Advanced Systems Lab (Fall'16) – First Milestone

Name: ***Teymur Babayev***
Legi number: ***15-926-231***

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

Middleware starts by running RunMW[1] class, which runs MyMiddleware's thread (once). Mentioned class MyMiddleware[2] is just a helper class which only initiates other objects.

First initiated components are MiddlewareServer[3] and MessageProcessor[4], they share 1 thread. MiddlewareServer opens sockets for memaslap clients, reads the data and passes it to MessageProcessor. MiddlewareServer is also used to send responses back to memaslap clients. MessageProcessor initiates main queues for get/set operations and asynchronous/synchronous workers. Its .processData() method takes request from the server, hashes the key to obtain assignment to one of memcached servers, and, based on the type of request (get, set, delete), puts it into the corresponding queue for that memcached server. *delete* requests are put into set queue since they follow the same fashion as set requests. Hashing is done by MD5Circle[5] class instance.

Asynchronous and synchronous workers are implemented in classes AsynClient[6] and SynClient[7]. Since we are to employ a thread pool, there also exists a helper class SynThreadPool[8] (section 1.4). MessageProcessor initializes 1 instance of AsynClient and 1 instance of SynThreadPool per memcached server. Each AsynClient gains access to it's own *set/delete* queue, while each SynThreadPool gains access to it's own *get* queue.

Both of the clients connect to their primary memcached server (and to replication servers for AsynClient). They send requests to memcached servers, get the response and call MiddlewareServer's .send() method to send the response back to memaslap.

Class Message[9] is used to hold a request together with socketchannel of memaslap, our MiddlewareServer instance and the request itself. It also has fields for storing instrumentation information. Class RequestState[10] is used to track state of a request for our replications. It is used inside AsynClient. Class ChangeKey[11] is used in MiddlewareServer to change the keys for socket channels inside the run() method (since the keys are not thread safe).

[1] https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/asl-fall16-babayevt/src/ch/ethz/asltest/RunMW.java

[2] https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/asl-fall16-babayevt/src/ch/ethz/asltest/MyMiddleware.java

[3] https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/asl-fall16-babayevt/src/ch/ethz/asltest/MiddlewareServer.java

[4] https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/asl-fall16-babayevt/src/ch/ethz/asltest/MessageProcessor.java

[5] https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/asl-fall16-babayevt/src/ch/ethz/asltest/MD5Circle.java

[6] https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/asl-fall16-babayevt/src/ch/ethz/asltest/AsynClient.java

[7] https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/asl-fall16-babayevt/src/ch/ethz/asltest/SynClient.java

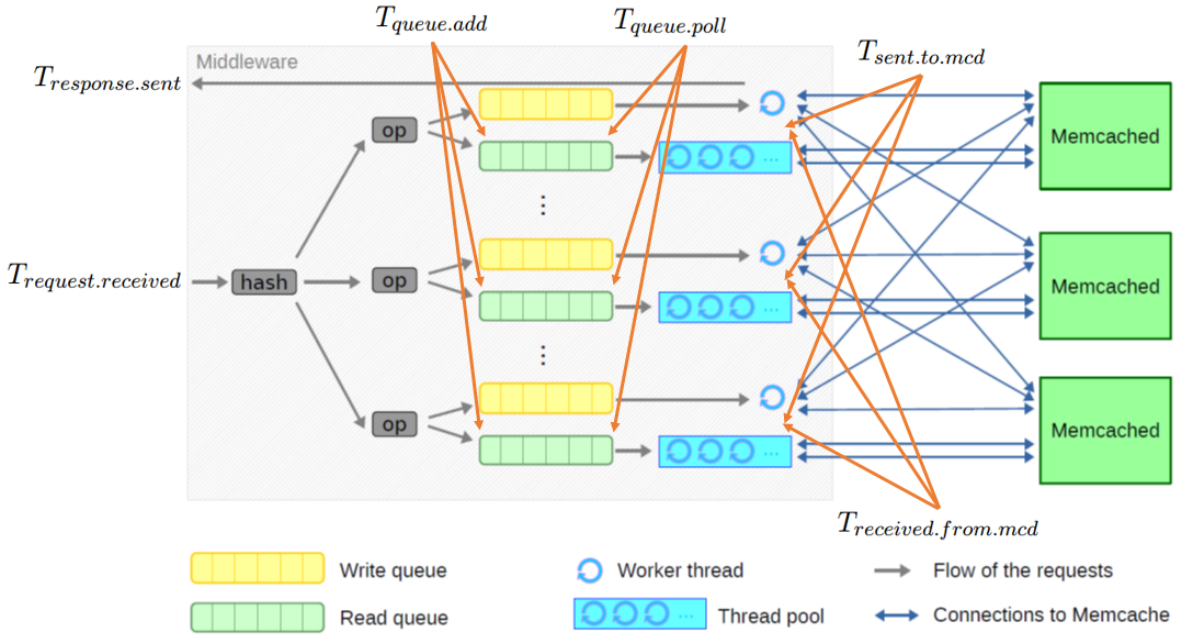
[8] https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/asl-fall16-babayevt/src/ch/ethz/asltest/SynThreadPool.java

