University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, Heraklion, GR-70013, Greece

A Distributed Key-Value Store based on Replicated LSM-Trees

Panagiotis Garefalakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

Thesis Advisor: Prof. *Angelos Bilas*

July 2014

This work has been performed at **Computer Architecture and VLSI** laboratory, **Institute of Computer Science (ICS)**, **Foundation for Research and Technology – Hellas (FORTH)**, and is partially supported by the CoherentPaaS (FP7-611068) and PaaSage (FP7-317715) EU projects.

University of Crete
Computer Science Department

A Distributed Key-Value Store based on Replicated LSM-Trees

Thesis submitted by
Panagiotis Garefalakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________
Panagiotis Garefalakis

Committee approvals: ______________________
Angelos Bilas
Professor, Thesis Advisor

____
Kostas Magoutis
Assistant Professor, Thesis Supervisor

____
Dimitris Plexousakis
Professor, Committee Member

Departmental approval: ______________________
Angelos Bilas
Professor, Director of Graduate Studies

Heraklion, July 2014

Abstract

Distributed highly-available key-value stores have emerged as important building blocks for data-intensive applications. Eventually-consistent versions of such stores have become popular due to their high availability (*"always writeable"*) features; they are however unsuitable for many applications that require strong consistency. In this thesis we describe the design and implementation of *ACaZoo*, a key-value store that combines strong consistency with high performance and high availability. *ACaZoo* supports the popular column-oriented data model of Apache Cassandra and HBase. It implements strongly-consistent data replication using primary-backup atomic broadcast of a write-ahead log, recording data mutations to a Log-structured Merge Tree (LSM-Tree). *ACaZoo* scales by horizontally partitioning the key space via consistent primary-key hashing over replica groups (RGs).

LSM-Tree compactions can hamper performance, especially when they take place at RG primaries. ACaZoo addresses this problem by changing RG leadership prior to heavy compactions, a method that can improve throughput by up to 60% in write-intensive workloads. To further improve response time *ACaZoo* uses client-side routing of requests, which is known to complicate the propagation of configuration changes to a large and dynamic client population. We address this problem by proposing an optimized version of ZooKeeper that can load-balance issuing of change notifications across its servers. We evaluate *ACaZoo* using the Yahoo Cloud Serving Benchmark (YCSB) and compare it to Oracle's NoSQL Database and to Cassandra providing serial consistency via an extension of the Paxos algorithm. We further evaluate application performance using *CassMail*, a scalable e-mail service, over both *ACaZoo* and Cassandra.

Περίληψη

Τα τελευταία χρόνια, τα κατανεμημένα συστήματα αποθήκευσης τα οποία προσφέρουν υψηλή διαθεσιμότητα αποτελούν αναπόσπαστο κομμάτι εφαρμογών που επεξεργάζονται μεγάλο όγκο δεδομένων. Εκδόσεις των εν λόγω συστημάτων αποθήκευσης που προσφέρουν χαλαρή συνέπεια δεδομένων έχουν γίνει δημοφιλείς λόγω της υψηλής διαθεσιμότητας τους, είναι όμως ακατάλληλες για εφαρμογές που απαιτούν ισχυρή συνέπεια. Στην εργασία αυτή περιγράφουμε το σχεδιασμό και την υλοποίηση του *ACaZoo*, ενός συστήματος αποθήκευσης δεδομένων που συνδυάζει ισχυρή συνέπεια με υψηλή απόδοση και διαθεσιμότητα. Το ACaZoo υποστηρίζει το δημοφιλές μοντέλο δεδομένων των Apache Casasandra και HBase και υλοποιεί συνεπή ομοιοτυπία δεδομένων (data replication) χρησιμοποιώντας πρωτόκολλα της οικογένειας Paxos καταγράφοντας τις αλλαγές δεδομένων που γίνονται σε ένα LSM-Tree. Το ACaZoo κλιμακώνεται χρησιμοποιώντας όλες τις διαθέσιμες ομάδες αντιγράφων (ΟΑ) και κατανέμει τα δεδομένα στις ΟΑ με βάση το πρωτεύων κλειδί.

Η συγχώνευση δεδομένων στα LSM-Trees μπορεί να μειώσει δραστικά την απόδοση του συστήματος όταν λαμβάνει χώρα στον αρχηγό μιας ΟΑ. Το ACaZoo αντιμετωπίζει αυτό το πρόβλημα εκλέγοντας νέο αρχηγό πριν από κάθε τέτοια λειτουργία, μια μέθοδος που μπορεί να βελτιώσει την απόδοση έως και 60%. Για την περαιτέρω βελτίωση του χρόνου απόκρισης το ACaZoo δρομολογεί τις αιτήσεις από την πλευρά του πελάτη, μια διαδικασία η οποία μπορεί να καθυστερήσει την διάδοση των αλλαγών σε ένα μεγάλο και δυναμικό πληθυσμό πελατών. Αντιμετωπίσαμε αυτό το πρόβλημα προτείνοντας μια βελτιστοποιημένη έκδοση του συστήματος Zookeeper που κατανέμει ισότιμα τους πελάτες στους διακομιστές του. Για την πειραματική μας μελέτη χρησιμοποιήσαμε το Yahoo Cloud Serving Benchmark (YCSB) και συγκρίναμε την απόδοση του συστήματος μας με τα συστήματα NoSQL Oracle Database και Apache Cassandra. Τέλος μελετήσαμε την απόδοση του CassMail, ενός κλιμακώσιμου e-mail service, χρησιμοποιώντας το Casssandra και το ACaZoo.

Acknowledgements

I would like to thank all the people that influenced me during this wonderful journey.

First of all my supervisor Prof. Kostas Magoutis, for his continuous support in my work and studies, for giving me the chances he did, and most importantly for showing me the right path through critical thinking and hard work. Also the remaining members of my committee, Angelos Bilas and Dimitris Plexousakis for the valuable comments and questions during my defence.

Next I would like to thank Institute of Computer Science (ICS), Foundation of Research and Technology – Hellas (FORTH) and more specifically Computer Architecture and VLSI (CARV) laboratory for the support during both my undergraduate and graduate studies.

My best thanks to my friends and colleagues Antonis Papadogiannakis, Giorgos Vasiliadis, Antonis Krithinakis, Lazaros Koromilas, Thanasis Petsas, Damianos Metalidis, Christos Papoulas, Laertis Loutsis and Evangelos Ladakis for making the lab a fun and interesting place.

I would like to express my appreciation to Panagiotis Papadopoulos for his team spirit and his excellent cooperation in DAIS'13 and SRDS'14 papers.

Last but not least I am more than grateful to my parents Georgia and Giannis and my sister Eirini for their endless love, support and encouragement through my studies and my life in general. I wouldn't have made it this far without you..

This work has been performed at **Computer Architecture and VLSI** laboratory, **Institute of Computer Science (ICS)**, **Foundation for Research and Technology – Hellas (FORTH)**, and is partially supported by the Coherent-PaaS (FP7-611068) and PaaSage (FP7-317715) EU projects.

An early report of this work appeared in the proceedings of the 13th International IFIP Conference on Distributed Applications and Interoperable Systems (DAIS 2013), June 2013 [1] and a later report will appear in the proceedings of the 33rd IEEE Symposium on Reliable Distributed Systems (SRDS 2014), October 2014 [2].

Contents

1 Introduction **1**
- 1.1 Thesis Contributions . . . 3
- 1.2 Thesis Organization . . . 3

2 Background **5**
- 2.1 Replication . . . 5
- 2.2 A primary-backup replication system . . . 7
- 2.3 NoSQL Systems . . . 8
- 2.4 LSM Trees . . . 11
  - 2.4.1 Write Requests . . . 11
  - 2.4.2 Read Requests . . . 12
  - 2.4.3 Compactions . . . 13
- 2.5 Consistency . . . 13

3 Related Work **17**

4 System Design **19**
- 4.1 ACaZoo architecture . . . 19
- 4.2 ACaZoo replication . . . 20
- 4.3 RG leadership change . . . 21
- 4.4 Load balancing client notifications . . . 21

5 Implementation **23**
- 5.1 ACaZoo . . . 23
- 5.2 Client-coordinated I/O and configuration management . . . 26
- 5.3 Dynamic load balancing requests . . . 27
- 5.4 ZooKeeper data path and optimizations . . . 28

6 Evaluation **31**
- 6.1 Performance impact of LSM-tree compactions . . . 32
- 6.2 Performance of a 3 node replication group . . . 34
- 6.3 Performance of a 5 node replication group . . . 35
- 6.4 Timing of compactions across replicas . . . 37
- 6.5 Timing of garbage collections across replicas . . . 38

6.6 Availability of RG under leader failure 38
6.7 Load balancing notifications 40
6.8 Application performance 42
6.9 Impact of client-coordinated I/O 44

7 Future Work **45**

8 Conclusions **47**

List of Figures

2.1 Replication strategies 6
2.2 ZooKeeper 7
2.3 Partitioning and replication of keys in Dynamo and Cassandra [3] . 10
2.4 Column data model used in systems such as BigtTable, Cassandra and ACaZoo 11
2.5 Schematic picture of an LSM-tree of two components [4] 12
2.6 Schematic picture of an LSM-tree compaction [4] 13
2.7 Paxos Basic [5] 15
2.8 Paxos with commit and current value read [5] 16

4.1 The ACaZoo architecture 19
4.2 ZAB-based replication of Cassandra's write-ahead log 20

5.1 System components and their interactions 24
5.2 ACaZoo storage server 25
5.3 ZooKeeper data path 28

6.1 Cassandra schema designed for CassMail [6] 32
6.2 CassMail system design [6] 33
6.3 34
6.4 Openstack - 3 nodes Replication Group 35
6.5 Openstack - 5 nodes Replication Group 37
6.6 ACaZoo compaction behaviour under different replication groups (RG) (a) 3 nodes RG (b) 5 nodes RG (c) 7 nodes RG 39
6.7 ACaZoo garbage collection behaviour under different replication groups (RG) (a) 3 nodes RG (b) 5 nodes RG (c) 7 nodes RG . . . 40
6.8 YCSB throughput of ACaZoo under leader failure 41
6.9 YCSB throughput of Oracle NoSQL under leader failure 41
6.10 Client notification latency from ZooKeeper 42
6.11 CassMail: 3 Node Replication Group 43
6.12 CassMail: 5 Node Replication Group 43

Chapter 1

Introduction

The ability to perform large-scale data analytics over large data sets has in the past decade proved to be a competitive advantage in a wide range of industries (retail, telecom, defence, etc.). In response to this trend, the research community and the IT industry have proposed a number of platforms to facilitate large-scale data analytics. Such platforms include a new class of databases, often referred to as NoSQL data stores, which trade the expressive power and strong semantics of long established SQL databases for the specialization, scalability, high availability, and often relaxed consistency of their simpler designs.

Companies such as Amazon [3] and Google [7] and open-source communities such as Apache [8] have adopted and advanced this trend. Many of these systems achieve availability and fault-tolerance through data replication. Google's BigTable [7] is an early approach that helped define the space of NoSQL *key-value* data stores. Amazon's Dynamo [3] is another approach that offers an eventually consistent replication mechanism with tunable consistency levels. Dynamo's open-source variants Cassandra [8] and Voldemort [9] combine Dynamo's consistency mechanisms with a BigTable-like data schema. These systems use consistent hashing to ensure a good distribution of key ranges (data partitions, or *shards*) to storage nodes. The applications of key-value stores include email systems [6], file systems [10] and data analysis [11].

Eventual consistency works well for applications that have relaxed semantics (such as maintaining customer carts in online stores [3]) but is not an option for a broad spectrum of applications that require strong consistency. When embarking on the *ACaZoo* project we decided to target strongly-consistent sharded data-intensive applications, a large and growing class of applications. One of our design goals for *ACaZoo* was to use a standard data model and cross-platform API. We thus opted for the Apache Cassandra/HBase column-oriented data model and the Cassandra Thrift-based API. Another design goal was to use a consistent-hashing based scheme for shard partitioning due to its simple implementation and lack of a centralized metadata service. Finally, we wanted to leverage a Log-structured Merge (LSM)-Tree [4] based storage backend due to its benefits over

B+-tree organized schemes in organizing local storage, combined with a consistent replication scheme.

LSM-Trees is a particularly attractive indexing scheme used in a variety of systems (Bigtable [7], HBase [12], Cassandra [8], Hyperdex [13], PNUTS [14], etc.). In ACaZoo we decided to combine LSM-Trees with a strongly-consistent primary-backup (PB) scheme (ZAB [15], also found at the core of Apache ZooKeeper). A well known limitation of primary-backup schemes however is the requirement to go through a single master of the replica group (RG) for both read and write operations. Besides being a scalability limitation (which can be addressed by sharding over several RGs), performance can be hampered at times by periodic background activity at the master. Implementations of LSM-Trees are prone to such a challenge, especially under write-intensive workloads, as compaction operations aiming to merge data files into a smaller set drain server CPU and I/O resources.

In this thesis we propose a solution to this problem that leverages the fact that compaction schedules of the nodes in a RG usually have little overlap (we experimentally validated this claim but could also enforce it if needed). Our solution ensures that the RG master is never a node that undergoes significant compaction activity. We achieve this by forcing a reconfiguration of an RG when the master is about to start a heavy compaction. Finally our solution ensures that the impact of reconfigurations on overall performance is low. This is achieved by rapidly propagating the change to clients, piggybacked as responses to standard RPCs. Our experiments show that compaction activity at the master hurts performance and that compaction-aware RG reconfiguration policies can lead to a significant performance improvement.

In *ACaZoo* we also apply an orthogonal optimization, client-coordination of I/O requests, as a means of reducing response time and relieving servers from forwarding I/O traffic. This method has been investigated and has been shown to provide benefits in previous systems (Amazon Dynamo [3] and Petal [16] are but a few of them). The drawback of this scheme is the need to rapidly update a large and potentially dynamic client population. Configuration services such as ZooKeeper and Chubby can fill this need, operating in either *pull* (clients call into the service) or *push* (service notifies clients) mode. The push mode is expected to result in lower response time, however the service should be able to efficiently handle the load of notifying a large client base. While solutions such as a DNS frontend can load balance notification requests (also termed *watch* events), client churn may result into an unbalanced system (where some servers have many more notifications to perform compared to others). In this thesis we experimentally demonstrate that a well balanced ZooKeeper cell can result in 42% better notification response time compared to an unbalanced system and sketch an algorithm with which ZooKeeper can internally load balance client requests (obviating the need for a DNS frontend).

1.1 Thesis Contributions

In this thesis, we make the following main contributions:

- A high-performance data replication primitive combining the ZAB [17] protocol with a single-node implementation of LSM-Trees [4]; while in principle similar to previous data replication approaches such as SMARTER [18] and BookKeeper [19], ACaZoo combines log replication with checkpointing using LSM-Trees [4] and addresses the associated challenges.

- A novel technique that addresses the impact of LSM-Tree compactions on write performance by forcing reconfigurations of RGs, changing leadership prior to heavy compactions at the master. Our experiments show that this technique can improve throughput by up to 60% in write-intensive workloads.

- A dynamic load balancing technique for client notification across ZooKeeper replicas. This technique is shown to be an effective solution improving response time up to 42% in large populations of clients and also scalable with larger ZooKeeper replica groups.

