The Pennsylvania State University
The Graduate School

DESIGN AND IMPLEMENTATION OF AN AUTOMATED EXPERIMENTATION FACILITY FOR A RECONFIGURABLE GEO-DISTRIBUTED KEY-VALUE STORE

A Thesis in
Computer Science and Engineering
by
Praneet Soni

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2020

The thesis of Praneet Soni was reviewed and approved by the following:

Bhuvan Urgaonkar
Associate Professor of Computer Science and Engineering
Thesis Co-Advisor

Viveck Cadambe
Associate Professor of Electrical Engineering
Thesis Co-Advisor

Chita R. Das
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

Abstract

Distributed Storage Services (DSS) have proven their worth, in providing highly available and reliable data access to the applications served by them. But with the surge in demand for these services, a cost-aware performance analysis of the system becomes more pertinent. Erase Coding (EC) can provide significant savings in storage capacity and network costs, while providing same fault tolerance guarantees, as compared to the replication based schemes employed by current state-of-the-art systems. EC provides a competitive alternative for storage services which require strong memory consistency guarantees. This work aims to provide an automated experimentation facility, which can be used to explore the system performance under different storage configurations. The framework implemented in this work, supports a combination of EC based and replication based linearizable key-value stores, across a geo-distributed setting. The framework facilitates protocols for concurrent non-blocking read/writes by multiple clients. The framework can also generate client requests, based on workload specifications provided by the user. The specification allows workload attributes to change over the course of the experiment. In conjunction with dynamic workloads, the framework also implements a reconfiguration protocol to apply the new storage policies chosen for the workload, while ensuring minimal client service disruption. The framework provides sufficient flexibility in exploring the design space of storage configurations, without compromising on system performance.

Table of Contents

List of Figures vi

Acknowledgments vii

Chapter 1
Introduction 1

Chapter 2
Background and Motivation 4
2.1 Linearizability . . . 4
2.2 Erasure Coding . . . 5
2.3 The Case for Reconfiguration . . . 6
2.4 Contributions of this work . . . 7

Chapter 3
System Overview 8
3.1 Hybrid Key-Value Store . . . 8
3.2 Automated Experiment Facility . . . 9
3.2.1 Scalability of the Facility . . . 10
3.2.2 Workload Characterization . . . 10
3.3 Reconfiguration Framework . . . 11
3.3.1 Introduction . . . 11
3.3.1.1 Basic Steps of reconfiguration . . . 11
3.3.1.2 Challenges . . . 12
3.3.2 Performance Trade-offs for the reconfiguration . . . 12

Chapter 4
Overall Design and Implementation 13
4.1 Introduction . . . 13
4.2 System Components and their Key Responsibilities . . . 14
4.2.1 Client Proxy . . . 14
4.2.2 Data Server . . . 14
4.2.3 Controller . . . 15

Chapter 5
Automated Experiment Facility **16**
5.1 Introduction 16
5.1.1 Input and Control Variables 16
5.1.1.1 Input Configuration 16
5.1.1.2 Environment Setup 17
5.1.1.3 Control Variables 18
5.2 Implementation Details 18
5.3 Implementation Challenges 21

Chapter 6
Reconfiguration Framework **22**
6.1 Control Flow for a reconfiguration 22
6.2 Reconfiguration library at the Controller 23
6.2.1 Initialization 23
6.2.2 Implementation of the protocol 24
6.3 Reconfiguration library at the Client Proxy 25
6.4 Reconfiguration library at the Server 26
6.4.1 Blocking the Client Requests 26
6.4.2 State Information per Reconfiguration 27
6.4.3 Additional Request Messages 27
6.5 Implementation Challenges 27

Chapter 7
Conclusion **29**

Bibliography **30**

List of Figures

5.1 Summary of the *Properties* struct 17

6.1 Parsing the input *WorkloadConfig* struct 24

Acknowledgments

I would first like to thank my thesis advisor Prof. Bhuvan Urgaonkar, of the Department of Computer Science and Engineering at Pennsylvania State University. The door to Prof. Urgaonkar's office was always open whenever I hit a roadblock in my research or writing. He consistently provided me the guidance to ensure that I was making positive progress in the project.

I would also like to thank my co-advisor Prof. Viveck Cadambe, of the Department of Electrical Engineering at Pennsylvania State University. He has been an important contributor of many novel ideas that have been implemented in this work. I am gratefully indebted to him, for his very valuable insights on the project implementation.

I would also like to thank my research colleagues, Nader Alfares and Hamidreza Zare, as their contributions have not only planted the initial seeds of this work but are also indispensable for its completion. Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Chapter 1
Introduction

Most web applications in the recent years, have made a transition to cloud storage systems for their storage needs. These storage systems ensure high availability and faster response times for all global clients, by carefully placing the data across a variety of geo-distributed servers. The replication of data across multiple storage sites provides fault-tolerance and enables the application to meet tighter Service Level Objectives (SLOs) for all clients. Most applications organize their data as a set of key-value pairs and the access to data is provided by a Get/Put interface API. Spanner [1] and Cassandra [2] are some popular object stores that use this interface. Much work has been done to make these replication based systems perform faster. A prominent technique is to use in-memory key-value stores to improve the server response time and avoid the I/O overhead of disk access. Some really successfull implementations of in-memory stores include, Memcached [3] and Redis [4], which are widely used in Facebook, twitter and Linkedin. These solutions offer a significant performance improvement over disk based solutions, but they have some challenges. Since, the amount of data that can be stored in memory is constrained, the performance of these systems relies a lot on optimal cache eviction policies and efficient memory utilization techniques.

However, the availability and fault-tolerance of replication based systems comes at a high cost. The entire objects need to replicated across multiple machines, which has a significant storage overhead. Additionally, the network traffic also increases in proportion to the number of object replicas stored. To provide a fault-tolerance of f failures, these systems need to maintain atleast $f+1$ object replicas, at all times. The number of replicas needed increases to $2f + 1$ for Quorum based implementations - in which Read and Write Quorums are defined separately, with some overlap. Having different sized Read and Write Quorum allows for latency optimisation, based on the workload read/write ratios. Erasure coding (EC) is a space-efficient way to provide data redundancy. EC

can provide the same availability and fault-tolerance at a fraction of the storage cost [5]. EC primarily works by encoding an object into multiple small chunks, which are then stored on different servers. This requires that multiple servers be contacted on each Read and Write operation, when erasure coding is used to store the objects. This incurs a latency penalty, especially in the context of geo-distributed data servers. This may lead us to conclude that EC doesn't provide optimal latency bounds. To analyze that claim more closely, we need to take into consideration another workload requirement - memory consistency.

