Advanced Systems Lab (Fall'16) – Second Milestone

Name: *Rami Khalil*
Legi number: *16-932-568*

Grading

Section	**Points**
1	
2	
3	
Total	

Preliminary

The information in this section should be kept in mind by the reader while while reading all experiments.

0.1 Statistical Reasoning Methods

In this report, extensive use of the Central Limit Theorem (CLT)[1] is made in order to gather and report throughput. We rely upon the idea of estimating the mean of the throughput in order to draw comparisons between different configurations of the system under test (SUT).

As a working example, consider the following two configurations of the SUT when dealing with a read-only workload and utilizing 5 key value stores:

1. 30 Connected Clients, 20 Reader Threads
2. 20 Connected Clients, 100 Reader Threads

If we proceed to sample the mean of the throughput of the system under these configurations 5 times, then calculate the mean for these samples (mean of the mean), then we end up with an estimate of where the true mean throughput lies. If we calculate that, for example, the means of the mean throughput are estimated to be as follows for the previous two configuration, then we can, to a certain extent, reason that the first configuration should be expected to provide higher throughput, on average, than the second one.

1. $\bar{tps}_{30,20} = 10040$, $\sigma_{30,20} = 290$
2. $\bar{tps}_{20,100} = 7870$, $\sigma_{20,100} = 160$

We forgo analysis using means and standard deviations on changes in response times since means give no indication of the service-level agreement that is to be expected from the system, and can lead to misrepresentation of the system behavior[2] since the response time distributions have a long tail. Our main metrics of interest are the times spent by 50%, 90% and 99% of requests made during an experiment of a configuration (percentiles).

0.2 Differences In Performance Between Similar Configurations

The experiments carried out in different sections (1, 2, 3) were done on different dates at different times of day. The Azure cloud platform is unreliable and the time at which an experiment was carried affects the performance attained. Any stark differences in performance between two experiments carried out in different sections using similar configurations are due to Azure's unreliability.

0.3 Experiment Infrastructure

The infrastructure for experimentation is composed of 10 machines on the Azure cloud platform, Up to 7 of which can be designated as memcached servers, utilizing the "Basic A2"[1] hardware configuration. Two machines are the workload generators, utilizing the same hardware. One machine, utilizing the "Basic A4"[2] hardware configuration, is designated as the middleware server, at which the load generators are targeted, and from which connections to the memcached servers are initiated. This hardware infrastructure is the same across all experiments, and only the number of used memcached servers is varied according to the experiment design.

[1] 2 Cores, 3.5GB RAM
[2] 8 Cores, 14GB RAM

0.4 Experiment Automation

To aid with the process of experimentation, multiple scripts were written to automate the procedures[3]. The bulk of the automation was in scheduling each experiment and safely storing its resulting data. In order to synchronize all machines to initiate the middleware and load generators at the same time, the *at* unix command was made use of to synchronize the timing of execution of the experimentation scripts.

Each memaslap experiment was run for 1 minute and was scheduled at the same time across all machines, but was given a 2 minute window to completely execute as to account for any preamble done by memaslap (read-only workloads). These 2 minutes are broken down as follows for each experiment: During the first 10 seconds, the middleware was given time to safely store all pre-existing log files, then start and connect to the memcached servers. The remaining 1 minute and 50 seconds were left for the memaslap clients to initialize, possibly send a preamble of set requests (read-only workload), then begin running the workload for 1 minute.

0.5 Logging

There are 3 logs produced by each experiment: The first two of which are the outputs of the memaslap clients, and the third is the output of the middleware. Logs for each experiment from each client machine and from the middleware were kept for presentation and analysis.

The data used to calculate throughput and mean response time is only aggregated from the middle 30 seconds of each experiment. Each experiment runs for 1 minute, however, we account for warmup and cooldown times of 15 seconds each. Therefore, for each log file produced, we ignore data generated in the 1st 15 seconds and the last 15 seconds of the experiment.

Throughput is calculated from the memaslap logs. The T_{Total} percentiles are calculated by interpolation from the aggergation of memaslap logs for each configuration. All other times are calculated from the middleware logs.

1 Maximum Throughput

1.1 Simplest Configuration for Maximum Throughput

1.1.1 Introduction

In this experiment we aim to find the minimum load required to achieve the maximum throughput of the system under test. In other words, we wish to find the parameters, in terms of clients and reader threads, that saturate the middleware using the hardware configuration. The behavior of the throughput versus clients and read threads is expected to be as follows:

1. For a fixed number of *reader threads*, the behavior of **mean throughput** versus number of clients is expected to increase at first until a point of saturation, whereby throughput remains relatively the same, until a point of decline, after which performance starts to degrade, as shown in figure 1

2. For a fixed number of *clients*, the behavior of the **mean throughput** versus number of reader threads is also expected to increase until a point of saturation. This point of saturation is expected to approximately be equal to the number of clients divided by the number of servers[4]. After this point of saturation, adding more threads should not significantly affect throughput, until a point of decline is reached whereby the middleware performance starts to degrade as more threads as added.

[3]automation/*

[4]This is due to constant hashing. Because the read-only workload is equally divided across all servers, when the total number of reader threads in the middleware is equal to the number of clients, then every client basically

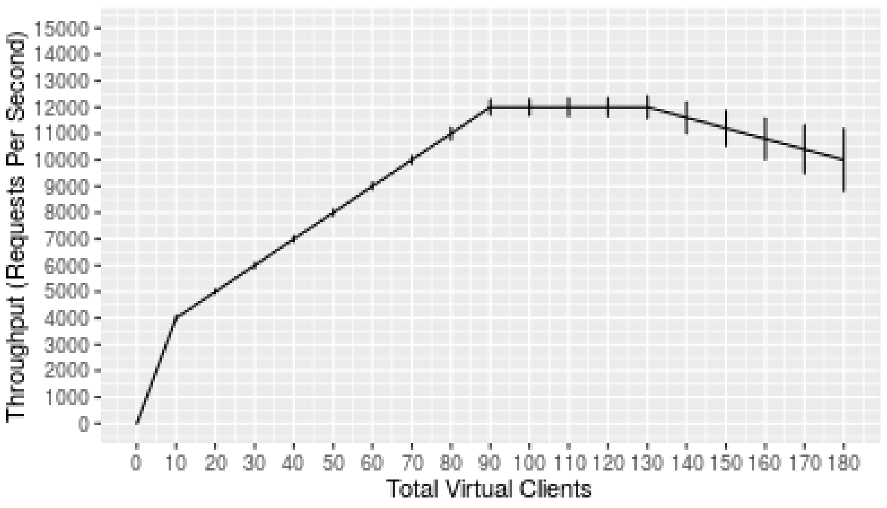


Figure 1: Expected behavior of throughput versus number of clients with constant reader thread count, with standard deviation.

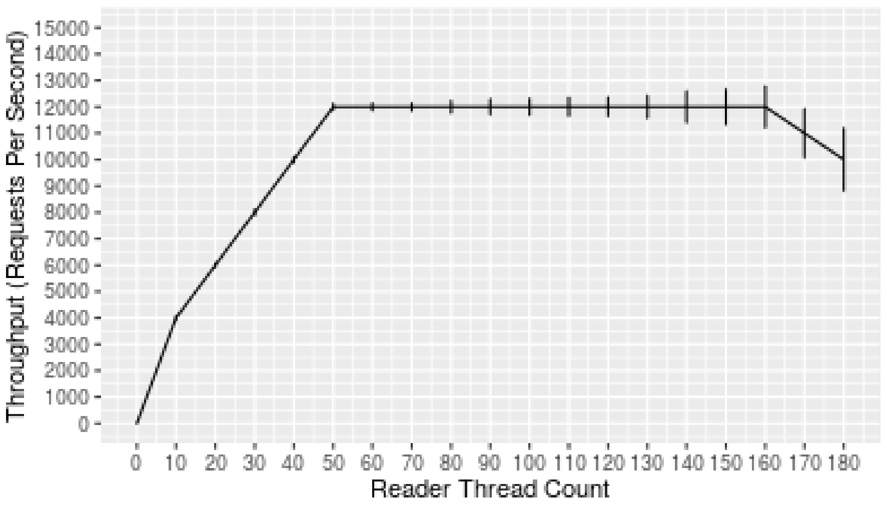


Figure 2: Expected behavior of throughput versus number of reader threads with constant number of clients, with standard deviation.

Where the aforementioned saturation points exactly lie will differ depending on the hardware resources available to run the middleware and the configuration of the middleware to utilize them.

has their own thread servicing their requests. Adding more reader threads beyond this point should have no effect beyond alleviating the low probability situations where certain servers are more overloaded than others.

1.1.2 Methods

Number of servers	5
Replication	R = 1
Number of client machines	2
Virtual clients / machine	5 to 100 (step of 5)
Reader Threads	10 to 100 (step of 10)
Workload	Key 16B, Value 128B, Read-Only
Window Size	1,000
Runtime x repetitions m	1m x 5
Log files	throughput/*
Design	Full-Factorial
Experiments	10*20*5 = 1000

The workload is mainly composed of read-only "GET" operations. The generators begin by pre-setting random values of 128 bytes each under 16 byte keys. Then they proceed to fetch these values throughout the remainder of each experiment.

1.1.3 Results

In this section, the results of the run experiments are presented. Log files and scripts used to generate the following plots are in [5].

