Advanced Systems Lab (Fall'16) – First Milestone

Name: ***Antonios Karkatsoulis***
Legi number: ***15-925-928***

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

The system we implemented consists of a middleware application that handles GET, SET and DELETE requests from *memcached* clients, forwarding them to a number of *memcached* servers, while ensuring proper load balancing across them, as well as replication of the SET and DELETE operations to a given number of servers.

The abstract architecture that had to be implemented is shown in the following diagram. Along with it, we mark the points where we instrumented our implementation to save timestamps for each request.

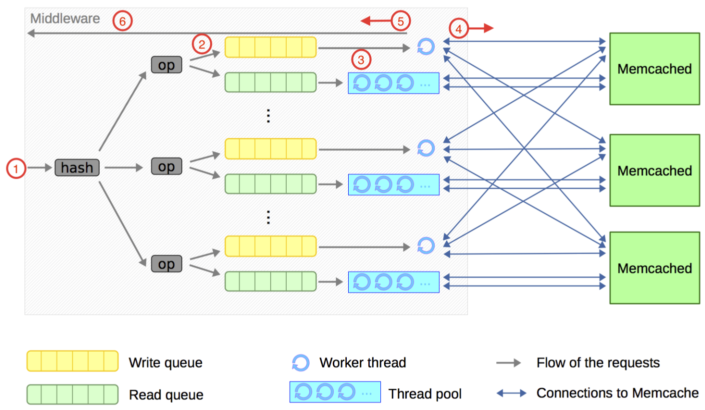

Figure 1: Architecture of the system with markers where the code is instrumented.

Marker ID	Timestamp Name	Description
1	$T_{requestreceived}$	Time when the request is first received by the middleware.
2	$T_{enqueued}$	Time when the request is enqueued by the main thread.
3	$T_{dequeued}$	Time when the request is dequeued by the SET or GET thread.
4	$T_{senttoserver}$	Time when the request is sent to the first server.
5	$T_{receivedfromserver}$	Time when last server sends its response back to the middleware
6	$T_{responsesent}$	Time when the response is sent back to the client.

Table 1: Description of the markers shown in Figure 1.

As far as the implementation of the architecture is concerned, the middleware consists of:

- A main thread[1] that is responsible for receiving requests from the clients using Java NIO, decoding the operation, hashing the key of the request and putting it into the respective queue.
- A GET and a SET queue for each of the servers for the respective requests.
- A custom pool of threads per server to handle the GET requests concurrently and synchronously.[2].
- A thread per server that handles the SET requests asynchronously.[3].
- A class that describes the memcache requests and keeps all the timestamps for the requests, as well as their type, key etc.[4]

[1]`https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/src/asl/Middleware.java`
[2]`https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/src/asl/GetThread.java`
[3]`https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/src/asl/SetThread.java`
[4]`https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/src/asl/MemcacheRequest.java`

The main design decisions include: Firstly, the use of NIO throughout the implementation for uniformity, as well as performance reasons. Secondly, the use of one SET thread that carries out both the tasks of removing requests from the queue and processing the responses from the servers. If there was freedom for the design we would have used two separate threads to carry out these separate tasks.

1.2 Load Balancing and Hashing

Our middleware is responsible for load balancing the requests across the servers as uniformly as possible. For this purpose we implemented a consistent hashing mechanism[5], where all the possible outputs of our hash function (the set of integers from 0 to $2^{32} - 1$) are represented along a circle. The memcached servers are also uniformly distributed across the circle. The key of the incoming request is hashed and the server immediately following the point of the hash is selected. We should note here that consistent hashing is mostly used in distributed load-balancing systems and has the advantage of adapting to server failures without needing to remap the keys for the remaining servers. Although in our scenario, no server failures are assumed, we still decided to use consistent hashing as it suits a real-world scenario better.

