Advanced Systems Lab (Fall'16) – First Milestone

Name: ***Rami Khalil***
Legi number: ***16-932-568***

Grading

Section	Points
1.1	
1.2	
1.3	
1.4	
2.1	
2.2	
3.1	
3.2	
3.3	
Total	

1 System Description

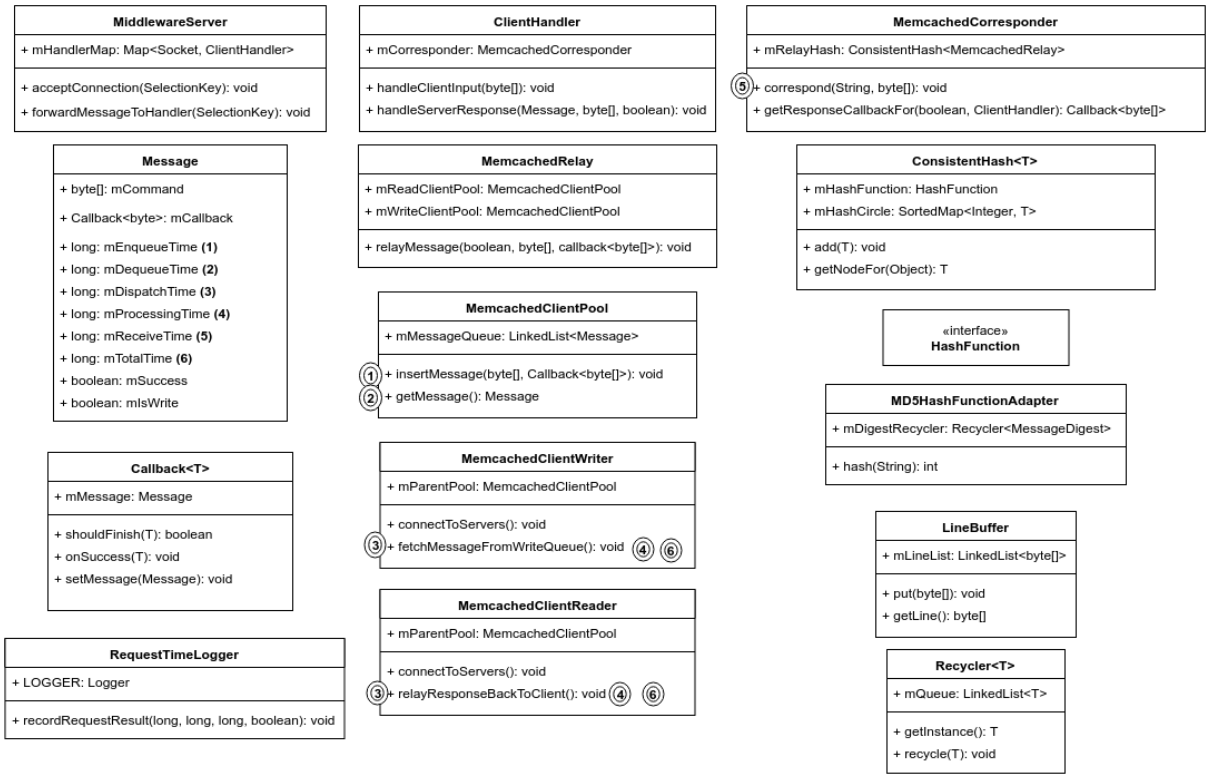

Figure 1: System Diagram

1.1 Overall Architecture

The MiddlewareServer class listens for and accepts connections, then creates a ClientHandler object that is responsible for each connection. The ClientHandler class is designed to keep track of a client's state and handle reading its input command and responding to it with memcache output. These two classes are responsible for managing middleware clients and their connections. Communication between these classes and Memcache servers is facilitated by the MemcachedCorresponder class.

The MemcachedCorresponder keeps track of each MemcachedRelay using a ConsistentHash object. When the Corresponder receives a message from a ClientHandler, it extracts the key from the message, uses the ConsistentHash to look up the relay responsible for that key, and forwards the message to the relay.

A MemcachedRelay is an object that is designed to be in charge of relaying messages to a pair of read and write queues that are primarily made for one server. Messages that are received by the relay from the corresponder are enqueued in either the read queue or write queue depending on their types.

The queue structures inside a MemcachedRelay are of type MemcachedClientPool. The client pool instantiates MemcachedClientReader or MemcachedClientWriter objects depending on whether it is of type read or write. These reader and writer objects dequeue jobs from the client pool and process them. The jobs are of class Message, which is a datastructure designed to keep track of all the timestamps related to the processing of the message, the message itself, and a Callback that is to be executed on the responses memached servers give after receiving this message.

