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, to which all components are submitted.
- Request is a wrapper class for all requests. It contains the request body, the timestamps, and code for responding to the request. It also logs the timestamps for every N-th request where N is a parameter set to 100.
- LoadBalancer is the front of the middleware. It runs an infinite loop in which 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 worker threads that process requests from those queues.
- WriteWorker and ReadWorker implement the write and read thread for a given MiddlewareComponent. WriteWorker also handles DELETE requests. More in Sections 1.3 and 1.4.

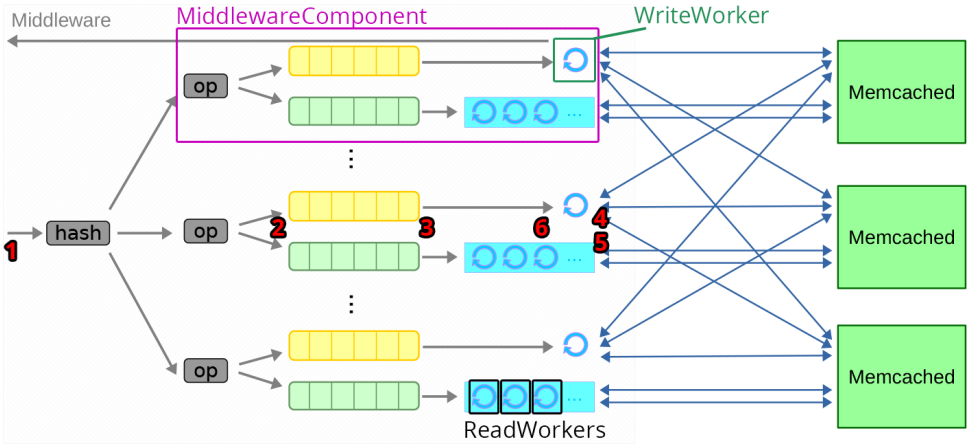

Figure 1: Middleware architecture.

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 write requests and R servers for read 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.

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.

Figure 3.2 shows the amount of time spent in different parts of the middleware. 90% of requests spend less than 10 ms in the queue, and 99% of requests spend less than 20 ms. The roundtrip from middleware to memcached and back also takes a small amount of time: below 6 ms for 90% of requests and below 13ms for 99% of requests. Time spent on all other steps is negligible.

This implies that the maximum rate at which writes can be carried out will be limited by the maximum throughput of the write thread – we cannot add more than one write thread per server. To fix this bottleneck, we could add memcached servers (and keep R fixed), in which case the bottleneck will be waiting for responses from all memcached servers.

The mean response time to write requests in the fully replicated system is 25.8 ms (as measured by memaslap). Without running the trace with $R = 1$, it is hard to estimate the latency for the non-replicated case. However, in my WriteWorker implementation, there are two factors influencing this difference: a) how long it takes to write a single write request to each server, and b) what is the longest response time from of the N memcached servers. a) is small because it consists only of writing a ByteBuffer to a SocketChannel. b) is slightly higher than the mean response time of a single server because we're sampling the same random variable (which has a distribution with a nonzero variance) multiple times and taking the maximum one. In summary: the non-replicated latency will not be significantly lower than in the replicated case in the implemented system.

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 32 virtual clients, from which point on it starts to saturate: throughput increases more slowly

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

with additional virtual clients. This means that the knee of the system is at roughly 32 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 32 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.

The high standard deviation is caused by the heavy-tailed distribution of response times, i.e. the distribution is not normal (nor symmetric). When the system starts to saturate, standard deviation also increases. This is caused by a queue forming at the memcached server: if the number of clients is low, the queue is almost always empty and every request will be processed instantaneously. When we increase the number of clients, the queue will be filled more often, and the amount of items in the queue at any time is varies, which in turn causes the the response time to vary more and thus a high variance in response times.

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 three minutes and the last two minutes were removed to account for the warm-up and cool-down phase, respectively. 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

From Figure 3.1 we can see that the throughput remains at the same level of 7000 to 8000 operations per second throughout the whole experiment and thus the middleware can handle a long-running workload without a degradation in performance. The variance can partly be explained by the random nature of the experiment: it is affected by the conditions (especially network conditions) on Azure. However, the sudden falls in throughput in the graph are likely to be caused by the Java garbage collector starting its operation.

