

Advanced Systems Lab (Fall'16) – First Milestone

Name: *Taivo Pungas*
Legi number: *15-928-336*

Grading

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

1.2 Load Balancing and Hashing

The hashing is implemented by [UniformHasher](#). The hashing scheme for a given key works as follows:

1. s is hashed into a 32-bit signed integer i using Java's native `String.hashCode()`.
2. The index of the primary machine is calculated as $i \bmod N$ (adding N if the modulus is negative) where N is the number of servers.

The uniformity of hashing was also validated in tests (see [UniformHasherTest](#)). For 1 million random strings and 13 target machines, the distribution to different machines was the following:

Machine	0 got	77229 hits.
Machine	1 got	76702 hits.
Machine	2 got	76769 hits.
Machine	3 got	76860 hits.
Machine	4 got	76773 hits.
Machine	5 got	77169 hits.
Machine	6 got	76650 hits.
Machine	7 got	76831 hits.
Machine	8 got	77061 hits.
Machine	9 got	76955 hits.
Machine	10 got	76644 hits.
Machine	11 got	77432 hits.
Machine	12 got	76925 hits.

As apparent, the distribution is indeed uniform.

Selection of replicated machines for a given replication factor R was done by first selecting the primary machine using the scheme described above, and then selecting the next $R - 1$ machines for replication. E.g. for a setup with 8 memcached servers and $R = 5$, a key whose primary machine is 5 would be replicated to machines 6, 7, 0, and 1.

1.3 Write Operations and Replication

The write operations are handled by [WriteWorkers](#). Each [WriteWorker](#) runs on one thread and has exactly one connection to each memcached server it needs to write to, so in total R connections (where R is the replication factor).

[WriteWorker](#) runs an infinite while-loop in which it does two distinct things.

Firstly, if there are any requests available in the queue of SET-requests, it removes one request r from the queue. It then writes to each of the replication servers in a serial manner without waiting for a response, i.e. it writes the whole SET-request to the first server, then to the second server, and so on. For the non-replicated case, only one request is sent.

Secondly, [WriteWorker](#) checks all memcached servers to see if any of them have responded. This is done in a non-blocking manner using `java.nio`: if a server is not yet ready to respond, other servers will be checked; if no server is ready to respond, the first step is run again. [WriteWorker](#) keeps track of all responses to the same request and once all servers have returned a response, the worst out of the R responses is forwarded to the client. For the non-replicated case, this process reduces to just forwarding the response from memcached to the client.

TODO: Estimate of latencies – replicated and non-replicated Give an estimate of the latencies the writing operation will incur, and generalize it to the replicated case. What do you expect will limit the rate at which writes can be carried out in the system (if anything)?

1.4 Read Operations and Thread Pool

The read operations are handled by [ReadWorkers](#). Each [ReadWorker](#) runs on one thread and has exactly one socket connection to its assigned memcached server.

Every [ReadWorker](#) runs an infinite while-loop in which it takes a request r from its assigned queue of GET-requests, writes the contents of r to its assigned memcached server, blocks until the response from memcached arrives, sets the response buffer of r to what it received from memcached and finally sends the response to the client corresponding to r .

Since multiple [ReadWorkers](#) read from the queue of GET-requests concurrently and [LoadBalancer](#) is inserting elements at the same time, the queue needs to be safe to concurrent access by multiple threads. For this reason, BlockingQueue was chosen; in particular, the ArrayBlockingQueue implementation of BlockingQueue. The maximum size of the queue was set to a constant 200 (defined in MiddlewareMain.QUEUE_SIZE), because 3 load generating machines each with 64 concurrent clients can generate a maximum of $3 \cdot 64 = 192 < 200$ requests at any time, which in the worst (although unlikely) case will all be forwarded to the same server.

2 Memcached Baselines

TODO:

This section will report experimental results. All such parts will start with a short description of the experimental setup. The log files should be identified by a short name, or number, which will be explicitly listed at the end of the document (see Logfile Listing at the end). **If this table is missing or the logfiles listed can't be found in your repository the experiment could be considered invalid, and no points will be awarded!** For baseline measurement of memcached provide **two** graphs (Section 2.1 and 2.2), one with aggregated throughput and one with average response time and standard deviation as a function of number of virtual clients. Increase these in steps from 1 to 128.

