with the experiment according to setup 2.1.1. On the y-axis is the throughput shown as get/sec and KB/sec. Figure 3 shows on the x-axis the number of clients (total client connections), figure 4 the value size in bytes. **Saturation points** are described in table 1.

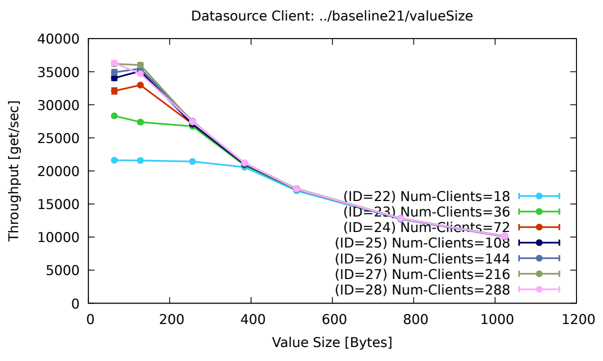


(a) throughput [gets/sec] vs Clients

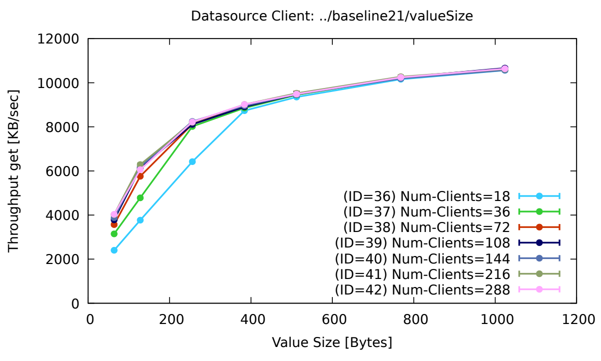


(b) throughput [KB/s] vs Clients

Figure 3: throughput baseline 21 (clients)



(a) throughput [gets/sec] vs Value Size



(b) throughput [KB/s] vs Value Size

Figure 4: throughput baseline 21 (value size)

2.1.3 Response Time

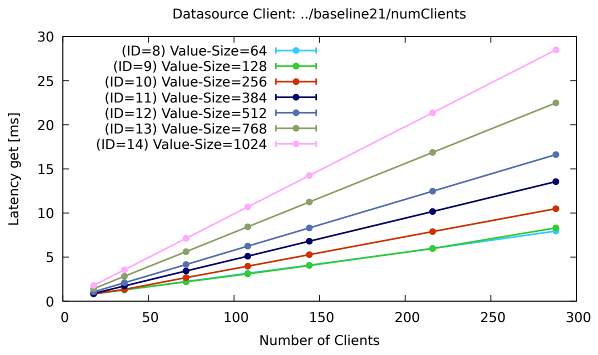
The plots in this section show the results collected with the experiment according to setup 2.1.1. The y-axis shows the latency of a get in ms. Figure 5a shows on the x-axis the number of clients (total client connections), figure 5b the value size in bytes. **Saturation points** are described in table 1.

2.1.4 Result Analysis

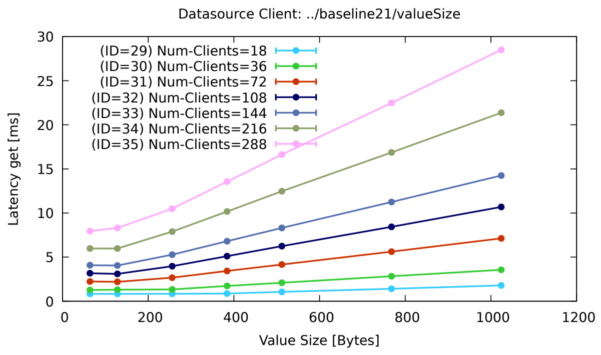
The results are explained by the interactive law $R = \frac{N}{X} - Z$. Figure 6a shows throughput and latency (for 288 client connections) as observed during the experiment. Figure 6b shows the derived response time using the interactive law with the parameters $N = 288, X = TP, Z = 0$. As one can observe, it conforms nearly perfect. Remark: I use $Z = 0$ (think time), as it is a fraction of a millisecond.

2.1.5 Explanation

Saturation points table 1: The memcached server is for configurations with more than 36 clients or 256 bytes data size saturated. The network tx rates are growing from 9'500 KB/sec (256b value size) up to 11'000Kb/sec (for 1024b value size). For value sizes 64b and 128b the cpu load reaches 95%. There is no oversaturated phase (degradation of performance due to problems). Analysis is done in 2.3.1 Bottleneck Analysis.

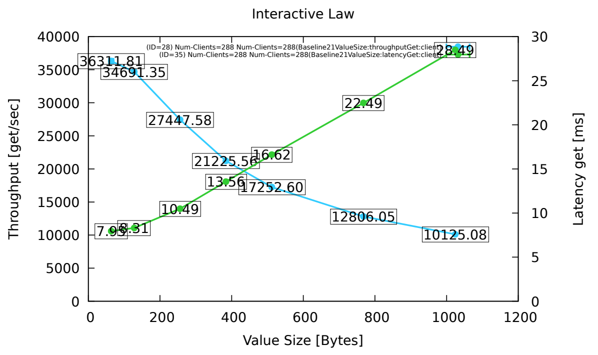


(a) latency [ms] vs Clients

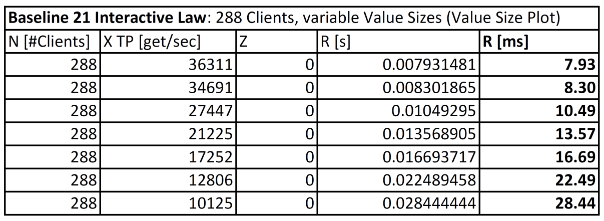


(b) latency [ms] vs Value Size

Figure 5: latency baseline 21



(a) Blue TP, Green Latency



Baseline 21 Interactive Law: 288 Clients, variable Value Sizes (Value Size Plot)				
N [#Clients]	X TP [get/sec]	Z	R [s]	**R [ms]**
288	36311	0	0.007931481	**7.93**
288	34691	0	0.008301865	**8.30**
288	27447	0	0.01049295	**10.49**
288	21225	0	0.013568905	**13.57**
288	17252	0	0.016693717	**16.69**
288	12806	0	0.022489458	**22.49**
288	10125	0	0.028444444	**28.44**

(b) Interactive Law: Derived Response Time

Figure 6: Interactive Law: baseline 21

Remark: As we do not have a clear definition of the saturation points, I use the reasoning as outlined in the book *The Art of Computer Systems Performance Analysis* [3]. In short: The system is undersaturated *before the knee* (compare figure 3.3 in the book). Before the knee, the response time does not increase significantly but the throughput rises as the load increases. To make sense out of this definition, a response time limit must be pre specified. I use 4ms. **Throughput and Response Time: table 2** gives a lower bound for the needed number of clients to fully use the network capacity. Given the capacity of the link, for a varying number of packet sizes (164, 228,...) the total number of packet/sec that one needs to send (with the given packet size to **fully use the capacity**) is calculated. The forth row lists the needed number of clients to generate the load in the first place ($\#clients = \frac{RTT*Capacity}{DataSizeOnWire}$). We observe, that 36 clients are enough to fully use the network capacity for value sizes 256+ (compare Figure 3a).

#c \d	64	128	256	384	512	768	1024
18	🟩	🟩	🟩	🟩	🟩	🟩	🟩
36	🟩	🟩	🟩	🟧	🟧	🟧	🟧
72	🟩	🟩	🟧	🟧	🟧	🟧	🟧
108	🟩	🟩	🟧	🟧	🟧	🟧	🟧
144	🟧	🟧	🟧	🟧	🟧	🟧	🟧
216	🟧	🟧	🟧	🟧	🟧	🟧	🟧
288	🟧	🟧	🟧	🟧	🟧	🟧	🟧

Table 1: Saturation Points Baseline 21

This implies that the maximum throughput can be reached with 36 clients. If we increase the number of clients to 72+ the response time has to grow as we already fully use the networks capacity ($ResponseTime = \frac{\#clients * DataSizeOnWire}{Capacity}$). The response time is then equal to the RTT + Overhead. If we use less than 36 clients, the throughput will be lower. The value size has a similar effect on throughput and response time. As long as we send below the networks capacity, an increasing value size generates higher throughput. Once the throughput reaches the networks capacity, the response time grows (compare last formula). This explanation is true, as long as there is no other bottleneck. Value sizes of 64b and 128b need a high number of packets to fully utilize the network. This introduces another source for increasing response time (very high CPU load) as will be explained in 2.3.1 Bottleneck Analysis. This is the further conclusion for this experiment: The memcached server has an upper bound of 35'000 requests / sec that can be processed (very high CPU load). The load doesn't vary for different packet sizes, but for the number of packets.

Throughput 88 Mbits/s = 11'000 KByte/s

Value Size	64	128	256	384	512	768	1024
Data Size On Wire [Bytes]:	**164**	**228**	**356**	**484**	**612**	**868**	**1124**
Packet / sec	68683	49404	31640	23273	18405	12977	10021
#clients for RTT 0.85ms	58	42	27	20	16	11	9

Table 2: Lower bound #-clients to use network capacity

2.2 Three Servers

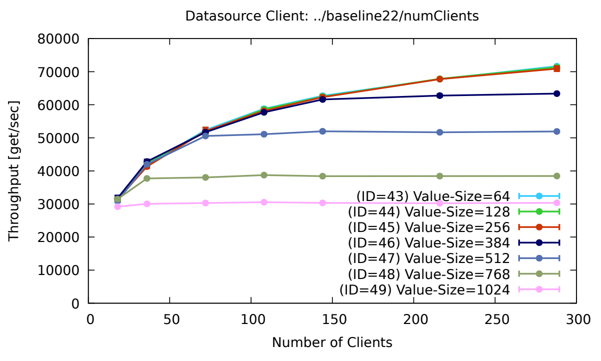
2.2.1 Setup

Number of servers	3
Number of client machines	3
Instances of memtier per machine	3
Threads per memtier instance (CT)	1
Virtual clients per thread (VC)	2, 4, 8, 12, 16, 24, 32
Total client connections	**18, 36, 72, 108, 144, 216, 288**
Workload	get
Value Size (Bytes)	**64, 128, 256, 384, 512, 768, 1024**
Number of middlewares	0
Worker threads per middleware	N/A
Repetitions	3 (each 80s)

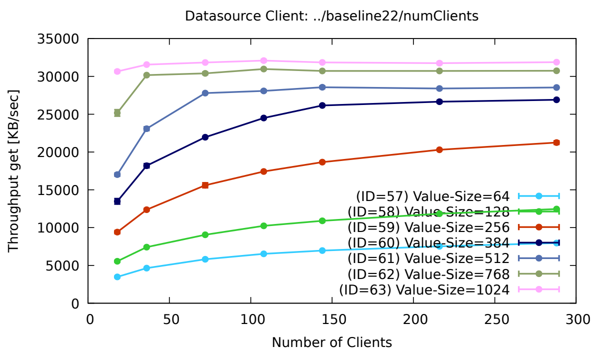
The setup is as follows: There are three clients each running three memtier instances. Each client is connected with one instance to one of three memcached servers. The clients use a fixed number of 1 thread per memtier instance and a varying number of virtual clients per thread. There is no middleware used in the experiment. The data is collected in 3 runs (each 80s). Compare Experimental Setup for details on how the experiments have been run. All experiments in this report use the same deployment (compare Additional Remarks: Network and Azure). The network bottleneck in this experiment (as observed with iperf): **Round trip time:** 0.9ms (+-0.2ms) and **Bottleneck Link Capacity:** Memcached server 3x throughput TX 264 Mbits/s = 33'000 KByte/s

2.2.2 Throughput

The plots in this section show the results collected with the experiment according to setup 2.2.1. On the y-axis is the throughput shown as get/sec and KB/sec. Figure 7 shows on the x-axis the number of clients (total client connections), figure 8 the value size in bytes. **Saturation points** are described in table 3.

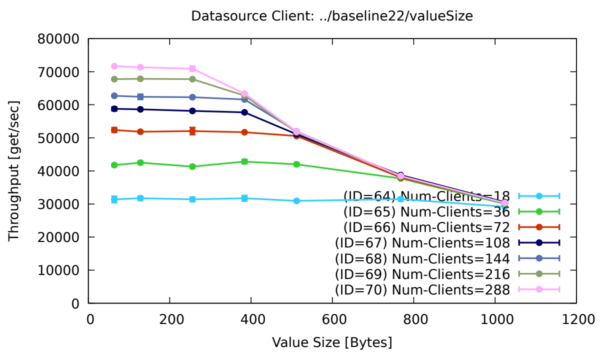


(a) throughput [gets/sec] vs Clients

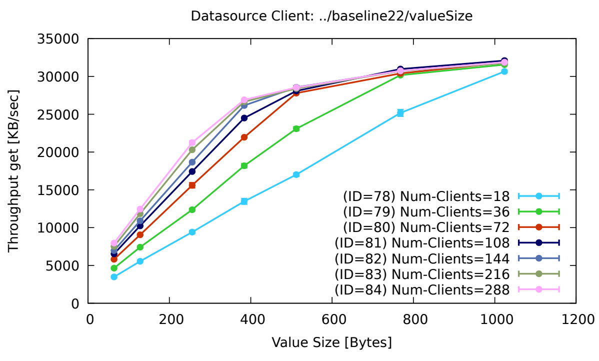


(b) throughput [KB/s] vs Clients

Figure 7: throughput baseline 22 (clients)



(a) throughput [gets/sec] vs Value Size



(b) throughput [KB/s] vs Value Size

Figure 8: throughput baseline 22 (value size)

2.2.3 Response Time

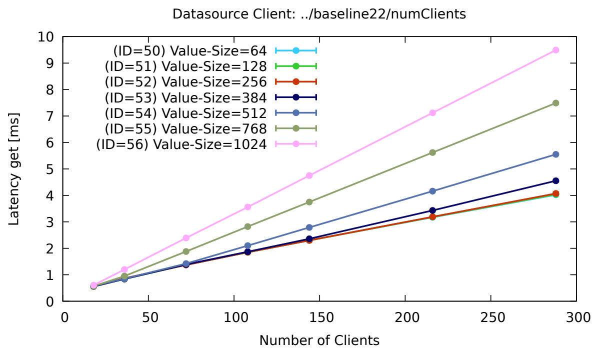
The plots in this section show the results collected with the experiment according to setup 2.2.1. The y-axis shows the latency of a get in ms. Figure 9a shows on the x-axis the number of clients (total client connections), figure 9b the value size in bytes. **Saturation points** are described in table 3.