[9] https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/asl-fall16-babayevt/src/ch/ethz/asltest/Message.java

[10] https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/asl-fall16-babayevt/src/ch/ethz/asltest/RequestState.java

[11] https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/asl-fall16-babayevt/src/ch/ethz/asltest/ChangeKey.java

Instrumentation schematic:



Instrumentation note: success flag by default is true, gets marked false if we get unsuccessful response for set/delete commands or if we get only and *END* for get commands.

1.2 Load Balancing and Hashing

When the message is passed from MiddlewareServer to MessageProcessor, it's key is extracted and hashed and based on that hash value it is assigned to one of memcached servers.

Using Java's Object.hashCode() method didn't yield good results, because each time the key was hashed this hash function returned a different hash value, and getmisses were obtained. That's why preference was given to more robust MD5 hashing.

However, consistent hashing with splitting the circle of hash space into #servers arcs and assigning to server based on which arc the hash value of that key falls into did not yield uniform balancing. Thus, more advanced technique was employed by using so called *virtual nodes*, where the circle is split into many more arcs. This circle with virtual nodes is initiated in the constructor of my MD5Circle class.

To initiate these nodes in uniform fashion I have 7 random strings initiated in the constructor of MD5Circle, and only #servers of them will be used to position nodes on the circle. Replication nodes are added by hashing concatenation of these strings with numbers 0..#replication-nodes.

Method .get() takes a byte array and looks up at the TreeMap (which represents our circle) to define which of those strings it belongs to. After obtaining the string, I use a HashMap to extract memcached server ID corresponding to that string and return it to MessageProcessor.

In my experiment I was replicating each "string" on the circle 20 times. Eventually it gave me load balancing which is very close to uniform. Also, amount of requests sent on each server can be checked in my code by uncommenting lines 128 - 138 in MessageProcessor.java.

1.3 Write Operations and Replication

Part of the system that is responsible for sending and receiving responses for *set* and *delete* request is implemented in my class AsynClient. Each instance of this class uses 1 thread. It connects to primary memcached server and servers to which it will replicate, so it will use R socket channels.

AsynClient is initiated in MessageProcessor's constructor. Exactly there it gets a reference to its corresponding write queue, which is also initiated in MessageProcessor.

After polling request from the queue, the request state is initialized (to track replication status) and request gets written on all socket channels. After writing, AsynClient attempts to read the channels. Since we use non-blocking mode here, we may face a situation where nothing is received over the channel. In this case, we continue writing. It is also worth to mention that we continue to read even if the queue is empty, since we may still be waiting for some response.

