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 System Description

1.1 Overall Architecture

The most important classes in this implementation all belong to the package main.java.asl and are as follows. Whenever a Java class is referenced, it will link to GitLab, e.g. MiddlewareMain.

- MiddlewareMain is responsible for setting up all parts of the middleware, and creating the ExecutorService with a fixed thread pool of $(T+1) \cdot N + 1$ threads.
- Request is a wrapper class for all GET- and SET-requests.
- LoadBalancer is the front of the middleware. It reads all incoming requests from all clients using java.nio, hashes the requests using UniformHasher and forwards them to the correct servers for writing or reading. More in Section 1.2.
- MiddlewareComponent is a lightweight class that holds the read and write queues for a given server, and starts the threads that process requests from those queues.
- WriteWorker and ReadWorker implement the write and read thread for a given MiddlewareComponent. More in Sections 1.3 and 1.4.

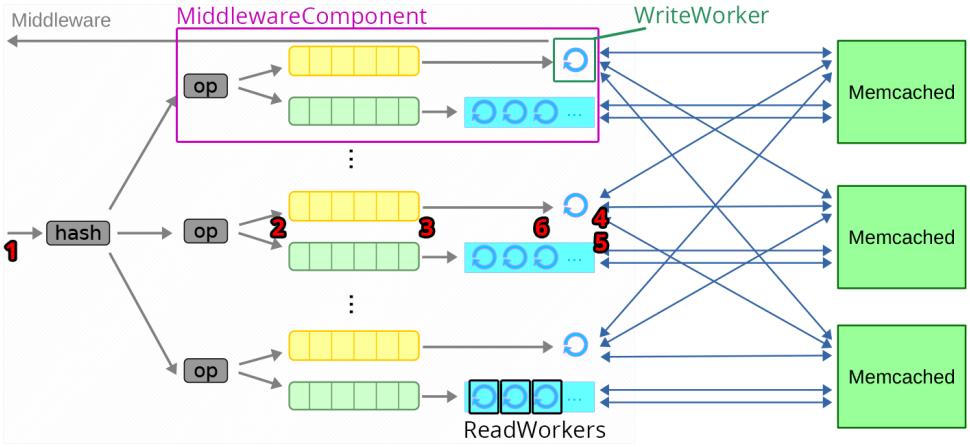

Figure 1: Middleware architecture.

The middleware is instrumented at six points that are also shown in Figure 1.1:

1. $t_{created}$ – when the request is received in LoadBalancer.
2. $t_{enqueued}$ – when the request is added to the queue.
3. $t_{dequeued}$ – when the request is removed from the queue.
4. $t_{forwarded}$ – when the request has been sent to all the servers (one server for GET requests and R servers for SET requests).
5. $t_{received}$ – when responses to the request have been received from all servers.
6. $t_{returned}$ – when the response is returned to the client.

TODO: "Shortly outline the main design decisions?"

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 mod 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

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*

In the baseline experiments, two clients sent requests directly to one memcached server. 37 different values in the range $[1, 128]$ for the total number of virtual clients were tested with 5 repetitions for each. The machines were not restarted between repetitions; however, memcached was restarted after each repetition. Both the clients and the memcached server ran on Azure Basic_A2 machines. All machines were accessed through their private IPs in the virtual network. Logs were parsed using Python and results plotted using R.

2.1 Throughput

From Figure 2.1, we observe that the throughput of memcached grows almost linearly up to 24 virtual clients, from which point on it starts to saturate: throughput increases more slowly with additional virtual clients. At roughly 110 virtual clients, the system is completely saturated and additional virtual clients don't increase throughput any further. From the low standard deviation of throughput we can conclude that these results will not change much with further repetitions.

2.2 Response time

From Figure 2.2 we observe that response time grows slowly up to 24 virtual clients, from which point mean response time starts to grow linearly with the number of clients, and the standard deviation of response time increases significantly. The mean increases because memcached is unable to service all incoming requests immediately so some requests have to wait.

When the system starts to saturate, standard deviation also increases. TODO: WHY? Statistical explanation: heavy tail for this distribution.

