Advanced Systems Lab (Fall'16) – First Milestone

Name: ***Iveri Prangishvili***
Legi number: ***15-926-140***

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

1.1 Overall Architecture

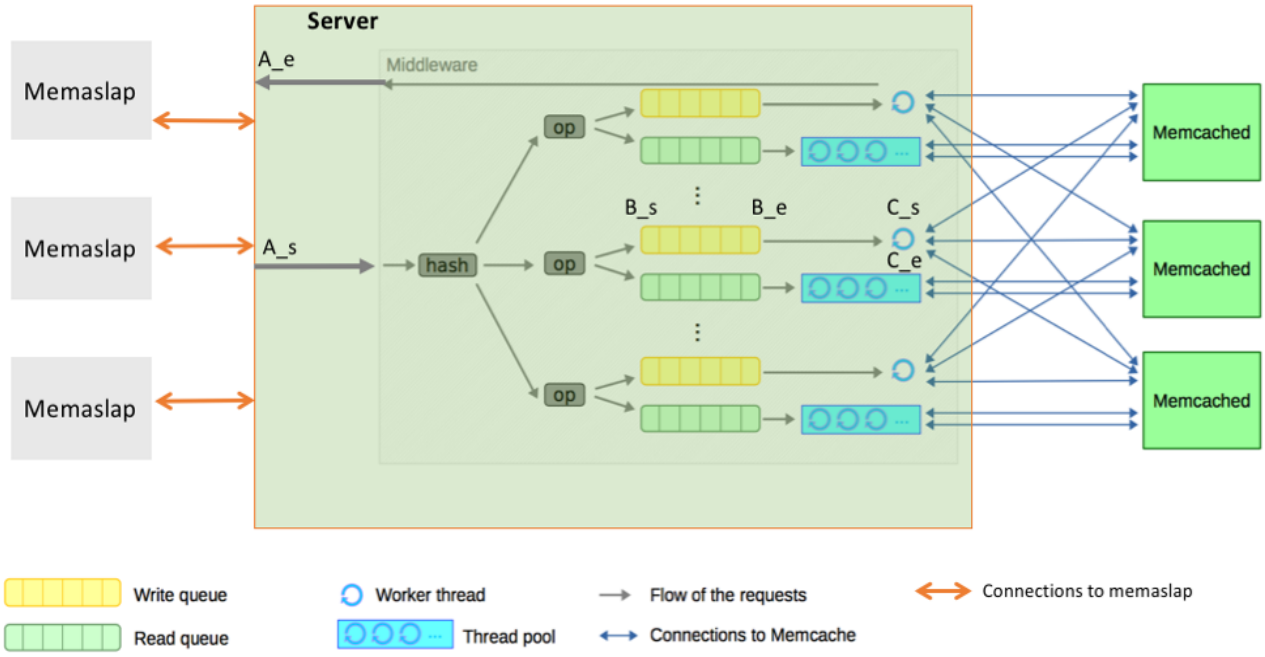

Figure 1: Abstract Architecture

The abstract architecture has been implemented in terms of the following classes:
1. **Server**[1] - handles communication with memaslap, Creates an instance of **Middleware**[2] class and forwards request data to it for further processing. Also handles part of code instrumentation.
2. **Middleware**[2] - Parses the request based on command {get, set, delete}, calls a method from **ConsistentHash**[3] class to select a memcached server (that will process the request) and pushes the request to the relevant queue based on the command and selected server.
3. **SyncClient**[4] - Establishes a synchronous connection to specified memcached server. Continuously pulls requests from the associated queue, sends it to the memcached server, waits for the response and sends it back to the memaslap through **Server**[1] class method.
4. **AsyncClient**[5] - Establishes asynchronous connection to specified memcached server and to replication servers (if replication is on). Pulls continuously from associated queue and sends data to memcached server or servers during replication. Once memcached sends response back it processes it and sends to memaslap through **Server**[1] class method.
5. **ConsistenHash**[6] - Implements consistent hashing, uses MD5 hash function to hash memcached severs and put them on a circle. Implements method to hash an input key and locate nearest hash corresponding