2.2.4 Result Analysis

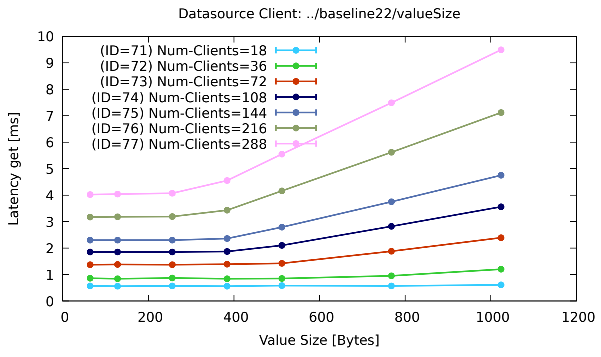
The results are explained by the interactive law $R = \frac{N}{X} - Z$. Figure 10a shows throughput and latency (for 288 client connections) as observed during the experiment. Figure 10b shows the derived response time using the interactive law with the parameters $N = 288, X = TP, Z = 0$. As one can observe, it conforms nearly perfect. Remark: I use $Z = 0$ (think time), as it is a fraction of a millisecond. The results are consistent with baseline 21. For bigger value sizes this is directly visible (3x tp). For smaller sizes we have to take into account a new client side bottleneck (2.3.1 Bottleneck Analysis).

2.2.5 Explanation

Saturation points table 3: The saturated points are listed in the table. Once we have enough clients (starting with 72) the network tx rates are between 10'000 KB/sec (384b) and 11'000 KB/sec (1024b). This data has been collected with NMON and will be further analysed in

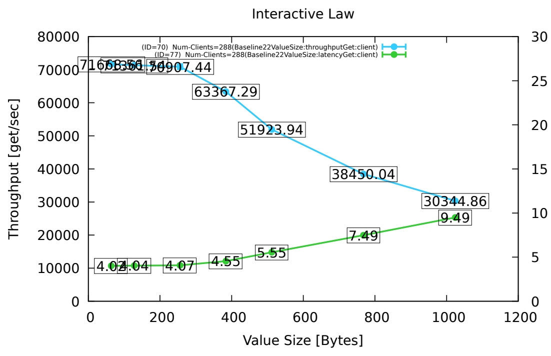


(a) latency [ms] vs Clients

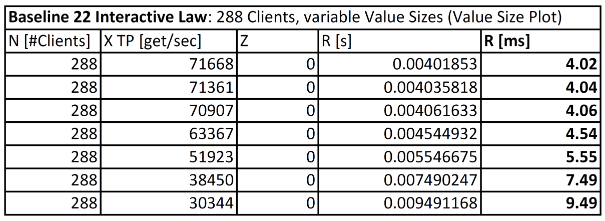


(b) latency [ms] vs Value Size

Figure 9: latency baseline 22



(a) Blue TP, Green Latency



Baseline 22 Interactive Law: 288 Clients, variable Value Sizes (Value Size Plot)				
N [#Clients]	X TP [get/sec]	Z	R [s]	**R [ms]**
288	71668	0	0.00401853	**4.02**
288	71361	0	0.004035818	**4.04**
288	70907	0	0.004061633	**4.06**
288	63367	0	0.004544932	**4.54**
288	51923	0	0.005546675	**5.55**
288	38450	0	0.007490247	**7.49**
288	30344	0	0.009491168	**9.49**

(b) Interactive Law: Derived Response Time

Figure 10: Interactive Law: baseline 22

2.3.1 Bottleneck Analysis. There is no oversaturated phase (degradation of performance due to problems).

Throughput and Response Time: table 4 gives a lower bound for the needed number of clients to fully use the network capacity. Compare 2.1.5 for details about the table. We observe, that 36 clients are (in contrast to baseline 21) not enough to fully use the network capacity for value sizes 256+ (compare Figure 7a). Compared to baseline 21, everything is shifted, as we have tripled the network bandwidth. The general explanation (throughput vs latency, influence of number of clients and value size) is still true. I assume I do not have to repeat this, as the details are in 2.1.5. Using the formulas introduced in 2.1.5, one can for example explain why baseline 22 has only 10ms latency for 288 clients and 1024b value size while baseline 21 has 30ms (for the same configuration). By comparing the two tables, we observe that we need 9

#c \d	64	128	256	384	512	768	1024
18	🟩	🟩	🟩	🟩	🟩	🟩	🟩
36	🟩	🟩	🟩	🟩	🟩	🟩	🟩
72	🟩	🟩	🟩	🟩	🟩	🟧	🟧
108	🟩	🟩	🟩	🟩	🟧	🟧	🟧
144	🟩	🟩	🟩	🟧	🟧	🟧	🟧
216	🟩	🟩	🟩	🟧	🟧	🟧	🟧
288	🟩	🟩	🟩	🟧	🟧	🟧	🟧

Table 3: Saturation Points Baseline 22

Throughput 264 Mbits/s = 33'000 KByte/s

Value Size	64	128	256	384	512	768	1024
Packet Size [Bytes]:	**164**	**228**	**356**	**484**	**612**	**868**	**1124**
Packet / sec	206049	148211	94921	69818	55216	38931	30064
#clients for RTT 0.85ms	175	126	81	59	47	33	26

Table 4: Lower bound #-clients to use full network capacity

respective 26 clients to fully use the network capacity. As a result is the *used number of clients to needed number of clients* ratio (288/9=32 vs 288/26=11) three times bigger. This has a direct influence on the response time.

Further conclusions for this experiment: The load on the clients CPU is quit high (80%) for data sizes 64b, 128b and 256b. The number of sent packets (as observed in NMON) is around 25'000 p/sec. As a packet is one request, we can conclude that a client has an upper bound of 25'000 req/sec (for a complete node, all instances).

2.3 Summary

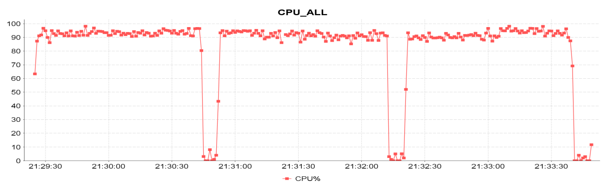
I fix the value size to be 256 bytes to analyse the system and fill out the table. The configuration for one memcached server provides a good trade-off of throughput and response time. This is expected (compare table 2), as 36 clients is enough to use all the network capacity, but not to high (36 instead of the optimal 27 clients) to let the response time explode. For the three memcached server setting the same rule (find closest client configuration to optimal configuration of 81 clients, compare table 4) leads to 72 clients. Nevertheless I use 108 clients. As (will be) outlined, the bottleneck is not the network capacity. The clients cpu is just not fast enough to generate the needed load. We can tolerate a slightly higher response time. This value as a compromise.

	Maximum Throughput	Corresponding Response Time	Configuration
One memcached server	26'752 [get/sec]	1.34 [ms]	36 clients, 256 bytes
Three memcached servers	58'155 [get/sec]	1.85 [ms]	108 clients, 256 bytes

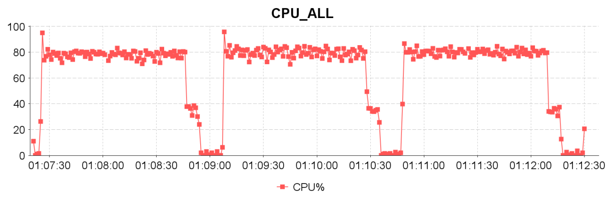
2.3.1 Bottleneck Analysis

Figure 12 shows the real throughput of the system in KB/sec. To create the plots I used as input data 3a and 7a. Memtier provides the throughput in KB/sec (3b and 7b) by using those numbers and then multiplying them by valueSize+50. But this is wrong. The packet size on the wire corresponds to valueSize+100 bytes, this is what the plots show (conforms with Wireshark and Server NMON plots). It is obvious that the two configurations show a different maximum throughput, as we have different network capacities. Please compare explanations (and tables) given in 2.1.5 and 2.2.5 to get the numbers needed in this part. The straight lines in figure 12 real throughput (top 4 lines left plot, top 3 lines right plot) show the **main bottleneck** of both setups, the network capacity. We see that we reach 11'000 KB/sec respective 33'000 KB/sec. This is the expected upper bound, given the networks performance as measured with iperf. We reach this upper bound, once we have enough clients, or value size (as outlined in 2.1.5 and 2.2.5). The gap between the different value sizes that come close to the networks capacity (the last 10%) can be explained with the different numbers of request that are needed to create the load. While we need just around 10'000 packets (per server) for 1024b we need over 31'000 packets for 256b (keep in mind that that the packet size on the wire is 1124b respective 356b). There is fewer processing overhead for bigger packets (in general for network cards). More important, having more requests has an influence on the throughput as the next bottleneck, the

CPU load outlines. In **baseline 21** the server CPU has a very high load for 64b and 128b. As explained (in further conclusions 2.1.5) one memcached server has an upper bound of 35'000 (to 37'000) requests it can handle. Figure 11a shows this very high load on the memcached server (64b, 288 clients). This high load is visible for 72+ clients. The same is true for 128b value sizes. Compare figure 3a, the clients requests are at 35'000/sec. **Baseline 22** has a similar bottleneck, but this time it is on the clients side (because one server never has to serve more than 25'000 request/sec, therefore its impact is limited). Figure 11b shows the clients cpu load for 64b and 288 clients. The same is true for 128b. If we use only 18 clients, the utilization is around 50% and increases between the different configurations. The cpu load for 256b is still at 75%, for 384b at 43% and for the rest between 30% and 19%. To conclude, the number of requests per seconds is mainly bound by the clients cpu for 64b to 256b. At the same time we can observe a server cpu load of 30%, while for the other configurations it is around 10%-20%. To conclude, the server cpu has a minor effect for baseline 21.

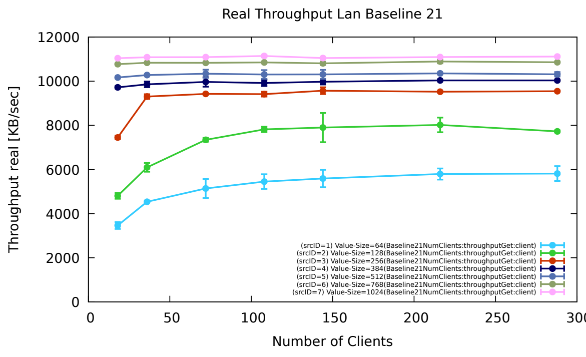


(a) Bottleneck Baseline 21 Server CPU

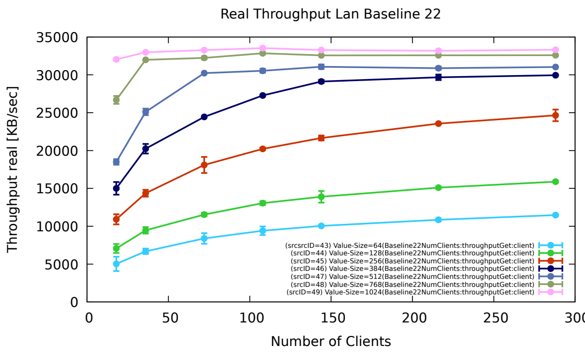


(b) Bottleneck Baseline 22 Client CPU

Figure 11: CPU Bottlenecks Baseline 21/22



(a) Lan Throughput Baseline 21



(b) Lan Throughput Baseline 22

Figure 12: real throughput

2.3.2 One and Three Servers Configurations

Response time: As we use the same number of clients in both settings, but have three times the network capacity in baseline 22, the response time is by a factor of 3 smaller for baseline 22. This has been derived in 2.2.5 Explanation (*used number of clients to needed number of clients* ratio). The **Throughput**: is 3x higher for baseline 22, as long as we do not reach a CPU limit (2.3.1 Bottleneck Analysis). This is the reason why we do not observe a 3x higher throughput for small value sizes. But I have to point out, that this is a client side bottleneck.

3 Baseline with Middleware

3.1 One Middleware, One Server

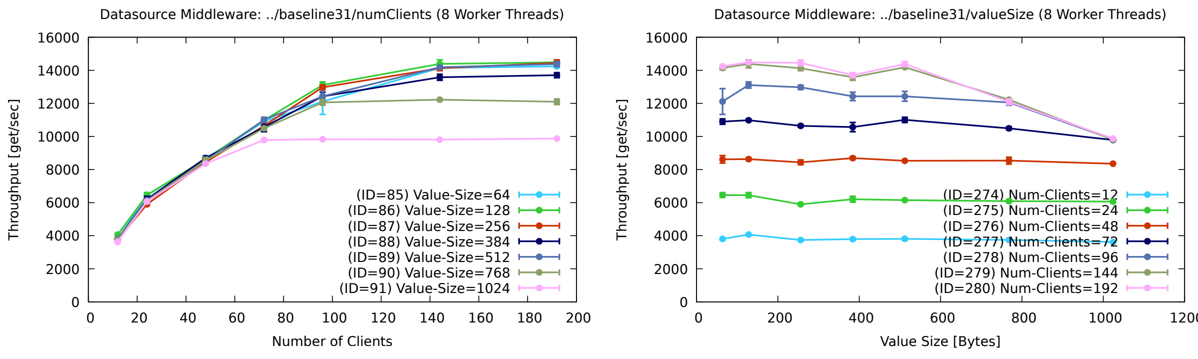
3.1.1 Setup

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance (CT)	2
Virtual clients per thread (VC)	2, 4, 8, 12, 16, 24, 32
Total client connections	**12, 24, 48, 72, 96, 144, 192**
Value Size (Bytes)	**64, 128, 256, 384, 512, 768, 1024**
Value size (Bytes)	64, 128, 256, 384, 512, 768, 1024
Number of middlewares	1
Worker threads per middleware	8, 32, 64
Repetitions	3 (each 80s)

The setup is as follows: There are three clients running a memtier instance. Each client is connected to the same middleware. The clients use a fixed number of 2 threads per memtier instance and a varying number of virtual clients per thread. There is one middleware used, connected to one memcached server. The data is collected in 3 runs (each 80s). Compare Experimental Setup for details on how the experiments have been run. All experiments in this report use the same deployment (compare Additional Remarks: Network and Azure). The network bottleneck in this experiment (as observed with iperf): **Round trip time:** 0.9ms (+-0.2ms). And **Bottleneck Link Capacity:** Memcached server 1x throughput TX 88 Mbits/s = 11'000 KByte/s.

