Advanced Systems Lab (Fall'16) – Second Milestone

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

Grading

Section	**Points**
1	
2	
3	
Total	

1 Maximum Throughput

The aim of this experiment is to find the highest achievable throughput of the system when being placed in front of five server machines with no replication and a read-only workload configuration, and to find the minimum number of threads and clients that together achieve this throughput. The experiment has been run *three* times, in order to first localize the optimal configuration point, and then gradually focus around it.

1.1 Experimental Setup

The setups of the three runs of the experiment are outlined in Table 1, where the only changes to the configuration between runs are marked in different colors, and separated by a bullet:

Run 1 • **Run 2** • **Run 3**.

The experiment is carried out by the script located at `https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/experiments/03_maximum_throughput`, which collects all the `memaslap` and middleware outputs (`*.log`, inclusive repetitions), the extracted throughput data (`tps.data`), and also produces plots of the latter. The notation $a : b : c$ denotes the set of numbers $a, a + b, a + 2b, \ldots, c$. Note that throughout the whole experiment, the five client machines always run with the same number of virtual clients. Thus, e.g. for the first run, at the beginning a total of $5 \cdot 10 = 50$ *virtual clients* connect to the middleware, while at the end $5 \cdot 90 = 450$.

Servers	5 × A2
Threads / server machine	1
Clients	5 × A2
Virtual clients / machine	**10 : 20 : 90** • **10 : 10 : 70** • **40 : 4 : 60**
Keys size	16 Bytes
Values size	128 Bytes
Writes	0%
Overwrite proportion	90%
Window size	1 KByte
Statistics	1 Second
Middleware	1 × A4
Pool threads	**8 : 8 : 40** • **16 : 8 : 32** • **16, 24**
Replication	1
Runtime × repetitions	90 Seconds × 4
Log files	`max-tps1` • `max-tps2` • `max-tps3`

Table 1: Maximum throughput experiment setup.

1.2 Practical Issues

From a practical standpoint, some noteworthy difficulties were encountered while designing the experiment. The major issue was that with the read-only configuration, the `memaslap` clients would initially spend some time filling up the `memcached` servers,[1] and as a consequence they would then start at different times. This is because they would finish filling `memcached` at different times, reaching time differences of up to 15 seconds, when many clients connect simultaneously to the middleware.

To bear with this additional difficulty, a 90 seconds runtime was selected to allow the clients to reach a stable phase all together. This phase was assumed to contain the interval from 30

[1] This is the reason for setting the window size to 1k, as stated in Table 1: with such window size, the initial fill-up phase takes much less than with the default value of 10k.

to 60 seconds, from which the output of the clients was then read. Note that this in particular implies that the throughput values of the `memaslap` clients are not synchronized; that is, the throughput of some client at "its" time 30 is *not* the throughput of that same client at time 30 of some other client. Nevertheless, since the average behavior of the middleware is sought, it makes sense to add up throughputs of different clients at same times, since it is assumed that they have all reached a stable phase.

1.3 Results

This section presents the result of the three runs of the experiment. Each successive run is more granular in the number of virtual clients, and the number of threads is progressively reduced, until an optimal configuration which yields the *maximum throughput* can be clearly observed. For each configuration, the *mean* μ_{cr} and *variance* σ^2_{cr} of the throughput is calculated from the samples of the middle third (see previous section), where c is the index of one of the $C = 3$ clients and r is the index of one of the $R = 4$ repetitions. Then *mean* $\bar{\mu}$ and *standard deviation* $\mu\sigma$ are recorded according to

$$\bar{\mu} \doteq \frac{1}{R}\sum_{r=1}^{R}\sum_{c=1}^{C}\mu_{cr}, \quad \bar{\sigma} \doteq \frac{1}{R}\sum_{r=1}^{R}\sqrt{\sum_{c=1}^{C}\sigma^2_{cr}}. \tag{1}$$

1.3.1 Run 1: Searching the Optimal Point Using a Wide Range

As outlined in Table 1, the first run investigates the behavior of the middleware when connected to five servers and five client machines, where each of the latter runs with $10, 20, \ldots, 100$ virtual clients. Therefore, the behavior of the middleware with $50, 100, \ldots, 500$ virtual clients is investigated. The number of threads is variated from 8 to 40, with a step of 8. The result of this run of the experiment is summarized in Figure 1.

Hypothesis 1. *The throughput as a function of connected virtual clients should initially grow quasi-linearly, and then reach a saturation point at which no more growth is observed. Eventually, when the number of clients becomes too large, the throughput may start to decrease, and in general the system should become more unstable. Also, the larger the number of threads in the pool thread, the higher is the expected throughput, but at some point having more threads should eventually provide no more gain, but instead decrease the performance.*

Analysis. First, note that the point at 450 virtual clients / 16 threads on Figure 1 clearly appears to be outlier. This is because in one of the four repetitions for that specific configuration, one of the clients has not produced any output, due to an unknown reason. That point *should therefore be ignored.*

Otherwise, the general behavior of the system is exactly as expected. From 50 to 150 virtual clients a quasi-linear growth is observed, and after that a saturation point appears to be reached. Anyway, it is clear that starting already from 350 virtual clients, irrespective of the number of threads, the system becomes quite unstable. This fact is supported by the variance, which gets larger and larger for each number of threads, as the number of virtual clients increases. Therefore it can be safely assumed that the optimum point lies somewhere between 150 and 350 virtual clients.

Regarding the number of threads in the thread pool, there is a clear gap between 8 and all other configurations. In fact, in the region of interest from 150 to 350 virtual clients, the throughput seems to reach similar values for all configurations of $16, 24, 32$ and 40 threads. But this also implies that a point where having more threads does not increase the performance is reached. In fact, almost always, the throughput with 40 threads is less than the throughput with $16, 24$, or 32 thread. Therefore, the search space for the number of threads can be safely reduced to $16, 24, 32$. ♢

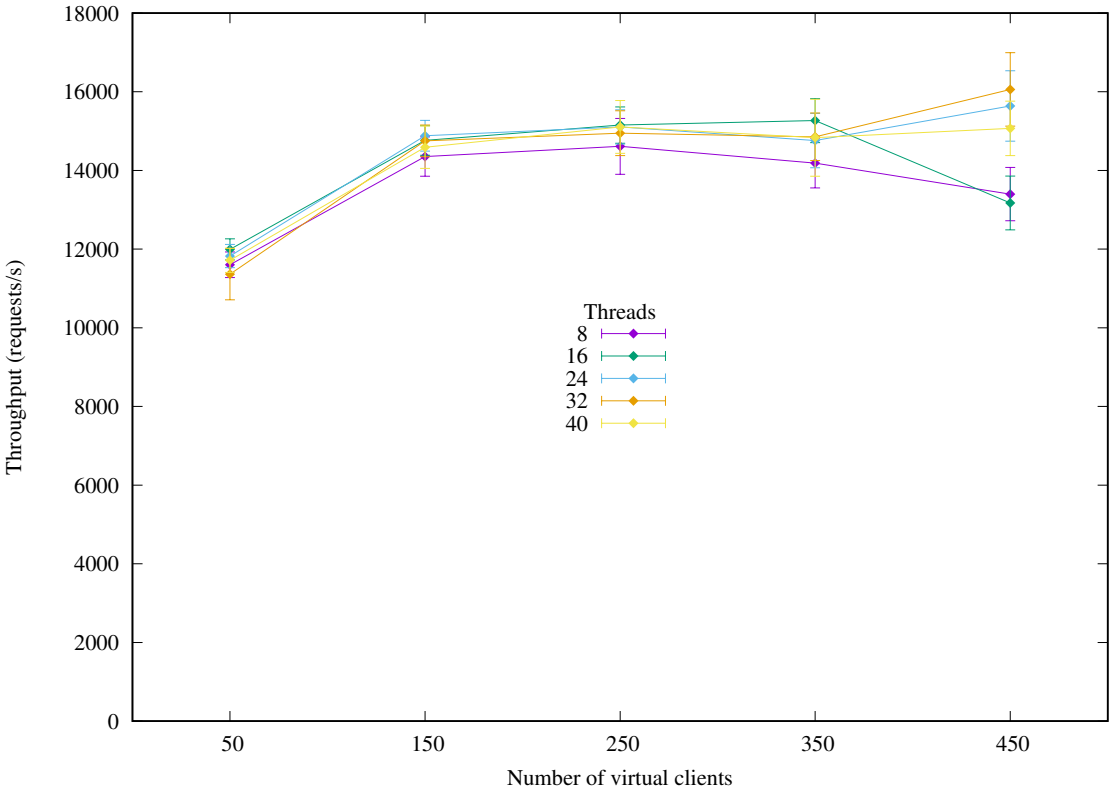


Figure 1: Throughput of **Run 1** of the maximum throughput experiment.

