Adaptation through Replica-group Reconfiguration in NoSQL Data Stores

A Thesis

submitted to the designated
by the General Assembly of Special Composition
of the Department of Computer Science and Engineering
Examination Committee

by

Dimitrios Valekardas

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WITH SPECIALIZATION
IN SOFTWARE

University of Ioannina
December 2018

Acknowledgements

I would like to thank my advisor professor Kostas Magoutis for sharing his knowledge on the thesis scientific field with me. Through his lectures, I was attracted to distributed systems and motivated to master replication systems in this thesis. Furthermore, I would like to thank Prof. Vassilios V. Dimakopoulos and Prof. Evaggelia Pitoura for agreeing to participate in my examination committee and for their comments on my thesis draft. I would also like to thank the open-source community for providing code repositories of widely used systems, giving researchers the ability to experimentally study, modify, and understand them.

Table of Contents

List of Figures

List of Tables

Abstract

Dimitrios Valekardas, M.Sc. in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, December 2018.
Adaptation through Replica-group Reconfiguration in NoSQL Data Stores.
Advisor: Kostas Magoutis, Assistant Professor.

Modern stateful services are able to adapt by dynamically adjusting the level of fault tolerance and performance to ensure that their service characteristics closely match user requirements, which may change over time. In this thesis we focus on adaptive stateful services using replicated NoSQL (key-value) stores for maintaining state, and on replica-group reconfiguration as the primary way to adjust the level of fault tolerance and performance delivered by each replica group in the system. We extend the state of the art in this field by considering the pros and cons of different ways to reconfigure a replica group and by experimentally evaluating reconfiguration variants in the context of the Etcd key-value store, based on the Raft algorithm. We experimentally study the costs and benefits of replica-group reconfiguration in various adaptation scenarios of practical interest, using an implementation of Etcd extended with the joint-consensus reconfiguration method, as well as the default single-server reconfiguration offered in the publicly available implementation. Our results demonstrate that the main adaptation costs incurred in both reconfiguration variants are transferring state and electing a new leader. Reconfiguration actions may be scheduled to reduce either the duration or the performance impact of reconfiguration. In both cases the two variants perform comparably, differing only in qualitative aspects such as implementation complexity.

Εκτεταμένη Περίληψη

Δημήτριος Βαλεκάρδας, Μ.Δ.Ε. στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Δεκέμβριος 2018.
Δυναμική Προσαρμογή μέσω Αναδιοργάνωσης Ομάδων Αντιγράφων στα Συστήματα Αποθήκευσης Δεδομένων τύπου NoSQL.
Επιβλέπων: Κώστας Μαγκούτης, Επίκουρος Καθηγητής.

Οι σημερινές μοντέρνες διαδικτυακές και μη υπηρεσίες που λειτουργούν ως μηχανές καταστάσεων, έχουν τη δυνατότητα να προσαρμόζουν δυναμικά το επίπεδο ανεκτικότητας σε σφάλματα όπως και το επίπεδο απόδοσης. Με αυτόν τον τρόπο, οι υπηρεσίες προσαρμόζονται στις απαιτήσεις και της ανάγκες του χρήστη, οι οποίες μεταβάλλονται κατά τη διάρκεια της ζωής και εξέλιξης ενός συστήματος. Σε αυτή τη διατριβή, εστιάζουμε στην δυνατότητα προσαρμογής ομάδων-αντιγράφων που χρησιμοποιούν ως αποθηκευτικό χώρο βάσεις κλειδιού-τιμής (τύπου NoSQL) μέσω χρήσης μηχανισμών αναδιοργάνωσης. Η αναδιοργάνωση (αλλαγή πλήθους αντιγράφων) είναι ο κύριος τρόπος να τροποποιήσουμε την ανεκτικότητα ενός συστήματος σε σφάλματα, αλλά και το επίπεδο απόδοσης των ομάδων-αντιγράφων που ανήκουν σε ένα σύστημα. Συμβαδίζοντας με την τελευταία λέξη τη τεχνολογίας, ερευνήσαμε τις δυνατότητες των μηχανισμών αναδιοργάνωσης σε ομάδες αντιγράφων, λαμβάνοντας υπόψιν τα πλεονεκτήματα και μειονεκτήματα των διάφορων τρόπων προσαρμογής. Μελετήσαμε και υλοποιήσαμε γνωστή παραλλαγή του μηχανισμού αναδιοργάνωσης, στο πλαίσιο κατανεμημένου συστήματος αποθήκευσης κλειδιών-τιμών, ονομαζόμενου Etcd, το οποίο βασίζει την αντιγραφή δεδομένων μεταξύ αντιγράφων στον αλγόριθμο συμφωνίας Raft. Μελετήσαμε πειραματικά τα κόστη και τα οφέλη της αναδιοργάνωσης ομάδων αντιγράφων σε ποικίλα σενάρια προσαρμογής μεγάλου και πρακτικού ερευνητικού ενδιαφέροντος. Μελετήσαμε τη δυνατότητα να προσαρμόζουμε την ανεκτικότητα του συστήματος μέσω προσθήκης νέων αντιγράφων, και τη δυνατότητα βελτίωσης της απόδοσης με αντικατάσταση των αντιγράφων με αντίγραφα που χρησιμοποιούν περισσότερους πόρους. Η μελέτη αυτή έγινε με χρήση της τροποποιημένης από εμάς έκδοσης του Etcd, η οποία χρησιμοποιεί την μέθοδο της από κοινού συμφωνίας αναδιοργάνωσης (joint consensus reconfiguration), με την οποία μπορούμε να αυξομειώσουμε το πλήθος των αντιγράφων μονομιάς. Η από κοινού συμφωνία αναδιοργάνωσης είναι ένα εργαλείο που μας επιτρέπει να αλλάξουμε το πλήθος μιας ομάδας αντιγράφων με μια εντολή, η οποία περιλαμβάνει πολλαπλές ταυτόχρονες προσθήκες/αφαιρέσεις/αντικαταστάσεις κόμβων. Μας επιτρέπει να μεταβούμε από μια οποιαδήποτε ομάδα αντιγράφων σε μια εντελώς καινούργια, ανεξαρτήτου πλήθους αντιγράφων. Μεταβαίνοντας στο νέο σύνολο αντιγράφων, προστίθεται ενδιάμεσα μια επιπλέον φάση στην οποία οι αποφάσεις λαμβάνονται από κοινού μεταξύ της παλιάς και της διάδοχης ομάδας αντιγράφων. Μελετήσαμε και αξιολογήσαμε και την προϋπάρχουσα μέθοδο αναδιοργάνωσης, η οποία αυξομειώνει το πλήθος της ομάδας-αντιγράφων κατά ένα αντίγραφο τη φορά (single server reconfiguration), και είναι διαθέσιμη σε δημόσιο ηλεκτρονικό αποθηκευτικό χώρο. Τα αποτελέσματά μας επιδεικνύουν τα βασικά κόστη αναπροσαρμογής, τα οποία προκαλούνται από τη μεταφορά της κατάστασης από τον αρχηγό της ομάδας αντιγράφων προς τους νεοεισερχόμενους κόμβους, και την πιθανή εκλογή νέου αρχηγού της ομάδας αντιγράφων, και είναι κοινά και στους δυο τρόπους αναδιοργάνωσης των ομάδων. Συμπεραίνουμε πως και οι δυο τρόποι αναδιοργάνωσης έχουν παρόμοια απόδοση, διαφέροντας μόνο στην πολυπλοκότητα υλοποίησης.

Chapter 1

Introduction

Adaptation in stateful services, such as a database, file system, or other distributed storage system, is defined as the ability to dynamically adjust the level of fault tolerance (resilience to one or more simultaneous faults) and performance (ability to handle a specific amount of load at a certain response-time) in such systems. Adaptation is an important way to ensure that the above defined non-functional service characteristics closely match changing user requirements and goals over time, rather than provisioning a system at deployment and then not being able to match its resources and characteristics (and thus, its cost) to the actual needs of its users. Dynamically adapting to potential faults (e.g., node crashes) and workload variations should be possible while any necessary changes (such as increasing the number and type of nodes) result to as little downtime as possible, minimizing financial costs and end-user disappointment.

Adaptation in stateful services is more challenging to implement and manage compared to stateless services (e.g., stateless web caching), especially in those stateful services that use data replication as a primary way to mask failures, as there is a need to prepare (transfer state) to new replicas before they can be operational. In this thesis we focus on the class of replicated NoSQL (key-value) stores as a foundation for maintaining service state due to their scalability and high availability properties, which have made them quite popular in recent years. Our main adaptation mechanism is *replica-group reconfiguration*, as a way to adjust the level of fault tolerance and performance delivered by each replica group in the system. Adjusting the level of data *sharding* (number of replica groups) and thus service parallelism in the system is another adaptation mechanism in such systems, however it is orthogonal (and complementary) to replica-group reconfiguration and beyond the scope of this thesis. Previous work on adaptation in replicated stateful services (Fitch [1], SMART [2], ZooKeeper [3] [4] etc.) has demonstrated the benefits of dynamically adjusting the level of fault tolerance and performance of replica groups. In this thesis, we extend the state of the art in this field by considering the pros and cons of different reconfiguration primitives and experimentally evaluating dynamic replica-group reconfiguration within a key-value store (Etcd [5] [6]) that is based on the Raft [7] consensus algorithm, a popular system that has not been evaluated in this context before. Our focus is on the costs and benefits of replica-group reconfiguration in various adaptation scenarios of practical interest (such as changing the number and type of replicas to improve fault tolerance and availability, to decommission replicas, or to "rejuvenate" a service by replacing its replicas with new nodes), using an extended implementation of Etcd with the Raft joint consensus mechanism. The implementation of Raft joint consensus was developed, tested, and evaluated as part of the work performed for this thesis.

Techniques for data replication are generally known for quite some time [8] [9] [10] and their implementations have initially focused on ensuring consistency and performance in *static* configurations. More recently, increased interest in dynamic adaptivity in distributed key-value stores was fueled by the variability of Internet workloads and as the notion of *elasticity* (dynamically adjusting system resources to workload needs) became a first-class system property in cloud environments. Replication techniques based on the Paxos [11] family of algorithms support dynamic reconfiguration of replica groups (SMART), however the impact of such reconfiguration depends on the specific technology of implementing Paxos and there is still no comprehensive experimental study that evaluates service availability and tradeoffs in reconfiguration primitives. Replication techniques based on the primary-backup model (ZAB [12], Raft [7] [13]) support dynamic reconfiguration and are more popular in real implementations, however there have been few studies on the methodology of how (and when) specifically to carry out such reconfigurations and what the corresponding performance impact is for the application. Another research question addressed is how a dynamic reconfiguration primitive that supports arbitrary replica-group changes in a single atomic operation compare to a simpler reconfiguration primitive that supports only single-server changes per invocation. This thesis contributes to better understanding the benefits, limitations, and tradeoffs in replica-group reconfiguration, with the objectives described next.

1.1 Objectives

The objectives in this thesis are:

- To evaluate different adaptation scenarios using a production-quality NoSQL data store that is able to dynamically reconfigure a replica group. Our aim is to allow any modification in the number and type of replicas (including replacing all current replicas with new replicas). Our focus is on the procedures involved in such a reconfiguration, namely the steps to plan the necessary changes depending on the goals (performance and availability of new system, and level of impact (latency, time) during reconfiguration), proactively prepare the new replicas, and effect the reconfiguration to move to a new system.

- To implement a reconfiguration algorithm that allows arbitrary changes in the replica group (rather than a single-server change at a time), termed *joint consensus*, in a widely used production distributed key-value store (Etcd) whose publicly available implementation features only single-server reconfiguration within the Raft algorithm. Our implementation adds joint consensus as a new reconfiguration mechanism (a new feature) in Etcd, while allowing the pre-existing single-server implementation as another option.

- To evaluate our implementation of Raft joint consensus in Etcd replica-group reconfiguration. There are few implementations of Raft today that implement and use joint-consensus for cluster reconfiguration, and no studies comparing it experimentally to the Raft single-server reconfiguration variant. We perform experiments that provide insight to the costs and benefits of replica-group reconfiguration in general (regardless of the specific reconfiguration mechanism) and comparing joint-consensus to single-server.

1.2 Thesis Structure

The rest of this thesis is composed of the following chapters: In Chapter 2, we provide background information on this thesis and explain how it relates to our work. In Chapter 3, we describe in detail the implementation steps that we made to support our experimental work. In Chapter 4, we perform a detailed experimental evaluation and analysis of our system under a variety of scenarios. Lastly in Chapter 5, we conclude and discuss open issues and future work.

Chapter 2

Background

In this chapter, we make a brief introduction and analysis of the Etcd system used in this thesis and the technologies it comprises. In Section 2.1 we provide basic information about the language Go, used to develop Etcd, and compare its key features to those of other popular languages. InSection 2.2 we describe Raft and related consensus algorithms that have been proposed in the past, with special focus on their reconfiguration mechanisms. In Section 2.3, we examine the features of Etcd and the architecture of other similar systems. In Section 2.4 we refer to the characteristics of the Etcd (as well as alternative) backends. Last, in Section 2.5 we discuss other work related to adaptivity through reconfiguration.

2.1 The Go language

Go is Google's open-source programming language, conceived by Robert Giesemer, Rob Pike, and Ken Thompson in September 2007 [14] and announced in November 2009. All inspection and profiling tools in Go are also free and open-sourced. Go is similar to the C language (some call it as C of the 21st century), but also borrows good ideas and functionalities from other languages. Go is ideal for building infrastructure and network servers but also useful for other programming purposes. Go is statically typed, guarantees memory safety, and makes an automated use of garbage collector. It is considered a light object-oriented language: It does not have the notion of a class, but gives programmers the ability to have an object-oriented approach to their code with the use of structs. Go structs and their associated methods serve the same goal of a traditional class and are obviously (although synonymous) more powerful than C structs. Structs only hold the state, and struct methods provide the behavior, enabling structs to change state. In addition, Go has its own distinct way for private and public data. If the name of a variable/struct/function starts with an upper-case letter is considered as package-public, else it is considered as package-private. Most programming languages like Java and Python were conceived in 90's (the "single-threading era"). Although they support multi-threading, problems (such as race conditions and deadlocks) often occur in concurrent executions. These problems make it hard to create multi-threading applications on these languages. If we create a thread in Java, it can consume approximately 1MB of the heap size and one can imagine what happens when a lot of threads are deployed. As Go was released in 2009, when multi core processors had already been in widespread production and use, it was designed with concurrency in mind. Go uses Goroutines instead of threads, an equivalent construct that consumes around 2KB of the heap size, so that millions of routines can be started. Goroutines also have quicker start-up than threads. In addition, they have growable segmented stacks which means that they use more memory only when necessary. Two or more goroutines can communicate easily with the use of gochannels. Gochannels can be thought as pipelines that connect different concurrent parts of a code that send/receive data. Directions can be set and define which pipeline edge receives and which delivers data. These differentiations make Go language a good choice for deploying distributed datastores such as Etcd.

2.2 Raft and related algorithms

In this section, we describe Raft and other related algorithms, with a special focus on their reconfiguration mechanisms. A consensus algorithm aims to achieve agreement, namely for a group of servers to agree on the sequence of operations to execute so as to also agree on the data values stored. Distributed consensus is needed to reach agreement between physical separate servers. The role of the algorithm is to guarantee that there will be no mismatch in the replicas of data stored across servers through appropriate synchronization between them. A consensus-based replication algorithm also provides the properties of fault-tolerance and high availability by being able to tolerate one or more server failures, maintaining the availability of the overall service.

Raft is a consensus algorithm proposed by Diego Ongaro in 2014. The purpose of Raft is to provide a replicated state machines (RSM) algorithm that is easier to understand compared to the classic Paxos consensus algorithm. Raft is also widely adopted and used in production systems as a reliable replication protocol that guarantees fault tolerance and consistency between participating servers. In Raft, servers can have one of three roles: Leader, Follower and Candidate. A single leader proposes to the followers, while followers respond whether they accept proposals or not. Based on whether a majority of follower accepts the proposal, the leader decides to commit. If a Follower senses that the current leader has crashed, it becomes candidate and proposes itself as new leader.

The sequence of agreed values on every node forms a replicated data log, stored on each node's disk. Communication between servers happens with asynchronous message exchanges. Each message is characterized by two numbers: The term and the index. The recipient of a message can understand if it is missing any data by looking at the index. If the current log index is higher than the message index, the node realizes that message is outdated (possibly a repeat of a past, already agreed, message). When a new leader is elected, it advances term to term + 1. The next messages that leader will broadcast to the cluster will have their term increased. Therefore, term is the way for a node to learn if it is missing cluster configuration or not.