As a hash function we chose **CRC32**, that is simple, fast and outputs a fixed 32-bit output that covers the integer space. We know that a proper hash function outputs uniformly distributed hashes, when the inputs are not skewed. Since *memaslap* outputs randomly distributed keys, we also expect a uniform distribution of the requests across the servers. This is based on a property of the function called *2-universality*. Other stronger cryptographic hash functions such as *SHA-1* and *MD5* were not considered because in general they are costly.

We carried out a small experiment to find out how well our chosen hash function distributes the values produced by memaslap to 3 servers. We ran memaslap for 1 minute and below we plot the distribution of the requests to the servers. The experiment was carried out 5 times.

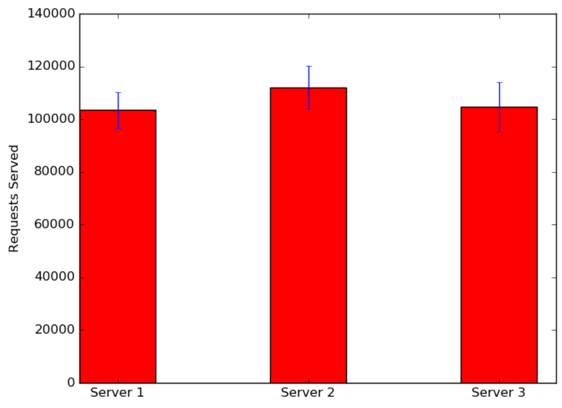

Figure 2: Distribution of the requests across the servers with CRC32 as a hash function.

1.3 Write Operations and Replication

For the writes (SET and DELETE operations) a single thread per server is responsible for 2 tasks:

- Taking a request from the queue, if any exists, and forwarding it to the servers dictated by the replication factor R. If R is 1 the request is only forwarded to the server chosen by the hash, while if it is greater than 1, the request is forwarded to the main server, as well as to the next *R-1* servers in the consistent hash circle as defined above.
- Listening for responses from the servers. The thread uses an NIO selector where each channel corresponds to the connection to each server. If one or more responses is present at the channel it processes them, otherwise it goes on to pick another element from the queue.

The main problem that we faced in the replicated scenario would be to find a way to match the replies from the servers with the requests sent, in order to send a single response back to the client when

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

all servers have replied to a certain request. Our solution to this problem was to create a FIFO queue per server that holds the requests for which there are pending responses. Once a request is sent to the servers, replicas of it are also put in the respective queues and a counter is kept per request of how many responses are pending for it to complete. Once a response arrives, the thread finds out from which server it was sent and removes the request from the respective queue. When all instances of the request have been removed from the queues, a response is ready to be sent back to the client. In general, the choice of a FIFO queue here is valid since we assume that the memcached servers are well behaved in the sense that they respond to requests in the order they receive them.

A simple estimate of the response time for a write operation can be given by:
$RTT_{request} = RTT_{client \to mw} + time_{mw} + \max_{r \in R}(RTT_{mw \to server_r} + time_{server_r})$
where R is the set of the replication servers for the request.

A SET operation in the general replicated case will be slower with high probability, as now for every write we have to wait for replies from all the servers, which are involved in the replication. The network now plays a lot more important role and with replication factor R, 1 out of R delayed responses is enough to delay the whole request.

Moreover, since the system is closed and each client waits for a response before it sends the next request, the rate at which writes can be carried out in the system is limited by the rate at which the memcached servers can process them. Multi-threaded memcached servers on stronger machines might be able to process every request faster. Moreover there is always the latency in the network that limits the throughput of the write requests. More specifically if *RTT* is the round-trip time to a server the throughput is lower bounded by *1/RTT*.

1.4 Read Operations and Thread Pool

For the GET operations many threads belonging in a thread pool are responsible for taking requests from a shared queue and forwarding them to the server they are responsible for. They then block waiting for the server response, read it, check whether the GET operation was a miss and respond to the client. A **LinkedBlockingQueue** is used for the implementation of the queues, since we don't have to specify a maximum size explicitly. For inserting and receiving elements from the queue we used the methods *put()* and *take()*, which are thread-safe according to the Java manuals. Thus, there is no way that the queue is accessed by the main thread and the GET threads in an unsafe way. Each thread in the thread pool opens a connection to the respective server by means of an NIO *SocketChannel* and keeps the connection open throughout its life. Moreover, a single **ByteBuffer** is allocated once per thread and reused for all the communication, in order to avoid the more expensive buffer allocation per request.