The times collected are marked with numbers in figure 1. The times are $T_{EnqueueTime}$,

$T_{DequeueTime}$, $T_{DispatchTime}$, $T_{ProcessingTime}$, $T_{ReceiveTime}$, $T_{TotalTime}$, with time spent between receiving a request from client and sending final response being equal to $T_{TotalTime}$, time each request spends in a queue equal to $T_{DequeueTime}$ and time between sending the request to the server and receiving the answer equal to $T_{ProcessingTime}$.

1.2 Load Balancing and Hashing

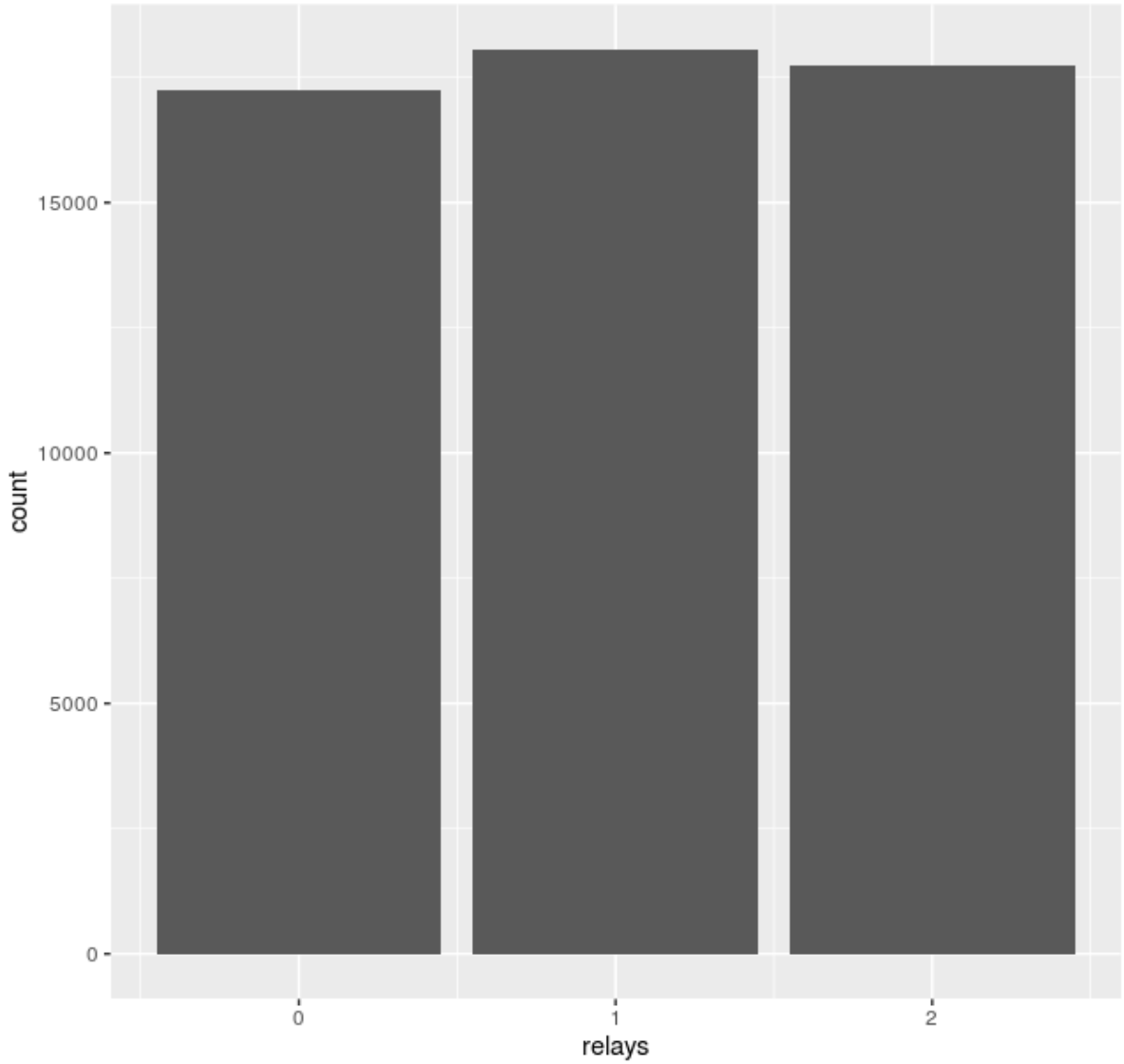

Figure 2: A bar plot of the request distribution over relays. A nice balance can be observed. Plotted using the *amended.logs.reqtimes* logs
, which were created by running a minitrace of the middleware augmented with an extra logging field for server id.