Raft is similar in some ways to another consensus algorithm, Viewstamped Replication (VR) [15], with some unique features. Just as in VR, Raft elects a strong leader. This means that log entries always flow from the leader to the other nodes. Raft also

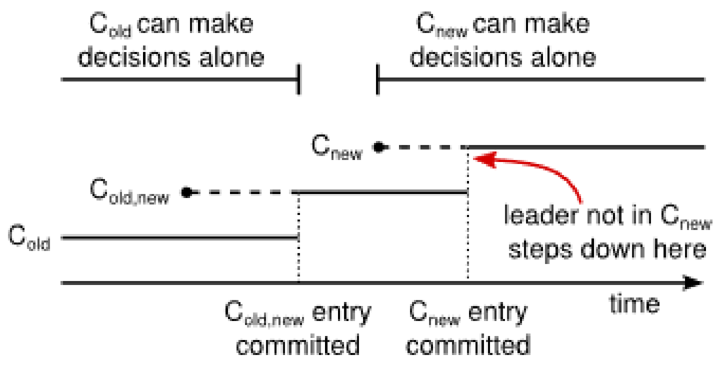

C_{old} can make decisions alone
C_{new} can make decisions alone
C_{new}
$C_{old,new}$
C_{old}
leader not in C_{new} steps down here
$C_{old,new}$ entry committed
C_{new} entry committed
time

Figure 2.1: Diagram explaining Raft joint consensus

uses randomized timers to elect leaders. The key difference is the way that membership changes happen. Raft uses joint consensus to change the existing configuration to a successor configuration. In joint consensus, there is a time period where the old and new configurations overlap (Figure 2.1), and as such, quorums from both configurations are needed for the common leader to decide. First, the leader proposes $C_{old,new}$ to the old and new configuration nodes. $C_{old,new}$ takes effect on each server, when it is applied to its log. When the leader receives quorums of responses from both configurations for $C_{old,new}$, it then proposes C_{new}. In the meanwhile, it needs quorums both of C_{old} and C_{new} to make decisions. Furthermore, a new leader may be elected under a joint consensus configuration. This new leader will need to be voted from quorums of both configurations to be elected. As soon as a server receives and stores the new configuration entry,it considers the new configuration to be in force. When leader proposes the C_{new}, it considers it to be in force (when the leader is part of the new configuration). C_{new} takes effect on followers as soon as it is seen. In case of the leader is not in the new configuration, C_{new} will take effect at commit rather than at the proposal. The leader will continue to replicate log entries without being leader of the new configuration until it receives a quorum of responses from the new configuration for C_{new}. Then the old leader will become follower, and new configuration will elect a new leader. In his PhD thesis D. Ongaro proposes a single-phase single-server variant of the reconfiguration approach, which was adopted in several systems. In this variant, a cluster can reconfigure only by adding or removing one server each time. When adding/removing one server, any majority of the previous configuration overlaps with any majority of the new configuration, thus a single phase suffices in that case, and the leader proposes Cnew to the nodes of the new configuration. Cnew is considered committed when a quorum of the Cnew nodes have applied it. The new configuration takes effect on each server when it is added to its log.

Paxos is a classic consensus algorithm proposed by Leslie Lamport. Comparing to Raft, it does not strictly require a leader and utilizes three classes of agents: Proposer, acceptor and learner. While always ensuring safety, some Paxos implementations additionally utilize a distinguished proposer (akin to a leader) to also ensure progress. When Paxos is not leader-based, every node is able to advocate a client request. A quorum of acceptors is required for a request to be committed. Learners implement the replication phase of the protocol. Once the acceptors have agreed on a client request, the learner agents execute the request.

Paxos can also be used to apply reconfiguration to the participating set of nodes (replicas). Let us assume that the current replica set consists of nodes A, B, C. Reconfiguration messages order to change the current configuration to A, B, D. If reconfiguration is proposed at slot *n*, reconfiguration will take place *a* slots later. The number *a* must be sufficiently big so that the leader will be able to make new proposals before consensus on reconfiguration is achieved (as the leader may not be part of the new configuration). If the number *a* is too big, it will lead to many unapplied proposals (to fill with no-ops). On the other hand, *a* small a may cause service unavailability. Thus, *a* must be appropriately chosen so that server logs have enough time to catch up with the most updated. Work in the SMART project pointed out that this technique comes with several vulnerabilities and hazards (consecutive reconfigurations, new configuration leader unawareness etc.). In SMART, when the configuration changes from A, B, C to A, B, D, it creates 3 *new* configuration replicas A', B', D which are part of configuration-2 (Paxos-2), while the previous replicas A, B, C which are part from configuration-1 (Paxos-1) are kept until the new configuration is established. If a node is in both configurations it will run two replicas (e.g., A and A'). Leader election (leader may not be part of the new configuration) does not have to deal with reconfiguration as soon as configurations are distinct (e.g., configuration-1 is distinct from configuration-2 even though they may overlap). We could say that SMART reconfiguration approach seems like the Raft joint consensus proposal of adding nodes that do not compete in making decisions, but only get up to date. When A', B', D are started, they acquire state to get up to date just like our learners (Chapter 3, Section 3.1). The time at which A', B', D are started must be such that the leader of the configuration-1 can make *a-1* proposals before it informs the clients to direct to configuration-2.

Viewstamped replication also offers a reconfiguration protocol. In VR, the system progresses through a sequence of views. In each view, one node has the role of the primary, and the secondary (backups) monitor it. When the primary node is out of order, the backups run a view-change protocol (election) to select a new node as primary. Like Raft, VR supports reconfiguration via Paxos [16]. When reconfiguration message arrives to the old configuration, cluster stops accepting client requests. The new configuration can start accepting messages when a majority of the new configuration nodes has completed state transfer. When it is done, system passes to a new epoch (term).

ZAB is a crash-recovery atomic broadcast consensus algorithm designed for Zookeeper. It is a leader-based protocol, where a primary process (leader) receives and executes the client requests and propagates the state changes to the backup (follower) replicas. Leader requires a quorum of responses (including itself) to make progress. The ZAB algorithm consists of three phases: discovery, synchronization and, broadcast. With few words, in discovery phase the prospective leader proposes a new epoch. When receiving positive responses of a quorum, it proceeds to synchronization phase. In this phase leader proposes new-leader message and on receiving quorum of responses for this message, it proceeds to broadcast phase. Normal operation happens in this phase, where client requests are served by the cluster and data is replicated to the ZAB nodes. In the beginning, ZAB membership changes happened using a "rolling-restart" fashion, where all servers are shut down and restarted in such way there will always be a quorum of running servers that will contain at least one server with the latest state.

This proved to be problematic and hard to execute, so they proposed a reconfiguration mechanism [17] that is very similar to Raft joint consensus. Primary (leader) sends state to the nodes of the new set (configuration) that have no state (similar to learners). When ready, primary proposes cop (reconfiguration) message to all the nodes. The proposals that arrive to the primary after the cop proposal require quorums of both sets. When cop is acknowledged by quorums from both sets, the primary (when member of the new set) activates the new set by sending an active message to backups. If the old primary does not participate in the new set, it keeps proposing requests scheduled after cop but it is the responsibility of the newly elected primary of the new set to determine commitment on these entries. This is an important difference with Raft, where a departing leader proposes and announces commitment of log entries beyond the proposal of C$_{new}$ and up to the commitment of C$_{new}$ at which point it steps down.

2.3 Etcd and related systems

In this section, we take a look into Etcd and related key-value stores. We mainly focus on Etcd, as it is the system which we modify but also refer the main functionalities of the other systems.

Etcd is a highly available distributed key-value store which provides reliability in cross-machine data storing. Its name comes from the "/etc" directory commonly found in UNIX variants, typically storing system setting files; the "d" stands for distributed. Etcd is an open source system and available in Github. There is a large active community that maintains the code and extends the abilities of the system. It is written in Go and uses the Raft consensus algorithm to provide consistency, fault-tolerance and highly-available replicated logs. Etcd uses BoltDB [18] as its backend storage. Etcd is widely used in production by several companies. For example, it is used by Kubernetes [19] [20] as backing store for all cluster data. While Etcd is commonly used for storing metadata, it can also be used to save other types of data. It is also preferred to be deployed on SSDs, but can also be on HDDs.

Etcd can modify the cluster size using single server reconfiguration mechanism. Reconfiguration goes through two phases: First, the cluster is informed about the new configuration. This happens with an API call (e.g. member addition), which returns when the cluster agrees on the configuration change. Second, the new member is started. It contacts the cluster and verifies if the cluster configuration matches with its cluster interpretation. When the member is started successfully, the cluster has reached the expected configuration. More than one node additions can take place with multiple single-server addition reconfigurations.

As for the data model, Etcd keeps revisions. Each time the key space is modified, Etcd increases its revision counter. Revisions become more useful combining with backend multi-version concurrency control (MVCC). It stores data in a multi-version persistent key-value store. This store keeps the previous version of a key-value pair when value is replaced with a new one. This means that through MVCC, we can view the evolution of a key-value store from a past revision.

Etcd has a broad API and can serve different types of requests, such as reads, writes, transactional writes (wraps multiple requests into one transaction), and deletes. Transactional writes are atomic compare-and-swap operations that can be used to build a distributed lock service. Etcd is tunable, and a user can adjust the system to its needs. Although most of the features that are referred below are automated, Etcd also provides an easy management API, with which a user can externally observe and manage the state of the cluster. Specifically, Etcd provides us the ability to modify cluster participants (add/remove server). We can also make several actions on the nodes like taking snapshots, key-space history compaction (keeps key information from a specific revision and forward) and defragmentation mechanism (release unused space -result of fragmentation- back to the system). A user can trigger leader election in the cluster and can specify the node that will be the next leader (leadership transfer). A user is able to get informed about cluster and node health or observe the future modifications of specific keys (watch command). Specifically, since Etcd maintains the history (revisions) of the keys, the watch command enables the user have access to their previous values. Other important features of Etcd are: distributed locks, leases, adding roles (user, root) and permissions to them.

TiKV [21] is a distributed key-value store that relies on the Raft consensus protocol. It eventually stores the data on a RocksDB backend. TiKV supports horizontal scaling by using sharding. Each shard partition is called region and every region is replicated to servers (replicas). Shard replication and load balancing is orchestrated by Placement Driver, an Etcd cluster, which also stores metadata such as key shard locations. PD does not replicate regional data. TiKV is part of TiDB [22] [23], a distributed SQL database, but can also be deployed as standalone system. TiKV is the NoSQL part of the system, while TiDB is the SQL part of it. TiKV is an open-source system with a Github repository, and written in the Rust language [24].

Like TiKV, CockRoachDB [25] [26] is a data store that supports sharding and bases its consensus mechanism on Raft algorithm, using a RocksDB backend. In CockRoachDB, shards are called *ranges*. Code is written in Go and hosted in a Github repository. Regarding CockRoachDB architecture, SQL statements are transformed to key-value operations and are distributed across the cluster. The distributed KV store communicates with CockRoachDB nodes. Each CockRoachDB node may comprise many physical devices, each holding one store. Each store may contain many ranges.

Ranges are replicated across the CockRoachDB nodes via Raft consensus algorithm.

Consul [27] [28] is a distributed key-value store that provides a data-center solution for service discovery across distributed infrastructure. It is written in Go and uses the Raft consensus algorithm. It differs from Etcd in that it does not support multi-version concurrency control in its back-end. Consul uses a gossip protocol to manage membership and to broadcast messages to the cluster. Gossip is provided through the use of Serf [29], which uses SWIM [30] gossip protocol. Consul make use of two different gossip pools: The LAN Gossip pool is used for client-server and client-client intra-datacenter communication. Clients use this pool to automatically discover servers. LAN gossip pool also helps to quickly and reliably broadcast events such as leader election. Failure detection is not performed by the servers, but is distributed via the gossip protocol as well. WAN Gossip pool is used for cross-datacenter communication as all servers participate in this protocol. WAN pool enables to perform cross-datacenter requests and also allows the datacenters to discover each other in a low-touch manner.

Last, Zookeeper in yet another distributed key-value store and coordination service, like Etcd, that uses replication for high availability. As referred previously, it uses the ZAB consensus algorithm. With regard to the data model, data in Zookeeper is saved in znodes. Znodes are organized in a hierarchical namespace fashion. Internal znodes (akin to directories) may also contain data, just like leaf znodes. The data that a znode contains are always relevant to it. A znode can have children etc. Each znode is associated with an access control list (ACL). Except for regular znodes, there are also ephemeral znodes, which live only as long as the session that created them exists. User can create/delete and get children of a node.

2.4 BoltDB and related backends

BoltDB is a key-value store developed in Go. It is characterized of rapid serving of read-intensive workloads. It is used by Etcd and other systems as their backend. Its code is open-source and available in a Github repository. BoltDB saves data in a memory-mapped file, creating a copy-on-write B+ tree [31] which enables the co-existence of multiple versions of keys. This simplifies read-write concurrency and locks are not necessary between writer and readers. Copy-on-write makes BoltDB reads rapid. BoltDB allows only one writer per time but there is no limitation for readers. When records are deleted from BoltDB, although the specific disk space is not any more used, it is replaced by *tombstones*, released back to the system during defragmentation.

RocksDB [32] [33] is another persistent key-value store implemented as a library in C++, available in a Github repository. RocksDB uses log-structured merge-tree (LSM) [34]. This tree structure is known to be more suitable for write-intensive workloads. The reason that RocksDB performs fast write speeds, is that writes are straightly written in a *memtable* placed in memory, periodically flushed to disk. The LSM tree on-disk structure consists of *sorted-string-table* files (SST). RocksDB being an implementation of LSM trees requires periodic compactions (briefly, SST files are merged, multiple copies (duplicates or overwritten) of the same key are removed, and the end result written to a new SST file), which operates (and costs) similar to defragmentation.

2.5 Adaptive reconfiguration of stateful services

Fitch is an infrastructure which is used both for stateful and stateless services. Stateful service replication system of Fitch can be compared to our work. Fitch key-value store is based on BFT-SMaRt [35]. BFT-SMaRt is a leader-based protocol and implements reconfiguration based on the ideas presented by Lamport *et al.* [36]. Their system uses three different reconfiguration command types: add, remove replace. Based on them, they perform experiments to check the impact of proactive replica additions/removals on stateful services. This kind of experiments can be related to our experiments described in Section4.6. They also use their reconfiguration mechanism to scale up/scale down we do in our evaluation in Section 4.4. Their experiments are related to our work; however, we carry out a more thorough investigation of the replica addition costs, different options on timing additions (one learner at a time or group additions), etc.

Chapter 3

Implementation

In this chapter, we describe the Etcd mechanisms that we modified and the code we produced in order to implement and evaluate joint consensus. The original Etcd implementation can only perform single-server reconfigurations. As proposed in Ongaro's thesis [7], joint consensus implementation should add new servers initially as non-voting members. In this thesis we term such members as *learners* and add them to a live cluster as described in Section 3.1. The core of our changes lay within the Etcd Raft package, where we modified the existing Raft algorithm to perform joint consensus as described in Section 3.2. Along with the algorithm, we modified all the necessary external files that the Raft package uses. We also extended the Etcd client API (etcdctl) as a necessary step to be able to command and control reconfiguration of the system. We also modified the Etcd benchmark Section 3.3 to enable the client to detect leader changes so as to always send writes directly to the leader, avoiding forwarding costs. Finally, we modified the benchmark to record throughput and average latency per second.

3.1 Implementing the learner role

Learners (Etcd nodes that maintain replica state but are not counted towards consensus decisions) are of key importance to our joint consensus implementation. In the original Etcd implementation we started with (v3.3.0-rc.0 as of December 20, 2017) [37] there was an initial implementation of learners, mainly within the Raft package. The original intention of Etcd developers (judging from activity in Etcd's Github repository) with the concept of learners was to use them as backup nodes that will be able to vote for a new leader, in case of the members cannot achieve consensus about the new leader. In our concept, learners cannot vote or to be voted as a new leader. They only listen to the decisions of the voting-nodes (members) of the cluster. Learners only build their state and do not take part in decision making. The leader does not take into consideration the progress of the learners in order to advance the committed index of the agreed log entries. When a learner is online, it establishes the connections with all nodes that are part of the cluster. The leader will keep trying to connect to a learner if it stops receiving heartbeats from it, but that does not inhibit progress since the quorum is not affected. Learners are the only way to change an existing cluster configuration to a completely different one, without affecting the previous quorum.

To expose learner functionality to the user, we extended the original Etcd implementation to allow an Etcd user to add a node as a learner, not just as member, via changes to the command line interface of the Etcd client. Initially, there was only the ability to add member nodes with the *etcdctl member add* command. In the standards of this command, we add learners via learner add command. We specify the endpoints of the nodes of the cluster followed by the learner add command, the name of the learner and its peer url (etcdctl –endpoints = *endpoint 1, endpoint 2, ... endpoint n* learner add *learner name* –peer-urls = *peer url*).

The general communication between a client and the cluster involves a *request* message followed by a *response*. By typing the above command, the client sends the url of the new learner (request) and waits for the id, name and endpoints of the new learner (response). Requests and responses are messages which are actually data structures. Messages obey the Google Protocol Buffer [38] format. These messages and their composition of corresponding fields are saved in suitable *protocol buffer* files. In these files, we compose the corresponding services which use the related messages, as well. Generally, all protocol buffer messages correspond to structures in Go code. The following code excerpt describes the messages for learner addition and the corresponding service definition:

```
service Cluster {
  rpc LearnerAdd (LearnerAddRequest) returns (LearnerAddResponse) {
    option (google.api.http) = {
        post: "/v3beta/cluster/learner/add"
        body: "*"
    };
  }
}

// ResponseHeader pre-existed. Necessary for better comprehension.

 message ResponseHeader {
  // cluster_id is the ID of the cluster which sent the response.
  uint64 cluster_id = 1;
  // member_id is the ID of the member which sent the response.
  uint64 member_id = 2;
  // revision is the key-value store revision when the request was applied.
  int64 revision = 3;
  // raft_term is the raft term when the request was applied.
  int64 raft_term = 4;
}

message Learner {
  // ID is the learner ID for this learner.
  uint64 ID = 1;
  // name is the human-readable name of the learner. If the learner is not started,
      the name will be an empty string.
  string name = 2;
  // peerURLs is the list of URLs the learner exposes to the cluster for communication.
  repeated string peerURLs = 3;
  // clientURLs is the list of URLs the learner exposes to clients for communication.
```

```
      If the learner is not started, clientURLs will be empty.
  repeated string clientURLs = 4;
}

message LearnerAddRequest {
  // peerURLs is the list of URLs the added learner will use to communicate with the
      cluster.
  repeated string peerURLs = 1;
}

message LearnerAddResponse {
  ResponseHeader header = 1;
  // learner is the learner information for the added learner.
  Learner learner = 2;
}
```

The definitions of the above code snippet are compiled with *protoc* and as result, encode, decode and other necessary functions along with the RPC of the service are auto-generated. One of the nodes that are already online, receives the learner addition request. Depending on the health of the cluster (e.g. inactive quorum) or other settings (e.g. learner id mismatch), the node accepts or declines the request. If everything is as expected, this node creates a configuration change message (*ConfChange* of type *ConfChangeAddLearnerNode*) and proposes the configuration change to all nodes of the cluster. This configuration message has been extended by us, and is located is appropriate protocol buffer file in Raft package (beyond code snippet).

```
enum ConfChangeType {
   ConfChangeAddNode = 0;
   ConfChangeRemoveNode = 1;
   ConfChangeUpdateNode = 2;
   ConfChangeAddLearnerNode = 3; //added by us
   }

message ConfChange {
   optional uint64 ID = 1 [(gogoproto.nullable) = false];
```

```
    optional ConfChangeType Type = 2 [(gogoproto.nullable) = false];
    optional uint64 NodeID = 3 [(gogoproto.nullable) = false];
    optional bytes Context = 4;
}
```

The request proceeds at the Raft level only if previous steps were error-free. In Raft, the Leader receives a proposal message (*MsgProp*) containing *EntryConfChange* field. Raft message types will be extensively analyzed in Section 3.2. During learner addition, no other configuration change is permitted. Existing members and learners receive a message to apply (*MsgApp*). In Etcd Raft, this message –like all the others– is managed just a simple log entry to be added. Followers and learners send back to leader a message applied response (*MsgAppResp*) and the leader checks whether the message has been applied in a majority of members or not. Despite the fact that the leader receives heartbeats and responses from all nodes, learners are always informed about everything but are otherwise ignored. When a quorum of followers (members) have applied the configuration change entry, all nodes apply the configuration change at the server level, update their cluster information, save the new learner in their stores and backends and try to establish peer connections with the new node. Etcd control client (etcdctl) then receives a successful response that new learner has been added and the following terminal message appears: "Learner *learner-id* added to cluster *cluster- id*".

