Advanced Systems Lab (Fall'16) – First Milestone

Name: ***Fabio M. Banfi***
Legi number: ***09-917-972***

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 middleware has been implemented in Java 7, following the provided guideline as shown in Figure 1. The diagram also shows the structure of the program in terms of objects when the middleware is connected to three clients and three servers. The objects are boxed in blue, and the instance names are used as labels. Embracing a minimal design philosophy, the program consists of just five classes (not counting the provided interface `RunMW`), of which only three *core* classes contribute the actual logic of the middleware. Those are `Middleware`[1], `ServerWriter`[2], and `ServerReader`[3].

`Middleware` is responsible for starting the middleware itself and also for accepting, reading, and addressing clients' requests (by *load balancing*). This is all handled by the function `Middleware.run()`, which is documented in more detail in Section 1.2. As Figure 1 shows, `Middleware` has members `serverWriter`, an array of `ServerWriter` instances, and `serverReader`, an array of `ServerReader` instances.

`ServerWriter` handles all the `SET` requests generated by the clients, as well as `DELETE` requests (even though those are *not* generated by `memaslap`). On the other hand, `ServerReader` handles all the `GET` requests. Both classes possess an internal queue where the requests to be processed have to be written. More detailed description of those two classes are given in Section 1.3 and Section 1.4, respectively.

The other two classes are `Request`[4] and `Helper`[5]. `Helper` contains constant values (such as buffer sizes), some useful functions and structures used by the core classes, as well as some functions for testing and debugging. Finally, `Request` instances encapsulate client requests along with the associated client connections (as sockets), as well as timing information. The latter consist of six timestamps, T_{arr}, T_{left}, T_{enq}, T_{deq}, T_{sent}, and T_{recv}. The code has been instrumented to take those timestamps for each request as depicted in Figure 1. Note that the responses of the servers are given back to the clients by `ServerWriter` and `ServerReader` instances, contrary to what the diagram may suggest. More details on when exactly such timestamps are taken follow in the next sections.

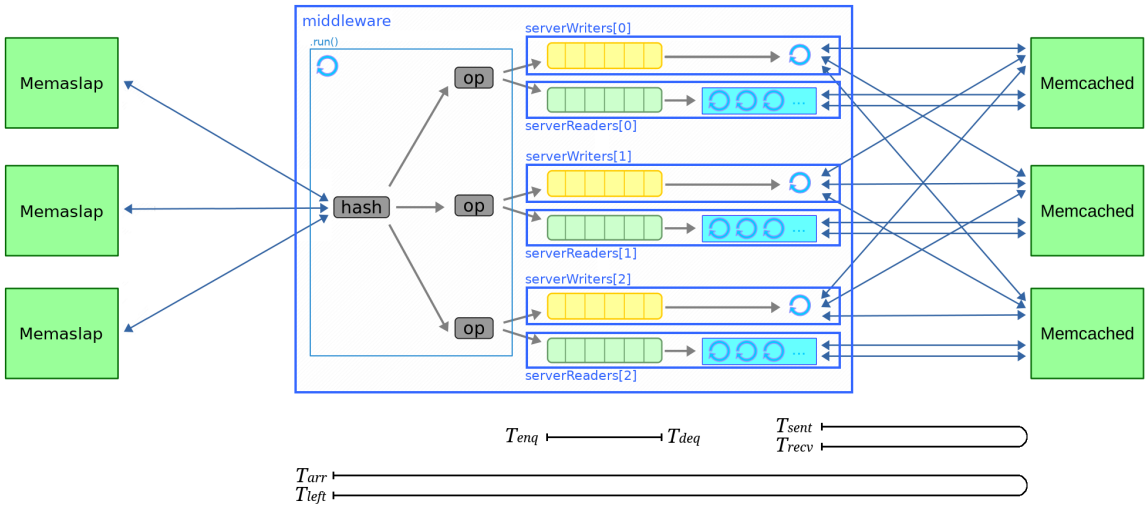

Figure 1: Middleware architecture diagram.