For weak memory consistency models, not all the replicas need to be in sync at all times. One of the popular weak consistency model is called eventual consistency [6], wherein the write operation can terminate by operating on a subset of the replicas and the changes are eventually communicated to other data sites. The replication based scheme, thus needs to access only a small set of servers for completing the read/write operation. So, EC doesn't offer any advantage for such services. But certain applications, such as Google Docs, require stronger consistency guarantees, which ensures that the data across servers is consistent at all times. This mandates that all or a majority of the copies, in both the schemes, be contacted and updated before the operation can terminate. This requirement eliminates the difference in latency penalties, for replication based schemes and Erasure Coded schemes.

Given the requirement of implementing a linearizable key-value store, EC based storage policy can provide significant benefits. EC just creates a small storage overhead over the original object, by creating some parity chunks, to provide strong fault-tolerance guarantees. Apart from the storage savings offered by EC, it also reduces network transfer significantly. Instead of transferring entire object to each server, now each server just communicates a single encoded chunk. Thus, the network cost savings for each operation is significant. EC also helps reduce tail latencies to a great extent by offering late binding [7]. An object value can be reconstructed by using a subset of the chunks, so the stragglers in the system won't be harming the tail latencies for the Get/Put operations. Such an analogy cannot be drawn for replication based scheme. Another challenge faced by replication based scheme is the popularity skew, which degrades the I/O performance for some machines due to the load imbalance. This can lead to slower response times for all keys being serviced from the busy machines. EC, by virtue of its design, avoids this problem. Since, the objects are divided in smaller chunks and distributed across servers, the load is also spread across many machines now. This property of EC allows it to increase the system throughput, as all chunks can be fetched in parallel now, instead

of being bandwidth bound.

However, certain arguments are made against the use of EC and its application to large-scale key-value stores, that have prevented its widespread acceptance as a viable alternative. One argument is that Erasure coding induces latency because of additional computational load, as it requires encoding and decoding for each operation. But with the advent of powerful compute resources and the development of high performance erasure code libraries [7–9], encoding/decoding can now be done online [10]. It's also argued that reconfiguration of data, to accommodate dynamic object demand, is too complex for EC based storage. We challenge these arguments in this work and provide protocols that prove the merit of EC in distributed KV-stores.

The rest of the thesis is organized as follows. In the next section, we talk about the requisite background information about Linearizability and Erasure Coding (EC). It concludes with a summary of the contributions of this thesis. Section 3 elaborates on the features of the framework implemented in this work. Section 4 provides a high level view of the system components. Sections 5 and 6 describe the details of the implementation, before finally concluding in Section 7.

Chapter 2
Background and Motivation

Linearizability provides a unique opportunity for Erasure coding. It establishes EC as a serious contender for an optimal storage policy. Let us understand the details of this memory consistency model

2.1 Linearizability

This is also called the strong consistency model. It ensures that all writes in the system appear ordered and the reads return the value of the latest write. Ideally, every application would prefer such a consistency model, but in the context of a distributed system such models incur additional latency and cost penalty for each operation. Hence, wherever possible applications operate with weaker consistency guarantees, for a faster system performance. But certain applications require strong consistency guarantees to function correctly. Strong consistency requires that a majority of the servers be accessed for each operation and all copies be updated in an atomic fashion. A straight forward way to address atomicity is to implement a lock-based blocking protocol. But this implementation will be severely impacted by the stragglers in the system.

A non-blocking protocol, that provides linearizability guarantees, for replication based Key-value stores has been proposed in the literature, called ABD[1] . Modified version of this protocol allows multiple clients to perform read/write operations concurrently. This is done by making both read and write 2-phase operations. The operations are committed only after propagating the values to a majority quorum of servers. That ensures that a subsequent operation will see the value of the last operation for sure and in this way linearizability constraints are respected. This protocol assumes that all servers

[1]Acronym has been derived from the names of the authors - Attya, Bar-Noy and Dolev [11]. Here we use it to refer to a multi-writer variant of the algorithm, which can be found in [12]

are symmetric and provides a decentralized approach to distributed consensus for the simple Get/Put operations. This helps avoid any latency bottlenecks.

Similarly, a non-blocking protocol, that accomplishes the aforementioned objectives, for EC based schemes is named Coded Atomic Storage (CAS) [13]. In principle, CAS also ensures that each operation has a write back stage, so that all future operations see the last completed operation. But it's more optimal as compared to ABD, as the write-back doesn't require additional propagation of data value, just the metadata. However, a challenge here is the requirement to hide any ongoing writes from the reads, until they are safely propagated to all servers. If that is not ensured, we may violate linearizability guarantees on server failures. This requirement warrants a 3-stage write operation in CAS. The reader can find additional qualitative assessment about these protocols in [14, 15].

This work uses a combination of replication and EC based schemes for it's key-value storage requirements. So, an implementation of ABD and CAS protocols are used, for servicing read/write operations in the respective storage scheme.

2.2 Erasure Coding

Erasure Coding provides a space efficient way to store data and provide redundancy. We are using a coding scheme called Reed-Solomon code (RS-code) [16], which is extremely popular in the systems deployed. The code is parameterized by two quantities: k, the number of data chunks and n, the number of total chunks. The code divides an object into k data chunks. It also generates $n - k$ parity chunks, by some linear combination of data chunks. It's sufficient to use any k chunks out of the total n chunks for reconstructing the original object. Thus, these codes are tolerant to $n - k$ server failures and the memory efficiency is given by k/n. Erasure coding has been long used in disk based system to provide fault-tolerance [17, 18].

Given these encoding characteristics, we can see that EC provides inherent load balancing capabilities, as the client requests can be diverted to any of the data or parity chunks. Also, we can significantly curtail the tail latency by incurring slightly additional bandwidth for each request. Since, any k chunks can be used to accomplish a read, if we send out additional read requests and accept the first k arrivals, we can achieve really competitive tail latency. It has been shown, that one additional request for a parity chunk is sufficient to reduce tail latencies and any additional chunk requests don't offer significant improvements in the latency [7]. This shows that we can effectively reduce the tail latency, by incurring a slight network overhead.

Thus, there is a strong case for EC as an optimal storage strategy, for strongly consistent key-value stores. Although, this framework uses RS-codes for it's operation, there are other erasure codes proposed which are more efficient for data repair operations, like Minimum-storage regenerating (MSR) codes [19].

2.3 The Case for Reconfiguration

The distributed storage backends support a variety of applications, deployed on the cloud ecosystem. These application can have a wide range of performance requirements and workload characteristics. As we previously discussed, latency bounds, memory consistency requirements, and cost budgets are some of the prominent examples of these requirements. To satisfy these requirements, the system administrator has to choose from a gamut of configurations, such as storage protocol, number of replicas, primary/secondary replicas, placement of replicas, fault tolerance and many more. This involves analyzing a slew of cost-performance trade-offs for each configuration choice and its interplay with respect to others. But the challenge that is faced by these backends is the fact that a static configuration chosen, after an extensive evaluation, maybe an optimal choice only for a small duration of time.