The table/heatmap cell in figure 3 includes two entries: The first (top) entry is the calculated sample mean throughput for corresponding reader threads/virtual clients configuration; the bottom (second) entry is the standard deviation for the calculated sample mean. The colors of the scale and of each cell in the heatmap are completely unaffected by the value of the standard deviation, but only set according to the value of the sample mean. The sample sizes used to calculate the means and standard deviations are of size 5. Each sample is derived from a 30-second window of an independently run experiment, as explained in the prelude.

1.1.4 Analysis

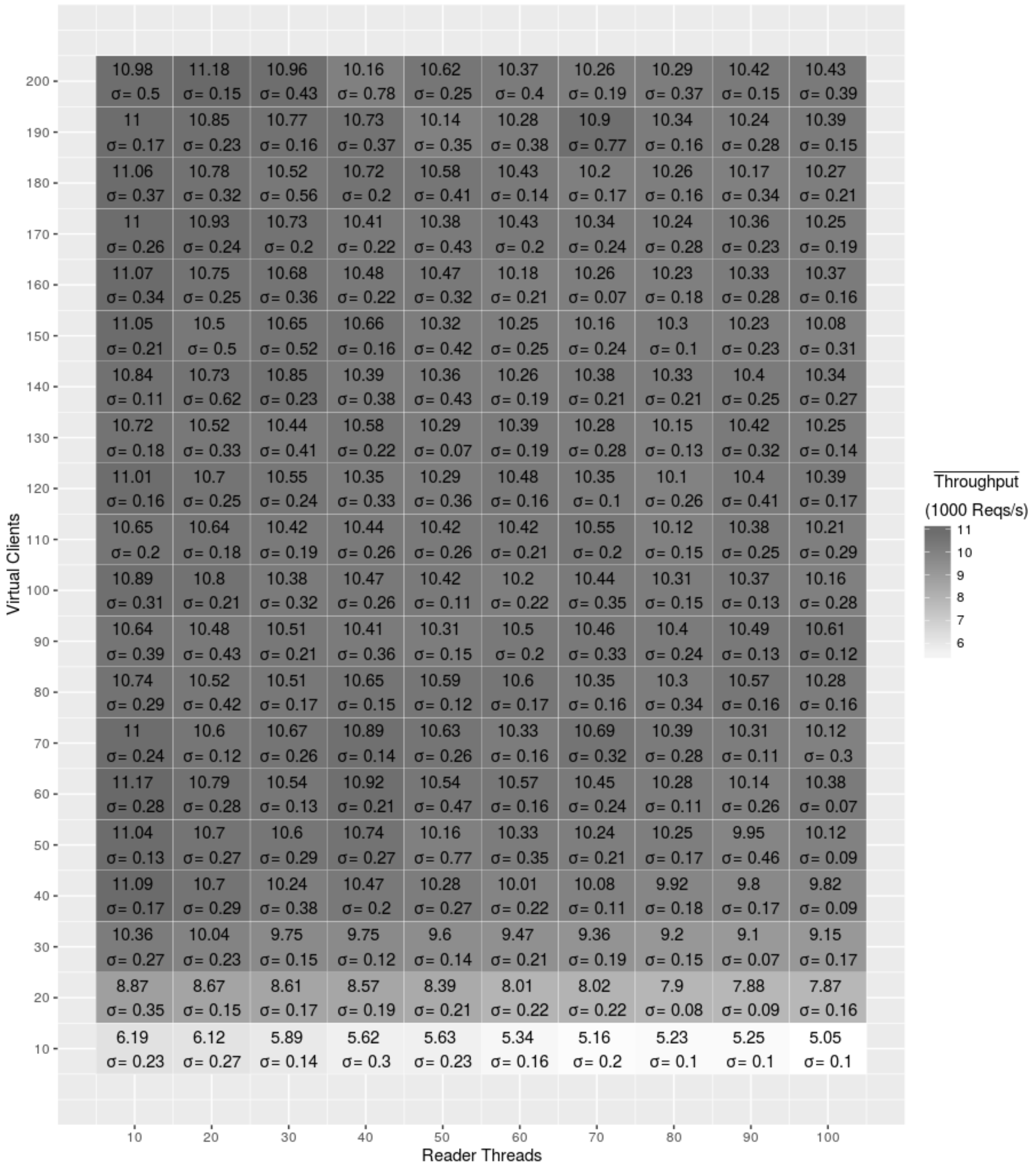
This analysis aims to either confirm or reject the expected behaviors aforementioned in the introduction subsection, while answering the question of what is the simplest configuration, minimally in terms of reader threads and virtual clients, that achieves maximum throughput on the SUT. We also attempt to explain why the system reaches its maximum throughput under that configuration and provide a detailed breakdown of the time spent by different operations within the middleware at that configuration.

Hypothesis #1: Mean Throughput under constant number of readers and varying number of clients

In order to affirm or reject this hypothesis, we should inspect the estimated throughput values across different number of readers configurations. Upon inspection of the data presented in in figure 3, it is clear that the outlined initial behavior is aparent. For every fixed number of reader threads (from 10 to 100), and as the number of clients is increased, the throughput increases from a low of about 5 thousand requests per second until it starts to saturate somewhere between 10 and 11 thousand and does not go signficantly further.

However, the experimentation did not yield any information about reaching a point of decline with respect to only varying the number of clients. As the number of clients was varied from 10 to 200, this points to the fact that the point of decline, in terms of number of clients, is

[5]throughput/*



Virtual Clients	10	20	30	40	50	60	70	80	90	100
200	10.98 σ= 0.5	11.18 σ= 0.15	10.96 σ= 0.43	10.16 σ= 0.78	10.62 σ= 0.25	10.37 σ= 0.4	10.26 σ= 0.19	10.29 σ= 0.37	10.42 σ= 0.15	10.43 σ= 0.39
190	11 σ= 0.17	10.85 σ= 0.23	10.77 σ= 0.16	10.73 σ= 0.37	10.14 σ= 0.35	10.28 σ= 0.38	10.9 σ= 0.77	10.34 σ= 0.16	10.24 σ= 0.28	10.39 σ= 0.15
180	11.06 σ= 0.37	10.78 σ= 0.32	10.52 σ= 0.56	10.72 σ= 0.2	10.58 σ= 0.41	10.43 σ= 0.14	10.2 σ= 0.17	10.26 σ= 0.16	10.17 σ= 0.34	10.27 σ= 0.21
170	11 σ= 0.26	10.93 σ= 0.24	10.73 σ= 0.2	10.41 σ= 0.22	10.38 σ= 0.43	10.43 σ= 0.2	10.34 σ= 0.24	10.24 σ= 0.28	10.36 σ= 0.23	10.25 σ= 0.19
160	11.07 σ= 0.34	10.75 σ= 0.25	10.68 σ= 0.36	10.48 σ= 0.22	10.47 σ= 0.32	10.18 σ= 0.21	10.26 σ= 0.07	10.23 σ= 0.18	10.33 σ= 0.28	10.37 σ= 0.16
150	11.05 σ= 0.21	10.5 σ= 0.5	10.65 σ= 0.52	10.66 σ= 0.16	10.32 σ= 0.42	10.25 σ= 0.25	10.16 σ= 0.24	10.3 σ= 0.1	10.23 σ= 0.23	10.08 σ= 0.31
140	10.84 σ= 0.11	10.73 σ= 0.62	10.85 σ= 0.23	10.39 σ= 0.38	10.36 σ= 0.43	10.26 σ= 0.19	10.38 σ= 0.21	10.33 σ= 0.21	10.4 σ= 0.25	10.34 σ= 0.27
130	10.72 σ= 0.18	10.52 σ= 0.33	10.44 σ= 0.41	10.58 σ= 0.22	10.29 σ= 0.07	10.39 σ= 0.19	10.28 σ= 0.28	10.15 σ= 0.13	10.42 σ= 0.32	10.25 σ= 0.14
120	11.01 σ= 0.16	10.7 σ= 0.25	10.55 σ= 0.24	10.35 σ= 0.33	10.29 σ= 0.36	10.48 σ= 0.16	10.35 σ= 0.1	10.1 σ= 0.26	10.4 σ= 0.41	10.39 σ= 0.17
110	10.65 σ= 0.2	10.64 σ= 0.18	10.42 σ= 0.19	10.44 σ= 0.26	10.42 σ= 0.26	10.42 σ= 0.21	10.55 σ= 0.2	10.12 σ= 0.15	10.38 σ= 0.25	10.21 σ= 0.29
100	10.89 σ= 0.31	10.8 σ= 0.21	10.38 σ= 0.32	10.47 σ= 0.26	10.42 σ= 0.11	10.2 σ= 0.22	10.44 σ= 0.35	10.31 σ= 0.15	10.37 σ= 0.13	10.16 σ= 0.28
90	10.64 σ= 0.39	10.48 σ= 0.43	10.51 σ= 0.21	10.41 σ= 0.36	10.31 σ= 0.15	10.5 σ= 0.2	10.46 σ= 0.33	10.4 σ= 0.24	10.49 σ= 0.13	10.61 σ= 0.12
80	10.74 σ= 0.29	10.52 σ= 0.42	10.51 σ= 0.17	10.65 σ= 0.15	10.59 σ= 0.12	10.6 σ= 0.17	10.35 σ= 0.16	10.3 σ= 0.34	10.57 σ= 0.16	10.28 σ= 0.16
70	11 σ= 0.24	10.6 σ= 0.12	10.67 σ= 0.26	10.89 σ= 0.14	10.63 σ= 0.26	10.33 σ= 0.16	10.69 σ= 0.32	10.39 σ= 0.28	10.31 σ= 0.11	10.12 σ= 0.3
60	11.17 σ= 0.28	10.79 σ= 0.28	10.54 σ= 0.13	10.92 σ= 0.21	10.54 σ= 0.47	10.57 σ= 0.16	10.45 σ= 0.24	10.28 σ= 0.11	10.14 σ= 0.26	10.38 σ= 0.07
50	11.04 σ= 0.13	10.7 σ= 0.27	10.6 σ= 0.29	10.74 σ= 0.27	10.16 σ= 0.77	10.33 σ= 0.35	10.24 σ= 0.21	10.25 σ= 0.17	9.95 σ= 0.46	10.12 σ= 0.09
40	11.09 σ= 0.17	10.7 σ= 0.29	10.24 σ= 0.38	10.47 σ= 0.2	10.28 σ= 0.27	10.01 σ= 0.22	10.08 σ= 0.11	9.92 σ= 0.18	9.8 σ= 0.17	9.82 σ= 0.09
30	10.36 σ= 0.27	10.04 σ= 0.23	9.75 σ= 0.15	9.75 σ= 0.12	9.6 σ= 0.14	9.47 σ= 0.21	9.36 σ= 0.19	9.2 σ= 0.15	9.1 σ= 0.07	9.15 σ= 0.17
20	8.87 σ= 0.35	8.67 σ= 0.15	8.61 σ= 0.17	8.57 σ= 0.19	8.39 σ= 0.21	8.01 σ= 0.22	8.02 σ= 0.22	7.9 σ= 0.08	7.88 σ= 0.09	7.87 σ= 0.16
10	6.19 σ= 0.23	6.12 σ= 0.27	5.89 σ= 0.14	5.62 σ= 0.3	5.63 σ= 0.23	5.34 σ= 0.16	5.16 σ= 0.2	5.23 σ= 0.1	5.25 σ= 0.1	5.05 σ= 0.1

