Advanced Systems Lab (Fall'16) – Second Milestone

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

Grading

Section	**Points**
1	
2	
3	
Total	

Contents

Modifications to the middleware

In the last milestone submission, my middleware implemented all functionality as necessary. However, the resource usage was extremely wasteful: each read thread took up nearly 100% of the resources allocated to them and never went to a sleeping state. This caused more than 10-fold drops in performance when going from $T = 1$ to $T = 4$ (for $S = 5$), and would have made the maximum throughput experiment useless. The changes can be seen on GitLab.

To verify that the system is still stable, I re-ran the trace experiment. The throughput and response time are shown in Figures 1 and 2, and are confirmed to be stable (and throughput is roughly 30% higher). The Interactive Response Time Law also still holds (to within 0.46%). For explanations of the figures, see Milestone 1 report.

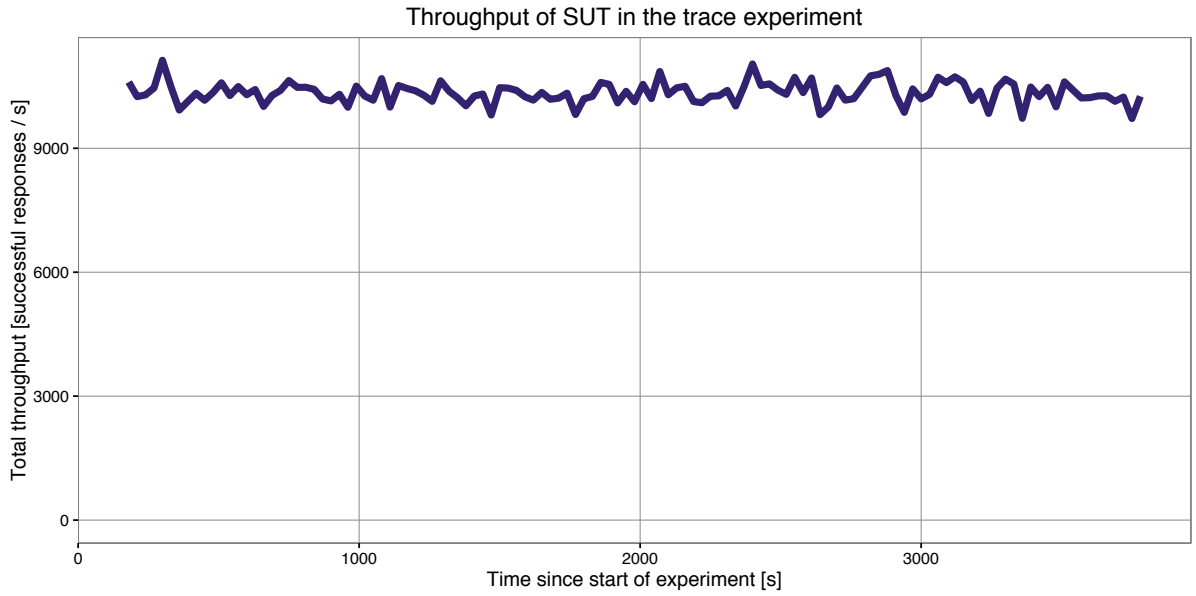

Figure 1: Throughput trace of the middleware.

Figure 2: Response time trace of the middleware as measured by memaslap.

Definitions and setup

In all experiments, the following definitions hold.

- The *system under test* (SUT) is the middleware together with the connected memcached servers, running on Ubuntu virtual machines in the Azure cloud.
- *Throughput* is the number of requests the SUT successfully responds to, per unit of time, as measured by memaslap.
- *Response time (memaslap)* is the time from sending to receiving the request to the SUT including any network latencies, as measured by the client (memaslap).
- *Response time (middleware)* is the time from receiving the request in the middleware ($t_{created}$) to returning it to the client ($t_{returned}$), as measured by the middleware. This is the measurement used in most graphs here; the reasoning behind this is shown in Appendix A.

In all experiments, the following holds about the experimental setup:

- The middleware was run on Basic A4 instances, and both memaslap and memcached were run on Basic A2 instances.
- The first 2 and last 2 minutes of each experiment were discarded from analyses as warm-up and cool-down time.
- The request sampling rate for logging is set to $\frac{1}{100}$ in throughput experiments (Section 1) and $\frac{1}{10}$ in replication and write proportion experiments (Sections 2 and 3).
- Response times inside the middleware were measured with a 1 millisecond accuracy.
- The system is closed because memaslap clients wait for a response before sending a new request.

1 Maximum Throughput

1.1 Experimental question

In this section, I will run experiments to find out a) the maximum throughput of the SUT, b) the number of read threads (T) in the middleware that achieves this c) the number of virtual clients (C) that achieves this.

To this end, I will measure throughput as a function of T and C, in 10-second time windows. I will find the maximum sustained throughput of the SUT, i.e. the throughput at which the response time does not increase rapidly with additional clients. For each parameter combination, I will run experiments until the 95% confidence interval (calculated using a two-sided t-test) lies within 5% of the mean throughput.

1.2 Hypothesis

I approximate that the maximum throughput will be 17200 requests per second using 50 read threads in the middleware at a load of 550 clients. The maximum sustained throughput will occur in a range of 200 clients.

1.2.1 Number of threads

Given that requests spend most of their time ($\sim$ 90% in the trace experiment) waiting in the queue, increasing T will increase throughput. If we reduce the queueing time by a factor of 10, it will no longer be the bottleneck (then waiting for memcached's response – which takes $\sim$ 9% of response time in the trace experiment – becomes the bottleneck). Assuming the time spent in the queue scales linearly with the number of read threads, we should increase T 10-fold, i.e. $T = 50$ maximises throughput.

1.2.2 Number of clients

Throughput is maximised at roughly 110 virtual clients per memcached server, so 550 virtual clients in total. This is based on the fact that in the Milestone 1 baseline experiment, the throughput of a single memcached server without middleware saturated at around 110 virtual clients. However, the knee of the graph was at 40 to 50 clients per server, so we can expect the knee to occur at around 200 clients in our setup. The maximum sustained throughput will be in that region because after the knee, additional clients don't increase throughput much but significantly increase response time.

1.2.3 Throughput

In the trace experiment the throughput was roughly 10300 requests per second so we have a lower bound for the expected throughput. Naively assuming that the throughput of GET requests scales linearly with the number of servers S would yield an expected throughput of $\frac{5}{3} \cdot 10300 = 17200$ requests per second. However, this does not take into account that we will also increase the number of threads (from $T = 5$ in the trace experiment). Thus I expect the maximum sustained throughput to be definitely more than 10300 requests per second, and likely to be more than 17200 requests per second.

I predict that the graph of throughput as a function of the number of clients will look like in Figure 3: rapidly increasing at first, then reaching the knee after which throughput growth is much slower, and then completely saturating. After saturation, the throughput may fall due to unexpected behaviour in the middleware.