[1] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/Server.java
[2] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/Middleware.java
[3] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/ConsistentHash.java
[4] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/SyncClient.java
[5] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/AsyncClient.java
[6] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/ConsistentHash.java

to a server (clockwise). Implements same functionality for the replication case, where it returns specified number of consecutive servers on a circle (clockwise).
6. **ManageQueue**[1] - Holds get and set queue and holds a reference to an instance of **AsyncClient**[2] class. Is used by **Middleware**[3] class to manage internal architecture of delegating requests based on specific memcached servers. Uses **AsyncClient**[2] reference to wake up its selector, so that it can pull from the queue and process the request.
7. **RequestData**[4] - Holds an information on a request. The data of the request, the instrumentation associated with each request and the response data from memcached. If the replication is on, RequestData holds replication server addresses and also maintains the counter for number of replications performed for its instance.
8. **SocketChangeRequestInfo**[5] - Used in Server class for managing interest operations of selection keys. Holds information on socket channel, type of operation (registration, interest operations change) and new interest operation code.
9. **MyCustomFormatter**[6] - Custom formatting class used for the logging to enforce a csv format of code instrumentation.

Code instrumentation point references in abstract architecture (see Figure 1)

A_s	A_e	B_s	B_e	C_s	C_e
$T_{requestReceived}$	$T_{responseSent}$	$T_{enqueue}$	$T_{dequeue}$	$T_{requestSentToServer}$	$T_{responseReceivedFromServer}$

T_{mw}	$T_{responseSent} - T_{requestReceived}$
T_{queue}	$T_{dequeue} - T_{enqueue}$
T_{server}	$T_{responseReceivedFromServer} - T_{requestSentToServer}$
$F_{success}$	True/false

[1] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/ManageQueue.java
[2] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/AsyncClient.java
[3] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/Middleware.java
[4] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/RequestData.java
[5] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/SocketChangeRequestInfo.java
[6] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/MyCustomFormater.java

