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

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

Abstract. This report studies the system etcd under the perspective of the Large Scale Systems Engineering course taught in Instituto Superior Técnico. We are mainly focused on studying its scalability and pontential bottlenecks inherent to a large scale deployment of the system.

Keywords: Key-value storage · Scalability · Performance · Distributed

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 `519cec60097af9b2013b053ee6bd2445846bb276`.

2 System Description

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

It's 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 pratical 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.

3 Performance Evaluation

3.1 Benchmarking Results

According to the project assignment we devised a benchmark and evaluated the system with it. Since etcd already has a benchmark suite, we picked the "txn-mixed" workload, where multiple clients issue transaction requests, each composed of multiple reads or a write. We also tuned the parameters in order to develop an appropriate synthetic workload, namely:

1. Read consistency - we tested both linearizable and serializable reads.
2. Key size - 256 Bytes.
3. Number of clients - set to 1000.
4. Number of concurrent connections - set to 100.
5. Total number of transactions - set to 500000.

Then, we took time measurements with the time command line utility.

```
# Serializable Reads
time benchmark txn-mixed --consistency s --key-size 256
 ↪  --clients 1000 --conns 100 --endpoints \$ENDPOINTS --total
 ↪  500000
```

```
# Linearizable Reads
time benchmark txn-mixed --key-size 256 --clients 1000 --conns
 ↪  100 --endpoints \$ENDPOINTS --total 500000
```

We ran these commands multiple times, with clusters composed of 1 to 11 nodes and computed the throughput per case. The resulting graphs are presented in Figure 2.

By analyzing the plots, we can conclude that in both cases the performance increases as we add replicas, maxing out at 3. Taking the pipeline into consideration, we can infer this increase is due to the smaller amount of client requests each node has to process. Since each node is limited to one CPU core, the gRPC

channel is easily saturated. By adding more replicas, since the client picks the replica it sends the request to in a round-robin fashion, the amount of requests is uniformly distributed among N channels (where N is the number of replicas). As this is the bottleneck for a smaller amount of nodes, throughput increases.

We can also observe that from 3 to 11 nodes, throughput decreases, this is caused by the amount of messages exchanged by Raft during the replication of the leader's log. Although the message complexity for this phase is only O(n) if the leader remains stable, it is enough to saturate each node.

3.2 Comparison with the 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.

Using the tool provided by the course faculty, we computed the λ, σ and κ coefficients ($\lambda = 9215.75$, $\sigma = 0.54$ and $\kappa = 0.047$ for figure 3 on the left, and $\lambda = 8546.83$, $\sigma = 0.56$ and $\kappa = 0.044$ for figure 3 on the right) and plotted the lines.

If we compare the coefficients obtained for linearizable vs. serializable reads, we see an increased performance coefficient in the latter, meaning there is a larger increase in throughput per node added (up until the maximum value). This is due to the weaker consistency required for serializable reads, since there is no need to do a consensus phase. The coefficients representing the serial portion and the crosstalk factor remain similar. Although the peak throughput obtained experimentally is slightly above the expected value in the Universal Scalability Law, the number of replicas where it occurs is the same ($\lfloor 3.13 \rfloor = 3$ for the plot in figure 3 on the left, and $\lfloor 3.17 \rfloor = 3$ for the plot in figure 3 on the right), therefore, the line is well adjusted to the experimentally obtained data.

4 Pipeline Analysis

To study the etcd pipeline we assume a request which requires a consensus step since this better represent the whole system. So let's assume a put request:

1. First, the 'etcdctl' command-line interface (CLI) in the client machine initiates a request directed to an 'etcdserver' within the cluster.

2. The 'etcdserver' runs an API responsible for forwarding this request to the 'Raft' module.
3. Inside the 'Raft' module, the protocol engages in message exchange with other 'etcdserver' instances to reach a consensus on the command.
4. Once consensus is achieved, the command is recorded in the log and subsequently relayed to the 'MVCC' (Multi-Version Concurrency Control) module. Concurrently it writes the new log to disk for fault tolerance.
5. The 'MVCC' module serves as the actual key-value store, often referred to as a Replicated State Machine. Here, it processes the command and generates a response, which is then sent back to the 'API' module. Concurrently it writes the value to disk for fault tolerance.
6. The 'API' module, in turn, communicates with the client, providing the appropriate response.

