Advanced Systems Lab (Fall'16) – Second Milestone

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

Grading

Section	**Points**
1	
2	
3	
Total	

Contents

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 B.
- S denotes the number of memcached servers in SUT.
- R denotes the replication factor. "No replication" means $R = 1$, "half" or "50%" replication means $R = \lceil \frac{S}{2} \rceil$, "full replication" means $R = S$.
- W denotes the proportion of `SET`s in the workload.
- C denotes the total number of virtual clients (i.e. summed over all memaslap instances).

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

- The system was modified compared to the last milestone. The modifications and new trace results are shown in Appendix A.
- 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 can be considered 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 Optimal 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 Optimal 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

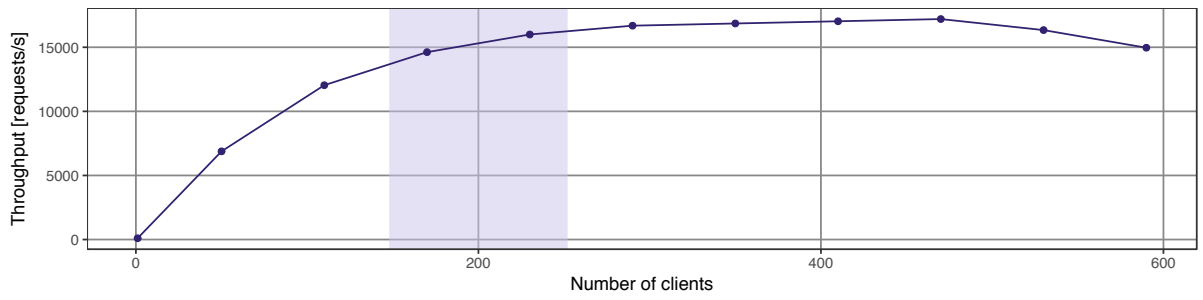

Figure 1: Expected graph of throughput as a function of number of clients (for the optimal value of T). The shaded area shows the range where maximum sustained throughput (and knee of the graph) will occur.

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 1: 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.

1.2.4 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.

1.3 Experiments

Number of servers	$S = 5$
Number of client machines	$\in \{1, 3\}$
Virtual clients	$C \in \{1, 36, 72, 144, 180, 216, 288, 360, 432, 504, 576, 648\}$
Workload	Key 16B, Value 128B, $W = 0\%$
Middleware: replication factor	$R = 1$
Middleware: read threads	$T \in \{1, 16, 32, 64\}$
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

1.4.1 Maximum sustained throughput

Figure 2 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 3 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.

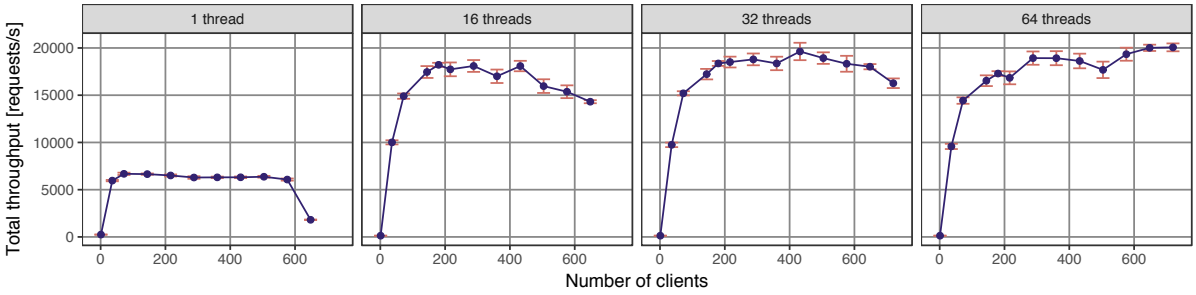

Figure 2: Throughput as a function of C for different values of T. Errorbars show the 95% confidence interval around the mean value which is shown with points connected by lines.

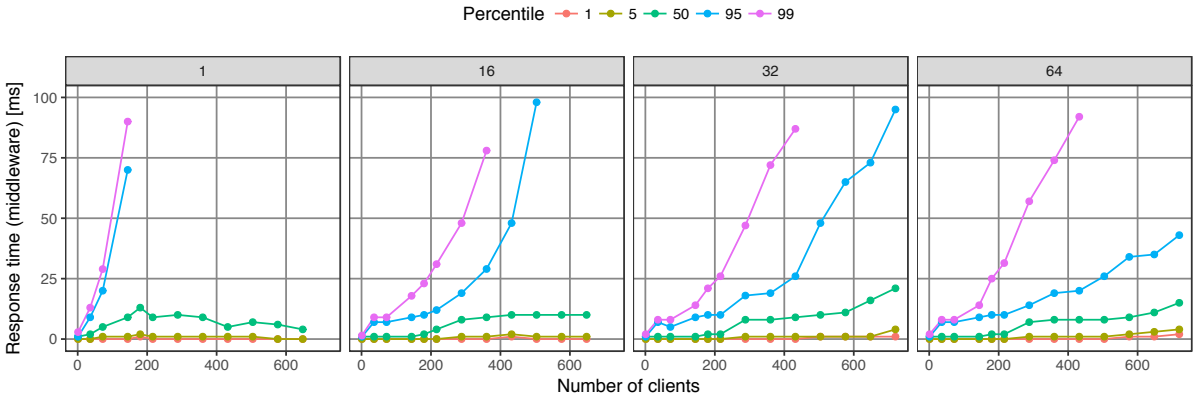

Figure 3: The 1%, 5%, 50%, 95% and 99% percentiles of the response time (middleware) distribution, as a function of C for different values of T. Values above 100ms are not shown.