[1] https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/src/middleware/Middleware.java
[2] https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/src/middleware/ServerWriter.java
[3] https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/src/middleware/ServerReader.java
[4] https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/src/middleware/Request.java)
[5] https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/src/middleware/Helper.java

1.2 Load Balancing and Hashing

The function `Middleware.run()` *asynchronously* accepts connections from `memaslap` clients by means of `java.nio`'s selectors and channels. Each connected client's request is read at once into a buffer, and the byte-sequences corresponding to the operation (`SET`, `GET`, or `DELETE`) and the key (assumed to consist of exactly 16 bytes) are extracted. *Load balancing* is implemented by first hashing the key into a value from 0 to $N-1$ (where N is the number of servers), and then using the obtained integer as an index to select the appropriate `ServerWriter`/`ServerReader` instance (depending by the operation) to which forward the request.

The hashing of the keys is implemented by first converting the sequence of 16 bytes into a string, then calling `String.hashCode()` on the latter, and finally computing the reminder of the obtained number when divided by N. For strings, Java's `hashCode()` uses a product sum algorithm over the entire text[6]. Therefore, combining those steps and assuming that the key consists of the byte sequence $K = k_0, k_1, \ldots, k_{15}$, the obtained hash value is the integer[7]

$$h_N(K) \doteq \left[\sum_{i=0}^{15} k_i \cdot 31^{15-i} \bmod N \right] \in \{0, \ldots, N-1\}.$$

Since the middleware must act as a load balancer, it is important that it distributes the requests uniformly among the N servers. Assuming a source of uniform random sequences of 16 bytes (where each possible sequence of 16 bytes is picked with the same probability, or equivalently, each number in the interval $\{0, \ldots, 2^{128}-1\}$ is equiprobable), one would expect that the hash function h_N defined above, when attached to such a source, acts itself as a source of uniform random integers in the interval $\{0, \ldots, N-1\}$. More formally, we require $h_N(\mathbf{U}_{2^{128}}) = \mathbf{U}_N$, where $\mathbf{U}_n$ is defined as the uniform random variable over $\{0, \ldots, n-1\}$.

As a formal proof of this fact would be out of scope, an heuristic argument has been used to justify the choice of h_N. Let define the *statistical distance* between two probability distributions $\mathbf{P}$ and $\mathbf{Q}$ over a finite set $\mathcal{X}$ as the *total variation distance*, that is, the 1-norm of the difference of the two probability distributions, formally

$$\delta(\mathbf{P}, \mathbf{Q}) \doteq \frac{1}{2} \sum_{x \in \mathcal{X}} |\mathbf{P}(x) - \mathbf{Q}(x)| .$$

Thus, letting $\mathbf{H}_N \doteq h_N(\mathbf{U}_{2^{128}})$, we wish for $\delta(\mathbf{H}_N, \mathbf{U}_N)$ to be *as close as possible to zero*. For $N = 2, \ldots, 7$, the function `Helper.testHashFunction()` prints the statistical distance as defined above between the uniform distribution and an empirical version $\widetilde{\mathbf{H}}_N$ of the distribution $\mathbf{H}_N$. Concretely, for each value of N, 10'000'000 pseudo-random 16 bytes sequences were fed to h_N, and the output was used to fill the corresponding of N buckets (each one representing a server). The empirical distribution $\widetilde{\mathbf{H}}_N = h_N(\widetilde{\mathbf{U}}_{2^{128}})$ was then drawn from those buckets, where the pseudo-random source $\widetilde{\mathbf{U}}_{2^{128}}$ has been instantiated by `new BigInteger(128, new Random())`. The results of a run of this experiment are shown in Table 1, which clearly shows that $\widetilde{\mathbf{H}}_N$ is very close to the uniform distribution $\mathbf{U}_N$ for all used values of N, exactly as desired.

N	$\delta(\widetilde{\mathbf{H}}_N, \mathbf{U}_N)$	N	$\delta(\widetilde{\mathbf{H}}_N, \mathbf{U}_N)$
2	0.000236	5	0.000292
3	0.000321	6	0.000239
4	0.000262	7	0.000221

Table 1: Results of the statistical distance test on the chosen hash function.

[6] http://bugs.java.com/bugdatabase/view_bug.do?bug_id=4045622

[7] The notation $[k \bmod n]$ represents the remainder of k divided by n.