This pipeline is illustrated in Figure 4. The read pipeline (in Figure 5) is almost similar to the put pipeline but with fewer steps and with the optimization of having a fast path for serializable reads.

In our analysis of this pipeline's components, we initially operated under the assumption that the 'Raft' module constituted the primary bottleneck within the system. To validate this hypothesis, we employed the etcd benchmark tool, enabling us to selectively target the key-value store module by directing 'put' requests to the MVCC.

As anticipated, the system exhibited notably improved processing speeds for these requests in comparison to the prior runs. This outcome confirms our earlier conjecture, affirming that the Raft module indeed stands as the biggest contributor to system performance degradation.

5 Pipeline Optimization

Diving into pipeline optimization techniques, we will discuss the strategies already in place and what else could be added to further improve the system performance.

Our primary focus is directed towards the Raft module, as it emerges as the most apparent throughput bottleneck. Optimizing this module holds the promise of yielding the most substantial performance gains within the system.

5.1 Exploit common case

When conducting benchmarking studies on key-value stores, it's common to place a greater emphasis on read operations rather than write operations. This preference stems from the practical observation that in most scenarios, a system handles a higher volume of read operations compared to write operations. Consequently, it's important to establish a fast path that optimizes the common use case of read operations, ensuring efficient and rapid access to data. This fast path already exists, achieved by downgrading read operations from linearizable to serializable. This allows replicas to quickly respond to clients with local

MVCC values, improving read operation efficiency while maintaining reasonable consistency.

5.2 Concurrency

Enabling concurrent execution of multiple pipeline stages is a proven method for mitigating latency and boosting system performance. Yet, in the case of etcd, the pipeline inherently follows a linear progression, limiting opportunities for significant concurrency. The exception to this rule are the writes to disk which are done concurrently with the propagation of the ordered log and the response to the API module.

5.3 Batching

Batching, which involves grouping multiple requests into a single consolidated request, is a change for enhancing the performance of etcd. In theory, it's conceivable to aggregate requests from a single client while preserving their local order, thus circumventing the need for N separate consensus operations (N representing the batch size). This technique could be implemented at either the API module level or within the Raft module.

The original Raft paper [1] briefly mentions the potential for batching requests but doesn't offer specific guidance, leaving it as a topic for future exploration. Within the realm of batching, there are additional techniques that could be applied to further amplify performance, such as overwrites (e.g., a sequence of 'put' followed by 'delete' resulting in no net operation) or dallying (delaying a write with the expectancy that a delete will overwrite it).

It's important to note that batching introduces trade-offs. While it can enhance efficiency during high usage periods, it could lead to periods of inactivity during lower usage, and there's a risk of losing client requests in the event of a crash if the batch hasn't been persisted to disk.

6 Conclusion

Our results show that if we only consider performance, 3 replica clusters are ideal, but according to etcd's documentation [3], if we want to prioritize fault-tolerance, 5 replicas is the ideal amount, since it grants reasonable fault-tolerance, tolerating up to two member failures.

After our analysis of the request pipeline, we believe batching to be a strong candidate to improve the performance as the number of replicas increases, since it decreases the number of consensus instances, and subsequently the number of messages leading to a lower crosstalk factor.

As we suspected before analyzing etcd, the lack of scalability when new replicas are added proves its focus on achieving high consistency and fault-tolerance instead of high performance.