Breakdown of response time I expect that the most expensive operations inside the middleware will be queueing ($t_{dequeued} - t_{enqueued}$) and waiting for a response from memcached ($t_{forwarded} - t_{received}$). Queueing takes time because for a C that gives a high throughput, the queue will also be non-empty and requests will need to wait. Requesting a response from memcached takes time because of a) the time it takes for memcached to process the request and b) the round-trip network latency.

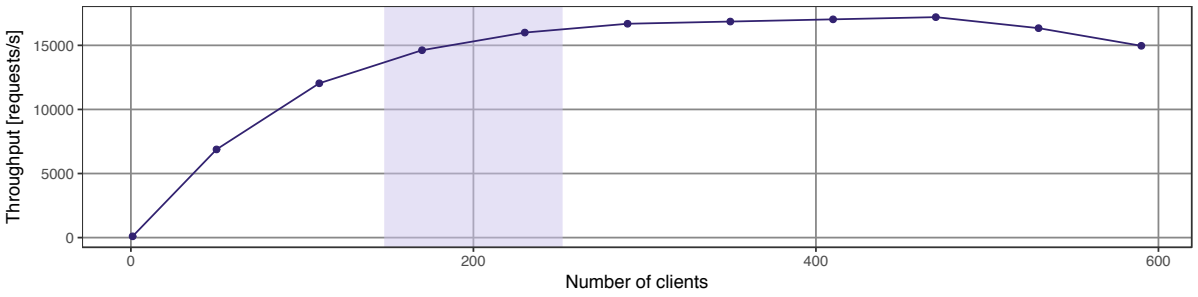

Figure 3: Expected graph of throughput as a function of number of clients (for the optimal value of T). The shaded area shows the range where maximum sustained throughput (and knee of the graph) will occur.

1.3 Experiments

Number of servers	5
Number of client machines	$\in \{1, 3\}$
Virtual clients	$\in \{1, 36, 72, 144, 180, 216, 288, 360, 432, 504, 576, 648\}$
Workload	Key 16B, Value 128B, Writes 0%
Middleware: replication factor	1
Middleware: read threads	$\in \{1, 16, 32, 64\}$
Runtime x repetitions	at least 6min x 1; more in some cases
Log files	throughput-C*-T*-r*

Three client machines were used for all experiments, except for the 1-client experiment, where only one machine was used.

The values of T to test were $T = 1$ as the lowest possible value, and then from $T = 16$ in multiplicative steps of 2. The reason for the small number of tested values of T is pragmatic: it doesn't require hundreds of experiments and at the same time gives a reasonable approximation of the optimal T.

Some parameter combinations did not yield the required confidence interval in the first 6-minute repetition of the experiment. When that was the case, I re-ran the experiment (in some cases for a longer time), thus producing more datapoints and decreasing the confidence interval.

1.4 Results

TODO: restructure with subsubsections

Maximum sustained throughput Figure 6 shows that the highest throughput was achieved using $T = 64$ at 720 clients (20100 requests/s), followed by $T = 64$ at 648 clients (20000 requests/s) and $T = 32$ at 432 clients (19600 requests/s). This is reasonably close to the expected value of $C = 550$. However, since we are trying to maximise sustained throughput, we also need to look at response times.

Figure 7 shows the percentiles of the response time distribution for each parameter set. It is apparent that for all values of $T > 1$, both the median response time (green line) and 95%

quantile (blue line) increase significantly after 216 clients. For this reason, we will exclude all values of $T > 216$ from consideration as unsustainable.

Of the remaining setups, the highest throughput is achieved both by 180 and 216 clients at $T = 32$. Thus we pick the one with the lower number of clients – **180 clients and 32 threads** – as the configuration we will declare optimal at a throughput of 18400 requests per second. Both C and throughput are close to the expected values; T is lower but not by an order of magnitude.

TODO: Explain why the system reaches its maximum throughput at these points and show how the performance changes around these configurations.

Breakdown of response time The distribution of time spent in different parts of the middleware is shown in Figure 4, and the means in Figure 5. As expected, the most expensive operations are queueing and waiting for a response from memcached. The distributions of *tQueue* and *tMemcached* are bimodal with a second peak at roughly 8ms because TODO: why?.

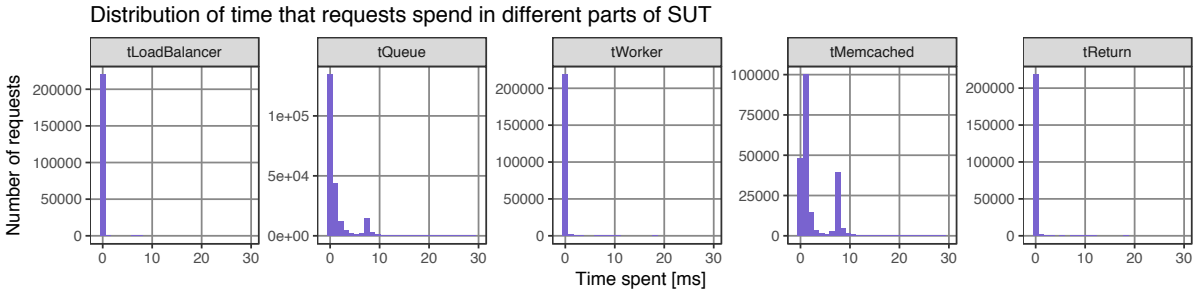

Figure 4: The distribution of times that GET requests spend in different parts of SUT. Note the time axis only shows values up to 30ms (this range includes almost all datapoints).

Name	Begin timestamp	End timestamp	Mean [ms]
tLoadBalancer	$t_{created}$	$t_{enqueued}$	0.0043
tQueue	$t_{enqueued}$	$t_{dequeued}$	1.40
tWorker	$t_{dequeued}$	$t_{forwarded}$	0.0191
tMemcached	$t_{forwarded}$	$t_{received}$	2.85
tReturn	$t_{received}$	$t_{returned}$	0.0221

Figure 5: The amount of time spent on different operations inside the middleware for the optimal run ($C = 180$ and $T = 32$).