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. Throughput drops rapidly when decreasing C and increases very slowly when increasing C. Both C and throughput are close to the expected values; T is lower but not by an order of magnitude.

1.4.2 Effect of threads and client load

The dependence of throughput on C for all values of T is as expected: there is a knee at a low value of C, a saturation region and a gradual degradation in performance (for $T = 64$, this degradation probably occurs at an even higher number of clients than tested here). The saturation regions for $T > 1$ have significant fluctuation; however, this is probably noise caused by varying conditions on Azure (this is easy to verify by running more repetitions of these experiments but I decided not to, because the trend is clear and I had limited Azure credit).

Adding threads to the system improves performance: going from $T = 1$ to $T = 16$ has a strong effect: both median response time and the 95th and 99th percentiles are significantly improved. Going to $T = 32$ improves the system much less: it mainly decreases the response time of outliers while keeping median response time similar. Going from $T = 32$ to $T = 64$ makes almost no difference. This happens because at $T = 16$, most of the CPU time available to the middleware is utilised, and at $T = 32$, almost all of it is – which means additional threads don't significantly improve performance.

1.4.3 Breakdown of response time

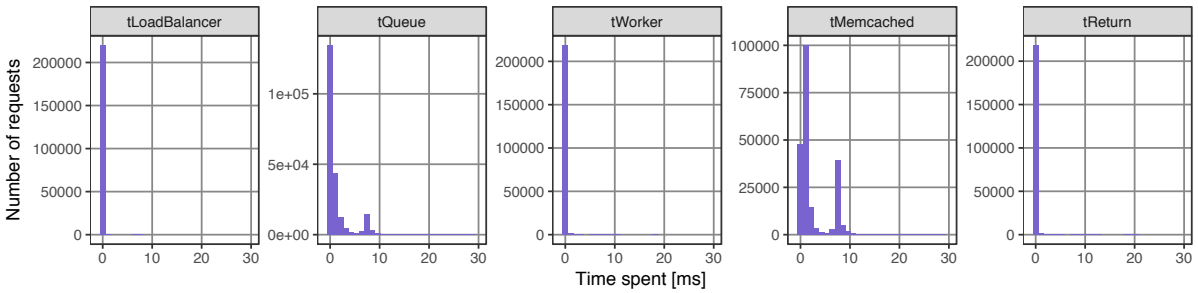

Figure 4: The distribution of times that `GET` requests spend in different parts of SUT. Note the time axis only shows values up to 30ms (this range includes almost all datapoints).

The distribution of time `GET` requests spend 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. The second peak in $tMemcached$ causes the peak in $tQueue$ (because if a request takes a long time in memcached, the next request waits longer in the queue); the peak in $tMemcached$ is most likely to be caused by unusual network conditions for a portion of the requests because there were no `GET` misses in the log.

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 $T = 32$), for `GET` requests.

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 `GET`s and `SET`s 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 `SET`s 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 6. 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.

2.2.2 Throughput

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).

2.2.3 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.

2.2.4 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 for `GET`s

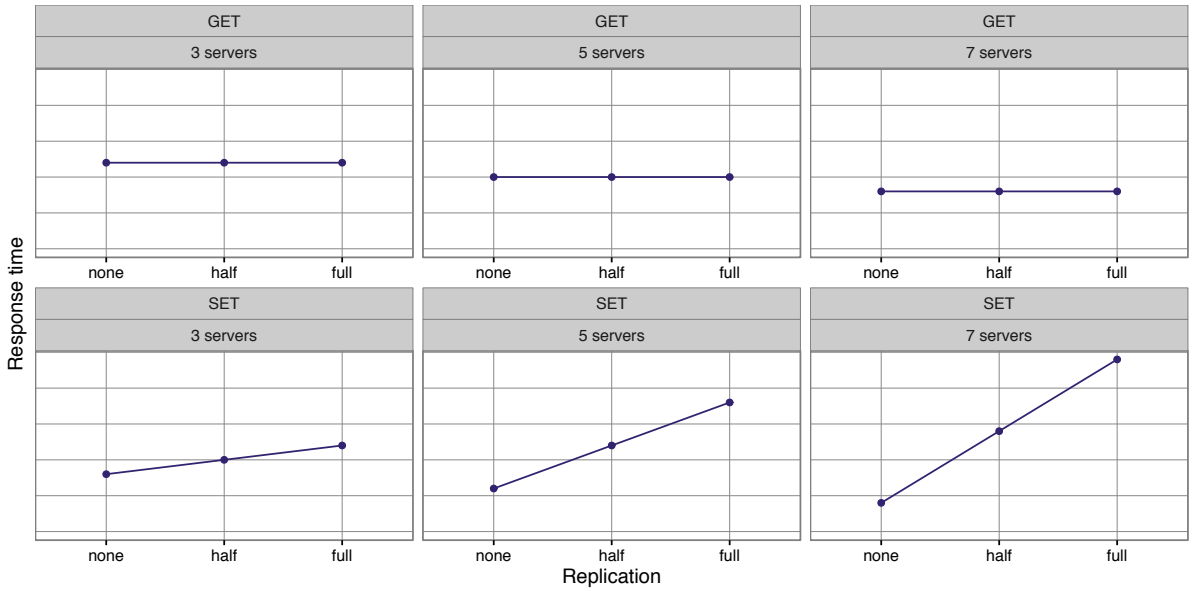

Figure 6: Expected response times of SUT. The vertical (time) axis has an arbitrary but fixed scale for all plots in the top row, and a different but also fixed scale for the bottom row.

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.