This is because, the workload characteristics of the application may undergo short-term and long-term changes. Request load from different parts of the globe may peak at different times of the day - this is an example of short-term change. Long term change can be driven by the application's use case and its popularity. An application may grow more popular in some regions over time and loose its demand in some other parts. In fact, a work by Facebook [20], argues for data reconfiguration not just to reduce latencies, but to avoid unnecessary inter-datacenter bandwidth utilization. Reconfiguration helps avoid network bottlenecks and also saves cost. This requires that the storage service periodically reconfigure the data for each key, based on it's requirements and the workload attributes.

This reconfiguration should happen automatically and that entails two important questions - what should the new configuration be and how to transition a live system onto the new configuration. A lot of research has been done to answer these questions for the case of replication based schemes. [21, 22] talk about the importance of reconfiguration and how to perform it optimally in a geo-replicated storage, for both eventual and strong consistency. Eventual consistency introduces additional complexity because of the presence of primary and secondary replicas. The work by [23] dynamically computes the

optimal configuration for eventually-consistent geo-replicated stores.

In this work, we answer these questions for the case of EC based storage and implement novel reconfiguration protocol to safely perform the reconfiguration for our hybrid key-value store. We also utilize an optimizer to determine the optimal configuration after each workload change, details of which can be explored here [14].

2.4 Contributions of this work

Given the discussion till now, this work aims to create an automated facility, for experimenting with a variety of dynamic configuration choices and the corresponding reconfiguration protocol. This will enable a quantitative and qualitative assessment of these configurations, on the cost and performance of the applications. Following is a summary of the key contributions of this work.

- Provides an implementation of Hybrid Key-value store, which can service Load/Store operations using both replication and erasure coded schemes.
- Provides an implementation of an Automated Experiment Facility, that is both scalable and highly configurable, for generating a variety of request loads.
- Provides an implementation of a reconfiguration framework, that automatically and periodically initiates the reconfiguration for requisite keys, and ensures the transition to new configuration in a safe manner.

This work will provide a flexible platform for a variety of evaluation experiments. The framework covers all aspects of the cost-performance trade-offs deemed important in the aforementioned discussion.

Chapter 3
System Overview

In this chapter, we will briefly describe the capabilities and features of the various system components. This sections aims to elucidate the functionality available with this framework.

3.1 Hybrid Key-Value Store

This work implements a Key-Value Store (KV store), which can support both replicated and erasure coded data objects as part of its storage system. These storage schemes can be chosen on a per-key basis and they are decided by an optimizer, based on the characteristics of the workload. The implementation uses non-blocking protocols for both the storage schemes, called *ABD* and *CAS*. Both of these protocols follow a quorum based approach to perform an operation. What that means is that each operation only operates on a subset of the data servers. If the quorum sizes satisfy some constraints, the safety and liveness properties of these protocols can be proved. These protocols utilize unequal quorum sizes for different stages of read/write operation and this can be used to prioritize latency reduction of one operation - based on the workload read/write ratio. These algorithms are also non-blocking distributed consensus algorithms. That makes them more resilient to failures. It also allows multiple concurrent operations, to the same key, be active in the system. This greatly reduces the tail latencies, as the operations are no longer bound by the stragglers in the system. To read more about these algorithms, please refer to [11, 13, 14]

The implementation of both these protocols has been done using *C++* classes. Each request made by the client encapsulates the storage protocol to be used, while fulfilling the request for that key. The server responds to the request based on the protocol specified by the client. There is sufficient modularity in the design to accommodate

further storage schemes in the future.

3.2 Automated Experiment Facility

This is a key component of the framework, which is responsible for generating client requests to the servers deployed. The underlying design was created with the intent to allow sufficient freedom, in specifying a variety of workload parameters. So that different configurations and protocols, deployed on the servers, can be studied extensively and their impact on performance can be quantified.

Through this facility, attributes of the workload required by user is provided to the Controller - which is the central entity of our system. The Controller then communicates these attributes to all the client proxies, which are currently deployed, for them to generate requests based on the requirements specified by the attributes. The Controller is aware of the client proxies which are currently active and communicates with only them. But our solution is still a distributed, leader-less one as the Controller is only responsible for coordinating activities in the control plane, as we will see in section 4.1. As long as Clients have information about the data placement for the key, read/write operations can continue without the Controller's intervention.

Since, a major motivation for such a facility was to experiment with dynamic reconfiguration, based on workload, the facility enables workload patterns to be changed over time. The client proxies are provided that information in the communication from Controller and they change their read/write request generation pattern at the specified offsets from the start time. The workload attributes can be defined for each key separately, if that's the need of the experiment. But to make the experiment tractable, generally keys with similar workload properties are combined into a group and a common set of attributes are specified for all the keys in the group. The facility can add or remove keys/key-groups, for which it is generating read/write operations, over the course of the experiment. This is important to simulate the variation in hot and cold keys, in the cloud systems. This work also allows control over the locations from which client requests should originate. The user can specify what portion of the overall requests for a key, should originate at a particular client proxy. This provision enables the simulation of many real-world cloud workloads, where the key popularity is skewed with respect to different geographic regions. We will now discuss some details on how the facility enables experiment scaling and will more clearly define the workload attributes accepted by the facility.

3.2.1 Scalability of the Facility

Any test facility should ensure that the experiments can be scaled both horizontally and vertically. This work ensures that both of these scaling can be enabled quickly.

- Horizontal scaling, in this context, implies increase in the number of entities generating the requests, i.e, client proxies. This facility can accommodate any number of client proxies. All the user needs to do is, provide network addresses of the proxies to the Controller, at the start of the experiment. No part of the design places any upper limit on the number of such proxies.
- Vertical scaling, implies increase in the number of requests per client proxy. The user can easily specify that request number as part of the input workload attributes. There is an upper limit that will be imposed by the underlying operating system, due to the way current implementation works. But that upper limit, of the order of tens of thousands of requests per second per client proxy, will be sufficiently large for most experiments. We will discuss how to maximize this scaling factor in section 5.1.1.

3.2.2 Workload Characterization

It's important to clearly define all the different workload properties that can be specified by the user, to generate the required workload from this facility. The following set of properties are specified for each key group (group can contain a single key if required).

- *Timestamp* : This is the offset, from the start of the experiment, at which these workload properties take effect at the client proxy.
- *Client Distribution* : Specifies the fraction of the overall request rate that will be generated by each client proxy. This is maintained as an array, with an entry for each active client proxy.
- *Object Size* : Specifies the size, in bytes, of the value that is read and written as part of the client operations.
- *Arrival Rate* : This is the total request rate for that key group. It specifies the number of requests per second that needs to be generated.
- *Read/Write Ratio* : Specifies the representation of reads and write in total requests.