1.3.2 Run 2: Seeking Confirmation and Reducing The Search Space

The configuration of this run is twice as granular as the previous in the number of virtual clients, but the search is only up to 350. For the number of threads, only the values $16, 24$, and 32 are considered. The result of this run of the experiment is summarized in Figure 2.

Hypothesis 2. *Same as Hypothesis 1.*

Analysis. The growth of the throughput is as expected (see analysis of Hypothesis 1). Due to the increased granularity, it emerges that the optimal point is localized at around 250 virtual clients, and it is almost identical for all numbers of threads. The result of this run allows to further shrink the search space to the interval $200, 300$ for the number of virtual clients. ♢

1.3.3 Run 3: Close-up Around the Optimal Point

For this run, the number of virtual clients has been set from 200 to 300 with a step of 20, with the hope of having peak somewhere in the middle (as suggests the previous run). Since from the previous run it also emerged that $16, 24$, and 32 threads yield very similar throughput values in general, in this run only 16 and 24 threads are considered for the thread pool. The result of this run of the experiment is summarized in Figure 3.

Hypothesis 3. *There should be a peak in the throughput at about* 250 *virtual clients, with one of the two possible values for the thread pool size resulting in a better performance.*

Analysis. Unfortunately, Figure 3 suggests that the hypothesis should be rejected. In fact, neither there is an evident peak in the middle of the graph, nor does one of the two numbers of threads always dominate the performance of the other. Overall, Figure 3 supports the previous hypothesis that in this interval a saturation point is reached, but also outlines that starting

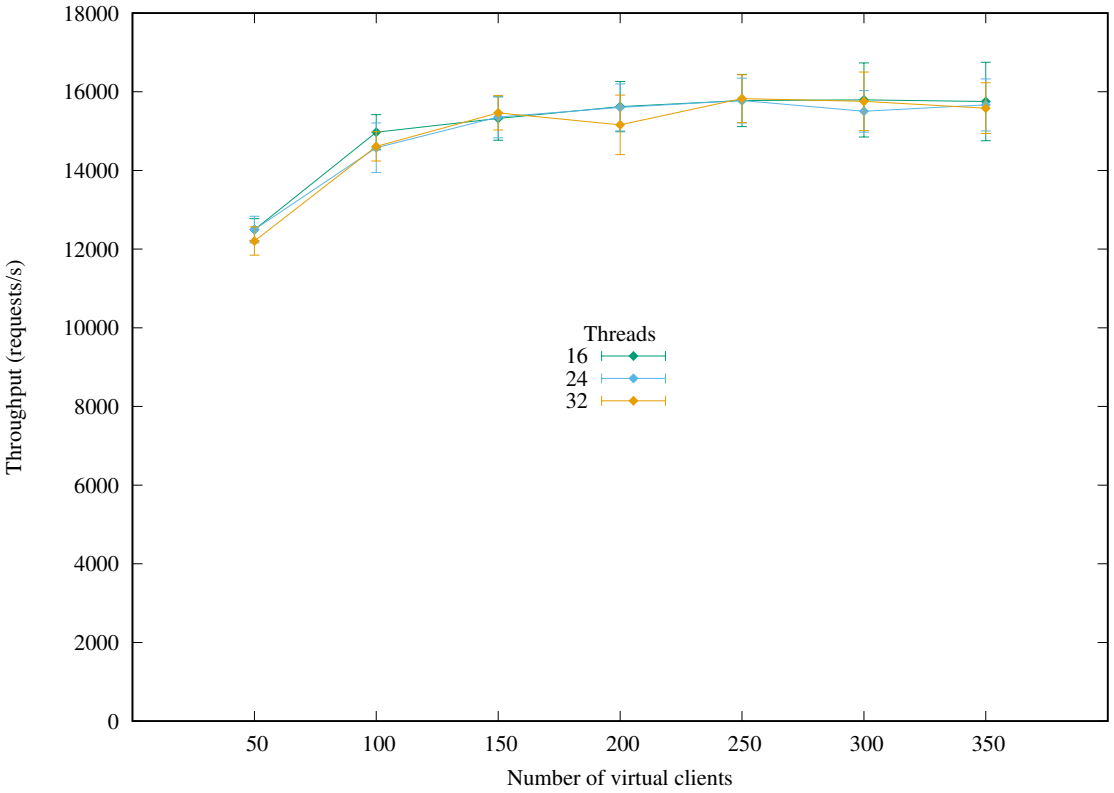


Figure 2: Throughput of **Run 2** of the maximum throughput experiment.

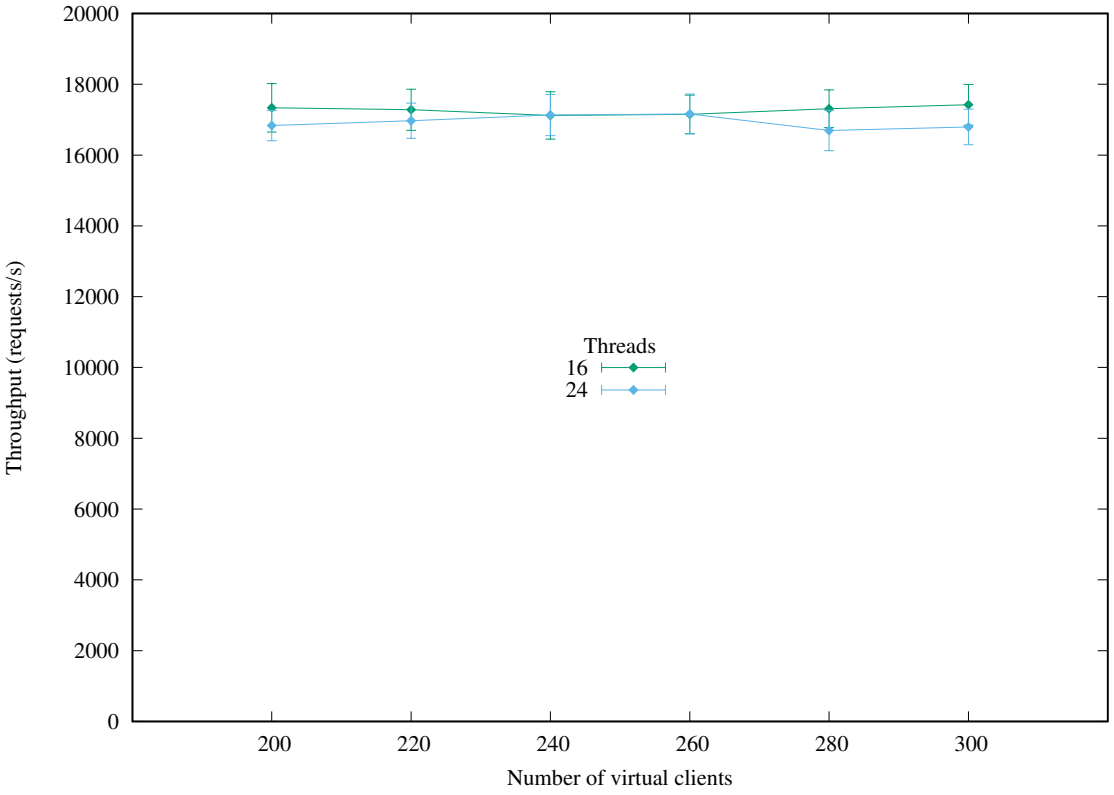


Figure 3: Throughput of **Run 3** of the maximum throughput experiment.

from 200 virtual clients, the change is very small, and it is not clear whether it is for the better or the worst. For all those reasons, since a minimal configuration is sought, the optimal number of virtual clients should be put at 200 and the optimal number of threads at 16. ♢

1.4 Optimal Configuration

From the previous analyses, it emerged that the maximum throughput achieved by the system consists of about 16 000 requests per second for this setup, but of course this number has to be taken with a grain of salt, since it is already clear from the above plots that the network introduces a lot of bias (for some runs, the throughput is constantly higher than in other runs).

But the most interesting result of the previous analyses, is *when* this maximum throughput is achieved for this setup. **Run 3** did not give any gain of information, thus relying on the results of **Run 2** instead, this *minimal* point can be safely placed at about

200 virtual clients and 16 threads.

This is also supported by Table 2, generated using `https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/analyses/03_opt`, which reports the percentiles of $\bar{\mu}$ for $T = 16, 24$ threads and $150, 200, 250$ virtual clients. The row corresponding to the chosen configuration has been colored in blue, and it can be seen that it is stable and better than any setup with 150 virtual clients.

				Percentile				
C	***T***	$\bar{\mu}$	$\bar{\sigma}$	**50th**	**80th**	**90th**	**95th**	**99th**
150	**16**	15 321.33	551.31	15 415	15 927	16 288	16 385	16 869
	24	15 357.75	529.41	15 497	15 968	16 214	16 471	16 716
200	**16**	15 624.83	639.44	15 704	16 439	16 649	16 840	17 163
	24	15 602.28	598.77	15 711	16 175	16 396	16 504	16 951
250	**16**	15 776.90	662.85	15 754	16 482	16 874	17 205	17 842
	24	15 777.95	568.35	15 851	16 406	16 678	16 882	17 466

Table 2: Mean, standard deviation, and percentiles of throughput for points of interest.

