Large-Scale Systems Engineering - Group 15
Project - Stage II

André Afonso Fitas Gonçalves (ist195532 - andre.a.goncalves@tecnico.ulisboa.pt), Diogo Miguel Da Silva Santos (ist195562 - diogosilvasantos@tecnico.ulisboa.pt), and Vasco Miguel Cardoso Correia (ist194188 - vasco.cardoso.correia@tecnico.ulisboa.pt)

Instituto Superior Técnico

1 Introduction

When granted the liberty to select a system for analysis, we gravitated towards projects in the domain of distributed systems, particularly those that use or implement algorithms covered in previous courses, such as Distributed Systems, Design and Implementation of Distributed Applications or Highly Dependable Systems.

Etcd, a consistent distributed key-value store, unquestionably stood out as the prime selection due to its open-source nature, transparent client-server architecture, distributed structure, implementation of the well-studied Raft algorithm, and its integral role as a fundamental building block of Kubernetes, where it stores and manages state, data configuration and metadata.

The project can be found at https://github.com/Vaascoo/esle and the commit to be considered is `0294e356e2200c19fb45afe836de424b79e3bfe4`.

2 System Description

Etcd is a consistent distributed key-value store, built on top of the Raft algorithm, a leader-based consensus protocol. It is mainly used as a coordination service in distributed systems.

It is heavily dependent on Raft, which is a consensus algorithm equivalent to the classic Paxos[2] algorithm in fault-tolerance and performance. Raft is a somewhat simpler alternative, being built with a focus in real-world use for practical systems.

Using the main concepts of leader election, consensus and terms, Raft manages to safely replicate a log between its machines. That log is then used by Etcd to decide the order of loads and stores in its multi-version concurrency control key-value store.

On top of the Raft package, Etcd also implements:

1. A simple user-facing API using gRPC;

2. A multi-version concurrency control key-value store;
3. Network communication between its nodes;
4. Security, using automatic TLS with optional client cert authentication.

The overall system architecture can be observed in Figure 1.

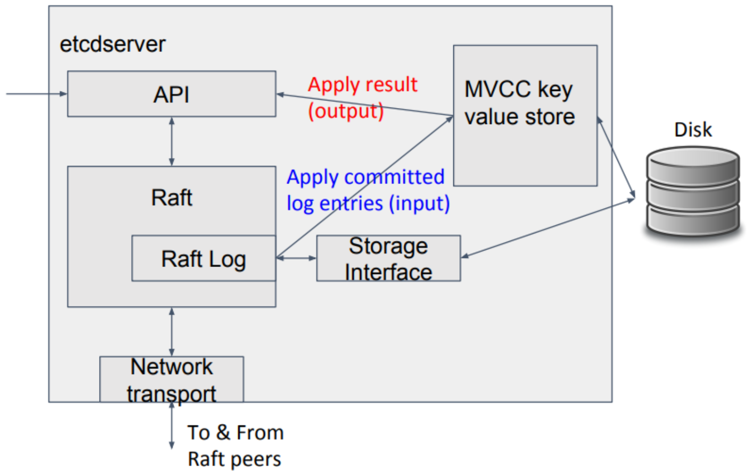

Fig. 1. Etcd system architecture

3 Experimental Design

3.1 Introduction

To continue the work done in Stage I and further analyse the Etcd system behavior in a more real setup, we first migrated the local cluster deployment to a remote deployment using Google Cloud platform, in particular Google Compute Engine. The infrastructure is deployed using Terraform scripts and Ansible playbooks.

3.2 Choosing Factors

To systematize the system analysis we designed an experiment to assess the effect of six different factors at two level each:

1. CPU Cores - Number of virtual CPU cores per Google Compute Instance;
2. Disk Configuration - Type of disk to where Etcd data is persisted;
3. Number of nodes - Number of Etcd nodes in the cluster;
4. Snapshot Count - Number of committed transactions to trigger a snapshot to disk;

5. Backend Batch Limit - Maximum amount of PUT operations before committing the transaction to the MVCC;
6. Go GC Percentage - How much the heap has to grow to trigger the Garbage Collector;

Several factors share similar rationales for their selection. For instance, both **CPU Cores** and **Disk Configuration** are chosen with the expectation that increasing cores and enhancing disk quality will boost performance. However, we aim to quantify these improvements and assess whether the associated resource costs justify the performance gains.

As we observed in the first stage, examining the **Number of nodes** is crucial, particularly in a remote deployment scenario where message latency between nodes significantly increases compared to our previous local setup.