TODO: Give a short explanation of memcache's behavior and find the number of virtual clients that saturate

TODO: Explain high std: Upon examining logs manually, lalala distribution

Number of servers	1
Number of client machines	1 to 2
Virtual clients / machine	0 to 64 (see footnote ¹)
Workload	Key 16B, Value 128B, Writes 1%
Middleware	Not present
Runtime x repetitions	60s x 5
Log files	baseline-m*-c*-r*

2.1 Throughput

See Figure 2.1.

2.2 Response time

See Figure 2.2.

TODO: mention that even though stdev goes below 0, in reality it can't be there (physically impossible)

TODO: explain why stdev is so high

3 Stability Trace

TODO:

¹For concurrency=1, one client machine was run with 1 virtual client and the other machine was idle.

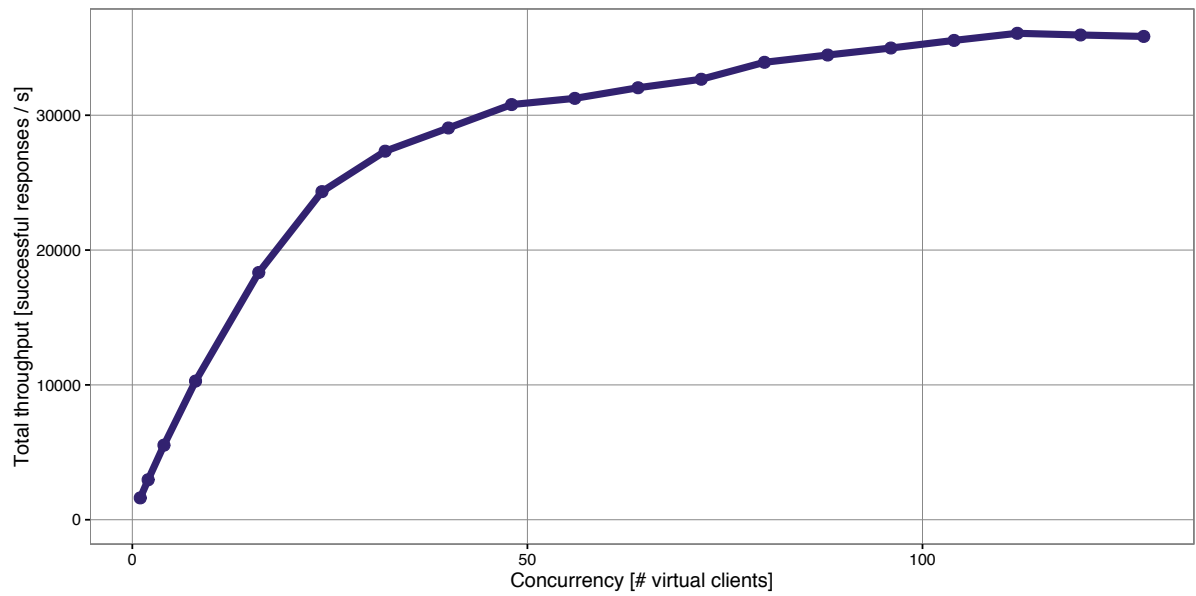


Figure 2: Throughput of memcached without middleware, as measured by memaslap.