1.3 Write Operations and Replication

An instance of `ServerWriter` handles all the `SET` and `DELETE` requests associated with a specific *primary* server. Since it is possible to define a replication factor R when starting the middleware, each instance of the array `Middleware.serverWriters` opens R parallel connections with R different servers, where $R-1$ of them are *secondary*. Assuming that `serverWriters[`i`]` is associated with primary server $1 \leq i \leq N$, then it opens connections with the servers

$$i, [i+1 \bmod N], \ldots, [i+R-1 \bmod N].$$

`serverWriters[`i`]` has member `queue` of type `LinkedBlockingQueue<Request>`, which is *thread-safe*. The method `serverWriters[`i`].enqueue(Request)` allows to safely put a new request (whose key's hash is i) into this queue. The timestamp T_{enq} is taken just *before* this function call.

`serverWriters[`i`]` also defines a thread class `ServerWriter.WriterThread`, of which it starts a *single* instance. The method `ServerWriter.WriterThread.run()` is where all the write logic takes place. The communications with the replication servers happen asynchronously. At the beginning, the top request on the queue `queue`, if any, is safely removed. This is accomplished by calling the semi-blocking function `queue.poll(10, TimeUnit.MILLISECONDS)`. Note that the second parameter indicates a 10 ms timeout[8] after which, if no elements was removed from the queue, the function call stops to block. If a request to be processed was successfully removed from `queue`, that is, the call to `poll` returned something other than `null`, then the timestamp T_{deq} is taken and the request is forwarded to the R servers. This is done in a sequential fashion, that is, the request is first sent to server i, then to server $[i+1 \bmod N]$, and so on until it is finally sent to server $[i+R-1 \bmod N]$. The timestamp T_{sent} is taken right *after* sending the request to the primary server i.

After polling the queue for new requests, `run()` checks whether there are responses from the servers ready to be read. If this is the case, all the available responses are processed one after the other. Once the responses are all processed (and hence only when there were responses to process in the first place), `run()` checks which responses it can forward back to which client[9]. For this, the policy is that a response is sent back to the client which originated the corresponding request only once *all* the replications servers gave their response. Moreover, in case of a `SET` request, the response `STORED` is sent back to the client only if *all* the responses from the servers were `STORED`, and otherwise the response `ERROR` is sent to the client. The same applies to `DELETE` requests, where `STORED` is replaced by `DELETED`, and `ERROR` by `NOT_FOUND`. The timestamp T_{recv} is defined to be the *maximum* between the times at which each server gave its response, which for each server is taken right *after* receiving the answer. Finally, the timestamp T_{left} is taken right *after* the response has been sent to the clients. After logging the timing information of the request, the thread starts this process afresh by calling again `queue.poll(10, TimeUnit.MILLISECONDS)`.

A quick inspection of the three clients log files from `trace` reveals that write operations take about one and a half of the time needed for read operations (about 15 000 μs vs. 10 000 μs for total average response time).

Hypothesis 1. *The rate at which writes can be carried out in the system is likely to be limited by either the specified number of replication servers, the sequential sending to the replication servers, the saturation of* `memcached`*'s database, or a combination of those factors.*

[8] The timeout has been used to avoid that the middleware consumes 100% of the CPU time when there are no clients requests. It has been observed that the throughput is positively affected by this, since it is higher compared to when there is no timeout.

[9] This is accomplished by keeping a queue of sent requests, but for which a response has not yet been received, as well as a queue for each server which holds all its responses which have not yet been forwarded to the clients. Once *all* the response queues are non-empty, it means that a response is ready for the request at the top of the sent requests queue. Assuming that `memcached` keeps responses ordered in time, the response to this request corresponds exactly to all the top elements of the servers' queues.

1.4 Read Operations and Thread Pool

An instance of `ServerReader` handles all the `GET` requests associated with a specific server. Even though it is possible to define a replication factor R when starting the middleware, each instance of `Middleware.serverReaders` only accesses the value on the primary server, ignoring the content of the secondary servers (if any) for the specific key.