Figure 3: Heatmap/table of Throughput (in terms of a thousand requests per second) with standard deviation, for each configuration of reader threads and virtual client count.

possibly beyond 200. Therefore we have only provided evidence in favor of the first part of the hypothesis, which states that:

> For a fixed number of *reader threads*, the behavior of **mean throughput** versus number of clients is expected to increase at first until a point of saturation, whereby throughput remains relatively the same..

We cannot accept or reject the remaining part of the hypothesis with the data we have gathered.

Hypothesis #2: Mean Throughput under varying number of readers and constant number of clients

The behavior of the data suggests that all information outlined in the hypothesis related to the behavior of the system as the number of reader threads is increased for a constant number of connected clients is incorrect.

Throughput does not appear to increase as the number of reader threads is increased, but shows a pattern of decline from the very beginning. It appears that the reader threads configuration achieving the maximum number of throughput the most is the minimum tested of 10 threads, while the lowest performing configuration is that with 100 reader threads per server.

This behavior was quite unexpected, but it indicates that the overhead of adding more reader threads to the system is quite large.

1.1.5 Conclusion

From the observed data in figure 3, we see that the maximum mean throughput is estimated to be around the value of 11,000 operations per second, and most frequently occurs in configurations that utilize only 10 reader threads to serve their clients. Values that are slightly within the range of 11,000 occur throughout various configurations that serve between 40 and 200 clients. Therefore, in order to choose the simplest configuration, we choose the configuration of 40 clients.

We will now designate the configuration of 10 reader threads and 40 virtual clients, which achieves an estimated mean throughput of $11,090$ operations per second with a standard deviation of 170, as the simplest configuration needed to achieve the maximum throughput of the system.

1.2 Detailed Time Breakdown

After finding the simplest configuration that achieves saturation, this section contains a detailed breakdown of the time spent by each operation type in that configuration. First we hypothesize how the behavior of time expenditure be. Then we investigate T_{Total}[6] from the perspective of the load generators, as recorded by Memaslap. Finally we inspect the detailed breakdown of how much time is spent by each operation type ($T_{Hashing}$[7], T_{Queue}[8], $T_{Processing}$[9]) inside each part of the middleware. The data used in this breakdown is extracted from the data[10] logged in the previous experiment. Therefore, no methodoloy section is provided as no additional experimentation was needed. We reuse the data from the experiments on 40 clients and 10 reader threads.

1.2.1 Hypotheses

1. T_{Total} percentile values will be significantly higher than all others, perhaps by an order of maginuted.
2. $T_{Hashing}$ percentile values are expected to be significantly smaller than all others, perhals by an order of magnitude.
3. T_{Queue} percentile values are expected to be on the same order of magnitude as $T_{Processing}$, between T_{Hash} and T_{Total}.

[6]Response time observed by client.
[7]Time taken by middleware to perform constant hashing and route the request.
[8]Time spent by a request within a queue.
[9]Time spent by a thread waiting for responses to a request from memcached.
[10]throughput/*

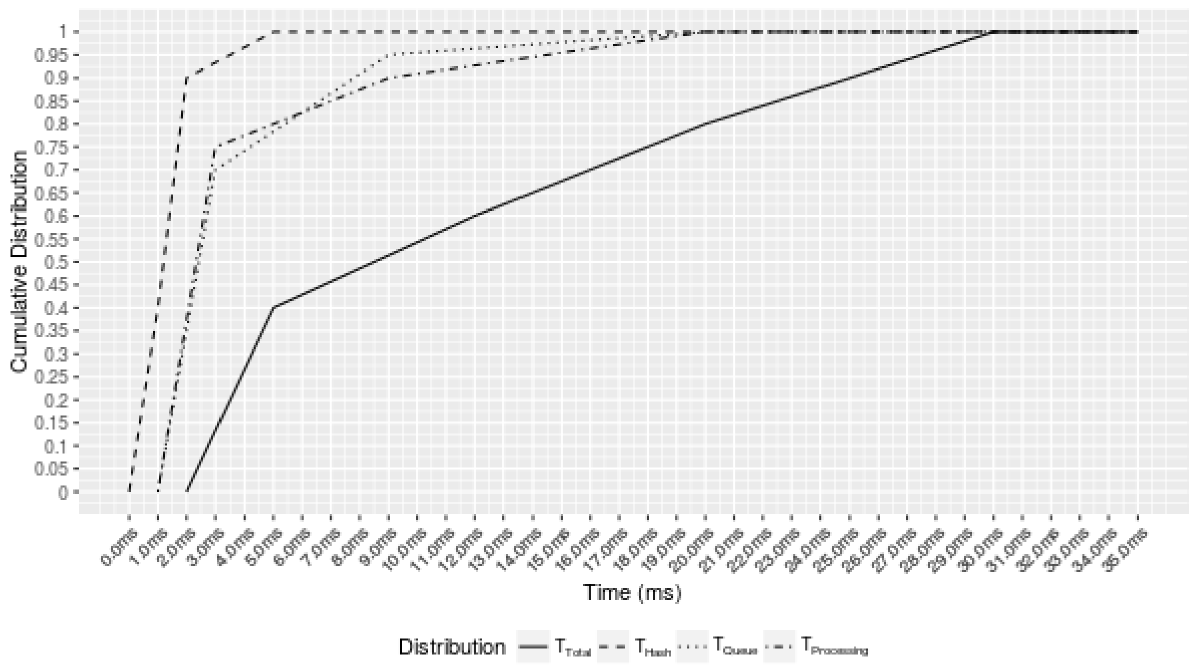


Figure 4: Projections of time expenditure distributions.

1.2.2 Results

Here we present the data related to time expenditure in the form of cumulative distribution functions, which highlight the differences in the distributions of the different times. The CDFs are calculated by aggregating the timing logs of all 5 experiment samples.

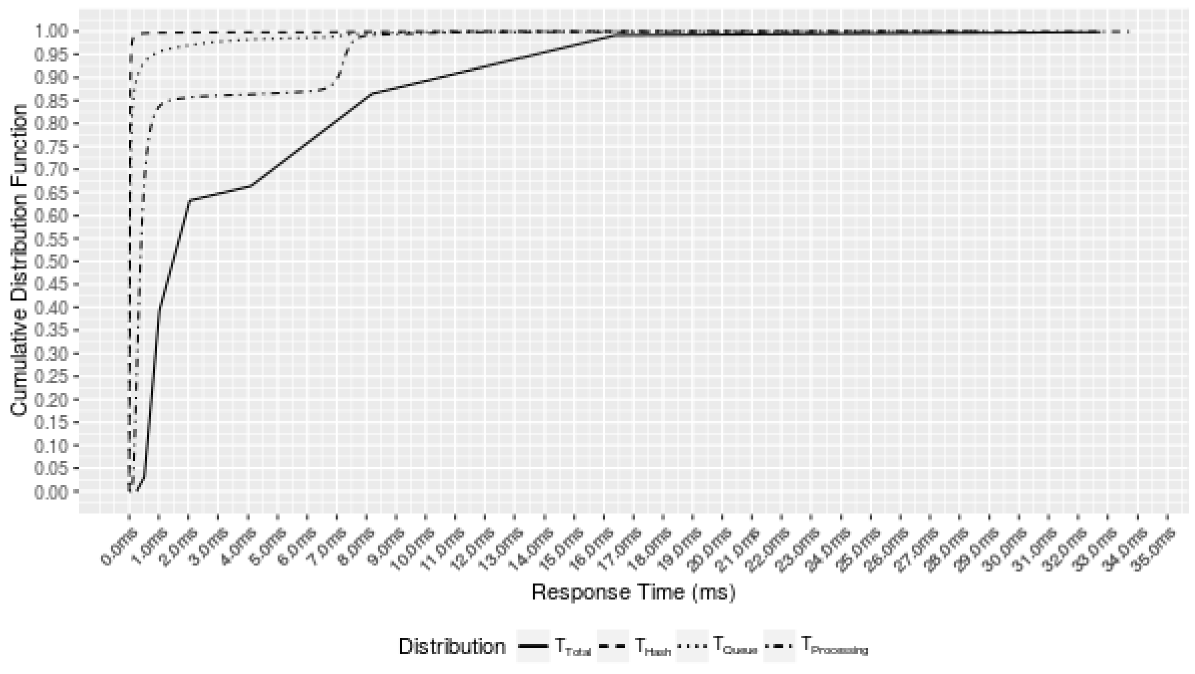


Figure 5: Cumulative distribution function of T_{Total} as seen by memaslap, and $T_{Hashing}$, $T_{Queueing}$, $T_{Processing}$ as recorded by the middleware.