Then we have Etcd specific parameters, such as **Snapshot Count** and **Backend Batch Limit**. Etcd appends all key changes to a log file. It functions as a complete linear history of every change made to the keys. To avoid the indefinite growth of this file, etcd makes periodic snapshots providing a way to compact the logs by saving the current state of the system and removing old logs. The creation of snapshots can be expensive so this operation is only triggered after a predefined number of changes. We intent to study how this value affects performance. Etcd also has a threshold for the number of not committed PUT commands that are kept in memory before being committed to the MVCC. This threshold is configurable via the **Backend Batch Limit** parameter, and we also study how its variation affects performance.

Finally we have **Go GC Percentage** [4]. Go's GC uses the mark-sweep technique which, in short, means that it first marks the values it encounters as live and then sweeps all the memory that is not marked available for allocation. This process involves two resources: CPU time and physical memory. Increasing the value of 'GOGC' wil lead to less frequent GC, increasing the memory usage but decreasing the CPU usage (and vice versa).

A common criteria for all these factors is that they exhibit unidirectional behavior, meaning that as the level of the factor increases so does its effect. Some of the factors are obvious, increasing the number of CPUs will lead to an increase in performance/throughput, however factors such as **Backend Batch Limit**, **Snapshot Count** and **Go GC Percentage** are not so straight forward.

We assumed that those three factors do indeed have a unidirectional behavior and the reasoning for each is:

1. Snapshot Count - A lower number of snapshot count leads to more writes to disk leading to lower performance. It is also true that a high number will lead to less snapshots but with increased size, however we consider these to be lighter.
2. Backend Batch Limit - Lowering the backend batch limit causes etcd to write to the disk more often, which in turn worsens performance;
3. Go GC Percentage - A lower percentage for Go GC leads to more frequent garbage collection leading to lower performance.

Given the time and resource constraints, we could not completely confirm these assumptions, but we ran benchmarks varying only one factor at a time to get a rough idea of their directionality. It is known that these experiments can lead to wrong results if factors interact but it is better than blindly assuming the unidirectionality. The insight provided by the tests confirmed that the factors have this property. It also provided some additional information: low percentages (5%) of Go GC cause performance instability, with lower values actually negatively impacting performance.

With that insight, we got the following factor table:

Chosen Factor	**Batch Size**	**Snapshot Size**	**Garbage Collector**	**Disk Type**	**# Cores**	**# Nodes**
Letter	A	B	C	D	E	F

Table 1. Chosen factors and corresponding letters

Moreover, we tuned the synthetic workload's common parameters to replicate a real-life scenario more accurately.

1. Read-write ratio - 4:1
2. Read consistency - Serializable
3. Key size - 256 bytes
4. Number of clients - 1000
5. Number of connections - 100
6. Number of requests - 500000

3.3 Network Latency Bottleneck

During the first runs of the benchmark we also noticed the impact caused by the latency between the client and the cluster. Having the machines deployed in 'northamerica-northeast1-b' while using a local client (in Portugal) lead to significant loss of throughput. In the general case, etcd is usually deployed closed to the client (as in Kubernetes), so we decided to also deploy a machine in the same region as the cluster to act as the client and run the benchmarks.

3.4 Fractional Factorial Experiment

If we want to understand how all the different variables influence the outcome and get a complete picture of how they interact with each other, the best approach would be to conduct a Full Factorial experiment. However, this can be quite costly in both time and money, as even if we assume the unidirectionality of every factor and only run the experiments with two levels, we would have to run $2^6 = 64$ experiments.

An alternative strategy is to reduce the number of variables by conducting a Fractional Factorial experiment, where we confound a certain number p of factors out of the original k. In our case, we decided to choose $p = 3$, effectively reducing the number of experiments we had to run by a factor of 8 ($2^{6-3} = 2^3 = 8$ instead of the original 64).

In order to do so, we built a sign table based on the 3 "unadulterated" factors (A, B and C) and their interactions (A.B, A.C and A.B.C). Afterwards, we chose which of the interactions to confound with our remaining factors (D, E and F).

The most important part of this step was choosing which factors would be confounded and which would remain standalone, since a poor choice could lead to a significant interaction between the standalone factors, invalidating our remaining factors (since we can not differentiate confounded factors). With that in mind, we chose the following factors to be left standalone:

1. Batch Size (**A**);
2. Snapshot Size (**B**);
3. Go Garbage Collector Percentage (**C**).

Our rationale was that these factors should not have any significant interactions between each other, while the remaining factors should have a significant effect, leading to a good choice for our confounding.