1.2 Load Balancing and Hashing

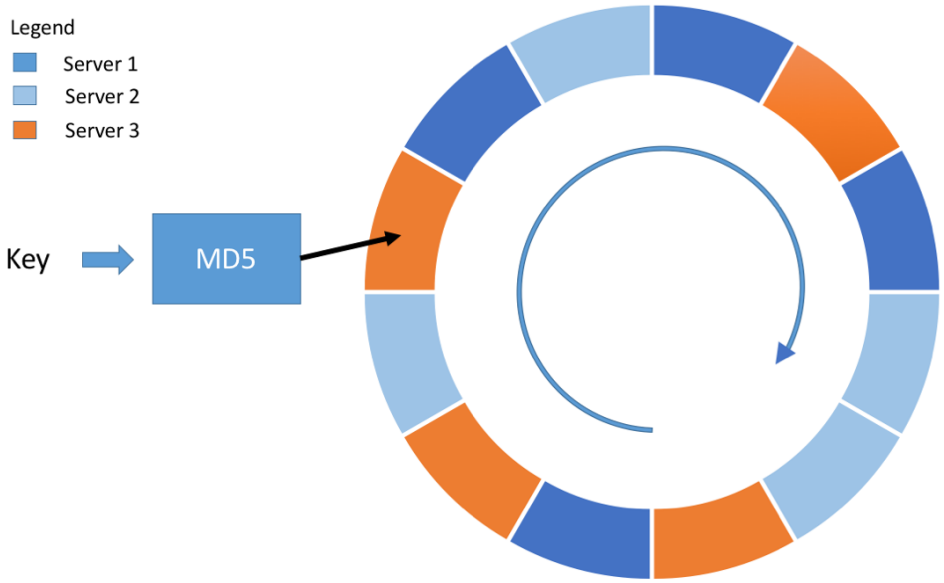

Figure 2: Consistent Hashing

For load balancing and server selection, I use consistent hashing technique, which is implemented in ConsistentHash[1] class. This method uses a hash function to create a key-space in the form of a circle, where each value generated by a hash function is mapped to the closest point it lies to on the edge of the circle in a clockwise direction. The issue with the current setup is that the server hash values may not be uniformly distributed and moreover the hashes of the request keys may not be uniformly distributed as well. In this scenario, load balancing fails. To mitigate this issue, virtual nodes are introduced. For each server, I create 200 additional hash values which map back to the same server and put them on the ring. This way, the interval is portioned into many smaller buckets, resulting in more uniform key space distribution. The random distribution of the virtual nodes across the ring ensure that uniformity is maintained even if request key hashes tend to fall more frequently in the same vicinity on the ring. In this case uniformity is maintained due to the fact that there are smaller server intervals across the circular ring space. This approach is visualized in the figure 2, where there are 3 servers and for each server there is additionally 3 virtual nodes created.

1.3 Write Operations and Replications

In the Middleware[2] class, the requests are parsed based on the operations. The write operations are identified and pushed to the relevant SET queue based on the primary server it will be processed by. For each memcached server, there exists a separate thread of AsyncClient[3] class instance that polls for the request from the queue associated with it. The write operations are handled in an asynchronous manner and the order of requests are maintained by using an Array Blocking Queue that enforces a FIFO order and which is also thread-safe. To track responses from memcached server (or servers during replication),

[1] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/ConsistentHash.java
[2] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/Middleware.java
[3] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/AsyncClient.java

I maintain a local queue in each AsyncClient[1] instance. This local queue also maintains a FIFO order of request, so that each memcached response can be mapped to the relevant request in the form of a RequestData[2] instance. In the case of no replication, once I receive a response from a memcached server, I parse the response, locate the associated RequestData[2] instance from a local queue and send back the response to memaslap.
However, in the case of the replication, I write the request to the primary memcached server associated with the instance of AsyncClient[1] and to the replication servers as well, which are provided in the RequestData[2] instance. In an instance of each AsyncClient[1] I maintain a separate socket channel for a primary memcached and replication memcached servers. I write to each of them and I parse their responses. For each individual response I associate it with a RequestData[2] instance and track its replication state. I send back the response to memaslap only when a replication counter for a RequestData[2] instance inside the local queue reaches the desired level. To achieve this functionality, I maintain a global hashmap mapping a corresponding socket channel associated with a memcached server connection to the current request (it will get the first response for) inside the local queue.

To get the replication memcached server identities from a request key, I use consistent hashing. I hash the key and look for closest points on the circle in clockwise direction that map to memcached server identities. However, in the presence of virtual nodes same server can be encountered when searching in clockwise direction and varying collection of replication servers may be generated for each primary server. To mitigate this issue, before I add virtual nodes to the consistent hash circle, I make a list of servers which are currently present on the circle, while maintaining their order. So once I get the memcached server identity using a request hash key, if the replication is on, I look at the local list in ConsistentHash[3], find the primary server and select as replication server elements right after it.

The latency that the writing operation will incur, will come from the time SET requests spend in the queue waiting to be processed by the write handler (AsyncClient[1] class) and the time the request spends in memcached server. In case with replication, the time for processing of requests will increase, since the response is sent back to memaslap only when the desired replication count is reached. Consequently, increasing the time, a request spends inside the middleware. The code instrumentation logs generated from the middleware experiment (Middleware_trace3) support the fact that during the replication, SET requests generate an additional overhead due to the extra time spent waiting for the replication responses to arrive.

1.4 Read Operations and Thread Pool

The read operations are identified and pushed to the relevant GET queue based on the primary server it will be processed by. For each memcached server, I create a numThreadsPTP number of threads of SyncClient[4] instance that communicate with that specific memcached server and pull the requests from the associated GET queue. To make sure that the queue between main receiving thread and read handlers is not accessed in unsafe concurrent manner, I use Array Blocking Queue, which implements thread-safe operations. All threads in a specific thread pool make a connection with a specific memcahed server associated with that thread pool.