3.1.2 Throughput

The plots in this section show the results collected with the experiment according to setup 3.1.1. On the y-axis is the throughput shown as get/sec. On the x-axis the number of clients (total client connections) or the value size in bytes. Figure 13, 14, 15 contain the results for 8, 32 and 64 worker threads. **Saturation points** are described in table 5.

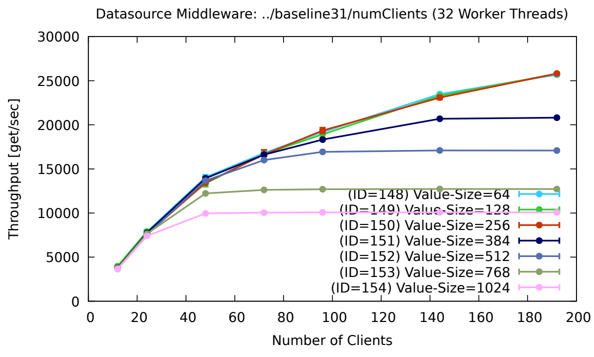


(a) throughput [gets/sec] vs Clients (b) throughput [gets/sec] vs Value Size

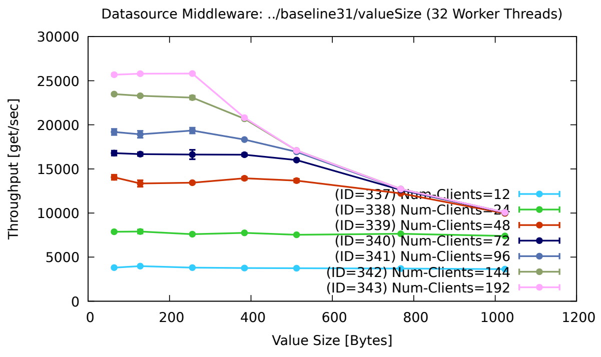
Figure 13: throughput baseline 31 (8 worker threads)

3.1.3 Response Time

The plots in this section show the results collected with the experiment according to setup 3.1.1. The y-axis shows the latency of a get in ms. On the x-axis the number of clients (total client connections) or the value size in bytes. Figure 16, 17, 18 contain the results for 8, 32 and 64

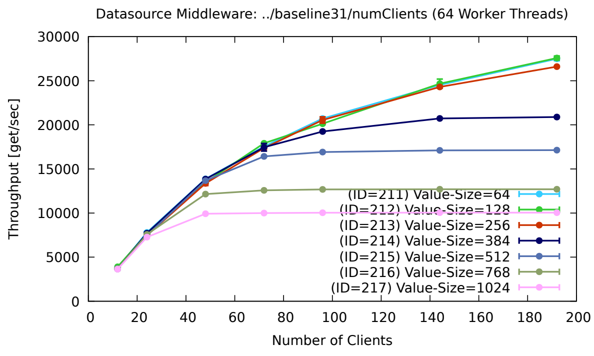


(a) throughput [gets/sec] vs Clients

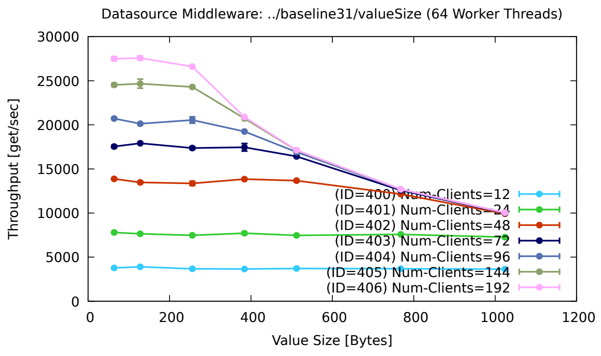


(b) throughput [gets/sec] vs Value Size

Figure 14: throughput baseline 31 (32 worker threads)



(a) throughput [gets/sec] vs Clients



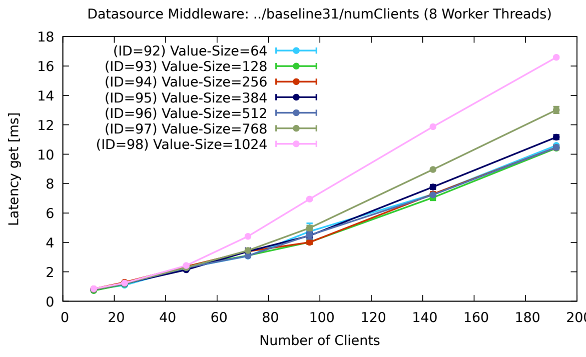
(b) throughput [gets/sec] vs Value Size

Figure 15: throughput baseline 31 (64 worker threads)

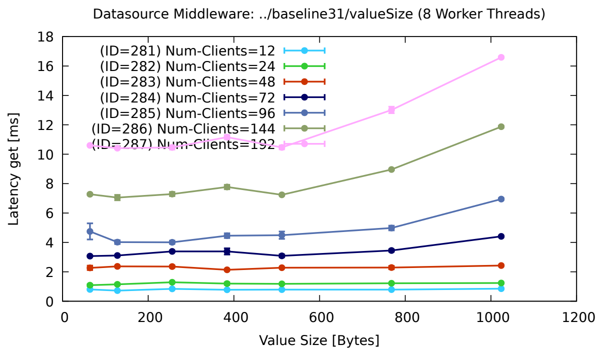
worker threads. **Saturation points** are described in table 5. **Remark:** For all middleware plots I define the response time to be the time during which a request occupies resources in the middleware: This interval starts when the networking thread adds the clients request until the response from the worker thread has been sent back to the client. This is equal to the queue waiting time plus a very small amount of worker thread processing time (fraction of ms) and the complete request time to get the response from the memcached server.

3.1.4 Result Analysis

The results are explained by the interactive law $R = \frac{N}{X} - Z$. Figure 19a shows throughput and latency (for 192 client connections, 32 Worker Threads) as observed during the experiment. Figure 19b shows the derived response time using the interactive law with the parameters $N = 192, X = TP, Z = 2.95$. As one can observe, it conforms. **General remark for all interactive law proofs:** Figure 20 Interactive Law: Think Time (32 WT's) shows the think time of the clients (as client and value plots). The think time depends mainly on the number of clients (if worker threads are fixed) and is not so much influenced by the value size. This is the reason why I took the value size plots to show conformity with the interactive law. I **define the think time** in all experiments to be equal to the interval between the worker threads sends back the response to the client and the next time the network thread adds the client to the request-queue. The think time therefore also contains the time needed to send the response back to the client and the next request to the middleware as well as the time needed

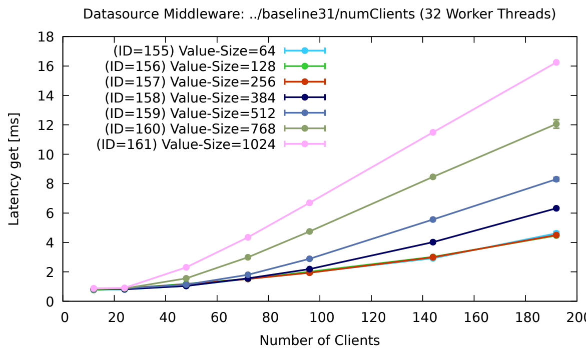


(a) latency [ms] vs Clients

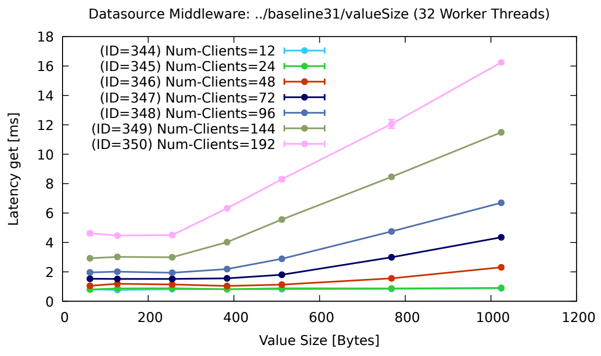


(b) latency [ms] vs Value Size

Figure 16: latency baseline 31 (8 worker threads)



(a) latency [ms] vs Clients



(b) latency [ms] vs Value Size

Figure 17: latency baseline 31 (32 worker threads)

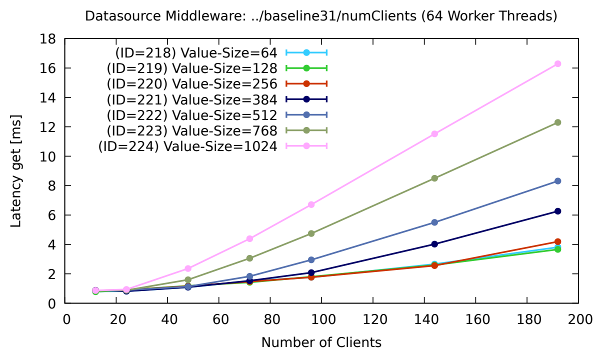
to check the connection for a new request. As a result it contains some jitter, but nevertheless, the interactive law can be explained very well. In the following chapters I will not show the plots, as we do not have enough pages for this report. But all 24 plots are available for all combinations of WT, client and value plots. The next section will explain why the results are consistent with the previous experiments.

3.1.5 Explanation

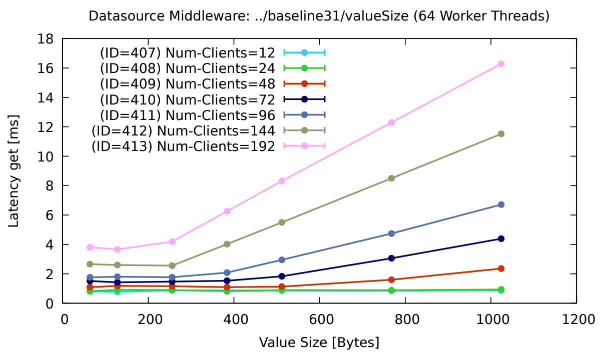
Saturation points are given in the table 5. Analysis is done in 3.5.1 Bottleneck Analysis. There is no oversaturated phase (degradation of performance due to problems). I used the same definition of saturation points as outlined in 2.1.5.

	8 Worker Threads							32 Worker Threads							64 Worker Threads						
#c \d	64	128	256	384	512	768	1024	64	128	256	384	512	768	1024	64	128	256	384	512	768	1024
12	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
24	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
48	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟧	🟧
72	🟩	🟩	🟩	🟩	🟩	🟩	🟧	🟩	🟩	🟩	🟩	🟩	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟧	🟧
96	🟩	🟩	🟩	🟩	🟩	🟧	🟧	🟩	🟩	🟩	🟩	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟧	🟧	🟧
144	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟧	🟧	🟧	🟧
192	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧

Table 5: Saturation Points Baseline 31

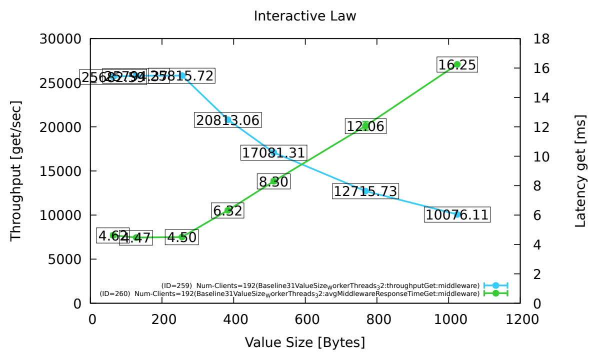


(a) latency [ms] vs Clients

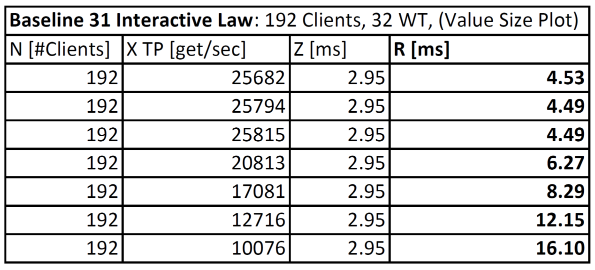


(b) latency [ms] vs Value Size

Figure 18: latency baseline 31 (64 worker threads)



(a) Blue TP, Green Latency

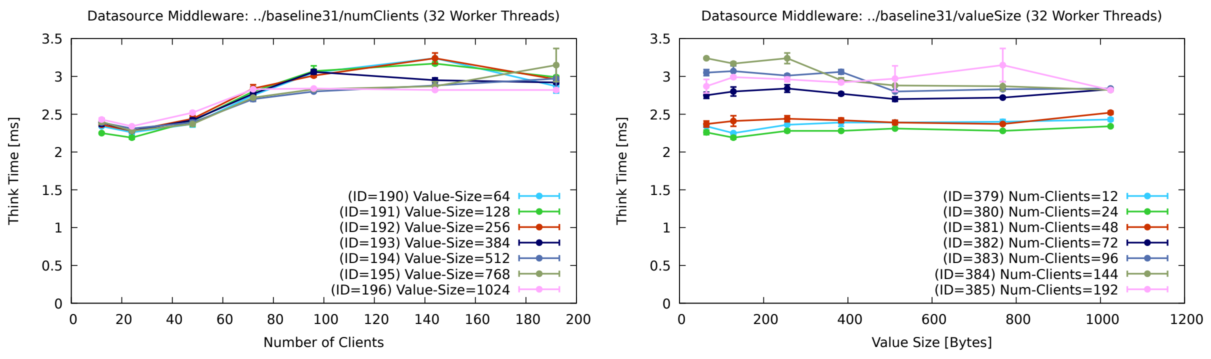


Baseline 31 Interactive Law: 192 Clients, 32 WT, (Value Size Plot)			
N [#Clients]	X TP [get/sec]	Z [ms]	**R [ms]**
192	25682	2.95	**4.53**
192	25794	2.95	**4.49**
192	25815	2.95	**4.49**
192	20813	2.95	**6.27**
192	17081	2.95	**8.29**
192	12716	2.95	**12.15**
192	10076	2.95	**16.10**