- *Duration* : Specifies the duration, in seconds, for which the client proxy should generate client requests based on these parameters.

3.3 Reconfiguration Framework

Now that we have a dynamic workload generator, we need to equip our KV-Store to handle these changes in the client requests, so as to not degrade the system performance adversely. This requires that we reconfigure the storage policies for keys, whose workload patterns have changed. This involves re-evaluating the storage protocol to use, *ABD* or *CAS*, and the corresponding data placement for the key. As we discussed in section 2.3, this is essential to operate the system in an optimal cost-performance setting.

3.3.1 Introduction

Reconfiguration involves two major operations. First, we need to decide what the new optimal configuration for the key should be. This is accomplished using an optimizer, that does an exhaustive search of the parameter space and finds the best configuration, which satisfies the read/write latency bounds. To understand the optimizer in better detail, refer to [14]. Second, we need to transition the current system to the new state, adhering to the configuration chosen by the optimizer. This is the operation that's being accomplished by the reconfiguration framework implemented in this work.

The framework implements one instance of a reconfiguration protocol, designed to safely transition the keys to the new storage configuration. The reconfiguration is initiated by the Controller and it coordinates with all servers involved in the configuration to complete it. Let's understand the basic steps that need to be undertaken by any reconfiguration protocol and what are the challenges in safely doing so.

3.3.1.1 Basic Steps of reconfiguration

In concrete terms, reconfiguration of a key involves changing the placement of data fragments in the system and changing the way those data fragments are interpreted on a read/write. In this work, we also make sure that the protocol is general enough to handle both ABD and CAS protocols. Following is a high-level summary.

1. Determine the metadata/tag for the latest write to the key, that is being reconfigured.

2. Fetch the value corresponding to the Tag, using the information from old configuration. In ABD, this involves just reading a replica from old quorums. In CAS, it involves reading fragments from the old quorum and decoding them based on the encoding parameters specified by the old configuration.

3. Writing the value to the new configuration. In ABD, this implies writing the replicas to new quorums. In CAS, this entails creating fragments based on the new encoding parameters and writing to the quorums specified in new configuration.

3.3.1.2 Challenges

We need to make sure that reconfiguration doesn't violate the linearizability guarantees provided by the KV-store. So, we need to ensure that the protocol is safe (the operations appear atomic) and live (all operations terminate). To accomplish these requirements, the servers involved in the old configuration block all client requests for that key when reconfiguration is initiated. After it's completion, all requests with key versions older than or same as the key version chosen by the protocol, are serviced by the old servers. All other pending requests are redirected to the new quorums. This is done so that incomplete writes are not serviced by the old quorum and they need to be re-attempted again with the new configuration.

3.3.2 Performance Trade-offs for the reconfiguration

The reconfiguration in its current form requires that client requests be blocked at the server for the duration of the reconfiguration. This introduces some overhead for the reconfiguration. However, our argument is that this doesn't take away anything from the significant gains that it offers. The duration of the reconfiguration protocol can be approximated by 3 Round-Trip-Time (RTT), from the Controller to the farthest data server. So, if the workload parameters are constant for epochs of duration larger than that, then the gains of an optimal configuration will provide substantial savings in latency and cost. This protocol also requires some state to now be maintained on a previously state-less servers. But the states are short lived and can be managed with an efficient implementation.

Chapter 4
Overall Design and Implementation

In this section, we will describe the individual components in the system and how they interact with each other to accomplish the goals of this work.

4.1 Introduction

The system implemented in this work consists of three main entities - Controller, Data Servers and Client Proxies. All these entities are standalone C++ applications, designed to be run on Virtual Machines(VMs) deployed all across the globe. All the data communication between these applications is serialized first and is then sent across using a sockets implementation provided by the underlying OS.

The users of the storage service issue commands to a system Front-end called client proxy, which is co-located with some Data Servers in a Datacenter. The requests from the users are expected to be received at the nearest Client Proxy. It's the role of the Client Proxy to initiate requisite storage protocol, either ABD or CAS, and communicate with the Data Servers to facilitate the client request. We do not implement the individual users, but instead simulate their presence by generating the requisite workload from each Client Proxy. This is sufficient for the performance analysis of the KV-store. User's connection to the Proxy doesn't have any bearing on the operational aspects of the storage service. Both the storage protocols, ABD and CAS, have been implemented with a C++ class, which has a client side component and a server side component.

In our setup, the Controller initiates the experiment by contacting all the available Client Proxies and providing them the requisite configuration information. Client Proxies then generate the required request rates, as dictated by the controller, and contact the

Data Servers quorums based on the saved configuration for each Key Group. At the specified time offsets, Controller initiates the reconfiguration protocol and client proxies also adopt the new workload attributes for that key group.

4.2 System Components and their Key Responsibilities

4.2.1 Client Proxy

The automated experiment facility, described in section 3.2, is deployed on the VMs designated as Client Proxies. They are compiled with the client-side implementations of the storage protocols, so that the class functions can be used to invoke Get/Put requests at Client Proxy. At the start of the experiment, the Proxy will be contacted by the Controller and all the requisite information for that experiment will be communicated to it. The proxy is basically provided with a list of workload attributes, wherein each entry is timestamped - by providing an offset value from the start time.

It's the responsibility of this facility to generate client requests for each key group, while ensuring all the workload properties for each group are satisfied, at each instant in time. The implementation also needs to ensure that the timestamp at which workload configurations are stipulated to change, must be followed religiously by the facility. It's because, the proxies do not explicitly synchronize with each other. Finally, the client implementations at these proxies should respect the reconfiguration protocol and ensure that the Get/Put operations failed during the reconfiguration, should be retried with the servers in the new configuration.

4.2.2 Data Server

Data Servers are the backbone of the KV-Store and these are the entities which interface with the storage devices to manage all the Key-Value pairs. Servers also have an implementation of both the storage protocols, ABD and CAS, so that they can coordinate with the clients and provide them the requisite value. Each Data Server maintains an in-memory cache to serve Get/Put requests for the most frequent keys faster. It also uses a persistent storage for all the data it stores. This work uses a slightly tailored LRU cache implementation from [24] and uses RocksDB [25] for persistent storage. Servers should ensure that the client requests are handled based on the storage protocol specified in each Get/Put request.

The servers will be contacted by the Controller whenever a reconfiguration is initiated. The servers should block client actions, for the corresponding key, until the reconfiguration is complete. It also needs to service the buffered requests in accordance with the reconfiguration protocol, after it's completion.

4.2.3 Controller

Controller is the central entity in our setup, which is responsible for initiating any experiment. Controller maintains the metadata information for all keys in the system. Since, the configuration for the keys keeps changing, the controller updates the mapping of each key to reflect its latest configuration. The metadata at the controller can be queried by clients for the most recent value, as it will represent a consistent view of the current configuration for each key.