The learner is now ready to start. The learner is started with etcd command, just like the members. Etcd developers had not predicted the insertion of learners into the cluster, and there is not separate way to start learners instead of members. For this reason, we define in code the names which correspond to learners and when a node starts, it learns whether it is learner or not. When online, it will be informed of the pre-existing members/learners of the cluster, save them into its store and backend, and establish peer connections with all of them. It will also save all the necessary data that describe itself. It will then receive replica state from the leader to bring the learner up-to-date.

In the same manner, we added the ability to remove a learner, with learner remove command. As input, we give the id of the learner node to be removed. We create the corresponding request and response messages and the service that implements the remove RPC. Just like learner addition, the same process of actions also stands for learner removal. A node receives the request from the client and decides whether to forward it or not. *ConfChange* message is composed again but of *ConfChangeRemoveNode* type this time. Leader receives *MsgProp* message and sends *MsgApp* messages to the rest of the nodes. Leader receives *MsgAppResp* messages and when the message is replicated to logs of a quorum of members, nodes update their cluster interpretation, delete the learner from their store/backend, and remove the peer connections and any other data about the deleted learner.

Overall, learner addition/removal mechanisms are quite similar to member addition/removal mechanisms. They both use the single-server configuration approach. We take advantage of single-server configuration to add/remove learners into the cluster. This is justified since a learner addition/removal will not increase/decrease fault tolerance ability of the cluster, as they are passive nodes. This is an unavoidable step to make some nodes part of the cluster without affecting quorum. In our implementation, quorum can be affected only when we pass from a small (less members) to bigger (more members) or smaller cluster via joint consensus. For example, we can have a 6-node cluster (consisting of 3 followers and 3 learners) which requires at least two-member votes and pass to another 6-node cluster configuration (6 members) where at least 4-member votes are required. Joint consensus enables nodes to change their "nature" and become members (followers/leader) from learners and vice versa. Further details for the joint consensus functionality and code implementation is described in Section 3.2.

3.2 Implementing the joint consensus mechanism

3.2.1 Design

The original Etcd reconfiguration mechanism supports only single-server reconfiguration, namely a user can add or remove one member at a time to or from an existing cluster respectively. To implement joint consensus, as we have already said, we firstly had to find a way to proactively start nodes that only build state and do not participate in consensus. For this reason, we insert learners as key part of the joint consensus reconfiguration, as soon as we need nodes that will be up-to-date when reconfiguration is ordered. By our perspective, it is meaningful to add learners into the cluster when there is a plan to replace existing members with new nodes. We then had to give system the ability to apply multiple reconfigurations at once. We developed suitable command and all the necessary parts of the respective RPC to serve multiple reconfigurations. In algorithmic level, we need an extra phase (proposal) that will order the cluster to move to a wider one, where decisions will be made commonly from the previous and the successor configuration. This is why we added extra message and role types and methods which serve this need.

Our joint consensus implementation mechanism enables the Etcd user to change from any configuration to any other it wishes, in a single shot, and as many times as it wants to. In order to implement joint consensus, we experienced several difficulties and challenges since original Etcd did not have the hooks to and enable multiple reconfiguration modes (single-server and joint) in the future. Nonetheless, we managed to implement joint consensus and overlay a twofold nature to Etcd enabling the co-existence (where a user can choose one or the other) of both single-server and joint consensus reconfiguration. The power of joint consensus reconfiguration is that it is both a scale-up (change node type and improve performance) and vertical-scale mechanism (increase or decrease fault tolerance with more or less members) depending on the needs of the system.

3.2.2 Protocol buffers, RPCs, and CLI

In order to implement reconfiguration, we had to develop the proper command line tool along with the respective protocol buffer messages necessary for the RPCs. We did not replace the existing reconfiguration mechanism but extended Etcd's reconfiguration capabilities by introducing the *reconfiguration* command in its CLI. From now on, by reconfiguration, we will mean joint consensus reconfiguration. The reconfiguration command syntax is: *./etcdctl –endpoints=<existing node 1 ip:2379>, <existing node 2 ip:2379>, ...<existing node n ip:2379> reconfiguration <node 1 id>, <node 2 id>, ... <node n id>*. In –endpoint argument, we can place any of the existing nodes endpoints, which are used from the client to send the reconfiguration command to the cluster. After the *reconfiguration* term, we list the ids of the nodes that will be part of the new configuration.

We developed proper protocol buffer message structs and extended the service that protoc uses, to generate the respective reconfiguration RPC and encode and decode routines, depicted in beyond code snippet. Client sends a *ReconfigurationRequest* message and waits for *ReconfigurationResponse* message.

```
service Cluster {
rpc Reconfiguration (ReconfigurationRequest) returns (ReconfigurationResponse) {
      option (google.api.http) = {
        post: "/v3beta/cluster/reconfiguration"
        body: "*"
    };
  }
}
message ReconfigurationRequest {
  repeated uint64 ConfIDs = 1;
}
message ReconfigurationResponse {
  ResponseHeader header = 1;
  repeated uint64 ConfIDs = 2;
}
```

The ids of the nodes in the new configuration are stored in an array of unsigned integers. The ids are sent on both the request and response messages. If reconfiguration is completed successfully, the message "*configuration changed to id1 id2 ...idn*" appears in the client terminal. One of the existing nodes delivers the reconfiguration order from client and directs it to the leader. It composes a *ConfChange* message of type *Reconfiguration* which also contains the new configuration member ids. In the code beyond code snippet, we note with proper comments the new fields in the existing message structures.

```
enum ConfChangeType {
   ConfChangeAddNode = 0;
   ConfChangeRemoveNode = 1;
   ConfChangeUpdateNode = 2;
   ConfChangeAddLearnerNode = 3;
   Reconfiguration = 4; //added by us
}
message ConfChange {
    optional uint64 ID = 1 [(gogoproto.nullable) = false];
    optional ConfChangeType Type = 2 [(gogoproto.nullable) = false];
    optional uint64 NodeID = 3 [(gogoproto.nullable) = false];
    repeated uint64 ConfIDs = 4 [(gogoproto.nullable) = false]; //added by us
    optional bytes Context = 5;
}
```

In the case of reconfiguration command, no action is taking place at the server level (connections are established, stores/backends are updated, every node is aware of the rest of nodes), it is used more as a link with the Raft package. The node which delivered the reconfiguration message, composes a new Raft message called *MsgPropRec* (message proposal of reconfiguration) which contains the ids of the nodes that compete in new configuration and delivers it to the leader. When reconfiguration is completed at the Raft level, all nodes are informed backwards and client get informed about the state of its reconfiguration request. In this implementation, reconfiguration is encountered only as a Raft internal process and main actions are taking place in Raft package. We can say that our code actions have given to Etcd Raft a semantic interpretation in message handling.

3.2.3 Message types and state transitions

Here is a short description of the new message types and states (roles) we inserted in Etcd Raft. All of the following message types contain the id of the nodes that are part of the new configuration. When a leader receives *MsgPropRec* (reconfiguration proposal), it sends *MsgAppRec* (apply reconfiguration message) to the followers/learners. When a node receives this type of message, it knows that it should pass in joint consensus configuration. These nodes respond with *MsgAppRecResp* (apply reconfiguration response message). Then, leader sends *MsgAppNewConf* (apply new configuration message) to inform the followers/learners that system passes to new configuration. Followers/learners respond with *MsgAppNewConfResp* (apply new configuration message response).

In etcd/raft/raft.go, where the core of our implementation lies, we extend the existing *StateType* map. StateType represents the role of a node in the cluster. We introduce the types of *StateJointLeader*, *StateLeavingLeader*, *StateJointFollower*, *StateLearner*, *StateJointLearner*, *StateJointCandidate* and *StateJointPreCandidate*. The full list of state types, including the new types we introduced, are listed in the beyond code snippet.

```
var stmap = [...]string{
   "StateFollower",
   "StateJointFollower",
   "StateLearner",
   "StateJointLearner",
   "StateCandidate",
   "StateJointCandidate",
   "StateLeader",
   "StateJointLeader",
   "StateLeavingLeader",
   "StatePreCandidate",
   "StateJointPreCandidate",
}
```

Depending on the state type, a node differentiates its actions. For example, the leader should take different actions when joint consensus reconfiguration is in progress compared to a non-reconfiguration operation. Every role (leader, follower, and candidate) has its own *step function*, which determines the message handling (*StepLeader*, *StepFollower*, *StepCandidate*). The Etcd developers did not create a step function for the learners to handle Raft-level messages and so we had to develop it (*StepLearner* function). In step functions, state types help the nodes choose suitable actions, depending on the configuration state (reconfiguration or not) and their role. In our implementation, reconfiguration results in type changing for all nodes. When the leader receives MsgPropRec message, it becomes leader under joint configuration and changes its state type from StateLeader to StateJointLeader. This state type changing happens by calling *becomeJointLeader* function. The same stands for followers and learners. When a follower receives a MsgAppRec message it changes its state from StateFollower to StateJointFollower by calling *becomeJointFollower*. Learner changes from StateLearner to StateJointFollower with becomeJointFollower, if its id is included in new configuration id list. When the id is not in this list, it changes from StateLearner to StateJointLearner by calling becomeJointLearner function. This differentiation exists because when a node is not part either from the old or new configuration, it is not allowed to compete in consensus when in transitional configuration state. When "Joint leader" knows that quorums of both configurations have become "joint", it

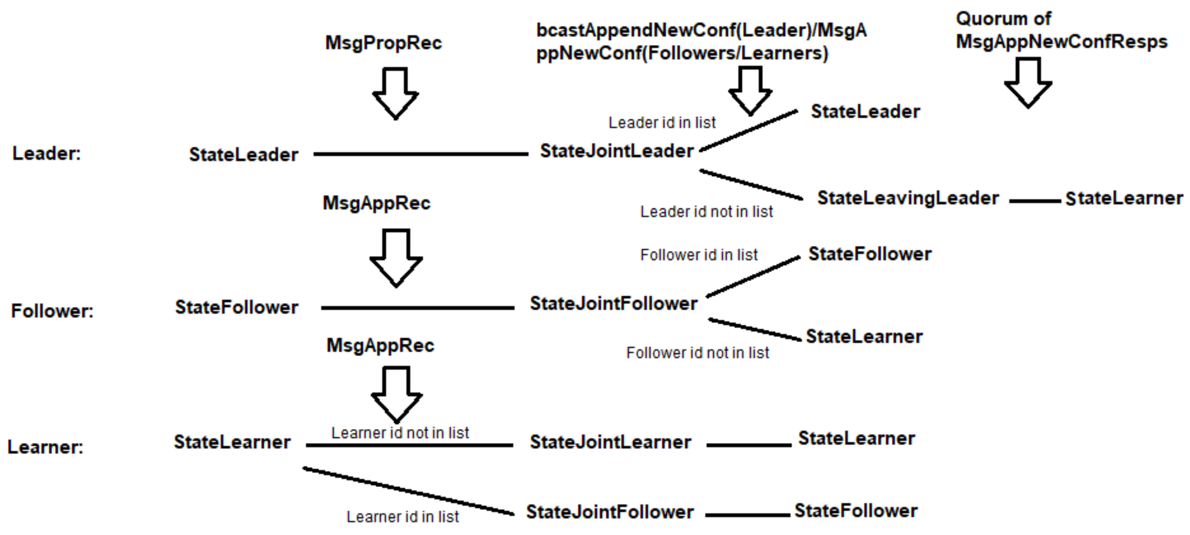


Figure 3.1: State-changing process with joint consensus

broadcasts MsgAppNewConf. Obeying in the Raft thesis, if leader participates in new configuration, it straightly changes state type to StateLeader, otherwise it changes to StateLeavingLeader. When it is sure that a majority of new configuration nodes have reverted their state to StateFollower, it becomes learner. To make this clearer, we depict the state-changing process in Figure 3.1.

If a candidate/precandidate node receives a message (MsgApp, MsgAppRec, MsgAppNewConf, MsgHeartbeat, MsgSnap), depending on its state, it steps into "joint" follower or follower. Since learners cannot become candidates, it means that candidates were not learners or "joint" learners previously. Therefore, they cannot change their state type to StateLearner or StateJointLearner. A candidate/precandidate can have one of these two states: StateCandidate/StatePreCandidate or *StateJointCandidate/StateJointPreCandidate*. If the leader becomes inactive while a reconfiguration is in progress, the follower node that decides to become candidate must change its state to StateJointCandidate. This type of candidate will expect to be voted by majorities of both the old and new configurations and proper state type helps that. We use the existing functions becomeCandidate and becomePreCandidate, where we embed suitable "joint" state type checks for the proper election format to be decided. Figure 3.2 depicts the process of changing from follower to candidate and vice versa. By the "Message" arrow, we mean all the type of messages that were referred to influence the role changes (candidate to follower). "MsgHup" is the type of message that a node receives when election timeout exceeds time limit and knows that it can be

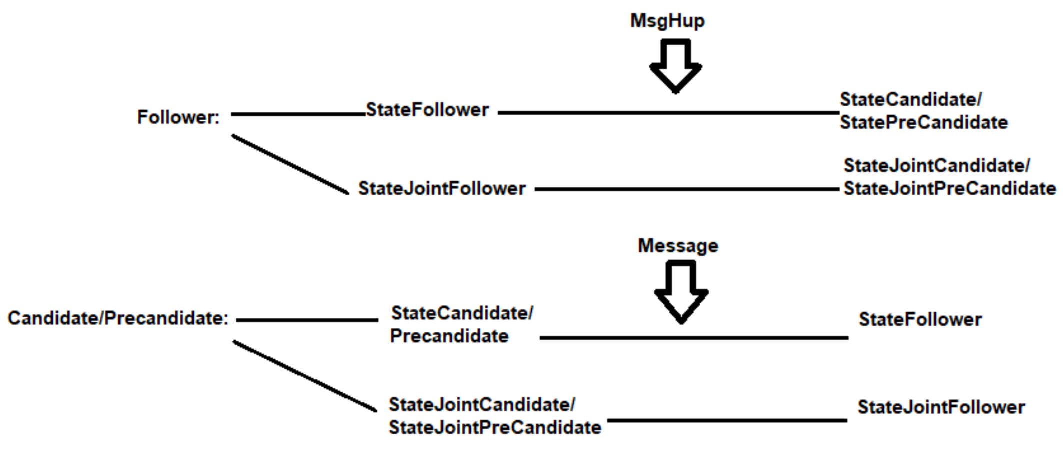


Figure 3.2: Process of changing from follower to candidate and vice versa

candidate/precandidate. Leader and learners ignore any MsgHup messages.

3.2.4 Operation of Etcd joint consensus

Taking a more detailed look in our implementation, as regards the election, Etcd has a function named *poll*, which is responsible for vote counting. Votes are saved in separate vote array for each candidate. We added a similar function *newpoll*, which counts the votes of the nodes whose ids are in the id list. Here, the votes of the successor configuration are saved in an array called *newvotes*. The candidate becomes leader when the votes are equal or more than the quorum of the nodes. When candidate needs votes from both old and new configuration, it becomes leader when the amount of the votes of the previous configuration members are equal or more than the quorum of previous configuration nodes, and the amount of the votes of the successor configuration members are equal or more than the corresponding quorum, as well. These are checked with *quorum* function which returns the quorum of the old configuration. We developed a function called *newquorum* which returns the quorum of the successor configuration. To sum up, nodes can change roles only through joint consensus reconfiguration. In case of candidates, we adjust the election mechanism functions and the roles of nodes to satisfy Raft joint consensus principles.

Now, let us take a look into the way that leader realizes that a quorum of nodes has appended a log entry into their own log. In his Raft thesis, Diego Ongaro proposes

Table 3.1: Example of match indices in maybeCommit function

Node types	Match indices
Leader	1537
Follower 1	1498
Follower 2	1520
Follower 3	1501

Table 3.2: Example of sorted match indices in maybeCommit function

Node types	Match indices
Leader	1537
Follower 2	1520
Follower 3	1501
Follower 1	1498

that leader should count responses of the cluster nodes in order to achieve consensus. When a quorum of nodes has responded to the leader for a specific log entry, the leader is sure that this entry has been replicated to a sufficient number of nodes. Etcd does things a bit different. For performance reasons, the leader does not count every single response for every single message. Leader saves its own progress and the progress of every single follower of the cluster in a map called *prs*, where each progress is a struct. Leader uses the *Match* field of the struct of each these nodes except for learners. Match is the index of the highest known matched entry. Leader sorts the match indices in descending order, including itself. At the end of the day, it always checks the Match index located in the quorum–1 position of the sorted slice. Having the slice sorted, we know that the number saved in position 0, is usually leader's Match index. If quorum is 2 (the case we have at least two nodes), we have to check the Match index that is saved in position 1 and so on. The actions referred so far take place in *maybeCommit* function located in etcd/raft/raft.go file. In the following tables, we give an example. Slice creation is based on prs map.

