Advanced Systems Lab (Fall'16) – First Milestone

Name: ***Rishu Agrawal***
Legi number: ***15-945-355***

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

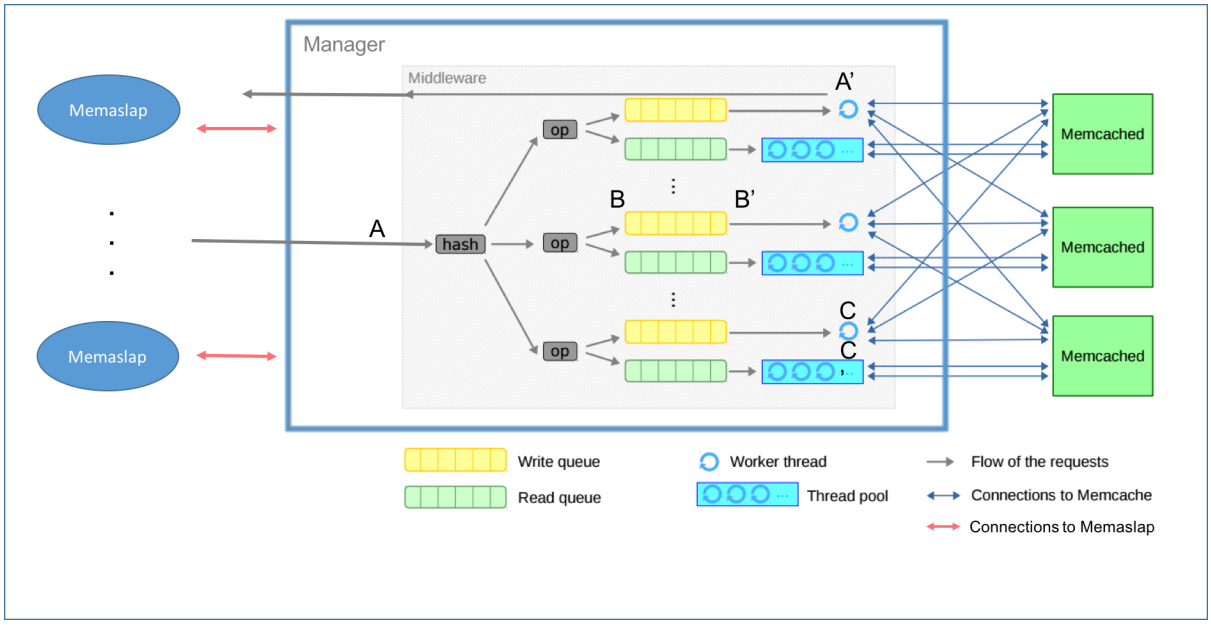

Figure 1: Architecture with instrumentation markings

The architecture has been implemented through the following classes:

- **Manager:**[1] This class is responsible for communicating with the memaslap clients. It initiates the connection, gets the requests from the clients and forwards it to the **Middleware**[2] object and finally sends back the response from the worker threads to the memaslap clients. It also maintains a counter of the number of set and get requests that is later used to sample the requests for logging.
- **Middleware:**[2] This class creates the **QueueManager**[3] instance and worker threads, both **Synchronous**[4] and **Asynchronous**[5], for each server. It hashes each request from the **Manager**[1], based on the key, and forwards it to their respective queues.
- **DataPacket:**[6] This class represents the request in the architecture diagram. It has an instance of the **Manager**[1] class and the socket channel connecting to the memaslap client. It also contains the actual read/write request, the servers it has to write to, in case of replication, and also stores the instrumentation measurements (e.g. T_{mw}, T_{queue}, T_{server} and $F_{success}$)
- **QueueManager:**[3] This class has the read and write queues as its attributes. It also has an **AsynchronousClient**[5] reference to wake up its selector.