The ConsistentHash class utilizes a Java SortedMap to implement a circular map of datapoints which keeps track of virtual markers that point to our memcached servers. Its implementation is based on an article [1] by Tom White derived from the work done in [2].

"Consistent hashing" assigns a set of items, in our case value keys, to buckets, in our case memcached servers, such that each bucket receives roughly the same number of items[2]. By utilizing MD5, a hash function that maps objects in a uniform and independent way, in combination with the search tree, a consistent hash function based on this concept was implemented

that efficiently balances our workload.

In our implementation, we use the first 32 bits of the MD5 hash of an object as our hash [1]. Since MD5 is classified as a PRF [3], which implies that its values are uniformly distributed, then the truncation of its first 32 bits is considered to be almost uniformly distributed, which is sufficient for our needs.

We go on to use this uniformly distributed MD5 based hash to populate nodes in the ConsistentHash for our MemcachedRelay classes, each of which represents a Memcached server. For each relay, we create two hundred uniformly distributed nodes in the consistent hash[2]. Each node covers a continuous interval of the hash function's output, and with this replication [3], we increase the probability that the intervals are similar in size[2].

In order to decide which server is the main server for a given key, we calculate its truncated MD5 hash value and use it to look up the relay that is in charge of the interval that the value lies in[4]. This relay is then used as the main relay for retrieval of this key's value, and the base relay for writing to it as described in section 1.3. Assuming that the keys received from the middleware clients are not synthesized to cause skewness in MD5 calculations, then their target servers should be designated in a uniform fashion, balancing the workload among all available servers.

As a side note, we do not consider the computational costs of MD5, however, the interval look ups that we perform using Java's SortedMap execute in logarithmic time[4]. Moreover, much of the versatility the ConsistentHash class we have implemented is left unused, as we do not take advantage of the ability to remove servers from it when they are unavailable.

1.3 Write Operations and Replication

Writes are handled by the MemcachedClientWriter [5] class, which fetches write messages that waiting to be sent from its parent MemcachedClientPool [6]. The client writer maintains non blocking connections using Java NIO to the memcached servers it writes to.

The client writer first holds a lock on the pool's message queue before extracting a message from it. That lock is then released and becomes free to be held by the Relay for insertion. After extracting a Message [7] from the queue, the writer sends it to a number of memcached servers, depending on replication, and enqueues the Callback [8] associated with the message in its own queues to handle the responses from the servers.

For each memcached server, a queue of callbacks is kept. Callbacks are executed on the responses returned from the servers in order. The Callbacks contain a shouldFinish method, which return true iff this callback has processed the responses it needs to satisfy relaying a response for its Message.

Without replication, only a single queue is kept, where each callback is executed on each response received and a result is returned immediately after to the middleware client. However, with replication, callbacks are executed on each response returned from each server before a final response is sent to the middleware client. This means that the callback queues for each

[1] MD5HashFunctionAdapter.java: `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/src/hashing/MD5HashFunctionAdapter.java`

[2] This value is based on the empirical data presented in [1] which suggests that values in the range of a few hundred replicas achieve acceptable workload distributions.

[3] MemcachedCorresponder.java: `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/src/client/MemcachedCorresponder.java`

[4] ConsistentHash.java: `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/src/hashing/ConsistentHash.java`

[5] `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/src/client/MemcachedClientWriter.java`

[6] `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/src/client/MemcachedClientPool.java`

[7] `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/src/util/Message.java`

[8] `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/src/util/Callback.java`

server may not necessarily have the same state, but callbacks are always inserted into them in the same order.x

The latencies incurred by writes are mostly due to buffering of write commands and processing their server responses. Write messages are buffered until they are complete [9] before being forwarded to the middleware as to be able to maintain a mechanism that properly does writes in order. Moreover, the buffered commands are read and inspected for the key value and the byte size, which involves relatively expensive string manipulation. Interpreting server responses in order to decide what to send back to the client is estimated to be the second largest source of latency in the system. Again due to string manipulation, the callbacks for the write are estimated to take up a relatively significant amount of time to decide what the final response to the middleware client will be depending on the success of the command and how many servers the writer is communicating with.