1.2 Thesis Organization

The rest of this thesis is organized as follows: In Chapter 2 and we provide background and in Chapter 3 we relate ACaZoo to other work in the field. In Chapter 4 we describe the overall design and in Chapter 5 we provide details of our implementation. In Chapter 6 we present the results of our system evaluation. and in Chapter 7 directions of ongoing and future work. Finally in Chapter 8 we conclude.

Chapter 2

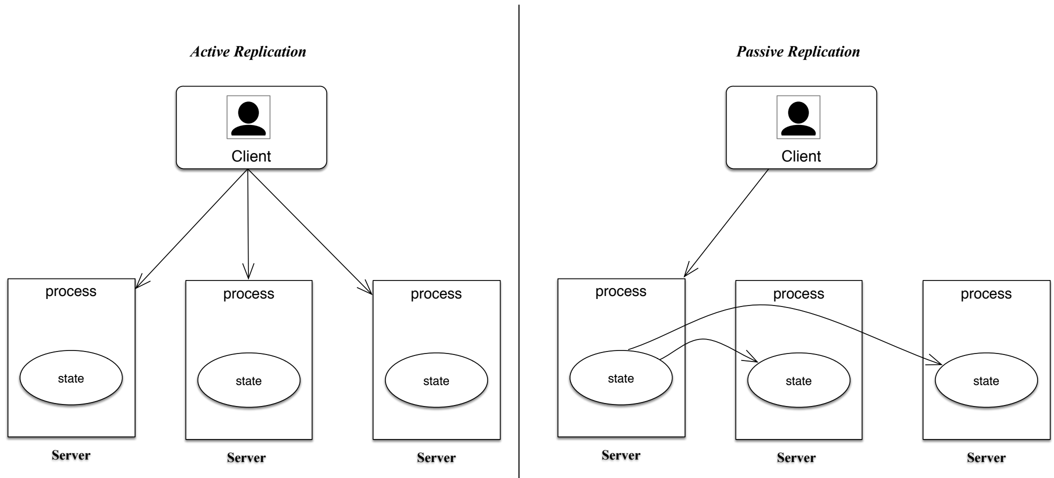
Background

2.1 Replication

Large-scale distributed storage systems usually built over failure-prone commodity components. Failures are quite common in those systems, and replication is often the solution to data reliability and high availability. The main approaches for implementing replication are: state machine replication [20], primary backup [21], process-pairs [22] and quorum systems [23]. A variety of configuration management systems based on the replicated state machine approach have recently emerged. These systems use distributed consensus algorithms such as Paxos at the core [24, 25]. Systems such as Petal [16] and Niobe [26] also use Paxos implementations for maintaining configuration information while other systems such as Chubby [27] Oracle NoSQL [28] and Apache Cassandra [8] use Paxos as an underlying technology mainly used for locks, leases and leader elections. Apache ZooKeeper [19] is another practical example of a system built on a Paxos-like distributed protocol. Like Chubby, ZooKeeper exposes a file-system like API and is frequently used for leader election, locks, cluster membership services, service discovery and assigning ownership to partitions in distributed systems.

State machine replication, Figure 2.1(a), also known as *active replication*, is a general protocol for replication management that has no centralized control. Active Replication was first introduced by Leslie Lamport [25]. State machine replication requires that the process hosted by the servers is deterministic. Deterministic means that, given the same initial state and a request sequence, all processes will produce the same response sequence and end up in the same final state. In order to make all the servers receive the same sequence of operations, an atomic broadcast protocol must be used. An atomic broadcast protocol guarantees that either all the servers receive a message or none, plus that they all receive messages in the same order. From the client point of view, all correct replicas should appear having the same state.

Another approach is the primary-backup approach, Figure 2.1(b), also known as *passive replication.* In this approach one server is designated as the primary (or



(a) Active Replication (b) Passive Replication

Figure 2.1: Replication strategies

leader), while all the others are backups (or followers). After processing a request, the primary server updates the state on the other (backup) servers and sends back the response to the client. If the primary server fails, one of the backup servers takes its place (also termed fail-over). This scheme ensures linearizability due to the order in which the primary receives invocations defines the order for all servers. The acknowledgement sent by the backups and awaited by the primary ensures request atomicity. Unlike state machine replication, primary-backup replication does not waste extra resources through redundant processing, and permits non-deterministic operations. However, responses will be delayed by the failure of the primary. Furthermore, primary-backup replication requires additional application support for the primary to update the state of the other copies.

Another important method for achieving replication is process pairs. One of the systems first used this method is Tandem [22] which could tolerate partial failures with the advantage of hiding any failure events from the clients. Process pairs in general consist of two identical processes that run independently and maintain identical state. When one of these processes fail the other one continues processing requests providing the illusion that the service never fails.

Finally quorum systems offer another alternative of replicating data, ensuring consistency under network partitions. They were firstly introduced by Gifford [23] who demonstrated an algorithm for storing data in multiple hosts using votes. Each copy of a replicated data item is assigned a vote. Each operation then has to obtain a read quorum R or a write quorum W to read or write a data item. If a given data item has a total of V votes, the quorums have to follow the rules:

$$R + W > V \tag{2.1}$$

$$W > V/2 \tag{2.2}$$

Rule 2.1 ensures that a data item is not read and written by two transactions concurrently. Rule 2.2 ensures that two write operations on the same data item from two transactions cannot occur concurrently.

2.2 A primary-backup replication system

In this section we will describe an instance of a primary-backup implementation we used in this thesis, Zookeeper [19]. Zookeeper implements a hierarchical namespace of fixed-size objects (referred to as *znodes*) accessed via a filesystem-like API. The Zookeeper namespace is organized as a hash-table memory structure kept consistent across a set of servers called Quorum Peers (QPs). The set of QPs is referred to as a cell. A cell is organized as a single leader and number of followers as shown in Figure 2.2. Each request (otherwise known as a proposal) towards a *znode* corresponds to a transaction with a specific ID (referred to as a zxid) eventually committed into a Commit Log. Consistency is achieved via the ZAB atomic-broadcast protocol [17] a variation of Paxos algorithm. Similar to Paxos, ZAB consists of two parts; the broadcast protocol executed by the leader guaranteeing sequential consistency semantics and an atomicity protocol ensuring that there is only one message per proposal number simplifying message recovery. The fundamental difference between Paxos and ZAB is that the former is designed for state machine replication while the latter for primary-backup systems. In state-machine replication, replicas must agree on the execution order of client requests,

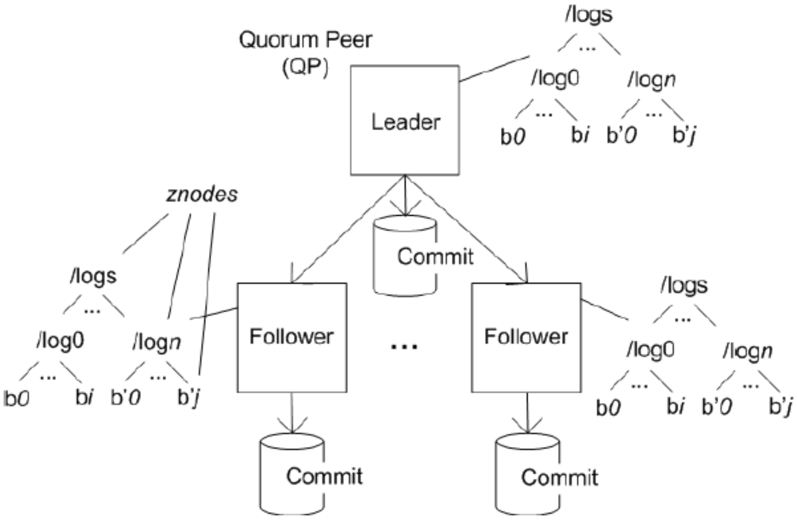


Figure 2.2: ZooKeeper

even if these requests are submitted concurrently by clients and received in different orders by the replicas, for example request A must always be applied before B. This is achieved by Paxos. On the other hand with ZAB replicas can concurrently agree on the order of multiple state updates without harming correctness, for example request C can be applied before B. This is achieved by adding one more synchronization phase during recovery compared to Paxos, and by using a different numbering of instances based on zxids as described by F. Zunqueira et al [17]. *Znodes* do not map to persistent locations on disk. Instead, the entire namespace is periodically serialized and snapshotted to disk. Zookeeper can thus recover *znode* state by loading the most recent snapshot and running its commit log, up to the most recent committed transaction. The total amount of data Zookeeper can store is bounded by the physical memory of the least-provisioned node in the system.

The leader is connected to each follower through direct FIFO channels (implemented as TCP streams) in a tree pattern. TCP connections between QPs in the chain are persistent across proposals. Each leader receives proposals from clients, moves them out of sockets, forwards them to all followers and then writes them to disk. Upon receiving a proposal, a follower writes it to disk and then acknowledges with the leader. The leader commits a proposal and responds successfully to clients only after it has received ACKs from a majority of followers. To avoid overload the leader uses an application-level flow control protocol (i.e., stop receiving data from TCP sockets) to throttle clients when the queue of outstanding (not yet committed) proposals exceeds a configurable threshold. A group commit protocol comes into action to avoid the cost of flushing dirty buffers to disk at each operation.

2.3 NoSQL Systems

The development of the Internet and cloud computing made access to data easier than ever. Services access and capture data through third parties such as Facebook, Google and others. Such services need to process that large amount of data effectively creating a new need which SQL databases were never designed for. *NoSQL* is increasingly considered a viable alternative to relational databases when there is the need for big data and real-time web applications. A *NoSQL* or *Not Only SQL* database provides a mechanism for storage and retrieval of data that is modelled differently than the tabular relations used in relational databases. Motivations for this approach include simplicity of design, horizontal scaling and finer control over availability. By design, *NoSQL* databases and management systems are schema-less. They are not based on a single model, like a relational model, and each database, depending on their target-functionality, adopts a different one. There is a variety of operational models for *NoSQL* databases. Nevertheless a basic classification based on the data model is listed below with the most popular examples in each category:

- *Key/Value*: In key-value-store category of *NoSQL* database, a user can store data in schema-less way. A key may be strings, hashes, lists, sets, sorted sets and values are stored against these keys. Popular systems: Dynamo [3], Redis [29]

- *Column*: A column is a key value pair, where the key is an identifier and the value contains values related to the key (identifier). Popular systems: Cassandra [8], HBase [12]

- *Document*: Data is stored as documents which can be variable in terms of structure (JSON, XML and others). Popular systems: MongoDB [30], Couchbase [31]

- *Graph*: Data is stored as a collection of nodes, where nodes are analogous to objects in a programming language. Nodes are connected using edges. Popular systems: OrientDb [32], Neo4j [33]

Column based data stores are extremely powerful and they can be reliably used to keep important data of very large sizes. We chose to implement our work in Apache Cassandra, which is a Dynamo-based distributed data store replicating data using a combination of the approaches described in 2.1. Cassandra uses the peer to peer distribution model ensuring that there is no single point of failure and every node's functionality in the cluster is identical. Concerning the distribution model, inherited from Amazon's Dynamo [3], nodes in the cluster can be considered to be located in a ring. Nodes partition the data based on the row key. Each Cassandra node is assigned a unique token that determines what keys it is responsible for. Based on the replication factor one or more nodes can be responsible for each token and we can find them by walking the ring clockwise. For example in Figure 2.3, node B replicates the key K at nodes C and D in addition to storing it locally (*replication factor = 3*). Node C will store the keys that fall in the ranges (G, A], (A, B], and (B, C], and node D will store the keys that fall in the ranges (A, B], (B, C], and (C, D].

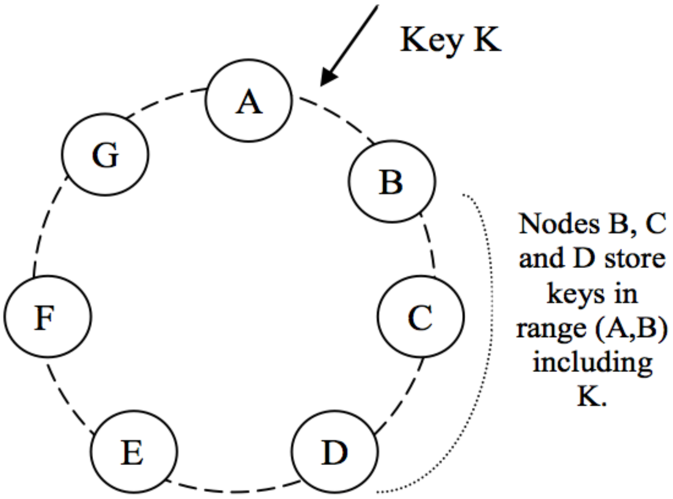


Figure 2.3: Partitioning and replication of keys in Dynamo and Cassandra [3]

Inspired by BigTable [7] Cassandra's data model is a schema-optional, column-oriented data model. This means that, unlike a relational database, you do not need to model all of the columns required by an application up front, as each row is not required to have the same set of columns. Columns can be added by an application at runtime. In Cassandra, the keyspace is the container for application's data, similar to a database or schema in a relational database. Inside the keyspace there are one or more column family objects, which are analogous to tables. *Column families* contain *columns* and a set of related columns is identified by a *row key*. In each *row* data are stored in the basic storage unit which is a *cell*. Cassandra similar to Bigtable allows multiple timestamp versions of data within a cell. A cell can be addressed by its' *row-key*, *column family*name, *Column qualifier* and the version as shown in Figure 2.4. At the physical level, data are stored per column family contiguously on disk sorted by row-key, column name and version (column oriented data model).

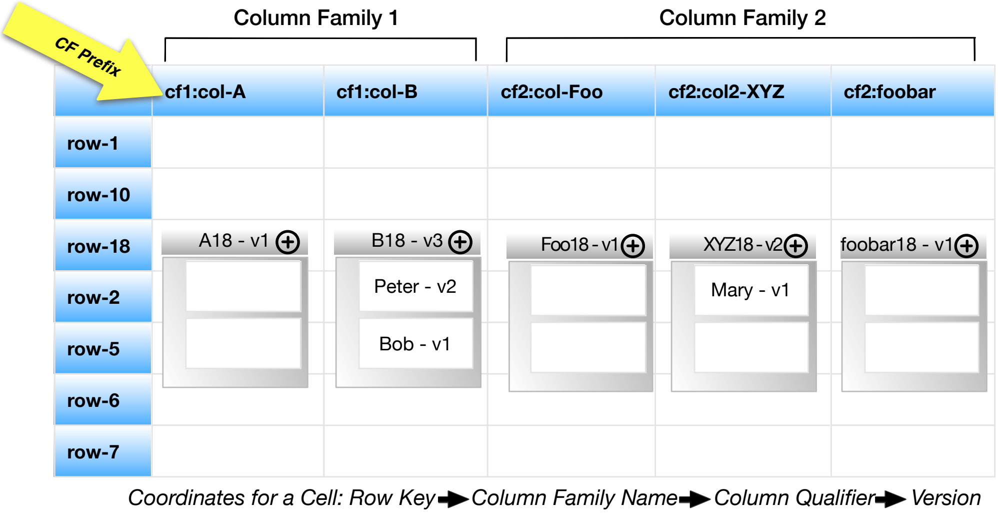

Column Family 1
Column Family 2
CF Prefix
cf1:col-A
cf1:col-B
cf2:col-Foo
cf2:col2-XYZ
cf2:foobar
row-1
row-10
row-18
row-2
row-5
row-6
row-7
A18 - v1
B18 - v3
Peter - v2
Bob - v1
Foo18 - v1
XYZ18 - v2
Mary - v1
foobar18 - v1
Coordinates for a Cell: Row Key → Column Family Name → Column Qualifier → Version

Figure 2.4: Column data model used in systems such as BigtTable, Cassandra and ACaZoo

2.4 LSM Trees

Cassandra uses the LSM-tree data structure as its storage back-end. LSM-tree offers faster insertion performance than B+-tree variants and it is also the choice of other systems such as HBase [12], BigTable [7] and Hypertable [34]. LSM-tree is actually a collection of two or more tree-like components but it is being treated as a single key-value store. In a two component LSM-tree as shown in Figure 2.5, a smaller component will be entirely memory resident, the C0 tree, and a larger component will reside on disk, the C1 tree. Despite being disk resident, frequently referenced pages from C1 will remain in cache and as a result they can be considered in memory. When a new row is inserted, a log record to recover this insert is first written to the sequential log file (CommitLog). Then an index entry for that row is inserted into C0 tree, in memory, after which it will eventually migrate to C1 tree, on disk. Inserting data is done sequentially in the append only LSM-tree and this is the main reason writes are performing that fast.