[1] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/AsyncClient.java
[2] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/RequestData.java
[3] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/ConsistentHash.java
[4] https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/src/SyncClient.java

2 Memcached Baselines

I ran a memcahed baseline experiment with the specification present in the table below. I started two memaslap client machines simultaneously, with an increment of 8 virtual clients. For each increment of virtual clients, the experiment is repeated 5 times for 30 seconds.

Number of servers	1
Number of client machines	2
Virtual clients / machine	8 to 128; increments of 8
Workload	Key 16B, Value 128B, Writes 1%
Middleware	Not present
Runtime x repetitions	30s x 5
Log files	Baseline_run1

2.1 Throughput

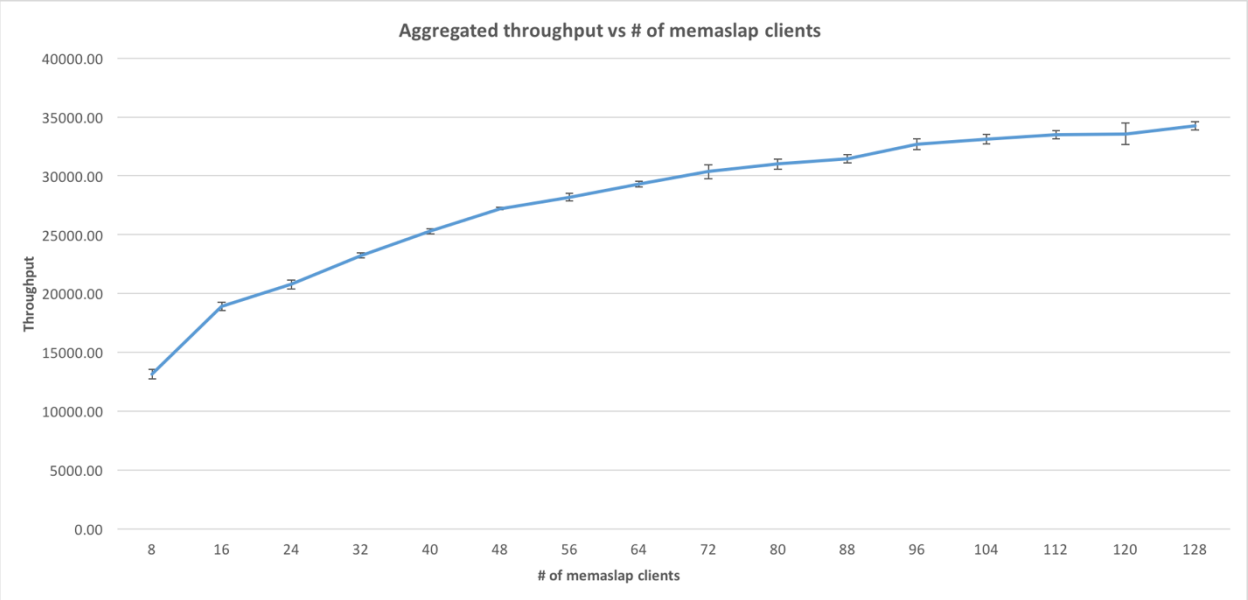

Figure 3 Baseline experiment aggregated throughput vs number of memaslap clients; based on Baseline_run1 data

Figure 3 represents graph of aggregated throughput versus number of virtual clients. At each step I aggregate throughput of two client machines and compute the mean and standard deviation across 5 iterations. As seen from the Figure 3, the aggregated throughput increases drastically from 8 to 16 number of virtual clients. From 16 to 48 virtual clients the throughput is continuously increasing, though the slope has noticeably decreased. From 48 to 88 virtual clients the same behavior is illustrated on the graph, the slope has decreased, but the throughput is still noticeably increasing. However, from 88 to 128 virtual clients the behavior changes, there are occasional increases in the throughput, but at the same time the slope is getting flat meaning the saturation area is being reached. Based on the graph I would assume the saturation point is after 112 virtual clients, since in the range from 112 to 120 the slope is mostly flat.