For any $1 \leq i \leq N$, `serverReaders[`i`]` starts a *thread pool* of size T, as specified when starting the middleware. The threads in the pool are all instances of `ServerReader.ReaderThread`, and each first opens its own separate connection with the primary server i. `serverReaders[`i`]` also defines a queue `queue` for incoming requests to be forwarded to the associated server. Since potentially more than one thread can read from the queue `serverReaders[`i`].queue`, the latter has to be *thread-safe*, and thus it has been instantiated as a `LinkedBlockingQueue<Request>`. The method `serverReaders[`i`].enqueue(Request)` allows to safely put a new request (whose key's hash is i) into this queue, and internally it just calls `queue.put(Response)`. The timestamp T_{enq} is taken right *before* this function call.

Being synchronous, the reading logic is very simple. It takes place inside the function `ServerReader.ReaderThread.run()`. First, the thread extracts an element from the queue by calling `queue.take()`, which blocks until an element was successfully removed from the queue (and thus in particular blocks the thread as long as the queue is empty). The timestamp T_{deq} is taken right *after* this function call. Subsequently, once an element has been successfully extracted, the thread sends it to the server and waits (blocking) until the server produces an answer. The timestamps T_{sent} and T_{recv} are taken right *after* sending the request to the server right *after* receiving an answer, respectively. Finally the thread forwards the response to the client that originated the request, and takes the timestamp T_{left} right *afterwards*. After logging the timing information of the request, the thread starts this process afresh by calling again `queue.take()`.

2 Memcached Baselines

The memcached baseline experiment aims at analyzing the behavior of `memcached` without the presence of the middleware. The experiment setup is outlined in Table 2.

Servers	1 × A2
Threads / machine	1
Clients	2 × A2
Virtual clients / machine	2 to 64, step 2
Keys size	16B
Values size	128B
Writes	1%
Overwrite proportion	90%
Middleware	Not present
Runtime × repetitions	30s × 5
Log files	`baseline`

Table 2: Memcached baselines experiment setup.

The experiment is carried out completely by the script located at https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/experiments/01_memcached_baselines, which collects *all* the `memaslap` outputs (`*.log`, inclusive repetitions), all the extracted data (`tps.data` for the throughput and `rt.data` for the response time), and also produces plots of the latter. Note that the throughput measured at each second by the two `memaslap` clients have been added up, while mean and standard deviation of the response times have been averaged. The throughput is depicted in Figure 2, and the response time in Figure 3.

2.1 Throughput

Hypothesis 2. *The throughput as a function of connected clients should initially grow quasi-linearly, an then reach a saturation point at which no more growth is observed. Eventually, when the number of clients becomes too large, the throughput may even start to decrease.*

Analysis. Figure 2 confirms in part this hypothesis. The quasi-linear growth is observed from 2 to about 16 clients. There seems to be a saddle point around 20 clients, but this is probably an artifact caused by the latencies of the connections. Then the throughput continues to grow, but at a much slower rate, which indicates that a saturation point has been hit, or is soon to be hit. Starting from 96 clients an even lower increase is observed, strengthening the possibility of having a saturation point at about 100. Nevertheless, the number of clients is too low so see an effective saturation, or even a degradation of the throughput.