As for the fact that nothing may be received, there also may be an issue that we may receive several responses in 1 read buffer (buffer size is sufficient for not receiving partial messages). Because of that, we maintain a local queue which will keep track of whose response we are waiting on which channel. After parsing is done, we do changes to the state of that request. As mentioned above, this is done due to replication requirements. When number of responses for a concrete recipient reaches the number of replications, we send the response to recipient using MiddlewareServer's .send() method. Based on what the responses are, we may send 3 different responses: STORED, DELETED, ERROR. First two responses are sent in case when all of the responses received by AsynClient are STORED/DELETED. In case at least one of them contains anything else (which is an error), we send ERROR to recipient.

When we have R=1 (no replications), we update state of the request when we receive response as mentioned above. Number of successful replications instantly becomes 1, and then it is equal to R. This means that in this context, response will be sent to recipient right after receiving it.

As for writing latencies, it is hard to make some structural assumptions based on memaslap output. Since in context of this milestone we didn't do detailed analysis of instrumentation log there are no statistically significant results, but skimming through this log may give some idea about latency. It can be seen that pretty often $T_{received.from.mcd} - T_{sent.to.mcd}$ takes more time than $T_{queue.poll} - T_{queue.add}$. Of course this is due to the fact that we are doing replications and $T_{received.from.mcd} - T_{sent.to.mcd}$ is finalized only when all responses of replications are received. Also, we receive only 1 percent of set requests, so $T_{queue.poll} - T_{queue.add}$ should not be large. In my stability trace I obtained mean of response time to be 15.7 ms. If we analyze baseline logs then with an assumption of linear progression of response time with increasing number of clients, we may assume that average response time for 192 clients without middleware may be 4 ms, so this is the Azure network latency. Again, based on quick skimming the instrumentation log, we can make an assumption that out of 11.7ms left 4ms are spent on processing replications and parsing and 2ms is spent in queue. The left 5.7ms would then be split between hashing request and putting it into queue and sending it back to the recipient. In no replications case, time spent on processing replications would probably decrease, so we may except some gain in performance then. However, assuming estimates mentioned above, it would probably be hard to get response time less than 11-12ms with my middleware configuration.

1.4 Read Operations and Thread Pool

A single-threaded *get* worker is implemented in my SynClient class. Each instance of that class employs one socket channel to memcached server it is assigned to. In the run() method a request is taken from the *get* queue, then it is written to memcached, then SynClient is reading. Since our socket channel to memcached is configured to be blocking, this approach is considered to be synchronous, because thread gets blocked until response is received. Afterwards, new iteration starts.

Since we want to implement a thread pool, I have an additional class SynThreadPool, which initiates t (where t = number of threads per thread pool) instances of SynClient. Each instance receives reference to the same *get* ArrayBlockingQueue from MessageProcessor.

SynThreadPool is initiated in the constructor of MessageProcessor. To ensure that *get* queue is accessed in safe manner, SynClient uses .take() method of BlockingQueue interface, which

blocks the thread if queue is empty. Choice has been made to use ArrayBlockingQueue, and not LinkedBlocking queue, to test the boundaries of the queue. In my stability experiment I used queue size of 5000 and it did not overfill the queue. It's worth to mention that I used .add() method to add requests to the queue. While this method is not thread safe, this choice has been done to test if we will get an overfill and receive an error. However, ArrayBlockingQueue can always be substituted by LinkedBlockingQueue to be unbounded. As mentioned above, no queue overfill error has been reported.

So, one instance of SynThreadPool creates t instances of SynClient, all accessing the same queue, and each of these SynClients uses one socket channel to connect to memcached.

2 Memcached Baselines

Baseline experiment parameters were up to the following table:

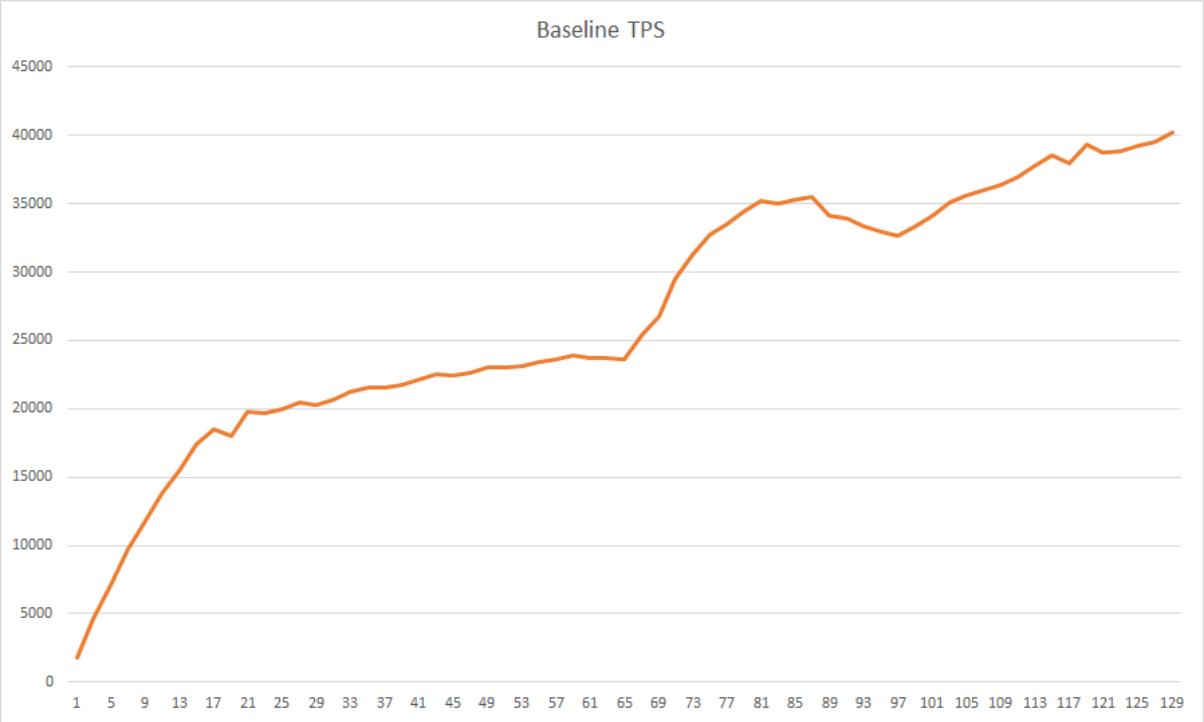
Number of servers	1
Number of client machines	1 to 2
Virtual clients / machine	1 to 64
Workload	Key 16B, Value 128B, Writes 1%
Runtime x repetitions	30s x 5
Log files	baseline1, baseline2, baseline_all

Since we had some freedom in choosing how to run the experiment, I did it the following way: first only one machine was running changing clients number from 1 to 64 with step size of 2, then, same machine was always running 64 clients and second machine was increasing number of clients from 1 to 64 with step size of 2.

It is worth mentioning that memcached server was restarted after first machine reached 64 clients. The result of that will be seen in the following plots.

A note on log files: first two log id's link to the folder, since there are too many log files. First folder contains log files for the first part of baseline run with a single client machine, whereas the second one contains two separate folders for the first and second machine respectively. Log file names follow next scheme: *log-iteration_id-_repetition id.log*