The total processing time for write requests increases from an average of 3700us in the baseline (for 128 clients) to an average of 7400us in the middleware trace. With replication in mind, the delays caused by server response interpretation are amplified, since for a single request, multiple memcached server responses have to be read and interpreted before the request can be completely served by the middleware and receive a final response. According to the instrumentation logs, the mean $T_{ProcessingTime}$ is equal to 3101075ns (3101us), with a standard deviation of 4875887ns (4875us). This implies that half of the time spent by the middleware while processing write requests is to communicate with the memcached servers

It is important to note that the MemcachedClientWriter class contains the ServerResponseReader class, which is a Thread that manages the interpretation and relay of memcached server responses to middleware clients. This frees up the MemcachedClientWriter from having to synchronously wait for requests to be completed before fetching more from the message queue and sending them to the memcached servers. Only one MemcachedClientWriter exists for its MemcachedClientPool [10].

1.4 Read Operations and Thread Pool

Reads are handled by the MemcachedClientReader [11] threads that are running in the reading MemcachedClientPool. These threads attempt to obtain the lock on the message queue inside the pool in order to extract a Message [12] from it, after which the lock is released. Each thread processes only one message at a time to completion before attempting to dequeue another message for processing.

Each client reader thread holds only a single connection of its own to the server it reads from. After obtaining a Message in a thread-safe manner, the Reader relays it to its memcached server and waits for a response. Once the response is returned, the Message's callback is executed directly on it, which simply forwards the response back to the client.

Processing of read operations is relatively straight forwards and involves much less overhead and processing that processing write operations. Therefore, the latencies for reads are estimated to be lower than those for writes when using the middleware. However, the number of reader threads needs to be optimized to make up for their synchronous nature so that throughput is increased.

[9] ie the entire set command along with its value is sent to the middleware

[10] ie the system has only one writer thread

[11] `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/src/client/MemcachedClientReader.java`

[12] `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/src/util/Message.java`

2 Memcached Baselines

Number of servers	1
Number of client machines	2
Virtual clients / machine	1 to 64
Workload	Key 16B, Value 128B, Writes 1% [13]
Middleware	Not present
Runtime x repetitions	30s x 5
Log files	data.1/*, data.2/*

The experiment for running the baselines has been carried out on Azure. One server, using hardware predefined in Azure's 'Basic A4' [14] configuration, ran the main memcached server. Two client machines, with 'Basic A2' [15] configuration, ran memaslap clients simultaneously to take the measurements. All measurements presented in the upcoming subsections rely on the memaslap outputs produced by the two clients.

The total number of virtual clients ranges from 8 to 128 in steps of 8 (in steps of 4 for each client machine). For each number of virtual clients, the experiment was repeated 5 times, where the virtual clients generated load on the memcached server for 30 seconds using the workload configuration specified in 'smallvalue.cfg'. No middleware was present in these experiments as the virtual clients connected directly to the memcached server.

The scripts used to setup the machines can be found in the gitlab folder [16]. Since both client machines had synchronized clocks, the linux 'at' command was used to orchestrate synchronous memaslap workloads.

The resulting memaslap log data [17] was then processed using a small bash script [18] to extract the end statistics calculated by memaslap. These statistics were then aggregated and plotted using an R notebook 'asl.rmd' [19]. Corresponding log files from each machine were combined to produce coherent results, which were collected and plotted with their standard deviations in the following two plots.

[13] As predefined in memaslap-workloads/smallvalue.cfg

[14] 8 cores, 14.00 GB RAM

[15] 2 cores, 3.50 GB RAM

[16] https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/tree/master/report/baseline.scripts

[17] 1.smallvalues/* and 2.smallvalues/*

[18] `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/report/baseline.logs/tailor.sh`