2.2 Response Time

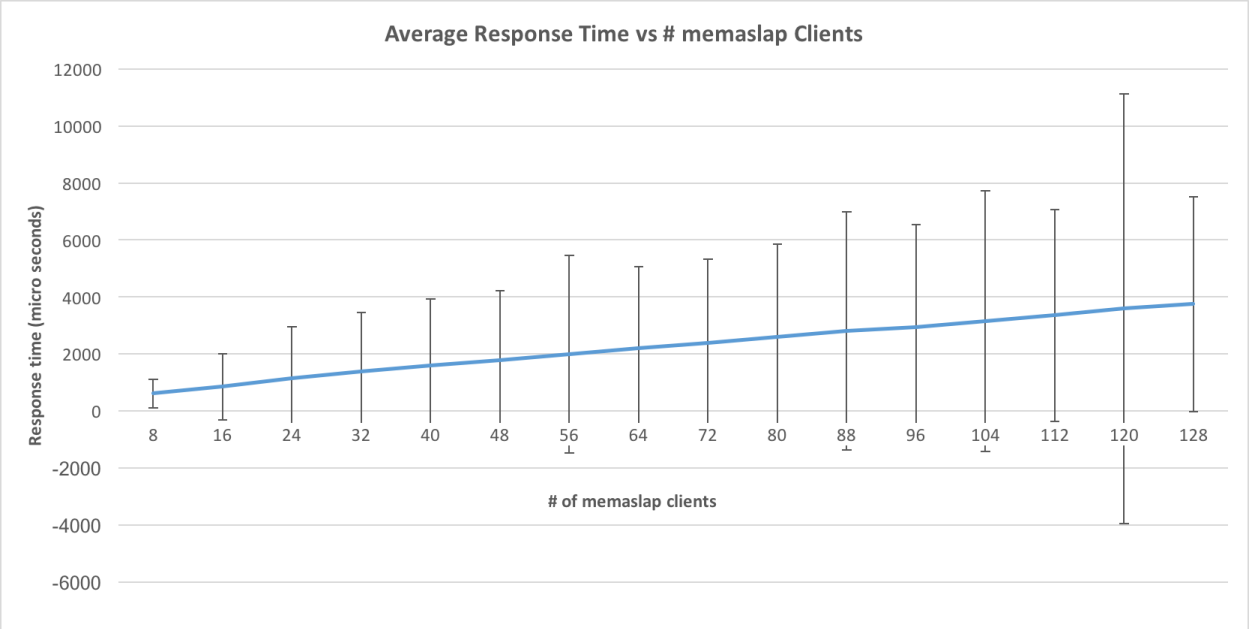

Figure 4: Baseline experiment Average Response time vs # of memaslap clients; based on Baseline_run1 data

Figure 4 represents graph of average response time vs number of virtual clients. At each step I average the mean response time and variation across 5 iterations of each client machine. To get the standard deviation I take square root of the average variation.
Figure 4, depicts the continuous almost linear increase of the response time as a function of number of virtual clients. This behavior is expected, as the number of requests increase, the waiting time of the request may increase as well. The standard deviation is also increasing with the response time. However, the saturation point can't be identified from this figure, since there is no drastic increase in response time and hence no significant slope variation. To conclude, this graph of the response time as a function of the number of virtual clients is not very informative in this case and doesn't tell much about the behavior of the system.

3 Stability Trace

I ran the middleware stability trace according to the specification present in the table below. I started all three clients simultaneously and ran each experiment for 90 minutes to be able to identify warm up and cool down phases. The graphs presented below have the first and last two minutes cut off due the warm up and cool down phase.