The main responsibility of the Controller includes reading the workload properties provided by the user. It then parses it and stores it in a local data structure. The Reconfiguration framework we discussed in section 3.3, is deployed on the Controller VM. The framework uses the local data structure as an input to the Cost Benefit Analyzer and generates a list of client configurations. This output is then serialized and sent to all the active Client Proxy nodes. Another important role of the Controller is initiation of the reconfiguration protocol. At the specified time offsets, Controller begins the reconfiguration protocol and ensures that key metadata is updated as part of the protocol.

Following are the input files read by the Controller. All the input files are placed inside the *config* folder in the project directory root.

- `input_workload.json` : A list of workload properties for each key group, with a timestamp for every entry. Also contains some global configuration parameters, such as start time of the experiment, number of retries, etc.
- `setup_config.json` : Network addresses (IP and port number) of the Data Servers.
- `deployment.txt` : Network addresses of the Client Proxies which are part of the experiment.

Chapter 5
Automated Experiment Facility

In this chapter, we will be understanding the specifics of the implementation and also how to initialise it.

5.1 Introduction

The entire implementation of this experiment facility is deployed on the Client Proxy node. This facility needs to be initialized by the Controller, before it can begin it's operation. This facility aims to support a high request rate per second, for each key and that implies a very high degree of concurrency in the system. Since, each client operation is a high latency operation, owing to the geo-distributed setting of the experiment, we leverage the help of multi-threading to sustain high request rates at each Client. We will see what are the challenges and the opportunities such a multi-threading approach presents. The design choices made were motivated by two primary goals. First, to ensure easy scalability of experiments and second, provide a flexible format for input workload specification.

5.1.1 Input and Control Variables

We will now elaborate on the input configuration of the facility. We will also discuss the Linux environment setup that needs to be done, so that scaling bottlenecks can be alleviated.

5.1.1.1 Input Configuration

All the input configurations to this facility have to be provided by the Controller. So, it's imperative that the Controller is aware of the Client Proxy being active. To

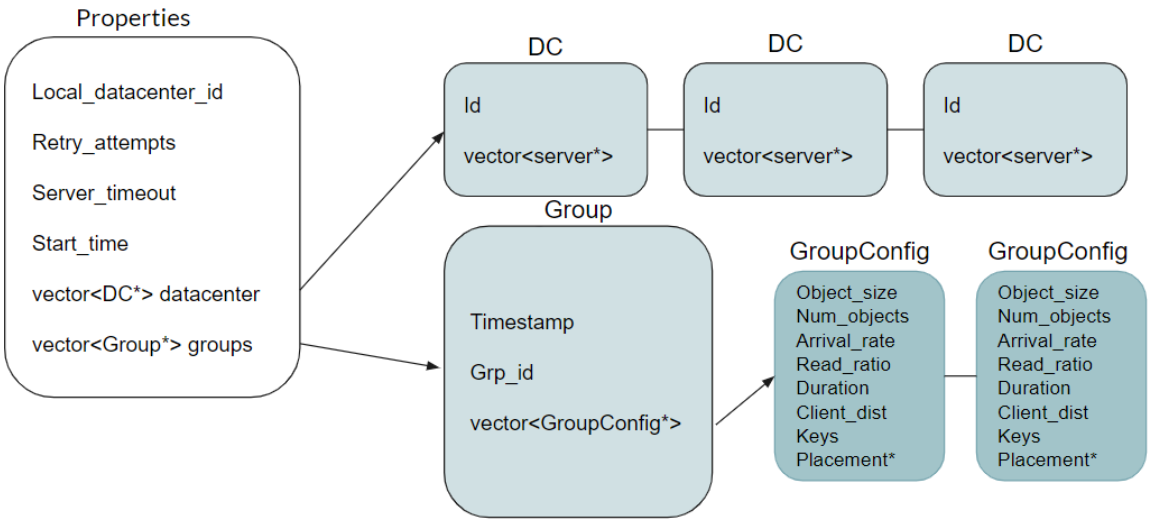

Figure 5.1. Summary of the *Properties* struct

accomplish that, user should specify the network address of the Client proxy as part of the file `deployment.txt`. The important input properties required by this facility are start_time, retry_attempts and workload attributes. The *Properties* struct defined in `Util.h`, describes the information sent by the Controller to the Proxy. The figure 5.1 provides a high level view of the data contained in the struct. The struct is defined in such a way that the user has full autonomy over the number of key groups active in the system and the workload attributes of each key-group. The implementation ensures that the keys can be added to the system at any time, and keys can also migrate between different key groups.

5.1.1.2 Environment Setup

As discussed earlier, ensuring a scalable experimentation facility was a key goal of this work. However, given the nature of the implementation, we need to increase the default limits of some of the system resources on Linux. This is necessary to ensure that the experiment can be scaled to sufficiently high rates.

1. Open files per process : This restricts the total number of file descriptors for each process. Use *ulimit -n* to set it to the hard limit provided by your system.

2. Size of the TCP receive and accept queues: These sizes are specified by the kernel system parameters `net.ipv4.tcp_max_syn_backlog` and `net.core.somaxconn`. You can set these parameters either using *sysctl* command or write directly to the

parameter files in /proc/sys/. Also, make sure that you set the constant `BACKLOG`, in file `Util.h`, to a value higher than or equal to `net.core.somaxconn`.

A comprehensive list of other possible system imposed scaling bottlenecks can be found in this article [26].

5.1.1.3 Control Variables

The variables that are taken into consideration, by the experimentation facility, for generation of client requests are: Client Distribution, Object size, Arrival Rate, Read ratio (write ratio is the complement of that), and the duration (in secs). All of these variables can be found in the struct *GroupConfig*, from the file `Util.h`

5.2 Implementation Details

When the facility is first run by the user, it starts a socket server and waits for a connection request from the Controller. Once it receives the data, the server and the socket connection, both are closed. The data received is then deserialized, using the Protobuf [27] library, and fed into the struct *Properties*. This struct contains the start time, as a numerical entity. This numerical entity is converted to a valid *timePoint* object of C++. This timePoint object is the reference time, which is communicated to all the Client Proxies. Since, all these applications(Clients and Controller) can use a global system clock to synchronize their actions, they don't explicitly use messages to synchronize. Now, all the offsets provided in the workload attributes are added to this reference time and the absolute time for that configuration is calculated.

An important aspect of this implementation is the design that's responsible for generating the specified Arrival rates(Request rates). As it was previously discussed, each GET/PUT is a relatively high latency operation because of multi-stage protocols over a geo-distributed setting. So, the only way to support high request rates per second is, to create multiple execution contexts, namely threads or processes. Each execution context issues one request and completes it. Careful consideration was given to the choice of threads or processes, for the preferred means of creating this concurrency. Threads were preferable because of their relatively low overheads in creation and context switching. However, not only are there scaling limits in using just threads, such as maximum number of open file descriptors per process, there were also other challenges. One such challenge was caused by the choice of our Erasure coding library, called liberasurecode [28].