Leader sorts in descending order the indices as shown in Table 2. Quorum here is 3. Quorum - 1 position is 2 (counting from 0).

In this position we find Follower 3 whose match index is 1501. Then leader takes this index and calls a function called *maybeCommit* located in etcd/raft/log.go file. The name of the function gives an indication as to its action. Leader stores the index in which a majority of nodes have agreed in their logs, in a variable named *committed*. If the index of Follower 3 is bigger than the committed index, leader increments the committed index to Follower 3 Match index. If not, it means that no majority of nodes has yet applied their log entries up to this index. Nevertheless, cluster does not stop serving the new log entries to be applied when a leader cannot increase the committed index. Leader continues to send messages to the followers and all of them continue to increase their log entries. Thus, leader cannot serve reads for log entries that are not surely replicated to a quorum of nodes. In this way, Etcd achieves better responses to the client and higher availability comparing to the case it would count responses for each message. This particularity, forced us to adjust Raft joint consensus to the needs of Etcd.

Therefore, we create *maybeCommitJoint* function which is used by the leader when it needs to advance the committed index under joint consensus conditions. When leader receives the MsgPropRec message, it also saves the progresses of the nodes of the new configuration in a map called *newprs*, along with the map of progresses of the nodes of the old configuration which are saved in prs. It also saves the proposal index of the MsgPropRec. In maybeCommitJoint, leader gets the Match indices of both of the nodes of previous and the successor configurations which are stored in separate slices. The slices are also sorted in descending order. Just like maybeCommit, in maybeCommitJoint we get the Match index of the node in position quorum-1 of the sorted slice that contains the Match indices of the old nodes. Additionally, we take the Match index of the node in position newquorum-1 of the sorted slice that contains the Match indices of the nodes of the new configuration. Slices may overlap but this is not a problem. For example, if the new configuration is subset of the old configuration, there is a possibility for both of the Match indices to refer to the same node. In this case, the same node will be double checked in the log.go maybeCommitJoint function we developed. In this function, it is now more difficult for leader to advance its committed index, as it requires quorums both of the configurations. MaybeCommitJoint in log.go checks the following scenarios:

- MsgPropRec index is bigger than the Match index of the node positioned in quorum-1: This means that no quorum of the old configuration has received the message to proceed in joint consensus. Therefore, if the Match index of the node positioned in quorum–1 is higher than the committed index of the leader,

Table 3.3: Example of match indices in maybeCommitJoint function (old configuration)

Node types/ Old Conf	Match indices
Leader	1537
Follower 1	1498
Follower 2	1520
Follower 3	1501

Table 3.4: Example of match indices in maybeCommitJoint function (new configuration)

Node types/ New Conf	Match indices
Leader	1537
Learner 1	1490
Learner 2	1510
Learner 3	1487

leader will advance it. In any other case, it will not. Therefore, committed index can advance up the index before reconfiguration message is sent.

- If the MsgPropRec index is higher than the Match index of the node positioned in quorum -1 but lower than node positioned in newquorum-1 (quorum of old configuration nodes knows about the reconfiguration but not a quorum of new configuration nodes knows), leader will not increase its committed index.

- If the previous checks are passed successfully, it means that MsgPropRec index is smaller both of the Match indices of the nodes in the quorum–1 and newquorum-1 positions. Both of the quorums are aware of the reconfiguration message. If both of the indices of the quorum–1 and newquorum-1 positions are higher than the committed index of the leader, leader will increase its committed index to the smaller of them.

Let us check an example in the following tables: First slice is created based on prs map and the second one on newprs map.

After sorting the slices in descending order, Table 3.3 and Table 3.4 become:

If the leader's committed index is lower than 1490, then it will advance it to 1490.

Table 3.5: Example of sorted match indices in maybeCommitJoint function (old configuration)

Node types/ Old Conf	Match indices
Leader	1537
Follower 2	1520
Follower 3	1501
Follower 1	1498

Table 3.6: Example of sorted match indices in maybeCommitJoint function (new configuration)

Node types/ New Conf	Match indices
Leader	1537
Learner 2	1510
Learner 1	1490
Learner 3	1487

In summary, maybeCommitJoint acts not only as a joint consensus increasing committed index mechanism, but includes the role of maybeCommit, as well.

3.2.5 Message processing

We will now describe in more detail, the actions that leader, followers, and learners take when reconfiguration message arrives to the system.

- Leader receives MsgPropRec: If another reconfiguration is in progress (MsgProp or MsgPropRec), it does not promote the message to the rest of the nodes. In case there is no pending configuration, leader will fill the newprs map of progresses (based on prs and learnerPrs), become "joint leader" and send MsgAppRec, to all the cluster with *bcastAppendRec*. This function actually broadcasts the message by calling *sendAppendRec*. SendAppendRec is the function that is responsible for sending MsgAppRec message to a node. When learners are added into the cluster, it takes some time to build their state and warm-up. During this period, they cannot receive new messages. Messages can also be dropped, because of full sending network buffers. For these reasons, in bcastAppendRec we check if a quorum of nodes of the new configuration is active before sending the messages. Without this check, old nodes used to become “joint” followers and “joint leader” while the learners did not change their state to JointFollower or JointLearner. In other words, a part of the cluster was aware for the reconfiguration and the others had completely no idea. This is why learners are checked for being active before sending MsgAppRec messages to the cluster. If the learners are not ready to receive reconfiguration, leader reverts its state from StateJointLeader back to StateLeader. By this way reconfiguration is declined.

- Followers and learners receive MsgPropRec: Message is redirected to the leader.

- Follower receives MsgAppRec: Competing on the new configuration or not, follower’s state always becomes StateJointFollower. Just like leader, follower can retrieve the progresses of all nodes. It checks which of the node ids belong to nodes progresses that are stored in *prs* (leader/follower progresses) and which belong to node progresses that are saved in *learnerPrs* (learner progresses). Based on them, follower saves the node progresses of the new configuration in *newprs* (progresses of nodes that compete in new configuration). For any reason, if a follower has received the MsgAppRec previously, it should not update the newprs again. We can know this by its state type. If state type is StateJointFollower, it means that follower has already received the reconfiguration message. Message acceptance or rejection happens in *handleAppendEntriesRec*. Follower answers back to leader with MsgAppRecResp.

- Learner receives MsgAppRec: Learner does exactly the same actions like follower. The only difference is that if its id is contained in the new configuration list, it becomes “joint” follower, otherwise it becomes “joint” learner. When “joint” follower, node can vote and be voted if election occurs.

- Leader receives MsgAppRecResp: When leader receives MsgAppRecResp, it proceeds to message handling only if its state is StateJointLeader. When state is StateLeader or StateLeavingLeader, it means that MsgAppRecResp is outdated. If quorums of both configurations have applied the MsgAppRec (it is checked through maybeCommitJoint), it broadcasts MsgAppNewConf with *bcastAppendNewConf* which calls suitable function named *sendAppendNewConf*. Before broadcast, “joint leader” replaces the prs map content with newprs map content. This happens with the help of another map of progresses which is called *oldprs*. The progresses of nodes that are saved in oldprs are transferred to learnerPrs (followers that do not compete in the new configuration become learners). The newprs progresses are transferred in prs progresses map, and the other progresses are deleted. Learners which do not compete in the new configuration, progresses remain in the learnerPrs map. The learners who compete in the new configuration (they become followers), are deleted from learnerPrs map. When prs and learnerPrs maps are updated, leader changes its state from StateJointLeader to StateLeader or StateLeavingLeader. In case of receiving a rejection of the message, or sending has delayed before, it sends MsgAppRec message again to this node. Leader receives more than one MsgAppRecResp response, though. With suitable checks, leader remembers whether it has broadcasted MsgAppNewConf previously. If it has, it will not broadcast again.

- Follower receives MsgAppNewConf: In case a follower did not receive the MsgAppRecMessage previously, this follower did not fill the newprs map of progresses and should fill when it receives MsgAppNewConf. When newprs is up to date, follower should replace the prs progresses with newprs progresses. This happens the same way the leader does, as described before. When prs and learnerPrs maps are updated, follower changes its state from StateJointFollower to StateFollower or StateLearner. Then it sends back to leader MsgAppNewConfResp with *handleAppendEntriesNewConf*.

- Learner receives MsgAppNewConf: The actions and message handling that a learner does when it receives MsgAppNewConf, are exactly the same with what follower and leader make, as well. The only difference, is that learner changes its state only from StateJointLearner to StateLearner because if it competed to the new configuration it would have changed its state type to StateJointFollower when it received MsgAppRec message.

- Leader receives MsgAppNewConfResp or MsgAppResp: At this point, things become a little complex. When we stressed Etcd with benchmark using a lot of clients, because of Etcd batched-alike responses, leader sometimes did not execute MsgAppNewConfResp code and although that new configuration was applied to the proper nodes, leader never learned it. This resulted the leader to remain "leaving leader" while it should change its state. It also happened that leader changed its state from StateLeavingLeader to StateLearner (depending on its participation to the new configuration or not), before a majority of nodes that participate in new configuration was informed to pass in new configuration (they had not changed their states yet). The actions of state type changing always happen in a different part of code. Although in maybeCommit/maybeCommitJoint calls leader knew when it should advance the committed index, it did not know when it should revert its state. Therefore, we needed a way for the leader to punctually change its state, not earlier or later. For these reasons, we decided to count responses of nodes that belong to the new cluster. When leader is under workload (writes), it sends MsgApp messages, receives MsgAppResp responses. Because as we said, not every single response is sent back to leader, leader may receive not all of the MsgAppNewConfResp messages or even worse none of them. This is why we do not count responses of MsgAppNewConfResp message types, but check for responses that their index is equal or bigger than the MsgAppNewConfResp message. This adaptation means that we adjusted leader to count such responses not only in MsgAppNewConfResp message handling but in MsgAppResp message handling, as well. If a node rejected the MsgAppNewConf message or sending has delayed before, leader will send MsgAppNewConf to it again. Additionally –because we do not know in which message handling, "leaving leader" will be sure for the new configuration transition– the leader can revert its state from StateLeavingLeader to StateLearner not only in MsgAppNewConfResp but also in MsgAppResp message handling. MaybeCommit and maybeCommitJoint function calls are also influenced by these adaptations. If leader receives MsgAppResp, and no reconfiguration is in advance (leader's state type is StateLeader), it acts as it would do before our code changes (leader calls maybeCommit). If reconfiguration is in advance (leader's state type is StateJointLeader), leader will call maybeCommitJoint. If leader's state type is StateLeavingLeader, it will call maybeCommit (new progress structures are updated before MsgAppNewConf broadcast). Here, maybeCommit function does not take into consideration leader's Match index, because leader does not participate in the new configuration. In MsgAppNewConfResp handling, when leader receives MsgAppNewConfResp and its state type is StateLeavingLeader, it means that leader has not yet received a quorum of responses with higher index than the MsgAppNewConfResp and thus reconfiguration is not yet completed. So, it continues to replicate log entries to the followers/learners without being leader of the new configuration. It also keeps observing the Match indices of the new configuration nodes with maybeCommit function in order to advance the committed index, despite of not being in the new configuration. Otherwise if leader has changed its state type to StateLeader (is part of the new configuration), it also uses maybeCommit, but now Match indices include the leader as well. In addition, it occurred that leader did not receive the MsgAppRecResp messages, as well. As we have already said, when leader receives MsgAppRecResp and majorities of both configuration nodes have received the reconfiguration message MsgAppRec (checked with maybeCommitJoint), it broadcasts MsgAppNewConf. Because of the Etcd batched-alike response implementation, leader may first receive a MsgAppResp message from a node. Quorum requirements may be met here (maybeCommitJoint), so leader has to broadcast MsgAppNewConf. If MsgAppNewConf is broadcast, it will not be broadcast in MsgAppRecResp/MsgAppResp handling again, if such message arrives to leader. We do not need equivalent response count mechanism for MsgAppRecResp and MsgAppResp messages. Leader first fills the newprs progress map and changes its state to StateJointLeader and secondly sends MsgAppRec messages to the rest of nodes. If cluster is not ready to "adopt" reconfiguration, leader reverts its state back to StateLeader. On the other hand, leader does need to be sure that a quorum of the new configuration nodes has agreed to proceed to the new configuration, in order to changes its state from StateLeavingLeader to StateLearner. When state is changed StateLearner, followers will realize the absence of the leader and the cluster will be forced to new election. Old followers and old leader -now learners-, will find out about the new leader and will increase their term like all the others. They are still part of the cluster, without obligations and rights. User can completely remove them and because they are learners, no node will care about that.

Concluding, the fact that leader may not receive the MsgAppNewConfResp messages and rely on the later MsgAppResp messages to change its state, means that old leader will keep on replicating the log entries to the new configuration for a little longer period (few log entries) than it should. This is not a problem, though.

3.2.6 Discussion

The weakness of this implementation is that cluster interpretations at server level are not updated. When the learners become followers and vice versa, each node at server level remembers the old cluster configuration. Although nodes know the truth at Raft level, it is not propagated to server level. For example, when we add a new learner after a reconfiguration in which the old followers have become learners and the reverse, incoming learner sees that the previously learners –now followers– and previous members (followers and leader) -now learners- are misplaced still in learner and member list respectively. Based on them, it builds incorrectly the prs and learnerPrs maps of progresses. We tried to solve this but it required import cycles, which are not allowed in Go. In order to be able to propose multiple reconfigurations for our experiments, when leader send MsgAppRec message, it passes in the message fields the ids of the nodes included in prs map, and the ids of the nodes included in learnerPrs map. When the new learner receives MsgAppRec, based on these lists, it corrects its own maps.

Message types, state types, role types maybeCommit/maybeCommitJoint functions, broadcast functions, all inter-related and bounded by the proper checks and restrictions which joint consensus requires. We managed to implement it according to Raft principles after significant development and testing effort. We stress-tested Etcd with several benchmark runs under different number of clients. Stress testing the joint consensus implementation led us to several code improvements that eventually resulted to a stable, robust system. Our joint consensus implementation takes about 1500 new lines of code.

3.3 Extending the Etcd benchmark

The standard benchmark that came with the Etcd implementation required changes to support the following requirements:

- Always find and target the leader node in a replica group, even when that node is not part of the initial configuration
- Report latency and throughput during a benchmark run

By default, Etcd benchmark uses the endpoints specified by the user via the command line interface (CLI). The user can specify whether the benchmark will hit directly the leader or the load will be distributed to several nodes. Nevertheless, the final receiver of the requests is the leader, as Etcd is a leader-based system (Raft uses strong leader). However, the current implementation is not able to redirect writes to a new leader after a reconfiguration action, relying instead on Etcd's ability to forward to the new leader. In addition, the original Etcd benchmark produces a response time histogram, latency distribution (expressed as percentiles) and statistics like average latency, average throughput etc.

In our implementation, we modified the functionality of the benchmark under a write-intensive configuration (benchmark put) as follows: the benchmark client is now able to detect the address of the new leader of the cluster, and ignore the endpoints that were initially given by the user. When the benchmark starts, it establishes connections with the given addresses. Within two requests (writes in our case), time interval is checked and if it exceeds a certain threshold (1.5 sec), the client checks whether the leader endpoint has changed. We fetch the member and learner list with suitable RPC calls from the client to the cluster and check if the new leader's id, matches with one of the ids fetched by the lists. The client thus finds out which node is the new leader, re-establishes connections with the new leader, and continues to stress the cluster with the workload.

The motivation to modify the benchmark client was due to the following problem: When we passed from one configuration to another and leader election took place, if we specified the target to be the new leader and that leader was not part of the previous configuration, the new leader would not be in the specified starting endpoints. If we set the addresses of the learners in the beginning, which were not yet active, the benchmark would try to establish connections with inactive nodes and crash. If we did not set the leader as target, the client had to use the endpoints of the old leader or the old members which led to increased latency after reconfiguration, as writes had to reach the leader indirectly (through learner).

We also adjusted the benchmark as follows: We record the latency and throughput per 1 sec interval. We firstly record the time in the beginning of the request production and take the time again when the next request is ready to be sent to the cluster (the benchmark is "closed loop", namely a client does not issue a request unless the previous request has completed). As long as we are within a specific 1 sec interval, we keep on summing up the response time of all requests so far. We also count the number of requests served within the interval. When time interval is equal or exceeds the 1-sec time limit, we divide the sum of response times of the served requests by the total number of operations and get the mean latency for this interval. Each 1-sec interval mean latency is saved into an array (a time series). By counting the number of served requests, we know the number of requests per 1 sec time interval. This counter is saved to another array where throughputs per each interval are stored. This process is repeated for all 1-sec intervals until the benchmark ends. When all requests are finished, these two arrays are written to two files, latencies.txt and throughputs.txt, respectively. With the same logic, we also record the 99th and 99.9th latency percentiles of each interval.

In the experiments reported in Chapter 4 of this thesis we mainly depict and study latency plots drawn from the records of latencies.txt, but during our investigation we have studied the corresponding throughput plots as well.