This lead to the following sign table (note that A.B.C is not confounded since we only have 6 factors):

# Experiment	A	B	C	D (A.B)	E (A.C)	F (B.C)	A.B.C
1	-1	-1	-1	1	1	1	-1
2	1	-1	-1	-1	-1	1	1
3	-1	1	-1	-1	1	-1	1
4	1	1	-1	1	-1	-1	-1
5	-1	-1	1	1	-1	-1	1
6	1	-1	1	-1	1	-1	-1
7	-1	1	1	-1	-1	1	-1
8	1	1	1	1	1	1	1

Table 2. Sign table with 3 factors confounded

We picked values that we considered reasonably high and low to represent the best case and the worst case (1 and -1 respectively) for each factor in the experiments.

Adapting our original sign table to the chosen factors and levels, we get the following experiment table:

# Experiment	Batch Size	Snapshot Size	Garbage Collector	Disk Type	# Cores	# Nodes
1	100	1000	10	SSD	16	3
2	10000	1000	10	HDD	1	3
3	100	100000	10	HDD	16	6
4	10000	100000	10	SSD	1	6
5	100	1000	100	SSD	1	6
6	10000	1000	100	HDD	16	6
7	100	100000	100	HDD	1	3
8	10000	100000	100	SSD	16	3

Table 3. Experiments setup

4 Results

4.1 Fractional Factorial Experiment

Using our previous experiment table, we conducted our 8 experiments with 3 repetitions each. As a metric, we chose throughput in requests per second. This gave us the following results:

# Experiment	Run #1 Throughput	Run #2 Throughput	Run #3 Throughput	Throughput Average
1	23690	24389	24531	24203
2	18817	18553	18313	18561
3	24128	21884	23852	23288
4	22054	23014	23660	22909
5	24419	23787	24258	24154
6	23714	24471	24391	24192
7	22458	22594	22554	22535
8	25092	24990	25103	25062

Table 4. Experiment results (500k requests per run divided by time it took in seconds)

Using our original sign table and the experiment results, we used the sign table method to calculate each factor's effect coefficient and their total variation, which allowed us to quantify their impact and importance.

First off, we calculated each factor's effect using the standard 2^k design formula:

$$q_\alpha = \frac{1}{2^{k-p}} \sum_i y_i x_{\alpha i} \tag{1}$$

Afterwards, we calculated their variation using:

$$SS_\alpha = 2^{k-p} q_\alpha^2 \tag{2}$$

Following that, we calculated the total variation of our throughput using the total variation formula:

$$SS_T = \sum_{i=1}^{2^{k-p}} (y_i - \overline{y})^2 \tag{3}$$

This gave us $SS_T = 28362649.66$.

Lastly, we calculated each factor's importance using this formula:

$$\text{Effect of } \alpha = \frac{SS_\alpha}{SS_T} \tag{4}$$

This is represented in the following table:

	Batch Size	**Snapshot Size**	**Garbage Collector**	**Disk Type**	**# Cores**	**# Nodes**	**A.B.C**
q	-432.15	335.31	872.77	969.08	1073.15	-522.79	-346.91
SS	1494061	899460	6093803	7512899	9213181	2186455	962792
Effect	0.0527	0.0317	0.2149	0.2649	0.3248	0.0771	0.0339
Percentage	5.3%	3.2%	21.5%	26.5%	32.5%	7.5%	3.4%

Table 5. Effect calculation based on sign table method

Looking at the results, we can infer that the **Go Garbage Collector Percentage** (21.5%), the **Disk Type** (26.5%) and the **Number of Cores** (32.5%) have the biggest impact in the system scalability, with the **Number of Cores** having the **biggest impact** of all the factors.

The results also justify our choice of confounding. We can see that both Disk Type and Number of Cores have a high impact, having a significanly bigger impact than the underlying interactions they are confounded with (A.B and A.C, respectively).

On the other hand, we expected the **Number of Nodes** to have a significant impact on the system, since a bigger number of nodes should increase cross-talk and decrease the overall system's performance. Our experimental results do not reflect this, with this factor's effect being only 8%. This may also be due to our inability to test the system with a higher number of nodes, since google places a limit on the number of vCPUs a project can have, and while this limit can be altered, for large increases it has to be manually approved by google and ours was not.

Overall, we can affirm that the **Number of Cores** has a far bigger effect than the other factors, so we will use it as a focus while discussing the scalability properties of the system.

4.2 Universal Scalability Law

Scalability can be informally defined as the capability of a system, network or process to handle a growing amount of work. However, in order to objectively measure the scalability of a system, this concept needs to be quantified, since there are multiple factors that influence it in different ways. Taking that into account, we use the following properties:

1. λ - the performance coefficient, that represents the increase in performance by adding more nodes to the system.
2. σ - the serial portion, representing the sections of the system that are sequential and cannot take advantage of horizontal scaling.
3. κ - the crosstalk factor, representing the overhead caused by the communication among the different nodes in the system.