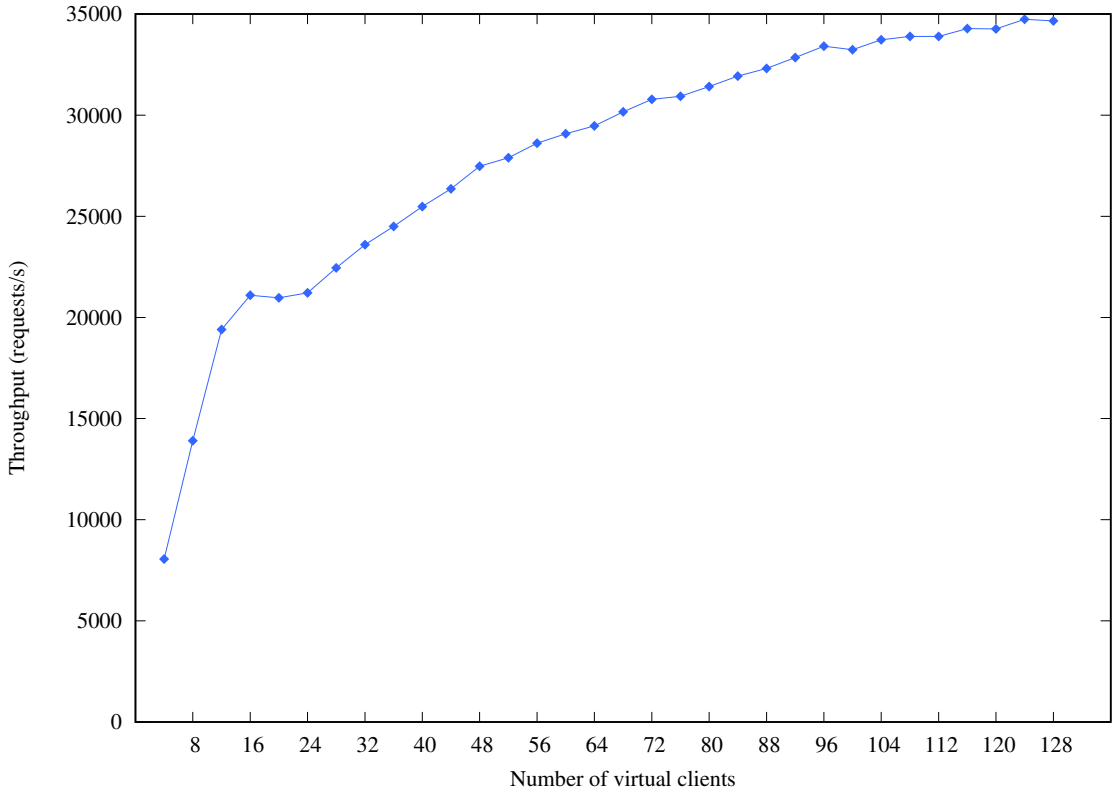

Figure 2: Throughput by baseline experiment.

2.2 Response time

Hypothesis 3. *The average response time in function of the number of connected clients should grow constantly for a short period of time, and then it should hit a saturation point where it should start to grow linearly. It may eventually hit a point where the grow rate increases even more.*

Analysis. From Figure 2, it results that the average response time is almost constant for up to 10 clients, but then clearly starts to grow linearly until 128. Nevertheless, a point where the grow rate increases even more is not observed. Figure 2 also shows the standard deviation of the response time. It results that this grows linearly as well.