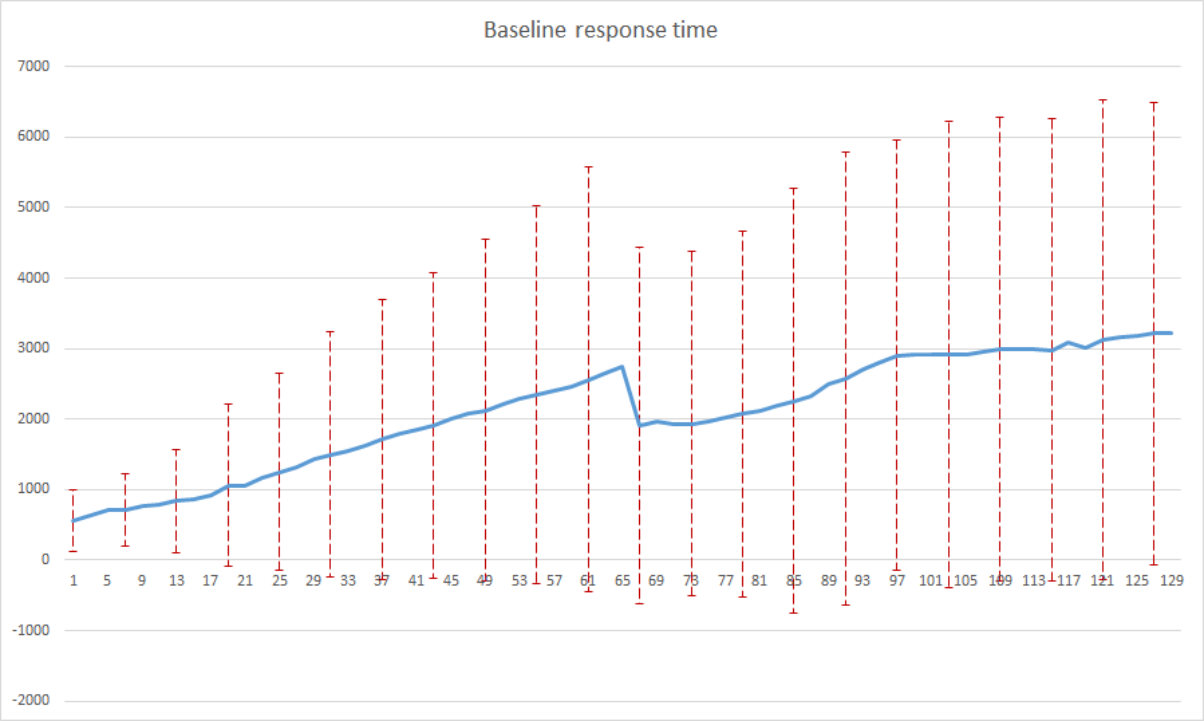
2.1 Throughput



Since memcached server was not restarted after each change of number of clients, slight saturation at point close to 64 clients may be explainable: server received a lot of data since the keys and values are random and started to respond slower by the end of the first half. It explains why when memcached was restarted, it again began to behave again in progressing, but not saturating fashion. Despite when reaching client number of 128 we can see some sort of a saturation, this can be explained by the fact mentioned above - server got too much data. Thus, there can be no clear assumption made about where the behaviour is saturating if we restart the server periodically and use memaslap parameter -o 0.9.

There is also a strange dip at #clients equal to 97. It actually starts at #clients = 89 at lasts until #clients = 105. However considering we had no middleware and looking at the first part of the plot there is only one explanation to that phenomena - with highest probability it is an inconsistency of virtual machines performance.

2.2 Response time



Response time plot follows exactly same fashion as the TPS plot. We have a slight decrease in the middle of the plot, due to the memcached restart. As well, we have a slight peak at #clients = 97. Regarding high variation of the response time, depicted by wide error bars, one can make a conclusion that this behaviour is somewhat normal and seeing something similar in stability trace should probably be expected.