For `SET`s, the ideal system would have linear speed-up if $R = const.$ because in that case, adding servers does not increase the amount of work done *per MiddlewareComponent* (again assuming the load balancer does not become a bottleneck). For full replication the ideal system would have sublinear speed-up because each `SET` will be serially written to S servers so the response time would have a component that linearly depends on S.

2.3 Experiments

Number of servers	$S \in \{3, 5, 7\}$
Number of client machines	3
Virtual clients	$C = 180$
Workload	Key 16B, Value 128B, $W = 5\%$
Middleware: replication factor	$R \in \{1, \lceil \frac{S}{2} \rceil, S\}$
Middleware: read threads	$T = 32$
Runtime x repetitions	6min x 3
Log files	replication-S*-R*-r*

2.4 Results

2.4.1 `GET` requests

From Figure 7 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 (`GET`s with high response times) increases. Figure 8 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.

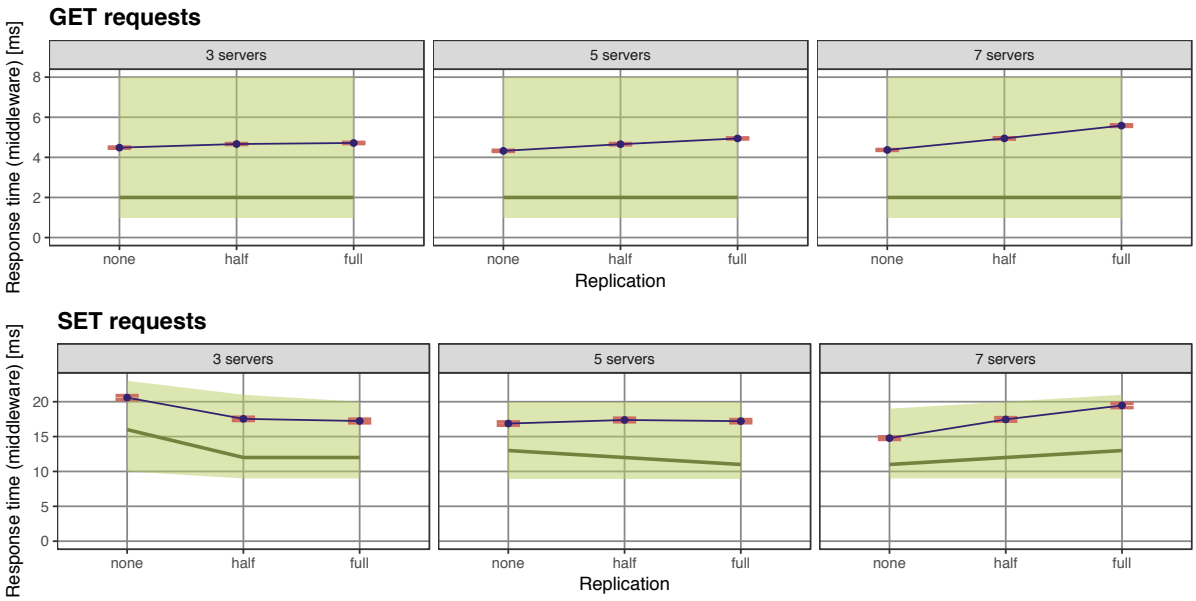

Figure 7: Response time (middleware) as a function of R, for different values of S. The blue line and points show the mean response time; red errorbars show the 95% confidence interval in a double-tailed t-test; the green area shows the 25% (bottom edge) and 75% (top edge) quantiles of response times; the green line shows the median.

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 8) 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 7 shows that increasing R does increase response time for $S = 7$ but unexpectedly, decreases response time for $S = 3$. This is counterintuitive: how can a system that is under a higher load also be faster?

From Figure 8 we see that queueing time actually decreases with R at all values of S and the increase in $tMemcached$ offsets the decrease at $S = 5$ and $S = 7$. Why, then, do `SET` requests spend less time in the queue as R increases? We can explain this by looking at the architecture of WriteWorker. Two steps are done in the same loop: first, if the write queue has any elements, one is taken and sent to all R servers. The second step is checking for responses from memcached (waiting up to 1ms using the function `Selector.select(long timeout)`). This means that if there were no responses from memcached servers, the thread just sleeps 1ms.

The result of this design is that a system with a larger replication factor – which means more responses from memcached servers – sleeps less at `Selector.select()` and thus can faster go back to processing elements from the queue.

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: for 50% replication, response time (and the time spent in each component) is constant and at full replication response time increases slightly with S because of increased $tMemcached$.