(b) Interactive Law: Derived Response Time

Figure 19: Interactive Law: baseline 31

Throughput and Response Time: table 6 gives a lower bound for the needed number of clients to fully use the network capacity. The table has been introduced in 2.1.5 Explanation. For chapter 3 I will reuse it with some changes. Also compare the formulas defined there. From 3.1.4 Result Analysis we know that the client think time is not the same if we change the number of clients. While increasing the number of clients we observe a think time between 2.5ms and 3ms for most of the cases (for 32/64 worker threads this is what has been observed with the given parameters). The last two rows show this (including a RTT of 0.8ms). This is the real latency we have to use to figure out how many clients are need to use all the given network capacity. It is obvious, that we need much more clients (3x or more) than in baseline 21/22 to fully use the capacity (because of the bigger latency). Further we have fewer clients than in baseline 21/22, only 2/3 for each data point (on client plots for example 192 instead of 288). Even if we have the totally unrealistic assumption that the middleware has zero processing time and behaves invariant under load, we will reach max throughput much further on the right side of the plots (this is the reason why the plots look different), if it is even possible because of the high number of needed clients (compare 8). The 4th row in the table lists again the needed number of connections to fully use the bandwidth assuming a 0.85ms RTT. In chapter three this corresponds to the number of worker threads (if we assume that we have enough request in the queue, we will reach max throughput for 64b with 58 worker threads). To conclude: We need much more clients than worker threads to fully use the given network capacity. This is the first big insight. The second one is the average queue length: It is mainly influenced by the number of clients and not at all by the value size (within the given parameter space). We can never have



(a) Client Plot (b) Value Plot

Figure 20: Interactive Law: Think Time (32 WT's)

Throughput 88 Mbits/s = 11'000 KByte/s (32/64 Worker Threads)

Value Size	64	128	256	384	512	768	1024
Packet Size [Bytes]:	**164**	**228**	**356**	**484**	**612**	**868**	**1124**
Packet / sec	68683	49404	31640	23273	18405	12977	10021
#WT's for RTT 0.85ms	58	42	27	20	16	11	9
#clients for RTT+TT 3.3ms	227	163	104	77	61	43	33
#clients for RTT+TT 3.8ms	261	188	120	88	70	49	38

Table 6: Lower bound #-clients to use full network capacity

a longer queue then total number of connections. The second important factor for queue length is the number of worker threads. Each thread will always process one client. This means the queue has an average upper length of number of clients minus number of worker threads. The total number of requests in the system will be a little bit lower than the number of clients, as some of the requests are on the way back to the clients, or the clients are preparing them, or they are on the way to the middleware (from the clients). If we combine this with the first insight, we observe that it is impossible to fully use the networks capacity with small value sizes (we need 100+ clients) while keeping the latency low. This is a huge difference to baseline 21/22. The latency as observed by the clients (or the middleware according to the definition above) is mainly driven by the queue waiting time (without network bottlenecks *) which itself depends mainly on the queue length. Those insights together with the table explain the behaviour of the system very well. 8WTs are enough to fully use the network bandwidth for 768b and 1024b, for 256b-512b we need 32WTs (compare 13a 14a 15a straight line, network bottleneck). This explains why we do not see much difference in throughput for 32WTs and 64WTs plots. Only the small value sizes can profit from 64WTs (minimal), but those configurations are limited mainly by the number of clients and we just have not enough. (*) In baseline 21 it has been explained, why the response time grows once we reach the networks capacity (compare those paragraphs and the outline about the real network throughput). This increase is again visible in the plots 16a 17a 18a. As long as we do not reach the networks capacity, the latency is driven by the queue length. Once the capacity is a bottleneck, we see an increasing latency. Further conclusion for this experiment have already be given in the explanation above to explain throughput and latency, namely the high number of clients needed and the inevitable circumstance of a longer queue to reach high throughput (KB/sec) for small value sizes.

3.2 One Middleware, Three Server

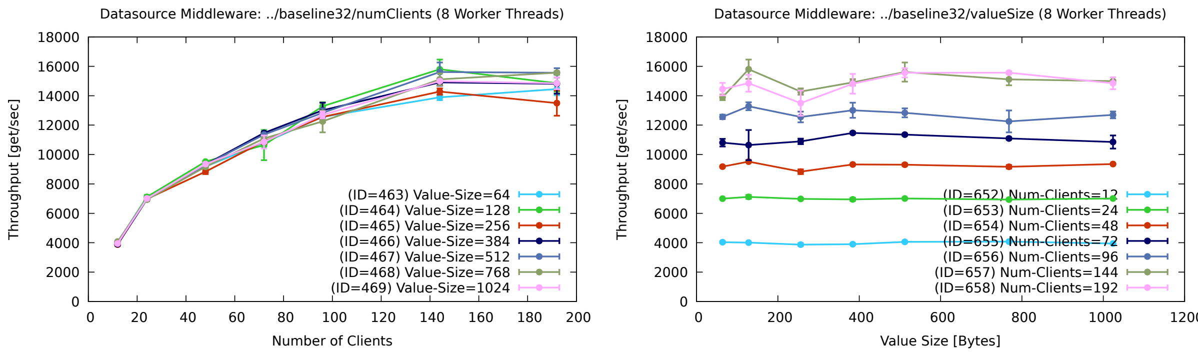
3.2.1 Setup

Number of servers	3
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance (CT)	2
Virtual clients per thread (VC)	2, 4, 8, 12, 16, 24, 32
Total client connections	**12, 24, 48, 72, 96, 144, 192**
Workload	get
Value Size (Bytes)	**64, 128, 256, 384, 512, 768, 1024**
Number of middlewares	1
Worker threads per middleware	8, 32, 64
Repetitions	3 (each 80s)

The setup is as follows: There are three clients running a memtier instance. Each client is connected to the same middleware. The clients uses a fixed number of 2 threads per memtier instance and a varying number of virtual clients per thread. There is one middleware used, connected to three memcached servers. The data is collected in 3 runs (each 80s). Compare Experimental Setup for details on how the experiments have been run. All experiments in this report use the same deployment (compare Additional Remarks: Network and Azure). The network bottleneck in this experiment (as observed with iperf): **Round trip time:** 0.9ms (+-0.2ms) and **Bottleneck Link Capacity** 3x Memcached server throughput TX 264 Mbits/s = 33'000 KByte/s.

3.2.2 Throughput

The plots in this section show the results collected with the experiment according to setup 3.2.1. On the y-axis is the throughput shown as get/sec. On the x-axis the number of clients (total client connections) or the value size in bytes. Figure 21, 22, 23 contain the results for 8, 32 and 64 worker threads. **Saturation points** are described in table 7.



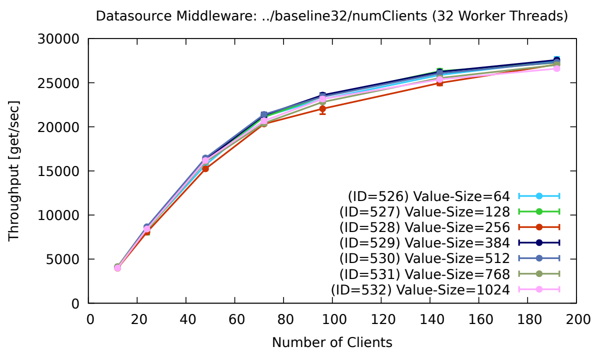
(a) throughput [gets/sec] vs Clients

(b) throughput [gets/sec] vs Value Size

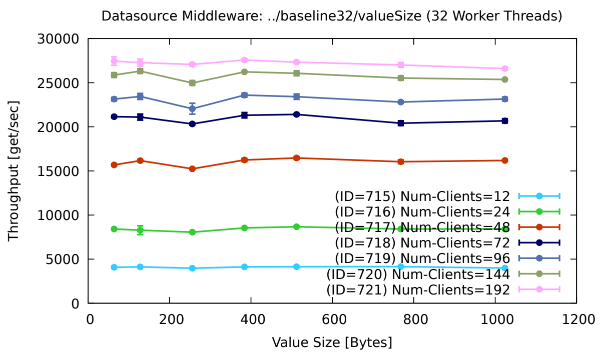
Figure 21: throughput baseline 32 (8 worker threads)

3.2.3 Response Time

The plots in this section show the results collected with the experiment according to setup 3.2.1. The y-axis shows the latency of a get in ms. On the x-axis the number of clients (total client connections) or the value size in bytes. Figure 24, 25, 26 contain the results for 8, 32 and 64 worker threads. **Saturation points** are described in table 7.

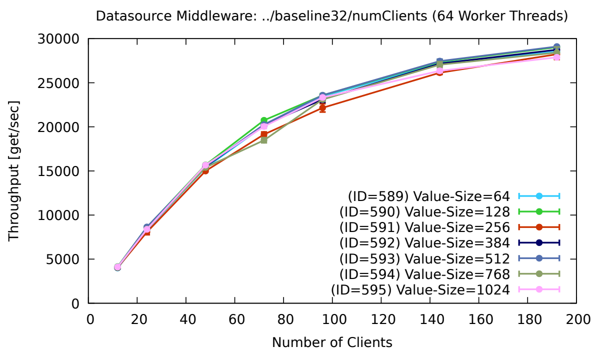


(a) throughput [gets/sec] vs Clients

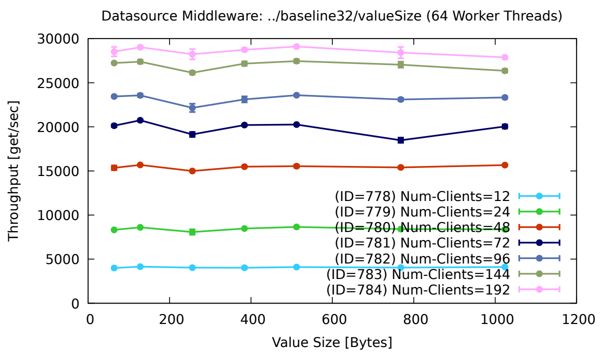


(b) throughput [gets/sec] vs Value Size

Figure 22: throughput baseline 32 (32 worker threads)



(a) throughput [gets/sec] vs Clients



(b) throughput [gets/sec] vs Value Size

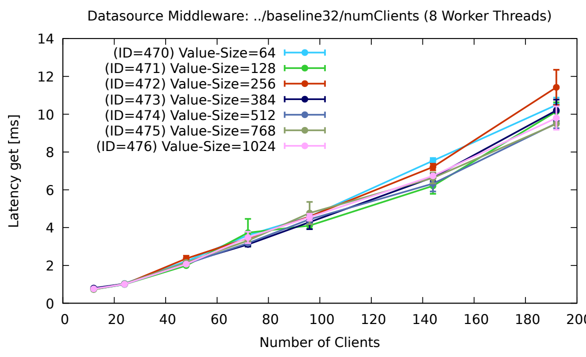
Figure 23: throughput baseline 32 (64 worker threads)

3.2.4 Result Analysis

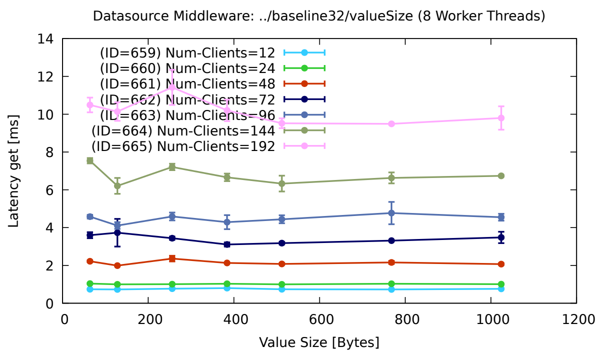
The results are explained by the interactive law $R = \frac{N}{X} - Z$. Figure 27a shows throughput and latency (for 192 client connections, 32 Worker Threads) as observed during the experiment. Figure 27b shows the derived response time using the interactive law with the parameters $N = 192, X = TP, Z = 3.2$. As one can observe, it conforms. Differences are assumed to be network jitter and or client load. Remark: The law hold also if I compute it for other graphs that are not flat. The next section explains why the results are consistent with the previous experiments.

3.2.5 Explanation (pts 20)

Saturation points: are given in the table 7. **Throughput and Response Time:** table 8 gives a lower bound for the needed number of clients to fully use the network capacity. **Please compare 3.1.5 Explanation for general remarks.** I decided to use the table with the same client think time as used in 6 (because it also fits for baseline 34), namely 2.5ms-3ms (even though in this experiment it is 2.5ms-3.5ms). The lower bound provided by the table shows that we need a very high number of packets (30'000+) to use all the capacity even for 1024b value size. Further we see the needed number of clients is out of the parameter space (192 clients). In throughput figures 21a 22a 23a and latency reports 24a 25a 26a is the absence of a bandwidth bottleneck visible. For the throughput this is obvious, as an increase in value size for

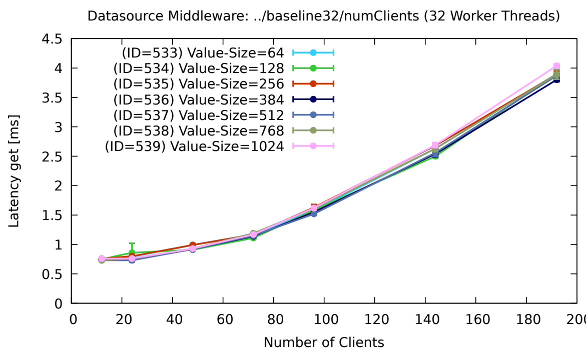


(a) latency [ms] vs Clients

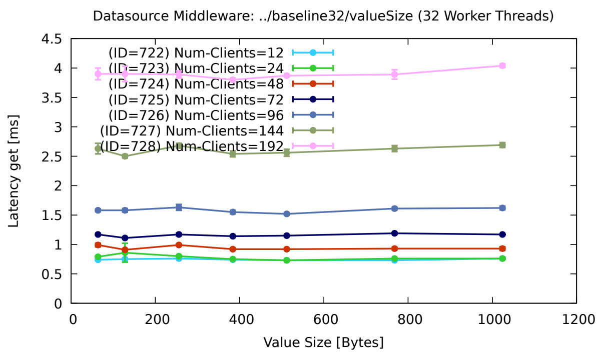


(b) latency [ms] vs Value Size

Figure 24: latency baseline 32 (8 worker threads)