This library needs a separate encoder instance to be created for each type of coding configuration. However, the functions that create and destroy instances of these encoders, were not thread safe.

Keeping these considerations in mind, a hybrid solution was used. This not only solves the thread-safety issues for the encoding library but also makes scaling the experiments much more tractable. The solution, currently implemented, creates a new process for each key group involved in the experiment. To issue requests at a specified rate for each group, that many threads are created and each thread is responsible for issuing one request. The inter-arrival time, for two consecutive requests by the same thread, is calculated based on the process distribution specified, namely uniform or poisson. For each key group, the encoding parameters are the same. So, in case of CAS, an encoder instance is created once and shared across all the threads for that group. This solves the race condition problem in liberasurecode.

The first thing done by each child process, is to initialize the database with all the keys in that key group. The *init* function performs an INSERT operation for each key in that group to accomplish that. This allows introduction of new keys at any point in the duration of the experiment. Each thread instantiates a new client object for the specified storage protocol(CAS or ABD). This requires that a unique clientId be created for each such thread. The clientId is created by the combination of datacenter_id, group_id (id assigned to each process) and the request_id, so that it's assured to be unique for each concurrent operation.

Following is a high level view of the algorithm deployed at this facility. The algorithm 1 will provide an understanding of how the concurrent threads and processes are managed, to fulfill the requisite arrival rates. As discussed before, each process starts a thread for each concurrent request it needs to support, i.e., number of threads is equal to the arrival rate per key-group per client. After the threads are generated, the parent process needs to make sure that the threads are active only for the duration specified. That's done with the help of a shared state variable, *thread_running* in the algorithm below. The state variable is unset after the stipulated duration has elapsed. All active threads finish their ongoing requests and don't generate any further requests, because the state variable is not set anymore. The parent process can then terminate after all the threads have been cleaned up by calling `join()`.

Algorithm 1: Experiment Facility

```
1  Properties pp = Deserialize(RecvMsg(socket)) ;
2  start_time = pp.start_time ;
3  for group in pp.groups do
4      sleep_until(start_time + group.offset) ;
5      for key_grp in group do
           /* Create a new process and run the following code in child process   */
6          if fork() == 0 then
7              init_db(keys);
8              create_liberasure_instance();
9              thread_running = TRUE ;
               /* Multiply by client distribution to get arrival rate per client  */
10             for i < key_grp.arrival_rate do
11                 create_thread( run_session() );
12             sleep_for(key_grp.duration);
               /* Threads will not create any more requests after this           */
13             thread_running = FALSE;
14             join_threads();
15             delete_liberasure_instance();
16 join_child_process();
```

Algorithm 2: run_session()

```
1  create_clientId(datacenterId, groupId, requestId);
2  create_CAS/ABD_Client(clientId) ;
3  while thread_running do
       /* Call the GET/PUT function based on the workload properties            */
4      create_request();
5      next_request = inter_arrival_time(uniform|poisson);
6      sleep_for(next_request);
```

5.3 Implementation Challenges

An important challenge faced during the implementation was managing the thread-safety of the erasure code library. The design decisions made to solve that problem have been discussed in section 5.2. It's important to understand this limitation of the library, before any design changes are made to the current implementation.

Another open problem, which needs a cleaner solution, is the collection of data from each process and the child process clean-up. Since, each child process terminates asynchronously, based on it's own duration period, the parent process can collect all the return values, by calling *wait()*, only at the end of the experiment. The return values in this case are communicating the average arrival rate for that process. So, for the duration of the experiment, zombie processes accumulate in the process list. This is not a serious concern, because a zombie process don't hold any resources, but a small opportunity for optimization lies here. If the already running processes can be reassigned to new key groups, instead of being terminated, there is a possibility of better resource utilization. The reuse of processes was considered in the current implementation, but that imposed some restrictions on the flexibility of the facility, so it was shelved for now.

Also, each process performs the INSERT operation for all keys in the group during initialization, although the operation is only needed for keys that have not been seen before. But it's much more expensive and complicated to maintain such state information for a possibly very large set of keys. So, a design decision was made to initialize all keys in the group, which is totally acceptable for our experimentation facility.

Chapter 6
Reconfiguration Framework

This work implements one specific reconfiguration protocol, that has been designed to safely reconfigure both erasure coded and replicated data fragments. The reconfiguration framework needs to be supported at all the system components for it to work correctly. Here we will describe the reconfiguration protocol used in this work and the role of each component in its implementation.

6.1 Control Flow for a reconfiguration

The Controller plays a central role in the implementation of this framework. The reconfiguration is always initiated by the Controller. Thus, the Controller needs to make a decision on when to reconfigure the system and for which key groups. We will discuss in more details about how that happens in section 6.2.

When the Controller decides to perform the reconfiguration for a key, it looks up the metadata of the key to find its current configuration. It then sends a *reconfig_query* message to all the servers in that configuration. This is supposed to indicate to all the servers that the reconfiguration has been started and they should block any subsequent requests of the clients for that key. If storage protocol in the current configuration is CAS, then it also sends a *reconfig_finalize* message to a quorum. After these messages, the servers respond back with the latest timestamp, that they have stored in their memory, for that key. They also send the data fragments corresponding to that timestamp and the Controller then reconstructs the value for that key.

This value is then written to the servers in new configuration using *write_config* message. The servers respond back with an acknowledgment of the successful write. The Controller then updates the local metadata of the key with the new configuration. Subsequently, it sends a *finish_reconfig* message, along with the new placement information,

to all the blocked servers. The servers then process the pending requests in accordance with protocol and provide the new configuration to the requests which cannot be serviced by them. The protocol stipulates that blocked requests which have a timestamp smaller than or equal to the timestamp chosen by the reconfiguration, can be serviced by the servers. All other blocked requests are sent the *operation_fail* message, along with new configuration with which they should retry the operation.

6.2 Reconfiguration library at the Controller

The controller's role as a pivotal entity, in managing the reconfiguration, has already been established in the discussions till now. However, to perform it's aforementioned responsibilities, the controller has to perform a series of operations as part of its initialization.

6.2.1 Initialization

At startup, the Controller reads some setup files and saves the network information of all the active Servers and Client Proxies. The Controller then reads the workload information provided by the user, which contains a time series of workload properties, and stores it as an array of *WorkloadConfig* struct. For each key group, the controller needs to decide what configurations should be chosen. The current implementation offsets this part of the decision making to a service called *CostBenefitAnalysis*. The Controller traverses the input array and makes a call to this service for every *WorkloadConfig* struct in the array. The Controller provides it the set of key groups for which the properties changed and their respective workload properties. The Controller expects placement information for each key group as an output from this service - which it will enforce for the specified duration. The service utilizes an optimizer to find the optimal system configuration for each key group and subsequently, applies certain heuristics to make a decision on the final placement configurations.