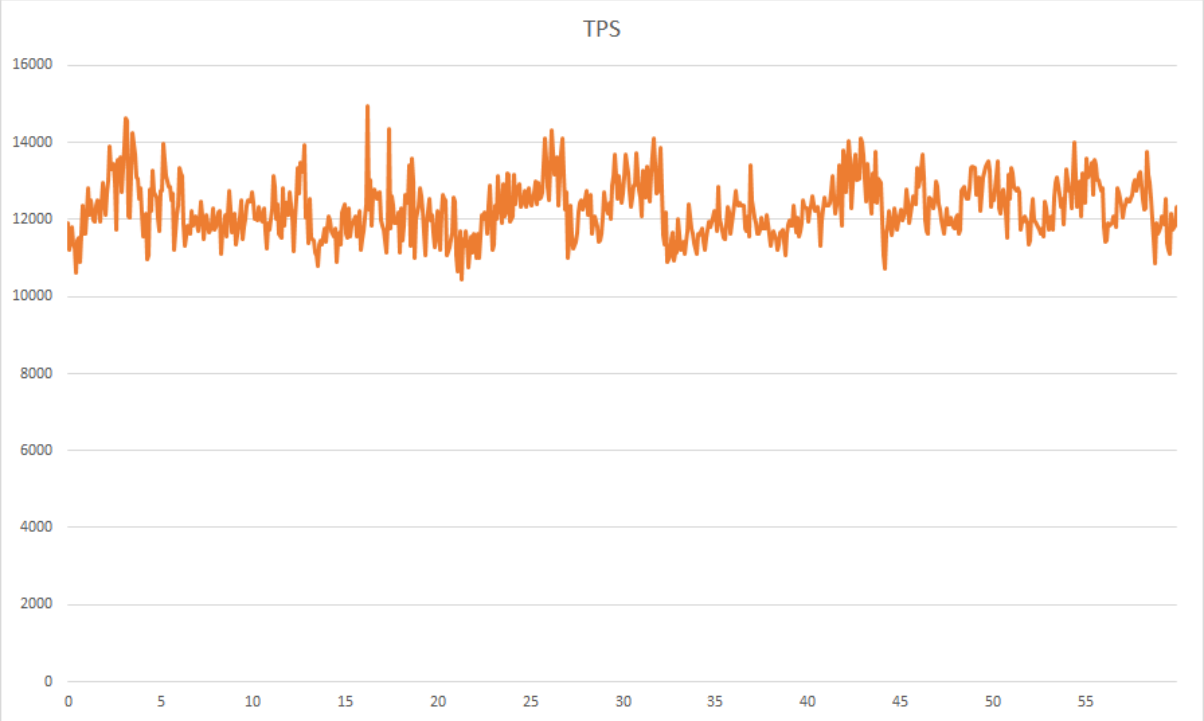
3 Stability Trace

Stability trace experiment parameters were up to the following table:

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)
Number of threads PTP	16
Runtime x repetitions	70min x 1
Log files	trace1, trace2, trace3, TPS_trace, RT_trace, instrum

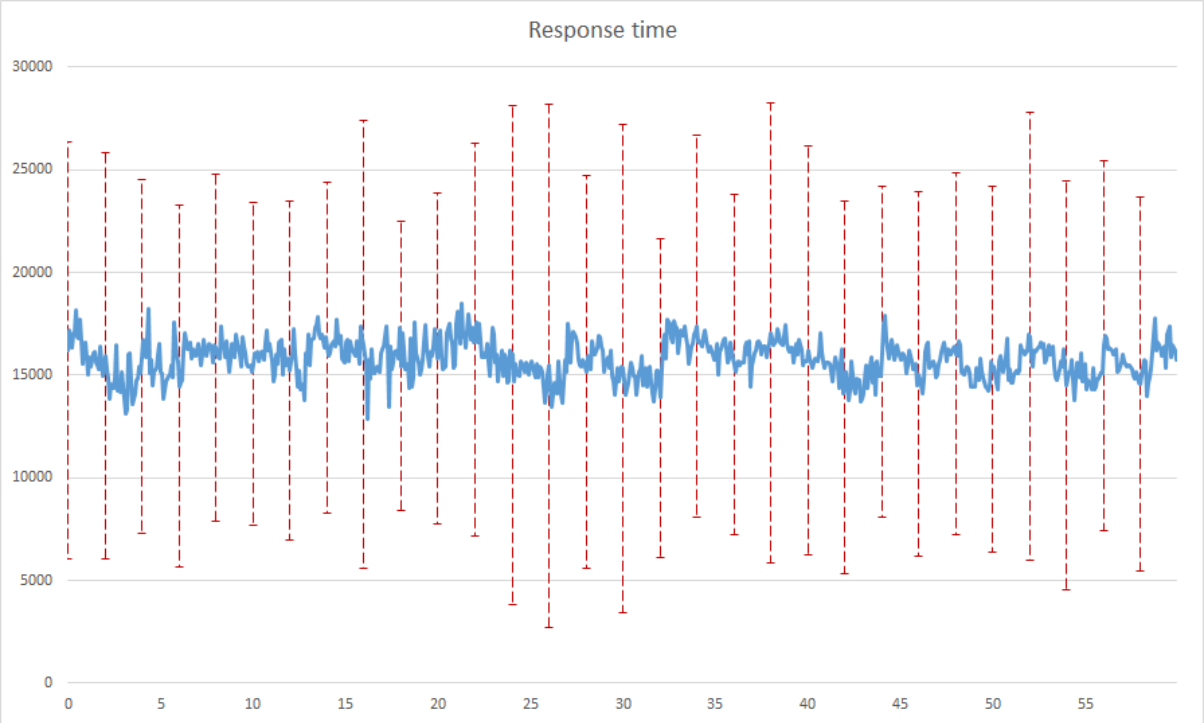
It's worth mentioning that stability test for performed for 70 minutes to prevent warmup/cooldown phases in the plots. The plots however show only 60 minutes of a run, but logs contain full 70 minute run information for the sake of completeness. Number of threads PTP was selected to be multiple of 8, since we use 8 core machine. Also, memaslap clients were tuned to output statistics once in 5 seconds.