2.4.1 Write Requests

When a record is being inserted into the LSM-tree, it will first enter the C0 in-memory tree, then with the merge process will eventually be shifted to Ck as depicted in Figure 2.6. In Cassandra more specifically, writes are first written to the CommitLog structure which is append only, then they are batched in memory,

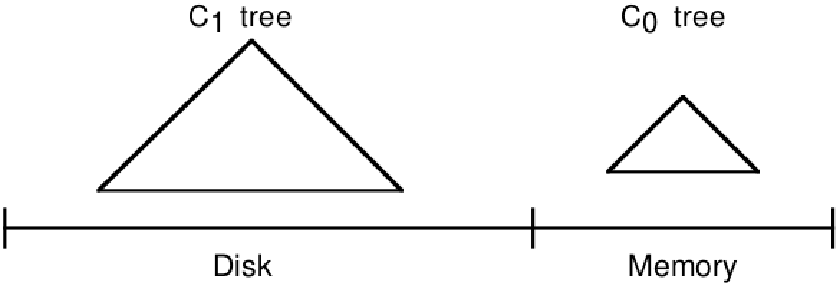


Figure 2.5: Schematic picture of an LSM-tree of two components [4]

to a structure called Memtable, and they are periodically written to disk to a persistent table structure called SSTable (Sorted String Table), an implementation of LSM-trees. Memtables and SSTables are maintained per column family. Memtables are organized in sorted order by row key and flushed to SSTables sequentially, resulting to sequential seeking while relational databases perform random seeking. SSTables are immutable meaning that they can not be modified after they have been flushed. This means that a row is typically stored across multiple SSTable files.

Cassandra allows configuring CommitLog to be synchronized periodically, by default every 10 seconds, with the potential cost of losing some data in case of a crash. Alternatively the batch mode, which is fully durable, can be selected. In this mode Cassandra guarantees that it synchronizes CommitLog before acknowledging writes by fsyncing it to disk. In batch mode the system waits for other writes for a period based on the configuration before performing the sync. Batch mode durability comes with additional I/O overhead causing noticeable performance drop. As a result in batch mode it is recommended to store CommitLog in a separate device to reduce the I/O cost.

2.4.2 Read Requests

Whenever a read request is received, the system will first search the Memtable by its row key to see if it contains the data. If not, it will look at the on-disk SSTable to see if the row-key is there. This technique is called ”merged read” as the system needs to look at multiple places for the data. To optimize this process, Cassandra uses an in-memory structure called a bloom filter. Each SSTable has a bloom filter associated with it. The bloom filter is used to check if a requested row key exists in the SSTable before performing any disk seeks.

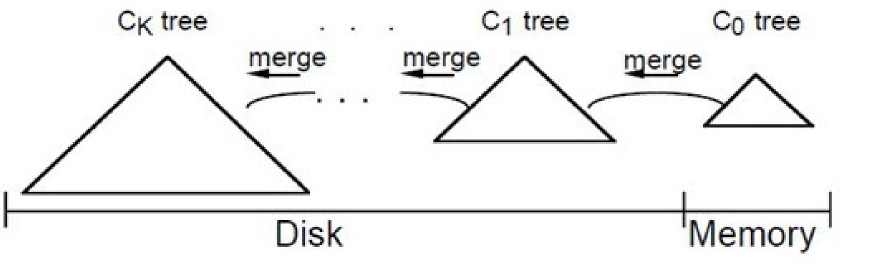


Figure 2.6: Schematic picture of an LSM-tree compaction [4]

2.4.3 Compactions

To improve read performance and save disk space, Cassandra periodically merges SSTables into larger SSTables using a background process called compaction. Compaction merges same rows together. Since the SSTables are sorted by row key, this merge does not require random disk I/O but it is still a time and resource consuming operation. There are two types of compactions, minor and major which do not differ in functionality, but rather in purpose. A minor compaction is when the merging occurs to bound the number or size of SSTables on storage. A major compaction operates similarly to a minor compaction except that it is scheduled to run periodically and is intended to ensure that all SSTables are eventually compacted within a set time-period.

2.5 Consistency

Inherited by distributed systems, NoSQL systems follow the same basic principles with the most fundamental defined by the CAP theorem. The CAP theorem, or Brewer's theorem [35] states that out of consistency, availability and partition tolerance, it is possible for a system to simultaneously provide only two of these three guarantees. Cassandra is designed for availability, partition tolerance therefore is offering weaker consistency, also termed as eventual consistency. Eventual consistency guarantees that if no further updates occur, eventually the system will become consistent. While many applications can tolerate a small delay in which a newly updated value may be inconsistent, eventual consistency does not provide any bound on this delay.

Similar to Dynamo, in Cassandra each key is randomly assigned to N replicas depending on the replication factor. Every write to the key is sent to all N replicas. When the coordinator issuing the write receives acknowledgements from Wreplicas, it sends an acknowledgement to the client application. The rest of the N replicas receive the write asynchronously in the background. We consider a read to be consistent only when it returns the latest value for which W acknowledgements have been received. Cassandra offers a number of different consistency levels with ANY being the lowest consistency with the highest availability, and

ALL being the highest consistency but with the cost of the lowest availability. QUORUM is an other alternative ensuring consistency, but still tolerating some level of failure. A quorum is calculated as: *(replication_factor / 2) + 1* , rounded to a whole number. Consistency levels in Cassandra can be selected per query basis.

Sometimes though, even the highest consistency defined in Cassandra is not enough for some applications. These applications require one request to be performed at a time and make sure when we run concurrent queries that we will get the same results as if they were really processed independently. This property is called linearizable consistency. This feature was recently introduced in Cassandra version 2.0 under the name *Lightweight Transactions*, also termed *Serial Consistency*, and it was implemented using an extension of Paxos algorithm. First I will explain the basic Paxos algorithm and then I will introduce *Lightweight Transactions* through an extension of the basic algorithm.

The protocol depicted in Figure 2.7 is the core algorithm of Paxos. Each instance of the basic Paxos protocol decides on a single value. The protocol proceeds over several rounds. A successful round has two phases.

In phase one a Leader, also termed as Proposer, creates a proposal identified with a ballot number B. This number must be greater than any previous proposal number used by this Leader. Then, the Leader sends a Prepare message containing this proposal to a quorum of Replicas (also knows as Acceptors). If the proposal's number B is higher than any previous proposal number received, they return a promise to ignore all future proposals having a number less than B.

In the second phase, if a quorum of Replicas promise to accept the leader's proposal, the Leader may proceed to the actual proposal of a specific value. If a Replica receives an accept request message for a proposal B, it must accept it only if and only if it has not already promised to consider proposals having an identifier greater than B. In this case, it should store the corresponding value and send back an accept message to the Leader.

In distributed systems one way to achieve linearizable consistency using Paxos is by routing all requests through a single master (Leader), similar to primary-backup systems. Although in a fully distributed master-less system like Cassandra, it is less obvious.

Paxos gives us the ability to agree on exactly one proposal. After one has been accepted, it will be returned to future leaders in the promise, and the new Leader will have to re-propose it again. In the extension of the Paxos algorithm depicted in Figure 2.8 there is a proposed a way to "reset" the Paxos state for the future proposals (one Leader per proposal). In order to achieve that and finally move the accepted values into Cassandra storage, a third phase of *commit/acknowledges* was added to the algorithm. As this functionality was exposed as a compare-and-set operation, there was a forth phase added to the algorithm reading the current value of the row from all Replicas checking if it matches the expected one *read/results*. This is how linearizability consistency was implemented in Cassandra 2.0 also paying the hight price of four round trips per (linearizable) operation.

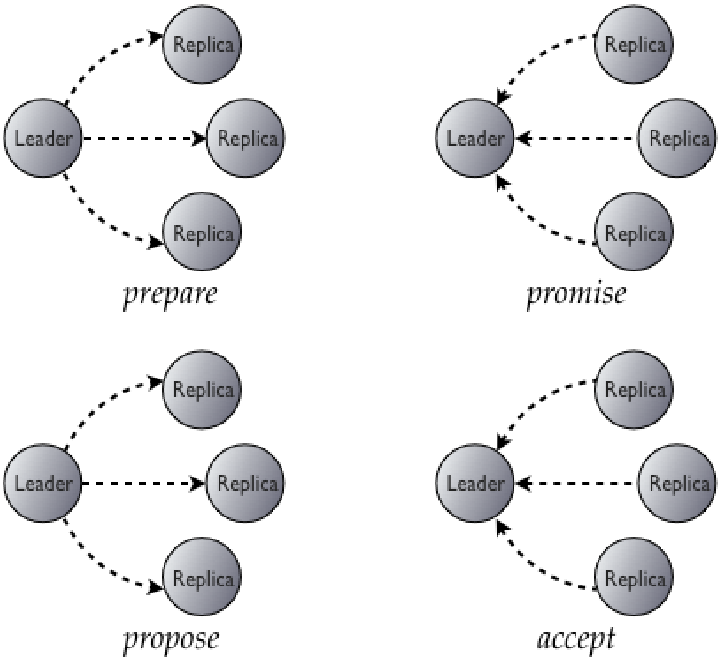

Replica
Leader
Replica
Replica
prepare
Replica
Leader
Replica
Replica
promise
Replica
Leader
Replica
Replica
propose
Replica
Leader
Replica
Replica
accept

Figure 2.7: Paxos Basic [5]

The problem of providing strong consistency without sacrificing availability and partitioning tolerance is orthogonal and open to different solutions. For example the solution Cassandra adopted comes with the high price of four round trips per write operation which means higher response time to the clients. The approach we adopted in *ACaZoo* has the benefit that it extends an already popular and widely used system, Cassandra and provides a simple solution using a very well known primary backup system, Zookeeper [17]. Existing implementations of LSM-Trees such as Cassandra are prone to compaction operations, which drain server CPU and I/O resources. In our system we propose a solution to this problem that leverages the primary backup replication scheme and ensures that the RG master is never a node that undergoes significant compaction activity. Finally in *ACaZoo* we apply another optimization by client-coordinating I/O requests, reducing response time and relieving configuration servers from forwarding I/O traffic. We also demonstrate the benefit of a load balanced configuration service resulting to better response time compared to an unbalanced one.

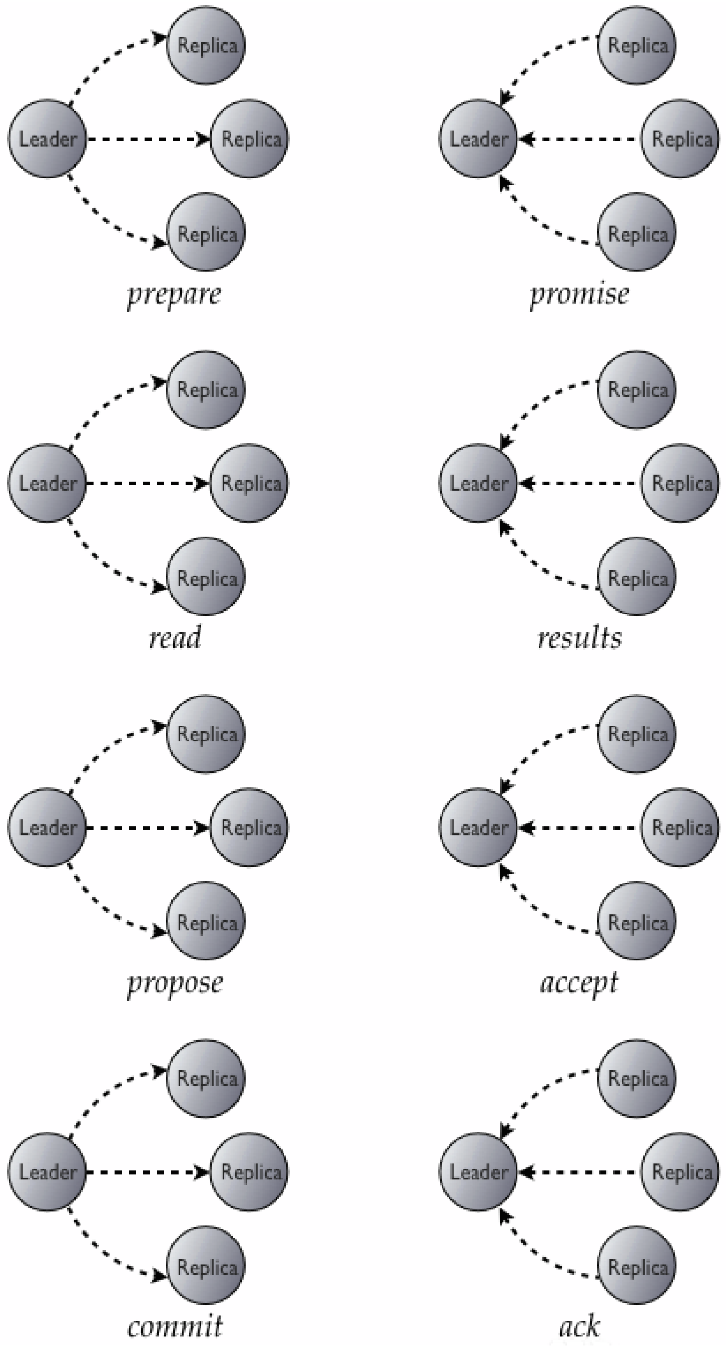

Leader
Replica
prepare
promise
read
results
propose
accept
commit
ack

Figure 2.8: Paxos with commit and current value read [5]

Chapter 3

Related Work

Our system is related to several existing distributed NoSQL key-vale stores [3, 7, 8] implementing a wide range of semantics, some of them using the Paxos algorithm [36] as a building block [27, 16, 26]. Most NoSQL systems rely on some form of relaxed consistency to maintain data replicas and reserve Paxos to the implementation of a global state module [16, 26] for storing infrequently updated configuration metadata or to provide a distributed lock service [27]. Exposing storage metadata information to clients has been proposed in the past [3, 16, 28], although the scalability of updates to that state has been a challenge.