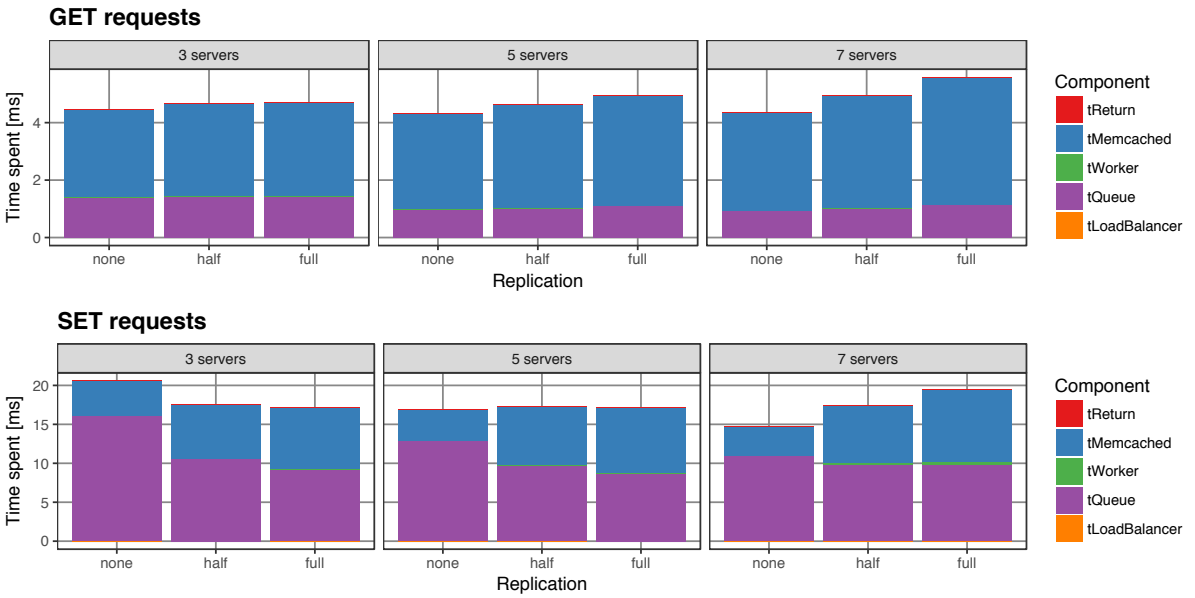

Figure 8: Absolute cost of operations inside SUT as a function of R, for different values of S. Each column is divided into sections by the *average* time spent in the respective component of SUT.

$tMemcached$ is almost constant at 50% replication because the difference between values of R is small: $R \in \{2, 3, 4\}$. At full replication tMemcached has a larger effect because the difference is larger: $R \in \{3, 5, 7\}$. (The slowest response determines $tMemcached$; it can be modelled as the maximum of R samples where each sample is the response time to one request from the middleware to a memcached server.).

2.4.3 Throughput

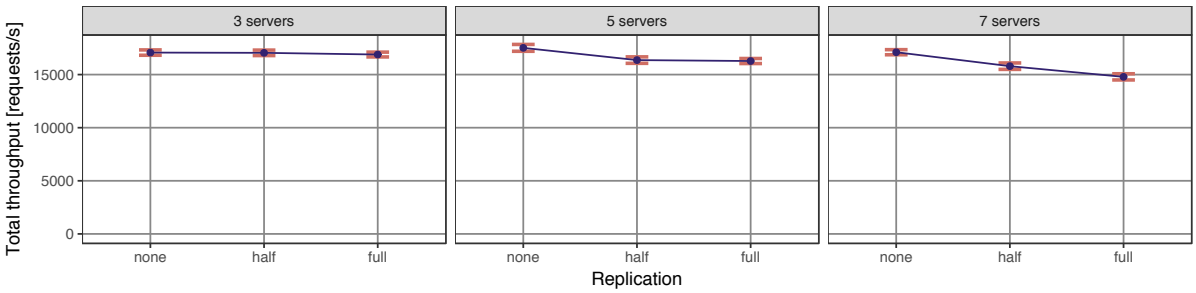

Figure 9: Throughput of SUT as a function of R, for different values of S. The line and points show the mean throughput; red errorbars show the 95% confidence interval over 10-second samples in a double-tailed t-test.

From Figure 9 we can see that throughput does indeed decrease with R – which is in line with the hypothesis –, and higher S amplifies this effect. At $S = 3$ throughput is almost constant; this is because the value of $R \in \{1, 2, 3\}$ does not change enough to make a significant difference, similarly to the previous section. Maximum throughput is achieved at $S = 5, R = 1$ which is likely because in Section 1 we picked the values of C and T that maximised throughput under exactly those parameters.

2.4.4 Relative cost of operations

As hypothesized, increasing R also increases $tMemcached$ for `SET` requests (see Figure 8). Unexpected though was the decrease in $tQueue$ for `SET` requests as R increased, and the increase in $tMemcached$ for `GET`s. Both are explained in previous sections of this chapter.