Chapter 4

Evaluation

In this chapter we describe the experimental evaluation of our joint consensus reconfiguration mechanism in etcd in a number of scenarios. We start in Section 4.1 by describing the experimental testbed, benchmark settings, and measurement methodology throughout the experiments. In Section 4.2 we take a closer look at the internal workings of the joint reconfiguration implementation under different load levels by examining Raft logs across replicas. In Section4.3 we observe how snapshot, log & history compactions and defragmentation work together. In Section 4.4 we demonstrate the power of replica-group reconfiguration in dynamically adapting stateful replicated services by scaling up a 3-node etcd cluster through increasingly more powerful node types. Section 4.5 describes how reconfiguration can be used as a tool to both increase performance and fault-tolerance. In Section 4.6 we characterize the costs of proactive reconfiguration in terms of adding nodes first as learners in preparation for a full reconfiguration action. In Section 4.7 we examine the performance impact of the back-end system. In Section 4.8 we describe regression-based prediction of periodic performance hot-spots, which –if predicted accurately– could be masked by reconfiguration actions. The methodology presented in this section is general and could be applied to other prediction needs. In Section 4.9 we compare single-server and joint consensus reconfigurations, along with different reconfiguration policies.

4.1 Experimental setup

Our experimental testbed consists of 15 servers, each equipped with a dual-core AMD Opteron 275 processor clocked at 2.2GHz with 12GB of main memory. All servers run Ubuntu 16.04.4 64-bit with a 4.4.0 Linux kernel and are interconnected via a 1Gb/s Ethernet switch. Servers used as etcd nodes have a base 72GB 10,000 RPM SCSI drive, and an additional 300GB 15,500 RPM SAS drive dedicated to storing data. All hard drives are formatted with ext4.

The main benchmark used in our experiments is the etcd benchmark provided by the developers of etcd in their GitHub repository. In this evaluation we focused on an all-writes (100% writes) workload with the ***benchmark put*** command, known to be placing significant resource strain on an Etcd replica group. Writes involve all replica nodes whereas reads are served only by the leader node. Our main metrics are response time and throughput measured at the client, and CPU resource usage at the leader replica (when not otherwise specified).

We varied etcd benchmark's argument values depending on the goals of each experiment and problems faced. Argument values that are varied in each experiment are always explained at the outset. All experiments use 8-bit keys and 256-bit values. The order of keys is always sequential. We perform experiments with various concurrency levels, number of total writes, and key space sizes. In experiments that contain reconfiguration, we have leader detection (Section 3.3) enabled. CPU utilization is measured by monitoring CPU idle % via the *top* command, which was set to record idle percentages for periods of time that cover the etcd benchmark runs. Record in-

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

Figure 4.1: EWMA formula

tervals are 1 sec. The exact command used is: top -b -d1 –nX | grep Cpu | cut -c 37-40 > cpuusagenode.txt where X is the total duration of the recording in seconds, and node is the node that is monitored. We convert to CPU busy % by transforming (i.e., 100 - idle) for each record.

We also removed fsync() from the file path of all replicas. In this way, the leader is not certain what log offset is recoverable from follower disks. Thus, a follower that crashes cannot recover from disk; instead, it is considered a new addition and receives full state before entering service. Our motivation to remove fsync() was to reduce the measurement noise that the disk introduces when in the critical path of performance.

Early in our evaluation effort and after examining several results we decided that smoothing time series using Exponentially Weighted Moving Average (EWMA) [39] helps in depicting key phenomena that are obscured by noise when plotting the raw data. Our EWMA transformation is based on the following standard formula:

Computation of the new observation St in each step is based on the actual observation Yt weighted by a constant α (always < 1). Then it is added with the value of St-1 which is computed in the previous step, weighted by $1 - \alpha$. In our experiments, we set α □ 0.125. The first observation of every experiment, is always weighted with 1 - a, in order to converge quickly to steady state values.

4.2 Insight into reconfiguration process under different load levels

The operation of the joint consensus reconfiguration process differs depending on a number of factors. The duration of reconfiguration and number of committed entries vary depending on the client load (proposal rate). To examine that, we perform reconfigurations that change the cluster to a completely new configuration. We consider the following cases:

- Measure overall time during reconfiguration, from the point of proposing $C_{old,new}$

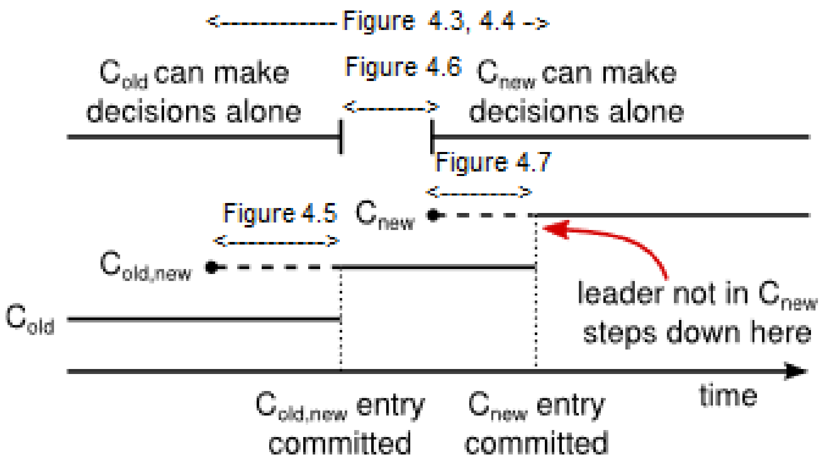


Figure 4.3, 4.4
C_old can make decisions alone
Figure 4.6
C_new can make decisions alone
Figure 4.7
Figure 4.5
C_new
C_old,new
C_old
leader not in C_new steps down here
C_old,new entry committed
C_new entry committed
time

Figure 4.2: Correspondence of results to the timeline of Raft joint consensus

to the point of finding out that the proposal for C_{new} has been committed, and the number of log entries that have been committed during this time interval (Figure 4.3, Figure 4.4)

- Measure the number of pending entries that are committed under C_{old} configuration. We measure from the point the leader has proposed the $C_{old,new}$ (MsgAppRec message) (Figure 4.5) till the point of $C_{old,new}$ commitment

- Measure the number of log entries that are committed between the log index at which the leader learns that $C_{old,new}$ is committed, to the index at which it proposes C_{new} via the MsgAppNewConf message (Figure 4.6). These entries are committed under $C_{old,new}$

- Measure the number of committed log entries from the point where the leader proposes C_{new} to the point it learns that it has been committed (Figure 4.7). The leader's state at that time of C_{new} proposal changes from StateJointLeader to StateLeavingLeader. These entries are committed under C_{new}. We consider all above cases at three different load levels, serving 1, 10, or 100 concurrent clients.

In Figure 4.3 we compute how long a reconfiguration proposal takes to be finished (from proposal creation till the commitment of the new configuration). In Figure 4.4 we record the amount of committed log entries during the reconfiguration time interval. All computations take place in leader and are based on suitable prints on leader's terminal. In these experiments we observe the relation between client con-

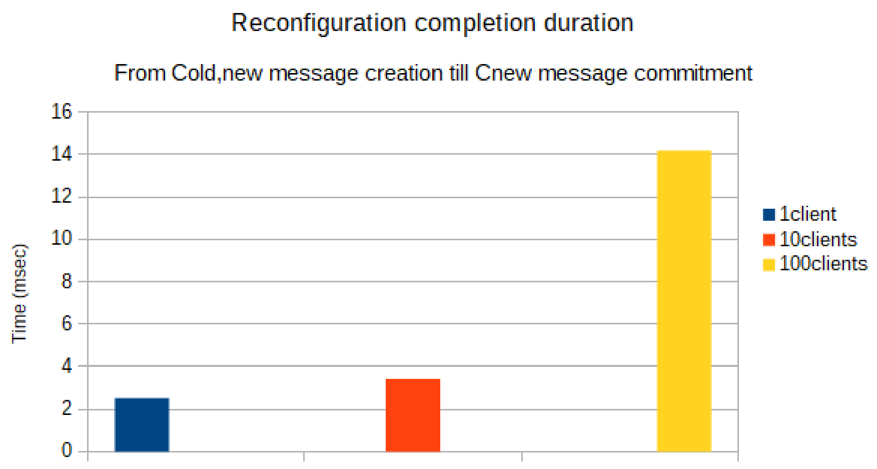


Figure 4.3: Duration of reconfiguration process (from start to completion)

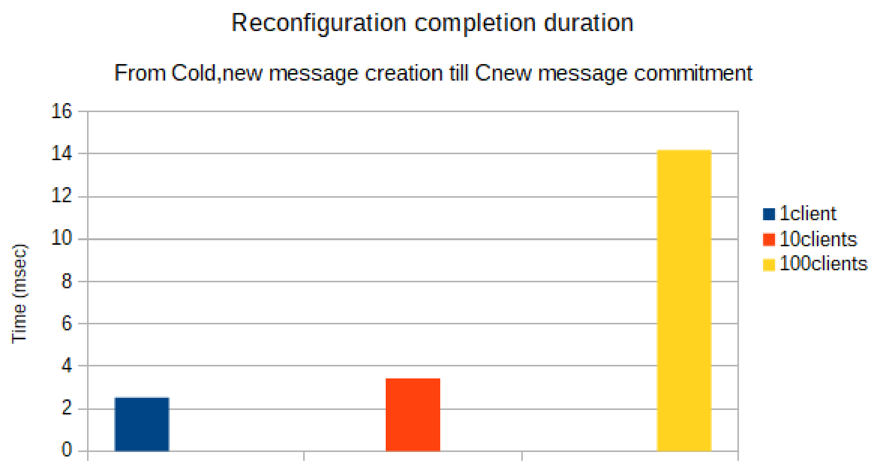


Figure 4.4: Number of committed log entries

currency and the duration of reconfiguration and number of committed log entries during reconfiguration. Differences are small in the cases of 1 and 10 clients. For 100 clients we see that the duration of reconfiguration and number of committed log entries increase significantly. The system in this case is under higher pressure due to the higher client load (proposal rate).

Analyzing the committed log entries during the three configurations (old, joint, or new) we perform a breakdown of how many of the incoming proposals were served at each configuration. In Figure 4.5, we depict the number of committed entries from the point where the leader has created the $C_{old,new}$ proposal (MsgAppRec message) to the point where it learns that it has been committed , we see that in the 1- and 10-client

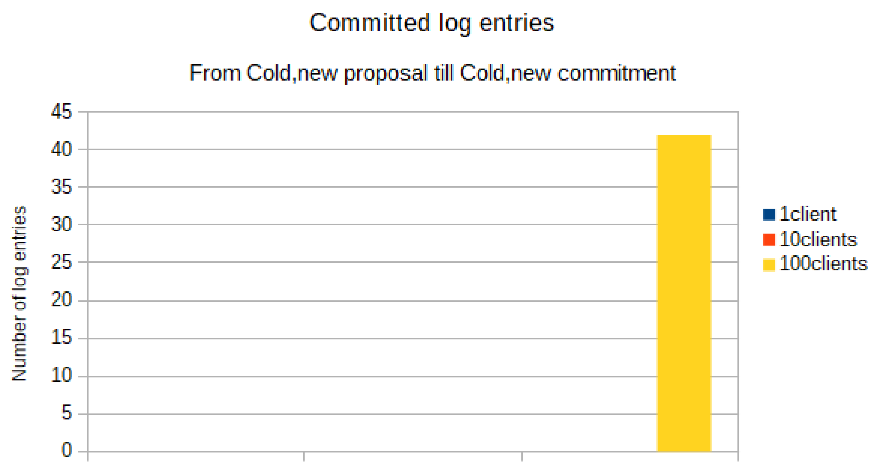


Figure 4.5: Committed log entries from $C_{\text{old,new}}$ proposal till $C_{\text{old,new}}$ commitment

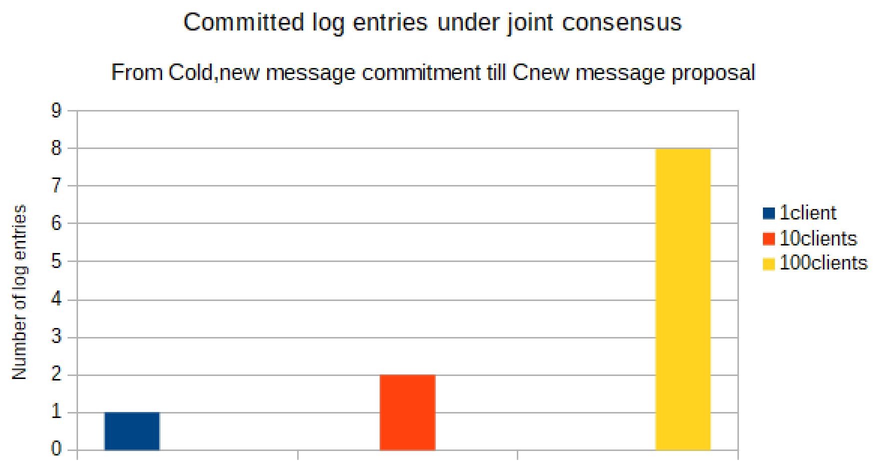


Figure 4.6: Committed log entries while leader is in joint consensus configuration

experiments, commitment of MsgAppRec message comes nearly immediately after the proposal, there is thus very little backlog of other proposals (outstanding, not yet committed) at the time of proposing $C_{old,new}$. However, in the 100-client experiment the leader learns of the commitment of up to 42 other proposals in the old configuration before learning of the commitment of the MsgAppRec proposal. In Figure 4.6, which depicts the number of log entries that are committed between the log index at which the leader learns that $C_{old,new}$ is committed, to the index at which it proposes C_{new} via the MsgAppNewConf message, we see that the processed requests under the joint-consensus configuration ($C_{old,new}$) are few (1—8) in all cases, namely the leader moves quickly in proposing C_{new} after learning that $C_{old,new}$ is committed. Figure 4.7 depicts

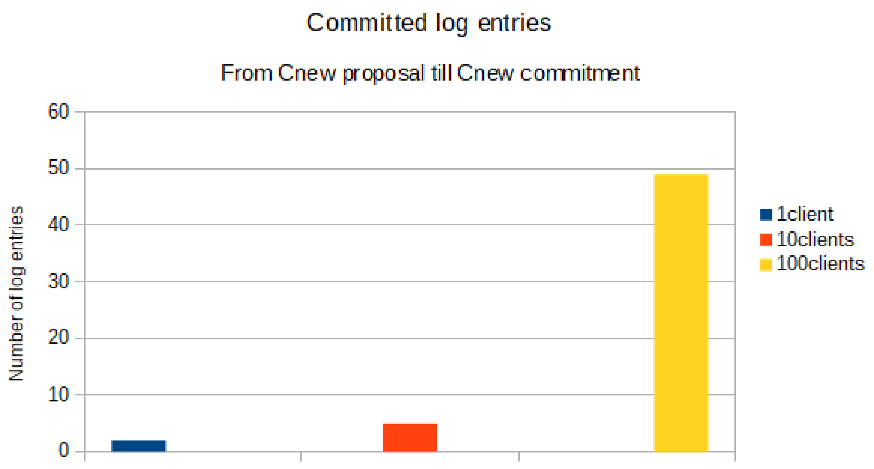


Figure 4.7: Committed log entries from proposal to commit of C$_{new}$

the number of committed log entries from the point where the leader proposes C$_{new}$ (MsgAppNewConf, leader changes its state to StateLeavingLeader) to the point it learns that it has been committed, i.e., it receives a quorum of responses with higher index than the MsgAppNewConf index, thus changing its state to StateLearner (if it is not part of C$_{new}$). In this case, although the leader is not part of the successor configuration, it continues to replicate log entries to the cluster, observing the Match indices of the new configuration members.

To ensure that, at least in these experiments, our joint consensus implementation maintains consistency, we extract and compare the contents of the Etcd database in each node using the *etcd-dump-db tool*. We save the extracted data in separate text files and then we compare the similarity of the text files with a diff checker tool. These text files are identical at the end of our experiments, providing a level of experimental confidence that replicated data remains consistent across nodes.

4.3 Etcd snapshot, compaction and defragmentation policy

In Etcd, the leader can bring a newly inserted node to an existing cluster up-to-date by sending it a snapshot of the full datastore state. After snapshotting, a node can trim the Raft log in the WAL directory, where segments of the log are persisted. This is termed *log compaction*, and differs from what is known as *history compaction*, which removes update revisions from the Etcd tree backend. Trimming the log purges WAL files with log entries smaller than the snapshot index, releasing disk space. The leader keeps up to 5000 most-recent log entries after trimming, so that the leader can update followers that are falling behind via log shipping rather than full snapshot transfer.

History compaction marks backend space with “tombstones” to declare deleted key-value revisions but typically does not release disk space; however, tombstones may be reused for future key-value entries. When history compaction is commanded at the leader, it is applied to the rest of the nodes as well. The database file size will increase again only when there is no more free space within the current file size to reuse.

To observe the performance characteristics of log and history compaction, snapshotting and defragmentation, we carry out experiments in which we apply history compaction on a 3-node cluster and immediately take a snapshot at the leader. Then we add a new node (a learner) in the existing cluster, which receives the snapshot from the leader. While one would expect that the learner would receive only live (non-history-compacted) data, we observe that it receives all data (equal to the amount of allocated disk space for the database). We also perform an experiment where the leader applies history compaction, defragmentation, and takes a snapshot, then sent to the learner.

The above observation prohibits, in the current Etcd implementation, use of reconfiguration to in-effect to replace a heavily fragmented cluster with one that contains a defragmented database, which could provide an alternative way to defragment an Etcd cluster (trading off the disk and CPU cost of in-place defragmentation with the CPU and network cost of transferring live data to new nodes).

In the two following experiments, five clients produce a total of 1M writes over 5 distinct keys. In both experiments, snapshots are taken at around 500K log entries. Each snapshot is followed by log compaction, preserving the most recent 5K log entries in the corresponding segment of the WAL directory.