There have recently been several approaches to high performance replication within a local area network environment (a datacenter). SMARTER [18] implements a highly-available data store using an optimized Paxos-based replicated state machine and has demonstrated performance close to the hardware limits and 12%-69% better than primary-backup versions. SMARTER replicates a periodically-checkpointed stream store whereas ACaZoo's storage backend and checkpointing mechanism is based on Cassandra's implementation of LSM Trees [4]. Cassandra recently (as of version 2.0) implemented a linearizable consistency mode using an extension of the Paxos protocol to reach consensus at each insert or update request [5]. As is validated by our experiments, this implementation (termed *Cassandra Serial*) incurs a significant performance penalty over competing systems in write-intensive workloads.

A recent work that proposes explicit replication of LSM-Trees is Rose (Sears *et al.* [37]). Rose shares our goal of leveraging LSM-Trees' superior write performance in a replication context; they however focus more on the benefits of data compression and less on the performance impacts of compaction. Google's BigTable can be credited with bringing LSM-Trees to the forefront since their inception in 1996 [4] and Apache HBase for contributing an open source implementation, in both cases over a distributed replicated file system. While similar in principle, *ACaZoo* differs from these two systems in its explicit management of replication as opposed to implementing a serial LSM-Tree and relying on an underlying layer for replication.

Several systems have experimented with sharding and replication over open-source storage engines such as MySQL, including Google's F1 [38], LinkedIn's Espresso [39], and Facebook's TAO [40], with varying results. F1 faced a problem with rebalancing data partitions under expanded capacity (and eventually opted to use Spanner [41]), whereas Espresso attempted to address these problems by avoiding partition splits (by overpartitioning), mapping several partitions to a single MySQL instances, and modifying MySQL to be aware of partition identities on log updates. Espresso uses *timeline consistency* (asynchronous or semi-synchronous replication) whereas F1 is a strongly-consistent system (synchronous replication).

Perhaps the closest approaches to ours are Scatter [42], ID-Replication [43], and Oracle's NoSQL database [28]. All these systems use consistent hashing and self-managing replication groups. Scatter and ID-Replication target planetary-scale rather than enterprise data services and thus focus more on system behavior under high churn than speed at which clients are notified of configuration changes. Oracle NoSQL leverages the Oracle Berkeley DB (BDB) JE HA storage engine and maintains information about data partitions and replica groups across all clients. A key difference with our system is that whereas Oracle NoSQL piggybacks state updates in response to data operations, our clients have direct access to ring state in the CM, receive immediate notification after failures, and can request reconfiguration actions if they suspect a partial failure. We are aware of an HA monitor component that helps Oracle NoSQL clients locate RG masters after a failure, but were unable to find detailed information on how it operates.

Load balancing a dynamic population of clients in a coordination service such as Zookeeper [17] and Chubby [27] while maintaining data consistency and system availability has proved to be a challenging problem. Usually that kind of services use an external name-service such as *DNS* to balance client connections across servers. However as servers join or leave the service, already connected clients create an unbalanced load leading to sever overloading, performance degradation and high latency. A recent work by researchers at *Yahoo* extends Zookeeper's functionality offering dynamic reconfiguration capabilities and client load balancing [44]. They propose a probabilistic client-driven load balancing scheme where clients detect changes in the service and they decide by a policy whether to connect to an other server or not. In that approach an external DNS service is also required. In *ACaZoo* we experimentally demonstrate that a well balanced coordination service can result in 42% better notification response time compared to an unbalanced system and we introduce an algorithm with which ZooKeeper can internally load balance client requests. Unlike existing approaches our solution obviates the need for a DNS front-end, requires minor changes to the client implementation and does not introduce extra overhead to the clients since it is server-driven.

Chapter 4

System Design

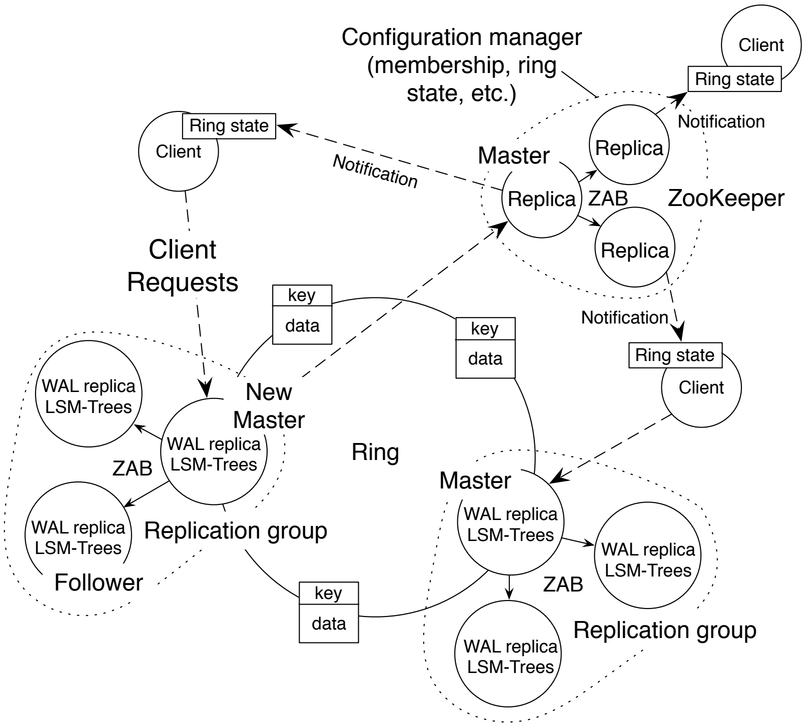

Configuration manager
(membership, ring
state, etc.)
Client
Ring state
Notification
Client
Ring state
Notification
Master
Replica
Replica
ZAB
ZooKeeper
Replica
Client
Requests
key
data
key
data
Notification
Ring state
Client
WAL replica
LSM-Trees
New
Master
WAL replica
LSM-Trees
ZAB
Ring
Master
WAL replica
LSM-Trees
WAL replica
LSM-Trees
Replication group
Follower
WAL replica
LSM-Trees
key
data
ZAB
WAL replica
LSM-Trees
Replication group
WAL replica
LSM-Trees

Figure 4.1: The ACaZoo architecture

4.1 ACaZoo architecture

The *ACaZoo* system architecture is depicted in Figure 4.1. The data model it supports, which derives from that of Apache Cassandra, HBase, and BigTable has the general structure shown in Figure 2.4. A unit of data (or *cell*) has the following coordinates: (*row key, column family name, column qualifier, version*). We use a consistent hashing mechanism [3] to map each row key to a replica group (RG) via the circular ring shown in Figure 4.1. Each RG is associated with a unique identifier that hashes to a point on the ring. Similarly each row key hashes to some point on the ring. A row key is assigned to the RG that maps nearest to it (clockwise) on the ring.

4.2 ACaZoo replication

The state being replicated is a write-ahead log (WAL) recording mutations to a set of LSM-Trees as shown in Figure 4.2. The replication algorithm being used is a primary-backed atomic broadcast found at the core of Zookeeper [45] (ZooKeeper Atomic Broadcast or ZAB [17]), a distributed coordination service. All accesses go through the master, ensuring order. In terms of durability, *ACaZoo* supports two modes of operation [1]: (1) writes considered durable when on disk and acknowledged by a quorum of replicas; or (2) writes considered durable when in memory and acknowledged by a quorum of replicas (while replicas periodically flush their memory buffers to disk). The second mode offers a strong level of consistency with a slightly weaker (but sufficient for practical purposes) notion of durability [46].

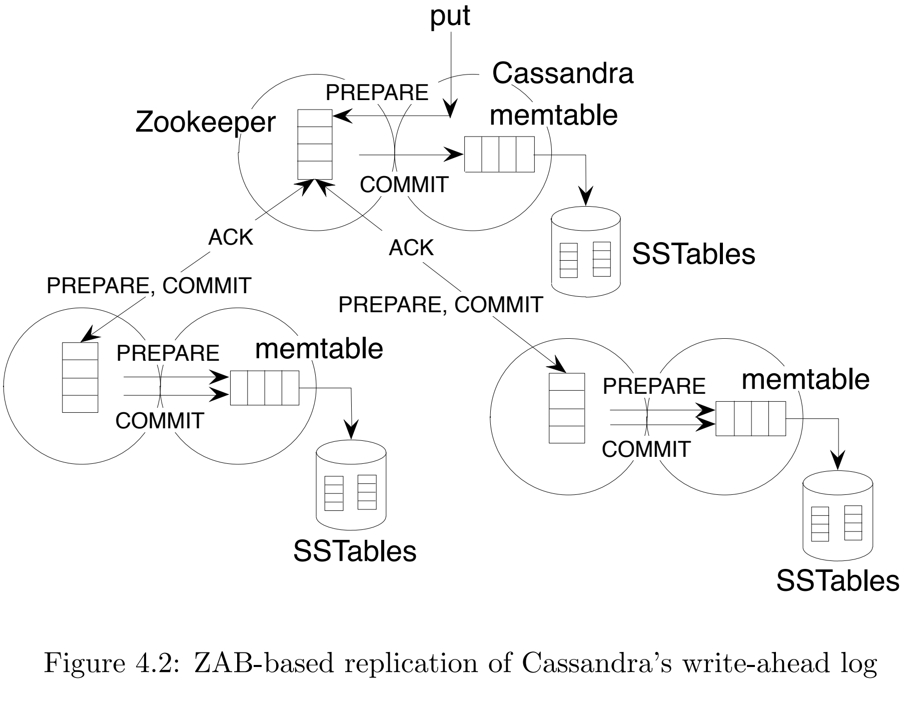

put
Cassandra
PREPARE
Zookeeper
memtable
COMMIT
ACK
ACK
SSTables
PREPARE, COMMIT
PREPARE, COMMIT
PREPARE
memtable
COMMIT
SSTables
PREPARE
memtable
COMMIT
SSTables

Figure 4.2: ZAB-based replication of Cassandra's write-ahead log

---

[1] Corresponding to the *batch* and *periodic* modes described in Section 5.1

4.3 RG leadership change

A key feature of *ACaZoo* is its ability to monitor periodic activities at the master node of any RG and to effect a change in leadership when a heavy compaction or other resource-intensive activity is expected to hamper performance of the master (and therefore of the entire RG). The current master of an RG is responsible for the monitoring and management of this process. Section 5.1 provides more information on the policies and implementation details of RG leader switch.

4.4 Load balancing client notifications

The ring state is stored on a Configuration Manager (or CM). The CM combines a *partitioner* (a module that chooses unique identifiers for new RGs on the ring) with a distributed coordination service (we use ZooKeeper/ZAB again here). Partitioner subdivides the ring to a fixed number of key ranges and assigns RG identifiers to key-range boundaries. This method has previously been shown to exhibit advantages over alternative approaches [3]. Each key-range in our system corresponds to a different RG. The CM contains information about all RGs, such as addresses and status of nodes (master or follower), and corresponding identifiers. Any change in the status of RGs (new RG inserted in the ring or existing RG changes master) is reported to the CM via RPC. The clients can query the CM to identify the current master of an RG (pull changes) or ask to be notified of any changes (push changes).

The CM re-balancing methodology we introduce identifies imbalances in notification load between ZooKeeper servers and decides which sessions should be exchanged between servers with the goal of assigning the same number of watch notifications to all servers. More specifically, every quorum maintains an internal znode in which each server owns a child node storing the up-to-date state. Periodically every quorum peer compares the number of current open watch sessions with the watch session of the rest of the peers by accessing the internal znode-state. To avoid frequent and unnecessary client redirections we set a common threshold among quorum that defines the upper limit of watches a server can handle without the need for re-balance. Periodically a server checks if re-balance is needed in a quorum of N nodes, by evaluating the expression below:

$$|myWatchers - watcherOfSrv[x]| > quorumThreshold, \forall x \in N$$

If the above expression is true, the server chooses the client who was most recently connected as well as the server with the less open sessions among the quorum's servers. Afterwards the server sends a notification, including the new server's address, to the client and notify him that server switch is needed. As soon as the client receives the redirection message, he tries to establish a new session with the target server without closing the previous session. This way, we ensure that not even a single watch notification will get lost during the new session establishing procedure, since the client closes the original session only by the time he has finished the transition and has successfully registered a watch at the new server. At this point we have to mention that it is crucial for the ZooKeeper servers to update their internal state not only after every new session they establish but also as soon as they detect a closed or expired session. Thus, every node in the quorum will be up-to-date and can take the right decision when it needs to redirect a client.

Chapter 5

Implementation

The *ACaZoo* implementation uses the Apache Cassandra NoSQL system as a baseline. It preserves the Thrift-based Cassandra client API for compatibility with existing Cassandra applications. ACaZoo extends Cassandra in several important ways: First, it replaces its eventually-consistent replication mechanism with a strongly consistent implementation (based on ZAB [17]) that replicates a write-ahead log used for recovery of the Cassandra server state. Section 5.1 describes the internals of the *ACaZoo* replication and storage back-end and Section 5.4 the internals of ZooKeeper and its implementation of ZAB, as well as optimizations we have applied to it. Second, *ACaZoo* uses client-coordinated I/O in conjunction with a ZooKeeper-based configuration management service. Section 5.2 describes implementation details and a method to load balance bursts of notifications across ZooKeeper servers. A general overview of *ACaZoo* components and interactions is shown in Figure 5.1.

5.1 ACaZoo

The ACaZoo storage backend (with the ACaZoo modifications over the base Cassandra implementation highlighted in bold) is depicted in Figure 5.2. More information on the original Cassandra implementation can be found in Lakshman *et al.* [8].

Replication

An *ACaZoo* storage server comprises two integrated software modules: A Cassandra storage server together with a Zookeeper server (Figure 5.2). The two modules interface at three crossing points: Cassandra invokes ZooKeeper's PrepRequestProcessor stage (described in Section 5.4), ZooKeeper calls Cassandra's *WAL insert* interface, and ZooKeeper notifies Cassandra of a leader election outcome, all internal server-side interfaces. Both modules are run as a single process (daemon) that starts and manages the ACaZoo storage server.

An *ACaZoo* replication group (RG) comprises a (configurable) number of storage nodes. At startup of each RG, elections are being held by ZooKeeper to elect

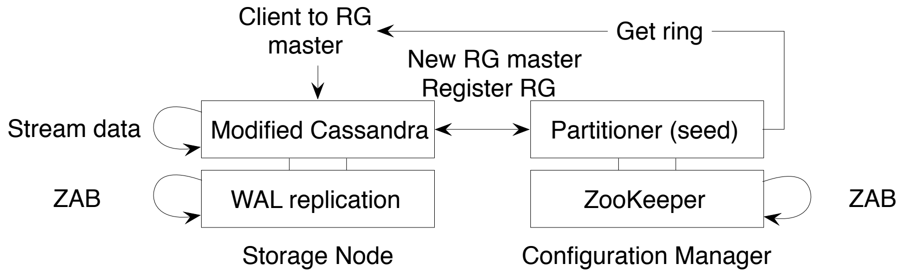


Figure 5.1: System components and their interactions

the master server. The Cassandra code at the master is notified of the result so that it assumes the handling of read/write requests.

Writes in *ACaZoo* (and Cassandra) must first be recorded as mutations to the WAL, and then written to an ordered per-Column Family memory structure called a Memtable. When Memtable is full, it is flushed to disk as an SSTable. Mutations represent changes to one or more tables (all belonging to the same keyspace) and all changes refer to the same partition key. Since these mutations represent only the changes made (they are *deltas* rather than full representations of a cell) they are an efficient way to transfer them over a communication link.