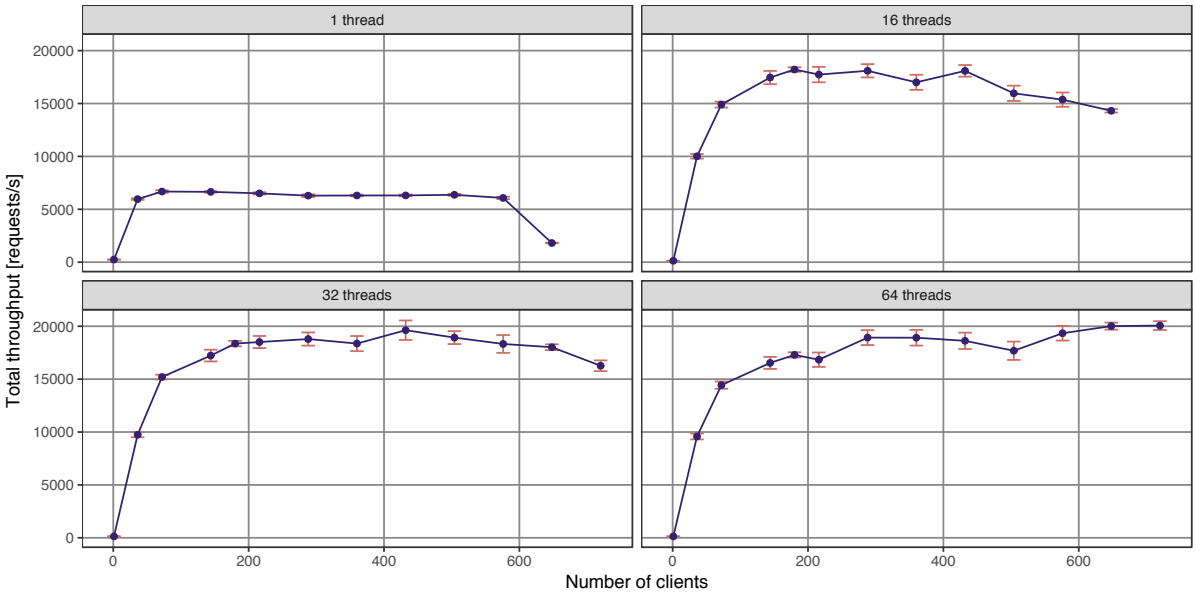

Figure 6: Throughput as a function of C for different values of T. Errorbars show the 95% confidence interval around the mean value which is shown with points connected by lines.

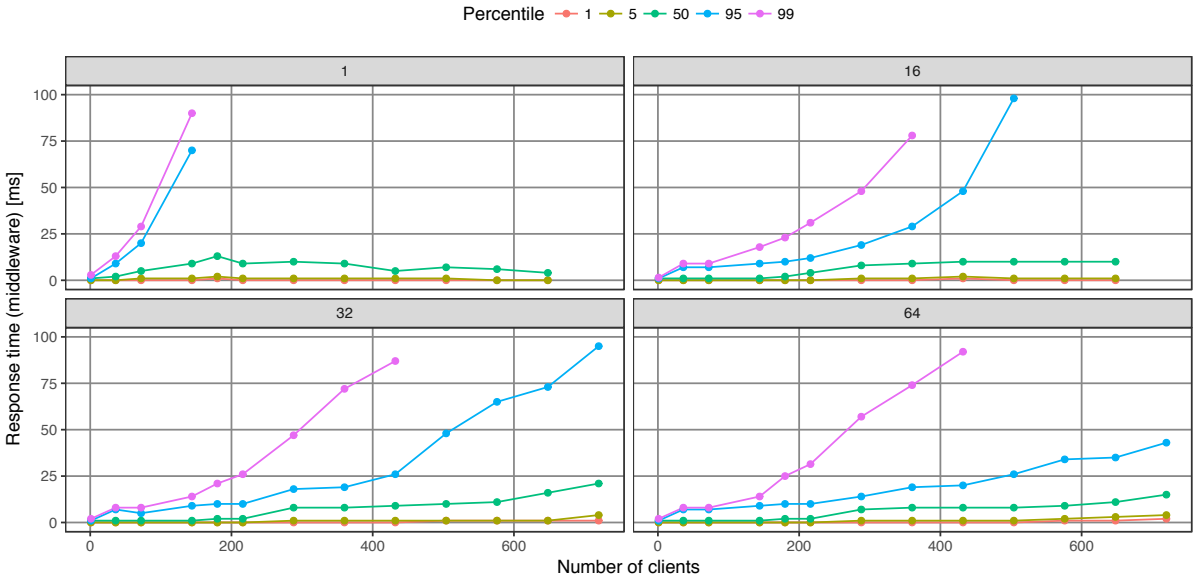

Figure 7: The 1%, 5%, 50%, 95% and 99% percentiles of the response time (middleware) distribution, as a function of C for different values of T. Values above 100ms are not shown.

2 Effect of Replication

2.1 Experimental question

In this section, I will run experiments to find out how the response time of SUT depends on the number of servers S and replication factor R. Additionally, I will investigate whether GETs and SETs are differently affected by these parameters. Finally, I will find out which operations become more time-consuming as these parameters change.

To this end, I will measure response time (middleware) for every 10th request as a function of S and R, and measure how long requests spend in each part of the SUT (based on the timestamps defined in Milestone 1). For each parameter combination, I will run experiments until the 95% confidence interval (calculated using a two-sided t-test) lies within 5% of the mean response time, but not less than 3 repetitions.

2.2 Hypothesis

I predict the following.

2.2.1 Get and set requests

Get and set requests will not be impacted the same way by different setups.

Get requests will be processed faster as we increase S because the same load will be distributed across more threads. Increasing R will have no effect on get requests because replication is only done for set requests (there may be secondary effects due to e.g. write threads requiring more CPU time, but this should be negligible).

Set requests will be strongly affected by R. If $R = 1$, set requests will be processed faster for higher S because each request is only written to one server, and for a higher S the same load is distirbuted across more write threads. However, if $R > 1$, response time of sets increases due to two factors: a) the request is written serially to R servers, and b) not all R responses are received at the same time. Assuming a) is negligible compared to b), we will observe an increase in the mean response time.

All of this is summarised in Figure 8. For get requests, response time will be independent of R for any fixed S. For set requests, response time increases linearly with increasing R, and the slope increases with S. I also predict the total throughput will decrease as R increases because the servers will need to do additional work (communicating more with memcached servers).

2.2.2 Relative cost of operations

As explained previously, more replication means that the middleware needs to send each set request to more servers and wait for more answers. Thus, as R increases, tMemcached will increase. Since each set request takes longer to process, this means that tQueue will increase as well. I also predict that the relative cost of get operations will not change.

2.2.3 Scalability

In an ideal system, a) there would be enough resources to concurrently run all threads; b) all memcached servers would take an equal and constant amount of time to respond; c) there would be no network latencies; d) dequeueing would take constant time.

For get requests, the ideal system would have linear speed-up (assuming the load balancer does not become a bottleneck). I predict that the SUT will have sublinear speed-up because the response time also includes network latency – a term that is not dependent on S: $response\ time = const. + \frac{const.}{S}$. In addition, since threads compete for resources in the SUT, the speed-up will be even lower than what's predicted by the formula above.