After the Controller gathers the Placement information of each key group for that timestamp, it stores the information as part of the struct *Properties*, along with other relevant information. It then parses the next entry in the input array.

After the construction of *Properties* struct is complete, it contains the storage configuration for all key groups, for the entire duration of the experiment. It also contains the network information about all the active Data Servers. Once, the initialization is

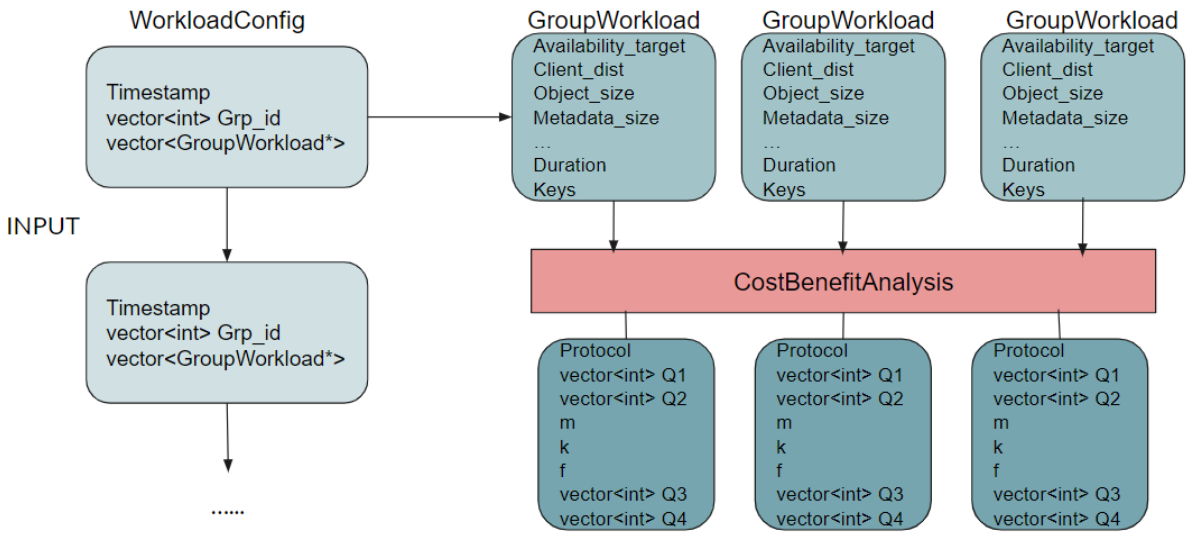

Figure 6.1. Parsing the input *WorkloadConfig* struct

complete, the Controller records the current timestamp and stores it in the aforementioned struct as *start_time*. The struct is then serialized and sent to all the active Client Proxies. This marks the beginning of the experiment.

6.2.2 Implementation of the protocol

The *start_time* included in the *Properties* struct is used as reference time for all the operations in the system. The timestamp value, for each group, is added to this reference time and the absolute time is calculated. The timestamp value of the first group in the *Properties* struct, will decide the actual start of the experiment. So, at requisite time offsets from the start, the Controller evaluates the placement configuration for each key. If the new placement for a key is different from its current placement, the reconfiguration protocol is initiated. At the same time, the Controller makes sure that the local mapping of keys to their configuration information is updated in a coherent manner. The algorithm 3 describes the high level details of the Controller implementation. When the controller decides to initiate the reconfiguration for a key, it calls a function similar to `start_reconfig()`. This function implements the reconfiguration protocol which has been described in section 6.1. The protocol assumes that reconfigurations for the same key are issued sequentially, and under this assumption, provides a safe way to transition the system to the new configuration.

Algorithm 3: Controller Main Logic

```
   /* Initialize the struct after reading input Workload and calling
      CostBenefitAnalysis API to populate the placement information          */
 1 Init Properties;
 2 Properties.start_time = system_time_now();
   /* Send the struct to all the Client Proxies                              */
 3 SendMsg(Serialize(Properties));
 4 for group in Properties.groups do
 5     sleep_until(Properties.start_time + group.timestamp);
 6     for grp_cfg in group do
 7         for key in grp_cfg.keys do
 8             if placement_info(key) != NULL and old_placement !=
                 new_placement then
 9                 create_thread( start_reconfig() );
10             else
                   /* If the key placement info not found or if the placement didn't
                      change, then no reconfiguration required                */
11                 update_placement_info(key);
12         join_threads();
```

6.3 Reconfiguration library at the Client Proxy

The Client Proxy has minimal information about an ongoing reconfiguration. But the reconfiguration protocol still needs some implementation support at the Client Proxies, because it affects them in 2 primary ways. First, the clients need to be cognizant, that their requests may experience additional delays occasionally because of reconfiguration protocol. This directly impacts how the request timeouts at the Clients should be decided. The current implementation doesn't use any client request timeouts, and defers this to a future implementation. Second, the Clients may receive an *operation_fail* message from the servers, along with some configuration information. The Client Proxy needs to parse this configuration information on the receipt of such a message. The current implementation, retrieves the new configuration from the message and then retries the operation with that configuration. Also, all subsequent requests for that key, will be made to servers based on the new configuration. This response handling has been described in algorithm 4. It presents a high level view of the implementation at Client, with the help of an example - `get_timestamp()`. As seen in line 12, at the receipt of *operation_fail*, the client updates its placement information and retries the operation.

Algorithm 4: **get_timestamp** implementation

```
1  while status == FAIL and retries– do
2      for i in placement.Q1 do
3          Init promise;
4          Init future = promise.get_future();
5          create_thread(_get_timestamp, datacenters[i], promise);
6      while future do
7          data = future.get();
8          if data.status == OK then
9              process timestamp received;
10         else
11             if data.status == operation_fail then
12                 Update_placement_information(data);
               /* Operation should fail and retry should occur                    */
13             Garbage Collect;
14             status = FAIL;
15             continue;
16     max_timestamp();
```

Algorithm 5: **_get_timestamp** implementation

```
1 socket_connect();
2 sendMsg();
3 data = deserialize(RcvMsg()) ;
4 promise.set_value(data);
5 close_socket();
```

6.4 Reconfiguration library at the Server

The implementation at the Data Server follows a simple design. Each incoming connection is accepted and based on the request type, it is serviced by the appropriate function in the server libraries. However, to support the reconfiguration protocol, there are three additional requirements which need to be incorporated in the implementation.