The master of a RG is responsible for invoking the ZAB agreement protocol on each write. Internally, ZooKeeper takes the contents of a put operation (the mutation) and replicates it as the contents of a persistent *znode* treated by ZooKeeper as SEQUENTIAL (associated with a global sequence number appended to its name). As soon as the proposal is committed, the mutation is inserted into the ACaZoo WAL on any node that learns of the commitment and then applied to the local LSM-Trees. The *znode* and associated commit log entry can be erased as soon as the ACaZoo WAL acknowledges it. Minimizing ZooKeeper state has the benefit that periodic snapshot operations are inexpensive. ZooKeeper currently requires manual deletion of old snapshots and commit logs by an operator; this is something we intend to automate in ACaZoo.

Gets are handled by the master, first looking up its cache and then (in case of a miss) its local LSM-Tree. Just as in Cassandra, *ACaZoo* uses SSTable indexes and Bloom Filters to reduce I/O overhead in case of a read miss. In a RG reconfiguration, a new master is expected to have a cold cache and therefore clients will experience the related warm-up phase. Note that this issue is intrinsic to any replication system where a single master handles all read activity.

During early testing of our prototype we noticed that applying mutations in each replica's LSM-Tree was not sufficient to make the new state visible to clients. The reason was that metadata for these mutations were not being updated in the process. We solved the problem by forcing a reload of the local database schema on each put so that the schema gets rebuilt according to the current state. This

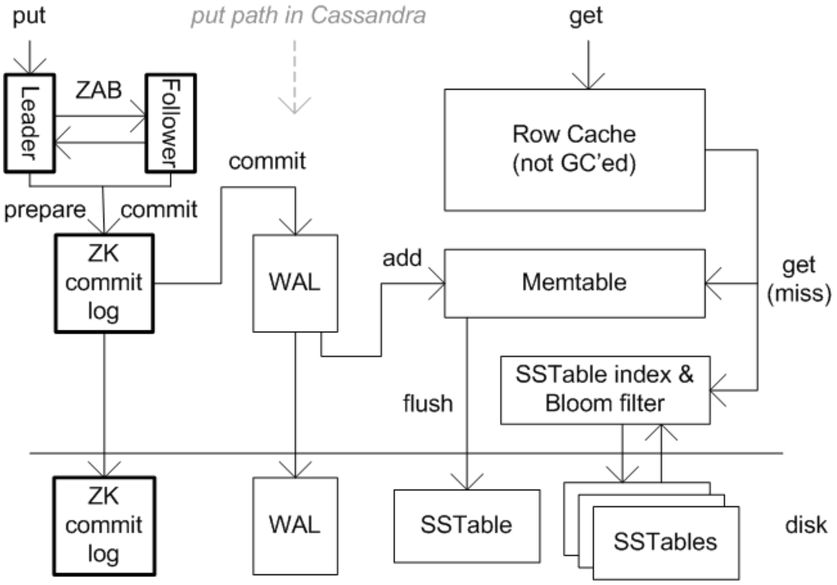

put
put path in Cassandra
get
Leader
ZAB
Follower
commit
prepare
commit
ZK commit log
WAL
add
Row Cache (not GC'ed)
Memtable
get (miss)
SSTable index & Bloom filter
flush
ZK commit log
WAL
SSTable
SSTables
disk

Figure 5.2: ACaZoo storage server

was a lightweight fix that does not impact performance.

In terms of durability, ACaZoo (inheriting from Cassandra) can be configured for either *periodic* or *batch* writes to its commit WAL. Periodic (the default mode) initiates write I/Os as soon as they appear in the execution queue and acknowledges them immediately with clients. Another thread periodically (as specified in `commitlog_sync_period_in_ms`) enqueues a sync-to-disk operation on the execution queue. The batch mode offers stricter durability by grouping multiple mutations over a time window (defined by `commitlog_sync_batch_window_in_ms`) and executes them in a batch. After each batch, it performs a sync of the commit WAL and then acknowledges the writes with clients.

We have two optimizations in mind to further improve the I/O performance of our prototype. One optimization has to do with avoiding a redundant commit step in ZK and the Cassandra WAL. Another has to do with further integrating the prepare disk write with the WAL (so that the data is not written twice to disk). Based on our performance results we believe that these optimizations are not critical and defer their implementation to a future edition of our prototype.

Masking the impact of periodic compactions

In *ACaZoo*, LSM-Tree compaction activities that are expected to impact a RG master (and thus the entire RG) are detected and acted upon: prior to compaction, a RG master triggers a leader election and freezes the compaction process until demoted to a follower. *ACaZoo* monitors periodic storage-node activities (primarily SSTable compactions but other activities as well, such as Memtable flushes and Java garbage collections). Our implementation disables the AutoCompaction feature of base Cassandra. It maintains an estimate of the amount of compaction work that accumulates over time and can hold or trigger compaction at specific time points.

We have modified the ZooKeeper leader election process to implement suitable election policies. We currently support the election of a random node excluding the current master (RANDOM policy), or the election of the next node in some sequence (round robin, RR policy). When the leader is nearing the point of having to perform a compaction, it sends a new leader-election event (a NEW_LEADER proposal in ZooKeeper terminology, see Section 5.4) towards the followers. A new leader is elected as soon as a quorum of servers commit to following him. Normally during leader election, ZooKeeper nodes broadcast their votes containing their current epoch, last transaction seen, and preferred master. By default the preferred master in that vote is themselves; nodes that are up to date have more chances to win. In *ACaZoo*, the node triggering the leader election is voting for someone else according to the policy (RANDOM or RR). *ACaZoo* followers that have a low anticipated compaction load respond positively to the leader election according to the policy.

During the election, which lasts typically between 200-500 ms the outgoing leader keeps accepting requests. When a new leader is elected, the previous leader returns a CUSTOM EXCEPTION with the identity of the new leader in RPCs. Thus the identity of the new leader is rapidly propagated to the requesting clients. The alternative path of clients learning of leadership changes through watch event notifications via ZooKeeper is still possible (its performance is studied in Section 6.6).

A caveat here is that frequent leadership changes can hurt performance. To avoid this situation, we estimate the compaction load of a server at all times (taking into account the amount of data that need to be compacted) and decide to trigger a leader election only when that load exceeds a configurable threshold (and thus expected to be a hard hit on performance).

5.2 Client-coordinated I/O and configuration management

ACaZoo clients are allowed to maintain ring state and thus be able to route operations appropriately rather than through a (possibly random) server. They obtain that state by connecting to the Configuration Manager (CM), the integration of a ZooKeeper cell and a Cassandra partitioner that jointly maintain RG identities and tokens, master and follower IPs, and the key ranges on the ring. We decided to use actual IP addresses rather than elastic ones due to the long reassignment delays we observed with the latter on certain Cloud environments. Each RG stores its identifier and token in a special Zookeeper *znode* directory so that a newly elected RG master can retrieve it and identify itself to the CM.

In more detail, CM creates a ZooKeeper directory named `system_state` and a *znode* in it for each token range in the system. A new master, aware of the token range it represents modifies the state of the corresponding *znode* to store information about its RG. We additionally store in `system_state` information about the RG followers and other system state to empower clients with the ability to take action as soon as possible when a failure occurs. The CM exports two RPC APIs to storage nodes: *register/deregister RG*, *new master for RG*; and a *get ring info* API to both storage nodes and clients.

ACaZoo clients have the option to either use the exposed CM API or request notifications of any changes by setting *watches*[1] on the ZooKeeper `system_state` *znode*. The CM is responsible for maintaining this *znode* up to date by monitoring the ring state using a JMX API. The CM thus learns of a new RG joining the ring or a new master election in an existing RG and triggers watch notifications.

5.3 Dynamic load balancing requests

During testing of our early prototype we noticed that ZooKeeper does not dynamically redistribute watch requests across its servers, relying instead on an external directory service (such as a DNS balancer) to achieve static load balancing at session establishment. Although there is already a reconfiguration solution proposed by Shraer et al. in [44], client departures can result in an imbalanced system. Our evaluation shows that this leads to higher notification latencies than feasible with a well balanced system. We thus designed an internal rebalancing methodology that dynamically redirects ZooKeeper client session to different servers based on dynamic load information.

Our re-balancing methodology identifies imbalances in notification load between ZooKeeper servers and decides which sessions should be moved between servers (from the *source* to a *target* server) to spread out the work evenly (with the goal of assiging the same number of watch notifications to all servers). For each session movement, the source server piggybacks a CHANGE_SERVER event (to target server) command to the next session heartbeat from/to the client. As soon as the client receives the event, it tries to establish a new session with the target server without closing the previous session. When the new session is established, the client closes the original session. The method ensures that a watch notification during the transition is received exactly once by the client: The source server first tries to deliver a watch notification directly to the client; if successful (that is, before the client closes the session), it takes no further action. If it fails (the client has closed the session), it then relays the notification to the target server who will deliver it to the client. A handover protocol between servers ensures orderly transfer of control to the target server.

---

[1] A *watch* is a request by a client to receive a notification by a ZooKeeper server should there be any modification to the referenced *znode*.

5.4 ZooKeeper data path and optimizations

This section provides insight to the internals of the ZooKeeper server system architecture. While important for understanding the operation of the ZAB protocol, its efficiency, and the integration of ZooKeeper into ACaZoo (Section 5.1), this is not prerequisite reading and the reader can skip to Section 6. ZooKeeper has a staged event-driven architecture (Figure 5.3). Each QP is structured as a sequence of Request Processors that operate in a pipelined fashion. Each RP is associated with a thread that performs operations on its input passing over the result to the next RP via a shared FIFO queue. The requests are passed by reference through the queues.

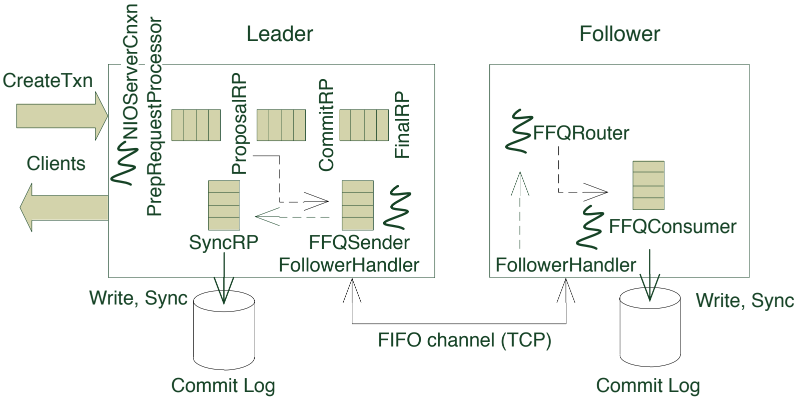

Leader
Follower
CreateTxn
Clients
NIOServerCnxn
PrepRequestProcessor
ProposalRP
CommitRP
FinalRP
SyncRP
FFQSender
FollowerHandler
FFQRouter
FFQConsumer
FollowerHandler
Write, Sync
Write, Sync
FIFO channel (TCP)
Commit Log
Commit Log

Figure 5.3: ZooKeeper data path

The lifecycle of a proposal comprises the following steps: The client sends a request to the leader by invoking the CreateTxn API. A thread (NIOServerCnxn) allocates a new buffer in which it copies the proposal's data payload. The leader may decide to throttle clients if it is running low on the number of available pre-allocated buffers. The leader queues the request into the FIFO queue of the PrepRequestProcessor stage. The PrepareRequestProcessor encapsulates the client request into a Quorum Packet and queues it into the queue of ProposalRP which run in the same context. ProposalRP will broadcast the quorum packet using another thread (FFQSender), which handles message transmission to all peers. The proposal is then passed to SyncRP, which appends it to the Commit Log. SyncRP periodically flushes written proposals to the disk using group commits. SyncRP passes the proposal on to CommitRP, which is responsible for counting ACK messages sent by QPs for this proposal. QPs may ACK a proposal only after they ensure that it has been successfully flushed to disk. When the leader receives a majority of ACKs it sends a COMMIT message to all QPs over the chain and the proposal is applied on the receiving of the COMMIT message. When a proposal is committed, the FinalRP stage sends a reply back to the client.

Followers have a simple structure in which a FFQRouter thread performs equivalent tasks to the leader's FFQSender. FFQRouter is spawned when the follower receives a NEW_LEADER message and is responsible for listening for new incoming TCP connections. It passes its output to the FFQConsumer thread, which decides if this message should be consumed or not based on its FollowerEpoch field. If the message is from a previous epoch (i.e., stale), it rejects it, otherwise it consumes it. If the message is a PROPOSAL it appends it to the Commit Log and then sends an ACK message to the leader. If the message belongs to any other category it appends it to the commit log and performs the appropriate proposal-specific action to it.

Optimizations

Writes to the ZK commit log use a group-commit mechanism. To achieve high I/O throughput the system needs to ensure efficient operation all the way to the disks. In a heavily write-intensive setup the filesystem buffer-cache should be continuously writing to disks to avoid stalling for too long at the periodic sync operation. The standard operating system setup (in Linux and other general-purpose operating systems) is to delay (defer) writes. We had to change the standard behaviour by tuning the thread responsible for destaging data from the buffer cache to disk to be invoked whenever there is anything in the buffer cache to be written. Note that this optimization requires platform-specific knowledge and is thus testimony to the fact that despite improved support in Java [47] it is not always possible to achieve fully platform-independent systems software in Java alone.

Chapter 6

Evaluation

Our evaluation platform is a private Cloud that builds upon OpenStack [1], an open-source cloud platform for private and public clouds. For the experiments we used virtual machines (VMs) having 2 CPUs, 2GB memory, and a 20GB remotely-mounted disk. The Apache Cassandra version used is 2.0.l, Apache Zookeeper is 3.4.5, and the Oracle NoSQL database is 2.1.54.

We chose the Oracle NoSQL commercial database as one point of comparison to ACaZoo since it closely relates to it in several aspects (key-value store with similar API and data model, sharding over replica groups, primary-backup replication, client routing of I/O requests, Java implementation) but differs from it in the use of a B+-tree storage organization vs. ACaZoo's LSM-Trees.

Our load generator is the Yahoo Client Serving Benchmark (YCSB) [48] version 0.1.4, which can be configured to produce a specific access pattern, I/O size, and read/write ratio. YCSB performs 1KB accesses (it creates and accesses a single table with one column family comprising ten columns of 100-byte cells) with configurable read/write ratio using Zipf (featuring locality) or uniformly-random probability distributions.

We further evaluate our system using a real scalable e-mail service application called *CassMail*. CassMail is a working implementation of the SMTP and POP3 protocols using Cassandra as a storage engine. Figure 6.1 displays the schema in which CassMail through Cassandra stores user and mailbox information. There are two tables, *Mailboxes* and *Users*, within a keyspace called *Mail*. Each row in *Users* is used to validate a user while each row in *Mailboxes* table contains blocks that hold the actual e-mail messages in a userś mailbox. The key for that table is a username concatenated with a mailbox name. The key for each block is a time-based universally unique identifier (UUID) stamped by the SMTP daemon when the message arrives and is stored. The value of a block is the e-mail itself. Since *ACaZoo* and Cassandra share the same data model we extended CassMail to support ACaZoo storage system and we compare the results.