Memaslap provides a frequency distribution of response times with bin sizes that are powers of 2, therefore the data for T_{Total} in figure 5 is interpolated for intermediate percentile values.

We can observe any percentile of interest from the plot in figure 5. For example:

- 50^{th} **Percentile:** $T_{Hash} \approx 0ms$, $T_{Queue} \approx 0ms$, $T_{Processing} \approx 0.1ms$, $T_{Total} \approx 1.5ms$
- 90^{th} **Percentile:** $T_{Hash} \approx 0ms$, $T_{Queue} \approx 0.1ms$, $T_{Processing} \approx 7ms$, $T_{Total} \approx 11ms$
- 99^{th} **Percentile:** $T_{Hash} \approx 0ms$, $T_{Queue} \approx 6.5ms$, $T_{Processing} \approx 8ms$, $T_{Total} \approx 17ms$

1.2.3 Analysis

Returning back to our initial assumptions about time expenditure, we see from figure 5 that we were correct about our first 2 hypotheses, and were significantly off in our estimates of how the distributions will actually look like (as estimated in figure 4), as we largely over-estimated the values each distribution will observe.

As for our third hypothesis, we expected that both queueing time and processing time would observe similar distributions, as the two times are closely interrelated. However, queueing time is observed to be a less significant contributor to the time spent in the middleware than processing time. Instead we see that queuing time is more comparable to hashing time, a fairly efficiently implemented procedure, as both distributions oberve a very low 90th percentile value ($< 0.1ms$).

Hashing time appears to be the most resilient distribution, as even at the 99th percentile its values are below 0.1 ms, while all other distributions observe a significant increase.

1.2.4 Conclusion

We conclude from this discussion the following points: Our assumptions about the expected contribution to performance by each component of our middleware implementation were invalidated, which means we must revisit our design descisions, particularly those related to Synchronous reader threads. Arguably, the core functionality of our middleware, which is to route requests based on the output of a function on their keys, costs very little computational power, which means that the added overhead of managing queues and ferrying requests degrades performance by orders of magnitudes; if this middleware were replaced by an advanced connection multiplexer, performance would be comparable to direct connections to memcached.

2 Effect of Replication

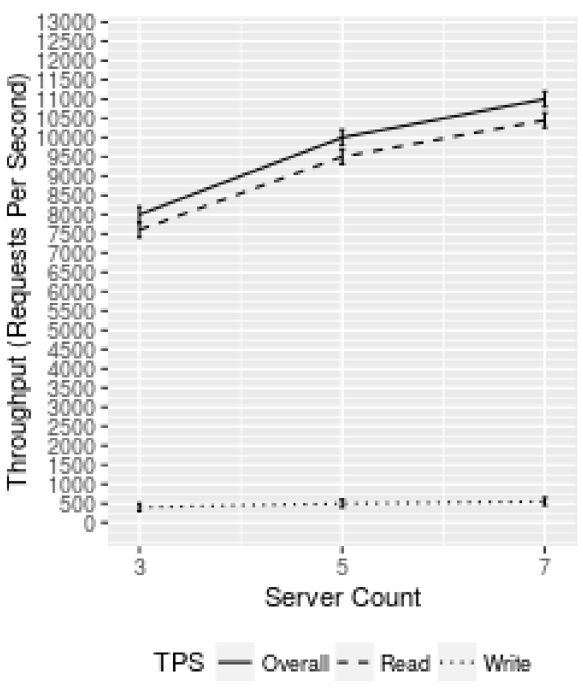
2.1 Introduction

In this section we explore the changes in behavior of our system in response to varying both the number of memcached key-value stores routed by our middleware and the replication factor used for write requests, all while generating a load on the system that is composed of 5% write requests and 95% read requests. From this point on, when refering to percentile values, we mean the 50th, 90th and 99th percentiles. We expect the following assumptions to hold true for our system:

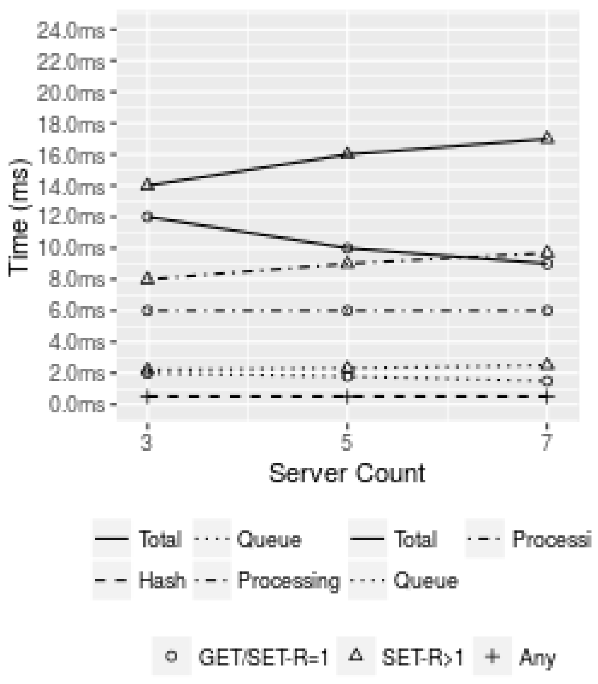
- For a **fixed replication factor**, and an **increasing number of memcached servers**:
  1. The overall mean throughput is expected to increase.
  2. The overall T_{Total} times are expected to decrease.
  3. T_{Hash} times will not change.
  4. SET Requests:
     (a) Mean throughput will increase.

(b) Depending on Replication Factor
    i. No Replication: T_{Total} will decrease. T_{Queue} times will decrease. $T_{Processing}$ will not change.
    ii. Half or Full Replication: T_{Total} will increase. T_{Queue} times will increase. $T_{Processing}$ will increase.

5. GET Requests: (a) Mean throughput will increase. (b) T_{Total} will decrease. T_{Queue} will decrease. $T_{Processing}$ will not change.



(a) Expected behavior of mean throughputs.



(b) Expected behavior of 90th percentiles of T_{Total}, T_{Hash}, T_{Queue}, $T_{Processing}$.

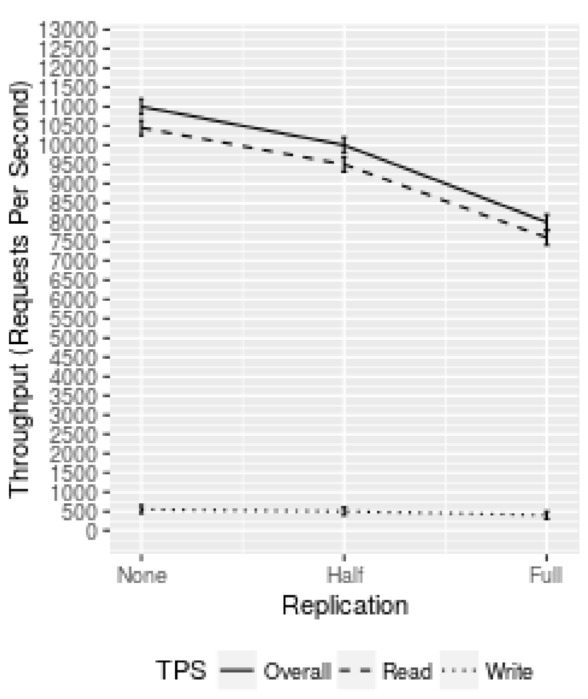
Figure 6: Plot of expected SUT behavior according to hypotheses.