1.5 Detailed Breakdown of Time Spent in The Middleware

In this section a detailed breakdown of the time spent by the requests in the middleware for the optimal configuration found above is presented. All the table values and plots are generated by the script located at `https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/analyses/03_breakdown`. The script first filters out only the entries relative to `GET` operations from the middleware log file for the selected configuration from **Run 2**. Then, since the stable phase is assumed to lie in the middle third of the runtime of the clients, only the middle third of the `GET` entries is taken. (Note that the entries are sampled every 100.) Since the experiment is run with 4 repetitions for each configuration, the middle thirds of each repetition are successively put together, and finally mean, standard deviation, 50th, 80th, 90th, 95th, and 99th percentiles are taken from this aggregated list of entries. These values are summarized in Table 3 for each of the timing reported by the middleware, i.e.,

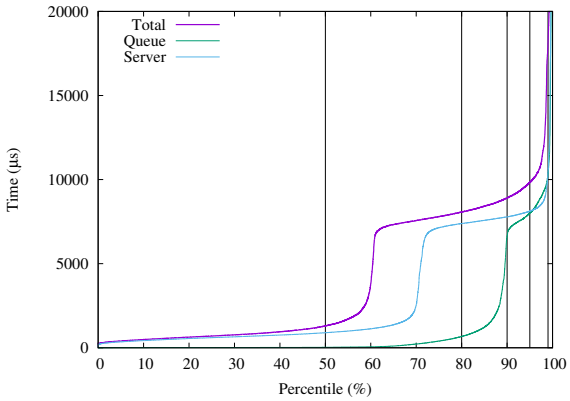
$$T_{\text{tot}} \doteq t_{\text{left}} - t_{\text{arr}}, \; T_{\text{que}} \doteq t_{\text{deq}} - t_{\text{enq}}, \text{ and } T_{\text{srv}} \doteq t_{\text{recv}} - t_{\text{sent}}, \tag{2}$$

Figure 4 shows the empirical distribution of these timings obtained by the samples of this run (Figures 4b to 4d), as well as a graph showing all the samples sorted from smallest to largest (Figure 4a). From the latter, any percentile for the three timings can be directly read. Note that in all the four graphs, times larger than 20 000 μs have been cut out.

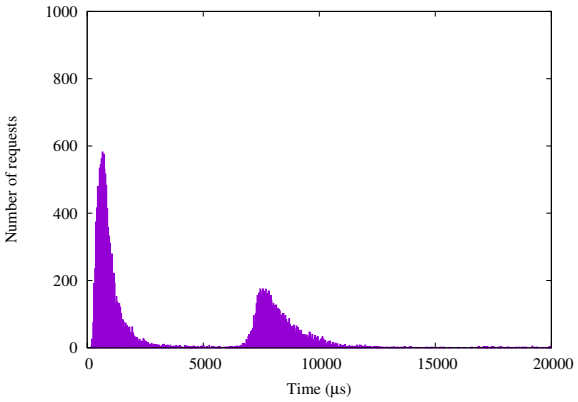
Hypothesis 4. *Each request should spend a relatively small amount of time in the queues, and the time spent in the servers should dominates the total time.*

			Percentile				
Times	**Mean**	**Std. Dev.**	**50th**	**80th**	**90th**	**95th**	**99th**
Total (T_{tot})	4 262.95	9 307.09	1 302	8 072	8 910	9 826	18 023
Queue (T_{que})	1 198.56	3 871.07	28	675	6 609	7 993	10 278
Server (T_{srv})	3 009.18	8 408.58	894	7 386	7 782	8 125	10 363

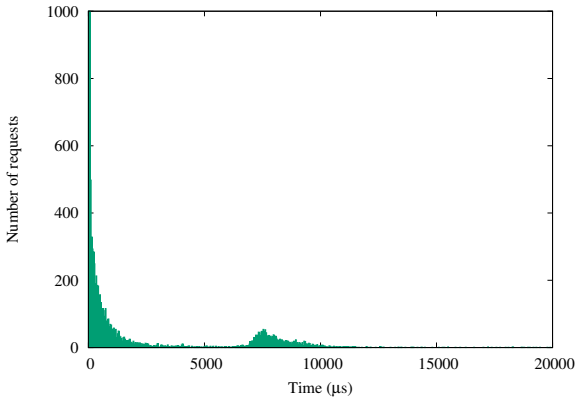
Table 3: Mean, standard deviation, and percentiles of times (in μs) spent in the middleware for the chosen optimal configuration of 200 virtual clients and 16 threads.



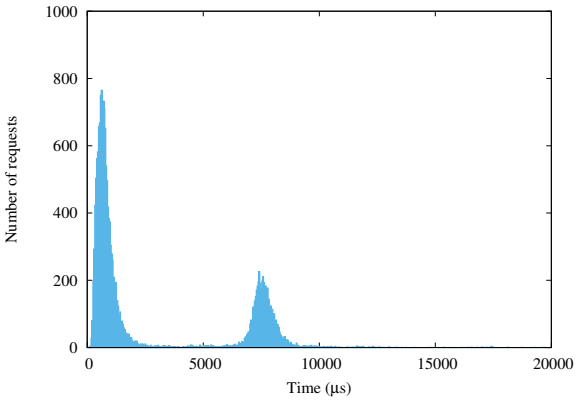
(a) Sorted timings for percentiles.



(b) Empirical distribution of **total** time T_{tot}.



(c) Empirical distribution of **queue** time T_{que}.



(d) Empirical distribution of **server** time T_{srv}.

Figure 4: Breakdown of time spent in the middleware.

Analysis. From the means in Table 3, it is already clear that the time spent by the requests in the servers dominates the total time. Standard deviation in this case results to be a poor measure, but looking at the percentiles, up to the 80th the difference between T_{que} and $T_{\text{tot}}, T_{\text{srv}}$ is still hugely marked. At higher percentiles this is not true anymore, as the time spent in the queue seems to equal that spent in the server. A reason for this, might be that requests taking longer times in the server block other incoming requests in the queues.

It is interesting to note that the times spent by the requests can be roughly grouped into two distinct intervals, centered around about 1 000 and 7 500 μs. This is clearly seen in Figures 4b and 4d for T_{tot} and T_{srv}, respectively, where two net spikes can be observed. Note that the spike is much less marked for T_{que} in Figure 4c, in support of the hypothesis that T_{srv} dominates. Also note that Figure 4c suggests that request usually spend very short times in the queues, since a peak around zero is observed. The second much smaller peak around 7 500 μs is can be explained by the same reasoning outlined above. ♢

2 Effect of Replication

The aim of this experiment is to explore how the behavior of the system changes for a 5%-write workload with $S = 3, 5$ and 7 server back-ends and replication factors $R = 1$ (no replication), $R = \lceil S/2 \rceil$ (replicate to half), and $R = S$ (write to all). The impact on both `SET` and `GET` operations is analyzed separately.

2.1 Experimental Setup

The setup of the experiment is outlined in Table 4. Based on the results of the previous experiment, 70 virtual clients for each of the three client machine have been chosen, so that the effective number of virtual clients connecting to the middleware is close[2] to the approximative optimum of 200. Also the optimal value of 16 threads has been adopted. The important parameters varied throughout the experiment are marked in bold. The experiment is carried out by the script located at https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/experiments/04_replication_effect, which collects all the `memaslap` and middleware outputs (`*.log`, inclusive repetitions).

Servers	$(\boldsymbol{S = 3, 5, 7})$ × A2
Threads / server machine	1
Clients	3 × A2
Virtual clients / machine	70
Keys size	16 Bytes
Values size	128 Bytes
Writes	5%
Overwrite proportion	90%
Statistics	1 Second
Middleware	1 × A4
Pool threads	16
Replication	$\boldsymbol{1, \lceil S/2 \rceil, S}$
Runtime × repetitions	50 Seconds × 4
Log files	repl-eff

Table 4: Replication effect experiment setup.