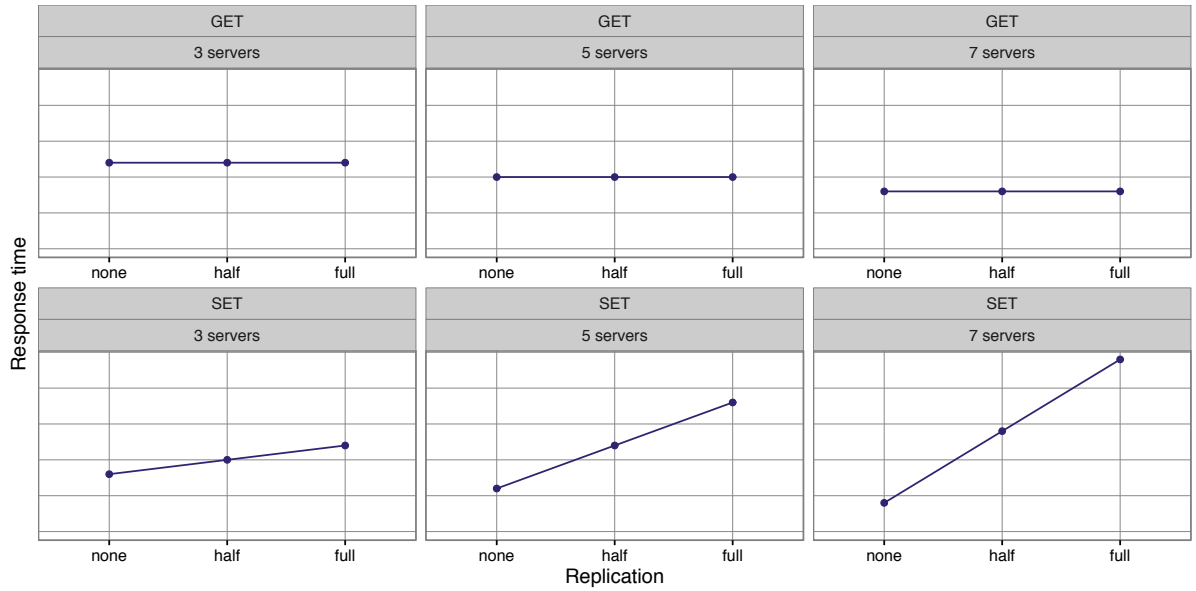

Figure 8: Expected response times of SUT. The vertical (time) axis has an arbitrary but fixed scale for all plots in the top row, and a different but also fixed scale for the bottom row.

2.3 Experiments

Number of servers	$\in \{3, 5, 7\}$
Number of client machines	3
Virtual clients	180
Workload	Key 16B, Value 128B, Writes 5%
Middleware: replication factor	$\in \{1, ceil(S/2), S\}$
Middleware: read threads	32
Runtime x repetitions	6min x 3
Log files	replication-S*-R*-r*

2.4 Results

2.4.1 Get requests

From Figure 9 we can see that increasing R from 1 to S does have an impact on the mean response time of get requests (contrary to the hypothesis) and this effect is amplified as S grows. However, the 25%, 50%, and 75% percentiles stay constant, implying that most of the requests aren't affected (in accordance with the hypothesis) – only the response time of outliers (gets with high response times) increases. Figure 11 shows that queue time is constant and the increase in response time comes almost entirely from waiting for memcached's response; this means the increase is caused by either increased network latency (due to more traffic at a higher value of R) or increased memcached response time.

I predicted that increasing S while keeping R constant would decrease the response time of get requests. In fact I was only partly right: the 25%, 50%, and 75% percentiles stay constant, but the mean decreases with S at $R = 1$ and increases at $R > 1$. Investigating the breakdown of time spent inside the middleware (Figure 11) gives an answer: queueing time does decrease with S for all replication levels, but this gain is offset by the increase in time spent waiting for memcached's response.

Given that $tMemcached$ increased with S even when R was constant, we can conclude that the performance degradation was mostly due to networking – if it had been caused by memcached's slower responses, $tMemcached$ would not have changed with S.

2.4.2 Set requests

Figure 9 shows that increasing R does increase response time for $S = 7$ but the trend is not so clear for $S = 3$ and $S = 5$. If we also consider the breakdown of response time by component (Figure 11) we can explain TODO:

Adding servers at $R = 1$ decreases response time to set requests – this is in line with the hypothesis. For $R > 1$ adding servers does not have a linear effect on response time, which corresponds to the prediction that two factors (serial writes and differing memcached response times) affect the response time, each in a different direction.

TODO:

TODO: Also mention throughput

2.4.3 Relative cost of operations

TODO:

2.4.4 Scalability

TODO:

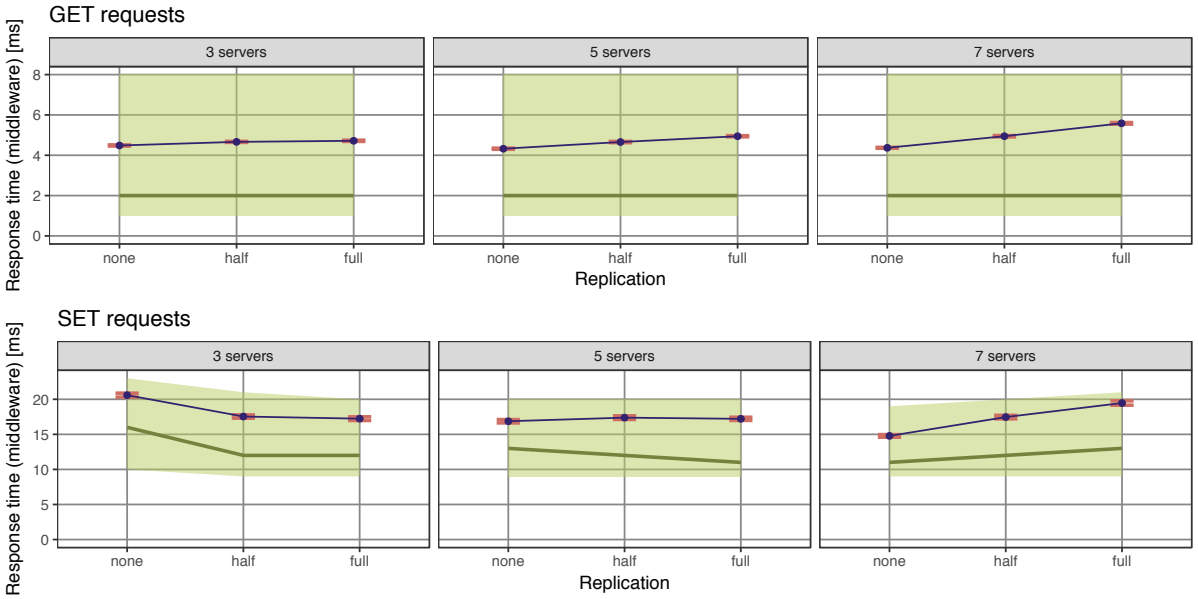

Figure 9: Response time (middleware) to GET and SET requests as a function of R, for different values of S. The line and points show the mean response time; red errorbars show the 95% confidence interval in a double-tailed t-test; the green area shows the 25% (bottom edge) and 75% (top edge) quantiles of response times.