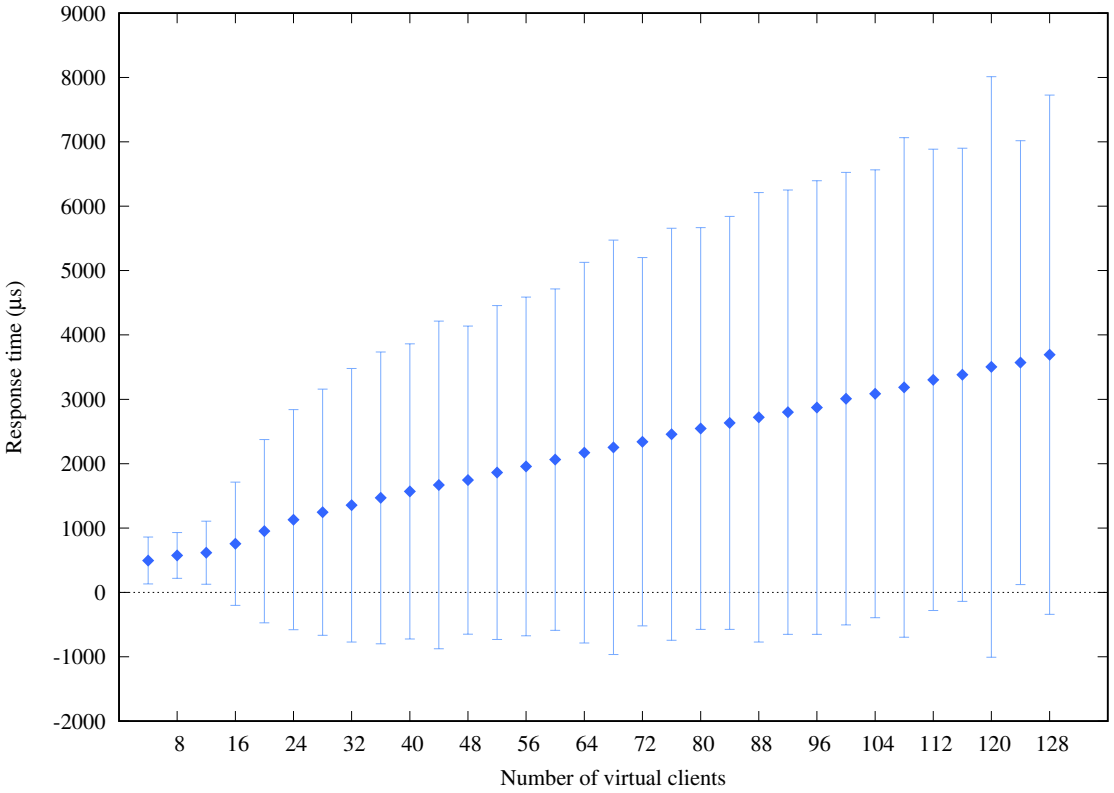

Figure 3: Average and standard deviation of response time by baseline experiment.

3 Stability Trace

The stability trace experiment aims at analyzing the behavior of the middleware when run under stress for a long time. The experiment setup is outlined in Table 3.

Servers	3 × A2
Threads / server machine	1
Clients	3 × A2
Virtual clients / machine	64
Keys size	16B
Values size	128B
Writes	1%
Overwrite proportion	90%
Statistics	1s
Middleware	1 × A4
Pool threads	16
Replication	Full
Runtime × repetitions	1h × 1
Log files	`trace`

Table 3: Stability trace experiment setup.

The experiment is carried out completely by the script located at https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/experiments/02_stability_trace, which collects *all* the `memaslap` outputs (`*.log`), all the extracted data (`tps.data` for the throughput and `rt.data` for the response time), and also produces plots of the latter. Note

that the throughput measured at each second by the three `memaslap` clients have been added up, while mean and standard deviation of the response times have been averaged. The throughput is depicted in Figure 4, and the response time in Figure 5.

3.1 Throughput

Hypothesis 4. *The throughput should stay constant over time.*

Analysis. Figure 4 shows that the throughput is oscillating, but stable over time, as expected. The oscillations appear to be contained, and no substantial peaks are observed. They are probably a consequence of the architecture of the middleware, to be investigated later. Therefore the middleware is functional and it can handle a long-running workload without crashing or degrading in performance, as hoped. From the plot it is obvious that apart from oscillations, the throughput has some noise. This is probably not correlated with the architecture, but due to the unpredictable latencies introduced by the network connections. Finally, the average throughput is 20 316 requests/s.