From our prior analysis, it became evident that **CPU Cores** exerted the most significant influence on system performance. Consequently, we have chosen it to study its impact using the Universal Scalability Law (USL).

As we are solely altering a single factor (CPU Cores), it is important to keep the other previously studied factors constant. The default values we adopted were those that demonstrated the most pronounced effect on performance, effectively simulating a best-case scenario.

1. Disk Configuration - SSD;
2. Number of nodes - 3;
3. Snapshot Count - 100000;
4. Backend Batch Limit - 10000;
5. Go GC Percentage - 100;
6. Compute Instance Type - n1-standard

An interesting deviation from our findings resides in the **Number of nodes** factor. Surprisingly, contrary to our initial Stage I benchmarks, reducing the node count from 6 to 3 displayed a negative impact on performance, a phenomenon that currently lacks a clear explanation. Furthermore, we intended to run tests with machine using up to 32 CPU cores and since we had a quota limit of 100 cores, using 6 machines would be impossible.

Using the previously described setup we ran several benchmarks measuring the time it took for the cluster to execute 500000 requests.

Using the tool provided by the course faculty, we computed the λ, σ and κ coefficients ($\lambda = 26396.6332445704$, $\sigma = 0.6160867027$ and $\kappa = 0.0013761883$) and plotted the lines as showed in Figure 2

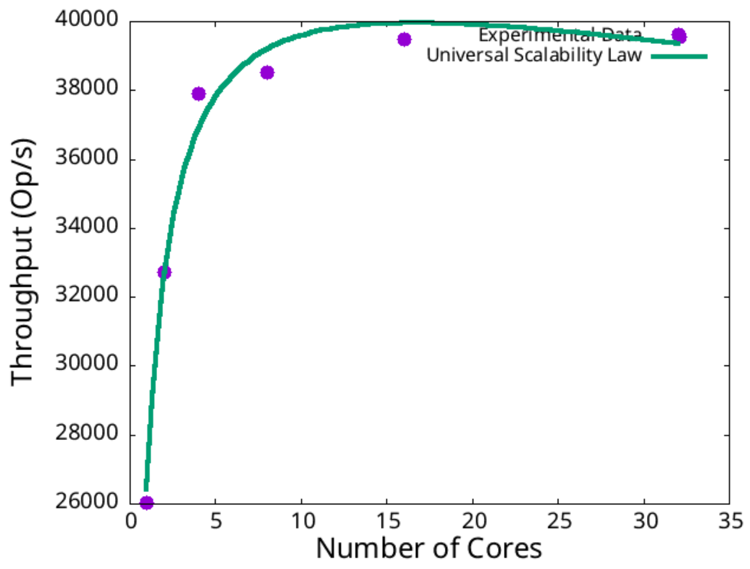

Fig. 2. Universal Scalability Law

It is clear that increasing the number of CPU cores will lead to an increase in performance. However, as discussed in class ideal parallelism is not always achievable as some portion of the work can only be executed serially. Essentially this graph shows what Amdhal's Law predicts, after a certain point, throughput does **not** increase by adding more CPU cores, in our benchmarks, that point is 16 CPU cores.

If we consider the prices for each n1-standard VM we can see that a n1-standard-8 VM costs 0.27€ while a n1-standard-32 VM costs 1.07€ (per hour). This is a 296.30% price increase for a 2.85% performance increase.

The plotted line attempts to fit the experimental results by generating a curve that, in our assessment, does not accurately mirror the real trend. This is because increasing the number of CPU cores does not necessarily result in a high cross-talk factor. There is no substantial increase in coherence delay between processors, suggesting that the curve should not peak and decline. Instead, it should tend toward a plateau or a flat line.

5 Conclusion

According to our findings, the most crucial factors in deploying an etcd cluster include the number of CPU cores, the disk type, and the Go garbage collector percentage. The first two factors are straightforward: better hardware generally yields superior results. Nonetheless, there is a threshold where investing more money might result in diminishing returns. Studying the ideal Go garbage collector (GC) percentage is a complex task that probably should not just be a measurement at a high and low value. Some companies, such as Discord, have written blog posts discussing this topic in detail [5]. The USL analysis results were a perfect example of Amdhal's Law and allowed us to do a price/performance study that is useful for companies trying to save on costs while keeping performance reasonably high.

References

1. Diego Ongaro and John Ousterhout, *In Search of an Understandable Consensus Algorithm*
2. Leslie Lamport, *Paxos Made Simple*
3. Etcd Documentation, *https://etcd.io/docs/v3.5/faq/*
4. A Guide to the Go Garbage Collector, *https://tip.golang.org/doc/gc-guide*
5. Why discord is switching from Go to Rust, *https://discord.com/blog/why-discord-is-switching-from-go-to-rust*