We can use the Interactive Response Time Law to evaluate whether the results are reasonable. Given that in the trace (excluding warm-up and cool-down) the mean response time is $r = 25.8$ms, the number of clients is $n = 3 \cdot 64 = 192$, and the clients can be assumed to have no think time ($z = 0$), the predicted throughput is $\frac{n}{r+z} = 7430$ operations per second. This is well in line with the actual throughput of 7460 operations per second.

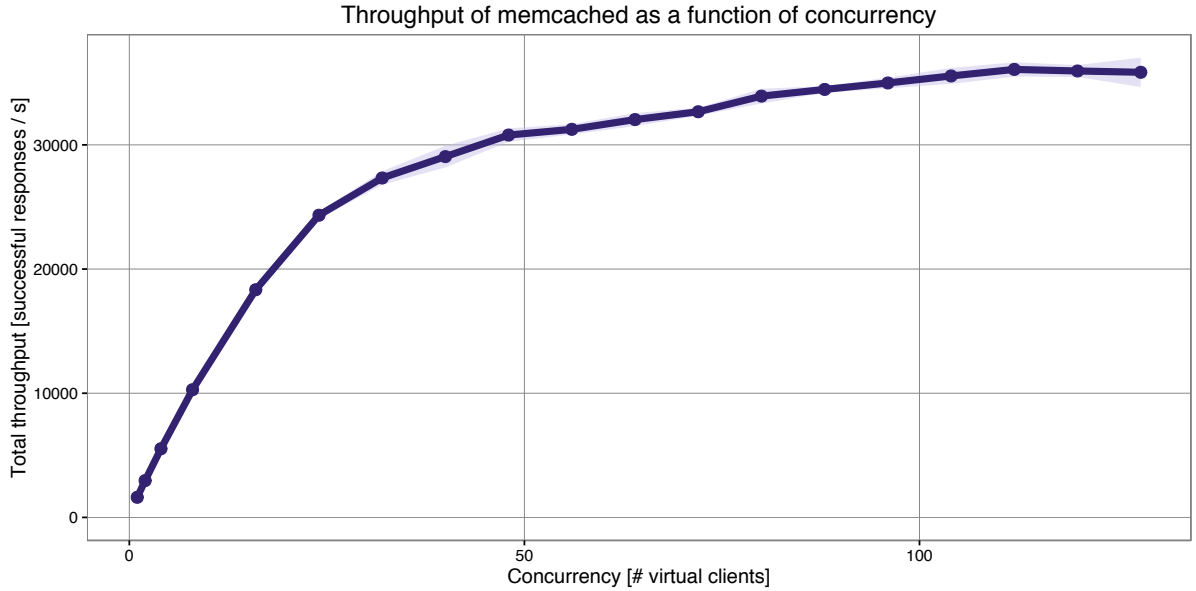

Figure 2: Throughput of memcached without middleware, as measured by memaslap. Dark dots connected by a line show the mean throughput and the light ribbon surronding the line shows ±1 standard deviation of throughput over 5 repetitions.