2.2 Analyses

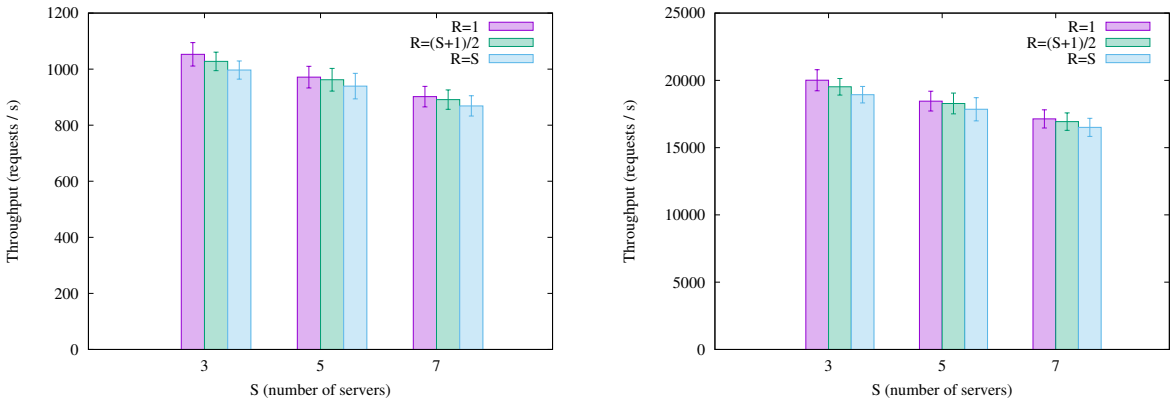
This section presents the results of some analysis carried out on the log files generated during the experiment. First, the log files of the `memaslap` clients are analyzed, and the throughput of the two operations `GET` and `SET` are compared for the different setups. Then a more detailed analysis of the time spent inside the middleware by the requests is carried out from the middleware's logs.

2.2.1 Impact on `SET`s and `GET`s

In order to investigate how the throughput of the `SET` and `GET` operations is affected by the changes in the configuration, the output of the `memaslap` clients has been first analyzed with the script located at https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/analyses/04_clients. For each of the 9 different configurations, the *mean* $\bar{\mu}$ and

[2] The script located at https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/analyses/04_5x40_vs_3x70 tests whether 5 physical clients with 40 virtual clients each yield similar throughput as 3 physical clients with 70 virtual clients each; this is confirmed by the logs from 5vs3 which when fed to the script report very similar mean for the throughput in the stable phase: 15 135.4 for 5 clients and 15 002.1 for 3.

standard deviation $\bar{\sigma}$ of the throughput has been calculated from samples 10 through 40 (in order to avoid warm-up and cool-down phases) of the clients' logs according to (1). The results of the analysis are summarized in Figure 5, which shows a comparison of the effect of the configuration of `SET` and `GET` operations. Note that in order to better visualize the data, the scale of the throughput axis is different for the two graphs.



(a) Effect of replication on `SET`.

(b) Effect of replication on `GET`.

Figure 5: Analysis of `memaslap` logs for replication effect experiment.

Hypothesis 5. *Increasing the replication factor should worsen the throughput of* ***`SET`*** *requests, but let more or less unaltered that of* ***`GET`*** *requests.*

Analysis. Figure 5 reveals that indeed there is a loss in performance in terms of throughput for the `SET` requests as the replication factor increases, but the hypothesis must be partially rejected, since it is clear that `GET` requests are affected as well. Moreover, comparing Figure 5a and Figure 5b it is evident that the relative decrease is the same for the two operations. This fact is also is supported by Table 5, computed using the script located at `https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/analyses/04_decrease`, which summarizes the performance loss[3] as a function of the increase of the replication factor for each configuration from 1 to $\lceil S/2 \rceil$ and S. The differences between `SET`s and `GET`s are outlined in bold, and it is striking to note that the largest is of only 0.02%.

But there is an obvious reason for this partially unexpected behavior: `GET` requests are affected too because `memaslap`, by design, first sends `SET` requests for a particular value, and then sends `GET` requests on that value only once it receives the notification that is was stored. This clearly artificially reduces the performance of `GET` requests too. ♢

	S=3		S=5		S=7	
	$\lceil S/2 \rceil$	S	$\lceil S/2 \rceil$	S	$\lceil S/2 \rceil$	S
`SET`	2.40%	5.34%	0.97%	3.29%	1.18%	3.67%
`GET`	2.42%	5.36%	0.95%	3.28%	1.18%	3.67%

Table 5: Decrease of throughput from $R = 1$.

2.2.2 Most Expensive Operation

In this section the length taken by requests inside the middleware as the replication factor grows is investigated. Since from the previous analysis it emerged that `SET`s and `GET`s behave similarly,

[3] If a is the base throughput and b the target throughput for which we want to compute the loss x from a, then $a - x \cdot a = b$, and so the values in the table are computed according to $x = 1 - b/a$.

and also in order to keep the analysis contained, both operations are considered together. The information is extracted from the logs of the middleware. Referring to (2), we introduce the timing

$$T_{\text{pro}} \doteq T_{\text{tot}} - (T_{\text{que}} + T_{\text{srv}})$$

which corresponds to the time taken by the middleware to *process* a specific request. Because of how the timestamps are taken in the middleware, as defined T_{pro} is exactly the time from when the request is received to when it is put into a queue. Therefore it accounts for the time necessary to *interpret* (distinguish between `SET` and `GET` requests), *hash* (determine the right server), and *address the request* (put it on the correct queue jointly defined by the operation and the hash value).

In this section the most expensive operation inside the middleware, as the replication factor increases, is sought. The operations (or stages) considered are those corresponding to the times T_{pro}, T_{que}, and, T_{srv}, which if summed up, correspond to the total time taken by a request inside the middleware. The script located at https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/analyses/04_breakdown generates means, standard deviation, and percentiles of those times, summarized in Table 6, and the three plots in Figure 6 which visualize the average time spent in each of the three stages.

					Percentile				
S	R	Time	$\bar{\mu}$	$\bar{\sigma}$	50th	80th	90th	95th	99th
3	1	T_{pro}	39.02	304.36	15	20	25	37	219
		T_{que}	2 818.06	3 012.74	1 764	4 944	7 362	8 989	12 248
		T_{srv}	1 934.81	2 020.09	1 187	2 691	4 446	6 731	8 835
	$\lceil S/2 \rceil$	T_{pro}	49.25	434.81	15	20	24	39	466
		T_{que}	2 621.82	2 922.25	1 559	4 613	7 117	8 743	11 653
		T_{srv}	1 922.03	1 958.81	1 175	2 636	4 496	6 782	8 617
	S	T_{pro}	46.69	405.06	15	20	24	37	322
		T_{que}	2 548.80	2 916.73	1 457	4 453	7 185	8 787	11 803
		T_{srv}	1 938.52	1 977.59	1 186	2 655	4 538	6 863	8 736
5	1	T_{pro}	42.44	364.10	16	20	25	38	322
		T_{que}	1 044.94	2 396.89	331	1 438	2 971	4 967	8 584
		T_{srv}	2 144.32	2 632.02	1 302	3 253	5 193	7 003	9 105
	$\lceil S/2 \rceil$	T_{pro}	43.90	396.77	16	20	25	38	315
		T_{que}	1 138.52	2 241.91	315	1 556	3 469	5 754	9 261
		T_{srv}	2 214.56	2 278.32	1 310	3 343	5 677	7 290	9 909
	S	T_{pro}	43.88	361.35	16	20	25	37	329
		T_{que}	1 124.60	2 020.97	306	1 538	3 398	5 875	9 027
		T_{srv}	2 303.63	2 302.26	1 375	3 569	5 929	7 378	9 754
7	1	T_{pro}	46.17	352.89	16	21	26	39	435
		T_{que}	536.37	1 270.64	60	671	1 349	2 506	6 749
		T_{srv}	2 042.30	2 181.86	1 137	3 086	5 426	7 127	9 364
	$\lceil S/2 \rceil$	T_{pro}	41.62	320.87	16	21	25	37	365
		T_{que}	617.66	1 402.43	92	764	1 602	3 051	7 369
		T_{srv}	2 193.53	2 296.09	1 208	3 536	6 037	7 310	9 493
	S	T_{pro}	49.27	415.18	16	21	26	40	398
		T_{que}	660.74	1 430.30	119	825	1 671	3 243	7 460
		T_{srv}	2 313.83	2 335.81	1 357	3 646	5 954	7 397	10 125

Table 6: Mean, standard deviation, and percentiles of times in the middleware in μs.

Hypothesis 6. *As the number of servers increases the load of each of them should decrease, and therefore also the times spent by requests in the middleware should decrease. But as replication is increased, for a fixed number of servers, an increase on the time spent in the middleware should be observed, specifically in the queue and in the server times.*

Analysis. From Table 6 it is clear that the time needed to process each incoming request, T_{pro}, stays almost always the same. In fact its means for the various configuration do not vary too much. But they appear to slightly increase for fixed number of servers as the replication factor is increased, but this is not always true (for $S = 3$ from $R = \lceil S/2 \rceil$ to $R = S$, $S = 5$ from $R = \lceil S/2 \rceil$ to $R = S$, and $S = 7$ from $R = 1$ to $R = \lceil S/2 \rceil$, they decrease). Looking at the percentiles, it emerges that for the 50th and 80th the maximal difference between any configuration is of only $1\,\mu s$, and for 90th and 95th of only $3\mu s$. It is a bit larger for the 99th percentile, with differences of up to $200\,\mu s$, but fixing the number of servers and varying the replication factor it is not true that it also increases (rather, it both increases and decreases). Therefore, in line with what intuition would also suggest, T_{pro} is *not* affected by the increase in the replication factor.