2.4.5 Scalability

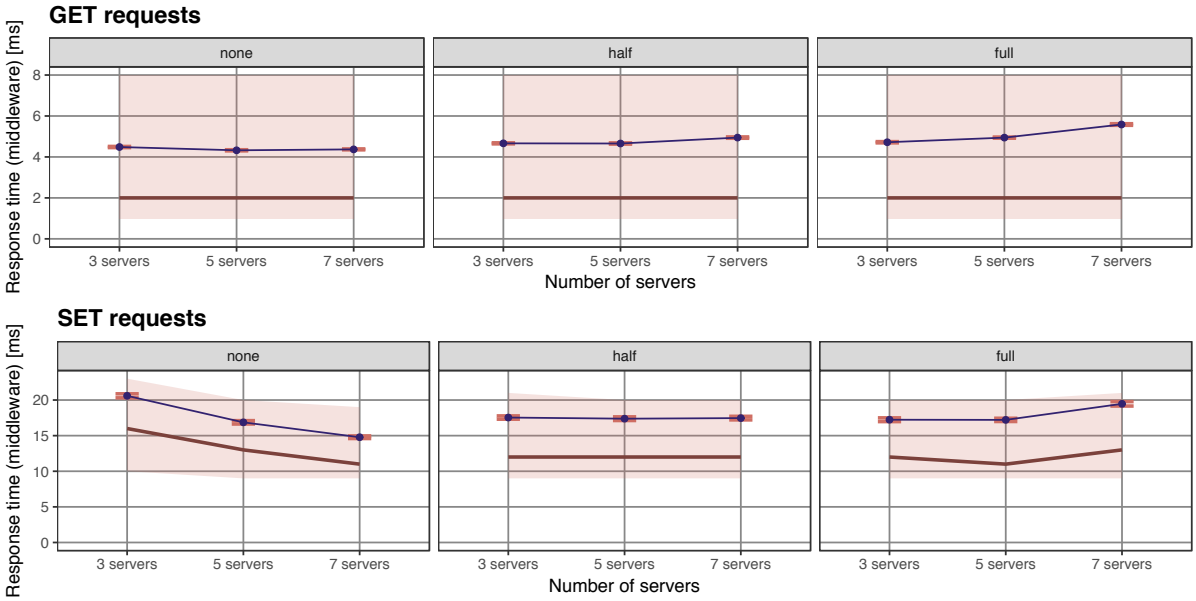

Figure 10: Response time (middleware) as a function of S, for different values of R. The line and points show the mean response time; red errorbars show the 95% confidence interval in a double-tailed t-test; the light red area shows the 25% (bottom edge) and 75% (top edge) quantiles of response times; the red line shows the median.

As Figure 10 shows, there is no speed-up for `GET` requests when we add servers, and there is even a slight increase the mean response time. Increasing S does decrease mean response time to `SET` requests, but only at $R = 1$ – as hypothesized – and sublinearly. At $R > 1$ there is no speed-up. In summary, SUT performs significantly worse than the ideal system described in Section 2.2.4, and worse than expected.

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 `GET` requests

Increasing W will have no impact on the performance of `GET`s. This is because (assuming SUT is not completely saturated) read threads will have a similar amount of requests to process ($1 - W \approx 1$). There may be secondary effects due to e.g. write threads requiring more CPU time, but this should be negligible.

3.2.2 `SET` requests

Increasing W will decrease total throughput and increase mean `SET` response time for any combination of S and R because `SET`s take longer to process than `GET`s, and because there are more requests per write thread compared to read threads. 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 `SET` request (i.e. the response time of each write request will be higher).

3.2.3 Relative impact

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 `SET` requests, which in turn increases the queue wait time $tQueue$.

3.3 Experiments

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

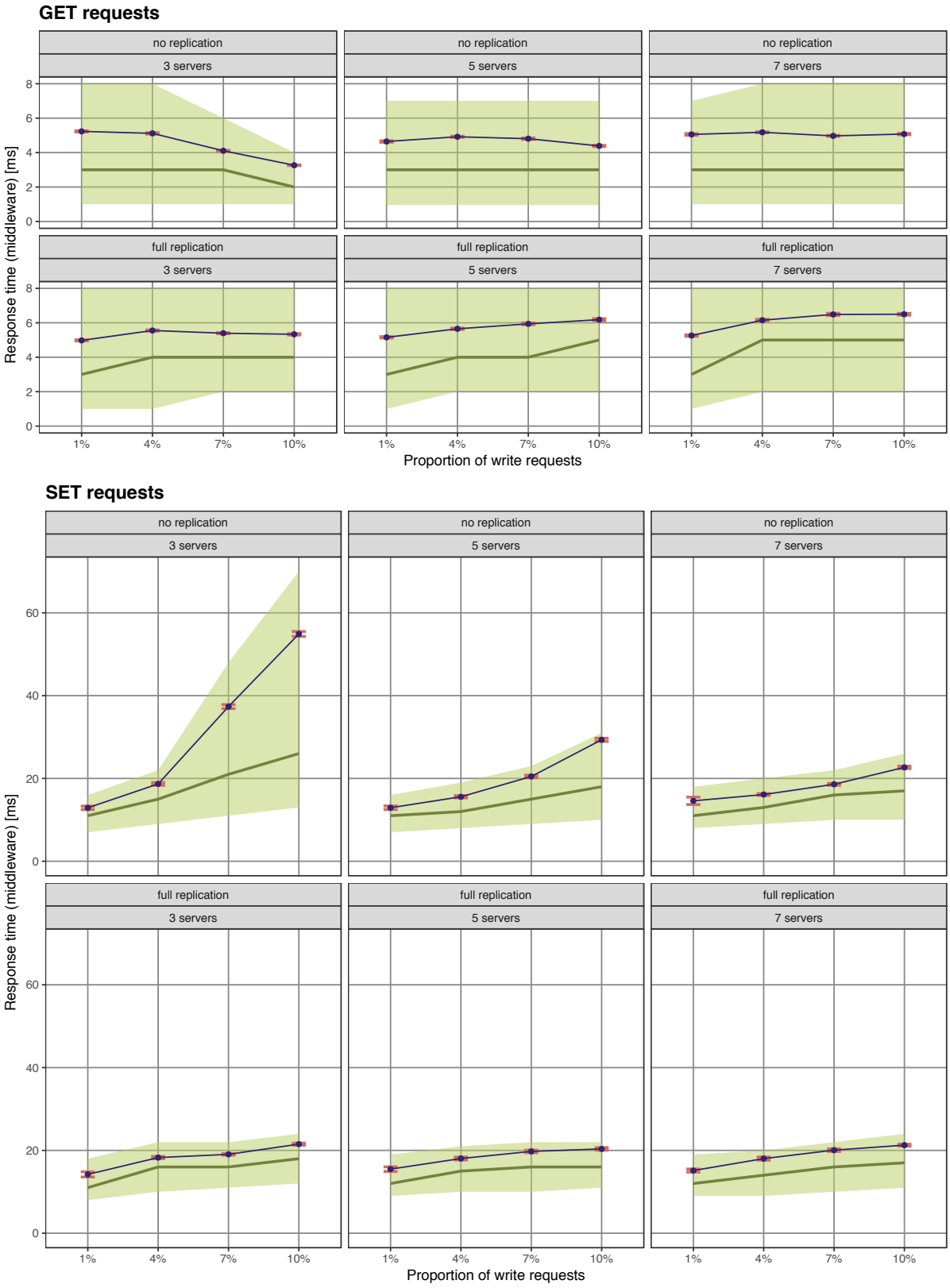

Figure 11: Response time (middleware) as a function of W, for different values of S and R. The blue line and points show the mean response time; red errorbars show the 95% confidence interval in a double-tailed t-test; the green area shows the 25% (bottom edge) and 75% (top edge) quantiles of response times; the green line shows the median.