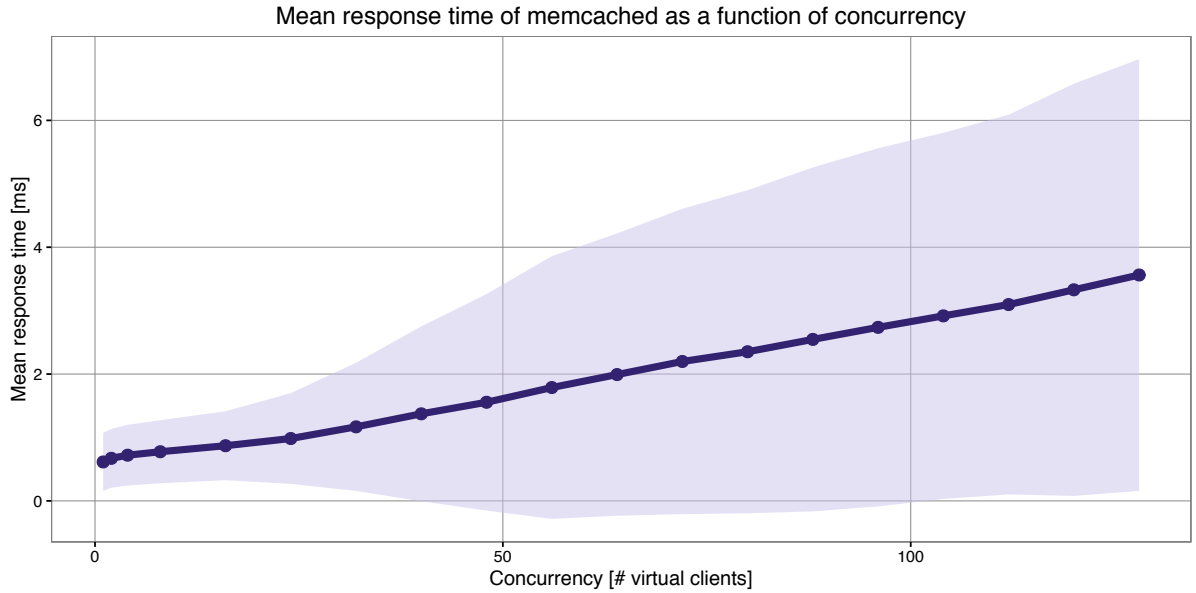

Figure 3: Response time of memcached without middleware, as measured by memaslap. Dark dots connected by a line show the mean response time and the light ribbon surronding the line shows ±1 standard deviation aggregated over all requests sent with that concurrency. Standard deviation values from different 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.

3.2 Response time

Figure 3.2 shows that the response time stays constant around 25ms for the whole duration of the trace experiment, with the standard deviation in roughly the same range. The load is high enough so that the queues are not empty and the response time depends on the amount of items in the queue, which itself can vary significantly – this causes variation in response time to be high similarly to the baseline experiment (coefficient of variation is roughly 1 here).

3.3 Overhead of middleware

In the trace experiment, three memcached servers need to service $3 \cdot 64 = 192$ clients, so the number of virtual clients per server is 64 (this assumes uniform load balancing, which was demonstrated in Section 1.2). For this reason, we should compare the performance of the system with the performance of the baseline at 64 virtual clients. Since the trace is run with full replication, i.e. all writes are done to three servers instead of one, these two numbers are not be directly comparable. However, since write requests make up only 1% of all requests, this difference is negligible.

The following table shows comparison of the baseline and actual (with middleware) performance of the system, shown with up to 3 significant digits. Using the middleware introduces a roughly 4-fold decrease in throughput (decreasing it by 24540 requests per second) and roughly 12-fold increase in response time (adding 23.6 ms), which is a significant deterioration in performance.

Metric	**Baseline**	**Middleware**	**Overhead**	**Slow-down**
throughput [requests/s]	32000	7460	-24500	4.3x
response time [ms]	2.2	25.8	23.6	11.7x