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/*

Appendix - Figures

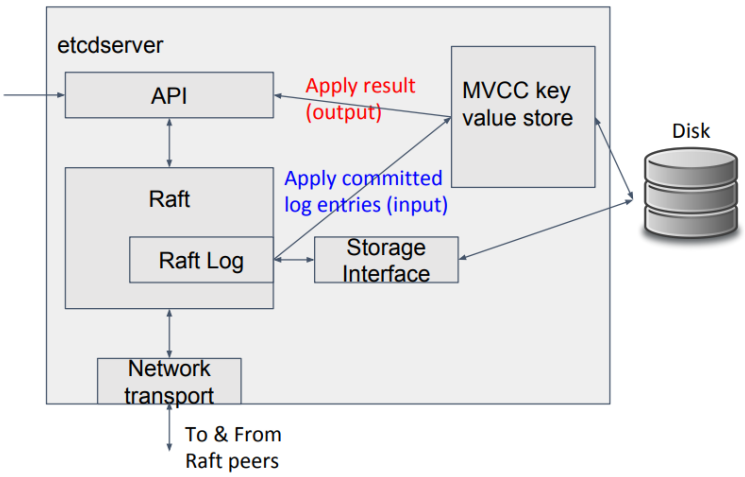
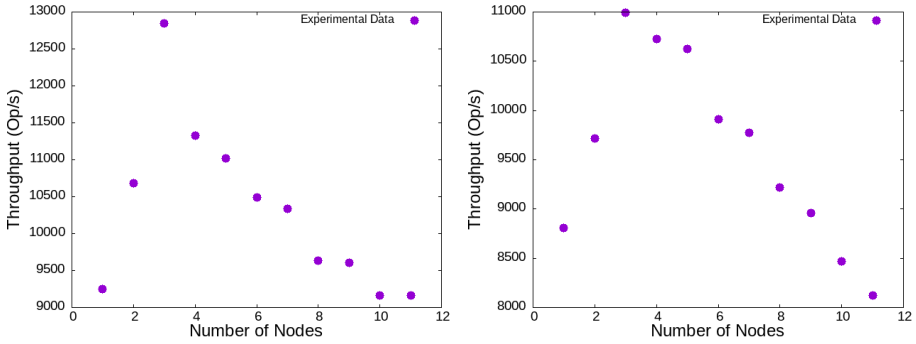
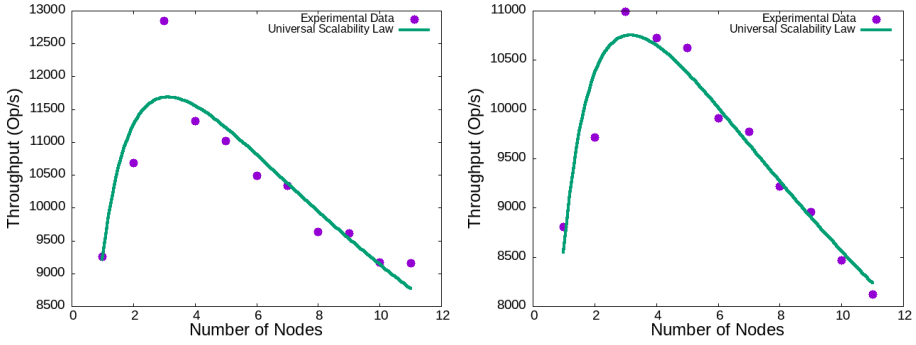


Fig. 1. Etcd system architecture



(a) Serializable Reads (b) Linearizable Reads

Fig. 2. Throughput per number of nodes



(a) Serializable Reads (b) Linearizable Reads

Fig. 3. Universal Scalability Law

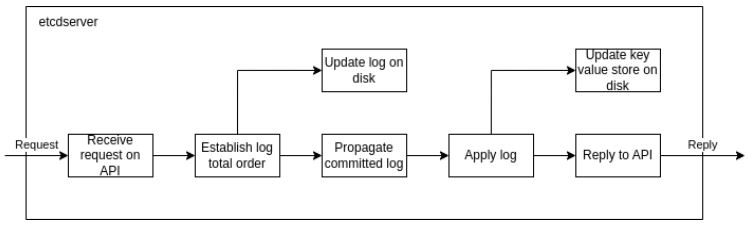


Fig. 4. Put request pipeline

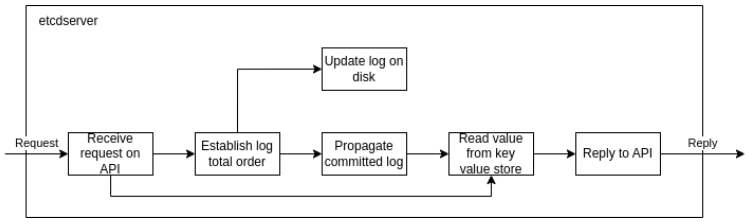


Fig. 5. Read request pipeline