Number of servers	3
Number of client machines	3
Virtual clients / machine	64
Workload	Key 16B, Value 128B, Writes 1%
Middleware	Replicate to all (R=3)
Runtime x repetitions	90m x 3
Log files	Middleware_trace1, Middleware_trace2, Middleware_trace3

3.1 Throughput

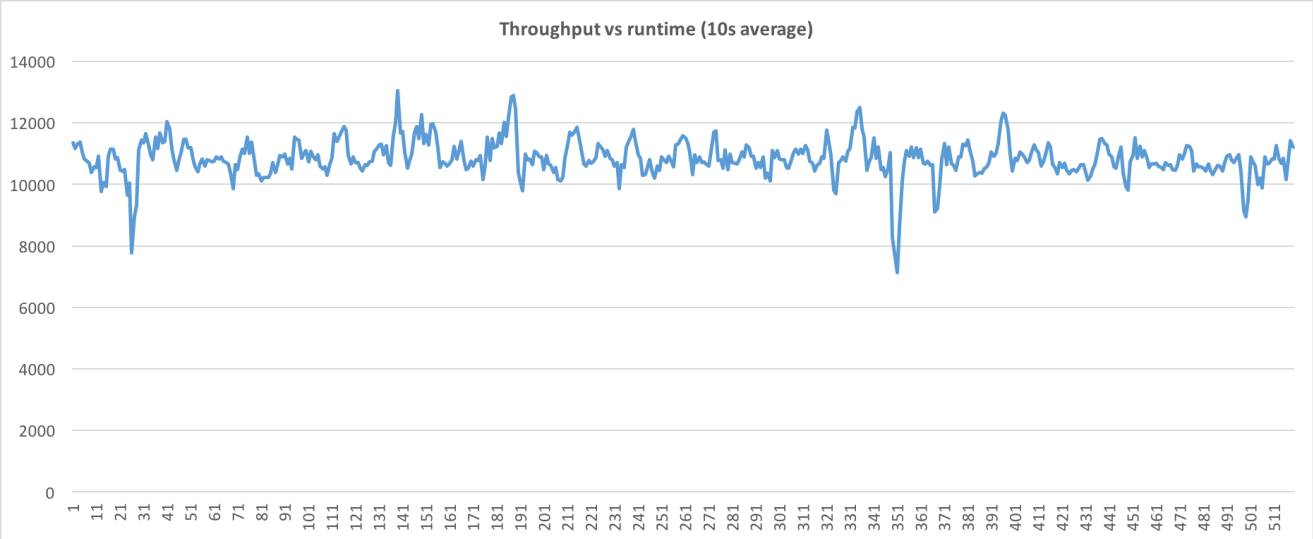

Figure 5: Middleware trace: Throughput vs run time (average over 10s interval) based on Middleware_trace2 data

The figure 5 presented above is based on the Middleware_trace2 data. The experiment was run with 8 number of threads for thread pool and no logging for code instrumentation. The x-axis is time in 10 second interval periods, and the y-axis is throughput average over 10 second period.
It can be seen from the graph that mean throughput stays constant at around 11,000. The throughput does vary across the run time, but it is not periodic and the variation is not significant, for the most part of the experiment throughput stays between 10,000 – 12,000 range. However, there are two visible drops in the range of 2000-3000 TPS observed on the figure. One drop is around point 21, and another drop is at point 351. These two drops seem random, they are not periodic and they occur within an interval of around 50 minutes. To further investigate these drops, I ran 1-hour trace without middleware to investigate the throughput pattern and response time over a long time period on Azure cloud (noMiddleware_1h_trace

data). The 1-hour trace without middleware, one memcached server and one memaslap client with 64 virtual clients generated a stable throughput throughout the runtime, with the exception of three drops. These drops are similar to the ones observed in Figure 5. The only difference is that in the graph presented on Figure 5 these drops last couple of second longer. This may be due to the overhead introduced by middleware in terms of load balancing, queueing and replication. So it is in accordance with logic that middleware takes couple of seconds longer to recover from these drops.