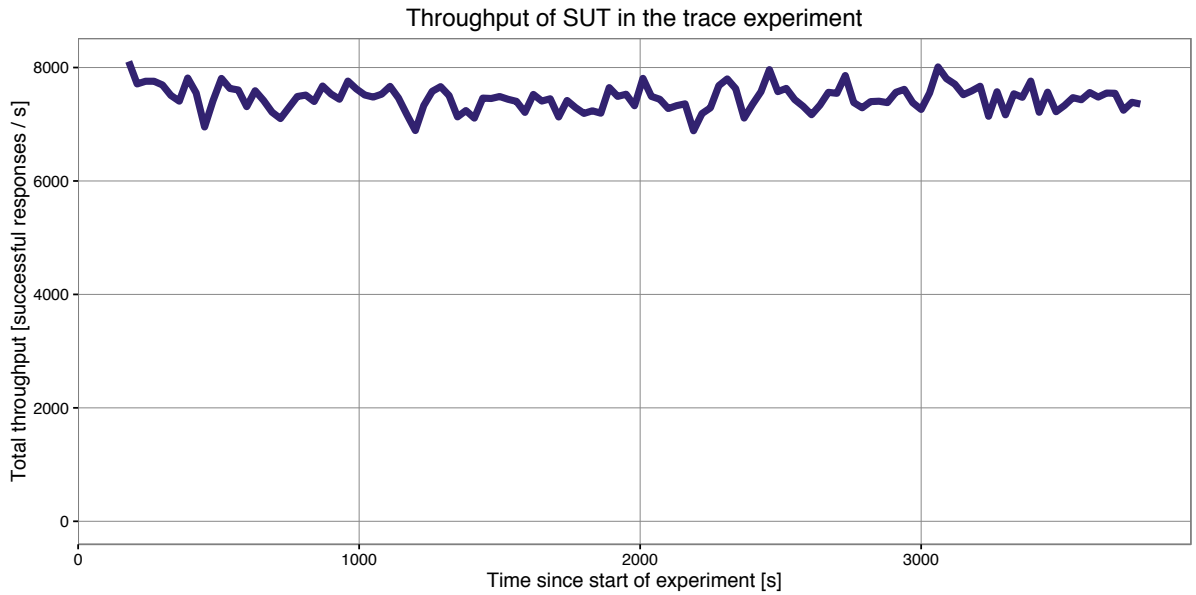

Figure 4: Throughput trace of the middleware. The dark line shows the total throughput, i.e. the sum over throughputs reported by the three clients.

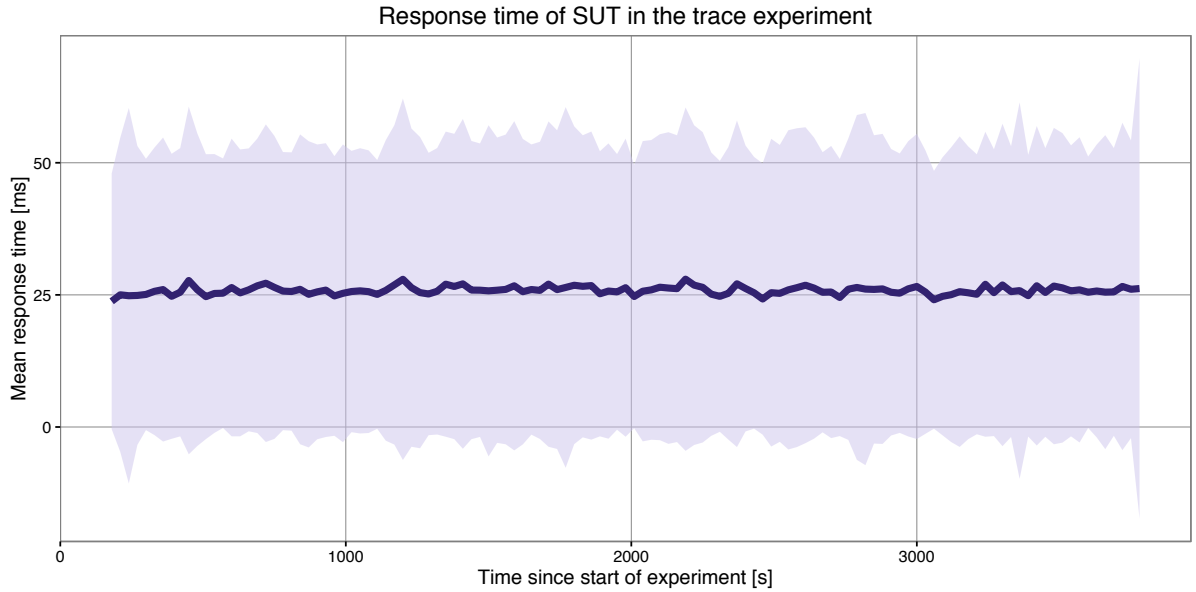

Figure 5: Response time trace of the middleware as measured by memaslap. The dark line shows the mean response time (calculated as the weighted average of mean response times reported by the three clients) and the light ribbon surrounding the line shows ± 1 standard deviation. Standard deviation values from different repetitions were aggregated in the same way as in Section 2.