In the first experiment of Figure 4.8, we perform history compaction with the *etcdctl compact* command at revision 450K, which takes 7.54sec to complete. All revisions prior to 450K are deleted. When compaction finishes, the database size at every node is 170MB. The database file stops growing as space corresponding to deleted entries is reused for future data. Database file size continues to grow from revision 900K and on. The incoming learner receives the snapshot whose size is 170MB although the database has been compacted. Had we not performed compaction, the

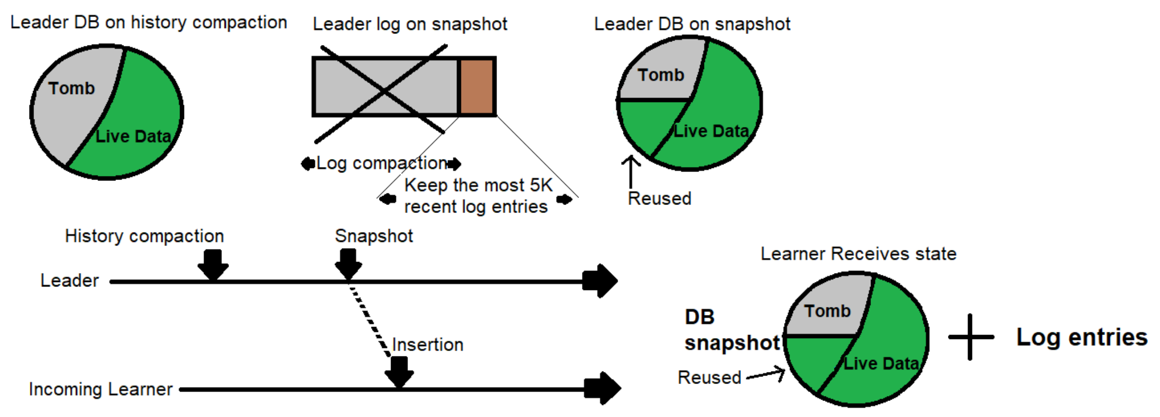


Figure 4.8: History compaction and snapshotting

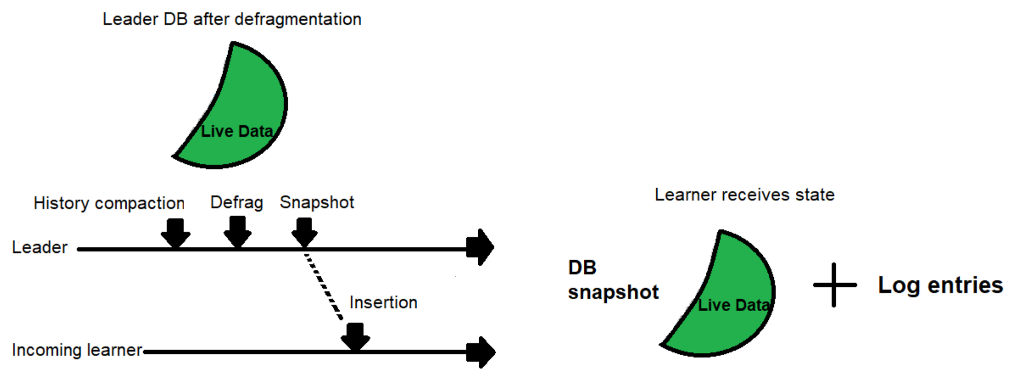


Figure 4.9: Effect of defragmentation on snapshot size

learner would receive a 188MB snapshot (as the file would have kept growing). Each database insertion is indeed considered as a revision; when we request compaction at revision 450K, compaction goes forward, evidence that 450K requests had indeed been performed. If there were fewer revisions, the request would not be immediately performed.

In the second experiment of Figure 4.9, we perform defragmentation at the leader by typing the command *etcdctl –endpoints defrag* at the console after history compaction is complete. Defragmentation is applied at the endpoints specified in the command; in our case we defrag the leader. We make history compaction at 450K followed by defragmentation, thus tombstoned disk space is removed from the database file, reducing it to 30MB. Snapshot is taken on 500K log entries and the incoming learner receives a snapshot sized at about 45MB. When the experiment is over, all nodes end up with a 198MB database. The leader and incoming learner have an equally sized database whose size is affected by defragmentation at the leader, whereas the other two nodes (followers) which are not defragmented, are not affected by defragmentation at the leader.

4.4 Scaling up using joint consensus

In this experiment we demonstrate the power of replica-group reconfiguration in dynamically adapting stateful replicated services by scaling up a 3-node etcd cluster through increasingly more powerful node types. Since all of the servers are equipped with the same technology, we had to emulate variable-resource-power nodes in order to evaluate the ability of our system to rapidly adapt. To achieve this, we used the cpulimit [40] command. With suitable arguments, cpulimit enforced an upper limit on CPU available to a process as set by the user. The control of CPU is achieved by sending SIGSTOP and SIGCONT POSIX signals to processes. Children processes and threads of the etcd processes share the same CPU percentage. In this way, we emulate nodes with three levels of CPU, *weak*, *medium*, and *strong*. Clusters will homogeneously use nodes with the same CPU level and will be termed accordingly, i.e. a cluster with weak CPUs will be call *weak*, and so on.

For this experiment, a total amount of 1.5M writes is produced over a range of 5 different keys. Snapshots are taken by the leader at 0.5M and 1M log entries. Just after the snapshots, learners are inserted. We use only one client appropriately set to avoid overload: increasing concurrency beyond a certain point overloads the nodes (especially due to CPU restriction used in this experiment), causes heartbeat intervals and leader election timeouts thresholds to be exceeded, thus causing unnecessary leader elections that interfere with the experiment.

In the experiment of Figure 4.10, the initial 3-member cluster of weak nodes (configuration C1) comprise node02 as leader and node03, node04 as followers (the client is deployed on node 01). We observe that taking snapshots has no latency impact. This can be explained by the fact that the snapshot is taken on the committed entries stored at the tree of the backend, so new entries are not prevented or being delayed from being replicated in the log structures of the nodes.

After the snapshot is taken, we add medium nodes 05, 06, 07 (configuration C2) as learners one after another (using our single-server extension of adding learners described in Section 3.1). In Figure 4.10 we observe that adding learners has a latency

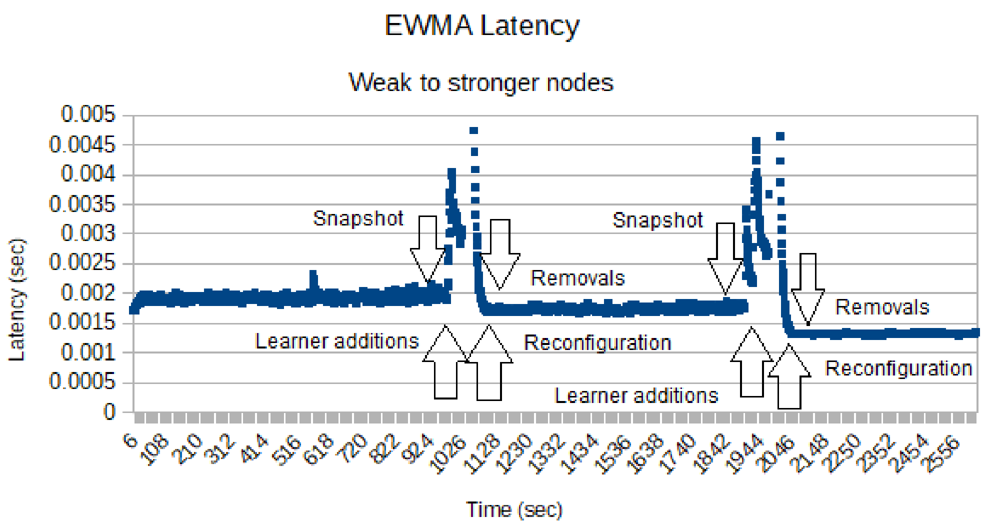


Figure 4.10: Latency of a 3-node group over two reconfigurations progressively increasing node CPU capacity

cost. This spike is highly related with the size of the snapshot (189MB and 3 seconds to be delivered) and/or amount of log entries that a node has to receive to get ready, as will be extensively explained later in the text (Section 4.8). It takes a small period of time for the new nodes to warm-up, establish their connections, receive the snapshot and store the configuration (other followers/learners) to their backend. An operator may observe the establishment of the new configuration and readiness of new nodes via messages on the respective server consoles.

When the new nodes are ready, they have received from leader the snapshot along with around 1500-3000 log entries and have caught up with the nodes of the previous configuration. Then the system is ready to pass from the old to the new configuration and the previous followers to become learners and vice versa. The reconfiguration itself (using joint consensus) causes a small period of unavailability (usually 1-2 sec at the client, also affected by timeouts there) primary because of leader election. When the new followers realize the absence of leader, they call an election and elect node 05. The time between a node becoming candidate and being elected leader is usually around 1.2-2ms measured by timestamps at the console of that node. We immediately remove the old followers (now learners) from the cluster to avoid their overhead on the leader. For the time interval between learner additions and old member removals (now learners), data is replicated to 6 nodes. After removals, the system is composed again of 3 nodes, but with better CPU performance. The same holds for the second reconfiguration but now, we pass from the medium to the strong cluster. Strong nodes now receive a 378MB snapshot which takes around 6 sec to be delivered.

As nodes of the strong cluster, we use nodes 02, 03, 04 again (configuration C3), unbounded by cpulimit settings, thus able to use as much of the CPU as necessary. Node 03 is elected as new leader in C3. In Figure 4.11 we depict the actual CPU usage percentages of the leaders of every configuration (C1 - C3).

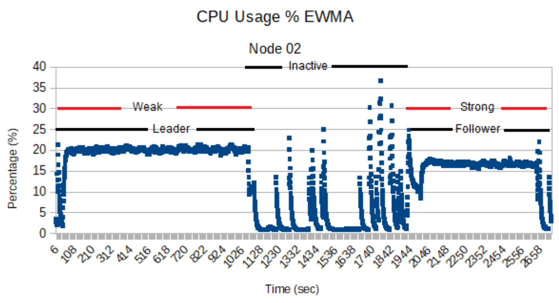
The CPU usage spikes observed when a node is inactive, correspond to records close to 100% around records close to 1-5 % and as such they are not smoothed out by EWMA. CPU usage of the leader node 02 in C1 (weak cluster) is around 20% (the maximum allowed by cpulimit). Similarly, CPU usage of node 05, leader of C2 (medium cluster), ranges at 22-23% (maximum allowed by cpulimit). Node 03 is the leader of C3 (strong cluster) with CPU usage nearly 30%. There is also a difference in CPU usage when nodes are followers. Weak follower CPU is 12-13% while strong follower is at around 17%. It is clear from Figure 4.10 and Figure 4.11 that relatively small differences in CPU strength result in observable differences in client latency.

In summary, reconfiguration is a mechanism that allows operator to vertically scale (scale-up) distributed key-value stores. When a system is considered overloaded ad to improve performance, a system administrator can use reconfiguration to achieve this. [t]

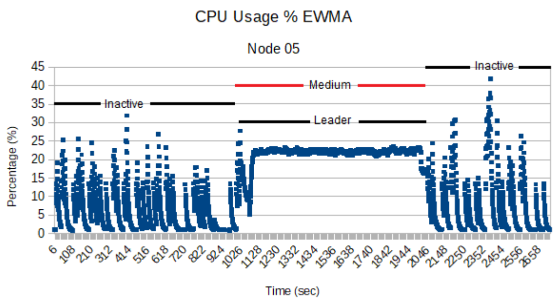
4.5 Simultaneously increasing fault-tolerance and performance

Reconfiguration enables us not only to vertically scale (scale up) a system and provide higher availability, but also make a system less sensitive to node crashes. In this experiment we use our reconfiguration mechanism to scale up and scale out, from a 3-member to a 5-member cluster at once. The 3-node cluster consists of weak nodes, emulated with the cpulimit restriction. The 5-node cluster CPUs are not bounded by cpulimit and are considered as strong nodes. By replacing the 3-node with the 5-node cluster, we use joint consensus reconfiguration mechanism to provide higher availability along with increased fault-tolerance. For this experiment, we also produce 1.5M writes over a range of 5 different keys. We also use only one client for the reasons referred in Section 4.4. Snapshot is taken by leader on 0.75M log entries.

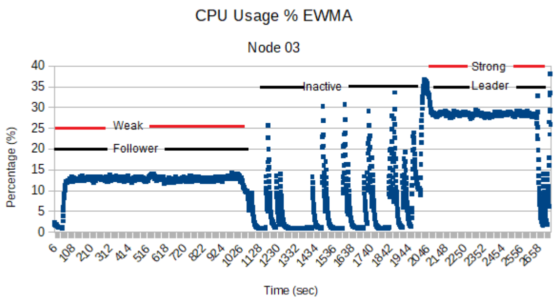
The weak cluster is consisted of node02 (leader), node03 (follower), node04 (fol-



(a) Node 2 CPU: leader in C1, inactive in C2, follower in C3



(b) Node 5 CPU: inactive in C1, leader in C2, inactive in C3



(c) Node 3 CPU: follower in C1, inactive in C2, leader in C3 CPU utilization of different nodes in C1-C3

Figure 4.11: (a-c): CPU utilization of different nodes in C1-C3

lower). We add node05, node06, node07, node08, and node09 as learners immediately after the snapshot. They receive a snapshot of 284MB, which takes 5 seconds to be delivered. When all of the learners are ready, we proceed to reconfiguration. As we have already said, reconfiguring to cluster in which leader is absent lead to 1-2 sec of unavailability. After the reconfiguration is completed (node06 is elected as new leader), we remove the old followers (currently learners). Between learner additions (now followers and leader) and follower removals (now learners) cluster consists of 8 nodes. After all, new configuration is composed of 5 members. Although data is now replicated to more nodes (system can now afford 2 node failures) base latency is decreased, because new nodes have more CPU capacity. In this section, we will not illustrate the CPU usage performances of weak and strong nodes, as we do not have something remarkably different comparing to the previous section to depict.

To increase performance, it may only be necessary to upgrade the leader node

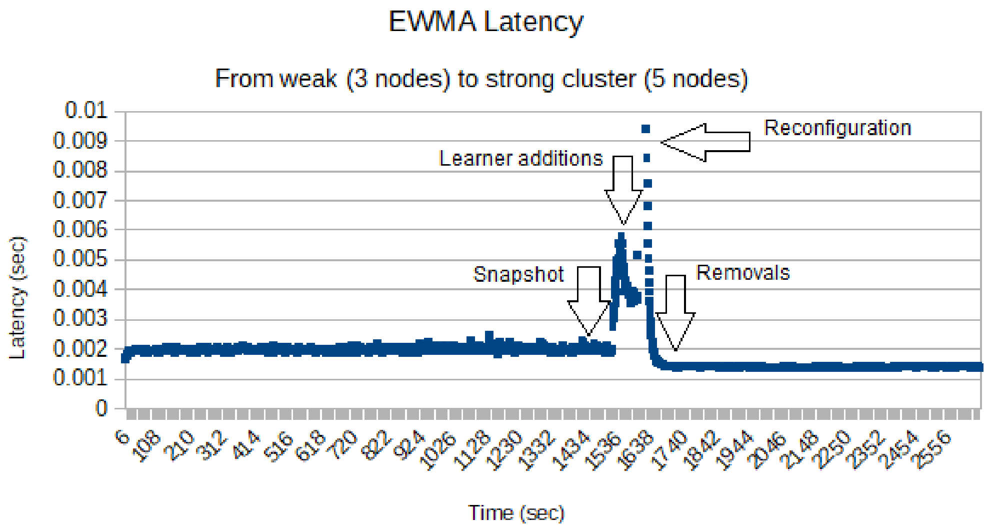


Figure 4.12: Latency of a 3-node to 5-node group over reconfiguration increasing node CPU capacity

(or any majority of nodes), as they are on the critical path to committing proposals in the replica group. However, having a single powerful node as a leader will work only for as long as this node remains a leader. After an election, a less powerful node becoming leader will bring down throughput (increase latency) for the entire replica group.

4.6 Characterizing the performance impact of learner additions

A key feature of any highly-available replica reconfiguration system is the ability to prepare replicas (transfer state to them and keep them up to date) proactively so as to minimize the impact of a reconfiguration action. In this section, we study the performance cost in terms of latency and CPU overhead, of adding learners. We study the relationship between the size of state that the incoming learner has to receive, with the latency penalty we observe at the client. We also observe that leader CPU increases during learner additions. We performed benchmark runs in which learners receive different sized snapshots. We spread learner additions over time and compare to the impact of concurrent additions. By concurrent additions we mean single-server additions (performed one after another) close in time, within a small (1 sec) interval. For these experiments, we increase the number of the clients to 5, as this concurrency

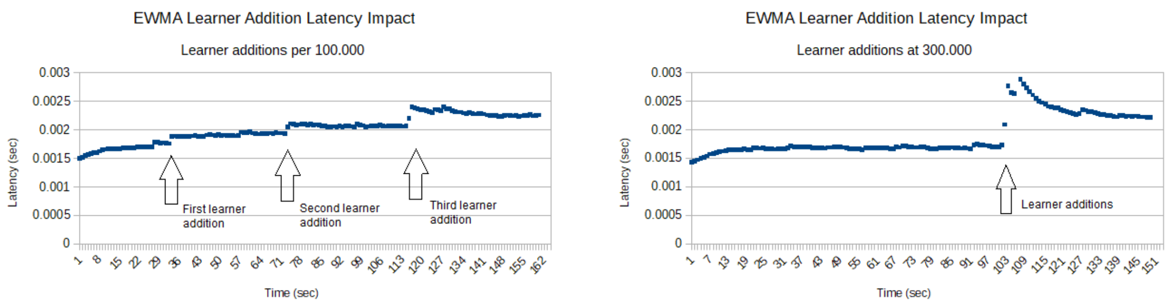


Figure 4.13: Latency impact of learner additions (Bench 1)

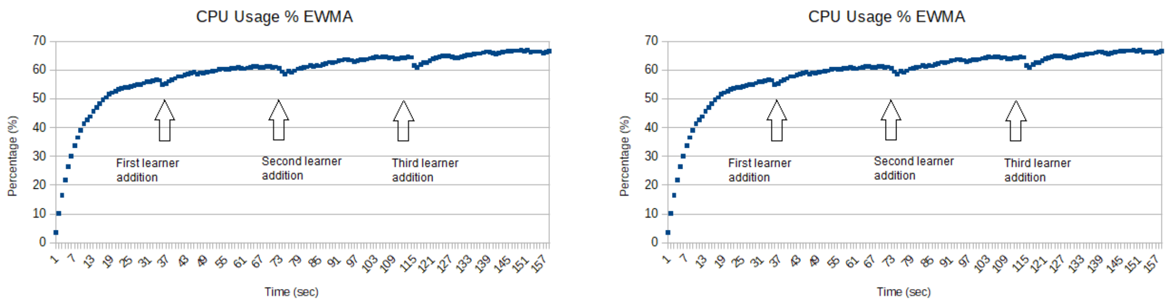


Figure 4.14: Leader CPU impact of learner additions (Bench 1)

level stretches sufficiently the servers (CPU usage of 55-65% at the leader and 30% for followers). We keep the same key space size equal to 5 distinct keys. We perform three types of experiments, each consisting of two scenarios:

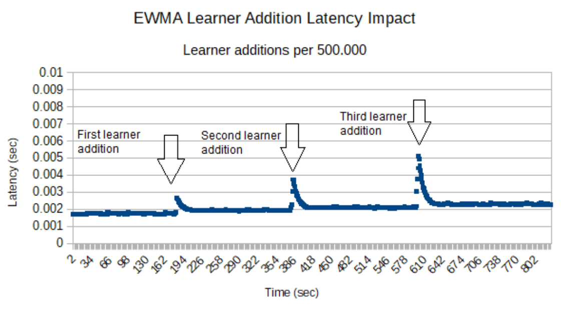
Bench 1: Benchmark runs with a total amount of 400K writes.