3.4 Results

3.4.1 Impact on `GET` requests

Figure 11 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 relatively stable with one exception. This is roughly as expected (response time does increase slightly with W, but this can be attributed to increased network latency because *tQueue* is stable across parameter combinations). The plot of breakdown of response time to `GET`s was not included in this report but is available on GitLab.

In the exceptional case, $(S = 3, R = 1)$, *tQueue* decreases as W increases. Given that $(S = 3, R = 3)$ has an otherwise similar behaviour without the decrease in *tQueue* we can conclude that the decrease is caused by read threads reaching a knee in their performance function: removing just a small proportion of the load gives a large gain in performance.

3.4.2 Impact on `SET` requests

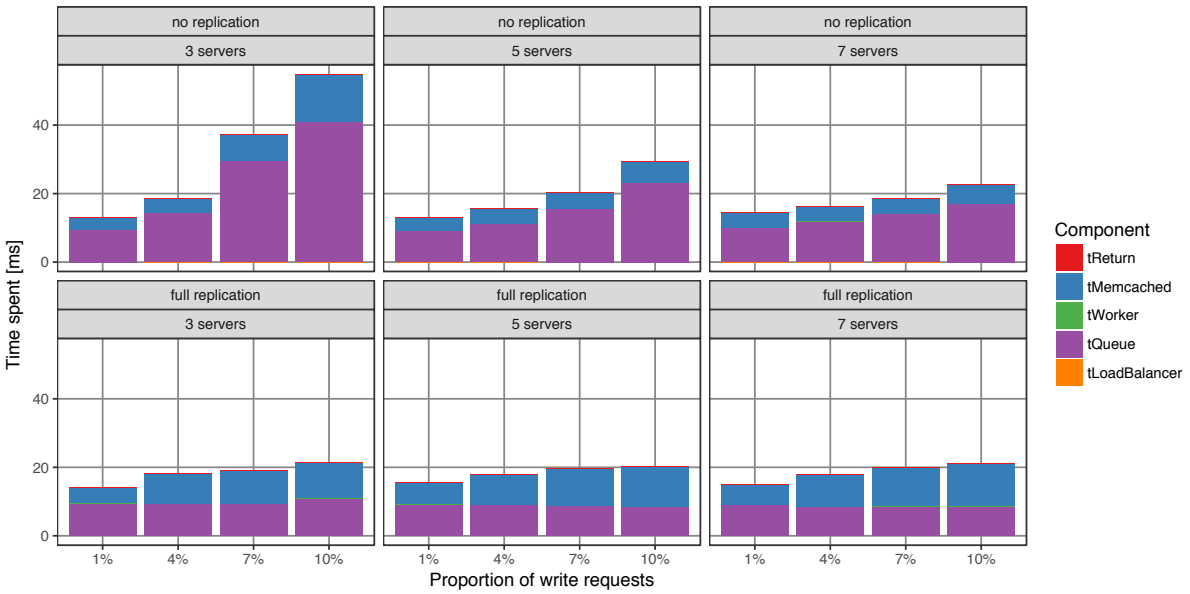

Figure 12: Absolute cost of `SET` requests inside SUT, for different values of S and R. Each column is divided into sections by the *average* time spent in the respective component of SUT.

Figure 11 shows the effect of W on `SET` requests, and Figure 12 shows the relative cost of operations inside SUT. It is clear that increasing W also increases response time – this is in line with the hypothesis. However, the prediction that fully replicated would suffer the most didn't hold: in fact, while response time seems to depend linearly on W in the fully replicated case, the dependence looks exponential in the case of $R = 1$. The reasoning for why $R = 1$ performs unexpectedly badly (especially for $S = 3$) can be found in Section 2.4.2.

3.4.3 Throughput

As Figure 13 shows, throughput does indeed decrease with W for all combinations of S and R, confirming the hypothesis. I also predicted that throughput would suffer more for fully replicated setups; this is indeed the case although the difference is not large (the slope of the line in Figure 13 is steeper in plots of the top row compared to the bottom row).