We decided to implement our own custom thread pool instead of more sophisticated solutions like **ExecutorService** because of its simplicity and the fact that it is simpler for each thread to keep state such as the connection to the servers. The number of threads per GET queue is an important parameter for the performance of the middleware. To find out their impact on the overall throughput we carried out a small experiment, adjusting the number of threads per GET queue from 1 to 32 and having one memaslap client machine with 64 virtual clients send requests for 5 minutes. The results presented in the plot below, indicate that the optimal number of GET threads per queue is 16.

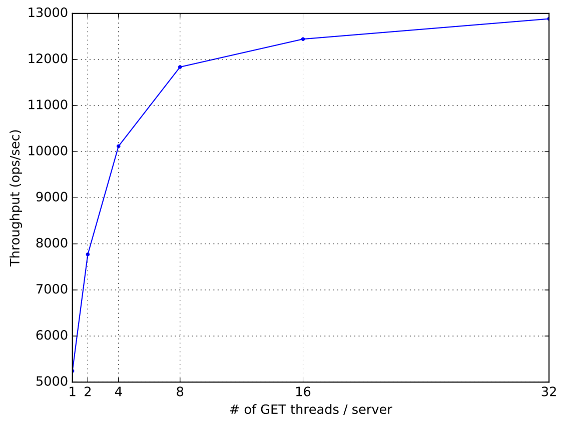

Figure 3: Throughput for different number of threads per GET queue.

2 Memcached Baselines

For the baseline experiments we had *memaslap* send requests directly to the *memcached* servers and measure the average throughput and the response time of the system. The machines that were used were VMs from the Microsoft Azure Cloud. The complete setup of the experiments is as follows:

Server Machine	1 X Basic A2 (2 cores, 3.5 GB memory)
Client Machines	2 X Basic A2 (2 cores, 3.5 GB memory)
Virtual clients / machine	1 to 64 with steps of 1 / machine
Workload	Key 16B, Value 128B, Writes 1%
Middleware	Not present
Runtime x repetitions	30s x 5
Log Folder	baseline_experiments

All the experiments were carried out using a Bash script[6] and all the log files are included as a reference. Also included is a Python[7] script that is responsible for parsing the throughput and response times and creating the plots.

2.1 Throughput

Below we plot the aggregate throughput from the 2 client machines for different numbers of virtual clients. Virtual clients were increased one by one simultaneously in both machines. Exception is of course the case for one virtual client where only one client machine is used.