Scenario 1: The leader takes snapshot every 100K log entries. After each of the first three snapshots is taken, a new learner is added to the cluster. They receive around 39, 77 and 115.5MB snapshots respectively. The three snapshots need 0.6 sec, 1.1 sec, and 1.7 sec to be delivered respectively.

Scenario 2: The leader takes one snapshot at 300K log entry. Then all nodes are added consequently. Each learner receives around 115.5 Mb snapshot. Each snapshot needs around 1.7 seconds to be delivered in each node.

Bench 2: Benchmark runs with a total amount of 2M writes.

Scenario 1: The leader takes snapshot every 500K log entries. After each of the first three snapshots are taken, a new learner is added to the cluster. Learners receive snapshots of 189, 378, 566MB respectively. The 500K-entry snapshot needs around 3 seconds, the 1M-entry snapshot needs 6 seconds, and the 1.5M-entry snapshot needs 8.2 seconds.



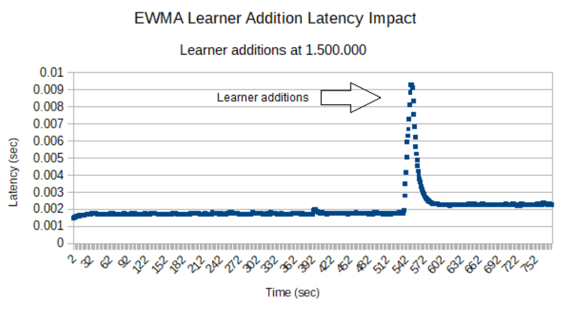
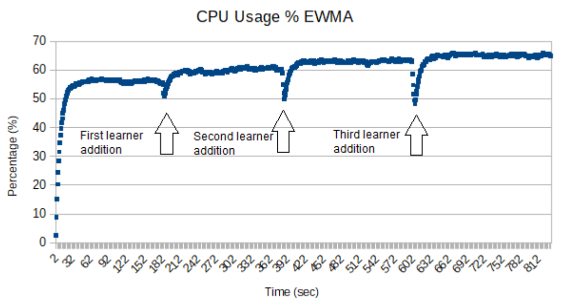


Figure 4.15: Latency impact of learner additions (Bench 2)



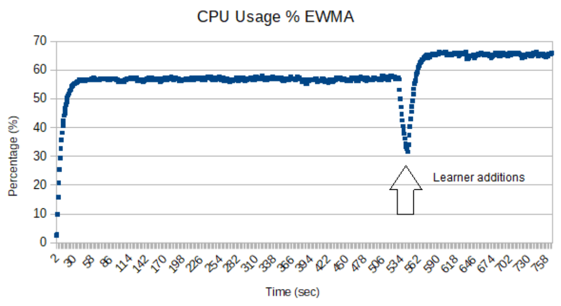


Figure 4.16: Leader CPU impact of learner additions (Bench 2)

Scenario 2: The leader takes one snapshot at 1.5M log entry. Then all nodes are added consequently. Each node receives snapshot size of around 566MB. Each snapshot needs around 8.2 sec to be delivered.

Bench 3: Benchmark runs with a total amount of 4M writes.

Scenario 1: The leader takes snapshot every 1M log entries. After each snapshot is taken, a new learner is added to the cluster. Learners receive around 377MB, 753.5MB and 1.13GB snapshots, respectively. Snapshots need around 6 seconds, 10 seconds, and 15 seconds to be delivered, respectively.

Scenario 2: The leader takes one snapshot at 3M log entries. Then all nodes are added consequently. Each node receives a snapshot of 1.13GB, which needs 15 sec to be delivered.

In Figure 4.13, Figure 4.15, and Figure 4.17 (latency impact of learner additions) we observe:

- Learner additions always impact steady-state latency. When we add the 3 learners spaced-out in time, steady state latency is gradually increased. On the other hand, concurrent additions lead to the steady-state latency of the third learner addition at once.

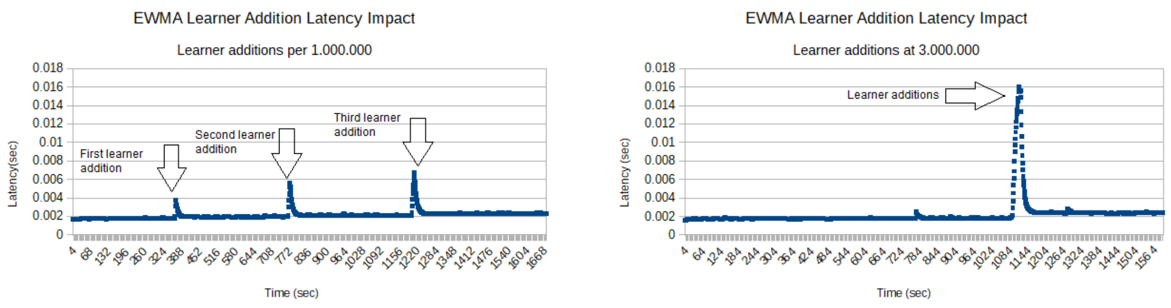


Figure 4.17: Latency impact of learner additions (Bench 3)

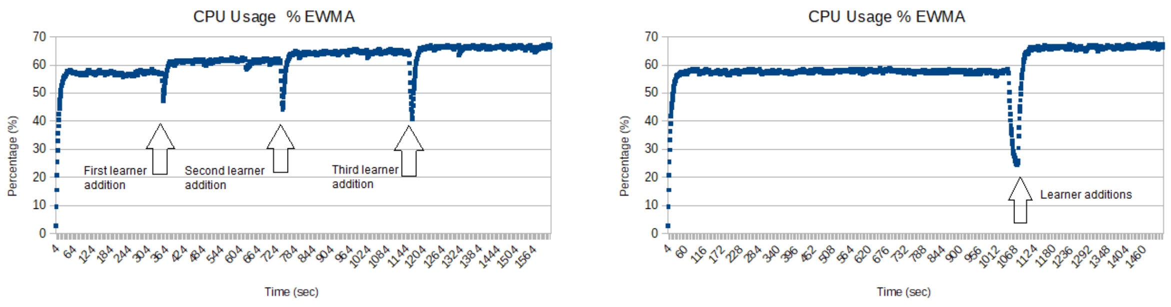


Figure 4.18: Leader CPU impact of learner additions (Bench 3)

- Not every addition cost is the same. In all cases, concurrent additions are more expensive comparing to spaced-out additions. It is costlier for the leader to prepare 3 nodes concurrently than prepare them separately.

- Latency spikes are strongly related to snapshot size. The more state is snapshotted by the leader, the bigger snapshot a learner will receive. This is clearly reflected to latency figures and results in latency spikes.

In Figure 4.14, Figure 4.16, and Figure 4.18 (CPU impact of learner additions) we observe:

- At the time of learner addition, leader CPU usage appears dips. When an addition is completed, leader CPU consumes more resources in order to serve an additional node. Adding all learners concurrently, increases CPU usage at once to level after the third learner has been added.

- Like latency, not every addition costs the same. Concurrent additions cost more than spare additions, as well. Leader is busy in sending state to the learners which affects CPU usage, depicted with sharper dips.

- Just like latency spikes, CPU dips are related with state size. The biggest state a learner receives, the deepest CPU usage dip we observe.
- CPU usage increment in the beginning of the CPU percentage figures is caused by the first record of top command, which is always 0-5 %

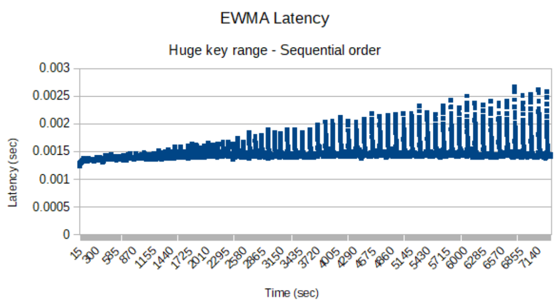
We also studied the same cases for the scenario that leader does not snapshot. The same conclusions also apply there. In these experiments, the latency cost when learners receiving snapshot is comparable to the case of learners receiving full log from the leader. However, by experience we have observed that learners that receive a snapshot are ready to join the cluster more quickly, compared to the case of receiving a full log.

In these experiments we have determined that learner additions incur costs during reconfiguration, however learner additions are an essential proactive action to avoid stalls during reconfiguration. The cost and timing of learner additions are a tunable parameter that depends on how soon a user wants learners to be ready and/or whether to limit the impact of learner additions.

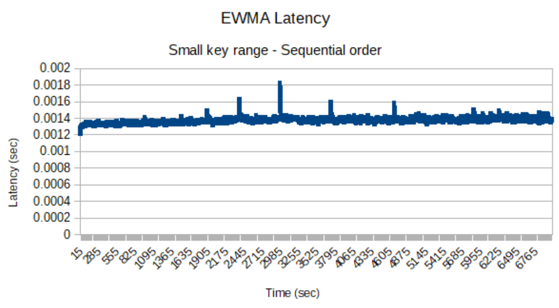
4.7 Impact of back-end

Performing experiments, we tried to increase the size of the key pool. We discovered that when we increase key space size, there is a background noise (jitter) in latency (Figure 4.19) and CPU (Figure 4.20) figures. We perform experiments that produce 5M writes in sequential order using one client. We also tried experiments with random key order to see whether order matters but observed no significant difference. The etcd cluster comprises a leader and two followers. In case (a), key space size is equal to the total number of writes (each key is unique), while in case (b), key space size is 5. This means that all writes except for the first 5, are updates (new versions) of the values of the previous keys. These two cases agree in base latency, but setting large key range leads to jitter effect that is always increasing across latency axis as time goes by.

We also studied the CPU behavior (Figure 4.20) and concluded that there is a strong relation between the latency and the CPU % usage jitters. Base usage fluctuates on the same level in both cases and differentiates in percentage range. We believe

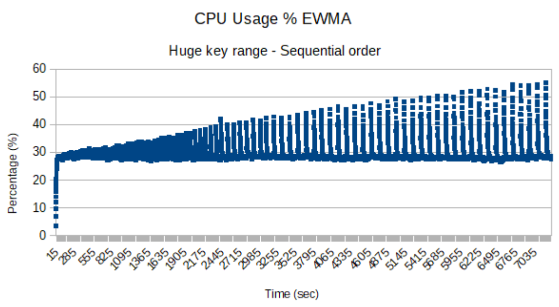


(a) Huge key space size

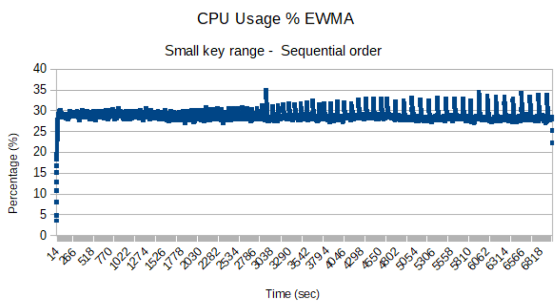


(b) Small key space size

Figure 4.19: Latency impact of key range size



(a) Leader CPU usage with huge key range



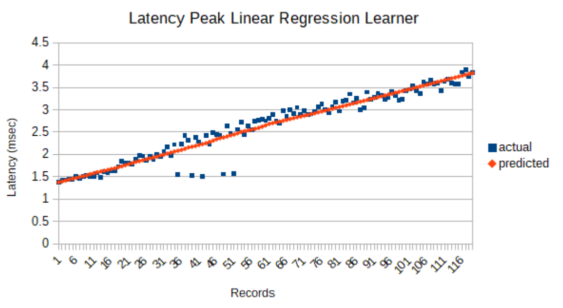
(b) Leader CPU usage with small key range

Figure 4.20: CPU usage impact of key range

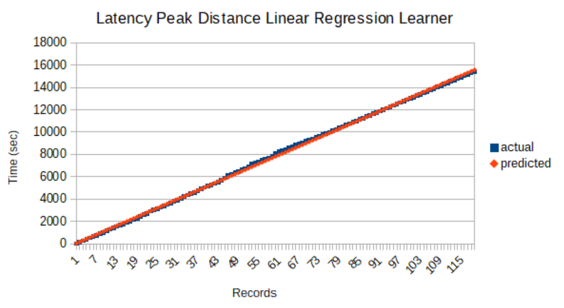
that jitter is caused by the B+ tree expansion of BoltDB. When key space size is equal to the number of the writes, tree is forced to always increase its leaves. Some of the insertions naturally lead to branch splits, which sometimes reach up to the root of the tree. All these splits cause this increasing jitter in CPU, which afterwards leads to the latency jitter. Studying jitter, we also noticed that not only spikes are increasing, but they appear less often as time goes by. This reinforces our claim because as tree grows up, fewer and fewer value redistributions are necessary, but when a branch split happens, it costs more every time.

4.8 Experimenting with prediction models

We trained prediction models that help us know with good precision when the next latency spike will appear and how big it will be. We build the regression models using Matlab Regression Learner Application. The Matlab *findpeaks* [41] function helps us



(a) Latency Peak Height Linear Regression Learner



(b) Latency Peak Distance Linear Regression Learner

Figure 4.21: Regression models

Table 4.1: Regression model metrics

	RMSE	R2	MSE	MAE
Linear Regression (Peak Heights)	0.19	0.94	0.03	0.11
Linear Regression (Peak Distances)	116.84	1.00	13651.40	97.88

detect the peaks of the jitter spikes. This function returns the exact positions (x, y) of the spikes. We then create two datasets: a dataset that contains the X axis positions of jitter spikes and a dataset that contains the Y (latency) axis values of the spikes. We train the data using 10-fold *cross validation*. Initial data was collected by benchmark run set up with one client and lasts 4 hours. Writes are served by a 3-member cluster.

After trying several models, we concluded that Linear Regression [42] model is exactly what we need. Model selection criteria is the least Root Mean Square Errors (RMSE), but in the case of the latency peak distance dataset, although we see that Gaussian Processes [43] RMSE is less than Linear Regression RMSE, data is overfitted. For that reason, we prefer simpler models like Linear Regression. In Figure 4.21 we can validate the prediction accuracy of the models. We subjoin additional metrics in Table 4.1, as well.

4.9 Joint consensus vs single server reconfiguration

In the previous experiments we observed that joint consensus reconfiguration costs can be separated in the following two phases, which may be interleaved in different ways:

- Learner additions: In Section 4.6 we observed that the cost of learner additions depends on the size of transferred state. Adding learners at about the same time has a higher impact (disrupts) the request pipeline, as the leader is busy with sending several snapshots to the learners. Request response times naturally increase during this time.

- Reconfiguration: Reconfiguration in Raft is not by itself an expensive process as it requires light processing (it is treated just as two normal requests) and does not stall request progress (Section 4.2). The real impact of reconfiguration is realized when the leader is not member of the new configuration and thus the cluster must elect a new leader, resulting to a short unavailability interval.

Full reconfiguration of a 3-member replica set (A, B, C to D, E, F) can be performed using 6 single-server reconfiguration actions that replace step-by-step the members of the cluster, instead of one reconfiguration action that changes all cluster members at once. Single-server reconfiguration can be performed either using our implementation of the two-phase joint consensus reconfiguration, or the specific single-phase single-server reconfiguration action (Section 2.2). Learner additions (state transfer) are needed in both methods of reconfiguration. The member-add procedure in the default Etcd reconfiguration (single server) operates similar to adding learners in our joint consensus reconfiguration. An important factor is how state-transfer operations are spaced in time.

Our general observations from our experiments are as follows:

- Simultaneous addition of many members or learners will cause higher latency spikes during state transfer, compared to additions that are more spread in time.