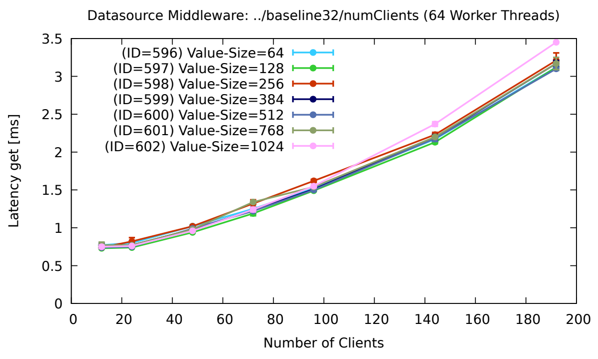
(a) latency [ms] vs Clients



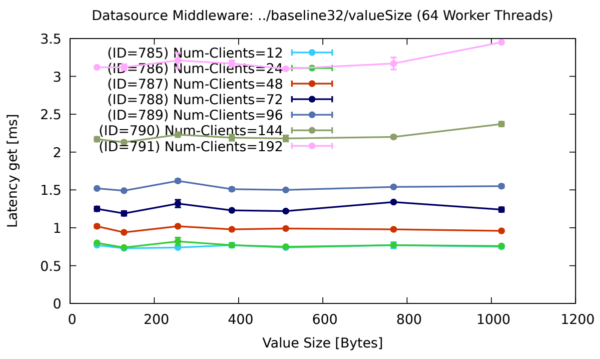
(b) latency [ms] vs Value Size

Figure 25: latency baseline 32 (32 worker threads)

a given number of packets is only possible, if the network can handle the load. In the latency plots, as explained in 3.1.5, the latency lines increase (in comparison with other value sizes), once the bottleneck capacity is reached (before that point the latency is mainly influenced by the queue length). I assume I do not have to repeat everything (how the latency is influenced by worker threads and number of clients and all the other things). The difference between 8WTs and 32/64WTs can also be explained by the table, we observe that we need at lease 26 WTs. Further conclusions for this experiment: The reason why we do not reach higher throughput than 30'000 packets / sec can be explained by a bottleneck inside of the middleware. The simplified model assumes, that all worker threads have direct access to the queue. But this is not true. All request have to go true the net-thread. And this thread seems to have an upper bound of around 30'000 requests / sec it can handle. This means the forwarding process takes around 0.03ms.

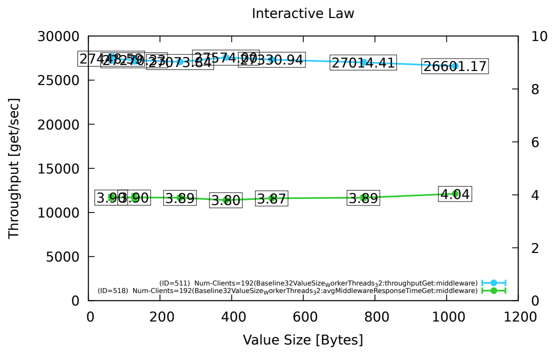


(a) latency [ms] vs Clients

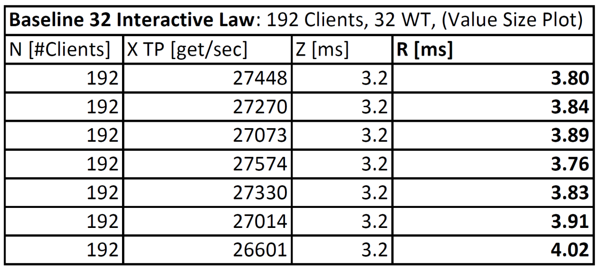


(b) latency [ms] vs Value Size

Figure 26: latency baseline 32 (64 worker threads)



(a) Blue TP, Green Latency



Baseline 32 Interactive Law: 192 Clients, 32 WT, (Value Size Plot)			
N [#Clients]	X TP [get/sec]	Z [ms]	**R [ms]**
192	27448	3.2	**3.80**
192	27270	3.2	**3.84**
192	27073	3.2	**3.89**
192	27574	3.2	**3.76**
192	27330	3.2	**3.83**
192	27014	3.2	**3.91**
192	26601	3.2	**4.02**

(b) Interactive Law: Derived Response Time

Figure 27: Interactive Law: baseline 32

	8 Worker Threads							32 Worker Threads							64 Worker Threads						
#c \d	64	128	256	384	512	768	1024	64	128	256	384	512	768	1024	64	128	256	384	512	768	1024
12	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
24	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
48	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
72	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
96	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
144	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟧
192	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧

Table 7: Saturation Points Baseline 32

Throughput 264 Mbits/s = 33'000 KByte/s (32/64 Worker Threads)

Value Size	64	128	256	384	512	768	1024
Packet Size [Bytes]:	**164**	**228**	**356**	**484**	**612**	**868**	**1124**
Packet / sec	206049	148211	94921	69818	55216	38931	30064
#WT's for RTT 0.85ms	175	126	81	59	47	33	26
#clients for RTT+TT 3.3ms	680	489	313	230	182	128	99
#clients for RTT+TT 3.8ms	783	563	361	265	210	148	114

Table 8: Lower bound #-clients to use full network capacity

3.3 Two Middlewares, One Server

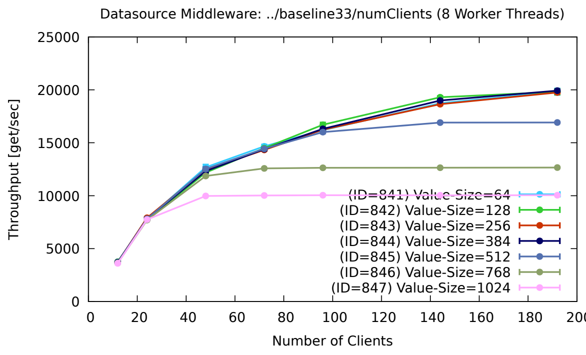
3.3.1 Setup

Number of servers	1
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance (CT)	1
Virtual clients per thread (VC)	2, 4, 8, 12, 16, 24, 32
Total client connections	**12, 24, 48, 72, 96, 144, 192**
Workload	get
Value Size (Bytes)	**64, 128, 256, 384, 512, 768, 1024**
Number of middlewares	2
Worker threads per middleware	8, 32, 64
Repetitions	3 (each 80s)

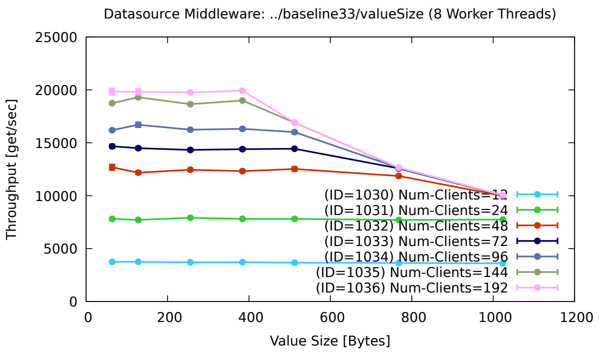
The setup is as follows: There are three clients each running two memtier instances. Each client instance is connected to a different middleware. The clients use a fixed number of 1 thread per memtier instance and a varying number of virtual clients per thread. There are two middleware used, connected to one memcached server. The data is collected in 3 runs (each 80s). Compare Experimental Setup for details on how the experiments have been run. All experiments in this report use the same deployment (compare Additional Remarks: Network and Azure). The network bottleneck in this experiment (as observed with iperf): **Round trip time:** 0.9ms (+-0.2ms) and **Bottleneck Link Capacity:** 1x Memcached server throughput TX 88 Mbits/s = 11'000 KByte/s.

3.3.2 Throughput

The plots in this section show the results collected with the experiment according to setup 3.3.1. On the y-axis is the throughput shown as get/sec. On the x-axis the number of clients (total client connections) or the value size in bytes. Figure 28, 29, 30 contain the results for 8, 32 and 64 worker threads. **Saturation points** are described in table 9.



(a) throughput [gets/sec] vs Clients

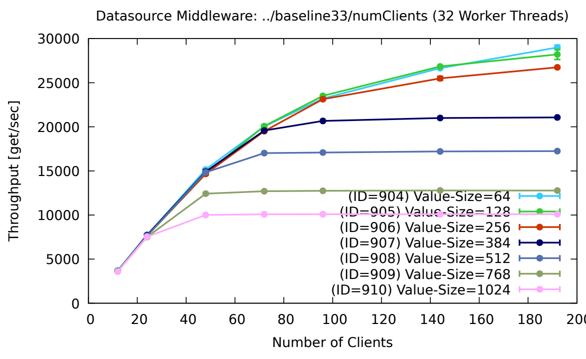


(b) throughput [gets/sec] vs Value Size

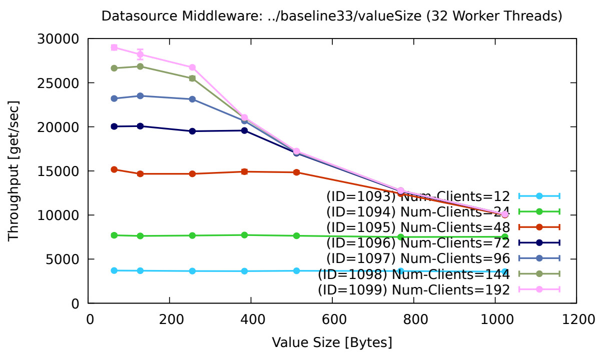
Figure 28: throughput baseline 33 (8 worker threads)

3.3.3 Response Time

The plots in this section show the results collected with the experiment according to setup 3.3.1. The y-axis shows the latency of a get in ms. On the x-axis the number of clients (total client connections) or the value size in bytes. Figure 31, 32, 33 contain the results for 8, 32 and 64 worker threads. **Saturation points** are described in table 9.

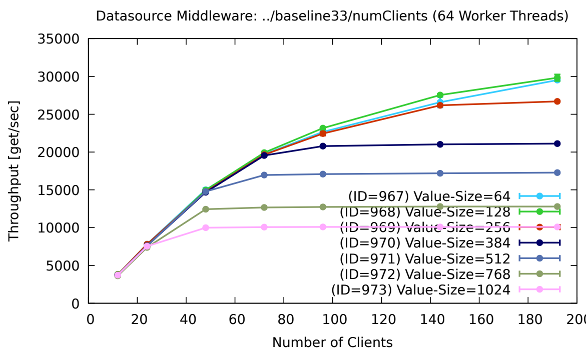


(a) throughput [gets/sec] vs Clients

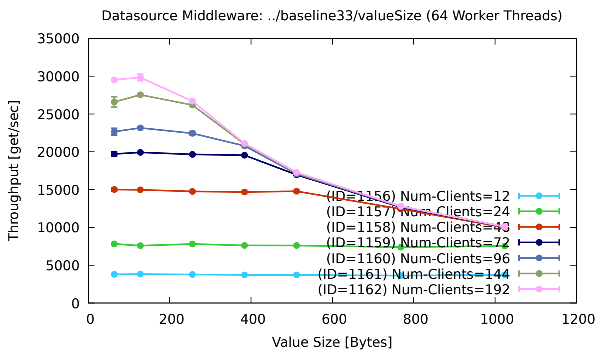


(b) throughput [gets/sec] vs Value Size

Figure 29: throughput baseline 33 (32 worker threads)



(a) throughput [gets/sec] vs Clients



(b) throughput [gets/sec] vs Value Size

Figure 30: throughput baseline 33 (64 worker threads)

3.3.4 Result Analysis

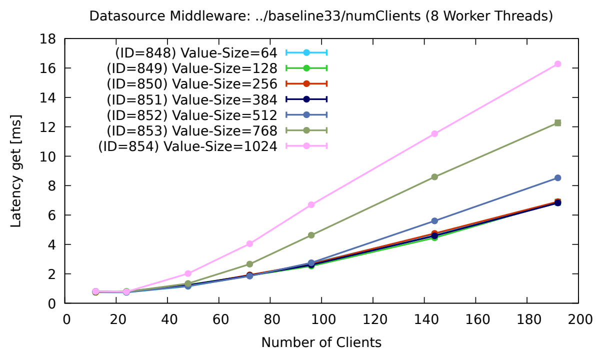
The results are explained by the interactive law $R = \frac{N}{X} - Z$. Figure 34a shows throughput and latency (for 192 client connections, 32 Worker Threads) as observed during the experiment. Figure 34b shows the derived response time using the interactive law with the parameters $N = 192, X = TP, Z = 3$. As one can observe, it conforms. Differences are assumed to be network jitter and or client load. The next section explains why the results are consistent with the previous experiments.

3.3.5 Explanation

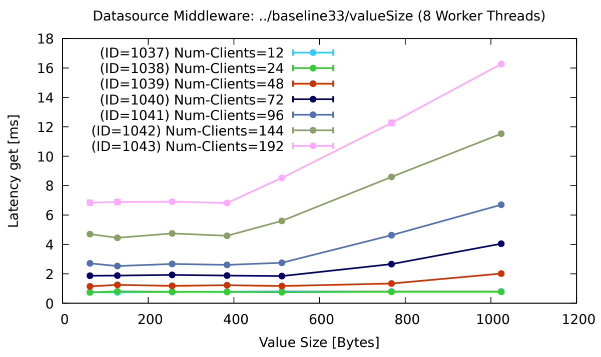
Saturation points: are given in the table 9.