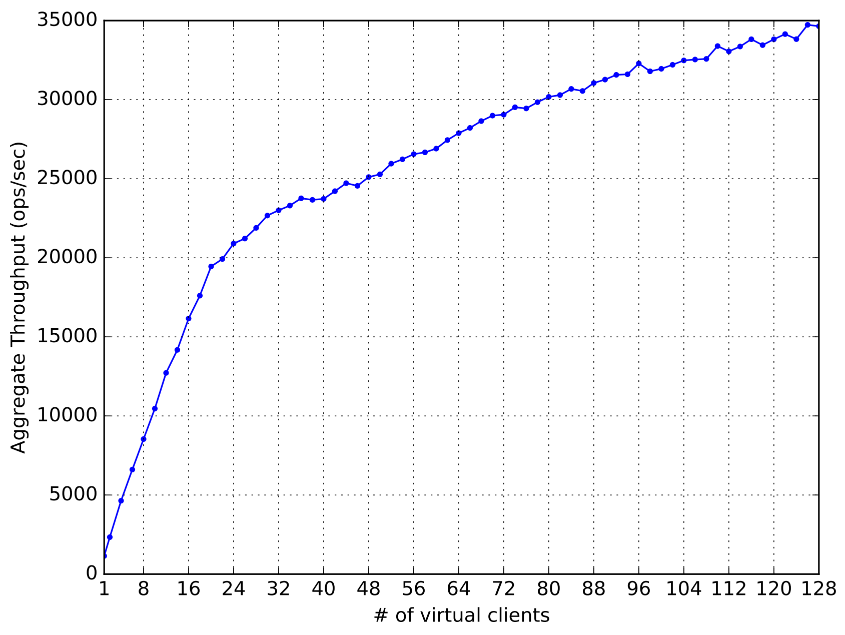

Figure 4: Aggregate througput for different number of virtual clients.

Although it is not very clear from the plot above, the server saturates at about 96 virtual clients. As the client/memcached system is closed, the number of requests in the system at any given time is at most the number of virtual clients. The saturation happens because at some point the servers cannot process so many requests simultaneously and the clients have to wait more time, before they continue

[6]`https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/run_baseline.sh`

[7]`https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/baseline_experiments/parse_plot_baseline.py`

sending requests, limiting the overall throughput. Moreover, more clients means more requests over the network, which as we described is always a bottleneck.

2.2 Response time

Below we present the average response time for the requests along with the standard deviation for the different number of clients.

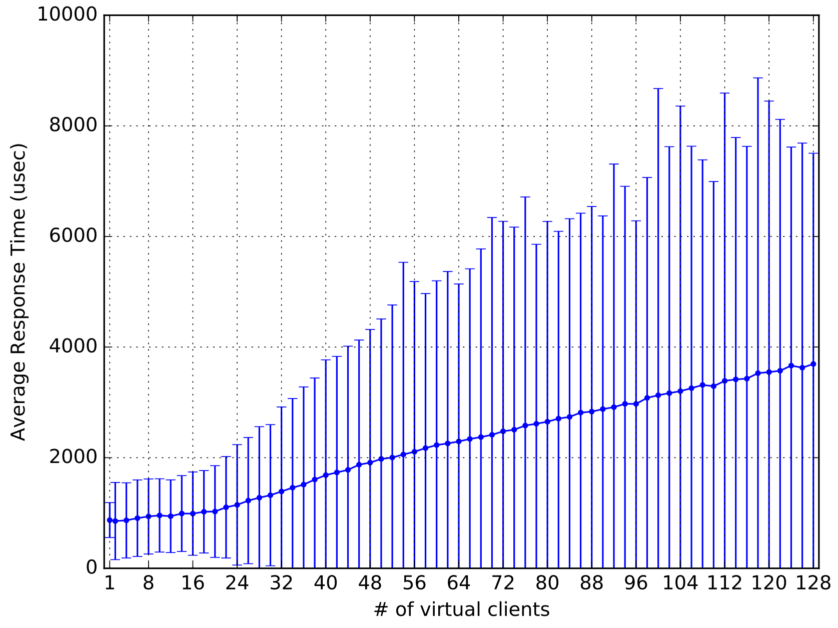

Figure 5: Average response time for different number of clients.

We notice that the response time grows almost linearly with the number of clients. This increase is reasonable, since more clients will naturally saturate the servers and on average requests will take longer to complete.

We also notice that the standard deviation of the response time grows with the number of clients. This can be attributed to the overhead of the network cloud infrastructure, as more clients imply a lot more traffic in the network, which is non-deterministic in cloud settings.

3 Stability Trace

For this part we carried out a stability experiment to prove that our system is functional over long periods of time without degrading performance. The experiment was carried out once on the Microsoft Azure cloud as before and the setup was as follows.

Table 2: Experimental setup for the Stability Trace experiment.