For T_{que} and T_{srv} it emerges from Table 6 and Figure 6 that for $S = 3$, on average, requests take more time in the in the queues than in the servers, while the opposite holds for $S = 5, 7$. This is explainable by the fact that with few servers, each of them receives more load, as hypothesised. This translates into having more requests waiting in the queues on average.

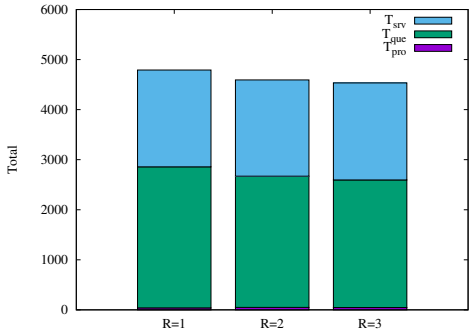
From Table 6, it also results that with $S = 3$, the time spent in the queue tends to slightly decrease as the replication factor is increased. This is evident by looking at the mean values of T_{que}, and it is also partly confirmed by the percentiles: for the 50th and the 80th this is true, while for the 90th, 95th, and 99th, there is a net decrease from $R = 1$ to $R = \lceil S/2 \rceil$, but for $R = \lceil S/2 \rceil$ and $R = S$ the values increase a bit (while still being lower for $R = \lceil S/2 \rceil$ than for $R = 1$). On the other hand, for $S = 5, 7$ the opposite holds: T_{que} appears to increase as the replication factor is increased. The means support this fact clearly, and for $S = 7$ *all* the percentiles confirm this. For $S = 5$ instead, full replication ($R = S$) seems to impact the queue time less than half replication ($R = \lceil S/2 \rceil$), but again full replication affects the queue more than no replication.

For T_{srv}, note that for $S = 3$ its means are all about the same for the three different replication factors, while for $S = 5, 7$ they increase as R increases. The percentiles of T_{srv} also speak in favor of the server time being unaffected by replication for $S = 3$, probably because of the high load to which the servers already undergo. For $S = 5, 7$, the percentiles indicate that the effect of replication is strong when at half, but from half to full the effect is attenuated; in fact some percentiles show lower values for full replication than for half, contrary to what the means suggest.

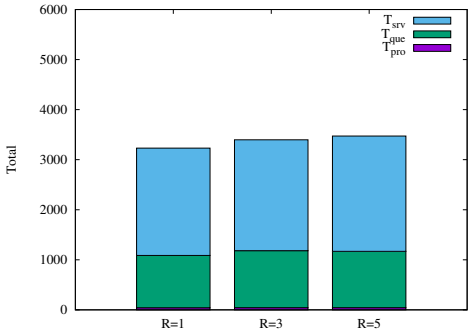
All in all, it emerges that the effect of replication on the various operations inside the middleware is strongly affected by the number of servers. For few servers ($S \approx 3$), it results that higher replication factors positively affect the efficiency of the middleware. On the contrary, for larger numbers of servers, the requests spend more time in the servers, as replication increases, while queue times are just slightly affected (and this as a consequence of the increase in server time). Therefore, it can be concluded that the operation most affected by the increase of the replication factor, is server time. This is also in line with the fact that internally, for `SET` requests the middleware considers the server time as the time from when the request is sent to the primary server to when the last response arrived from *all* the replication servers. ♢

2.2.3 Scalability

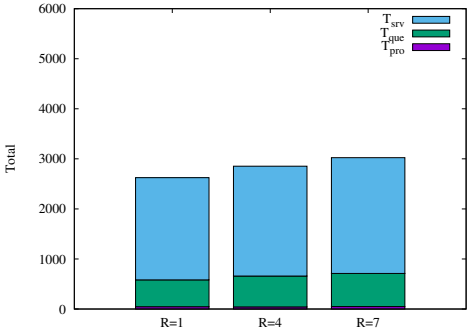
In this section the scalability of the system is compared to that of an ideal implementation. The latter is supposed to be a *perfect load balancer*: assuming for a moment no replication ($R = 1$), it is expected that if a total of x requests are generated by the clients, each of the S servers receives $\frac{x}{S}$ of them, on average. Taking into account replication is more involved, as some more sophisticated assumption on the ideal behavior of the writer thread would have to be made. But to keep things simple, here it is only assumed that the writer thread behaves perfectly asynchronously (which is not totally true, since the use of a timeout, as detailedly explained



(a) Middleware average times for $S = 3$ servers.



(b) Middleware average times for $S = 5$ servers.



(c) Middleware average times for $S = 7$ servers.

Figure 6: Analysis of `memaslap` logs for replication effect experiment.

in the first report, only guarantees an "artificial" asynchronous mechanism). Nevertheless, it is still assumed that `SET` request are sent serially to all replication servers (so in particular, the asynchronous behavior only manifests explicitly when reading responses from the server).

With these considerations in mind, it can be supposed that the load of each of the S servers, for any replication factor $1 \leq R \leq S$, and writes proportion $W \in [0, 1]$ is

$$\frac{WR \cdot x + (1 - W) \cdot x}{S} = \frac{x}{S} + W\frac{R - 1}{S} \cdot x. \tag{3}$$

Therefore it is clear that as the number of servers S is increased, the load per server diminishes, but having a replication factor R larger than 1 introduces an overhead of proportion $W\frac{R-1}{S}$ of the number x of requests on the response time (which clearly is 0 when $R = 1$).

Hypothesis 7. *Since the load for each server diminishes by increasing the number of servers, according to (3), the throughput of the system should ideally also increase. Moreover, still according to (3), a decrease should be observed when the replication factor is increased.*

Analysis. The first part of the hypothesis can be quickly rejected by looking at Figure 5. It is clear that, independently of the replication factor, the throughput decreases as S increases. This is explainable by taking into consideration, that in reality having more servers introduces considerable complexity, both in terms of network connections and needed resources: the number of parallel connections grows linearly with the number of servers for fixed replication factor, and since bandwidth is finite, a saturation point is eventually reached; also, having more servers means more threads in the middleware, and each thread takes a considerable portion of CPU time, which is also finite.

Instead, the second part of the hypothesis is clearly in line with the observations. ♢

3 Effect of Writes

The aim of this experiment is to study the changes in throughput and response time of the system as the percentage of write operations increases from 1% through 5% to 10%.

3.1 Experimental Setup

The setup of the experiment is outlined in Table 7. Again the optimal configuration of 200 virtual clients and 16 threads found in Section 1 has been adopted. The important parameters varied throughout the experiment are marked in bold. The experiment is carried out by the script located at `https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/experiments/05_writes_effect`, which collects all the `memaslap` and middleware outputs (`*.log`, inclusive repetitions).

Servers	$(\boldsymbol{S = 3, 5, 7})$ × A2
Threads / server machine	1
Clients	3 × A2
Virtual clients / machine	70
Keys size	16 Bytes
Values size	128 Bytes
Writes	**1%, 5%, 10%**
Overwrite proportion	90%
Statistics	1 Second
Middleware	1 × A4
Pool threads	16
Replication	**1, $\boldsymbol{S}$**
Runtime × repetitions	50 Seconds × 4
Log files	`wrt-eff`

Table 7: Writes effect experiment setup.

3.2 Analyses

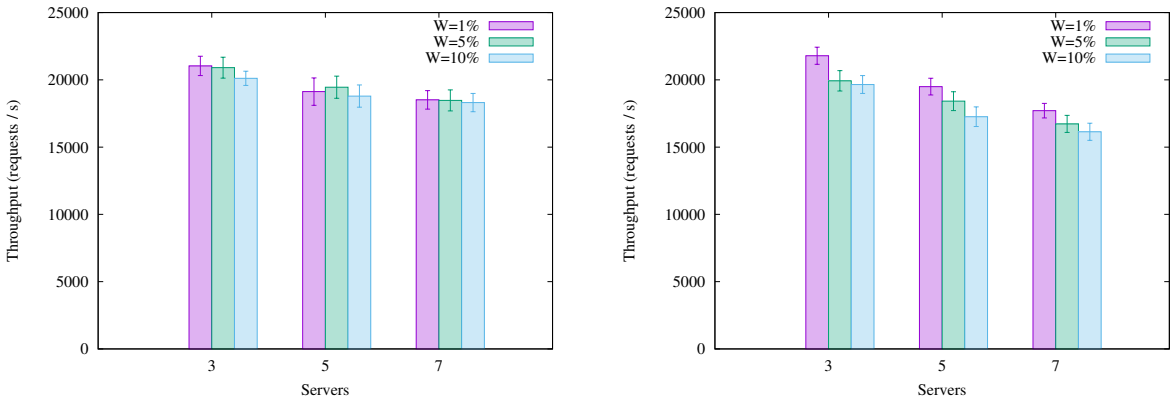
This section presents the results of some analysis carried out on the log files generated during the experiment.