	8 Worker Threads							32 Worker Threads							64 Worker Threads						
#c \d	64	128	256	384	512	768	1024	64	128	256	384	512	768	1024	64	128	256	384	512	768	1024
12	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
24	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
48	🟩	🟩	🟩	🟩	🟩	🟩	🟧	🟩	🟩	🟩	🟩	🟩	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟧	🟧
72	🟩	🟩	🟩	🟩	🟩	🟧	🟧	🟩	🟩	🟩	🟩	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟧	🟧	🟧
96	🟩	🟩	🟩	🟩	🟧	🟧	🟧	🟩	🟩	🟩	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟧	🟧	🟧	🟧
144	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟧	🟧	🟧	🟧
192	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟧	🟧	🟧	🟧	🟩	🟩	🟧	🟧	🟧	🟧	🟧

Table 9: Saturation Points Baseline 33

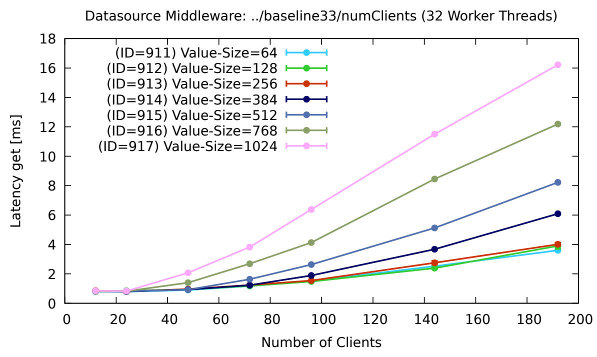


(a) latency [ms] vs Clients

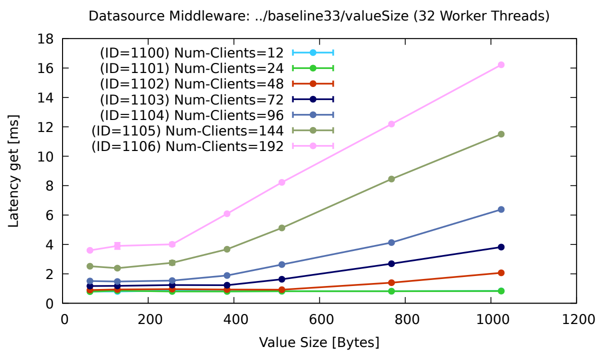


(b) latency [ms] vs Value Size

Figure 31: latency baseline 33 (8 worker threads)



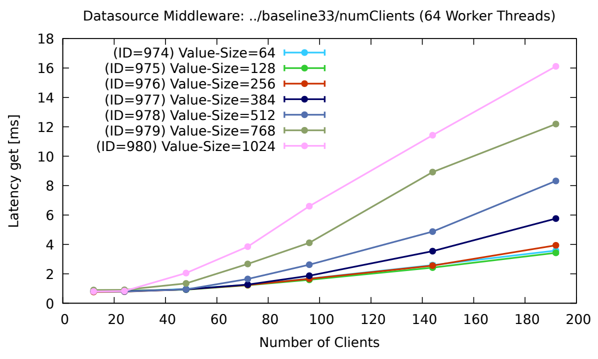
(a) latency [ms] vs Clients



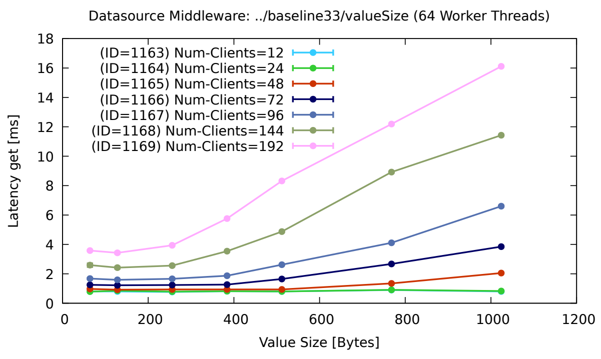
(b) latency [ms] vs Value Size

Figure 32: latency baseline 33 (32 worker threads)

Throughput and Response Time: table 6 (baseline 31) gives a lower bound for the needed number of clients to fully use the networks capacity. **Please compare 3.1.5 Explanation for general remarks.** Baseline 33 is very similar to baseline 31 (ignoring 64WT's for baseline 33 at the moment). We have the same network bottleneck (one memcached server) and we use the same number of clients. The only difference is that we use two middleware. This has some influence on the performance for two reasons. First we have doubled the number of worker threads. Therefore we can expect baseline 33 with 32 worker threads to have a similar performance than baseline 31 with 64 worker threads. Comparing figure 35 proofs this. We can observe, that the two plots match perfectly for configurations that are network bound. To explain the little gap between the two experiments (before the network capacity has been reached) we need to observe the following: if we compare the queue length on each middleware, without summing them together, we observe, that the queue length in baseline 33 is only half as long (per middleware) as in baseline 31. For itself this has not such a big influence, as the combined queue has the same length in both experiments. But it also means, that the net-thread has less work to do. As we know from baseline 32 there seems to be an upper bound of around 30'000 request each net-thread can handle. Now they only have to do half the work, giving this setting an advantage as long as it is not bound by the networks capacity. This performance advantage is also visible when we compare b33 32WT's and 64WT's. As opposed to b31, the system can easily handle 30'000 requests / sec and therefore make use of more worker threads. We see this in the plots. I assume I do not have to repeat all the other things (they have already been explained). Further insight of the experiment: At some point the system could benefit from

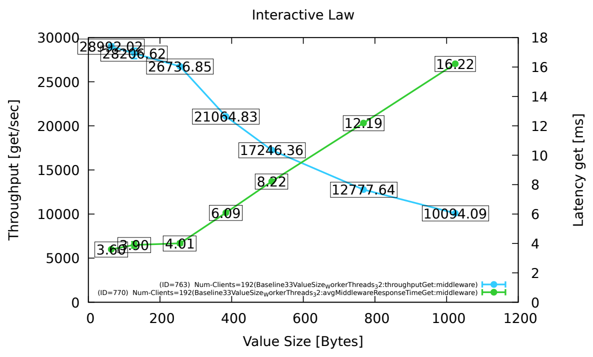


(a) latency [ms] vs Clients

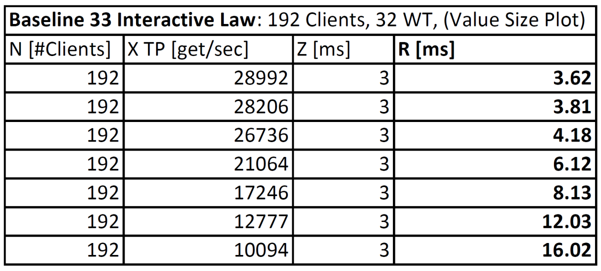


(b) latency [ms] vs Value Size

Figure 33: latency baseline 33 (64 worker threads)



(a) Blue TP, Green Latency



Baseline 33 Interactive Law: 192 Clients, 32 WT, (Value Size Plot)			
N [#Clients]	X TP [get/sec]	Z [ms]	**R [ms]**
192	28992	3	**3.62**
192	28206	3	**3.81**
192	26736	3	**4.18**
192	21064	3	**6.12**
192	17246	3	**8.13**
192	12777	3	**12.03**
192	10094	3	**16.02**

(b) Interactive Law: Derived Response Time

Figure 34: Interactive Law: baseline 33

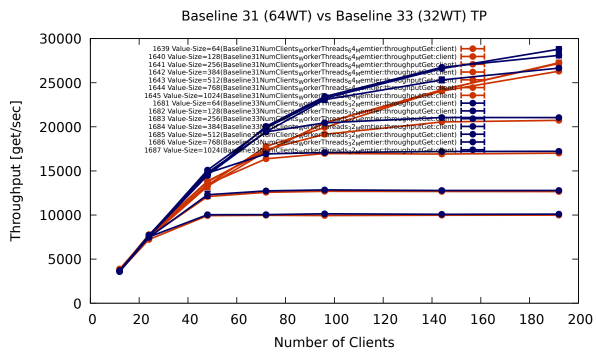
a second net-work thread. It is better to spend the resources for a second net-thread than to increase the number of worker threads (depends on setting, 30'000+ req/sec).

3.4 Two Middlewares, Three Server

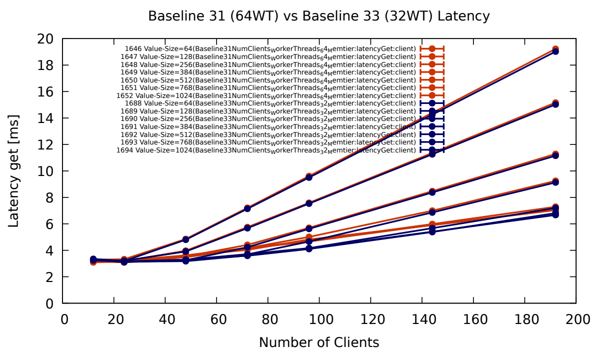
3.4.1 Setup

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance (CT)	1
Virtual clients per thread (VC)	2, 4, 8, 12, 16, 24, 32
Total client connections	**12, 24, 48, 72, 96, 144, 192**
Workload	get
Value Size (Bytes)	**64, 128, 256, 384, 512, 768, 1024**
Number of middlewares	2
Worker threads per middleware	8, 32, 64
Repetitions	3 (each 80s)

The setup is as follows: There are three clients each running two memtier instances. Each client instance is connected to a different middleware. The clients use a fixed number of 1 thread per memtier instance and a varying number of virtual clients per thread. There are two middleware used, connected to three memcached servers. The data is collected in 3 runs (each 80s). Compare Experimental Setup for details on how the experiments have been run. All experiments in this report use the same deployment. The network bottleneck in this



(a) Throughput (red=b31, blue=b33)



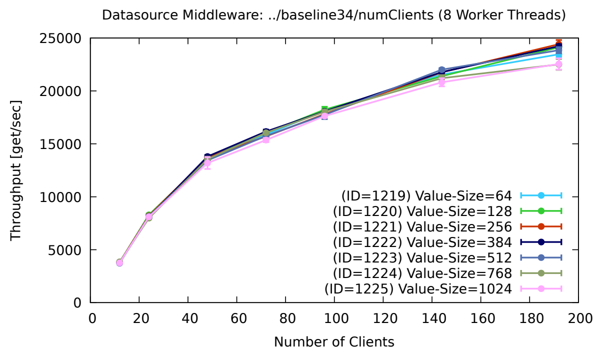
(b) Latency (red=b31, blue=b33)

Figure 35: Baseline 31 (64WT's) vs Baseline 33 (32WT's) - Memtier Data!

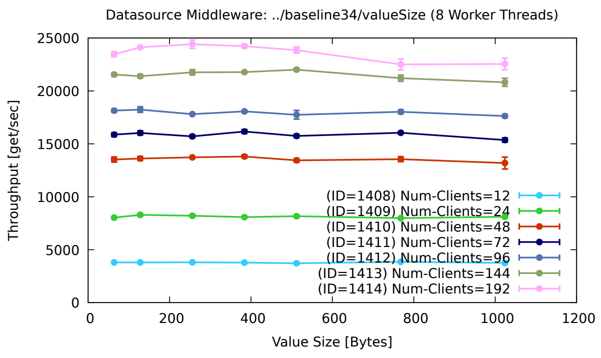
experiment (as observed with iperf): **Round trip time:** 0.9ms (+-0.2ms) and **Bottleneck Link Capacity** 3x Memcached server throughput TX 264 Mbits/s = 33'000 KByte/s.

3.4.2 Throughput

The plots in this section show the results collected with the experiment according to setup 3.4.1. On the y-axis is the throughput shown as get/sec. On the x-axis the number of clients (total client connections) or the value size in bytes. Figure 36, 37, 38 contain the results for 8, 32 and 64 worker threads. **Saturation points** are described in table 10.



(a) throughput [gets/sec] vs Clients



(b) throughput [gets/sec] vs Value Size

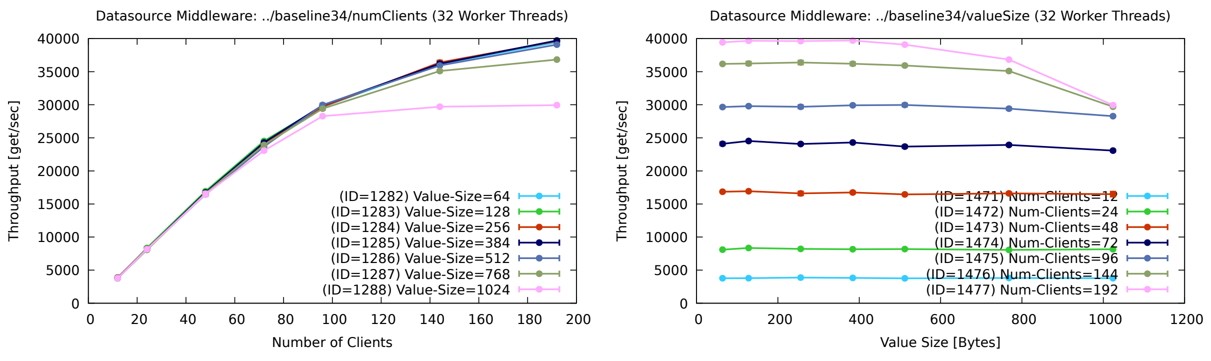
Figure 36: throughput baseline 34 (8 worker threads)

3.4.3 Response Time

The plots in this section show the results collected with the experiment according to setup 3.4.1. The y-axis shows the latency of a get in ms. On the x-axis the number of clients (total client connections) or the value size in bytes. Figure 39, 40, 41 contain the results for 8, 32 and 64 worker threads. **Saturation points** are described in table 10.

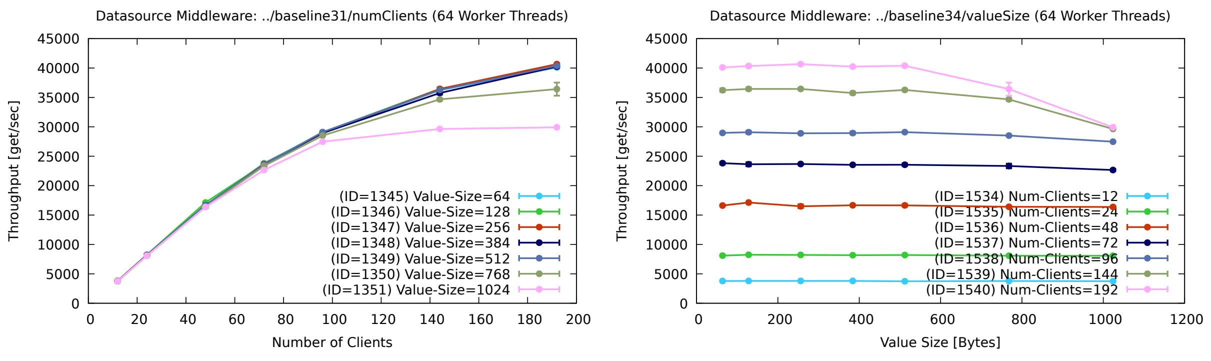
3.4.4 Result Analysis

The results are explained by the interactive law $R = \frac{N}{X} - Z$. Figure 42a shows throughput and latency (for 192 client connections, 32 Worker Threads) as observed during the experiment.