Server Machines	3 X Basic A2 (2 cores, 3.5 GB memory)
Client Machines	3 X Basic A2 (2 cores, 3.5 GB memory)
Middleware Machine	Basic A4 (8 cores, 14 GB memory)
Virtual clients / machine	64
Workload	Key 16B, Value 128B, Writes 1%
Middleware Settings	Replication = 3, Threads per GET queue = 16
Runtime x repetitions	1h x 1
Trace Files	client1 , client2, client3

3.1 Throughput

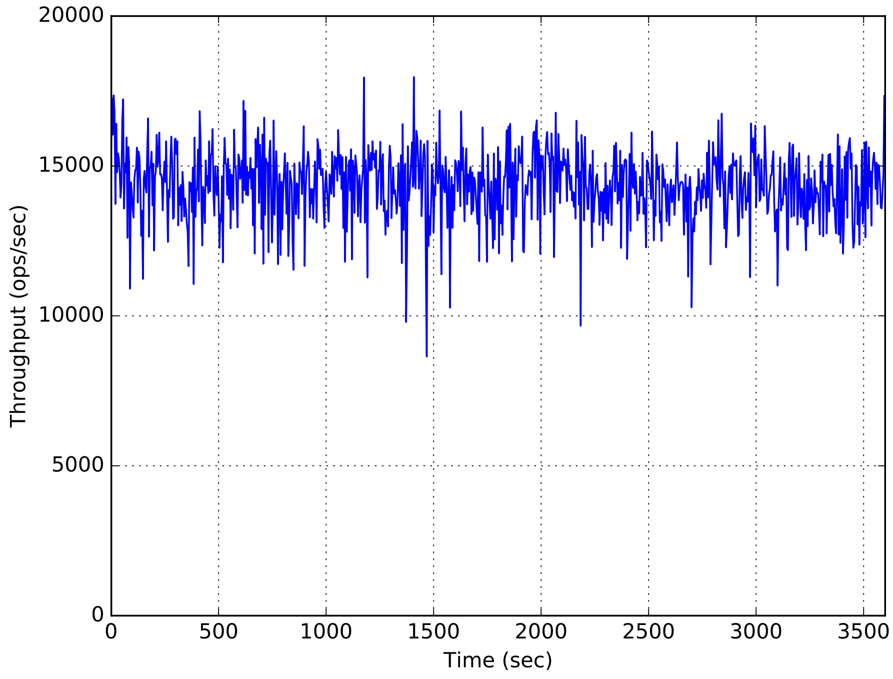

Figure 6: Aggregate throughput for the 3 client machines over 1 hour. (Sampling every 4 seconds)

We notice that the system is quite stable over 1 hour, serving requests at a rate of about 15000 operations per second. There are of course some dips in the performance, some minor and some quite larger. Most of these dips, especially the smaller ones can be attributed to the *Java Garbage Collector* that every now and then consumes CPU time. The bigger dips in performance could be attributed to the *latency of the network*, because, in a cloud environment the time between sending a request to a memcached server and receiving a response can become really high. This has also been confirmed by inspecting the timers for the requests. Since the time for a memcached server to process a request cannot vary that much, the only remaining overhead is the latency for the request to reach the memcached server over the network, a measurement which we are not in the position to take.

Finally, we would like to note here that there seems to be almost no warm-up time in these experiments and this happens because all the middleware resources needed (connections to the memcached servers, threads etc.) are preallocated, so our system can begin serving clients almost right away.

3.2 Response time

The average response time for a request is also quite stable at about 13 ms, with only a few requests taking a lot more than average to complete. The points in time of where these slow requests happen correspond clearly to the throughput dips from the previous figure as expected. Unexpected network latencies result in huge round trip times between the middleware and the servers. We note here that, even if our figure presents one hour of measurements sampled every 30 seconds for presentation reasons, the figure looks very similar even if the sampling is carried out every second, i.e. the spikes remain at the same positions and last for a few seconds. We thus come to the conclusion that it really is a matter of the cloud infrastructure, rather than our middleware.