To run *CassMail* we used a number of nodes with the same hardware specifi-

---

[1] OpenStack, "http://www.openstack.org," April 2012

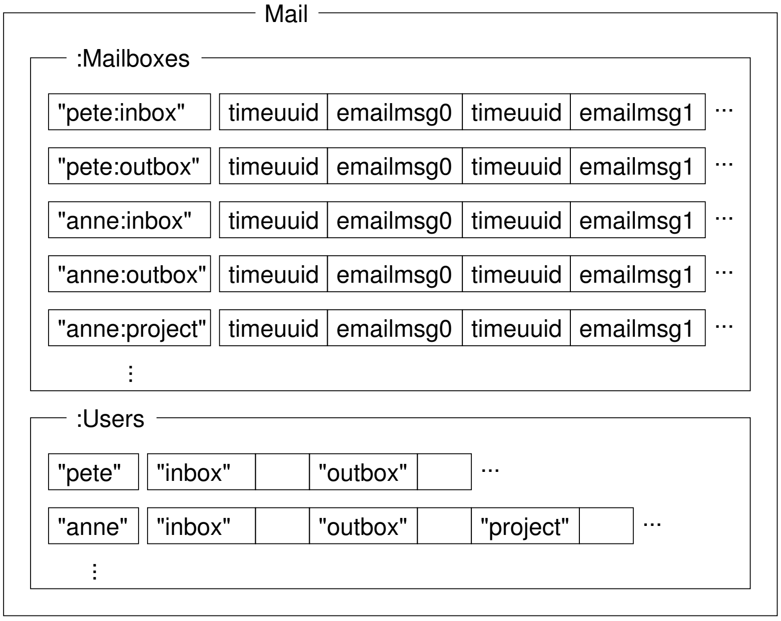

Mail
:Mailboxes
"pete:inbox" timeuuid emailmsg0 timeuuid emailmsg1 ···
"pete:outbox" timeuuid emailmsg0 timeuuid emailmsg1 ···
"anne:inbox" timeuuid emailmsg0 timeuuid emailmsg1 ···
"anne:outbox" timeuuid emailmsg0 timeuuid emailmsg1 ···
"anne:project" timeuuid emailmsg0 timeuuid emailmsg1 ···
⋮
:Users
"pete" "inbox" "outbox" ···
"anne" "inbox" "outbox" "project" ···
⋮

Figure 6.1: Cassandra schema designed for CassMail [6]

cation as described above. Each node maps to a specific position on the ring as shown in Figure 6.2 though a hash function. Each row maps to a position on the ring by hashing its key using the same hash function. In the example depicted in 6.2 *anne* connects through her mail-submission agent to the SMTP server on node1 to send an email to *pete*. When *pete* wants to check his messages, he accesses for example node n using his mail-retrieval agent. The mail agent first fetches the number of e-mails which is equal to the number of columns in *pete:inbox* and then asks for the message headers.

The benchmark used in this study was Postal [2], a widely used benchmark for SMTP servers for measuring write throughput. Postal operates by repeatedly and randomly selecting and e-mailing a user (USER@example.com) from a population of users as shown in the system architecture 6.2. We created a realistic population of users by using the usernames from our department mail server (about 1K users). The Postal client is a multi-threaded process that connects to a specific SMTP server.

6.1 Performance impact of LSM-tree compactions

We first quantify the performance impact of LSM-tree compactions on an ACaZoo replication group of three storage nodes with a fixed master (no RG leader changes). Figure 6.3(a) depicts the throughput of a write-intensive workload

---

[2] http://doc.coker.com.au/projects/postal/

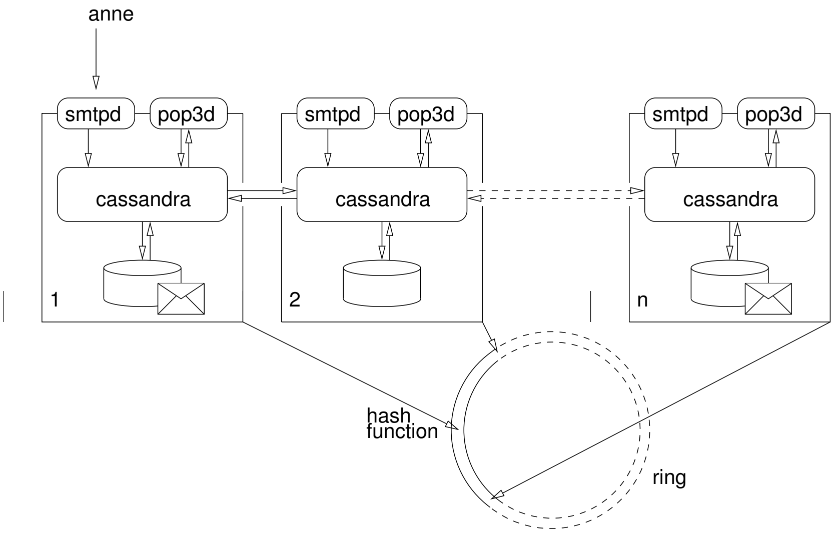
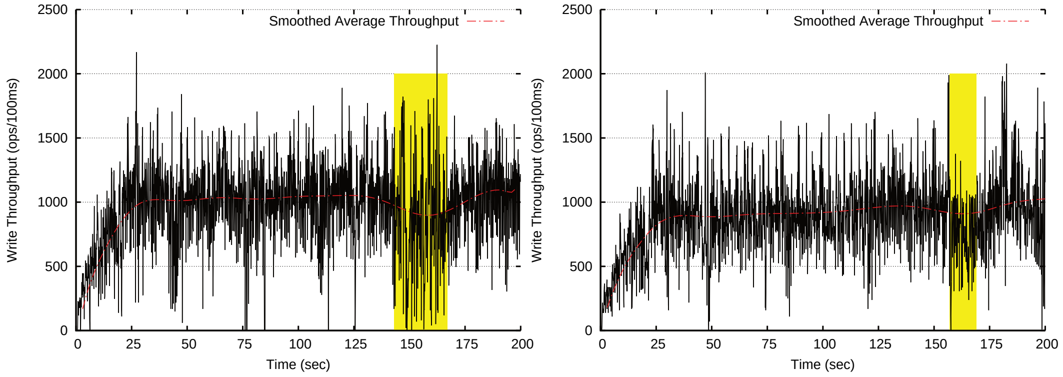


Figure 6.2: CassMail system design [6]

(YCSB with 64 threads performing 100% writes) over time. We observe that at 45″, 75″, and 113″ performance drops briefly but drastically due to memtable flushes taking place. From 143″-165″ a compaction task is seen to have a significant and enduring impact on system performance. This compaction task involves 4 SSTables and takes 22.34 sec to complete for a total of 310MB of data compacted (an effective throughput of 8.5 MB/s). Besides Memtable flushes and compaction activities, Java garbage collection is seen to have infrequent but measurable impact on performance.

Insert operations are indeed resource intensive due to message deserialisation, Memtable flushes, compactions, and intensive memory use. Through `iostat` monitoring we observed that our VMs are nearly always CPU-bound, turning I/O-bound when the amount data processed grows significantly. While we believe that this picture can be somewhat improved by increasing the allocation of resources to servers (e.g., assigning an additional disk spindle dedicated to the commit WAL as well as additional CPU cycles) this increases system cost and may not always be an option in large-scale deployments.

We next evaluate the performance improvement from RG leader changes. Figure 6.3(b) depicts YCSB throughput with 64 threads and a 100%-write workload under the RANDOM leader change policy. We observe the impact of Memtable flush events at 48″, 85″, 121″, 199″, and 244″. There is also a new leader election at 157″ just before the leader starts a compaction task. At that point, the client experiences a short (100ms) interval of unavailability (throughput drops to 0 ops/sec) and then continuing with the new leader. This can be contrasted to the long (about 23 sec) interval of performance degradation due to compaction observed in Figure 6.3(a). In Section 6.4 we show that when the master is compacting, the probability that a majority of RG nodes simultaneously compacting is low (and



(a) ACaZoo without RG leader changes (b) ACaZoo with leader changes (RANDOM policy)

Figure 6.3:

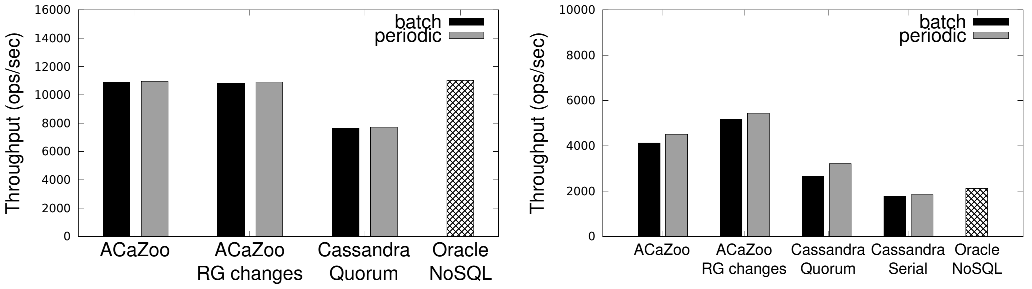
decreasing with RG size). Therefore a RG leader change is expected with high likelihood to produce a configuration that can make progress.

6.2 Performance of a 3 node replication group

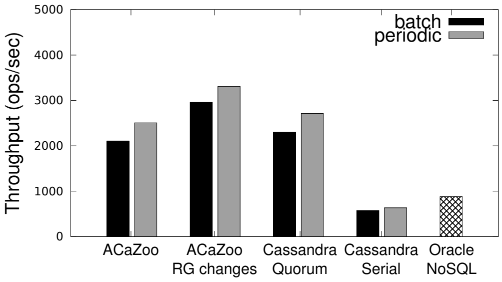
We next compare the performance of a single ACaZoo replication group of three storage nodes to an equivalent setup of Oracle NoSQL, and two Cassandra setups of three storage nodes in a ring with replication factor 3: a setup with *quorum consistency* reading/writing 2 out of 3 replicas (termed Cassandra Quorum) and another similar setup performing linearizable writes [5] (termed Cassandra Serial). Cassandra Quorum -a relaxed consistency system- is configured conservatively so as to approximate the semantics of the other three systems. We evaluate all systems under a YCSB workload of 256 concurrent threads with three different operation mixes (100/0, 50/50, 0/100 reads/writes). We test ACaZoo both with and without RG leader changes using the RANDOM policy, to evaluate the performance improvement from enabling this feature.

ACaZoo performs reads from the master replica only, just as Oracle NoSQL does (Oracle calls this *absolute consistency*). As discussed in Section 5.1, ACaZoo and Cassandra can be configured for either *periodic* (relaxed durability) or *batch* (strict durability) writes to their commit WAL. Oracle NoSQL is configured to perform writes in WRITE_NOSYNC mode, i.e., it initiates them as soon as they arrive at replicas but syncs them to disk only when a buffer of configurable size fills up (similar to the *batch* mode of ACaZoo and Cassandra).

In Figure 6.4(a) we observe that for 100% reads ACaZoo performs on par with Oracle NoSQL, while Cassandra trails due to the requirement to read from two rather than one replica. We omit Cassandra Serial from Figure 6.4(a) since its performance is identical to Cassandra Quorum (the two systems share their read path). ACaZoo with RG changes does not lead to a performance improvement in this case due to the absence of write (and therefore significant compaction)



(a) YCSB throughput: 100% reads (b) YCSB throughput: 50% reads, 50% writes



(c) YCSB throughput: 100% writes

Figure 6.4: Openstack - 3 nodes Replication Group

activity.

As the share of writes increases into the mix, performance drops for all systems. We observe in Figures 6.4(b) and 6.4(c) that ACaZoo in batch mode outperforms Oracle NoSQL (for both the 50%/50% and 100%-write mixes). We attribute the difference to the better pipelining of I/O operations in ACaZoo compared to Oracle NoSQL (we have empirically determined that BerkeleyDB JE –the underlying Oracle NoSQL storage engine– allows only a single outstanding batch write transfer between replicas at a time). For 50% reads, 50% writes (Figure 6.4(b)) ACaZoo with RG leader changes outperforms standard ACaZoo by 25% and 20% for batch and periodic operations. For 100% writes (Figure 6.4(c)) ACaZoo with RG leader changes outperforms standard ACaZoo by 40% and 33% for batch and periodic operations respectively. Results with the RR policy are nearly identical to those with the RANDOM policy leading us to conclude that both are equally effective in our workloads.

6.3 Performance of a 5 node replication group

Next we evaluate the performance of our system in a bigger RG to measure the effectiveness of RG leader change approach under different RGs. Similar to the experiment 6.2 we compare the performance of a single ACaZoo replication group,

	Count (#)	Longest (sec)	Average (sec)	Total (sec)
Compaction (RA)	11	78.44	17.96	197.64
Memtable flush (RA)	53	-	-	-
Garbage Collection (RA)	197	0.91	0.148	29.33
Compaction (RR)	12	72.65	15.94	191.39
Memtable flush (RR)	52	-	-	-
Garbage Collection (RR)	192	0.85	0.147	27.84

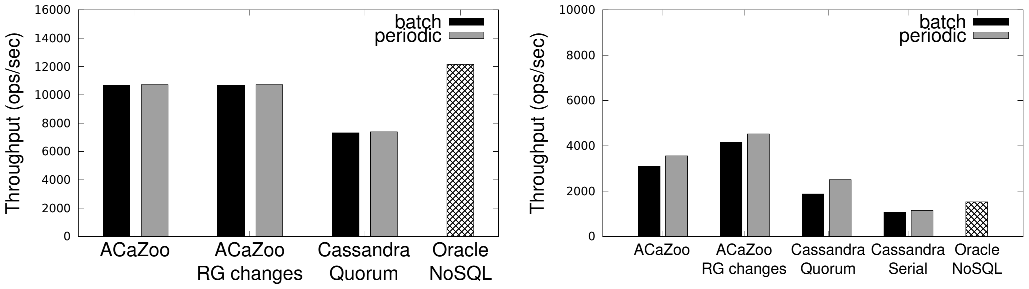
Table 6.1: Tasks during a 100%-write YCSB workload

of five nodes this time, to an equivalent setup of Oracle NoSQL, and two Cassandra setups of five storage nodes in a ring with replication factor 5: a setup with *quorum consistency* reading/writing 3 out of 5 replicas (Cassandra Quorum) and another similar setup performing linearizable writes (Cassandra Serial). We use again YCSB framework as benchmark YCSB with three different operation mixes (100/0, 50/50, 0/100 reads/writes) and 256 concurrent threads.

In Figure 6.5(a) we observe that for 100% reads Oracle NoSQL performs 12% better than ACaZoo, while Cassandra trails due to the requirement to read from three rather than one replica. While this is a rather small improvement we attribute this to the fundamental difference between B+-Trees and LSM-Trees where for random reads in a B+-Tree the complexity is *O(log N)* while in a SSTable index like ACaZoo and Cassandra it is actually *O(#sstables * log N)*. Meaning that without frequent compactions (SSTable merging), the number of SSTables is proportional to N. Finally ACaZoo with RG changes again does not lead to any performance improvement due to the absence of writes.