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

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

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

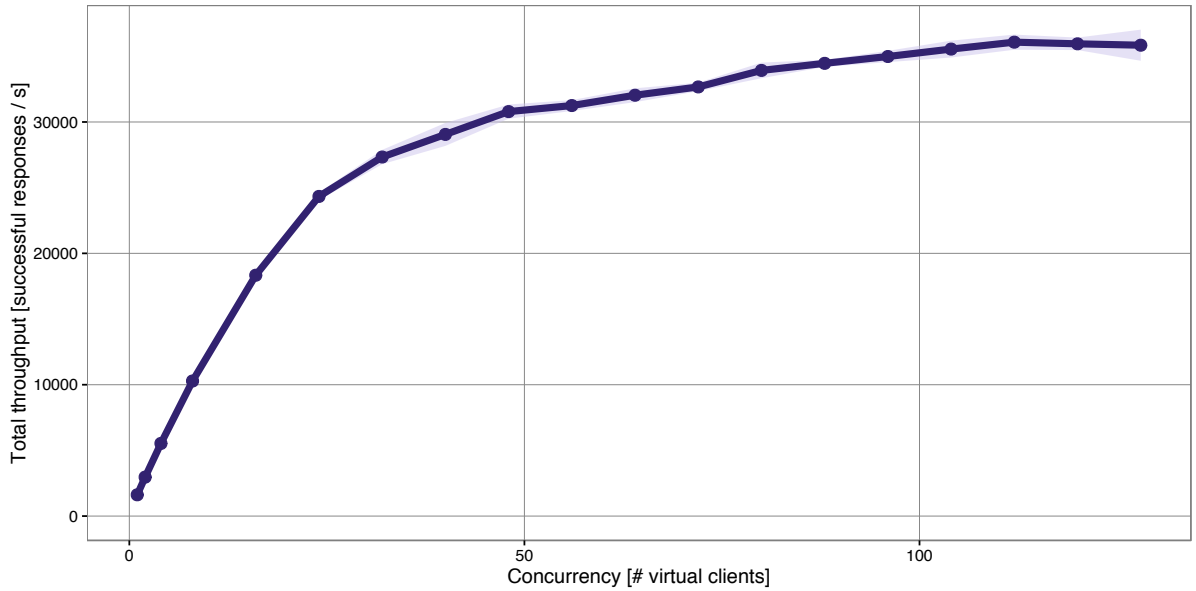

Figure 2: Throughput of memcached without middleware, as measured by memaslap. Dark dots connected by a line show the mean throughput and the light ribbon surronding the line shows ±1 standard deviation of throughput over 5 repetitions.

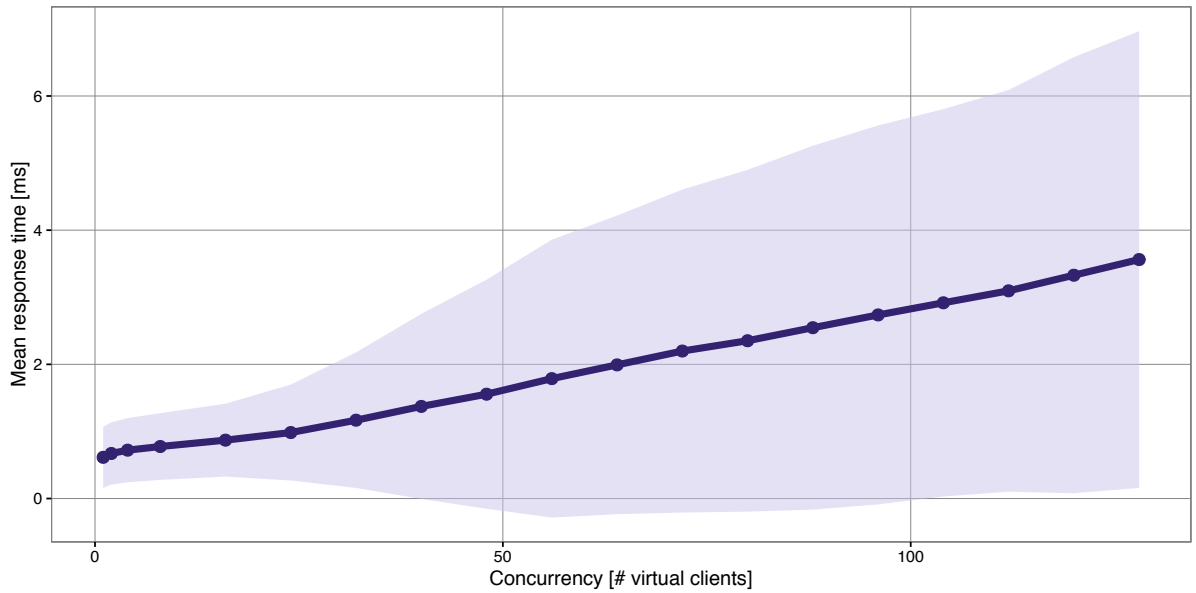

Figure 3: Response time of memcached without middleware, as measured by memaslap. Dark dots connected by a line show the mean response time and the light ribbon surronding the line shows ±1 standard deviation aggregated over all requests sent with that concurrency. Standard deviation values from different repetitions were aggregated by a) calculating the variance for each repetitions, b) finding the weighted average of variance (where the weight is the number of successful requests in a repetition), and c) finding the square root of this sum.

3 Stability Trace

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

The trace was run for 65 minutes; in the plots and analysis, the first 3 minutes and the last two minutes were removed to account for the warm-up and cool-down phase, respectively. The the number of read threads per server was set to $T = 5$ for the trace. Memaslap was set to record statistics in 30-second intervals.

The clients and the memcached server were run on Azure Basic_A2 machines; the middleware was run on a Basic_A4 instance. Both the middleware and memcached servers were accessed through their private IPs in the virtual network. Logs were parsed using Python and results plotted using R.

3.1 Throughput

TODO: Explain that MW is functional and can handle a long-running workload without crashing or degradi

TODO: Explain the spikes in the graph.