[1] https://gitlab.inf.ethz.ch/ragrawal/asl-fall16-project/blob/master/src/Manager.java
[2] https://gitlab.inf.ethz.ch/ragrawal/asl-fall16-project/blob/master/src/Middleware.java
[3] https://gitlab.inf.ethz.ch/ragrawal/asl-fall16-project/blob/master/src/QueueManager.java
[4] https://gitlab.inf.ethz.ch/ragrawal/asl-fall16-project/blob/master/src/SynchronousClient.java
[5] https://gitlab.inf.ethz.ch/ragrawal/asl-fall16-project/blob/master/src/AsynchronousClient.java
[6] https://gitlab.inf.ethz.ch/ragrawal/asl-fall16-project/blob/master/src/DataPacket.java

- **SynchronousClient:**[4] This class gets the requests from the read queue and processes them synchronously, and fetches the response from the corresponding server it is connected to. The response is then sent back through the send method from the **Manager**[1] instance in the request packet.
- **AsynchronousClient:**[5] This class gets the requests from the write queue and processes them asynchronously. Since, the write requests may have to be replicated to secondary servers, it fetches the response from all the corresponding servers and checks for errors (if any). The response is then sent back through the send method from the **Manager**[1] instance to the primary server.
- **ConsistentHash**[7] This class implements consistent hashing using the MD5 hash function. It adds the servers to the circular ring and fetches the primary and replica servers from the key.
- **ChangeRequest:**[8] This class manages the interest operations in selection keys in the **Manager**[1] class.

The following tables describe what the markings in Figure 1 refer to:

A	A'	B	B'	C	C'
$T_{RequestArrival}$	$T_{ResponseReady}$	$T_{Enqueued}$	$T_{Dequeued}$	$T_{SenttoServer}$	$T_{ReceivedFromServer}$

T_{mw}	$T_{ResponseReady}$ - $T_{RequestArrival}$
T_{queue}	$T_{Dequeued}$ - $T_{Enqueued}$
T_{server}	$T_{ReceivedFromServer}$ - $T_{SenttoServer}$

All the time references above are measured using *System.nanoTime()* method which returns the time of execution in naoseconds. Hence, the elapsed time is computed as a difference of two time instances.

1.2 Load Balancing and Hashing

For load balancing and server selection, I use the technique of consistent hashing. The notion behind this technique is that it uses a hash function and delineates the keyspace in the form of a circular ring. All the servers lie on this ring at different positions according to the value they hash to. Each key from the request is then hashed and the output is mapped to the closest server it lies to in the circular ring in the clockwise direction.
The issue that the current vanilla scenario presents is that the server hash values may not be uniformly distributed in the keyspace. Furthermore, the hashes of the request keys may not be uniformly distributed as well which defeats the purpose of load balancing. Therefore, to alleviate this issue and make the load distribution uniform, I added virtual nodes. For each server, I allocate 200 additional hash values and put them on the ring. This number is heuristically chosen. My reasoning for this is that it allows me to partition the ring into several smaller buckets instead, which results in the keyspace distribution being more uniform. The random distribution of the virtual nodes across the ring ensure that even if the keys hash to a small segment on the circle more frequently, the uniformity of load distribution is maintained since there are smaller server intervals across the circular ring space and the request keys are allocated to different servers in an equiprobable manner. I have tried to visualise this approach in the Figure 2, where there are 3 servers and for each server there are 4 virtual nodes.

1.3 Write Operations and Replication

The write operations are identified and put in the write (called **setQueue** in the code) queue of the primary server in the Middleware[2] class. For each operation, I have a distinct worker thread per server that polls for the requests from the queue. Since the writes are asynchronous, I maintain the order of requests by handling them through an Array Blocking Queue. These queues are also thread-safe[9]

[7] `https://gitlab.inf.ethz.ch/ragrawal/asl-fall16-project/blob/master/src/ConsistentHash.java`
[8] `https://gitlab.inf.ethz.ch/ragrawal/asl-fall16-project/blob/master/src/ChangeRequest.java`
[9] `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html`