- For a **fixed number of memcached servers**, and an **increasing replication factor**:

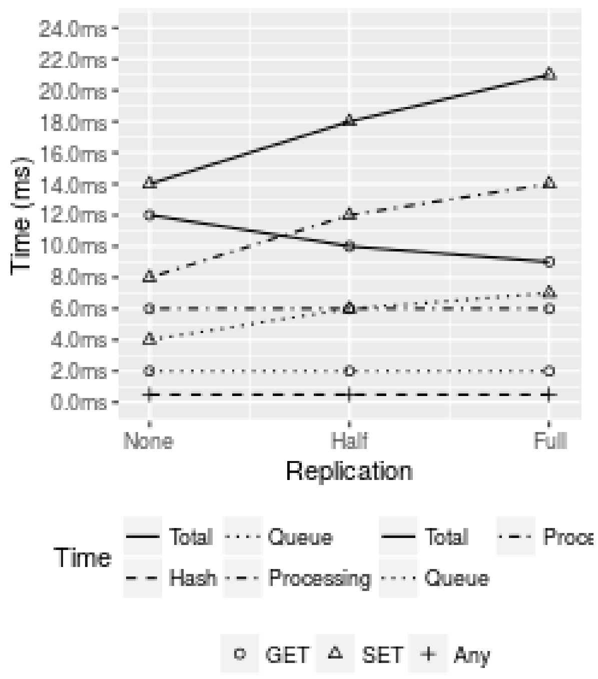
1. The overall mean throughput is expected to decrease.
2. The overall T_{Total} times are expected to increase.
3. T_{Hash} times will not change.
4. SET Requests: (a) Mean throughput will decrease. (b) T_{Total} will increase. T_{Queue} times will increase. $T_{Processing}$ will increase.
5. GET Requests: (a) Mean throughput will decrease. (b) T_{Total} will decrease. T_{Queue} times will not change. $T_{Processing}$ will not change.

GET requests and SET requests are not expected to be impacted the same way because their implementations within the middleware are different (synchronous vs asynchronous).

Our projections and estimations for response time behavior serve to give an idea of how our system's Speed-up and Scale-up also behave.



(a) Expected behavior of mean throughputs.



(b) Expected behavior of 90th percentiles of T_{Total}, T_{Hash}, T_{Queue}, $T_{Processing}$.

Figure 7: Plot of expected SUT behavior according to hypotheses.

So far we have defined our hypotheses and expectations about how the behavior of the SUT will change under different configurations. We will now define how an **ideal** implementation would behave in terms of scalability under changing configurations:

- For a **fixed replication factor**, and an **increasing number of memcached servers**:
  1. Mean throughput will increase **linearly**.
  2. T_{Total}, and T_{Queue} will decrease **linearly**.
  3. T_{Hash}, and $T_{Processing}$ will be unaffected.
- For a **fixed number of memcached servers**, and an **increasing replication factor**:
  1. Mean throughput will not be affected.
  2. T_{Total}, T_{Queue}, T_{Hash}, and $T_{Processing}$ will be unaffected.

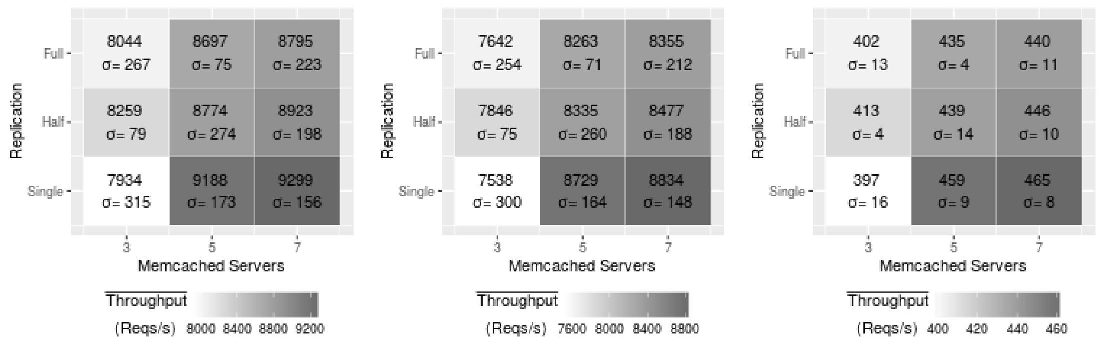
2.2 Methods

Number of servers	3,5,7
Replication	R=1,R=$\lceil \frac{S}{2} \rceil$,R=all
Number of client machines	2
Virtual clients / machine	20
Reader Threads	10
Workload	Key 16B, Value 128B, 5%-writes
Window Size	1,000
Overwrite Proportion (-o flag)	0.9
Runtime x repetitions m	1m x 5
Log files	effects/*
Design	Full-Factorial
Experiments	3*3*5 = 45

The workload was composed of 95% ”GET” operations, and 5% ”SET” operations. The generators sent set random values of 128 bytes each with 16 byte keys, and continuously fetch those values and overwriting 90% of them throughput the experiment. Each of the two load generators created 20 virtual clients (total 40) to connect to the middleware. The middleware operated 10 reader threads.

2.3 Results

In this section we present the collected data from the experiments in the form of heatmaps/tables. Throughput is presented as an estimated mean with standard deviation, while T_{Total}, T_{Queue}, T_{Hash}, and $T_{Processing}$ are presented using their respective 90th percentile values. The collected data from the experiments are presented in the form of heatmaps. For each experiment, the logs[11] are used as follows: Throughput is calculated from the memaslap logs. The T_{Total} percentiles are calculated by interpolation from the aggergation of memaslap logs for each experiment. All other times are calculated from the middleware logs.

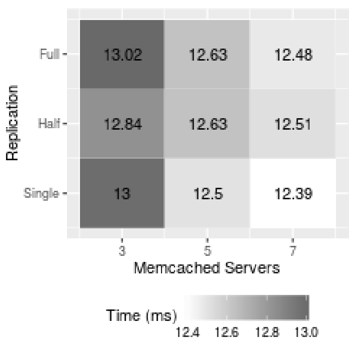


(a) Mean Overall Throughput (b) Mean GET Throughput (c) Mean SET Throughput

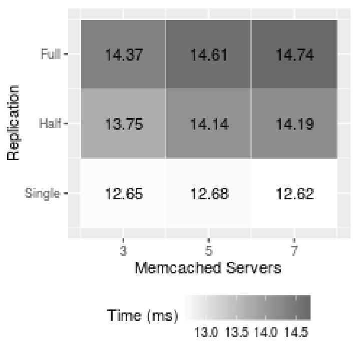
Figure 8: Throughput performance metrics for 5%-write workload.

This data presentation method (heatmap) was found to be the most suitable means to display the experiment data. The color encoding conveys at a glance where the minima and maxima are, along with an intuition of the gradient of the values. Overlaying the actual values on top of each heatmap cell provides the actual numerical data for reference. The color of each cell is based solely on the value of the top entry in the cell (the estimate) and is not affected in any way by the standard deviation (bottom entry).

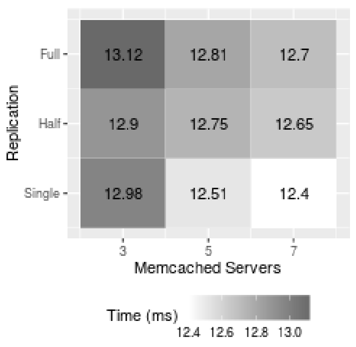
[11]effects/*



(a) Overall 90th Percentile Response Time

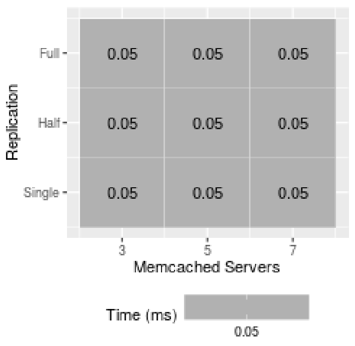


(b) GET 90th Percentile Response Time

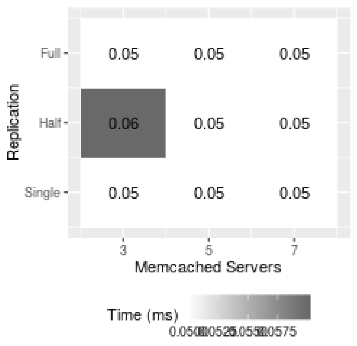


(c) SET 90th Percentile Response Time

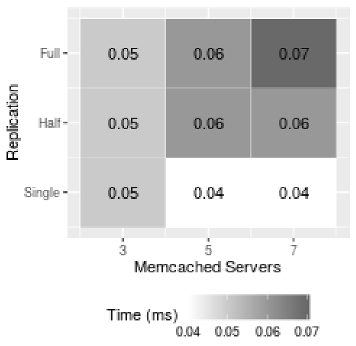
Figure 9: 90th percentile response time for 5%-write workload.



(a) Overall 90th Percentile Hashing Time

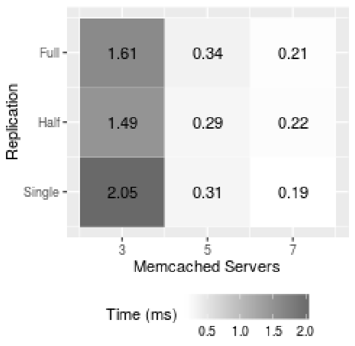


(b) GET 90th Percentile Hashing Time

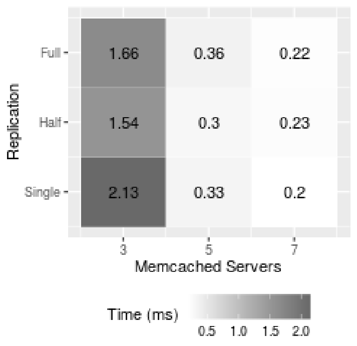


(c) SET 90th Percentile Hashing Time

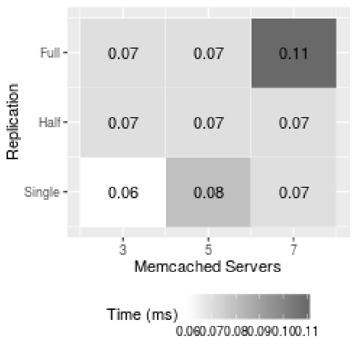
Figure 10: 90th percentile hashing time for 5%-write workload.