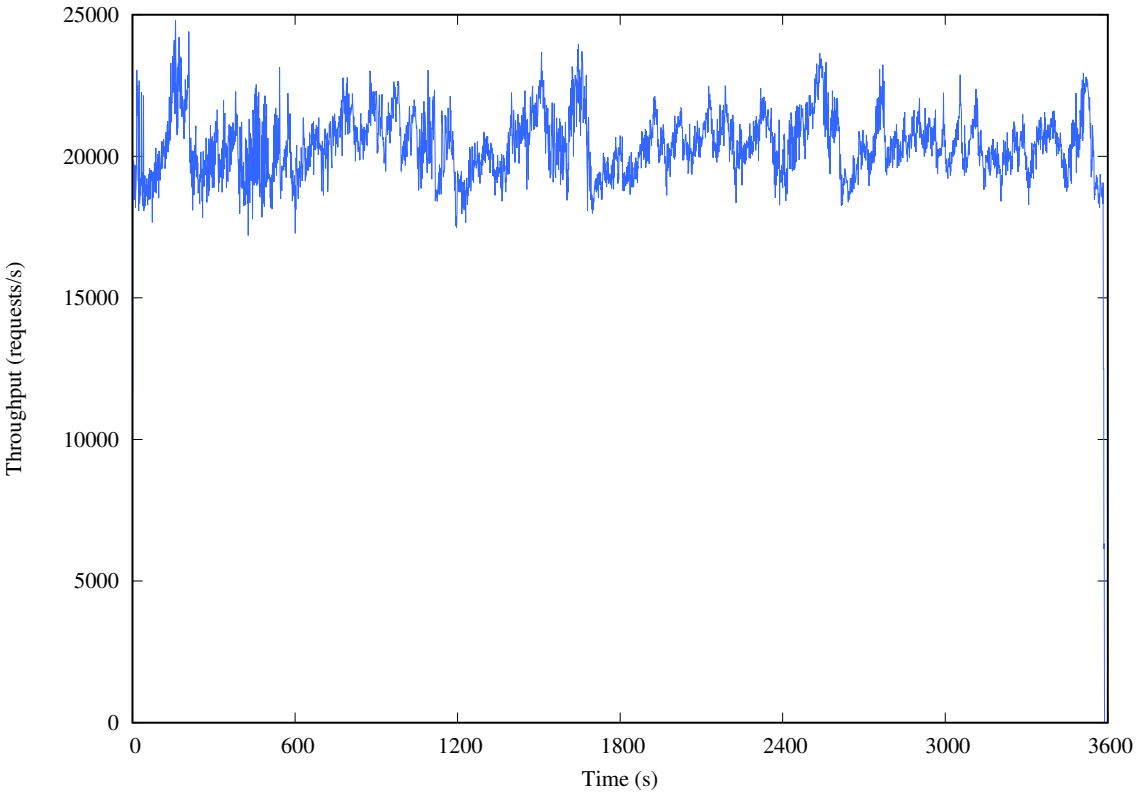

Figure 4: Throughput by stability trace experiment.

3.2 Response time

Hypothesis 5. *The average response time should stay constant over time.*

Analysis. Figure 5 shows that the average response time is also oscillating, but stable over time, as expected. Again, the oscillations appear to be contained, and no substantial peaks are observed. The standard deviation of the response time also results to be constant and contained, with except for a couple of peaks, which are probably due to the unpredictable latencies introduced by the network connections. Finally, the average of the average response time is 9 449.79 μs, while the average of its standard deviation is 4 732.55 μs.