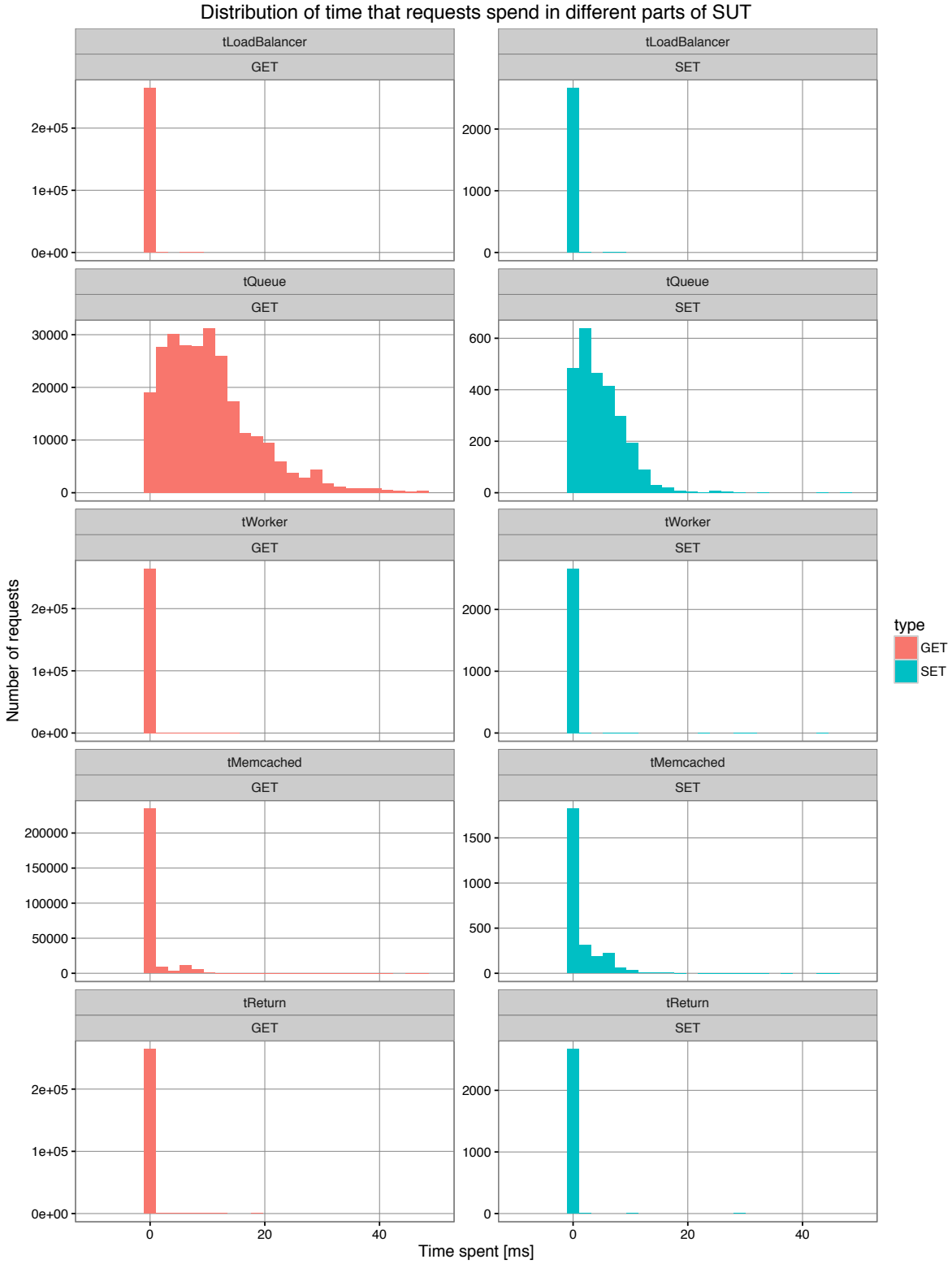

Figure 6: Distribution of time that requests spend in different parts of SUT during the trace experiment. Each plot shows the difference between two successive time stamps: $t_{LoadBalancer} = t_{enqueued} - t_{created}$, $t_{Queue} = t_{dequeued} - t_{enqueued}$, etc. The time (x) axis is cut off at 50ms. The graph shows only sampled requests (i.e. every 100th request) and excludes requests processed during the warm-up and cool-down periods.

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