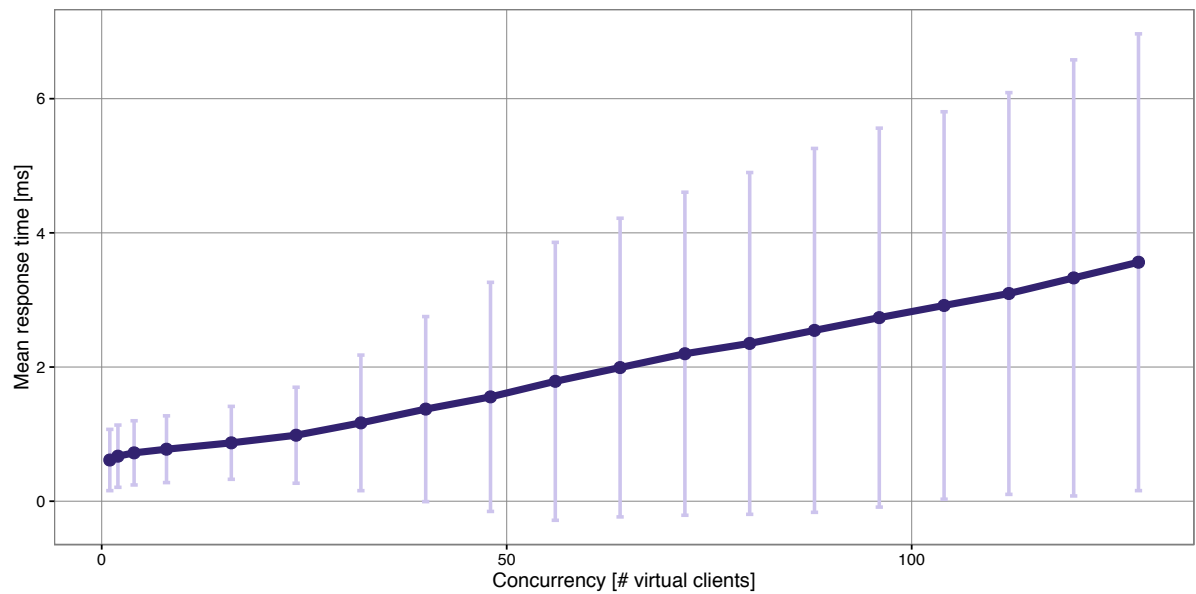


Figure 3: Response time of memcached without middleware, as measured by memaslap.

In this section you will have to show that the middleware is functional and it can handle a long-running workload without crashing or degrading in performance. For this you will run it with full replication for one hour connected to three memcache instances and three load generator machines. You will have to provide two graphs. The x-axis is time and the y-axis is either throughput or response time. Include standard deviation whenever applicable.

TODO: mention that even though stdev goes below 0, in reality it can't be there (physically impossible)

TODO: explain why stdev is so high

TODO: explain that first 5 minutes of experiment are not shown in graph – warmup 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	65min x 1
Log files	trace-ms4, trace-ms5, trace-ms6, trace-mw, trace-req

3.1 Throughput

See Figure 3.1.

3.2 Response time

See Figure 3.2.

3.3 Overhead of middleware

Compare the performance you expect based on the baselines and the one you observe in the trace and quantify the overheads introduced by the middleware (if any), Look at both response time and achievable throughput when making the comparison. Provide an overview of the overheads in a table form.