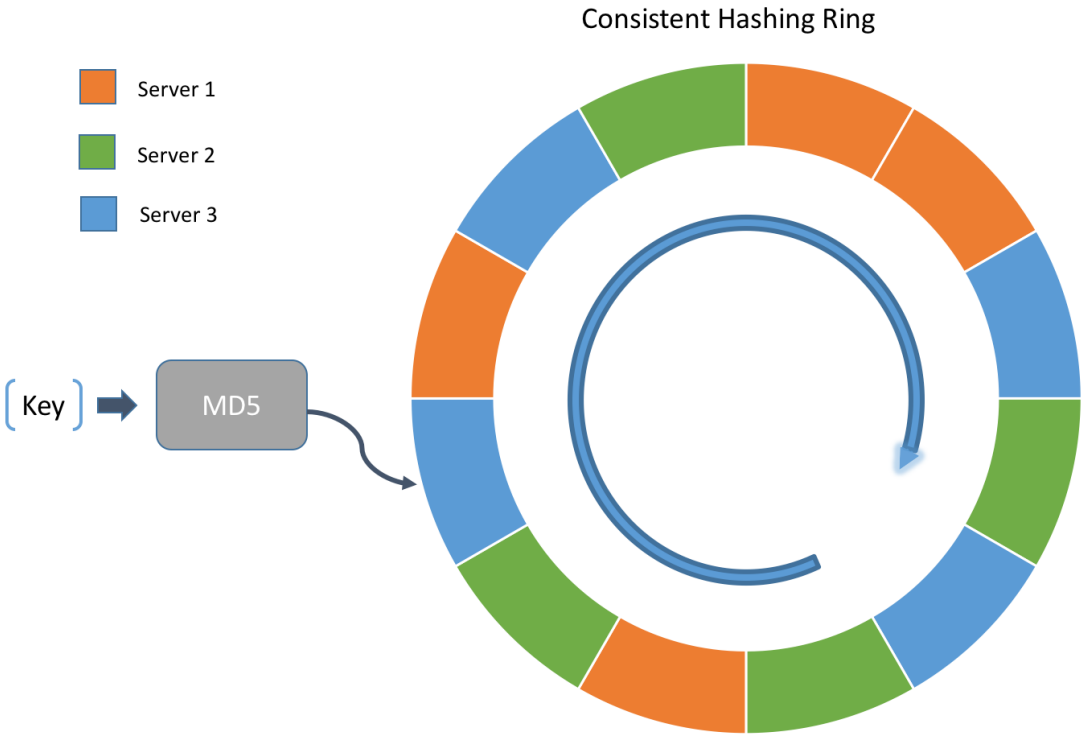

Figure 2: Consistent Hashing

and order elements by the **FIFO** policy[10]. Now, to track each individual response, I maintain a local queue in each asynchronous worker thread. This local queue also maintains the **FIFO** order of requests. I use this ordering to identify and map the memcached response to the corresponding request it belongs to.

In the case of replication, I first write the request to the primary memcached server associated with the instance of the **AsynchronousClient**[5] and then the replication servers, which are provided in the **DataPacket**[6] instance obtained from the *getWithReplication* method in **ConsistentHash**[7]. I initialise socket channels to all the servers in the constructor of the **AsynchronousClient**[5] class. However, since the replication may not be done in all the servers, I only use connections to the primary and secondary servers. Once requests are sent to the memcached servers, I change the selection key operation to read, and iterate through ready selection keys. Since each read event may contain several responses, I parse the response and for each individual response inside the ByteBuffer, I map them to the request they belong to and increment the replication count of this DataPacket. To achieve this functionality, I maintain a global hashmap mapping a socket channel associated with a memcached server to the number of requests it is waiting for inside the local queue. I also use a hashmap to map each **DataPacket**[6] with the number of responses it has received (replication count). Since the responses are ordered, the replication count and response from the memcached server can be traced back to the corresponding requests. Once all the responses are received from the primary and secondary servers, I remove this request from the local queue and send the appropriate response back to memaslap.

In a simple "write one" scenario, the mapping with the replication counter isn't required and once the response from a memcached server is received, it can be parsed and mapped to its associated **DataPacket**[6] instance from the local queue and then sent back to memaslap.

I sample the write operations and log it once after every 100 SET requests in the **instrumentlog**. Since the write operations are asynchronous, they don't spend a lot or time in the queue. I expect the

[10] `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html`

number of replication servers to which the worker thread has to write to limit the rate at which writes are carried because then the worker has to write to more servers and more responses have to be parsed until it gets all the responses from the primary and secondary servers. Also, the location of the servers in the network will also affect the rate of writes as longer network routes increase the response time. Since the averaging is not very informative, I tabulated the various response times from my **instrumentlog**.