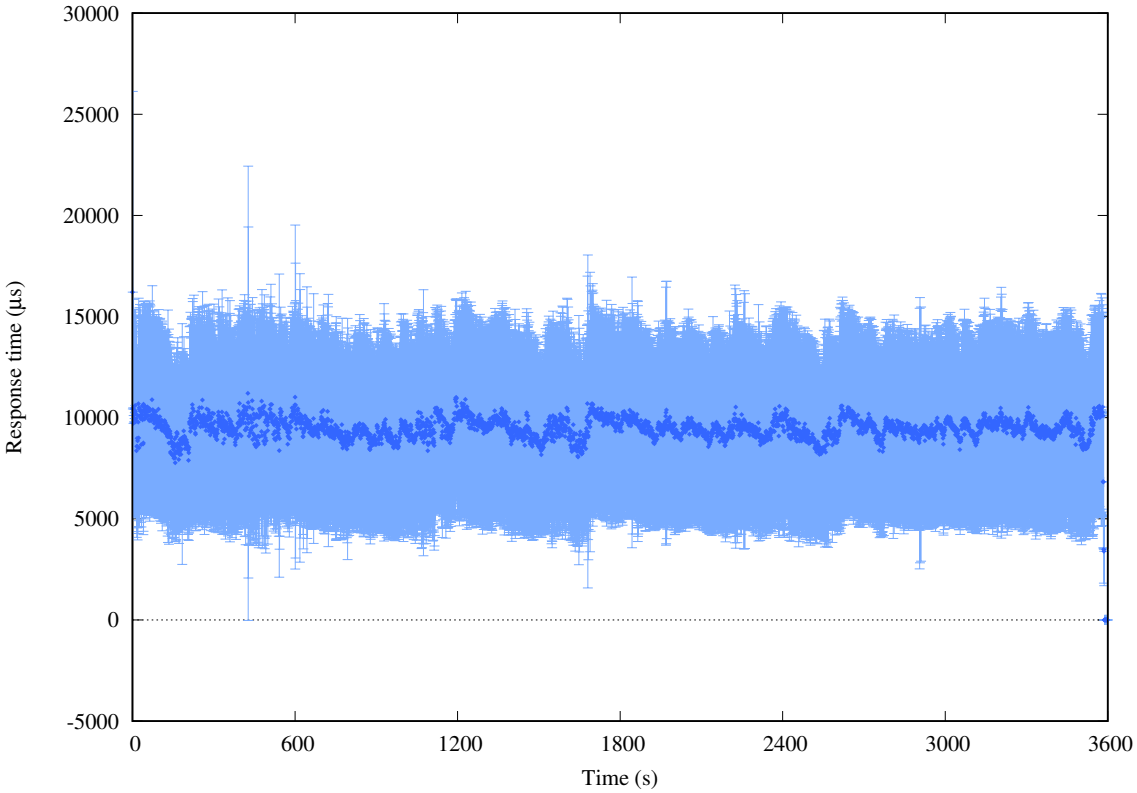

Figure 5: Average and standard deviation by response time of stability trace experiment.

3.3 Overhead of middleware

Hypothesis 6. *The middleware introduces some overhead.*

Analysis. Intuitively, the fact the the communication channel between the client and the server has been cut in the middle, and the fact that requests are accessed internally (when reading the operation and the key), suggests that the presence of the middleware introduces some overhead. This is confirmed by the above plots and by the results of the following analysis, reported in Table 4. Assuming that the middleware divides the load uniformly across the servers and that each `memaslap` client in the stability trace experiment produces the same amount of load, the experiment can be seen as three parallel instances of the middleware connected to one client and one server. Note that write operations may take longer than reads because of full replication, but since they account for just 1% of the load, their effect can be ignored. Since each client in the stability trace experiment instantiates 64 virtual clients, throughput and average response time have been compared with those found in the baseline experiment by 64 clients. Those are extracted directly from `baseline` as $\tau \doteq 29\,469.8$ requests/s and $\rho \doteq 2\,170.9$ μs, respectively. For the trace, since throughput is additive, the average throughput found in Section 3.1 has been divided by three, that is, $\tau' \doteq 20\,316/3 = 6772$ requests/s, and for the average response time the average value found in Section 3.2, namely $\rho' \doteq 9\,449.79$ μs, has been taken. The approximate overhead of the middleware is computed as *ratio* and as *increase* percentages. For the ratio, τ'/τ and ρ/ρ' were used, respectively. For the increase, $\tau'/\tau - 1$ and $\rho'/\rho - 1$ were used, respectively. The resulting overheads are collected in Table 4. As expected, the overhead ratio is consistent for throughput and average response time, and it is about 22%. Also as expected, the throughput with the middleware suffers a significant loss, while the average response time significantly increases.

	Overhead	
	Ratio	Increase
Throughput	22.98%	−77.02%
Response time	22.97%	335.29%

Table 4: Estimation of the overhead introduced by the middleware by 64 virtual clients.

Logfile listing

For each experiment, the logfiles (`*.log`) and the data files (`*.data`) are collected inside a single ZIP file, located at

https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/tree/master/log/experiments/

Table 5 indicates the names of the ZIP files of the two experiments. For both experiments, the data files `tps.data` and `rt.data` are reproducible from the logfiles using the scripts

https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/reproduce_data/01_memcached_baselines_rd

https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/reproduce_data/02_stability_trace_rd

(PDF plots of the data will be generated as well). Note that these scripts must be called from the root directory of the project and take as argument just the name of the project, without the prepended path, i.e.,

```
./scripts/reproduce_data/01_memcached_baselines_rd
    01_memcached_baselines_16-10-21_08:20:35
```

and

```
./scripts/reproduce_data/02_stability_trace_rd
    02_stability_trace_16-10-20_20:38:14.
```

Finally, note that for completeness `trace` also contains the logfile `mw_16-10-20_18:40:14.log` of the middleware, but this has been inserted manually into the ZIP file.

Short name	Location
baseline	01_memcached_baselines_16-10-21_08:20:35.zip
trace	02_stability_trace_16-10-20_20:38:14.zip

Table 5: Location of referenced data files.