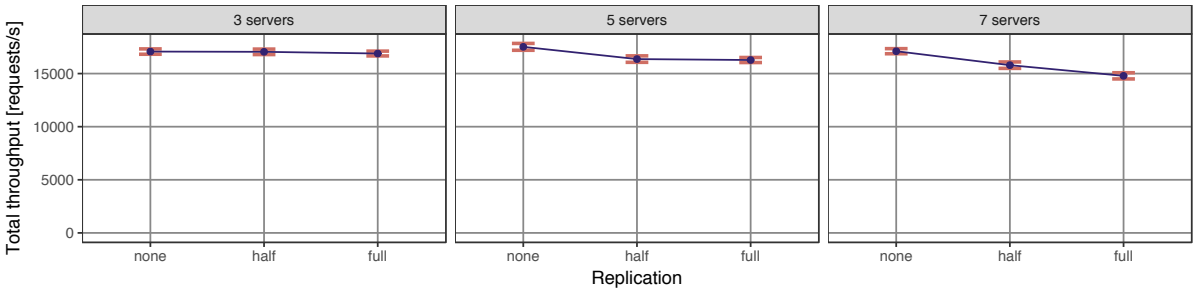

Figure 10: Throughput of SUT as a function of R, for different values of S. The line and points show the mean throughput; red errorbars show the 95% confidence interval over 10-second samples in a double-tailed t-test.

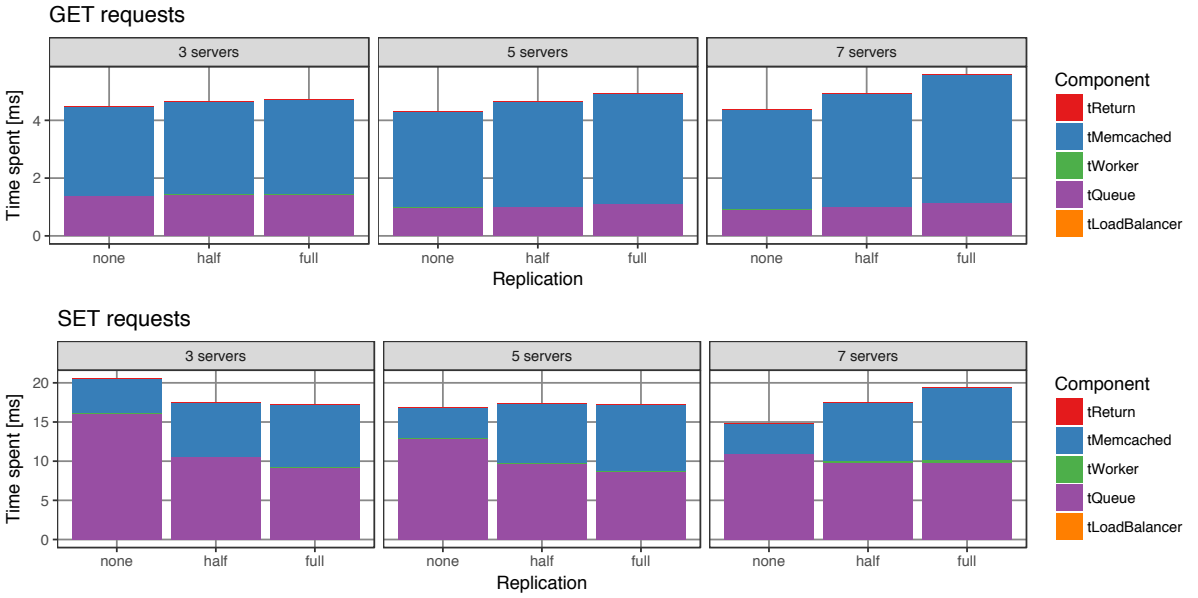

Figure 11: Absolute cost of operations inside SUT as a function of R, for different values of S. Each column is divided into sections by the *average* time spent in the respective component of SUT.

3 Effect of Writes

3.1 Experimental question

In this section, I will run experiments to find out how the response time and throughput of the SUT depend on the proportion of write requests, W. I will investigate this relationship for different values of S and $R \in 1, S$. Finally, I will find out the main reason for the reduced performance.

To this end, I will measure throughput (in 10-second time windows) and response time (for every 10th request) as a function of W, S and R, and measure how long requests spend in each part of the SUT (based on the timestamps defined in Milestone 1). For each parameter combination, I will run experiments until the 95% confidence interval (calculated using a two-sided t-test) lies within 5% of the mean throughput, but not less than 3 repetitions.

3.2 Hypothesis

I predict the following.

3.2.1 Performance impact

Increasing W will decrease throughput and increase mean response time for any combination of S and R because write requests take longer to process. Fully replicated setups ($R = S$) will suffer a larger performance decrease than setups with no replication ($R = 1$) because in the case of full replication, WriteWorkers do more work for each write request (i.e. the response time of each write request will be higher).

The setups with $S = 3$ servers will suffer the largest relative performance decrease (compared to $S > 3$) because there are fewer WriteWorkers dealing with the same load of write requests, which in turn increases the queue wait time $tQueue$.

3.3 Experiments

Number of servers	$\in \{3, 5, 7\}$
Number of client machines	3
Virtual clients	180
Workload	Key 16B, Value 128B, Writes $\in \{1\%, 4\%, 7\%, 10\%\}$
Middleware: replication factor	$\in \{1, S\}$
Middleware: read threads	32
Runtime x repetitions	8min x 3
Log files	writes-S*-R*-W*-r*

One parameter combination ($S = 3, R = 3, W = 1\%$) did not yield the required confidence interval in the first repetition of the experiment so I re-ran the experiment thus producing more datapoints and decreasing the confidence interval.

3.4 Results

Reporting experiment results. Comparison of hypothesis and experiment results.

3.4.1 Impact on get requests

Figure 12 shows the effect of W on get requests. While the mean response time does vary slightly (up to 1ms), the 25% and 75% quantiles are constant for all setups but one ($S = 7, W = 7\%$, full replication) which is still very close. Thus the performance of get requests does not depend much on W. TODO: comparison with hypothesis?

3.4.2 Impact on set requests

Figure 13 shows the effect of W on set requests. It is clear that increasing W also increases response time; the effect seems mostly linear. An outlier is $S = 3, R = 1$ where the effect is much stronger. This can be attributed to the server being overloaded, similarly to Section TODO: ref to exp2 discussion.