(a) throughput [gets/sec] vs Clients (b) throughput [gets/sec] vs Value Size

Figure 37: throughput baseline 34 (32 worker threads)



(a) throughput [gets/sec] vs Clients (b) throughput [gets/sec] vs Value Size

Figure 38: throughput baseline 34 (64 worker threads)

Figure 42b shows the derived response time using the interactive law with the parameters $N = 192, X = TP, Z = 2.95$. As one can observe, it conforms. Differences are assumed to be network jitter and or client load. The next section explains why the results are consistent with the previous experiments.

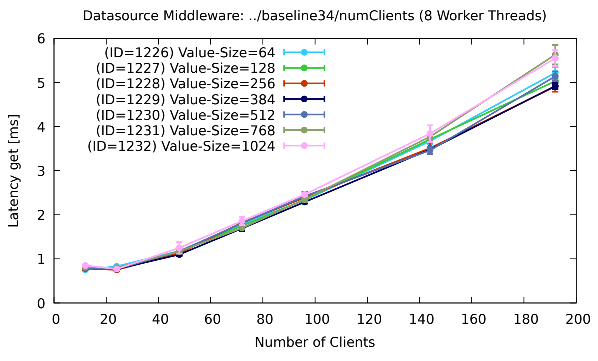
3.4.5 Explanation

Saturation points are given in the table 10.

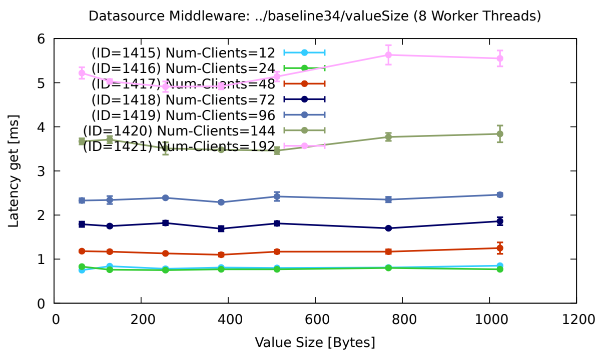
Throughput and Response Time: table 8 (baseline 32) gives a lower bound for the needed number of clients to fully use the networks capacity. **Please compare 3.1.5 Explanation for general remarks.** Baseline 34 is similar to baseline 32. We have the same network

	8 Worker Threads							32 Worker Threads							64 Worker Threads						
#c \d	64	128	256	384	512	768	1024	64	128	256	384	512	768	1024	64	128	256	384	512	768	1024
12	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
24	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
48	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
72	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
96	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
144	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟧
192	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟧	🟧

Table 10: Saturation Points Baseline 34

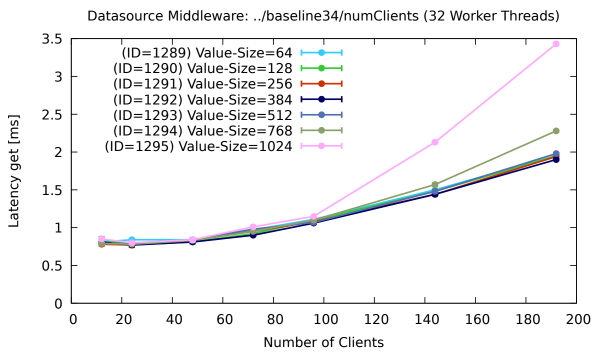


(a) latency [ms] vs Clients

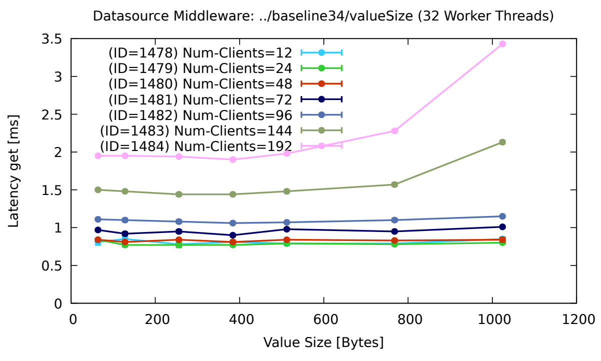


(b) latency [ms] vs Value Size

Figure 39: latency baseline 34 (8 worker threads)



(a) latency [ms] vs Clients



(b) latency [ms] vs Value Size

Figure 40: latency baseline 34 (32 worker threads)

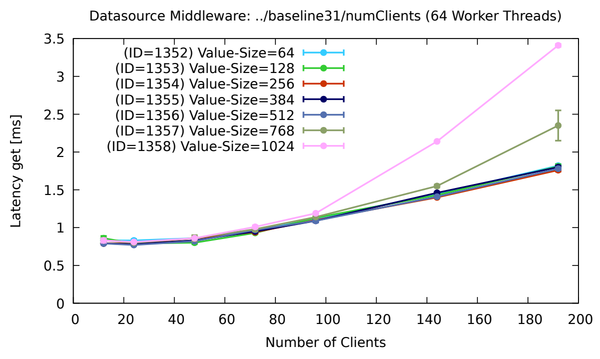
bottleneck. As outlined in 3.2.5 (also compare the table 8) we need a very high throughput (30'000+ req/sec) to fully use the network. With two middleware we can reach this throughput because we can use two net-threads (compare the discussion in the other parts). The table tells us, that we can reach the bottleneck capacity with the largest two value sizes. This is visible in the throughput plots 36a 37a 38a. The response time figures 39a 40a 40a show the expected increase in latency (compared to undersaturated lines) once a value size lines reaches the bottleneck capacity. Everything else has been outlined multiple times in all previous chapters.

3.5 Summary

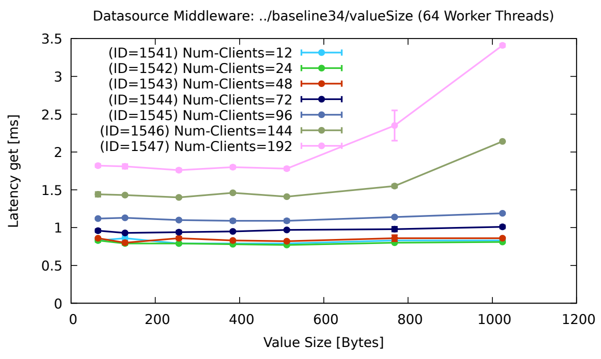
To fill out the table I fix the number of worker threads to be always equal to 64. The value size is fixed at 1024 bytes.

Maximum throughput for one and two middlewares.

	Throughput	Response time	Average time in queue	Miss rate
Section 3.1: Measured on middleware	9'916 [get/sec]	2.36 [ms]	0.32 [ms]	0%
Section 3.1: Measured on clients	9'928 [get/sec]	4.85 [ms]	n/a	0%
Section 3.2: Measured on middleware	26'349 [get/sec]	2.37 [ms]	1.09 [ms]	0%
Section 3.2: Measured on clients	25'834 [get/sec]	5.55 [ms]	n/a	0%
Section 3.3: Measured on middleware	9'991 [get/sec]	2.05 [ms]	0.25 [ms]	0%
Section 3.3: Measured on clients	9'941 [get/sec]	4.81 [ms]	n/a	0%
Section 3.4: Measured on middleware	27'471 [get/sec]	1.19 [ms]	0.32 [ms]	0%
Section 3.4: Measured on clients	27'402 [get/sec]	3.53 [ms]	n/a	0%

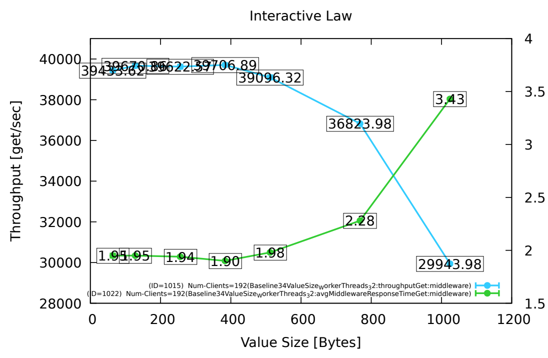


(a) latency [ms] vs Clients

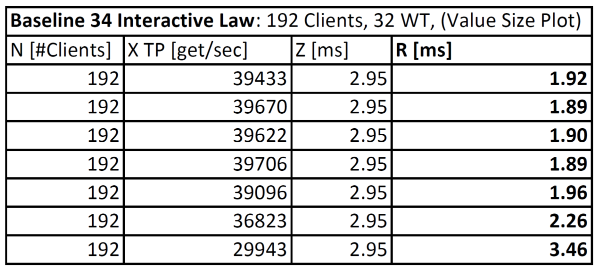


(b) latency [ms] vs Value Size

Figure 41: latency baseline 34 (64 worker threads)



(a) Blue TP, Green Latency



Baseline 34 Interactive Law: 192 Clients, 32 WT, (Value Size Plot)			
N [#Clients]	X TP [get/sec]	Z [ms]	**R [ms]**
192	39433	2.95	**1.92**
192	39670	2.95	**1.89**
192	39622	2.95	**1.90**
192	39706	2.95	**1.89**
192	39096	2.95	**1.96**
192	36823	2.95	**2.26**
192	29943	2.95	**3.46**

(b) Interactive Law: Derived Response Time

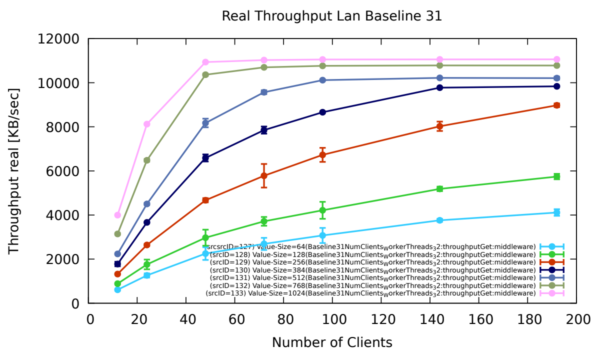
Figure 42: Interactive Law: baseline 34

3.5.1 Bottleneck Analysis

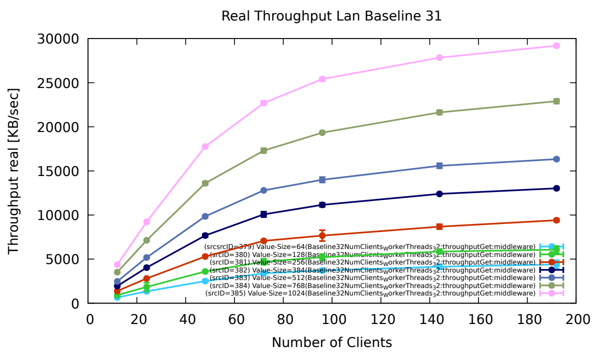
. The bottlenecks of the different setups have already been described in the previous 4 explanation sections. The same holds for the influence that different number of clients and value sizes have. Figure 44 real throughput presents the throughput achieved in the different experiments in KB/sec. Like in 2.3.1 Bottleneck Analysis they show the real bandwith as observed by the network (packet size = value size + 100 bytes). I give just a summary of the bottlenecks (as they have been described multiple times). A middleware has an upper bound of around 30'000 req/sec it can process (net-thread). The number of worker threads is important (compare the tables 6 8) to create enough load, this is especially true if we want to fully utilize the bandwidth of three memcached servers. Most of the time we do not reach the networks capacity because we simply don't have enough clients (this is not really a bottleneck). In 3.1.5 Explanation I outlined why we need so many clients, because the RTT (as observed by the client) is huge. To conclude: the limiting factor in section 3 is the increased latency which makes it necessary to use more clients to achive the same load. At some point the net-thread is a bottleneck. Once we reach the networks capacity, this is also a bottleneck.

3.5.2 One and Two Middleware Configurations

All details have already been discussed in the previous chapters, especially in 3.3.5 (comparison between baseline 31 and 33). I give a short summary. Two middleware means we have two

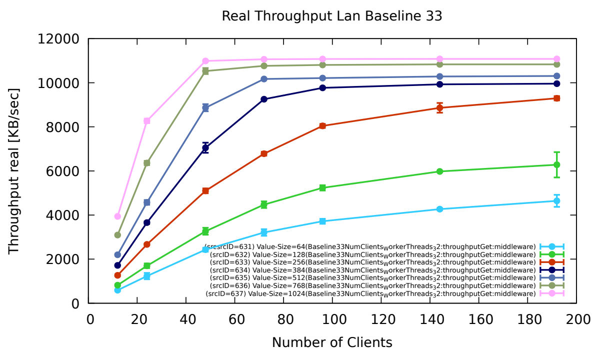


(a) Lan Throughput Baseline 31

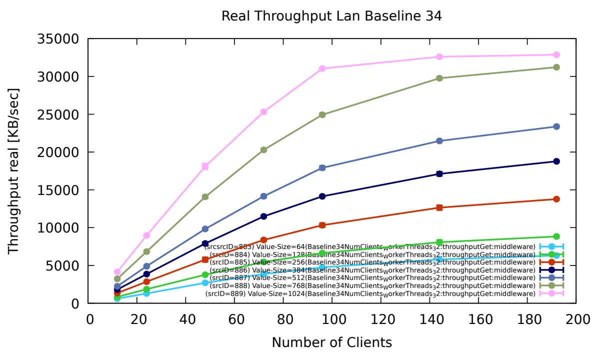


(b) Lan Throughput Baseline 32

Figure 43: real throughput



(a) Lan Throughput Baseline 33



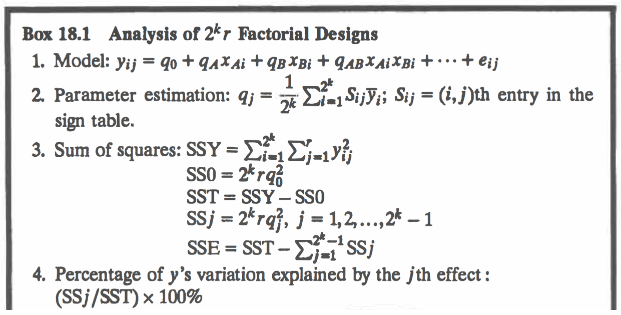
(b) Lan Throughput Baseline 34

Figure 44: real throughput

times the number of worker threads. For configurations where we need more than 64 worker threads (compare the two tables), this helps. But as outlined, it is not possible to increase the number of worker threads arbitrarily (even when we assume that the middleware can handle this), because we have the net-thread as bottleneck. With two middleware, we have therefore a twice as big throughput on the net-threads. This helps in cases where we need to process more than 30'000 req/sec.