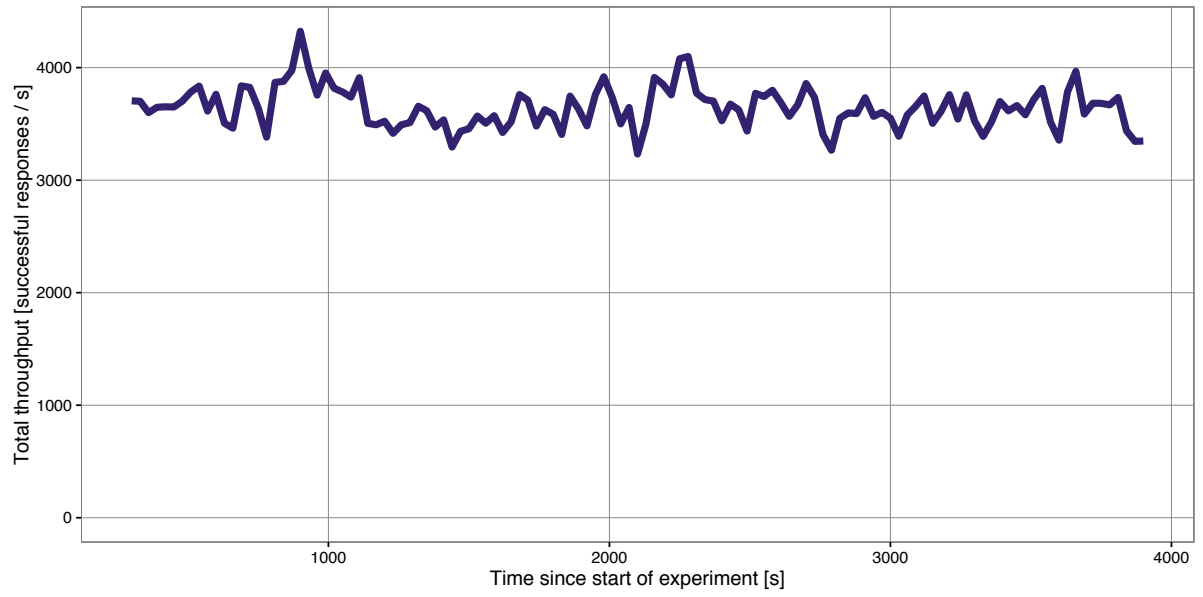


Figure 4: Throughput trace of the middleware as measured by memaslap.

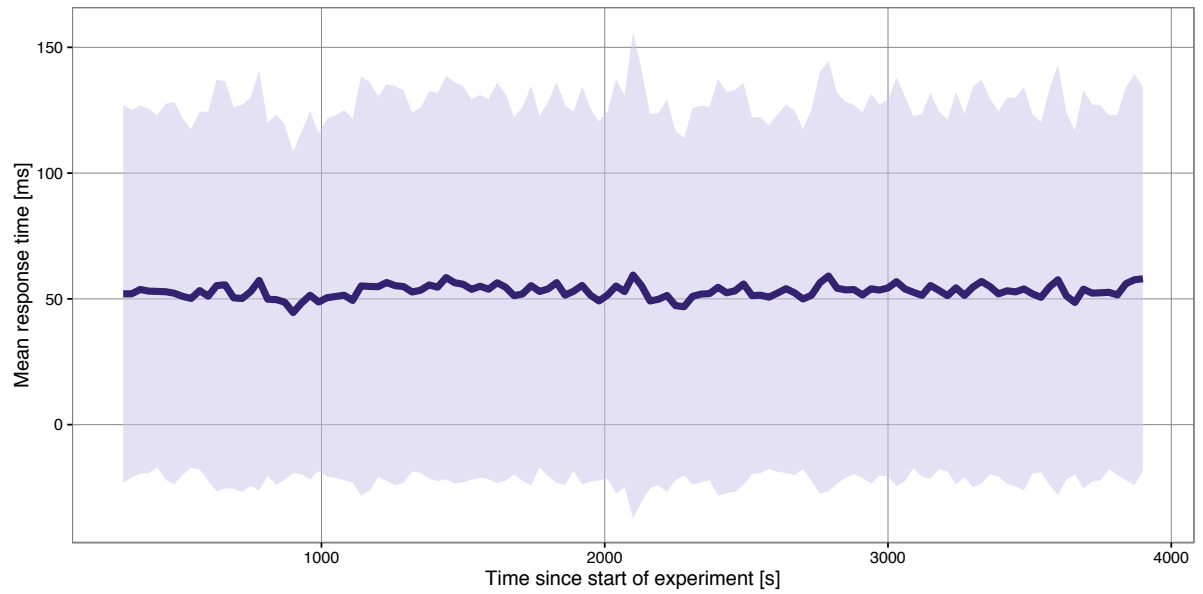


Figure 5: Response time trace of the middleware as measured by memaslap.

Logfile listing

Short name	Location
baseline-m*-c*-r*	gitlab.inf.ethz.ch/.../results/baseline/baseline_memaslap*_conc*_rep*.out
trace-ms4	gitlab.inf.ethz.ch/.../results/trace_rep3/memaslap4.out
trace-ms5	gitlab.inf.ethz.ch/.../results/trace_rep3/memaslap5.out
trace-ms6	gitlab.inf.ethz.ch/.../results/trace_rep3/memaslap6.out
trace-mw	gitlab.inf.ethz.ch/.../results/trace_rep3/main.log
trace-req	gitlab.inf.ethz.ch/.../results/trace_rep3/request.log