(in microsecs)	Min	1st Quartile	Median	3rd Quartile	Max
T_{mw}	289	611	851	3668	173880
T_{queue}	1	17	20	42	31959
T_{server}	235	563.5	787	1767.5	173826

We can see that more than 75% of the requests spend less that 42 microseconds in the queue and the majority of the time is spend in getting the response from the server. The more the replication, the more T_{server} increases, thus increasing the T_{mw}.

1.4 Read Operations and Thread Pool

The read operations are identified and put in the read (called **getQueue** in the code) queue of the primary server in the Middleware[2] class. For each operation, we have several worker threads, denoted by *numThreadsPTP*, per server that pulls the requests from the queue. These workers are instances of the **SynchronousClient**[4] class.

I make sure that the queue between the main receiving thread and the read handlers is not accessed in unsafe concurrent manner by using Array Blocking Queue which implements thread-safe operations.

The relation between threads in the thread pool and connections to servers is that all threads from a particular thread pool connect to a unique memcached server. Each read thread inside a thread pool pulls from the same read queue in a thread-safe manner and then proceeds to blocking write and read operations.

2 Memcached Baselines

I wrote a little script to run this experiment with the specifications mentioned in the following table. I ran these experiments on 3 basic A2 machines on Azure. I started both the memaslap clients at the same time.

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% [11]
Middleware	Not present
Runtime x repetitions	30s x 5
Log files	baselinebench

2.1 Throughput

Here, for each number of virtual clients, I have 5 output files per memaslap client machine. I aggregate the throughput of the two client machines over the 5 instances and then compute the average and the standard deviation across these 5 iterations. As seen from the graph in Figure 3, the aggregated throughput first increases significantly when the number of virtual clients are varied from 8 to 32. From 32 to 88, the increase isn't as pronounced and is modest. After this point, the throughput starts saturating. Although it is increasing with the number of virtual clients, which is somewhat expected as the more the number of clients, the more the number of requests per second, but the overhead due to the large number of requests is starting to show as the slope is starting to flatten. With this graph as reference, I would say the throughput saturates after the number of virtual clients are more than 96.

However, I would also like to mention that the memcached's behaviour isn't what I expected as I hoped to see a smooth function in the curve. But the increments correspond to a piecewise linear function. Despite multiple runs, the graph still isn't devoid of such behaviour.