3.5.3 One and Three Server Configurations

All details have already been discussed in the previous chapters. The most significant difference is the higher network capacity. In most cases (with the given parameter space) it is not possible to fully use the networks capacity. The reason is the huge number of client connections that are needed. If we use one middleware, it is not even possible to fully use the capacity with 1024b value sizes. We could argue that with three servers we can overcome the network bottleneck for the given value sizes. On the right side we see the cumulated values for the variation and the error.


Box 18.1 Analysis of $2^k r$ Factorial Designs

1. Model: $y_{ij} = q_0 + q_A x_{Ai} + q_B x_{Bi} + q_{AB} x_{Ai} x_{Bi} + \cdots + e_{ij}$
2. Parameter estimation: $q_j = \frac{1}{2^k} \sum_{i=1}^{2^k} S_{ij} \bar{y}_i$; $S_{ij} = (i,j)$th entry in the sign table.
3. Sum of squares: $\mathrm{SSY} = \sum_{i=1}^{2^k} \sum_{j=1}^{r} y_{ij}^2$
   $\mathrm{SS0} = 2^k r q_0^2$
   $\mathrm{SST} = \mathrm{SSY} - \mathrm{SS0}$
   $\mathrm{SS}j = 2^k r q_j^2,\ j = 1, 2, \ldots, 2^k - 1$
   $\mathrm{SSE} = \mathrm{SST} - \sum_{j=1}^{2^k-1} \mathrm{SS}j$
4. Percentage of y's variation explained by the jth effect: $(\mathrm{SS}j/\mathrm{SST}) \times 100\%$

Figure 45: 2k Formulas (Book)

4 2K Analysis

4.1 2K Model

I use the model as presented in the book *The Art of Computer Systems Performance Analysis* chapter 18 $2^k r$ factorial designs with replications [3]. I use the additive model, because the effects of factors are additive (insights for previous chapters) and the range of the covered values is not so large $y_{max}/y_{min} = 38'538/13'525 = 2.8$ (similar for latency). Figure 45 2k Formulas (Book) shows the formulas as presented in the book. Table 11 $2^k r$ analyse gives a compact overview of the analyse. The left upper part shows the sign table: I use S for server, MW for middleware and WT for worker threads. On the right upper part we see the three repetitions (per experiment) and the mean value. Everything is shown in blue for throughput and red for response time. The second part of the table shows the parameter estimation (Total, Total/8) (basically the summed product of the column above with the mean. The last to rows show the sum of squares SSj value and the percentage of variation that is explained by the effect.

									Throughput [get/sec]				**Response Time [ms]**			
Exp.	**I**	**S**	**MW**	**WT**	**S-MW**	**S-WT**	**MW-WT**	**S-MW-WT**	**y1**	**y2**	**y3**	**ymean**	**y1**	**y2**	**y3**	**ymean**
1	**1**	**-1**	**-1**	**-1**	**1**	**1**	**1**	-1	13915	13955	13526	13799	10.82	10.79	11.31	10.97
2	1	1	-1	-1	-1	-1	1	1	15169	13904	14834	14636	9.77	11.01	10.13	10.30
3	1	-1	1	-1	-1	1	-1	1	19799	19713	19928	19813	6.73	6.86	6.72	6.77
4	1	1	1	-1	1	-1	-1	-1	23515	24034	23624	23724	5.18	5.04	5.10	5.11
5	1	-1	-1	1	1	-1	-1	1	25531	25160	25455	25382	4.43	4.71	4.60	4.58
6	1	1	-1	1	-1	1	-1	-1	27471	26668	27102	27080	3.84	4.02	3.97	3.94
7	1	-1	1	1	-1	-1	1	-1	26856	26759	26957	26857	3.96	4.04	3.89	3.96
8	1	1	1	1	1	1	1	1	38313	38539	38280	38377	1.98	2.01	2.00	2.00
Total:	189668	17966	27875	45724	12896	8470	-2331	6748								
Total/8:	23709	2246	3484	5716	1612	1059	-291	844	Effect Throughput							
Variation:		**9.3%**	**22.3%**	**60.0%**	4.8%	2.1%	0.2%	1.3%	99.88% (Error 0.12%)							
SSj	13490273693	121045446	291382764	784018226	62360779	26903868	2037422	17076736								
Total:	47.64	-4.93	-11.96	-18.67	-2.32	-0.27	6.84	-0.33								
Total/8:	5.95	-0.62	-1.50	-2.33	-0.29	-0.03	0.85	-0.04	Effect Latency							
Variation:		4.3%	**25.0%**	**61.0%**	0.9%	0.0%	**8.2%**	0.0%	99.50% (Error 0.50%)							
SSj	851.0560	9.1302	53.6488	130.7350	2.0201	0.0268	17.5196	0.0418								

Table 11: $2^k r$ analyse

4.2 Throughput and Response time Impact

Remark: all throughput values are in [get/sec], all response time values in [ms]. The average **throughput** is 23'709. The throughput is mostly affected by the number of WT's. The parameter makes a difference of 5'716 and explains 60% of the variation. The second most important parameter is the number of MW's, its effect is 3'484 an explains 22.3% of the variation. A minor impact has the server parameter, it explains 9.3% of the variation. The impact between the parameters is low (below 5%). The average **response time** is 5.95, and the biggest influence on it has again the number of WT's. It explains 61% of the variation. A negative effect is good (for response time), as we want it to be low. The MW parameter is again the second most important, explaining 25% of the variation. The interaction between those two parameters is the thirt biggest source to explain the variation (8.2%).

4.3 Comparing Model Results with Experimental Results

The model explains the system very well. Compare the table 6 for value sizes 256b. For max throughput we need at least 27 threads, this explains why the number of WT's parameter is the most important one. If we use just 8, we have a mismatch factor of 3.375. If we use two middleware (each with 8 threads), we still have a mismatch factor of 1.6875. As long as we cannot increase the load, a second memcached server is useless. And the fastest way to increase the load (in the given system), is to use more worker threads. We further know that one memcached server is capable to serve 35'000 requests. To create so many request, we need as explained enough worker threads and then to overcome the middleware's net-thread limit of 30'000 req/sec a second middleware. This reasoning gives the same ranking of the parameters as the 2k analyse. We first increase the number of worker threads to 32, then we take two middleware and get 64 threads and more importantly a high enough throughput for the net-thread to finally utilize more than one memcached server. This is also visible in the plots in section 3. For example b31 (1S,1MW,32WT) has a higher tp than b33 (1S,2MW,16WT) (WT more important than MW). B31 (1S, 1MW, 32WT) has a higher tp than b32 (3S,2MW,8WT) (WT more important than S). Comparing B32 with B34 proofs that to use more than 1 server we first should increase to 32WT's (b32 32WT is better than b34 8WT) and then add a second MW. Finally a comparison between b33 (1S,2MW,8WT) and b32 (3S,1MW,8WT) proofs the last statement (MW more important than S).

5 Queuing Model

I fix the data size to be 1024b for M/M/1 and M/M/m and 256b for the queueing model. Everything that is needed has already be collected. Compare chapter 1. I measure the inter arrival time, the queue waiting time, the time spent in the worker thread and the time spent waiting for memcached's response. The service and arrival rate are simply $1/avgWTResponseTime$ respective $1/interArrivalTime$. The response time used in this report was always a combination of the queue waiting time and the worker thread response time. Chapter 1 describes this in detail. To sum up: the net-thread adds a timestamp to the QueueMetaData object (the client connection) once the client gets added to the request queue. The worker thread creates multiple timestamps (when it takes the object(*), when it forwards it to memcached, when the response has been arrived, when the response is sent back to the client). The last timestamp is again stored in the QueueMetaData object. The net-thread can use this timestamp the next time the client sends a new request to calculate the client think time. This means the arrival rate and the client think time are calculated inside of the net-thread, the rest in the worker thread. (*) The length of the queue can also we observed at this point.

5.1 M/M/1

The input parameters for the model are the inter arrival time (respectively the arrival rate λ) and the service time (respectively service rate μ). They have been measured as outlined in 5 Queuing Model. Table 12 M/M/1 model shows the two input parameters (λ, μ), the queue length and waiting time in blue (meaning the have been measured). The orange values are the ones that have been derived with the given model and the input parameters. The input parameters (and the measurements for comparison) have been chosen in the following way for each WT combination: Find **top throughput** in plots and take the corresponding **number of clients** to get the input values from the plot. For queue length and waiting time this can then simply be read out of the plots. For λ I took the arrival rate as observed by the net-thread. This is exactly what the model needs. For μ I had to multiply the value with the number of worker threads, because the service time (and therefore service rate) is given as an average over all threads. To get back the real service rate, this additional step is needed to get an approximation of the service rate. Remark: The values are from the plots implying the represent the average as observed over three runs. The stability condition is fulfilled as the traffic intensity p is less than 1. We observe that the model is a bad fit. The difference between the effective queue length and waiting time increases with the number of worker threads. But this is as expected. From the previous experiments we know that there is a bottleneck in the net-thread (30'000 req/sec). This cannot be modelled and therefore the model underestimates the queue length. This has a direct influence on the queue waiting time. The model captures this relation. But as it uses the wrong queue length estimation, this value is also too small.

	Arrival Rate [1/ms]	**Service rate [1/ms]**	**Service Time [ms]**	**Traffic Intensity**	**Queue Length**		**Queue Waiting Time [ms]**	
	λ	μ	$E[n_s]$	$p = \lambda/\mu$	**real**	$E[n_q] = p^2/(1-p)$	**real**	$E[w_q] = E[n_q]/\lambda$
WT=8	15.00	15.69	0.51	0.96	95.00	20.90	6.30	1.39
WT=32	26.50	32.00	1.00	0.83	100.00	3.99	3.00	0.15
WT=64	27.50	42.11	1.52	0.65	86.00	1.23	1.95	0.04

Table 12: M/M/1 model

5.2 M/M/m

The input parameters are the same as described in 5.1 M/M/1, except that this time the service rate is given as observed during the experiments (without multiplying it with the number of threads). In addition we have the parameter m, which can be identified as the number of WT's. In the model, m is used to define the number of identical servers, all waiting for jobs, and each with a service rate of μ jobs per unit time. This is exactly what the worker threads do. Table 13 M/M/m model shows the evaluation of the model according to the formulas in the book [3]. We see the same results as in 5.1 M/M/1, the model is again not a good fit (for the same reasons). We have to model the service rate of the net-thread to get better accuracy. This is what we do in the next and last section of this report.

	Arrival Rate [1/ms]	**Service rate [1/ms]**	**Service Time [ms]**	**Traffic Intensity**	**Queue Length**		**Queue Waiting Time [ms]**	
m	λ	μ	$E[n_s]$	$p = \lambda/(m\mu)$	**real**	$E[n_q]$	**real**	$E[w_q]$
WT=8	15.00	1.96	0.51	0.96	95.00	18.86	6.30	1.77
WT=32	26.50	1.00	1.00	0.83	100.00	1.07	3.00	0.04
WT=64	27.50	0.66	1.52	0.65	86.00	0.00..	1.95	0.00..

Table 13: M/M/m model

5.3 Network of Queues

I use a closed queueing network, as the jobs in the system keep circulating from one queue to the next. The number of jobs is constant. As the system is job flow balanced, this means the number of arrivals (A) is equal to the number of completions during an observed period, i.e. A=C (therefore I do not show arrival rate in the table). I model the network as outlined in 2 at the bottom. This means I use two queues. The first one is the connection queue, from it the net-thread checks the clients and forwards them to the second queue (if they want to send). The second queue is consumed by worker threads. I model the net-thread as one device and all worker threads as one server (including memcached). After the server, the responses should go back to the clients (another device) and then again into the connection-queue. As we are not interested in bottlenecks on the client side, I will not analyse this part. I just want to point out, that this can be modelled, as the client think time is known. I further decided to abstract the memcached server away and include it into the server device (where also the worker threads are). I assume this is okay for two reasons: Its a small task (20 pts) and the memcached servers aren't that much utilized in this baseline. For 8WT and 32 WT we observe a response time that is basically the networks RTT. For 64WT it is a bit higher, but still very small. Table 14 Queueing Network show the results. The input parameters for the model are blue. Throughput is the same for all devices, as every request has to flow through each of them. The service time for worker threads (S_{WT}) is from the plots I created. The service time for the net-thread (S_{WT}) is equal to $1/30'000$. This value corresponds to the maximum observed throughput for a net-thread in all experiments (compare 3.2.5 Explanation (pts 20)). It is assumed, that this is a constant, as the CPU's on the middleware are never fully utilized. This means the service rate for the net-thread is also always 30 req/ms (I do not show it in the table). As we have multiple net-threads (and worker threads) the busy time has to be corrected by that factor. The model shows the expected utilization for the individual components. For WT=8 we see the very high utilization of the worker threads. This can be seen in the plots of b34, we can increase the tp with more worker threads. The increased service time by using WT=64 instead of WT=32 is a problem in the setup. We have not enough clients to fully use the systems capacity. The response time is for both configurations the same. The difference is, that in the WT=64 case fewer clients are waiting in the queue. They are sooner inside of a worker thread (because there are more of them), and therefore forwarded to a memcached server. But as the memcached servers (and the network) are limited (they have for both WT configurations the same tp), the service time increases inside of the worker thread. At the same time we can observe a smaller queue length and queue waiting time (the clients just wait in another position in the system).

	observation time	**completions**	**Throughput**	busy time	**Service Time**	Utilization	busy time	**Service Time**	Service Rate	**Utilization**
	T [ms]	**C**	$X = X_i$ **[get/sec]**	$B_{net} = S_{net}/(C * 2)[ms]$	$S_net[ms]$	$U_{net} = B_{net}/T$	$B_{WT} = S_{WT}/(C * WT)[ms]$	$S_{WT}[ms]$		$U_{WT} = B_{WT}/T$
WT=8	1000	25000	25000	416.67	0.033	42%	937.5	0.6	13.33	94%
WT=32	1000	40000	40000	666.67	0.033	67%	562.5	0.9	35.55	56%
WT=64	1000	40000	40000	666.67	0.033	67%	375	1.2	53.33	38%

Table 14: Queueing Network

References

[1] iperf - https://github.com/esnet/iperf.

[2] NMON - http://nmon.sourceforge.net/pmwiki.php.

[3] ”The Art of Computer Systems Performance Analysis” - Raj Jain Wiley Professional Computing, 1991.