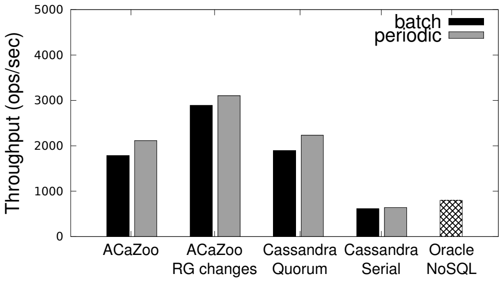
In Figures 6.5(b) and 6.5(c) we observe that ACaZoo in batch mode outperforms Oracle NoSQL and Cassandra (for both the 50%/50% and 100%-write mixes). For 50% reads, 50% writes (Figure 6.5(b)) ACaZoo with RG leader changes outperforms standard ACaZoo by 38% and 28% for batch and periodic operations. For 100% writes (Figure 6.5(c)) ACaZoo with RG leader changes outperforms standard ACaZoo by 60% and 47% for batch and periodic operations respectively. Results clearly show that the bigger the RG is the more effective the ACaZoo RG leader change technique is especially in write intensive workloads. We further discuss the effectiveness of this technique over different RGs in section 6.4.

To get a deeper understanding of the intensity of compactions and other periodic tasks in ACaZoo we logged these activities on the (initial) master node during a 20-minute 100%-write workload (the results are shown in Table 6.1). Although the master will change during the course of the run, the events are representative of the activity taking place at each of the nodes in a RG. The recorded events in a typical run with the RANDOM policy (depicted as RA) included 11 compactions, 53 Memtable flushes, and 197 Java garbage collections. It is interesting to note that garbage collection events are brief; compactions however are long and very



(a) YCSB throughput: 100% reads

(b) YCSB throughput: 50% reads, 50% writes



(c) YCSB throughput: 100% writes

Figure 6.5: Openstack - 5 nodes Replication Group

intensive. The longest compaction lasts for more than a minute and merges more than 700MB of data. Results with the ROUND ROBIN policy (depicted as RR in Table 6.1) are similar; a slight difference is that RA spends more time on garbage collection and compaction in this run (the two are related: a larger compaction (700MB in RA lasting 78″ vs. 600MB lasting 72″ in RR) leads to a longer garbage collection in RA).

6.4 Timing of compactions across replicas

In this experiment we observe compaction events on all replicas of an ACaZoo RG in configurations of 3, 5, and 7 nodes. Our goal is to determine the degree of overlap of compaction events over a long (90 minute) workload. We use YCSB with 256 threads producing a 50% read, 50% write mix for 60 minutes, then switch to a 100% write mix for 30 minutes in a more write-intensive phase. The ACaZoo configuration we used has the RG leader-change feature disabled. Our goal in this experiment is to determine the probability P that when compaction happens anywhere in the system only a minority of nodes in the RG are simultaneously involved in it and therefore there is a majority that can make progress. Because of significant non-determinism in the system we expect that in practice P (expressed in formula (1)) would be non-zero.

$$
\begin{aligned}
P(\text{ Any minority in RG compacting }) = 1 - \\
P(\text{ No compactions in RG }) - \\
P(\text{ Any majority in RG compacting })
\end{aligned}
\tag{6.1}
$$

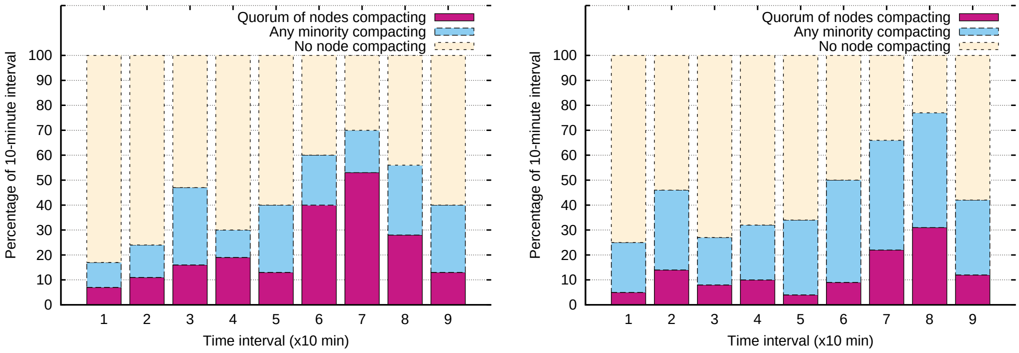
Figures 6.6(a)–6.6(c) depict the time breakdown per 10-minute interval between the following three states: No compaction anywhere in the RG; some (any) minority of RG nodes are compacting simultaneously; a quorum of RG nodes are compacting simultaneously. The fraction of time spent in the second of those states is a measure of P. Going from 3 to 7 nodes in an RG, P ranges (on average) from 21% (for 3 nodes) to 32% (5 nodes) to 44.5% (7 nodes), indicating that the RG leader-change technique is expected to be increasingly effective with larger RG sizes. The average probability that a quorum of RG nodes compacts simultaneously (the regime where the RG leader-change technique does not help) diminishes from 23% (3 nodes) to 13% (5 nodes) to below 12% (7 nodes). We note that if a higher degree of non-determinism is desirable, it could be achieved by using different configurations (e.g., min/max compaction thresholds) for each replica.

6.5 Timing of garbage collections across replicas

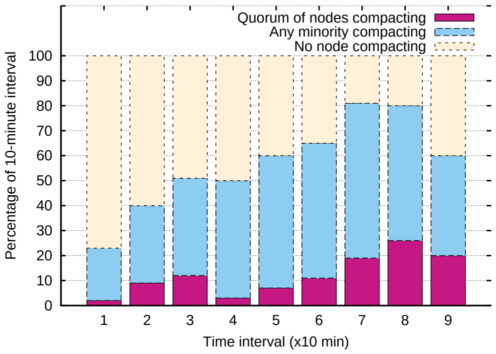
In this experiment we observe garbage collection events on all replicas of an ACaZoo RG in configurations of 3, 5, and 7 nodes. Running the same workload as described in Section 6.4 our goal is to determine the degree of overlap, this time of garbage collection events, over a long -90 minute- run. Figures 6.7(a)–6.7(c) depict the time breakdown per 10-minute interval between the following three states: No garbage collection anywhere in the RG; any minority of RG nodes garbage collecting simultaneously; a quorum of RG nodes garbage collecting simultaneously. We observe that there is no majority of nodes garbage collecting at the same time regardless RG size (3, 5 or 7 nodes). The reason is that unlike compactions which last several seconds, garbage collections are quite brief and they last a few ms. Going from 3 to 7 nodes though affects the minority compaction rates (on average) which rage from 6% (for 3 nodes) to 5% (for 5 nodes) to 8% (for 7 nodes). These periodic events can decrease the performance of our system but as we experimentally demonstrate they are non deterministic and less lengthy than compactions. This is the reason in *ACaZoo* we implemented the RG leader change technique to take into account compactions instead of other periodic events. Of course we could extend our approach to take into account more than one periodic events.

6.6 Availability of RG under leader failure

In this section we compare the availability of a replication group when the leader fails, comparing ACaZoo to Oracle NoSQL database under a YCSB read-only



(a) AcaZoo RG of 3 nodes (b) ACaZoo RG of 5 nodes



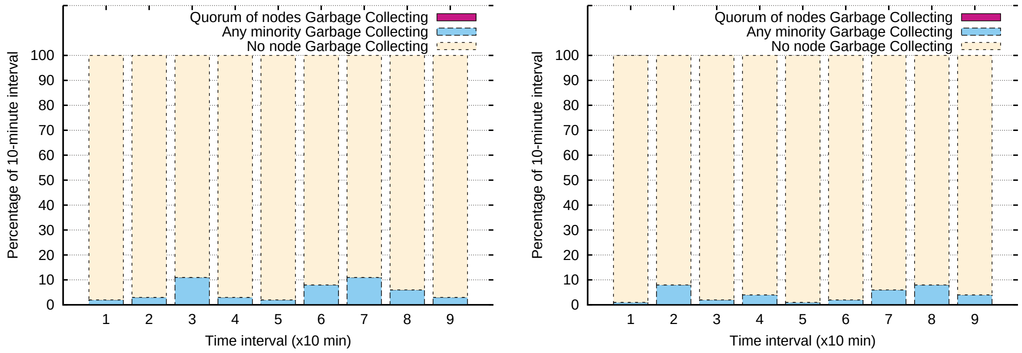
(c) ACaZoo RG of 7 nodes

Figure 6.6: ACaZoo compaction behaviour under different replication groups (RG) (a) 3 nodes RG (b) 5 nodes RG (c) 7 nodes RG

workload produced by 64 threads. Figure 6.8 shows an outage of about 3.24 sec from the time that the leader of the RG crashes until service resumes at the YCSB client. This interval breaks down to the following segments: (a) 1.19 sec between the time the leader crashes until the client notices; (b) 2 sec until the client establishes a connection with the new leader and restores service. Interval (a) further breaks down into: (1) 220 ms for the RG to reconfigure (elect a new leader); (2) 970 ms to propagate the new-leader information (e.g., its IP address) to the client through the CM.

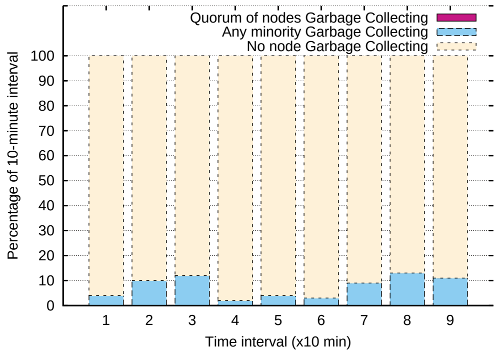
We observe that performance initially ramps up from about 1200 ops/100ms to about 1900 ops/100ms. Since this is a read-intensive experiment, client performance depends on the cache warm up phase in the storage node. After the failover, we observe the same phenomenon since the new leader also starts from a cold cache.

Figure 6.9 shows YCSB throughput observed in the same scenario under Oracle NoSQL. The client-observed outage in this case is about 3.5 sec. We observe that the Oracle NoSQL system starts from about half the throughput of ACaZoo (600 vs. 1100 ops/100msec) at the beginning of the run and also after a failover. We



(a) 3 nodes RG

(b) 5 nodes RG



(c) 7 nodes RG

Figure 6.7: ACaZoo garbage collection behaviour under different replication groups (RG) (a) 3 nodes RG (b) 5 nodes RG (c) 7 nodes RG

additionally observe a more noisy behaviour in Oracle NoSQL's throughput curve as opposed to ACaZoo. All else being equal, we attribute this difference to the efficiency of the LSM-tree indexing scheme vs. the B+-tree indexing scheme of Oracle NoSQL.

6.7 Load balancing notifications

In this experiment we evaluate the effect of load-balanced ZooKeeper notifications to end-user latency. We setup a client population ranging from 1 to 10,000 threads that (a) registers watch requests with a 1-node ZooKeeper cell; (b) registers watch requests with a 3-node ZooKeeper cell; (c) registers watch requests in a load-balanced (LB) manner with all server in the 3-node ZK cell. Client threads operate in a closed loop in which they establish a watch with the root *znode*, wait until receiving a watch notification, and then they repeat the same process. A dedicated thread touching the *znode* ensures a constant stream of notifications out of ZooKeeper and towards the clients. We define as latency the interval of time

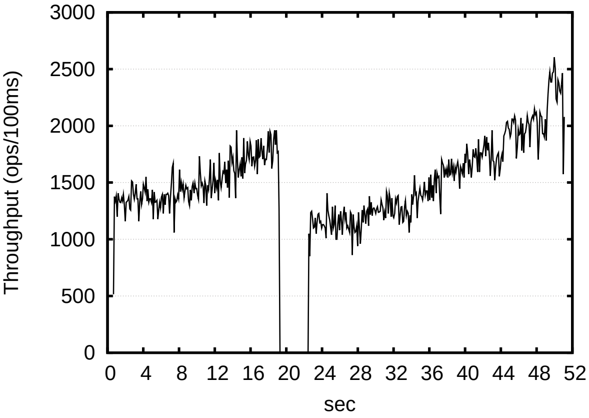


Figure 6.8: YCSB throughput of ACaZoo under leader failure

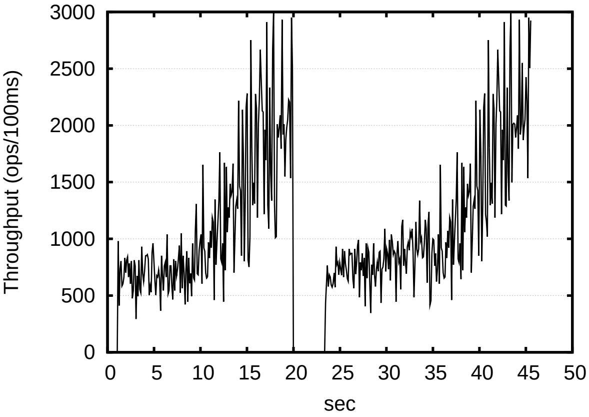


Figure 6.9: YCSB throughput of Oracle NoSQL under leader failure

measured by a thread between two successive notifications.

Cases (a) and (b) appear to behave identically, which is due to the fact that ZooKeeper does not internally load balance watch requests (the use of a front-end DNS server to spray requests equally across servers would be effective in creating an initial balance but cannot be used to re-balance an existing set of notification requests with a highly variable set of clients). Comparing cases (a) and (c) provides evidence that an internally load-balanced ZooKeeper cell will be beneficial for a large set of clients by breaking a single-server bottleneck and partitioning work across ZooKeeper servers. This approach can scale with the number of servers in the cell.