(a) Overall 90th Percentile Queuing Time



(b) GET 90th Percentile Queuing Time

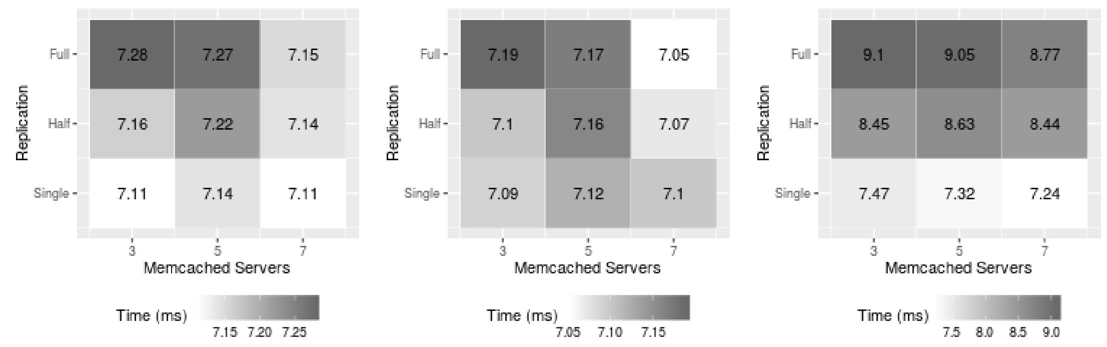


(c) SET 90th Percentile Queuing Time

Figure 11: 90th percentile queuing time for 5%-write workload.

2.4 Analysis

In this section we proceed to analyze the resulting data from our experiments. Our main goals are to analyze the impact of changing the number of servers and the replication factor on the performance of the middleware for each operation, and to analyze how scalable the system is as the backing infrastructure configuration and redundancy strategy change, when compared to an ideally scalable implementation.



(a) Overall 90th Percentile Processing Time (b) GET 90th Percentile Processing Time (c) SET 90th Percentile Processing Time

Figure 12: 90th percentile processing time for 5%-write workload.

We will again revisit our assumptions stated in the introduction to this section and verify their authenticity or reject them by leveraging the resulting experiment data. Keep in mind that the workload here is mostly composed of gets, so it's reasonable for the overall behavior to be dominated by the patterns in GET operation behavior.

2.4.1 Fixed replication factor, Increasing number of memcached servers

Assumption 1: As can be observed from the table in figure 8a, the estimated mean overall throughput does tend to increase as the number of backing memcached servers increases for each replication factor.

Assumption 2: Figure 9a slightly confirms this trend, as the overall response time does decrease as more memcached servers are employed.

Assumption 3: Figure 10 shows that hashing times do observe trendas that are specific to the request type while the overall distribution shows no trends. This evidence nullifies our 3rd hypothesis.

Assumption 4.a: The data in figure 8c does confirm this hypothesis. For any replication factor, adding more servers to the memcached pool does increase the estimated mean throughput.

Assumption 4.b.i: This assumption was based on the fact that the middleware would have to perform no additional work when processing write requests without replication as more servers are added, and will have more queues available to process incoming write requests. As can be seen in figures 9c,11c, and 12c, the above assumption appears to be partly false, since T_{Queue} seems to increase as more servers are added while using no replication.

Assumption 4.b.ii: All assumptions about T_{Total}, T_{Queue} and $T_{Processing}$ are clearly false according to the data in figures 9c,11c, and 12c. T_{Queue} observes no recognizable trend, while T_{Total} and $T_{Processing}$ behave opposite to what was predicted.

Assumption 5.a: Upon inspection of figure 8b, it can be inferred that this assumption holds true, and that the main contribution to the increase in overall throughput comes from the increase in GET request throughput.

Assumption 5.b: The data in figures 9b and 12b invalidate parts of this assumption. T_{Total} appears to increase for GET requests as more servers are added, while $T_{Processing}$ shows no apparent trends. Figure 11b however shows evidence that the predicted trend for T_{Queue} is valid, which aligns with our understanding of our implementation as more key value stores being added to the middleware entails more processing queues for GET requests being available.

2.4.2 Fixed number of memcached servers, Increasing replication factor

Assumption 1: Figure 8a confirms this assumption. It also aligns with our understanding of the system implementation, since increasing the replication factor entails more processing to be done for each SET request, and thus, more delays for our closed system, which decreases throughput.

Assumption 2: There seems to be little to no correlation between response time and replication factor in figure 9a. We choose to reject this hypothesis.

Assumption 3: Figure 10 shows that hashing times do observe trendas that are specific to the request type while the overall distribution shows no trends. This evidence nullifies our 3rd hypothesis.

Assumption 4.a: Estimated mean throughput for SET requests observes a decreasing trend as the replication factor is increased, as shown in figure 8c. The downward trend becomes more noticable as more servers are used.

Assumption 4.b: The increasing trend in T_{Total} is significantly present in the data in figure 9c. As the replication factor is increased, response times increase significantly. This is to be expected since more servers imply more work to be done when replicating, and therefore more delay in response. Trends for T_{Queue} are not visible from our data (figure 11c) however, but $T_{Processing}$ shows an increasing trend as predicted (figure 12c).

Assumption 5.a: Mean throughput for GET commands observes a significant decreasing trend when utilizing more than 3 servers and increasing the replication factor (figure 8b). This correlates with the other trends for response time stated in previous assumptions, since when write replication starts to be more significantly expensive to process as more servers are added, clients have to wait more for SETs to finish and therefore send less GET requests per second.

Assumption 5.b: Figure 9b shows that our assumption for T_{Total} in this hypothesis was incorrect, since response times for GET requests appear to increase as replication is increased. T_{Queue} for GET (figure 11b) shows no significant trend as the replication factor is increased, which aligns with our prediction. Moreover, $T_{Processing}$ for GET requests (figure 12b) seems uncorrelated to the replication factor, as predicted. This aligns with our understanding of the SUT implementation, since GET request processing is independent of replication.

2.4.3 Scalability VS An Ideal System

Our implementation is far from ideal. We will compare the performance aspects of our system to the defined behaviors of an ideal system we mentioned earlier. We should also mention that while we have not investigated whether the shifts in performance we observe in different configurations of our system come from external sources such as network congestion, we still hold our ideal system model to the standard that it should be able to compensate for these factors.

Increasing Server Count

Point 1: In comparison to the ideal system's throughput gains, our system's gains are not linear. Transitioning from using 3 servers to 5 servers shows the highest gains in throughput (8a), while adding more servers from that point onwards does not as significantly impact performance. This is because our model of an ideal system ignores any added overhead needed to manage more servers.

Point 2: We still do not observe the ideal system's linear decreases. As can be seen in figures 9a and 11a, a significant decrease in both T_{Total} and T_{Queue} happens when transitioning from using 3 servers to 5. After that, increasing the server count to 7 gives very little gain. We expect the gains to keep decreasing for our SUT.

Point 3: Our system also fails the two metrics in this point. Our SUT's T_{Hash} times (figure 10) are not impacted overall, but for each operation type there are different patterns.

GET requests seem to have reasonably non-changing behavior, but SET requests suffer in performance with server scaling. Moreover, $T_{Processing}$ also changes, unlike in the ideal system model. Hashing is a function that is ideally done independent of how many servers there are in use, and therefore its efficiency should not be affected by adding more servers to the back-end. Processing time also accounts for an amount of time spent waiting for a server to respond, and since servers should operate independently of each other, their wait times should also be ideally independent.

Increasing Replication Factor

Point 1: Ideally, increasing the replication factor should not negatively impact throughput. This could be more true in a system where more than one asynchronous writer thread operates, but we still hold our ideal system to this standard. Our SUT though, fails this standard as can be seen from the declines in performance as the replication factor is increased in figure 8.

Point 2: Increasing the replication factor is a change in configuration that makes the system do more work per SET request. Ideally, the system should be able to asynchronously handle this and not cause any delays in responding to requests. For our SUT, however, the data in figures 9,10, 11 and 12 show otherwise. Our SUT requries a significant amount of overhead in order to perform replication, even though its internal writer thread is programmed to be asynchronous.

2.5 Conclusion

So far we have concluded how our SUT behaves as it is given more resources under different configurations. This information allowed us to estimate the Speed-Up and the Scale-Up[2] behaviors of our SUT under different replication conditions. We also observe how our system obeys Amdahl's law as more servers are added to parallelize processing the workload.

3 Effect of Writes

3.1 Introduction

In this section we focus on investigating the impact of different workloads on system performance. Particularly, we are interested in measuring how the system under test reacts as the write percentage of the workload is increased. For workloads of 1%, 5% and 10% writes, we will vary the number of servers (3, 5, 7) and replication factor (1, full) while investigating the resulting changes in performance. We expect overall behavior to be mainly influenced by GET request behavior due to the high ratio of GET to SET.