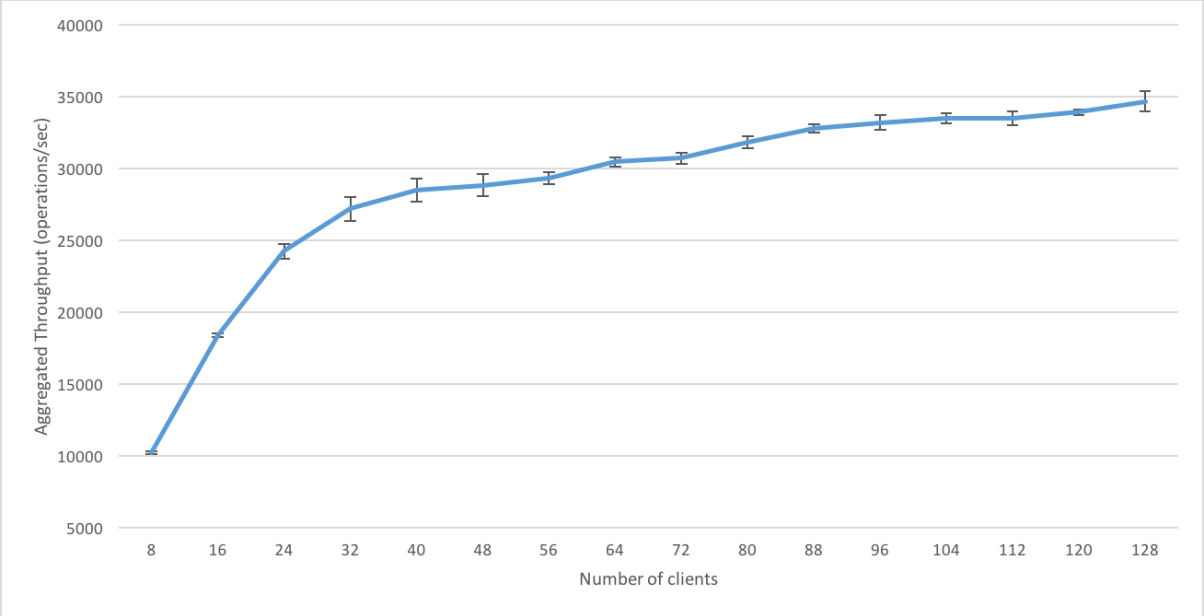

Figure 3: Baseline Throughput

2.2 Response time

Here, for each number of virtual clients, I have 5 output files per memaslap client machine. I average the response time across the 5 iterations and then average the output of both the memaslap machines as well. To compute the standard deviation, I repeat the same process across 2 clients with 5 iterations but I take the standard deviation values from memaslap and then compute the average by taking the square root of the average of the variance(square of standard deviation).

The average response time graph in Figure 4 shows a steady increase, which I expected as the increase in the number of requests may increase the waiting time of requests, but it gives no concrete evidence of saturation in itself. The increase in the number of clients also increases the standard deviation of the response times. At this point, I also see why it makes less sense to look at the average response time plot and why looking at something like a percentile distribution of the completion of requests vs time is more informative about the behaviour of the system.

3 Stability Trace

I wrote a little script to run this experiment with the specifications mentioned in the following table. I ran these experiments on 6 basic A2 machines and 1 basic A4 machine on Azure. I started all the 3 memaslap clients at the same time. I run each trace for 80 minutes to account for the warmup and cool down phase and plot the part after the warmup phase for 1 hour. However, the logs in my repository have information for 80 minutes.

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 in pool	8
Runtime x repetitions	1h x 3
Log files	trace1, trace2, trace3, instrumentlog