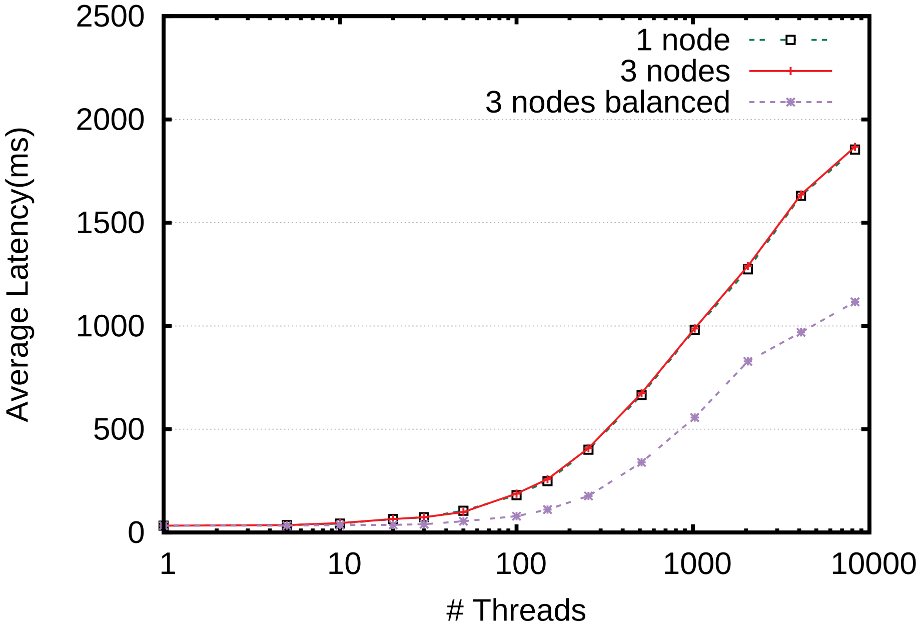


Figure 6.10: Client notification latency from ZooKeeper

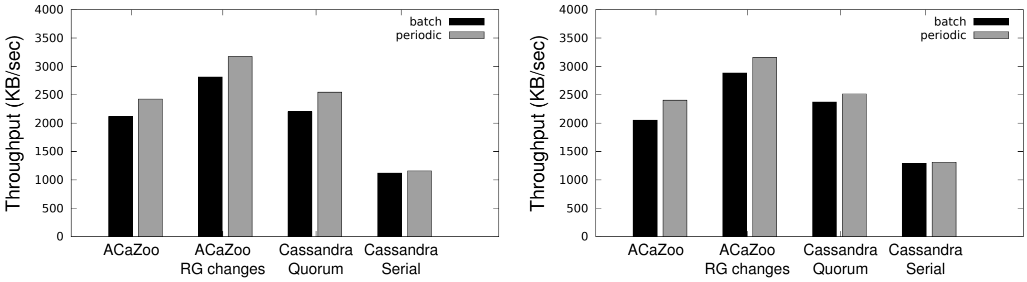
6.8 Application performance

In this experiment we compare the performance of CassMail e-mail service using a single ACaZoo replication group of three and five storage nodes to an equivalent setup of Cassandra with three and five nodes in a ring with replication factor 3 and 5 respectively. We tried two different Cassandra consistency-levels, QUORUM and SERIAL as described in 6.2 and as a benchmark we used Postal with message sizes drawn uniformly at random from a range of sizes. We experimented with two ranges:

- 200KB-2MB (typical of large attachments)
- 50KB-500KB (typical of small attachments)

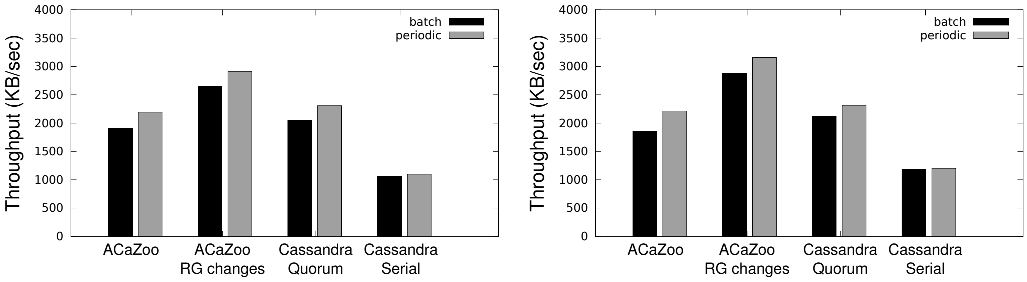
These ranges are chosen to reflect the increase in average e-mail size compared to a related study [49] due to a widespread use of attachments in everyday communications.

Each experiment consists of an e-mail-sending session blasting the CassMail service for about ten minutes. Each node ran an instance of the storage service (ACaZoo or Cassandra) and one node each time ran an instance of the SMTP server (CassMail - Python code). In the Cassandra cluster experiment that node was randomly picked each time while in the ACaZoo cluster experiment the SMTP server was hosted in the same node with the RG leader. In all cases performance was limited by the servers' CPUs. We used a dedicated client having 4 CPUs, 4GB



(a) Thoughput with 50KB-500KB messages (b) Thoughput with 200KB-2MB messages

Figure 6.11: CassMail: 3 Node Replication Group



(a) Thoughput with 50KB-500KB messages (b) Thoughput with 200KB-2MB messages

Figure 6.12: CassMail: 5 Node Replication Group

memory and 20GB remotely mounted disk to drive all experiments. We evaluate all systems under Postal workload of 32 concurrent threads with two different sizes for messages(200KB-2MB and 50KB-500KB). Our measurements are per-minute Postal reports of the sum of e-mail data sent during the previous minute. In all of our graphs we report aggregate average throughput of our measurements.

In Figures 6.11(a) and 6.11(b) we observe that in a replication group of three nodes ACaZoo with RG changes performs better than all the competitors both for small size messages(50KB-500KB) and for bigger size messages (200KB-2MB). In more detail in Figure 6.11(a) ACaZoo with RG changes in periodic mode outperforms standard by 30%, Cassandra Quorum by 24% and Cassandra Serial by 160%. Results for large attachment messages depicted in Figure 6.11(b) are similar. ACaZoo RG changes system in periodic mode performs better than basic by 31%, than Cassandra Quorum by 25% and Cassandra Serial by 140%. We also observe that Cassandra Serial storing larger size messages performs 17% better than storing smaller ones. We attribute the improvement to the Paxos algorithm implementation behind Cassandra Serial explained in Section 2.5. For each Serial operation Cassandra is paying the high price of four round trips. When the penalty being paid comes with larger data stored there is a noteworthy throughput improvement.

As the RG grows larger the performance drops for all systems. Comparing Figures 6.11(b) and 6.12(b) we discovered a 10% performance drop for ACaZoo basic, 8% drop for Cassandra Quorum, 9% drop for Cassandra which is expected when moving to bigger replication groups. In Figure 6.12(a) we observe that in a replication group of five nodes ACaZoo with RG in periodic mode outperforms standard by 32%, Cassandra Quorum by 26% and Cassandra Serial by 164%. In the same replication group, results for large attachment messages as depicted in Figure 6.12(b) are similar. ACaZoo RG changes system in periodic mode performs better than basic by 42%, than Cassandra Quorum by 36% and Cassandra Serial by 132%. Again we have to note that Cassandra Serial storing larger size messages performs 14% better than storing smaller ones.

6.9 Impact of client-coordinated I/O

In this final experiment we quantify the performance benefit due to client-coordination of requests. We run a YCSB read-only workload over a cluster of six RGs and observe improvement of 26% and 30% in average response time and throughput respectively, compared to server-coordination of requests (Table 6.2 summarizes our results).

	Throughput (ops/sec)	Read latency (average, ms)	Read latency (99%, ms)
Server-coordinated	317	3.1	4
Client-coordinated	412	2.3	3

Table 6.2: YCSB read-only workload

Chapter 7

Future Work

Efficient elasticity and data re-distribution is an important research area that we plan to focus on next. A brute force approach of streaming a number of key ranges to a newly joining RG is a starting point but our focus will be on alternatives that exploit the underlying replication mechanism (as in Lorch *et al* [50]). The immutability (write-once) characteristics of LSM-Trees lend themselves to efficient data movement primitives. Furthermore, we plan on further investigating the load balancing methodology for ZooKeeper watch notifications and demonstrating its benefits in increasing system availability under large and dynamic client populations. Another research challenge is in provisioning storage nodes for replication groups to be added to a growing cluster. Assuming that storage nodes come in the form of virtual machines (VMs) with local or remote storage on Cloud infrastructure, we need to ensure that nodes in an RG fail independently (easier to reason about in a private rather than a public Cloud setting).

Chapter 8

Conclusions

In this thesis we described ACaZoo, a NoSQL system offering strong consistency, high performance, and high availability for sharded data intensive NoSQL applications. These properties are achieved via the combination of a high performance replicated data store based on LSM-Trees, client-coordinated I/O, and fast client notifications of cluster configuration changes. The impact of heavy periodic background activity at the master (a challenge with frequent compactions in LSM-Trees) is handled via replica-group leader switches. We examined two policies for RG leader changes (random and round-robin) and found them both effective in delivering a performance improvement of up to 60% in write-intensive workloads. We note that the RG leader-change technique is generally applicable to any primary-backup replication system. ACaZoo overall is shown to exhibit excellent performance and availability compared to a commercial database with comparable architecture and consistency semantics.

Bibliography

[1] P. Garefalakis, P. Papadopoulos, I. Manousakis, and K. Magoutis, "Strengthening consistency in the cassandra distributed key-value store," in *Distributed Applications and Interoperable Systems*, pp. 193–198, Springer, 2013.

[2] P. Garefalakis, P. Papadopoulos, and K. Magoutis, "Acazoo: A distributed key-value store based on replicated lsm-trees," in *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on*, IEEE, 2014.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.

[4] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[5] DataStax, Inc., "Cassandra Lightweight Transactions." `http://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0`, 2013.

[6] L. Koromilas and K. Magoutis, "Cassmail: A scalable, highly-available, and rapidly-prototyped e-mail service," in *Distributed Applications and Interoperable Systems*, pp. 278–291, Springer, 2011.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[8] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, p. 35, 2010.

[9] A. Auradkar, C. Botev, S. Das, *et al.*, "Data Infrastructure at LinkedIn," in *Proc. of the 28th IEEE International Conference on Data Engineering (ICDE)*, (Washington, DC), April 1-5, 2012.

[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10, IEEE, 2010.

[11] X. Li, M. Lillibridge, and M. Uysal, "Reliability analysis of deduplicated and erasure-coded storage," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 3, pp. 4–9, 2011.

[12] Apache Inc., "HBase," 2013.

[13] R. Escriva, B. Wong, and E. Sirer, "HyperDex: A Distributed Searchable Key-vale Store," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 25–36, Aug. 2012.

[14] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s Hosted Data Serving Platform," in *Proc. of the VLDB '08*, (Auckland, New Zealand), August 2008.

[15] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," in *Distributed Systems (2Nd Ed.)* (S. Mullender, ed.), pp. 199–216, New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993.

[16] E. Lee and C. Thekkath, "Petal: Distributed Virtual Disks," *ACM SIGOPS Operating Systems Review*, vol. 30, no. 5, pp. 84–92, 1996.

[17] F. Zunqueira, B. Reed, and M. Serafini, "ZAB: High-performance broadcast for primary-backup systems," in *Proc. of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN'11)*, (Hong Kong, HK), June 2011.

[18] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li, "Paxos Replicated State Machines as the Basis of a High Performance Data Store," in *Proc. of the 2011 USENIX Networked Systems Design and Implementation (NSDI'11)*, (Boston, MA), March 2011.

[19] Apache Inc., "BookKeeper," 2013.

[20] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[21] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," *Distributed systems*, vol. 2, pp. 199–216, 1993.

[22] J. Gray, "Why do computers stop and what can be done about it?," in *Symposium on reliability in distributed software and database systems*, pp. 3–12, Los Angeles, CA, USA, 1986.

[23] D. K. Gifford, "Weighted voting for replicated data," in *Proceedings of the seventh ACM symposium on Operating systems principles*, pp. 150–162, ACM, 1979.

[24] 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*, pp. 8–17, ACM, 1988.

[25] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[26] J. MacCormick *et al.*, "Niobe: A Practical Replication Protocol," *ACM Transactions on Storage (TOS)*, vol. 3, no. 4, 2008.

[27] M. Burrows, "The Chubby Lock Service for Loosely-coupled Distributed Systems," in *Proc. of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, (Seattle, WA), 2006.

[28] Oracle, Inc., "Oracle NoSQL Database." `http://www.oracle.com/technetwork/products/nosqldb/learnmore/nosql-wp-1436762.pdf`, 2012.

[29] Redis, "Redis." `http://redis.io/`, 2013.

[30] C. Commons, "Mongodb."

[31] "Couchbase."

[32] OrientDB, "OrientDB." `http://www.orientechnologies.com/orientdb/`, 2013.

[33] Neo4j, "Neo4j." `http://www.neo4j.org/`, 2013.

[34] Judd, D., "Hypertable: An open source, high performance, scalable database." `.http://hypertable.org/index.html`, 2007.

[35] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[36] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.

[37] R. Sears, M. Callaghan, and E. Brewer, "Rose: Compressed, log-structured replication," in *Proc. of PVLDB'08*, (Auckland, New Zealand), August 2008.

[38] J. Shute, S. Oancea, B. Ellner, *et al.*, "F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business," in *Talk given at ACM SIGMOD-/PODS 2012 (industrial presentations)*, (New York, NY), June 22-27, 2013.

[39] L. Qiao, K. Surlaker, T. Quiggle, *et al.*, "On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform," in *Proc. of ACM SIGMOD-/PODS 2013 (industrial presentations)*, (New York, NY), June 22-27, 2013.

[40] N. Bronson, A. Z., G. Cabrera, *et al.*, "TAO: Facebook's Distributed Data Store for the Social Graph," in *Proc. of 2013 USENIX Annual Technical Conference (ATC'13)*, (San Jose, CA), June 26-28, 2013.

[41] J. C. Corbett, J. Dean, M. Epstein, *et al.*, "Spanner: Google's globally-distributed database," in *Proc. of 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12)*, (Holywood, CA), October 8-10, 2012.

[42] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable Consistency in Scatter," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (Cascais, Portugal), 2011.

[43] Shafaat, T. and others, "ID-Replication for Structured Peer-to-Peer Systems," in *Proc. of Euro-Par 2012*, (Rhodes, Greece), 2012.

[44] A. Shraer, B. Reed, D. Malkhi, and F. Junqueira, "Dynamic reconfiguration of primary/backup clusters," 2012.

[45] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *Proc. of the 2010 USENIX Annual Technical Conference (ATC 2010)*, (Boston, MA), June 2010.

[46] Birman, K., and others, "Overcoming CAP with Consistent Soft-State Replication," *IEEE Computer*, vol. 45, no. 2, pp. 50–58, 2012.

[47] "Java New I/O API and Direct Buffers," 2013.

[48] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with YCSB," in *Proc. of 1st ACM Symposium on Cloud computing (SoCC'10)*, (Indianapolis, IN), Jun. 2010.

[49] Y. Saito, B. N. Bershad, and H. M. Levy, "Manageability, availability and performance in porcupine: a highly scalable, cluster-based mail service," in *ACM SIGOPS Operating Systems Review*, vol. 33, pp. 1–15, ACM, 1999.

[50] 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 *Proc. of EuroSys'06*, (Leuven, Belgium), 2006.

Glossary

ACaZoo is a system combining the names of the two systems it loosely derives from: Apache Cassandra and Apache Zookeeper.

Bloom Filter is a space-efficient data structure used as an approximate membership query, or a negative-cache for lookups.

CAP this theorem states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees: Consistency, Availability, Partition tolerance.

CassMail a scalable e-mail service application written in Python using Cassandra/ACaZoo as a back-end.

CommitLog a log where all writes are logged to prevent data failure on crashes.

CPU or central processing unit is the hardware within a computer that carries out the instructions of a computer program by performing the basic arithmetical, logical, and input/output operation.

I/O is an acronym referring to Input-Output.

Lightweight Transaction termed introduced in Apache Cassandra version 2.0 with the same meaning as transactions linearlizability.

Linearizability is a system property where the result of any execution is the same as if the operations of all the processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.

LSM-Tree is a acronym referring to a Log-structured Merge-Tree data structure, it is also the underlying data structure behind SSTable and Memtable.

Memtable is referring to In-memory SSTable and is basically acting as a write-back cache of data where rows can be looked up by key.

Paxos is a family of protocols for solving consensus in a network of unreliable processors. Consensus is the process of agreeing on one result among a group of participants.

POP3 is one of the most widely accepted Internet standard protocols for e-mail retrieval.

Quorum is the minimum number of votes that a distributed transaction has to obtain in order to be allowed to perform an operation in a distributed system.

Replication is a general method for implementing a fault-tolerant service by copying state and data to multiple servers.

Replication Group a group of machines responsible for replication.

RG is a acronym referring to a replication group.

Serial Consistency termed introduced in Apache Cassandra v2.0 with the same meaning as transactions linearlizability.

SMTP is an acronym referring to Simple Mail Transfer Protocol which is an Internet standard for electronic mail transmission.

SSTable is a serialized memtable. It is an on-storage list of sorted tuples, along with a serialized secondary-index, and a serialized Bloom Filter.

VM is a software-based emulation of a computer. Virtual machines operate based on the computer architecture and functions of a real or hypothetical computer.