3.4.3 Relative impact

Figure 14 shows the impact of W on response times, compared to base cases. TODO:
TODO:

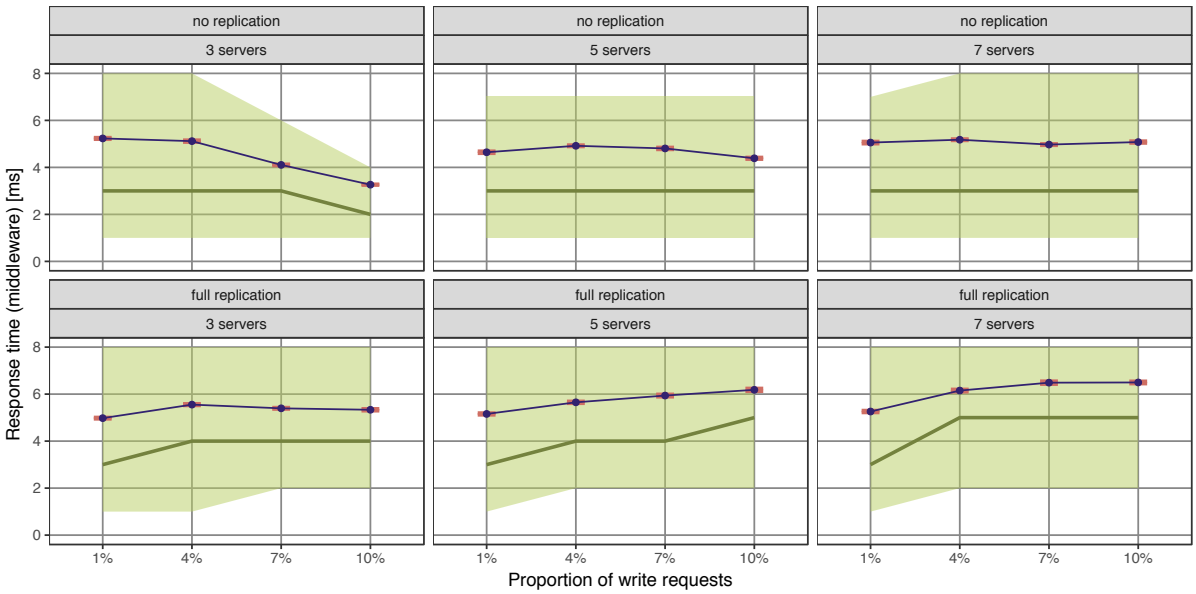

Figure 12: Response time (middleware) to **GET** requests as a function of W, for different values of S and R. The line and points show the mean response time; red errorbars show the 95% confidence interval in a double-tailed t-test; the green area shows the 25% (bottom edge) and 75% (top edge) quantiles of response times.

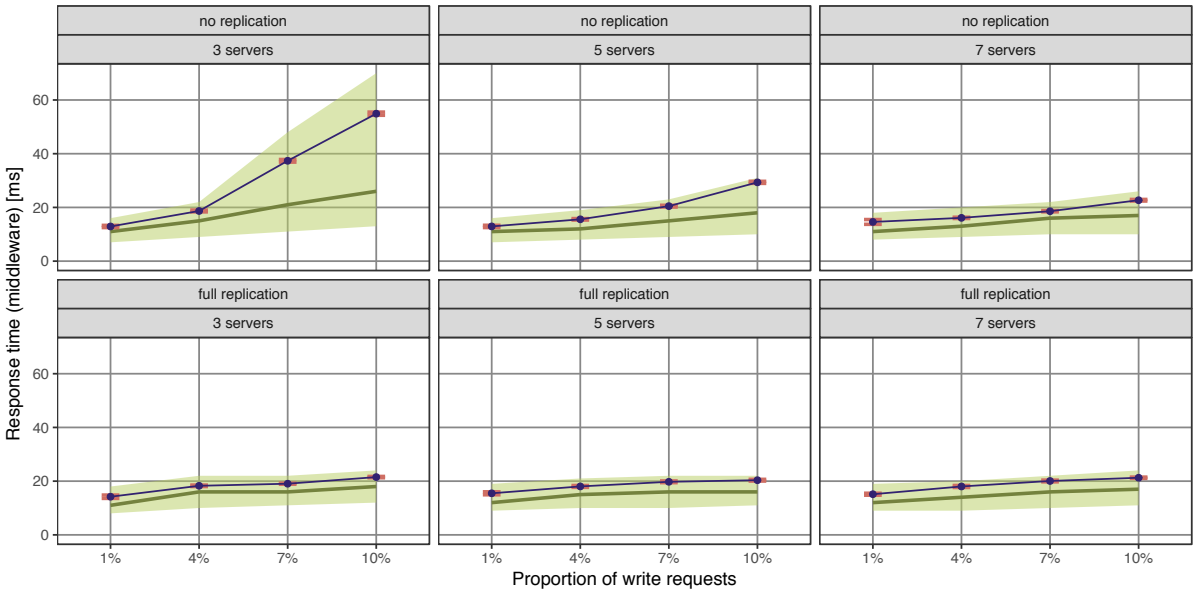

Figure 13: Response time (middleware) to **SET** requests as a function of W, for different values of S and R. The line and points show the mean response time; red errorbars show the 95% confidence interval in a double-tailed t-test; the green area shows the 25% (bottom edge) and 75% (top edge) quantiles of response times.