[19] `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/report/asl.rmd`

2.1 Throughput

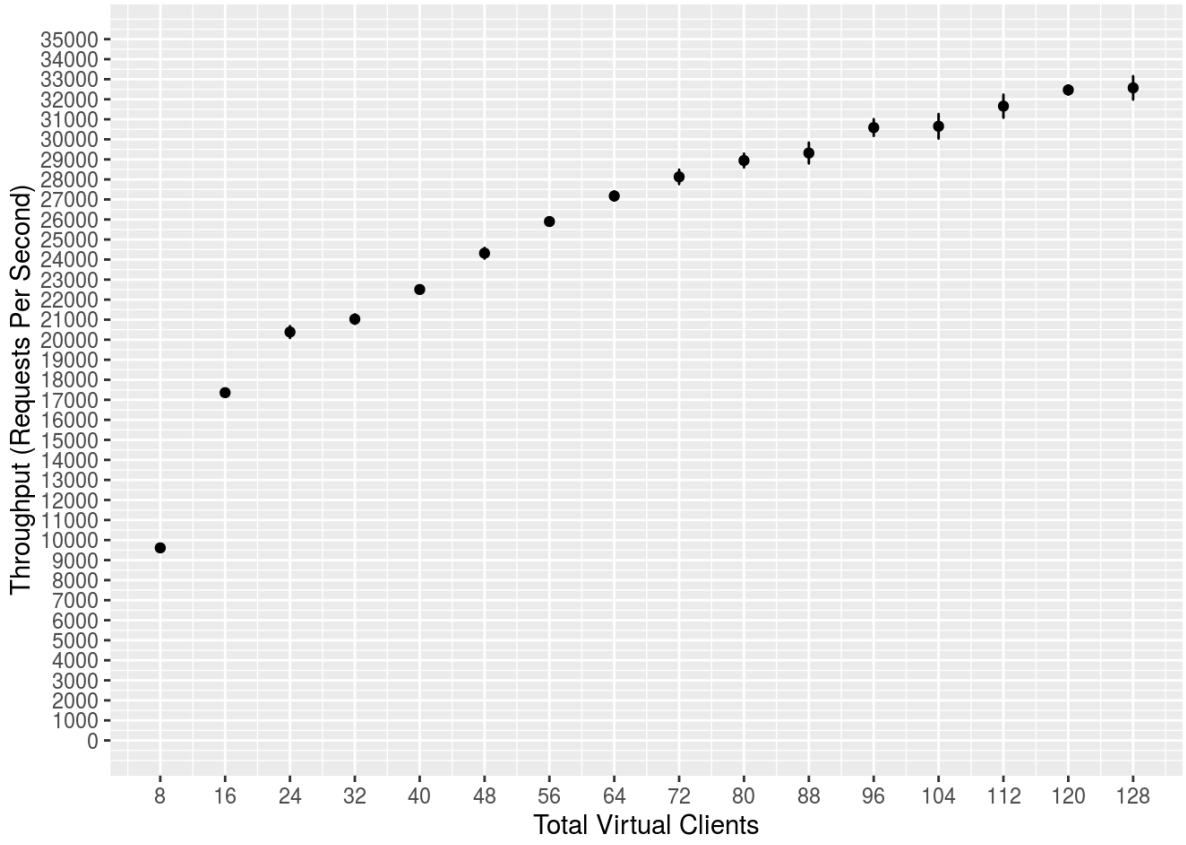

Figure 3: Average TPS (in **operations per second**) across different client counts with standard deviation. The standard error here is relatively small.

The throughput of the memcached servers seems to level off as the number of virtual clients increases. The low standard deviations observed in the experiments indicate that memcached is relatively stable when run under similar conditions. This plot was generated using R in the *asl.rmd*[20] notebook, a precompiled version of which can be found as *asl.html*[21]. The relevant sections of the log files are present under the *1.data* and *2.data* folders that are listed in the log files tables.