- The size of the state transferred (amount of data in snapshot plus log) impacts the breadth and height of the latency impact per added node

- Higher number of replica nodes (members or learners) increases base latency as the leader needs to replicate data to more replicas in its common path

- Removing a leader from the cluster causes unavailability

We distinguish two main policies for doing reconfiguration, one geared towards minimizing its latency impact (Min-impact) and another geared towards minimizing time

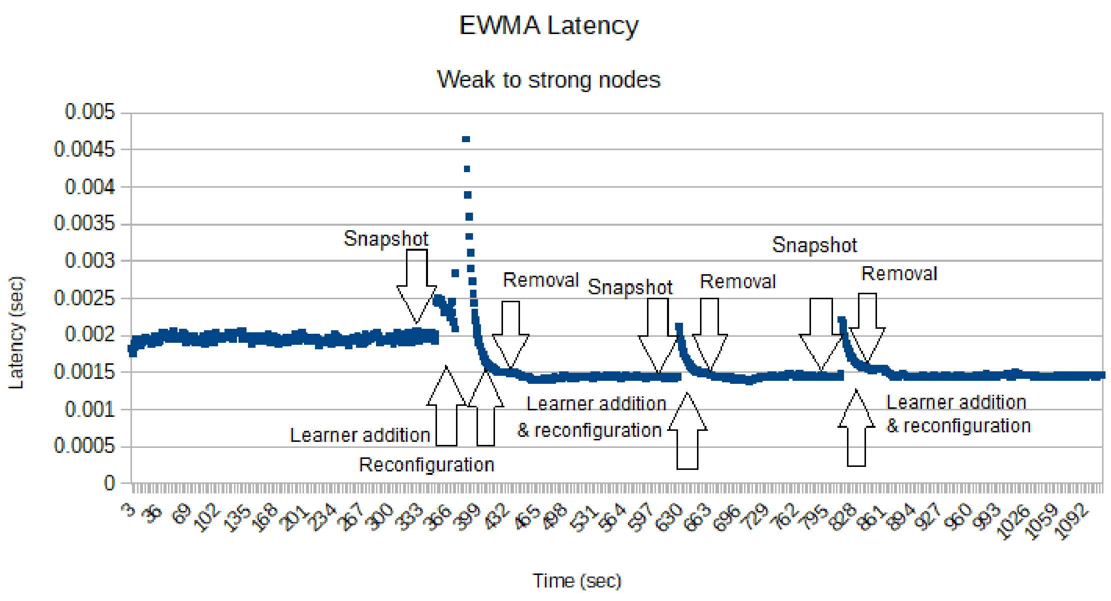


Figure 4.22: Full replica-group reconfiguration using min-impact policy with single-server changes

duration of reconfiguration (termed ASAP). Snapshot (and defragmentation) prior to state transfer are recommended to reduce state size in all cases.

- Min-impact: Schedule reconfiguration as successive new-member-add old-member-remove single-server reconfigurations [1]. The benefit of this policy is that base latency remains low and that the latency impact of state transfer is also bounded since state transfer happens a single replica at a time. However, this policy is expected to result to longer reconfiguration actions.

- ASAP: First add all replicas as learners (nearly simultaneously to reconfigure as soon as possible), then complete the reconfiguration. One downside of this is that the cluster may operate for some time with a large number of replicas (old and new), increasing base latency. The impact of this policy has already been examined in Figure 4.10. Combining multiple single-server reconfigurations into a single joint-consensus action is not expected to significantly reduce reconfiguration time, even in the case of large replica groups or under high load.

To demonstrate the min-impact policy we carry out an experiment (one client performing 600K writes, key space size 5) moving from a weak *directly* to a strong three-member replica group (Figure 4.22). Snapshots are taken at 150K (56MB, 0.7

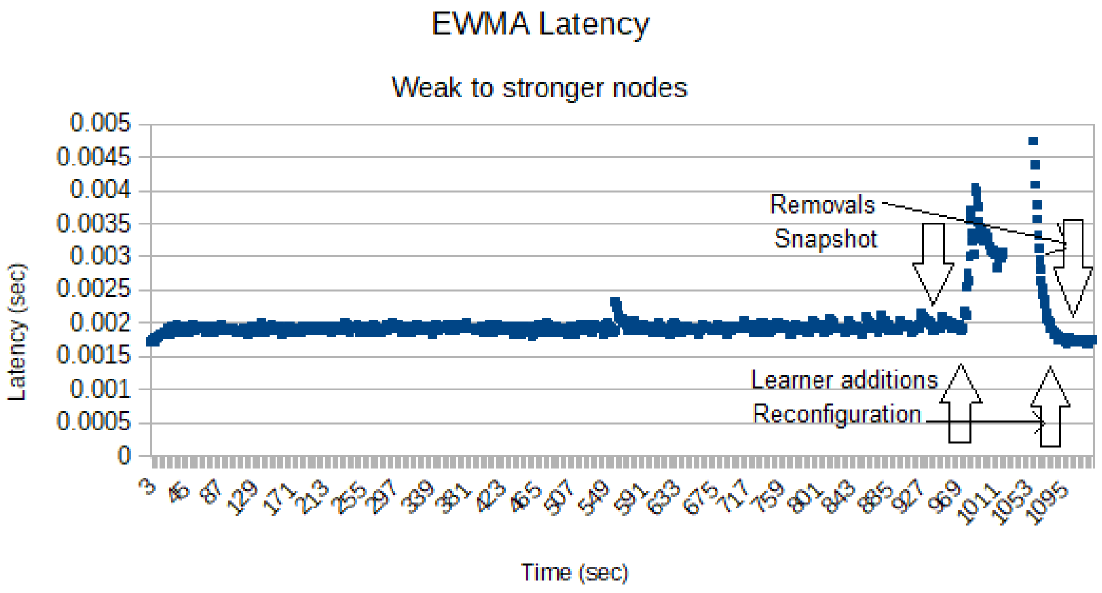


Figure 4.23: Full replica-group reconfiguration using ASAP policy (excerpt of Figure 4.10)

sec), 300K (113.4MB, 1.6 sec) and 450K (169.6MB, 2.5 sec) entries just before adding each learner. The first reconfiguration replaces the leader with as stronger (not CPU-restricted) node, causing a short latency impact due to the leader election but also a steady-state drop due to the stronger leader. The following two reconfigurations replacing weak with strong followers are low-cost since they do not involve leader elections, thus we only observe learner addition (state transfer) costs.

Comparing to the experiment of Figure 4.23 (excerpt of Figure 4.10) where reconfiguration takes place using the ASAP policy, our main observation is that the min-impact policy indeed results in lower latency cost due to learner additions (state transfer) being spread out in time.

In both policies, state-transfer and leader election costs cannot be avoided, although state-transfer costs can be regulated by controlling state size and time spacing of node additions. Joint consensus can in fact yield a short-term performance impact when reconfiguring large replica-groups (e.g., with a 20-member group would need to grow to a 20-member plus 20-learner group, for a total of 40 nodes, before reconfiguring). Joint consensus does not seem preferable over multiple single-server reconfigurations on the grounds of performance.

An advantage of joint consensus reconfiguration is that it is an atomic operation that can apply cluster changes at once, e.g. important when nodes of different types cannot simultaneously co-exist and/or interoperate. On the other hand, single server reconfiguration is easier to develop (our implementation of joint consensus adds up at least 1500 lines of new code, plus testing complexity). Generally, what can be done by joint consensus reconfiguration, can also be done by single-server reconfiguration.

Chapter 5

Conclusions and future work

5.1 Conclusions

In this thesis, we studied the adaptability of replication systems using joint consensus reconfiguration. Systems usually implement single-server techniques in order to add/remove a replication node, as developers naturally choose algorithms that are simpler to implement and test. In this thesis, we investigated the implementation of Raft joint-consensus reconfiguration and prototyped it in Etcd. We adjusted reconfiguration algorithm to the needs of Etcd, abiding by the principles of the initial algorithm proposal. We extended the Etcd client CLI in order to introduce learners, passive nodes that build and maintain replica state without participating in voting. The intention in adding a learner is to prepare for replacing an existing member or increase fault-tolerance through reconfiguration mechanism. To properly evaluate reconfiguration, we modified the Etcd benchmark by inserting a leader-detection feature to it. Our extensive experimental analysis of joint consensus reconfiguration latency and resource costs, and close look into the cross-reconfiguration differences, led us to the following conclusions:

- A large fraction of the reconfiguration cost is due to the leader election needed when the current leader is not in the new set of replicas; in that case, the outage lasts for less than 2 sec (at most two 1-sec measurement intervals). When the current leader remains in the new set and learners have already been prepared, the impact of reconfiguration is minimal in both the single-server and joint-consensus cases (Sections 4.4, 4.5, 4.9; Figures 4.10, 4.12, 4.22, 4.23)

- The performance of a replica group depends on the number of follower replicas and the capacity of the leader (that is, latency increases with more replicas, or when the leader is saturated -although followers may not be-) (Sections 4.4, 4.5, 4.9; Figures 4.10, 4.12, 4.22, 4.23)
- The cost of adding learners is part of the total reconfiguration cost (impact in baseline latency, and CPU impact on the leader during of state transfer) (Sections 4.4, 4.5, 4.6 , 4.9; Figures 4.10, 4.12, 4.13, 4.15, 4.17, 4.22, 4.23)
- Being proactive is important: Preparing learners before proceeding to a new configuration avoids stalls during reconfiguration for state transfer. But, the latency overheads and CPU overhead of learner addition are unavoidable. (Sections 4.4, 4.5, 4.6 , 4.9; Figures 4.10, 4.12, 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, 4.22, 4.23)
- Latency spikes (peak and duration) during learner additions depend on the size of state transferred and on whether learners are added concurrently or spaced out in time (Section 4.6; Figures 4.13, 4.15, 4.17)
- To reduce the amount of state transferred during learner additions, it is important to perform snapshot operations prior to adding learners (Sections 4.4, 4.5, 4.6 , 4.9; Figures 4.10, 4.12, 4.13, 4.15, 4.17, 4.22, 4.23)
- Executing joint consensus under high client load leads to a longer reconfiguration phase as more log entries are committed while reconfiguration is undergoing (Section 4.2; Figures 4.3, 4.4)
- Joint consensus reconfiguration is a more general form of single-server reconfiguration with which one can atomically effect multiple changes, however it does not seem to lead to practical benefits compared to the latter. Learner addition costs and possibly leader-election costs, which are important components of total reconfiguration costs, are common to both ways of reconfiguring (Sec-

tion 4.9; Figures 4.22, 4.23). Developers of replication solutions indeed prefer single-server reconfiguration as a simpler alternative.

- Jitter in latency and CPU usage increases with the number (range) of keys in the B+ tree of BoltDB and may be related to reorganization during key insertion (Section 4.7; Figures 4.19, 4.20). The exact height and timing of jitter spikes can be accurately predicted through training linear regression models (Section 4.8, Figure 4.21)

5.2 Future work

In terms of extending our evaluation in the future, it would be interesting to test our system in other experimental testbeds, using a larger number of nodes, SSD storage and/or more powerful machines. It would be interesting to compare different aspects of Etcd's joint consensus reconfiguration with other related systems. For example, the author of another thesis [44] replaced BoltDB with RocksDB in Etcd. To consider the impact of the backend (e.g., take advantage of better write performance of LSM-trees in RocksDB vs. B+ trees in BoltDB), it would be interesting to evaluate the Etcd reconfiguration mechanism over their implementation (however we have not yet as of the time of writing this thesis been able to have access to that codebase). We intend to perform experiments with a write-intensive client workload, where the system will perform reconfiguration and change to RocksDB-backed Etcd nodes from a standard set of BoltDB-backed nodes and vice versa. This requires a RocksDB to BoltDB (LSM to B+ tree) data mapping and a mechanism that will detect the type of workload and trigger reconfiguration. We intend to apply the prediction technique to predict the latency impact of changing the number and type of nodes in a replication group (and vice versa).

To maximize the impact of this work, we intend to make a stable version of our modified Etcd code eventually available to the community via a Github repository. One of the new features we intend to include is a separate way to start learners (they are currently started with the help of an if-statement, depending on the name of the node we decide whether to start learner or member). Additionally, we must find a way to update the server-level cluster interpretations of the nodes about the proper learner and member progress maps. There are many functions that have almost the same usage. With proper architectural and structure changes, we will aim to simplify and minimize the size of our code as an extension to baseline Etcd. Leader detection in Etcd benchmark also needs improvements: the current benchmark implementation is able to detect a limited amount of leader changes, which fits the needs of our experiments in this thesis. However, general availability of this feature will require code improvements in the benchmark to enable it to detect every leader-change event.

Bibliography

[1] V. V. Cogo, A. Nogueira, J. Sousa, M. Pasin, H. P. Reiser, and A. N. Bessani, “FITCH: supporting adaptive replicated services in the cloud,” in *DAIS*, vol. 7891 of *Lecture Notes in Computer Science*, pp. 15–28, Springer, 2013.

[2] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, “The smart way to migrate replicated stateful services,” in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys ’06, (New York, NY, USA), pp. 103–115, ACM, 2006.

[3] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “Zookeeper: Wait-free coordination for internet-scale systems,” in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC’10, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2010.

[4] “Zookeeper github repository.” `https://github.com/apache/zookeeper`. Accessed: 2018-12-5.

[5] “Etcd github repository.” `https://github.com/etcd-io/etcd`. Accessed: 2018-12-5.

[6] “Etcd official site.” `https://coreos.com/etcd/`. Accessed: 2018-12-5.

[7] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC’14, (Berkeley, CA, USA), pp. 305–320, USENIX Association, 2014.

[8] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, “Distributed systems (2nd ed.),” ch. The Primary-backup Approach, pp. 199–216, New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993.

[9] C. A. Thekkath, T. Mann, and E. K. Lee, “Frangipani: A scalable distributed file system,” in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP ’97, (New York, NY, USA), pp. 224–237, ACM, 1997.

[10] E. K. Lee and C. A. Thekkath, “Petal: Distributed virtual disks,” in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, (New York, NY, USA), pp. 84–92, ACM, 1996.

[11] L. Lamport, “Paxos made simple,” *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, December 2001.

[12] F. P. Junqueira, B. C. Reed, and M. Serafini, “Zab: High-performance broadcast for primary-backup systems,” in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN ’11, (Washington, DC, USA), pp. 245–256, IEEE Computer Society, 2011.

[13] D. Ongaro, *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014.

[14] “Go language explained by robert pike.” `http://9p.io/sources/contrib/ericvh/go-plan9/doc/go_talk-20091030.pdf`. Accessed: 2018-12-5.

[15] B. M. Oki and B. H. Liskov, “Viewstamped replication: A new primary copy method to support highly-available distributed systems,” in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC ’88, (New York, NY, USA), pp. 8–17, ACM, 1988.

[16] D. Mazières, “Paxos made practical,” 2007.

[17] A. Shraer, B. Reed, D. Malkhi, and F. Junqueira, “Dynamic reconfiguration of primary/backup clusters,” in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC’12, (Berkeley, CA, USA), pp. 39–39, USENIX Association, 2012.

[18] “Boltdb github repository.” `https://github.com/boltdb/bolt`. Accessed: 2018-12-5.

[19] “Kubernetes github repository.” `https://github.com/kubernetes/kubernetes`. Accessed: 2018-12-5.

[20] “Kubernetes official site.” https://kubernetes.io/. Accessed: 2018-12-5.

[21] “Tikv github repository.” https://github.com/tikv/tikv. Accessed: 2018-12-5.

[22] “Tidb github repository.” https://github.com/pingcap/tidb. Accessed: 2018-12-5.

[23] “Tidb & tikv official site.” https://www.pingcap.com. Accessed: 2018-12-5.

[24] “Rust language.” https://www.rust-lang.org/en-US/. Accessed: 2018-12-5.

[25] “Cockroachdb github repository.” https://github.com/cockroachdb/cockroach. Accessed: 2018-12-5.

[26] “Cockroachdb official site.” https://www.cockroachlabs.com/. Accessed: 2018-12-5.

[27] “Consul github repository.” https://github.com/hashicorp/consul. Accessed: 2018-12-5.

[28] “Consul official site.” https://www.consul.io/. Accessed: 2018-12-5.

[29] “Serf official site.” https://www.serf.io. Accessed: 2018-12-5.

[30] A. Das, I. Gupta, and A. Motivala, “Swim: Scalable weakly-consistent infection-style process group membership protocol,” in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN ’02, (Washington, DC, USA), pp. 303–312, IEEE Computer Society, 2002.

[31] O. Rodeh, “B-trees, shadowing, and clones,” *Trans. Storage*, vol. 3, pp. 2:1–2:27, Feb. 2008.

[32] “Rocksdb github repository.” https://github.com/facebook/rocksdb. Accessed: 2018-12-5.

[33] “Rocksdb official site.” https://rocksdb.org/. Accessed: 2018-12-5.

[34] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The log-structured merge-tree (lsm-tree),” *Acta Inf.*, vol. 33, pp. 351–385, June 1996.

[35] A. Bessani, J. a. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with bft-smart," in *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, (Washington, DC, USA), pp. 355–362, IEEE Computer Society, 2014.

[36] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *SIGACT News*, vol. 41, pp. 63–73, Mar. 2010.

[37] "Etcd edited version." `https://github.com/etcd-io/etcd/releases/tag/v3.3.0-rc.0`. Accessed: 2018-12-5.

[38] "Google protocol buffers." `https://developers.google.com/protocol-buffers/`. Accessed: 2018-12-5.

[39] J. S. Hunter, "The exponentially weighted moving average," *Journal of Quality Technology*, vol. 18, no. 4, pp. 203–210, 1986.

[40] "Cpulimit github repository." `https://github.com/opsengine/cpulimit`. Accessed: 2018-12-5.

[41] "Matlab findpeaks function." `https://www.mathworks.com/help/signal/ref/findpeaks.html`. Accessed: 2018-12-5.

[42] S. Weisberg, *Applied Linear Regression*. Hoboken NJ: Wiley, third ed., 2005.

[43] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[44] G. M. Kokkinou and N. Kozyris, *Optimizing Write Performance in the etcd Distributed Key-Value Store via Integration of the RocksDB Storage Engine*. PhD thesis, NTUA, 2018.

Short Biography

Dimitrios Valekardas is a M.Sc. graduate student at the Department of Computer Science and Engineering of University (CSE) of Ioannina, Greece. He is a member of CSE Distributed Systems Group since 2017. Dimitrios received his B.Sc. degree from the CSE Department in 2016. His research interests revolve around distributed systems, data stores, and availability and performance improvement of distributed applications.