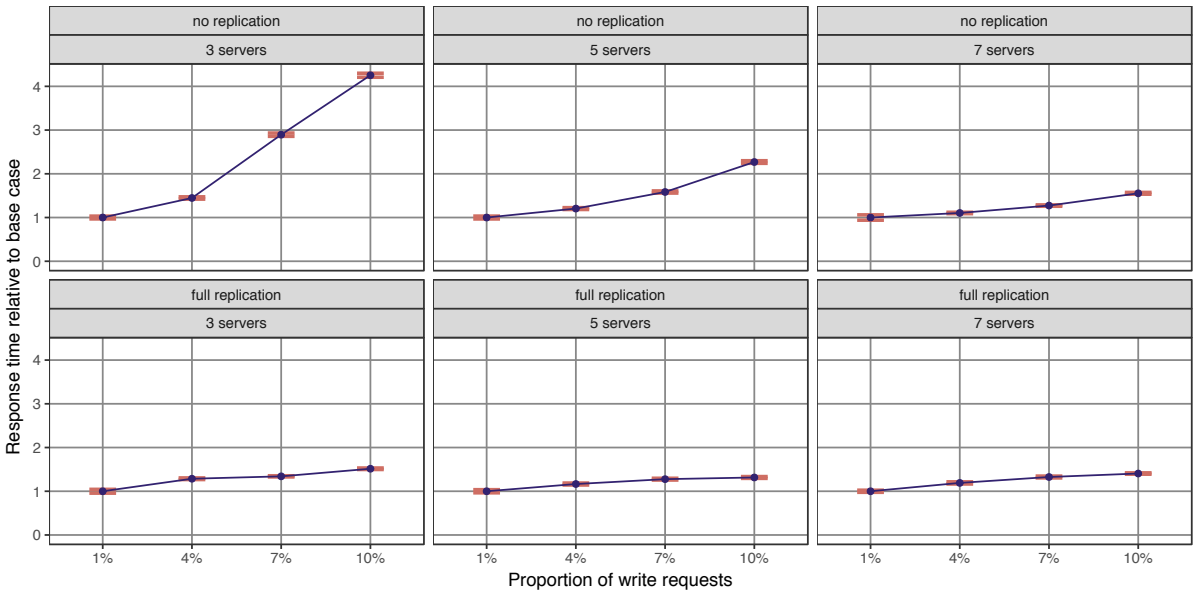

Figure 14: Relative performance of SET requests for different values of S and R: the response time (middleware) of each setup relative to (divided by) the response time in the base case. The base case is taken to be $W = 1\%$ for each combination of R and S. The line and points show the mean response time; red errorbars show the 95% confidence interval in a double-tailed t-test.

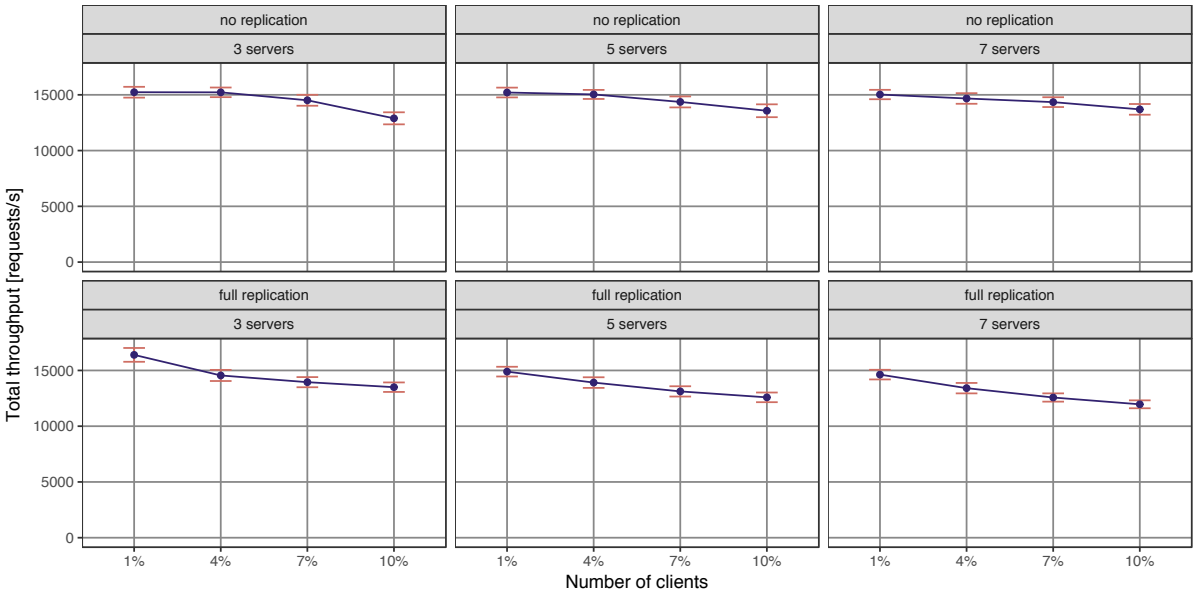

Figure 15: Throughput of SUT as a function of W, for different values of S and R. The line and points show the mean response time; red errorbars show the 95% confidence interval over 10-second samples in a double-tailed t-test.

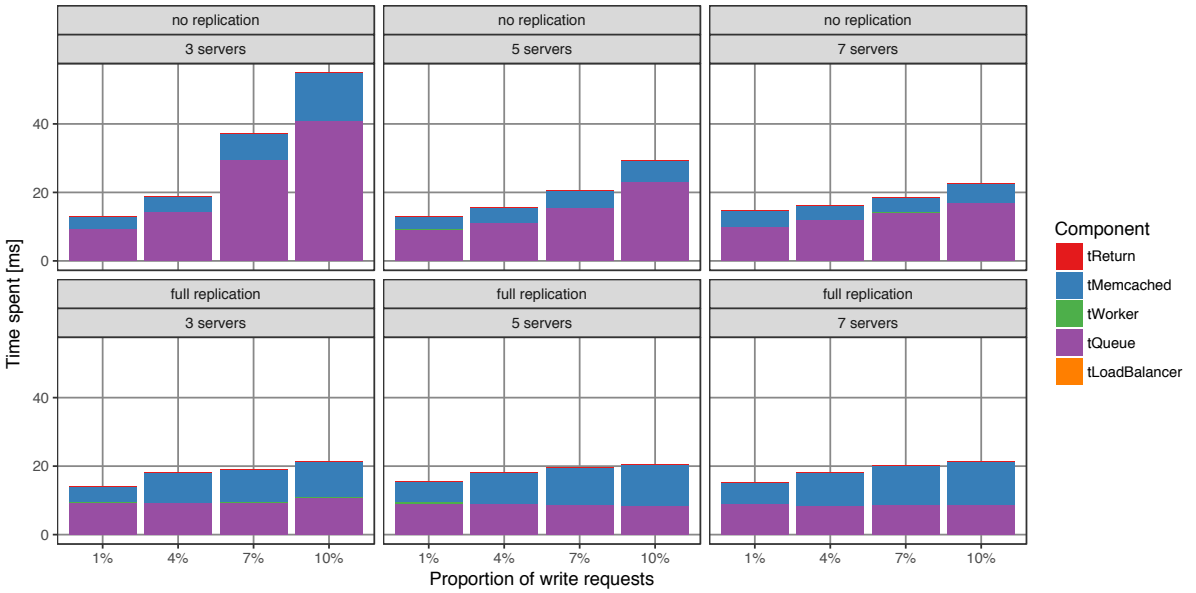

Figure 16: Absolute cost of operations inside SUT, for different values of S and R. Each column is divided into sections by the *average* time spent in the respective component of SUT.