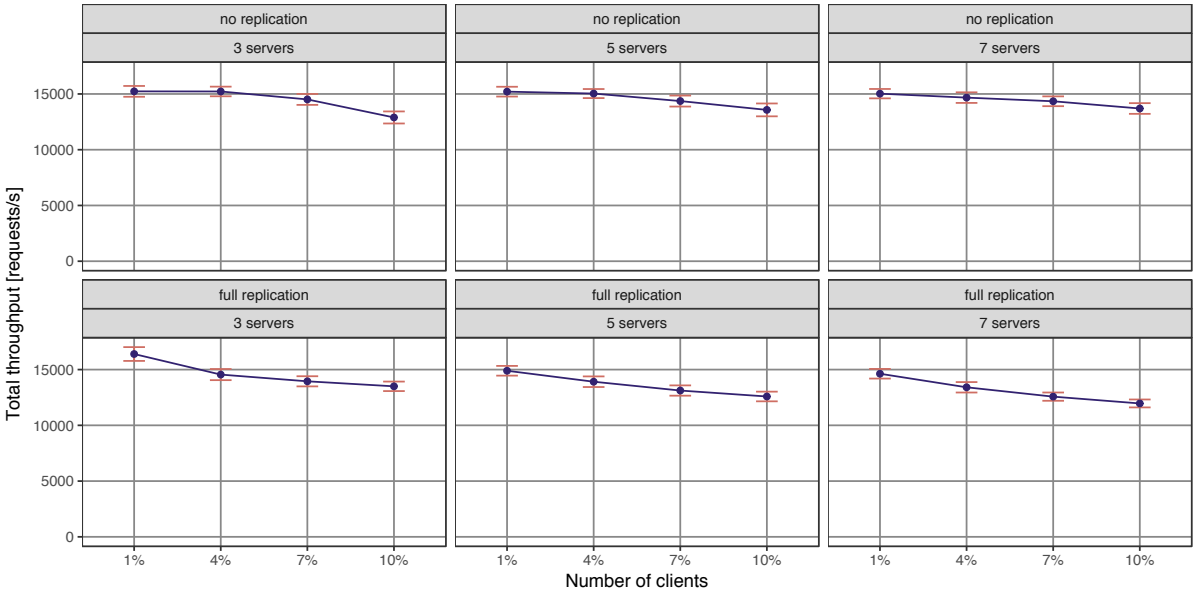

Figure 13: Throughput of SUT as a function of W, for different values of S and R. The line and points show the mean response time; red errorbars show the 95% confidence interval over 10-second samples in a double-tailed t-test.

3.4.4 Reasons for reduced performance

Figure 12 shows the relative cost of operations inside SUT. For full replication, the relationship is straightforward: *tQueue* is constant and the total response time is mostly affected by $tMemcached$ increasing; the reasons for this increase (increased network latency and/or higher load on memcached) have been discussed in Section 2.4.

For $R = 1$, however, the relationship is different. The major component in response time is *tQueue* which increases significantly with W. The design decisions causing the non-replicated system to perform much worse than the replicated one were discussed in Section 2.4.2 and should hold here as well.

3.4.5 Relative impact

I predicted that the largest relative performance decrease would be for a lower number of servers, i.e. $S = 3$ would be hit worst by increasing W; Figure 14 confirms this strongly for $R = 1$ and mildly for $R = S$.

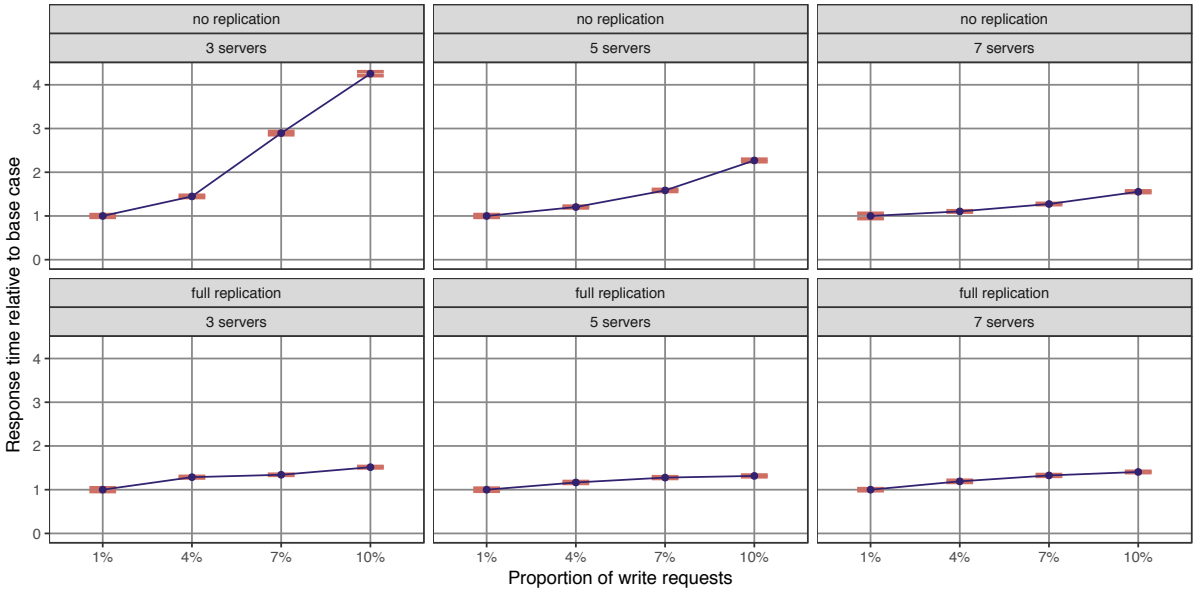

Figure 14: Relative performance of `SET` requests for different values of S and R: the response time (middleware) of each setup relative to (divided by) the response time in the base case. The base case is taken to be $W = 1\%$ for each combination of R and S. The line and points show the mean response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

Appendix A: 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 15 and 16, 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.