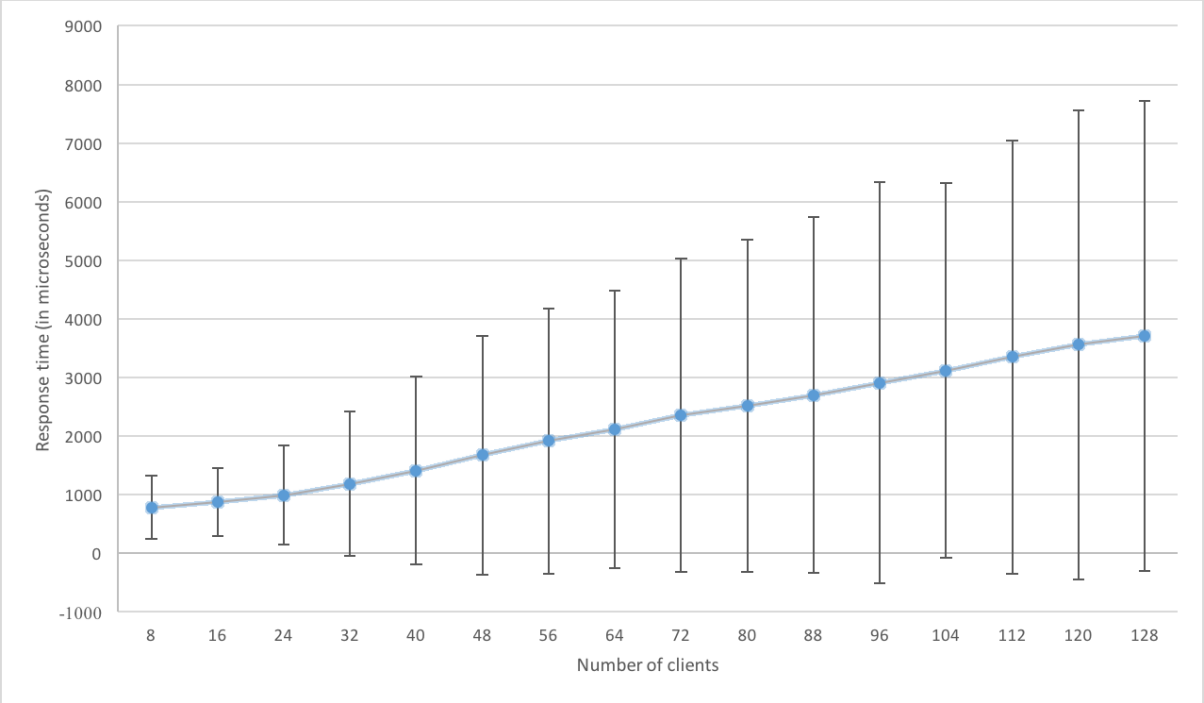

Figure 4: Baseline Average Response Time

3.1 Throughput

The graph in Figure 5 is generated from the **trace1** log. I log the throughput from memaslap every second but provide a throughput graph averaged every 10 seconds. For most part, the throughput remains stable at around 11000 operations per second. But as you can see in the graph, there are a few substantial dips for a few seconds. I tried to find out as to why this might have happened. I looked in my log file and found that the throughput drops to 3000-4000 for a few seconds and then boosts back to 10000-12000. For what it is worth, the dips don't exhibit any periodic behaviour. Yet for my sanity check, I tried monitoring the garbage collector but it doesn't seem to be the reason for the low throughput at this instance. In order to solidify my reasoning, I ran the traces 2 more times at different times of the day as I suspected that the traffic on Azure could have led to these random spikes in performance. The trace in **trace1** was run at 7 pm, the one in **trace2** was run at 11pm and the one in **trace3** was run around 10am.

From Figure 6, I inferred that the dips aren't caused by my middleware. It is very likely that they may happen because of the delay in the network in the cloud as some of my VMs were allocated to different subnets in Azure. I tried to find concrete proof for this and the only feasible evidence I could gather was that I observed momentary dips and rises in the CPU performance in the overview graph of the Azure VM I was running the memcached server on.

In order to further solidify my hypothesis, I ran the baseline trace (**basetrace**) for an hour without my middleware and I observed that even without the middleware, the baseline traces have random spikes as seen in Figure 7.

3.2 Response time

For most part, the average response time remains stable at around 18,000 microseconds in Figure 8. However, as in the case of the throughput, the response time also unsurprisingly has spikes at the same time period where the throughput drops. This can also be seen by the fact that the standard deviation for this period increases as well and is around 150,000 microseconds. Through the instrument log, I notice that during this time, the request spends most of its time in the queue which makes sense as 99% of the requests are GET operations which are synchronous and because the previous requests haven't been satisfied, they aren't pulled from the queue.