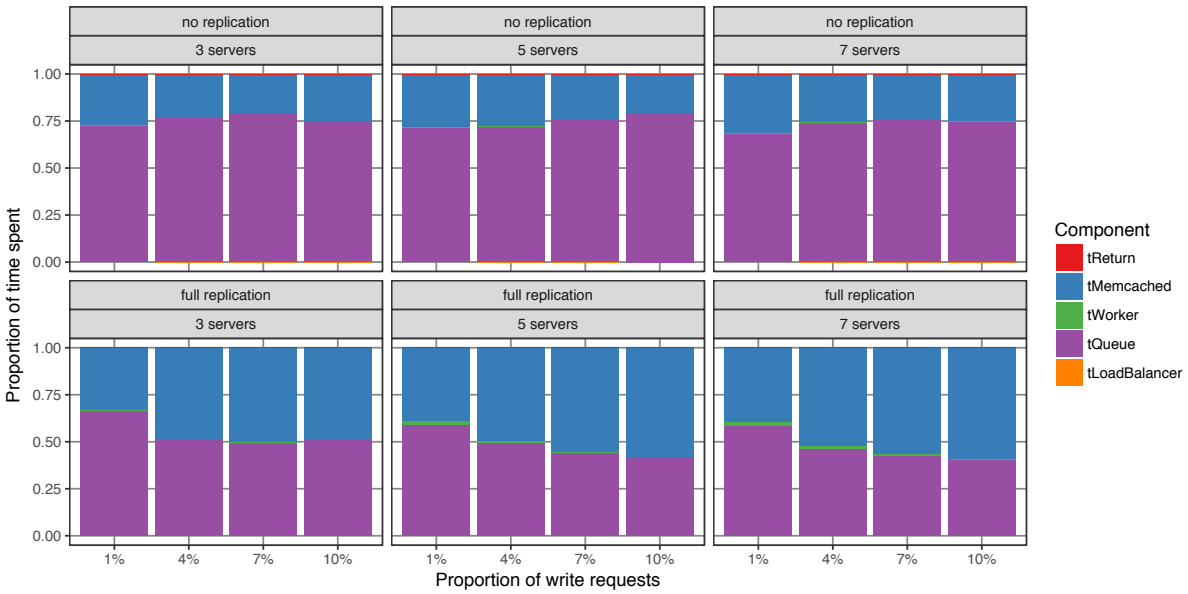

Figure 17: Relative cost of operations inside SUT, for different values of S and R. Each column is divided into sections by the *average* time spent in the respective component of SUT, and then normalised to 100%.

Appendix A: Comparison of middleware and memaslap data

The response time statistics that memaslap outputs are useful but limited. Since using middleware data allows studying the response time distribution in more detail, we would like to use response times measured by the middleware. To do this, however, we need to show that these two are interchangeable up to a constant delay caused by the network latency on the roundtrip between memaslap and the middleware.

Figures 18 and 19 show the mean response times as measured by memaslap and the middleware. It is clear that for all parameter combinations, the difference is indeed constant at about 5ms. Thus, we can rely on response times logged by the middleware.

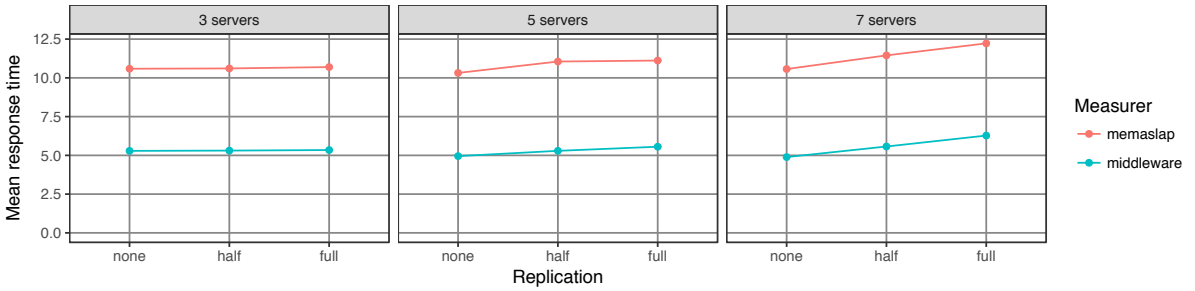

Figure 18: Mean response time as measured by memaslap and middleware in all experiments of Section 2.

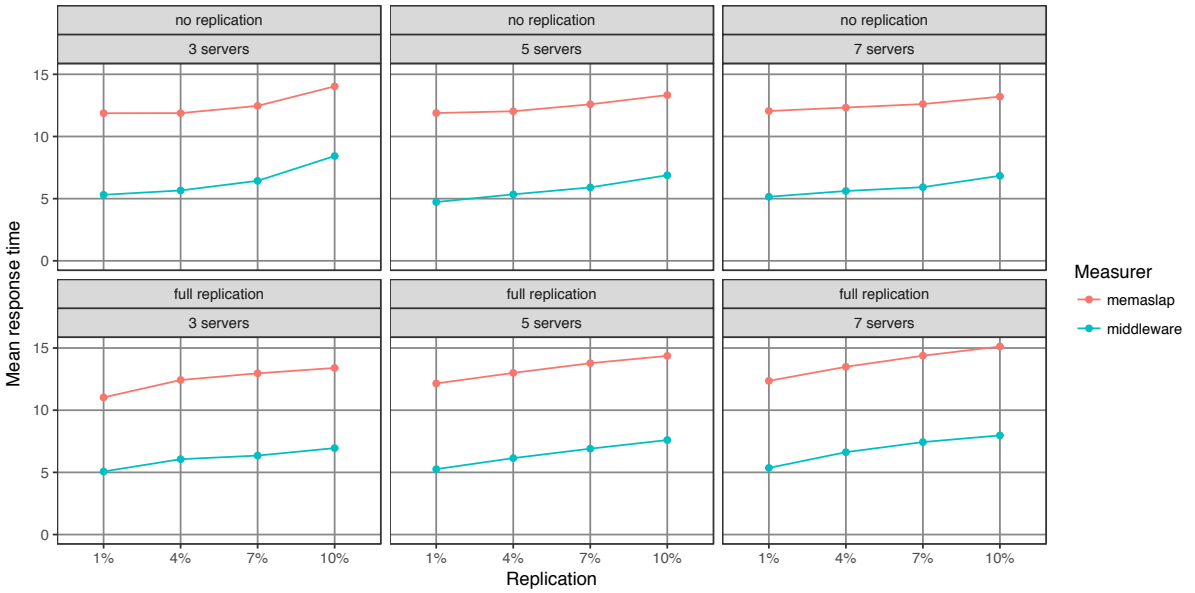

Figure 19: Mean response time as measured by memaslap and middleware in all experiments of Section 3.

Log file listing

Each experiment's logs are compressed into one or more `compressed.zip` files and should be extracted to the directory where the `.zip` file is located. Each location mentioned in the table below is a directory that contains the middleware log (`main.log`), the request log (`request.log`) and memaslap outputs (`memaslap*.out`).

Short name	Location
throughput-C*-T*-r*	gitlab.inf.ethz.ch/.../results/throughput/clients*_threads*_rep*
replication-S*-R*-r*	gitlab.inf.ethz.ch/.../results/replication/S*_R*_rep*
writes-S*-R*-W*-r*	gitlab.inf.ethz.ch/.../results/writes/S*_R*_writes*_rep*