3.2.1 Performance

In this analysis, the changes in performance in terms of throughput and response time from the base case of 1% write proportion has been investigated. The plots in Figure 7 are generated by the script located at `https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/analyses/05_clients`, and they summarize the values of average throughput and response time for the various configurations. Again, the standard deviation of the throughput has been computed according to 1, while for response time the one provided by `memaslap` has been used. Finally, the relative changes as the write proportion is increased are summarized in Table 8, and the values are generated by the script located at `https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/analyses/05_decrease`.

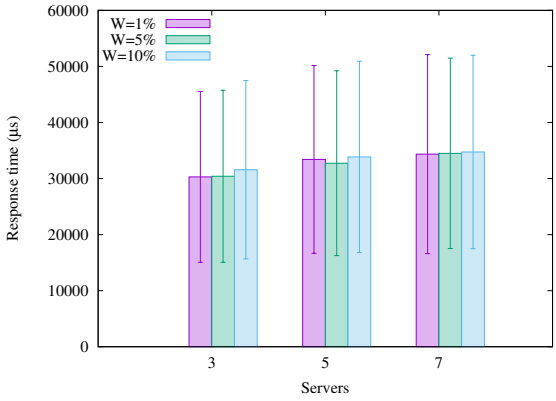
Hypothesis 8. *As the proportion of writes increases, with no replication no substantial reduction of performance should be observed by increasing the number of servers, while with full replication this should be true instead.*

Analysis. From Figures 7a and 7c, it can be seen that indeed the performance, as the write proportion is increased with no replication, is more or less the same for any number of servers.

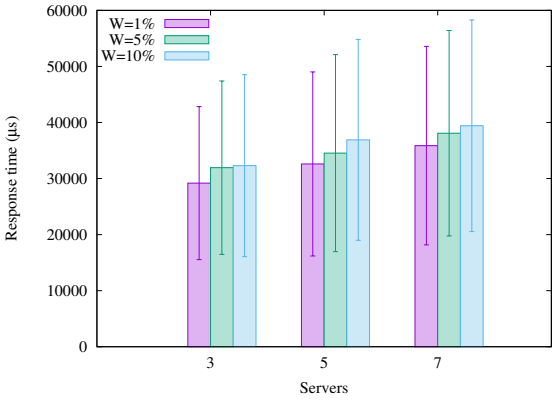


(a) Throughput for $R = 1$.

(b) Throughput for $R = S$.



(c) Response time for $R = 1$.



(d) Response time for $R = S$.

Figure 7: Analysis of `memaslap` logs for writes effect experiment.

The relative changesread from Table 8 confirm this. In fact the values on the first and the third rows (for $R = 1$) are relatively small (compared to the other two rows), and sometimes they are even negative (this counterintuitive result is probably due to the unreliability of the network). Anyway, still for $R = 1$, the biggest relative impact is registered with 3 servers.

In case of full replication ($R = S$), in line with the hypothesis, the performance worsen for all number of servers as the write proportion increases. This is evident from Figures 7b and 7d and the number in the second and fourth rows of Table 8 also confirm this. From these values it can also be observed that the impact is the largest for all number of servers when changing the writes from 1% to 10%, but the largest impact when changing the writes from 1% to 5% is registered with 3 servers (9.44% vs. 5.93% and 6.18%).

In summary, considering $R = S$, the biggest impact on performance relative to base case of 1% writes proportion is seen with $S = 5$ servers, the values confirming this are marked in bold in Table 8, and are to be compared with the corresponding values on the other two columns. ♢

		***S*=3**		***S*=5**		***S*=7**	
		5%	**10%**	**5%**	**10%**	**5%**	**10%**
Throughput	***R*=1**	0.62%	4.38%	-1.70%	1.75%	0.22%	1.10%
	R*=*S	8.56%	9.81%	5.54%	**11.47%**	5.55%	8.85%
Response time	***R*=1**	0.39%	4.21%	-2.00%	1.34%	0.41%	1.13%
	R*=*S	9.44%	10.70%	5.93%	**13.17%**	6.18%	9.89%

Table 8: Decrease of performance from write proportion 1%.

3.2.2 Behavior of the System

In this section the behavior of the system is investigated with the help of the logs from the middleware. Since it was concluded in the previous section that the effect of writes is the starkest for full replication with 5 servers for a 10% writes proportion, only the corresponding logs are analyzed. Full replication was also selected because it represents a more "realistic" scenario. All the table values and plots are generated by the script located at https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/analyses/05_breakdown. Since the stable phase is assumed to lie in the middle three-fifths of the runtime of the clients, only the middle three-fifths of the `SET` and `GET` entries is taken. (Note that the entries are sampled every 100 for both `SET` and `GET`.) Data for `SET`s and `GET`s is aggregated and presented is exactly as described in Section 1.5.

					Percentile				
W	**Op.**	**Times**	**Mean**	**Std. Dev.**	**50th**	**80th**	**90th**	**95th**	**99th**
1%	`SET`	**Total** (T_{tot})	6 662.51	3 017.97	6 812	8 975	9 845	10 650	13 019
		Queue (T_{que})	274.73	628.65	32	381	542	665	1 144
		Server (T_{srv})	6 321.36	2 933.82	6 256	8 435	9 295	10 235	12 530
	`GET`	**Total** (T_{tot})	2 928.73	2 909.80	1 673	5 535	7 540	8 802	11 400
		Queue (T_{que})	949.20	1 863.64	239	1 196	2 506	5 880	8 583
		Server (T_{srv})	1 933.32	2 183.90	1 041	2 466	6 229	7 010	8 485
10%	`SET`	**Total** (T_{tot})	6 066.56	3 636.92	5 951	9 067	10 180	11 678	15 608
		Queue (T_{que})	217.79	733.25	24	208	503	792	2 115
		Server (T_{srv})	5 773.66	3 464.64	5 643	8 748	9 887	11 391	14 368
	`GET`	**Total** (T_{tot})	3 288.79	3 326.61	1 837	6 375	8 013	9 538	13 140
		Queue (T_{que})	1 008.37	2 089.87	192	1 208	2 952	6 161	9 882
		Server (T_{srv})	2 222.08	2 479.35	1 137	3 295	6 633	7 499	9 813

Table 9: Mean, standard deviation, and percentiles of times (in μs) spent in the middleware by `SET` and `GET` requests $S = R = 7$ and $W = 1\%, 10\%$ writes proportion.

Hypothesis 9. *Based on the result of the previous analysis on the logs from* `memaslap`*, it should be possible to confirm that changing the writes proportion from* 1% *to* 10% *for* $S = R = 5$ *results in a significantly worst performance.*

Analysis. First of all, note that the empirical distributions in Figures 8b-8c and 9b-9c look less "nice" because by design the middleware logs one every 100 requests, separately for `SET` and `GET`, and therefore, since there are much less writes than reads, there are less samples for `SET`s to use to visualize the empirical distribution.

Confronting the sub-figures of Figure 8 with the corresponding sub-figures of Figure 9, it immediately seems that for the two configurations, the middleware actually behaves quite similarly, almost in contradiction with the hypothesis. In fact, the distributions of the three timings for `GET` requests look extremely close, and even the two typical spikes appear to be at the same spot. The distributions for `SET` requests unfortunately do not allow to say much, because they are likely to be imprecise due to the small sample population, but it seems as if the distribution is more uniform than for `GET`s.

Looking at the means in Table 9, it results that T_{tot} for `SET`s with $W = 1\%$ is slightly larger than with $W = 10\%$ (6 662.51 μs vs. 6 066.56 μs), while the converse it true for `GET`s (2 928.56 μs vs. 3 288.79 μs). This might indicate the presence of a possible threshold, situated somewhere after $W = 10\%$, at which write operation start becoming more expensive than read operations on average. Note that even though T_{tot} for `SET`s decrease, apparently against the hypothesis, the increase of T_{tot} for `GET` is large enough so support the thesis that overall response time is larger for $W = 10\%$. In fact, a rough estimate of the mean response time for $W = 1\%$ gives

$$1\% \cdot 6\,662.51\,\mu s + 99\% \cdot 2\,928.73\,\mu s \approx 2\,966.07\,\mu s,$$

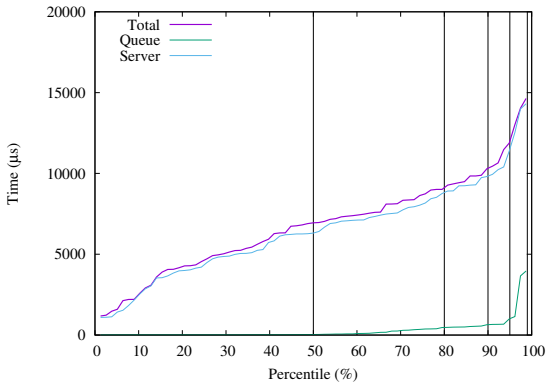
while for $W = 10\%$ it gives

$$10\% \cdot 6\,066.56\,\mu s + 90\% \cdot 3\,288.79\,\mu s \approx 3\,566.57\,\mu s,$$

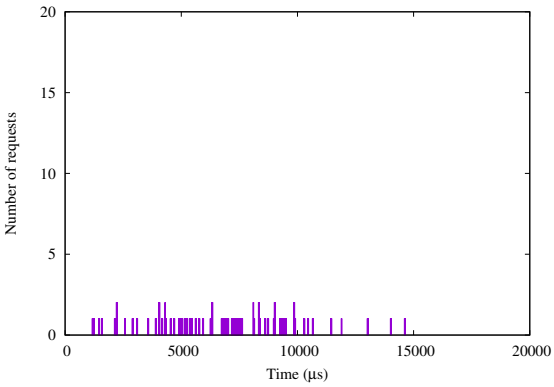
in line with the hypothesis that performance decreases in average as W is increased.