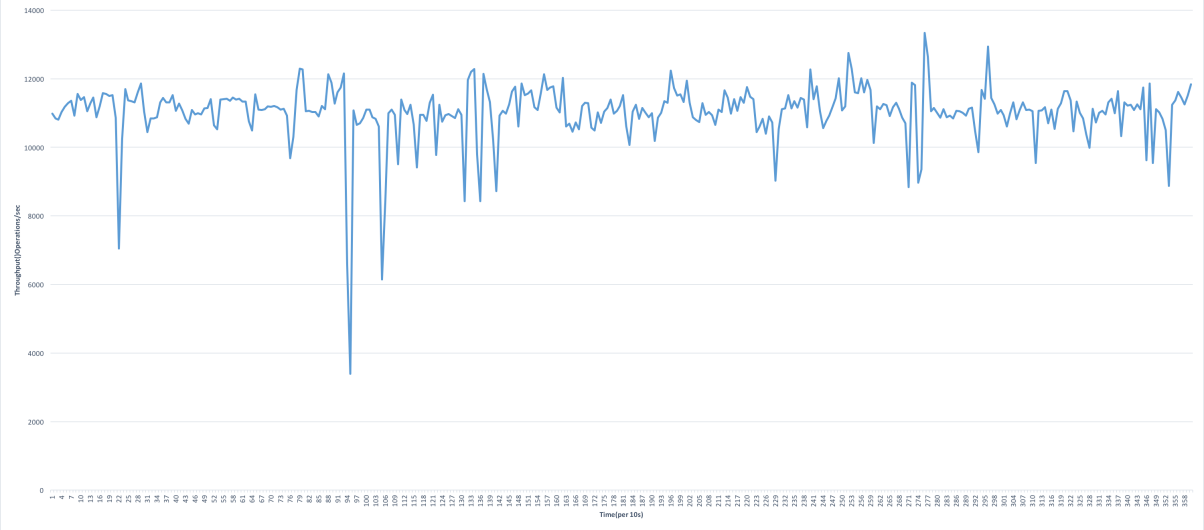

Figure 5: Throughput with Middleware

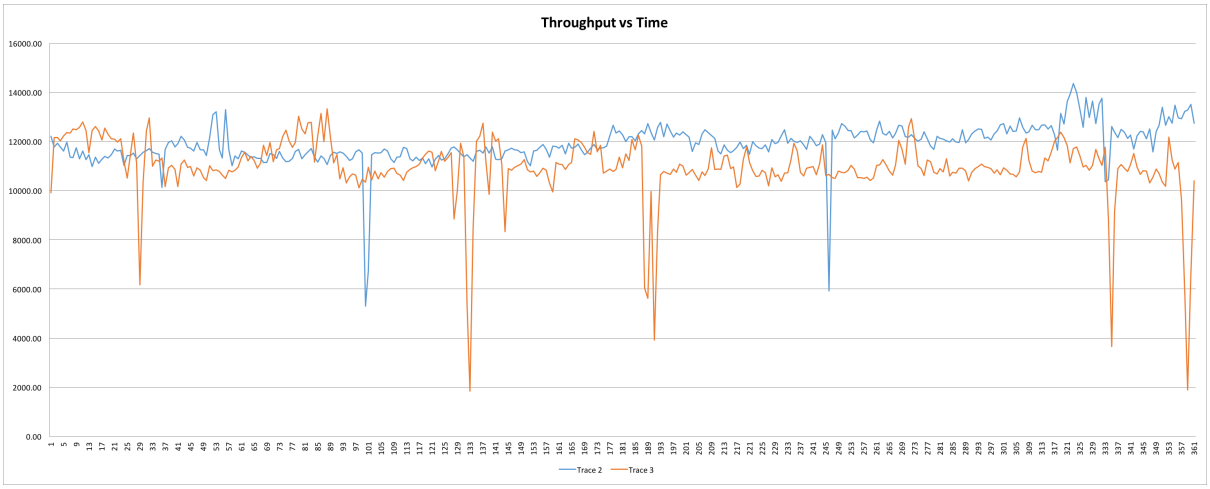

Figure 6: Comparison of trace2 and trace3 run at different times. The throughput is averaged every 10 seconds

3.3 Overhead of middleware

I expect the middleware to introduce overheads. After looking at the output of **instrumentlog**, I found that for a SET request, the most time consuming chunk in the T_{mw} is from the T_{server}. This can be explained by the fact that since we use a replicate to all strategy, the SET response has to wait until it receives a response from all the primary and secondary servers. For a GET request, however, the most time consuming operation in T_{mw} is the T_{queue}. This can be explained by the fact that since we use synchronous worker threads to complete GET requests, each request in the queue has to wait until a worker from the thread pool has completed the previous request.

Even though the experiments for the baseline and the middleware are run at different specifications and parameters, I compare this overhead in the following manner: Since the baseline experiment reaches a saturation point around 96 to 104 virtual clients with a throughput of about 33000 ops/sec and an average response time of 3000 microseconds for one memcached server, I assume that with 192 clients baseline experiment would reach a throughput of around 36000 ops/sec and have response time near 4100 microseconds.

The Middleware trace uses 3 memaslap machines with 64 virtual clients each, making a total of 192 virtual clients and 3 memcached servers. So, for comparing the throughput with the baseline, based on