[20] `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/report/asl.rmd`

[21] `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/report/asl.html`

2.2 Response time

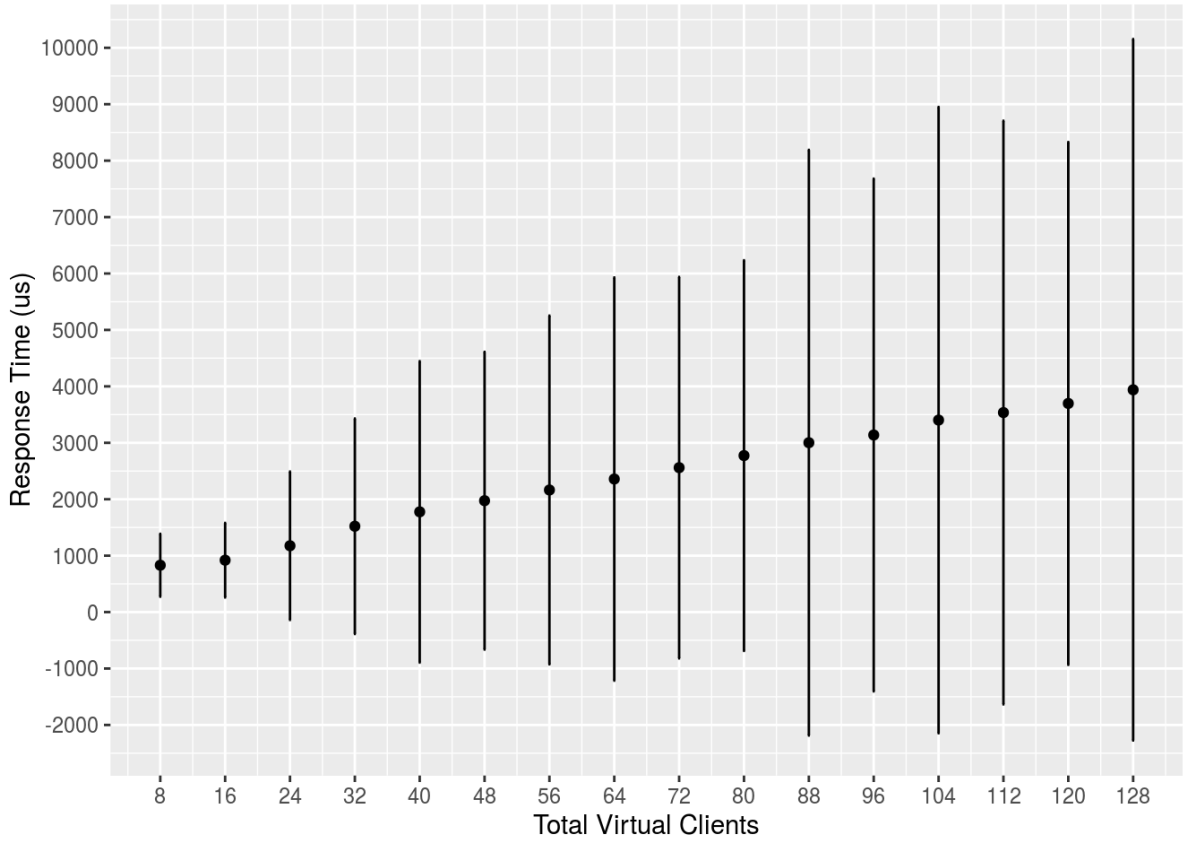

Figure 4: Average response times (in **microseconds**) across different client counts with standard deviation.

The mean response time for serving a request increases from approximately 800 micro-seconds to approximately 4000 micro-seconds as the number of virtual clients increases. The standard deviation, also, observes an increase from about 500 to about 6,000. This plot was generated using the same aforementioned methodology in section 2.1.

3 Stability Trace

Number of servers	3
Number of client machines	3
Virtual clients / machine	64
Workload	Key 16B, Value 128B, Writes 1% [22]
Middleware	Replicate to all (R=3)
Runtime x repetitions	1h x 1
Log files	relevant/*, trace.logs/*

The experiment for generating the stability trace was carried out on Azure's cloud platform. Three load generating client machines with a *Basic A2* hardware configuration ran memaslap and recorded its output to generate load on the system. Three memcached server machines with a *Basic A2* hardware configuration acted as the key-value stores. One machine with a *Basic A4* hardware configuration operated the middleware which received requests from the three load generators and forwarded them to the appropriate memcached server as determined by the middleware.

The relevant automation scripts used to setup the machines can be found in the *trace.scripts* [23] folder. The experiment was carried out over a period of one hour (3600 seconds) and the log data from all machines was collected in the *trace.logs* folder. The log data from the memaslap load generators was extracted to a separate folder [24] and processed using R to create a plot over time of the average throughput and a plot over time of the average response time with standard deviation. The system incurred no errors during its running time and was terminated gracefully at the end of the experiment. The memcached servers experienced no problems during the experiment. The memaslap logs showed no *get_misses* or any errors during the experiment.

An additional instrumentation log *logs.reqtimes*[25] was produced by the middleware, but was not inspected while writing this report.

[22] As predefined in memaslap-workloads/smallvalue.cfg

[23] `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/tree/master/report/trace.scripts`

[24] relevant/*

[25] `https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/report/trace.logs/logs.reqtimes`