We propose the following hypotheses about the expected behavior of our SUT as the percentage of writes increases:

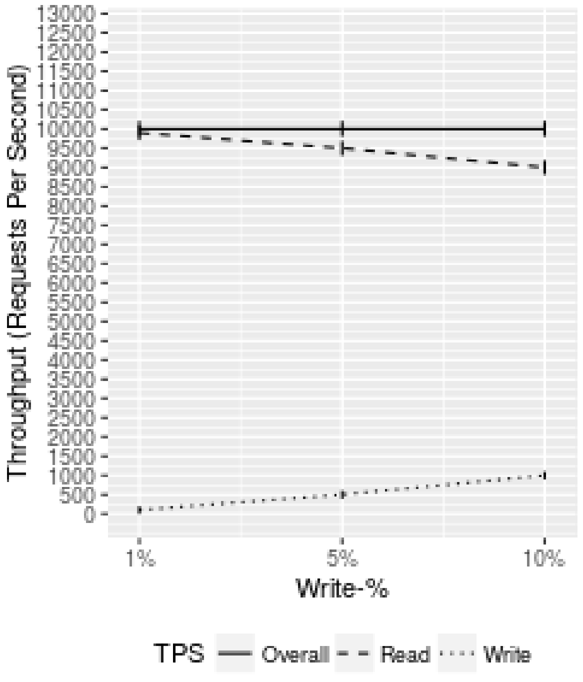
1. GET: (a) Throughput will decrease. (b) T_{Total} will increase, T_{Queue} will decrease, $T_{Processing}$ will increase.

2. SET: (a) Throughput will increase. (b) T_{Total} will increase, T_{Queue} will increase, $T_{Processing}$ will increase.

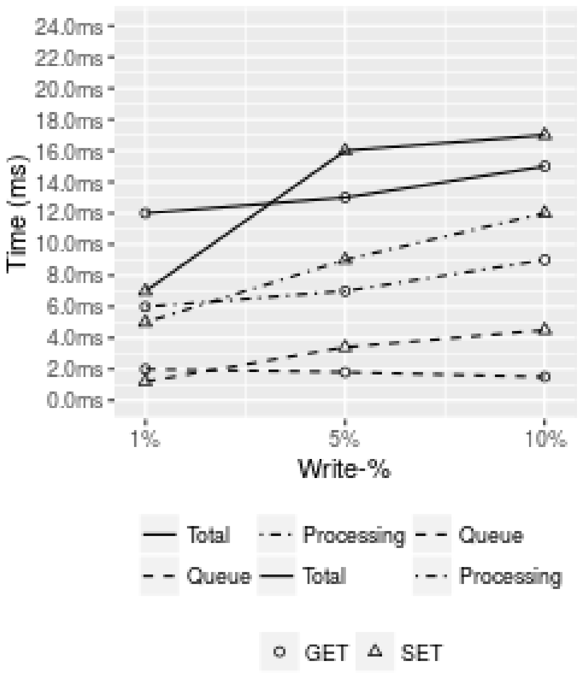
We hypothesize that the replication factor will only affect the significance of the above predicted trends. For example, SET throughput will always increase as the percentage of writes increases, but the increase for R=1 will be higher than for R=ALL.

Therefore, if these assumptions hold true, then it follows that the biggest impact in performance, relative to the base case of a 1%-Write workload utilizing 3 servers and no replication, would be observed at a configuration of 10% writes, 7 servers and full replication.

We are also interested in the Scale-Out behavior of out SUT in this section since our main focus is our SUT's response to work load changes.



(a) Expected behavior of mean throughputs.



(b) Expected behavior of 90th percentiles of T_{Total}, T_{Queue}, $T_{Processing}$.

Figure 13: Plot of expected SUT behavior according to hypotheses.

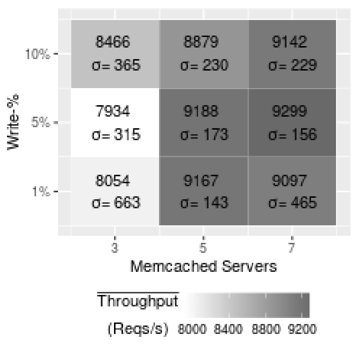
3.2 Methods

Number of servers	3,5,7
Replication	R=1,R=all
Number of client machines	2
Virtual clients / machine	20
Reader Threads	10
Workload	Key 16B, Value 128B , Writes: 1%,5%,10%
Window Size	1,000
Overwrite Proportion (-o flag)	0.9
Runtime x repetitions m	1m x 5
Log files	effects/*
Design	Full-Factorial
Experiments	2*3*3*5 = 90

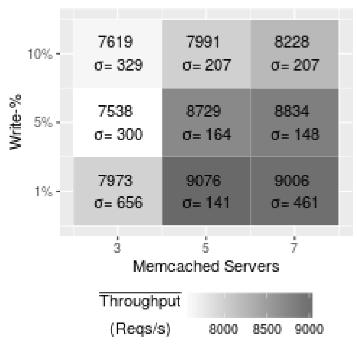
The workload was composed of only "GET" and "SET" operations. The percentage of SET operations was tested at 1%, 5% and 10%. The generators sent set random values of 128 bytes each with 16 byte keys, and continuously fetch those values while overwriting 90% of them throughput the experiment. Each of the two load generator created 20 virtual clients (total 40) to connect to the middleware. The middleware operated 10 reader threads.

3.3 Results

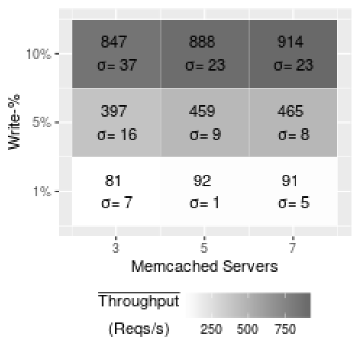
The collected data from the experiments are presented in the form of heatmaps.



(a) Mean Overall Throughput

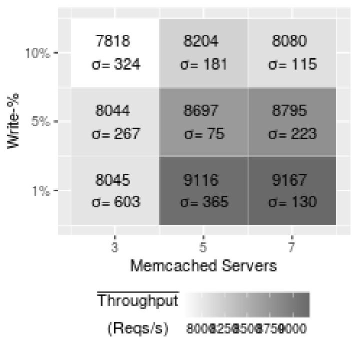


(b) Mean GET Throughput

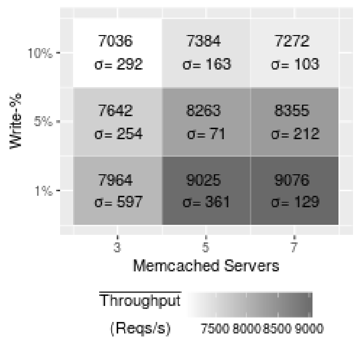


(c) Mean SET Throughput

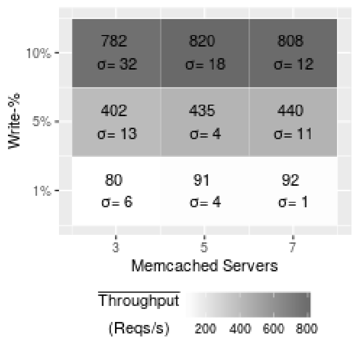
Figure 14: **R=1**. Throughput performance metrics.



(a) Mean Overall Throughput

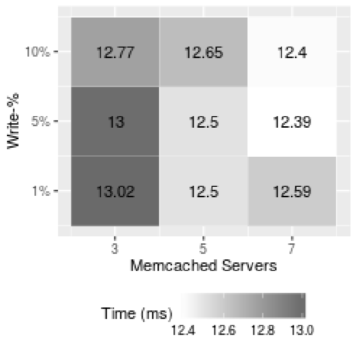


(b) Mean GET Throughput

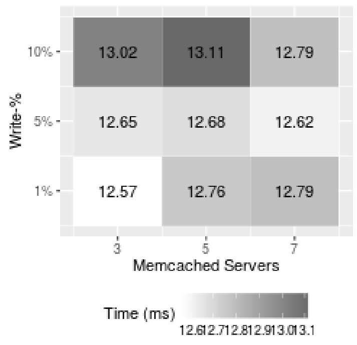


(c) Mean SET Throughput

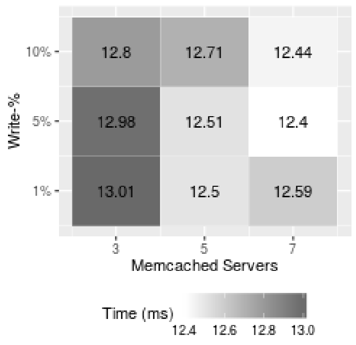
Figure 15: **R=ALL**. Throughput performance metrics.



(a) Overall 90th Percentile Response Time



(b) GET 90th Percentile Response Time

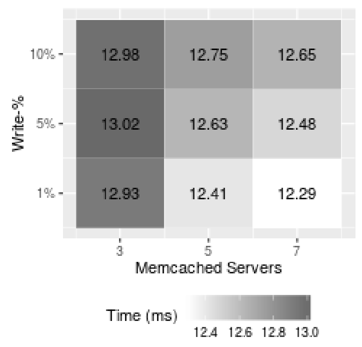


(c) SET 90th Percentile Response Time

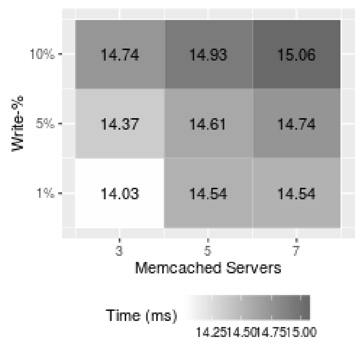
Figure 16: **R=1.** 90th percentile response times.