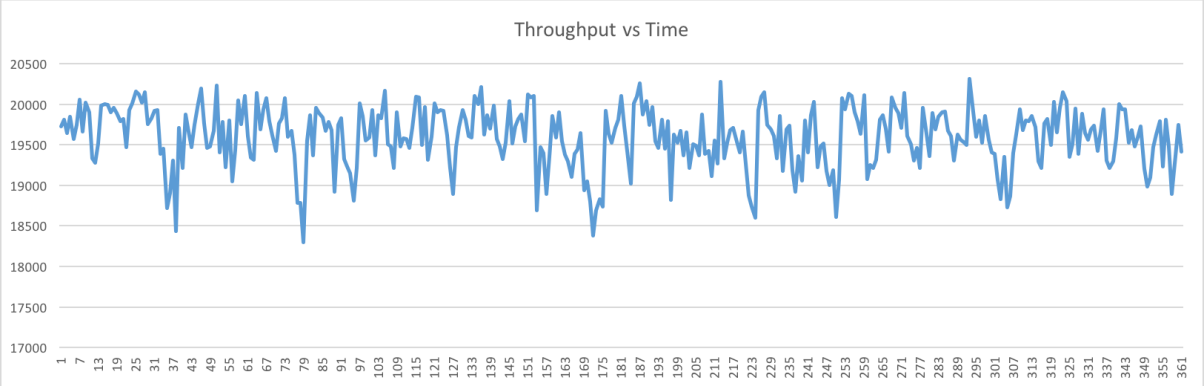

Figure 7: Baseline trace for 60 min, 1 client with 64 virtual clients and 1 server

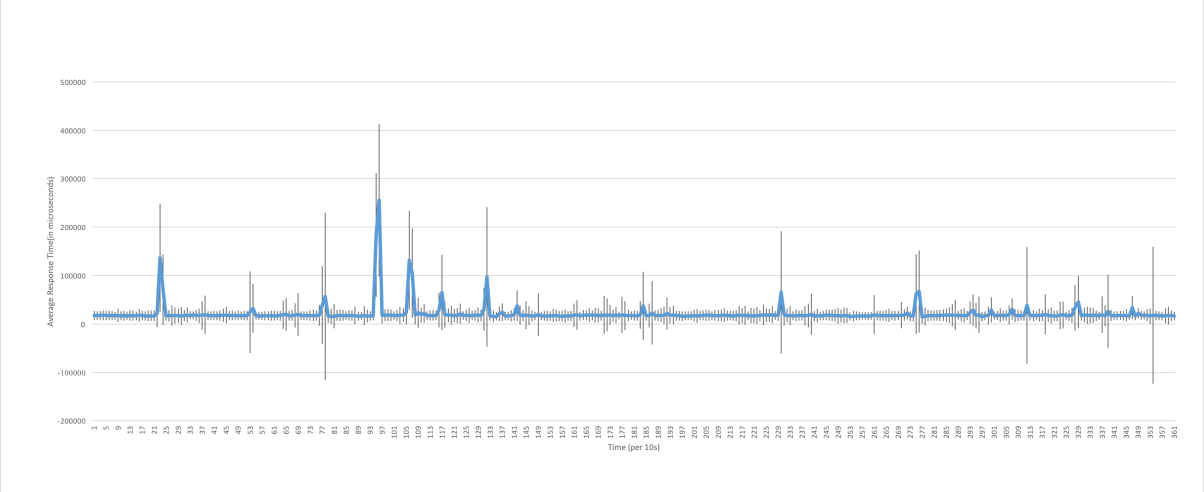

Figure 8: Average Response Time with Middleware

the notion that my hash has uniform distribution, I divide my throughput by 3. The average throughput of my experiment with the middleware based on the trace1 data is 11032.59 and the average response time is 21278.45 microseconds. After dividing by 3, the throughput is 3677.53.

	Throughput	**Response Time**
Baseline	3677.53	21278.45
With MW	36000	4100
Overhead by	9.79 times	5.19 times

Logfile listing

Short name	Location
baselinebench	https://gitlab.inf.ethz.ch/ragrawal/.../BaselineLogsSmallKey.zip
trace1	https://gitlab.inf.ethz.ch/ragrawal/.../trace1.zip
trace2	https://gitlab.inf.ethz.ch/ragrawal/.../trace2.zip
trace3	https://gitlab.inf.ethz.ch/ragrawal/.../trace3.zip
basetrace	https://gitlab.inf.ethz.ch/ragrawal/.../baselinetrace1h1server1client.zip
instrumentlog	https://gitlab.inf.ethz.ch/ragrawal/.../instrumentlogtrace1.zip