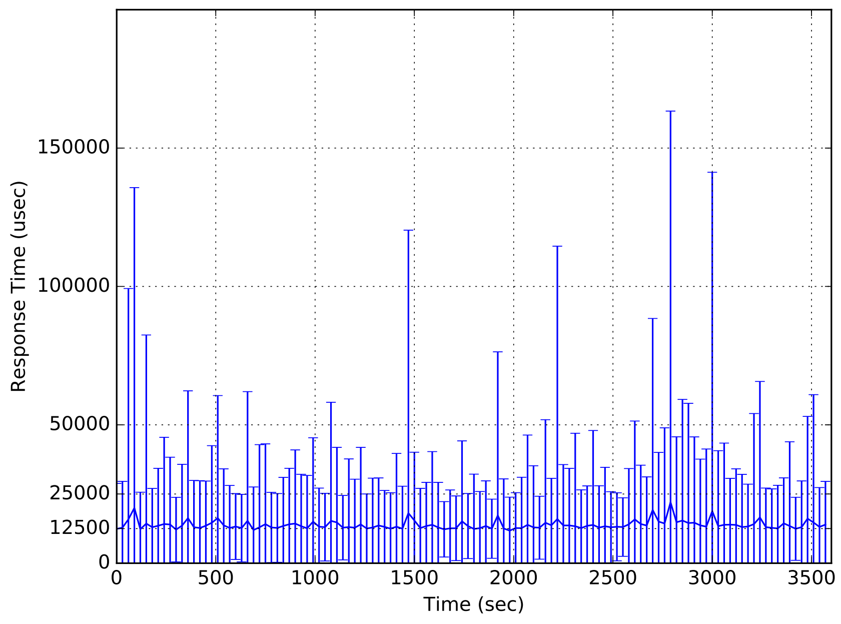

Figure 7: Average response time per request over 1 hour. (Sampling every 30 seconds)

3.3 Overhead of middleware

In this part we are going to compare the performance overhead introduced by our middleware. Unfortunately, the baseline experiments without the middleware and the experiment with the middleware are based on quite different setups and we cannot compare them directly. In order to make the comparison as fair as possible we make the following assumptions:

- In the experiment with the presence of the middleware we assume uniform load balancing, which means that the total of 192 virtual clients will have their requests evenly distributed across the memcached servers.
- The above assumption holds only for the GET requests, as each SET request in the replicated case is served by more than one servers and the baseline experiments cannot cover this case. But since the number of SET requests is only 1% of the total number of requests we can easily ignore them for our comparison.

Under these assumptions each server will serve about 64 virtual clients and the performance metrics of 3 client machines being served by 3 server machines with the middleware present can be compared to the metrics of 1 client machine (with 64 virtual clients) being served by 1 server machine.
We present the overheads in the table below:

Table 3: Overview of the overheads our middleware introduces.

Metric	Without Middleware	With Middleware	Overhead
Aggregate Throughput (ops/sec)	27885	4770	584 %
Response Time (usec)	2294 ± 2848.3	13413 ± 15502.1	585 %

In conclusion, our middleware introduces an overhead of about 600% both in throughput and response time. This can be attributed to the following factors:

- The fact that with the middleware each request has to traverse 2 links instead of 1 (one to the middleware and one from the middleware to the servers and back).
- The overhead introduced by the middleware itself, i.e the creation of a new Java object per request, the time the requests spend in the queues and the hashing of the keys.

Logfile listing

Short name	**Location**
client1	`https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/trace/client1.txt`
client2	`https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/trace/client2.txt`
client3	`https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/blob/master/trace/client3.txt`
baseline_experiments	`https://gitlab.inf.ethz.ch/akarkats/asl-fall16-project/tree/master/baseline_experiments`

Note: Baseline experiments in the Gitlab folder are in the form:
client_*[client_machine]*_vc_*[# of virtual clients*]_run_*[run]*
where:
[client_machine] : The id of the VM the client runs into (1 or 2)
[# of virtual clients] : Number of virtual clients per VM (-c option in memaslap)
[run] : Run of the specific experiment (1 to 5)