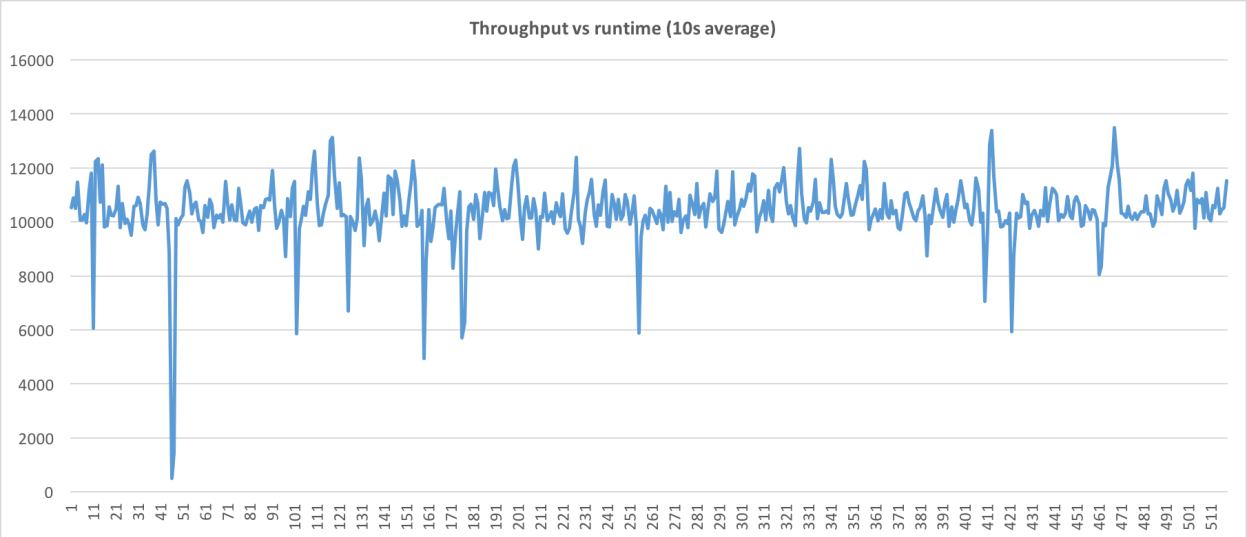

Figure 6: Middleware trace throughput vs run time (average 10s); based on Middleware_trace3 data

Figure 6 depicts a throughput vs run time graph based on Middleware_trace3 data with code instrumentation logging on. I conducted this experiment to gather the code instrumentation data and to further investigate the occurrences of throughput drops. As depicted on the Figure 6 the drops are more frequent in this case and one of the drops is exceptionally large. However, these facts do not contradict the previous conclusion on the randomness of the drops. The drops in the Figure 6 are more evident because these drops last several seconds longer then in Figure 5. The prolonged duration of drops can be attributed to the overhead introduced by logging every 100^{th} GET and SET request.
Furthermore, I examined yet another middleware trace experiment with the exact same configuration as the one presented in Figure 5, but conducted at different time period to account for traffic on Azure. The graph created based on the generated data from this experiment (Middleware_trace1), has no evident throughput drops, suggesting that the drops seen in the Figures 5, 6 are not due to the middleware.