3.1 Throughput



From the TPS plot above one can see that middleware is functional and it can handles a long-running workload without crashing or degrading in performance, as per the requirements of the milestone. There are no major peaks and dips detected. Calculated mean TPS from memaslap outputs (once in 5 seconds) reports 12262 operations per second. Standard deviation of TPS reported from memaslap outputs (5s) is 782 operations per second (slightly above 5% of the mean).

3.2 Response time



Response time plot follows the same trend as the TPS plot above. No major breakdowns or unexplained behaviour. Error bars seem a bit wide, but as was stated in the baseline RT plot explanation, this is somewhat expected behaviour. Calculated mean average RT from memaslap outputs (once in 5 seconds) reports 15.78 milliseconds. Standard deviation of average RT reported from memaslap outputs (5s) is 0.9 milliseconds (as well, slightly above 5% of the mean).

3.3 Overhead of middleware

Comparing performance of my middleware with baseline results is senseless until we do some assumptions and estimates, since the setup of these experiments is different. Since behaviour of "baseline" is more "predictable" because there exists no middleware and everything is straightforward, I will try to extrapolate baseline results to the stability experiment setup.

In the following table you can see stability trace results and estimated "baseline" results with 192 clients:

-	No Middleware	Middleware
TPS	60k	12k
RT	4ms	15.7ms

As mentioned in latency estimation1.3, we have no statistical evidence of what are the exact times spent in the middleware, but skimming through the instrumentation log may help to do some high level assumptions.

Since we have 4ms as mean response time w/o MW and 15.7ms with, we can imply that 11.7ms are added by the middleware. These 11.7ms consist of following components:

Time between receiving request and putting it into queue (includes hashing, e.g.);

Time spent in queue;

Time spent for parsing response, managing replication status;

Time between deciding to send something back and sending it back to memaslap;

As mentioned, it is yet not possible to define bottleneck clearly, but numbers in the table make sense. Despite calculating TPS from response time and vice versa is not always feasible in real life, with middleware we have 4 times larger response time, and 5 times less TPS. This is not exact inverse proportionality, but pretty close to that.

Logfile listing

Short name	Location
baseline1	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/tree/master/baseline_logs/1_64
baseline2	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/tree/master/baseline_logs/65_128
baseline_all	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/baseline_logs/all_in_one_sheet.csv
trace1	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/trace_logs/trace1.log
trace2	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/trace_logs/trace2.log
trace3	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/trace_logs/trace3.log
TPS_trace	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/trace_logs/TPS_trace.csv
RT_trace	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/trace_logs/RT_trace.csv
instrum	https://gitlab.inf.ethz.ch/babayevt/asl-fall16-project/blob/master/trace_logs/instrum.7z