3.1 Throughput

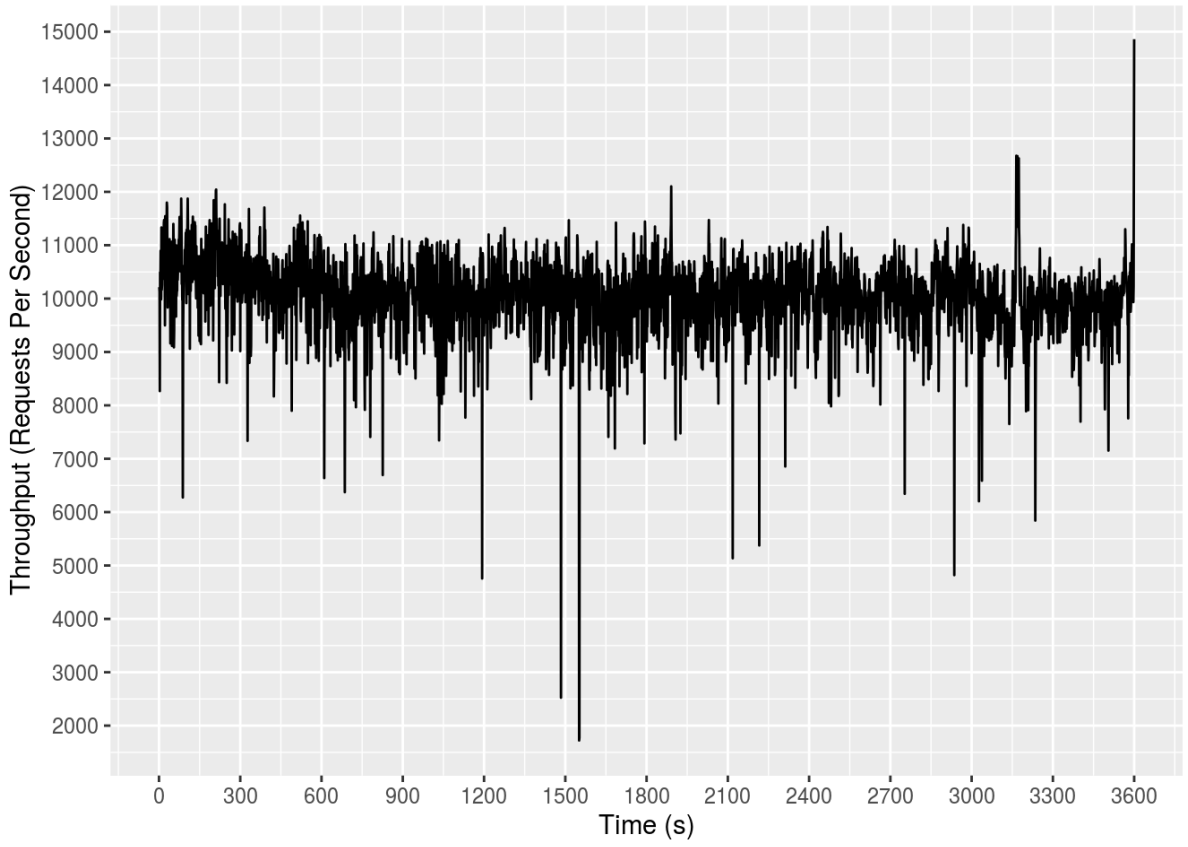

Figure 5: TPS (in **operations per second**) over time (**seconds**) while running the middleware for one hour.

The plotted values of throughput are the periodic values over 1 second observed during runtime. Standard error is not relevant in this plot as the measured values are a matter of fact for this single running instance.

The graph indicates that the system operates with a throughput that is in the range of 10,000 operations per second and suffers no gradual degradation of average performance during its run-time. However, some periodic negative spikes in throughput can be clearly observed. The cause of these spikes is unknown, however, they seem to leave no permanent impact on the system after occurring, and as the middleware produced no errors during this experiment, these spikes are not expected to be correlated with any malfunctions. These spikes could possibly be indicators of when the JVM garbage collection process takes place.

This plot was generated by aggregating the throughput values from the memaslap logs, under *relevant/**, of the three load generators using the aforementioned R notebook.