3.4 Analysis

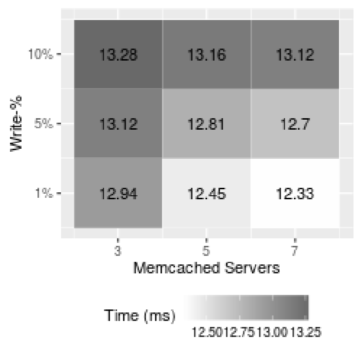
We proceed to analyse the data from our experiment in order to confirm or reject our hypotheses about what happens as the percentage of writes is increased, and investigate the causes for behavior.



(a) Overall 90th Percentile Response Time

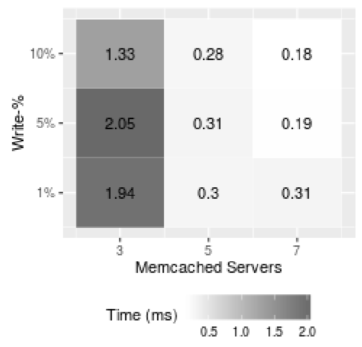


(b) GET 90th Percentile Response Time

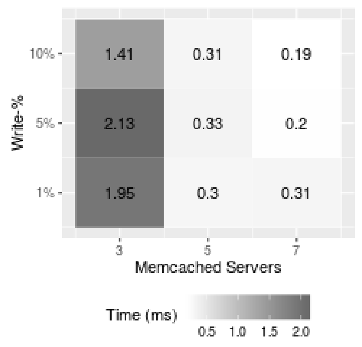


(c) SET 90th Percentile Response Time

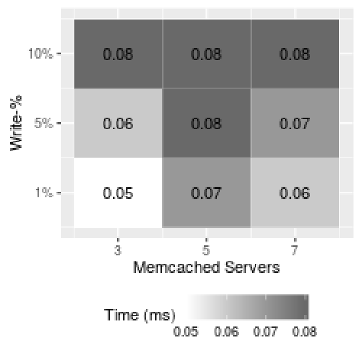
Figure 17: **R=ALL.** 90th percentile response times.



(a) Overall 90th Percentile Queuing Time

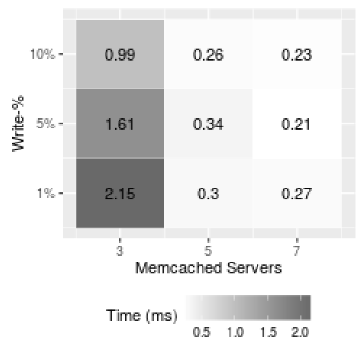


(b) GET 90th Percentile Queuing Time

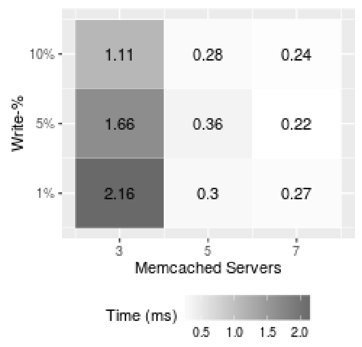


(c) SET 90th Percentile Queuing Time

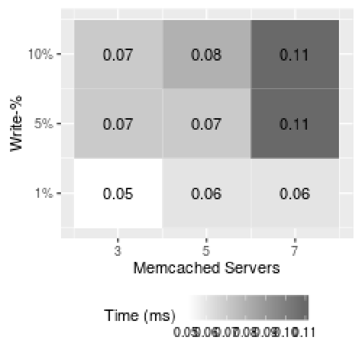
Figure 18: **R=1.** 90th percentile queuing times.



(a) Overall 90th Percentile Queuing Time



(b) GET 90th Percentile Queuing Time

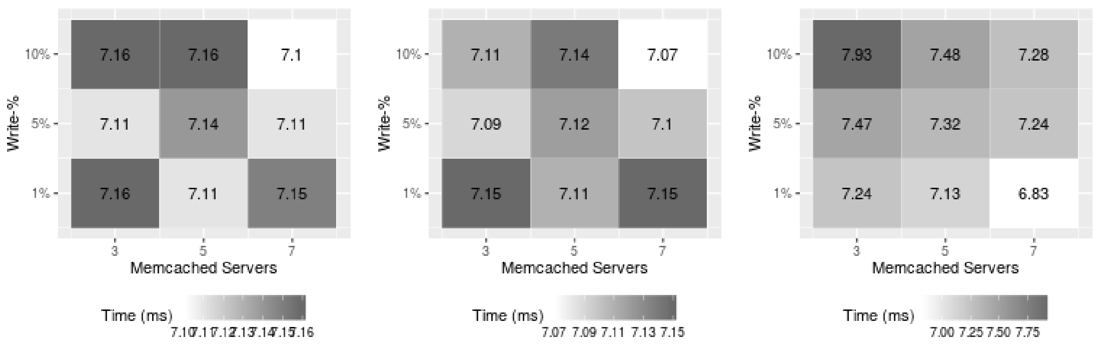


(c) SET 90th Percentile Queuing Time

Figure 19: **R=ALL.** 90th percentile queuing times.

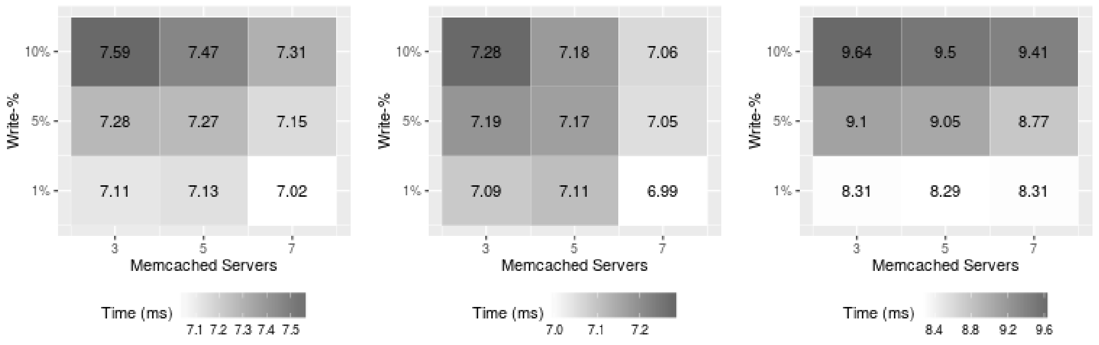
3.4.1 System Behavior

Hypothesis 1.a: Figures 14b and 15b show how the measured behavior or GET throughput aligns with our hypothesis. This degradation in throughput is expected as we believe that writes



(a) Overall 90th Percentile Processing Time (b) GET 90th Percentile Processing Time (c) SET 90th Percentile Processing Time

Figure 20: **R=1.** 90th percentile processing times.



(a) Overall 90th Percentile Processing Time (b) GET 90th Percentile Processing Time (c) SET 90th Percentile Processing Time

Figure 21: **R=ALL.** 90th percentile processing times.

are more expensive to process thatn reads. Since this is a closed system, clients will have to wait for more time as they increase the percentage of write requests that they send. Therefore, less GET requests are sent and processed per second. Relative to the base case, the biggest impact on TPS performance is seen when using 7 servers and full replication.

Hypothesis 1.b: T_{Total} trends in figures 16b and 17b allude to the validity of this assumption. The increase in response time becomes less noticeable as more servers are added without replication, but is exaggerated when adding more servers with replication. T_{Queue} trends in figures 18b and 19b also confirm our assumptions, except with a few spikes which we will consider as outliers. Queueing time for GETs decreases as more writes are sent simply because the queues for GET requests are separate from those for writes and therefore become less congested. $T_{Processing}$ trends in figures 20b and 21b also show the expected behavior, which is to increase, since we postulate that set requests are also more expensive within memcached, and therefore cause its performance to degrade. Also, since the writes are sent asynchronously, our middleware does not wait for a response for set requests from memcached before sending another set request, and therefore increasing the percentage of writes stresses memcached to dedicate more resources to responding to SETs rather than GETs. These trends are most significantly higlighted when using 7 servers with full replication.

Hypothesis 2.a: This is a trivial observation, since increasing the percentage of writes will increase the number of SET requests and thus their throughputs (figures 14c, 15c). However, we point out that when employing full replication, the increase in SET TPS with respect to the increase in write-% is dampened. This behavior is reasonable and it aligns with our understanding of our system, since full replication requires more processing by our middleware and would therefore impact SET request performance. The effects are most significantly observed for 7 servers with full replication.

Hypothesis 2.b: T_{Total} trends in figure 16c fluctuate and do not give solid indication as to how response time for SET requests is behaving as the write-% is increased without replication. However, figure 17c clearly highlight the expected behaviors of increase in response time. We mainly set forth this hypothesis since we expected write queues to become more congested as the SET-% increases. T_{Queue} trends in figures 18c and 19c show an increase, as predicted. The trends without replication (18c) appear to be more linear than those with replication (19c). $T_{Processing}$ increases are significantly apparent in figures 20c and 21c. All effects are most significantly observed for 7 servers with full replication.

3.5 Conclusion

Thusfar, we have investiged the impact of increasing write requests and compared it in different cases. We have also identified and detailed the main reason for performance loss, which is write replication. Our investigations allowed us to estimate the Scale-Out[2] behavior of our system under different workloads.

References

[1] Central Limit Theorem `https://en.wikipedia.org/wiki/Central_limit_theorem`

[2] Bukh, Per Nikolaj D., and Raj Jain. "The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling." (1992): 113-115.

Logfile listing

Short name	Location
throughput/*	`https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/tree/master/milestone2/`
effects/*	`https://gitlab.inf.ethz.ch/rkhalil/asl-fall16-project/tree/master/milestone2/`