See Figure 3.1.

3.2 Response time

See Figure 3.2.

TODO: Explain that MW is functional and can handle a long-running workload without crashing or degr

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

3.3 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.

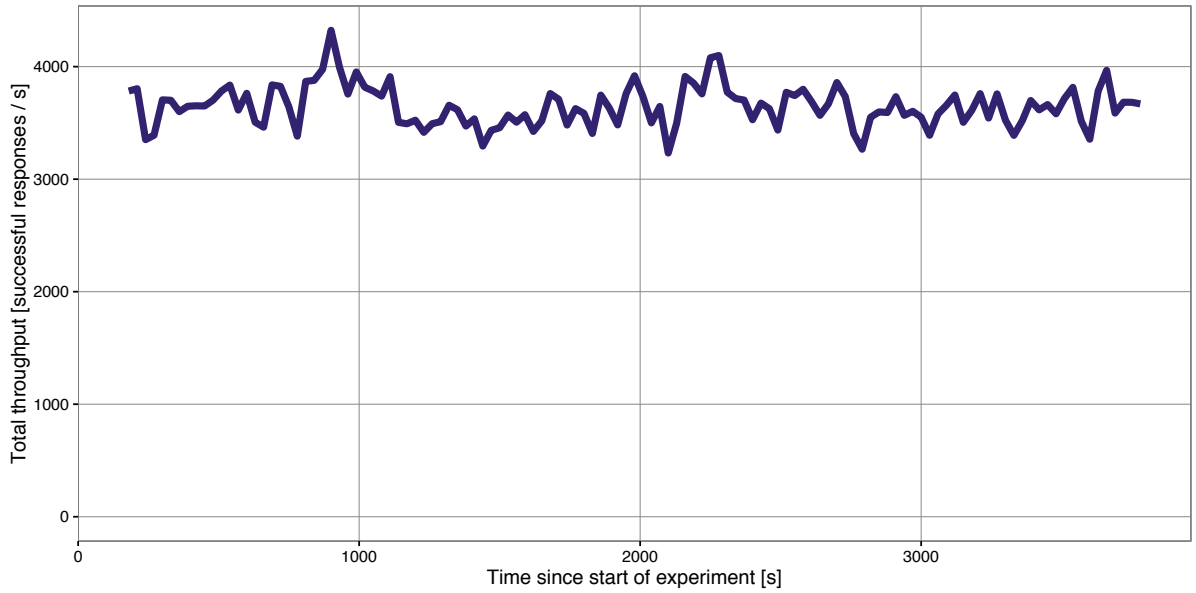

Figure 4: Throughput trace of the middleware as measured by memaslap.

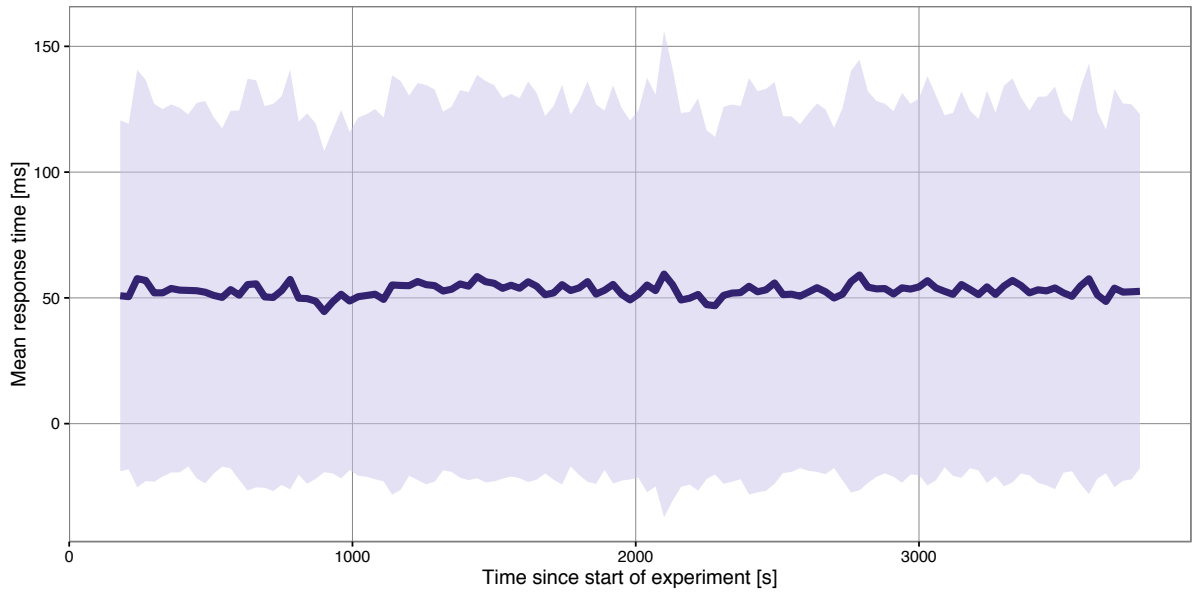

Figure 5: Response time trace of the middleware as measured by memaslap.

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