3.2 Response time

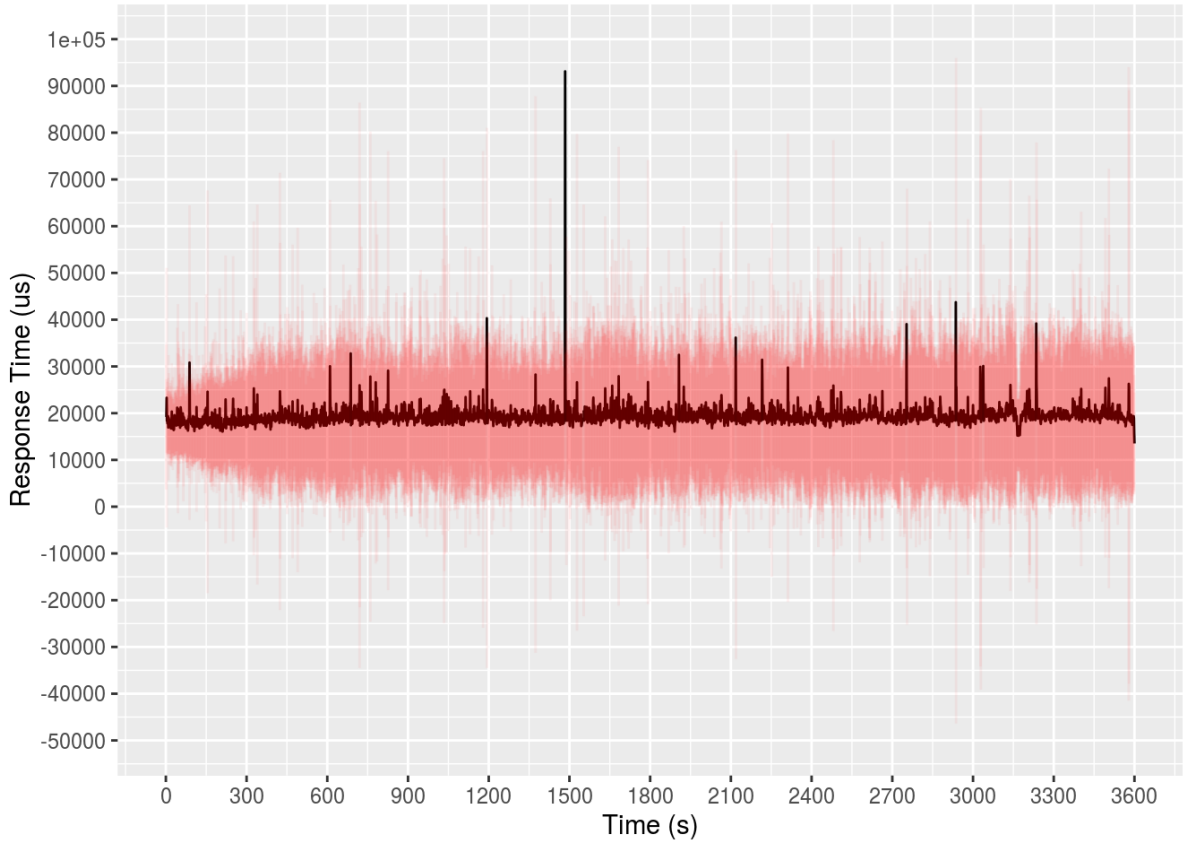

Figure 6: Average request response time (in **microseconds**) over time (**seconds**) with standard error.

The mean response time seems relatively stable overtime, with no observable trends. However, there are periodic spikes in the mean that cause increased delays of about 40,000 micro-seconds. There is also a single delay that increased response time to about 90,000 us.

The faint red around the the line graph is meant to give an indicator of the standard deviation values of response time. The exact values of standard deviations at a point in time can be found in the log files under *relevant/**. From the plot, it can be seen that the standard deviation values per 1 second periods vary often during the experiment.

The mean response time values, with the exception of the spikes, are centralized around approximately 20,000 micro-seconds. This indicates that the system does not suffer any permanent degradation of performance with time.

3.3 Overhead of middleware

Value	Without Middleware	With Middleware	Approx. Overhead
TPS Mean	33,000 ops/s	10,000 ops/s	-23,000 ops/s
Response Time Mean	3,800 us	20,000 us	+16,000 us
Response Time Std Deviation	6,000 us	15,000 us	+9,000 us

No expectations were previously set for the performance of the middleware based solely on the results of the baselines.

The middleware is observed in the trace plots to add a significant amount of overhead to the key-value store system, throttling the throughput to nearly one third, increasing the response times fourfold, and adding to their variance by about 9,000 us.

These results point to the fact that running this middleware as a front for a set of memcached servers under the configuration of full replication will come at significant costs of system performance and response time certainty.

However, the configurations of both experiments (baseline vs trace) are different, and comparing them in order to gain insight into the overhead will only lead to a rough idea of the difference in performance. The clients in the baseline had to deal with only one memcached server, while the middleware in the trace is dealing with three.

Logfile listing

Short name	Location
1.smallvalues/*	https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/tree/master/report/basel
2.smallvalues/*	https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/tree/master/report/basel
data.1/*	https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/tree/master/report/basel
data.2/*	https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/tree/master/report/basel
relevant/*	https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/tree/master/report/trace
trace.logs/*	https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/tree/master/report/trace
logs.reqtimes	https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/report/trace
amended.logs.reqtimes	https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/blob/master/report/trace

References

[1] Tom White: Consistent Hashing http://www.tom-e-white.com/2007/11/consistent-hashing.html

[2] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (STOC '97). ACM, New York, NY, USA, 654-663. DOI=http://dx.doi.org/10.1145/258533.258660

[3] Pseudorandom Function Family https://en.wikipedia.org/wiki/Pseudorandom_function_family

[4] TreeMap (Java Platform SE 7) http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html