3.2 Response Time

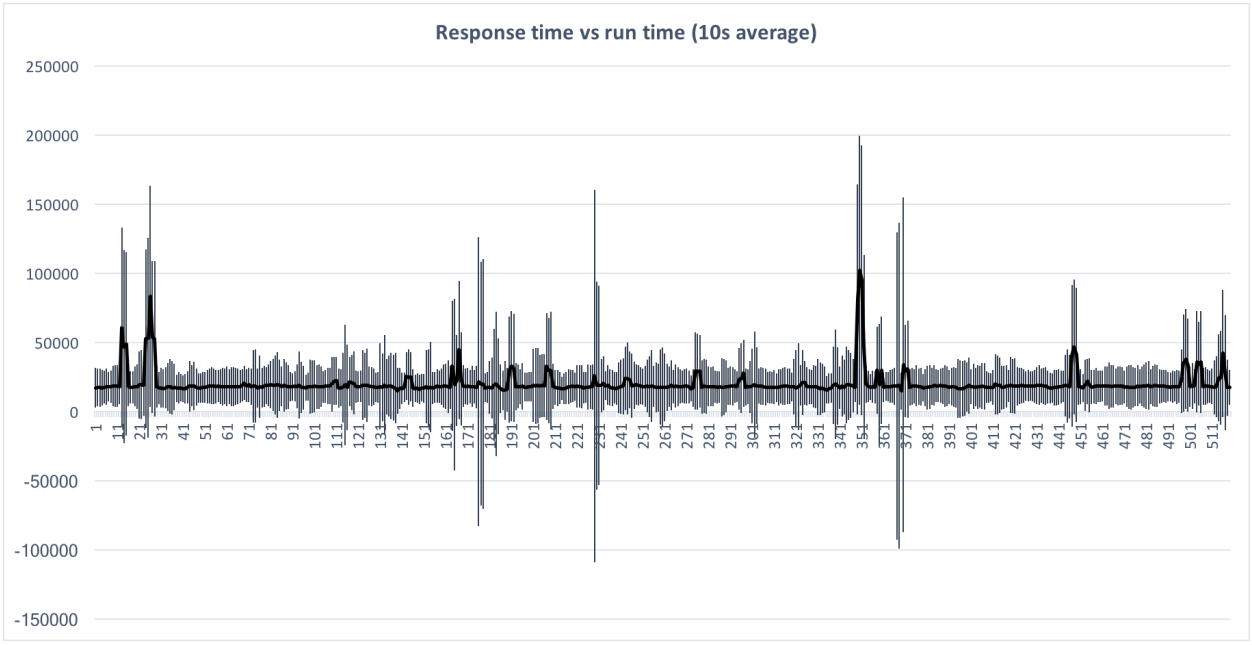

Figure 7: Middleware trace Response time vs run time (average 10s); based on Middleware_trace2 data

Figure 7 depicts a response time vs run time averaged every 10 second period based on Middleware_trace2 data. The response time in the Figure 7 is stable throughout the runtime, with the exception of 3 evident peeks at points around 11, 21 and 351. The two drops depicted in Figure 5 and two peeks in Figure 7 coincide based on the run time, they appear at point around 21 and 351. Therefore, the same logic applies to the cause of sudden increase in response time depicted in Figure 7. Standard deviation is also observed to be stable throughout the run time, suggesting the stability of the middleware.

3.3 Overhead of Middleware

The middleware introduces some overhead and reaching the performance of baseline is not expected. The functionalities that middleware implements like load balancing and replication introduce an apparent overhead. As a result, increasing the response time, and consequently decreasing the throughput. This becomes evident when comparing throughout and response time of baseline experiment with the results of middleware trace.

The Middleware trace (Middleware_trace2) uses 3 memaslap machines each with 64 virtual clients, making a total of 192 clients. Since the baseline experiment (Baseline_run1) reaches a saturation point right after 112 virtual clients, I assume that with 192 clients baseline experiment would reach a throughput of around 35,000 and response time of 4000 micro seconds.

The average throughput of middleware experiment based on Middleware_trace2 data is 10868.81 and the average response time is 20062.94 micro seconds. I assume, that due to implementation of load balancing using consistent hashing, the distribution of request along three memcahed servers is uniform. Based on

this assumption the throughput per memcached server becomes the third of the average throughput and response time stays the same. The table below depicts the comparison between baseline experiment and middleware trace assuming 196 virtual clients and one memcahed server.

	throughput	Response time
Baseline experiment	35,000	4,000
Middleware trace	10868.81/3 = 3622.93	20062.94
Middleware overhead	0.10 * Baseline throughput	5 * Baseline response time

Logfile listing

Short name	Location
Baseline_run1	https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/Logs/No_Middleware/Baseline_exp.zip
Middleware_trace1	https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/Logs/Middleware/clients3_servers3_replication3/run_1.zip
Middleware_trace2	https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/Logs/Middleware/clients3_servers3_replication3/run_2.zip
Middleware_trace3	https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/Logs/Middleware/clients3_servers3_replication3/run_3.zip
noMiddleware_1h_trace	https://gitlab.inf.ethz.ch/iverip/asl-fall16-project/blob/master/Logs/No_Middleware/client1_server1_trace.zip