To better understand the situation, it is necessary to turn the attention to percentiles. Looking at Table 9 it emerges that T_{tot}'s median (50th percentile) for `SET` requests is higher with $W = 1\%$, like for its mean. But already starting from the 80th percentile, this is not true anymore: in fact for higher percentiles, T_{tot} of `SET`s is always higher for $W = 10\%$, as hypotesized. On the other hand, for T_{tot} of `GET`s, all the percentiles follow the behavior of the means, as they are all higher for $W = 10\%$, as expected.

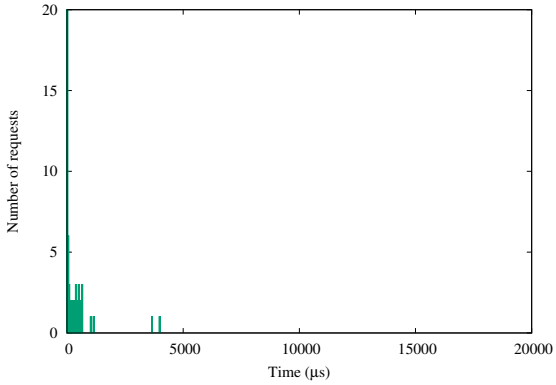
Comparing Figures 8a and 9a, it is visible that for $W = 10\%$ there are some exceptional outliers which take more time than the longest `SET` requests for $W = 1$. Also Table 9 speaks in favor of this fact: the respective 99th percentiles have a $\sim 2\,600\mu s$ difference, and for smaller percentiles the difference is considerably lower. Therefore, the reduced performance when W is increased is probably due to the increase on outliers in `SET` requests, which in turn slow down also `GET`s. This might be explained by the fact that increasing the number of write operations, with full replication it is more and more probable that some responses are slower than other, and since $R = S$ implies that `SET`s take as long as a response from all replication servers is perceived, when a response takes exceptionally long to arrive it affects significantly the overall performance of the middleware on average. ♢



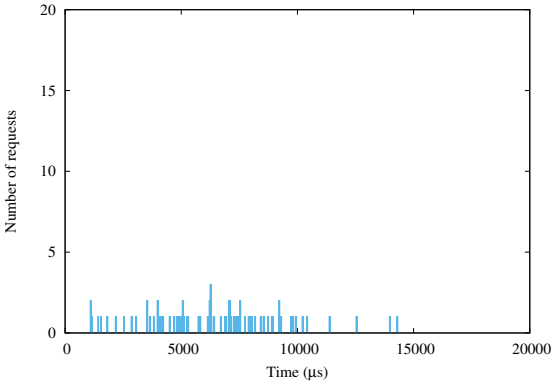
(a) Sorted timings for percentiles of `SET`s.



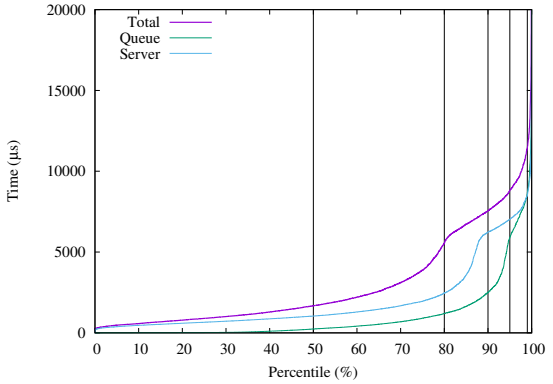
(b) Distribution of **total** time T_{tot} for `SET`s.



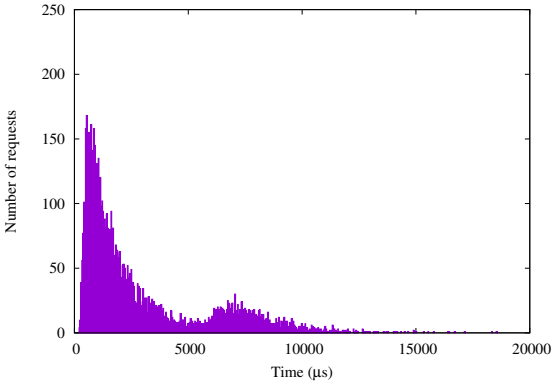
(c) Distribution of **queue** time T_{que} for `SET`s.



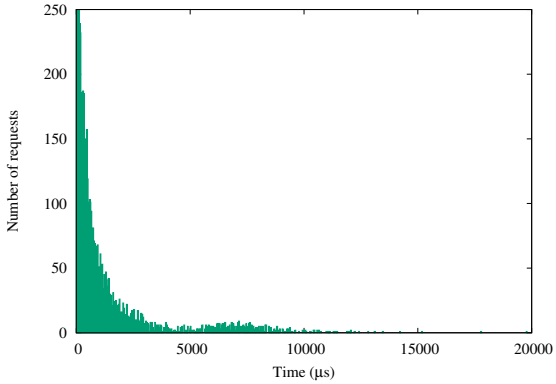
(d) Distribution of **server** time T_{srv} for `SET`s.



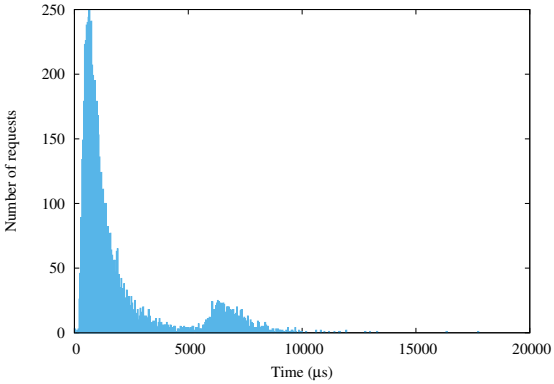
(e) Sorted timings for percentiles of `GET`s.



(f) Distribution of **total** time T_{tot} for `GET`s.

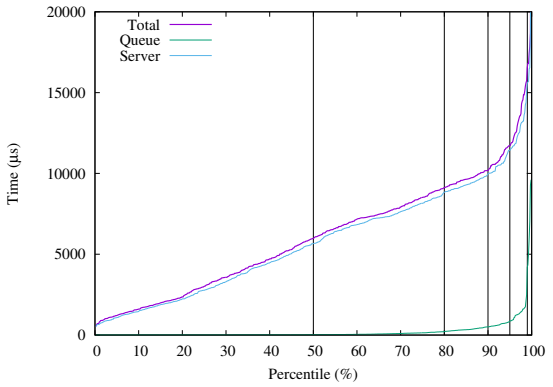


(g) Distribution of **queue** time T_{que} for `GET`s.

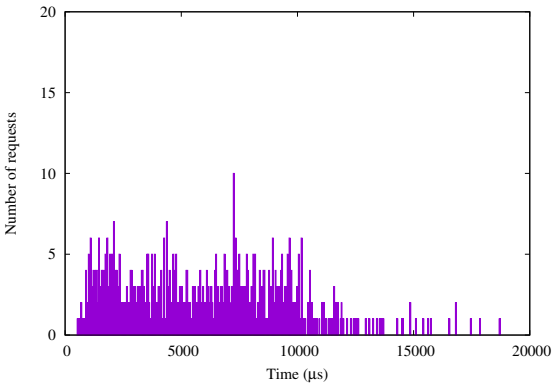


(h) Distribution of **server** time T_{srv} for `GET`s.

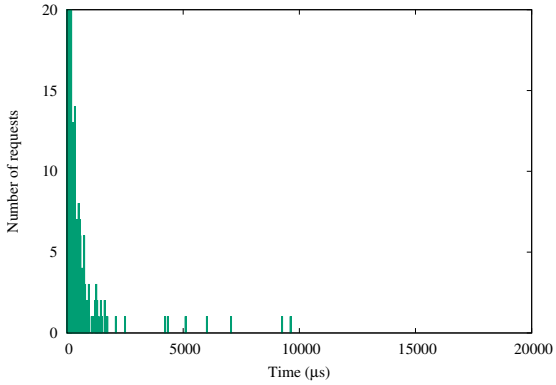
Figure 8: Breakdown of time spent in the middleware for 1% writes proportion.