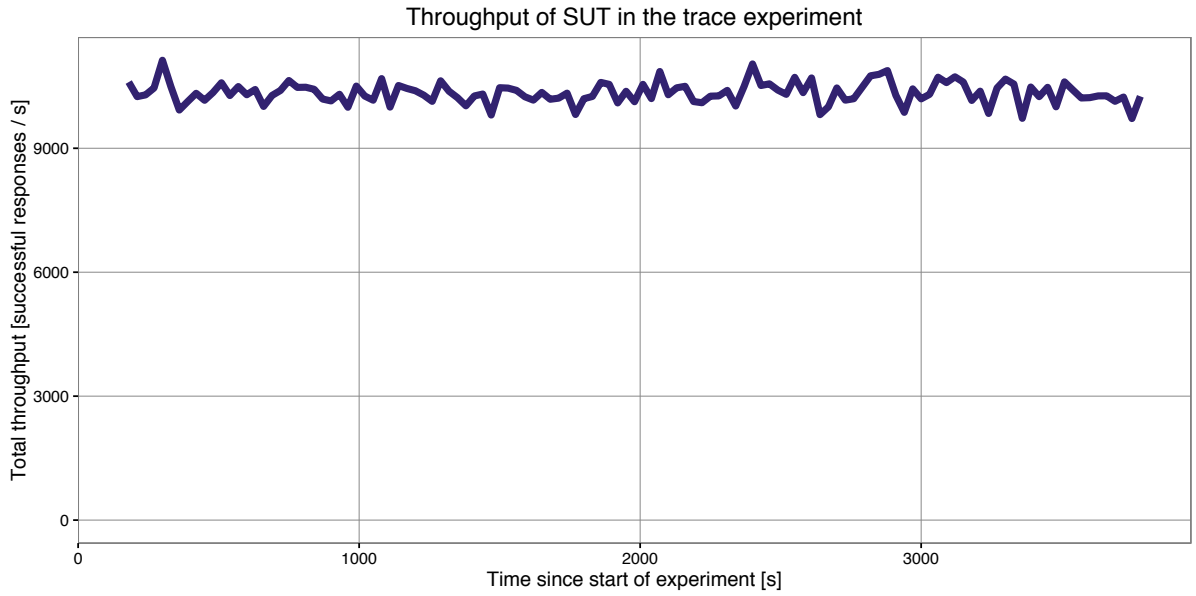

Figure 15: Throughput trace of the middleware.

Figure 16: Response time trace of the middleware as measured by memaslap.

Appendix B: 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 17 and 18 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.

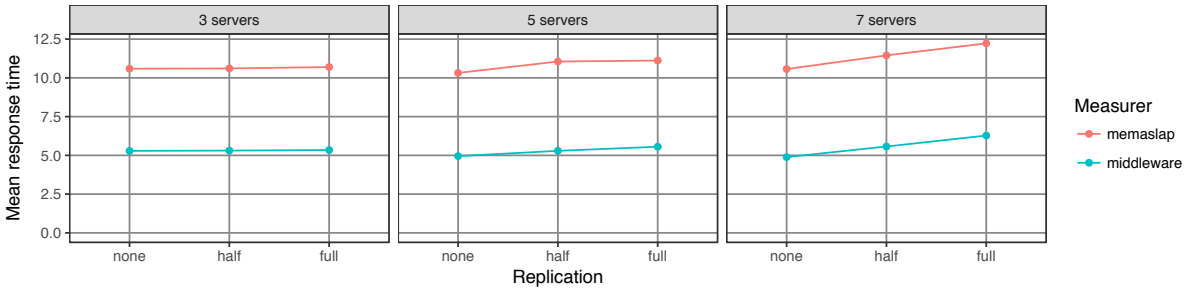

Figure 17: Mean response time as measured by memaslap and middleware in all experiments of Section 2.

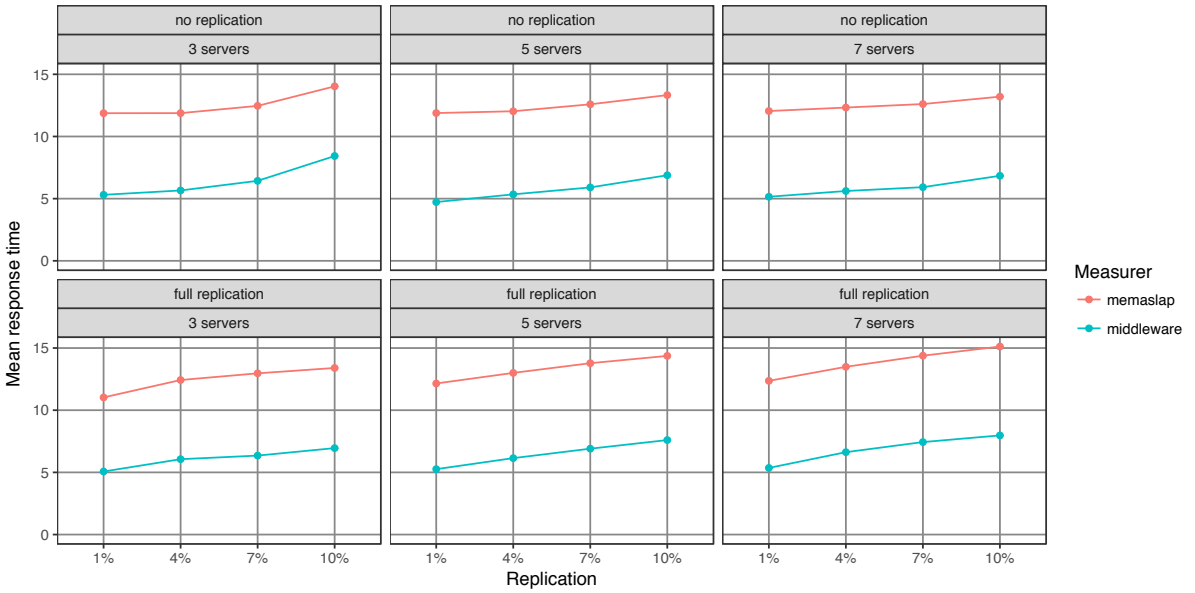

Figure 18: Mean response time as measured by memaslap and middleware in all experiments of Section 3.

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*