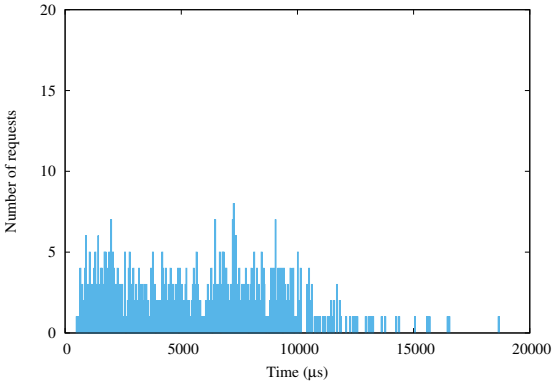
(a) Sorted timings for percentiles of `SET`s.



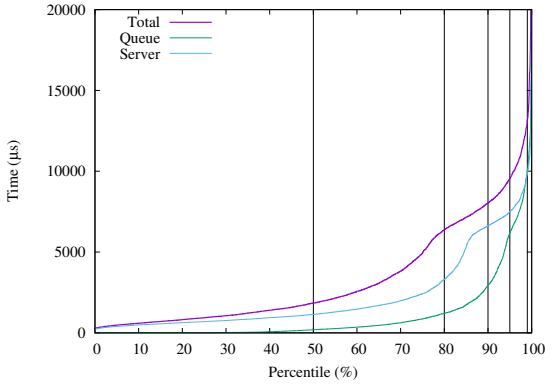
(b) Distribution of **total** time T_{tot} for `SET`s.



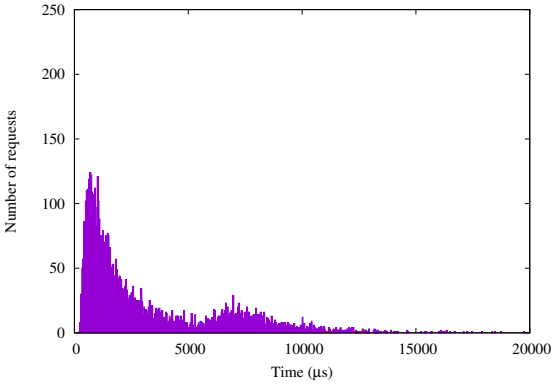
(c) Distribution of **queue** time T_{que} for `SET`s.



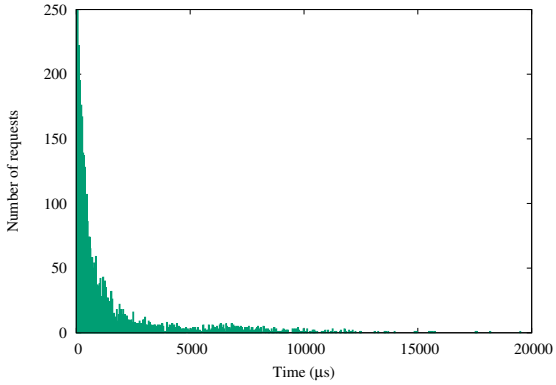
(d) Distribution of **server** time T_{srv} for `SET`s.



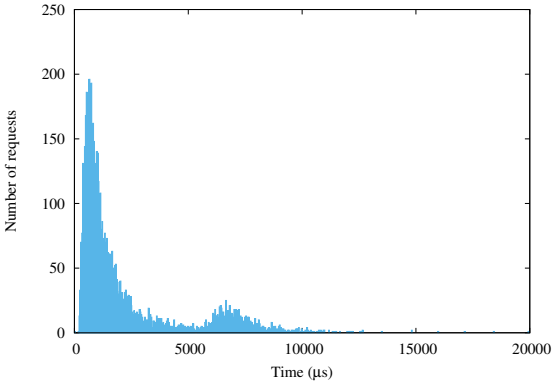
(e) Sorted timings for percentiles of `GET`s.



(f) Distribution of **total** time T_{tot} for `GET`s.



(g) Distribution of **queue** time T_{que} for `GET`s.



(h) Distribution of **server** time T_{srv} for `GET`s.

Figure 9: Breakdown of time spent in the middleware for 10% writes proportion.

Appendices

A Modification of the Middleware Source Code

After some considerations, a slight modification to the source code of the middleware has been performed. As outlined in the first report, the writer thread has been designed with the idea of avoiding *busy waiting*. The reason behind this, was that en empiric loss of performance in terms of throughput was observed, when adopting busy waiting. Non-busy waiting is usually achieved by blocking the thread, but due to the inherent multiplicity of the points at which the writer thread should have been blocked (by queue polling *and* by server data reading), a full blocking solution was avoided due to the expected high increase in complexity of the code. Instead, a more simple solution has been adopted: to block when polling the queue, but only for a certain amount of time. Using Java's `BlockingQueue`, this is achieved by invoking the method `queue.poll(long, java.util.concurrent.TimeUnit)`. Concretely, the timeout was set to 10 milliseconds. But this was problematic, because from the middleware logs, it emerged that requests spend much smaller amounts of time in some parts of the middleware. Empirically, a lower-bound of 5 microseconds has been observed for the time a request spends in a queue. Thus, such a large timeout would necessarily influence the overall behavior of the middleware introducing considerable complexity in a future formal analysis of its model. Since timing information in the logs are always rounded to microseconds, the timeout has been updated to **100 nanoseconds**. This can be considered negligible w.r.t. the timings registered in the logs. Finally, a slight loss in performance in terms of throughput has been observed, but this was very small in comparison to the one observed if the writer thread was to use busy waiting. Figure 10 depicts the outcome of the stability experiment performed with the same script as for the first report, for a total of 10 minutes. This shows that the system is still stable.

The second modification to the source code was of more technical nature. When performing the maximum throughput experiment, it emerged that with many virtual clients connecting to the middleware (about 500 virtual clients) on the cloud, sometimes some of them reported errors by writing, even though on the local machine such errors did not appear. Skipping details, it was found, that this was due to the internal size of the socket's buffers. The solution was to add the following two lines after the creation of the sockets in the writer thread:

```
serverSockets[i].setOption(StandardSocketOptions.SO_RCVBUF, 2097152);
serverSockets[i].setOption(StandardSocketOptions.SO_SNDBUF, 2097152);
```

This would make socket's buffer's sizes *larger* on the cloud, and avoid the problem.

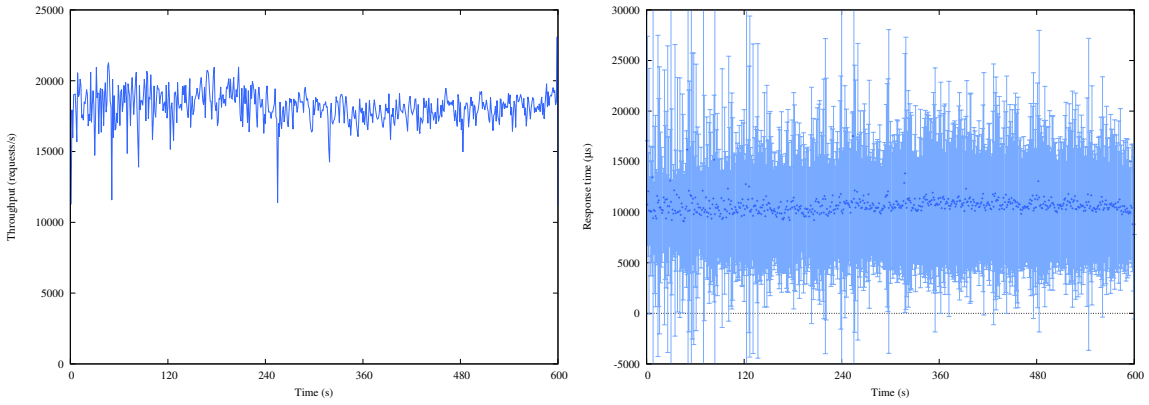


Figure 10: Small stability trace of the new code (`new-trace`).

B Logfile Listing

For each experiment, the logfiles (`*.log`) and the (eventual) 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 10 indicates the names of the ZIP files of each experiment. For the maximum throughput experiment, the data files `*.data` are reproducible from the logfiles using the script

https://gitlab.inf.ethz.ch/fbanfi/asl-fall16-project/blob/master/scripts/reproduce_data/03_maximum_throughput_rd

(PDF plots of the data will be generated as well). Note that the script has to be manually adjusted for each of the three runs.

Short name	Location
max-tps1	03_maximum_throughput_16-11-10_21:14:41.zip
max-tps2	03_maximum_throughput_16-11-11_21:56:33.zip
max-tps3	03_maximum_throughput_16-11-12_12:07:39.zip
repl-eff	04_replication_effect_16-11-20_13:16:14.zip
wrt-eff	05_writes_effect_16-11-20_09:37:46.zip
5vs3	5x40_vs_3x70.zip
new-trace	02_stability_trace_16-11-15_14:14:51.zip

Table 10: Location of referenced log and data files.