6.4.1 Blocking the Client Requests

Each server, on receipt of *reconfig_query* for a key, should block all subsequent client operations until the reconfiguration completes. In the current implementation, this is achieved by creating a shared global lock, to avoid race conditions with other ongoing

operations at the server. A state variable is set, to indicate that the client operations should be blocked. Since, each request creates a separate thread at the server, one condition variable is used per key to block all the threads for that key. When the reconfiguration finishes for a key, the condition variable for that key is signalled and all the pending requests are serviced, based on the protocol.

6.4.2 State Information per Reconfiguration

To fulfil the terms of the reconfiguration, the servers now need additional information for each ongoing reconfiguration. This is required, because the response of the server is no longer dependent just on the request type but also on the current state of the system. Also, the servers need to selectively service client requests, after the reconfiguration is finished and that requires some state information as well. The servers mainly store the following additional information.

- Highest timestamp chosen in the reconfiguration.
- New configuration provided by the Controller, after reconfiguration is finished.
- Number of waiting threads, to help in cleanup of state information after reconfiguration is done.
- Other implementation specific constructs, such as locks and conditional variables.

6.4.3 Additional Request Messages

This is a straightforward requirement, as the reconfiguration protocol requires servers to respond appropriately to the requests generated by the Controller. So, the servers need to have an implementation for these request messages. The implementation also needs to ensure that all pending requests which are serviced, after the reconfiguration is finished, are provided the data fragments from the old configuration state.

6.5 Implementation Challenges

The reconfiguration framework presented many interesting implementation challenges. Since, reconfiguration implies a change in the encoding parameters, working with the *liberasurecode* library became particularly challenging. An example of this can be seen at the Controller. As part of the reconfiguration protocol, the Controller has to decode the

value using old encoding parameters and subsequently encode that value, using encoding parameters from the new configuration. This involved dealing with multiple encoding instances at the same time. So, to mitigate this issue, the controller maintains a list of active encoding instances and creates instances as needed. The instances are only freed at the end of the experiment, to avoid any race conditions while free is called.

Even the server side implementation of the reconfiguration framework had some interesting challenges. This includes management & clean-up of per key reconfiguration state, blocking and un-blocking of requests per key and ensuring isolation of old data values from the data values written as part of the new configuration.

There are opportunities for improvement in the implementation as well. One of the them being the sequence in which reconfiguration for different keys is issued. With careful consideration, the possibility of configuring multiple keys together can also be explored. Some other opportunities include removing the global lock at the server, so that some concurrency can be exploited without violating safety of the protocol. One other area that requires additional thought is the selection of client request timeouts, in the presence of reconfiguration protocol.

Chapter 7
Conclusion

The ease of deployment and performance guarantees provided by the cloud services have made them attractive options for a variety of web applications. This has motivated a closer analysis of the possible optimizations in the system design of geo-distributed cloud services. However, this is a non-trivial task, as these services need to operate efficiently in a variety of workload settings. This work implements an experimentation facility which can be used to analyze system performance and explore optimal design strategies. The facility provides a hybrid key-value store, that supports both replication and erasure coding based storage protocols. The facility is also capable of generating client request workloads, based on attributes specified by the user. It allows the workload to be dynamic and change its properties at arbitrary time intervals. Finally, this work acknowledges system reconfiguration as an important tool to deal with dynamic input workloads. Thus, it implements a reconfiguration protocol, that safely migrates the system to a new configuration, when the input workload changes its characteristics. The use of erasure coding and periodic reconfiguration of the system to the most optimal storage policy, will provide savings in both cost and latency. The framework implemented in this work will play an important role in exploring these opportunities.

Bibliography

[1] Corbett, J. C., J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford (2013) "Spanner: Google's Globally Distributed Database," *ACM Trans. Comput. Syst.*, **31**(3).
URL https://doi.org/10.1145/2491245

[2] Hewitt, E. (2010) *Cassandra: The Definitive Guide*, 1st ed., O'Reilly Media, Inc.

[3] Fitzpatrick, B. (2004) "Distributed Caching with Memcached," *Linux J.*, **2004**(124), p. 5.

[4] "Redis Home Page," .
URL https://redis.io/

[5] Weatherspoon, H. and J. Kubiatowicz (2002) "Erasure Coding Vs. Replication: A Quantitative Comparison," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, Springer-Verlag, Berlin, Heidelberg, p. 328–338.

[6] Bailis, P. and A. Ghodsi (2013) "Eventual Consistency Today: Limitations, Extensions, and Beyond," *Commun. ACM*, **56**(5), p. 55–63.
URL https://doi.org/10.1145/2447976.2447992

[7] Rashmi, K. V., M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran (2016) "EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, USENIX Association, Savannah, GA, pp. 401–417.
URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/rashmi

[8] Burihabwa, D., P. Felber, H. Mercier, and V. Schiavoni (2016) "A Performance Evaluation of Erasure Coding Libraries for Cloud-Based Data Stores," in *Distributed Applications and Interoperable Systems* (M. Jelasity and E. Kalyvianaki, eds.), Springer International Publishing, Cham, pp. 160–173.

[9] "Intel storage acceleration library (open source version)," .
URL https://goo.gl/zkVl4N

[10] Zhang, H., M. Dong, and H. Chen (2016) "Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, USENIX Association, Santa Clara, CA, pp. 167–180.
URL https://www.usenix.org/conference/fast16/technical-sessions/presentation/zhang-heng

[11] Attiya, H., A. Bar-Noy, and D. Dolev (1995) "Sharing Memory Robustly in Message-Passing Systems," *J. ACM*, **42**(1), p. 124–142.
URL https://doi.org/10.1145/200836.200869

[12] Lynch, N. A. (1996) *Distributed Algorithms*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[13] Cadambe, V. R., N. Lynch, M. Mèdard, and P. Musial (2017) "A coded shared atomic memory algorithm for message passing architectures," *Distributed Computing*, **30**(1), pp. 49–73.
URL https://doi.org/10.1007/s00446-016-0275-x

[14] Sharma, C. (2018) *Design And Implementation Of A Cost-Effective Linearizable Geo-Distributed Key-Value Store Combining Replication And Erasure Coding*, Ph.D. thesis.
URL https://etda.libraries.psu.edu/catalog/15616cks5338

[15] Alfares, N. (2020) *Adapting Key-Value Storage Systems to Minimize Cost in the Public Cloud*, Ph.D. thesis.
URL https://etda.libraries.psu.edu/catalog/17549nna5040

[16] Reed, I. S. and G. Solomon (1960) "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, **8**(2), pp. 300–304.

[17] Sathiamoorthy, M., M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur (2013) "XORing elephants," *Proceedings of the VLDB Endowment*, **6**(5), p. 325–336.
URL http://dx.doi.org/10.14778/2535573.2488339

[18] Huang, C., H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin (2012) "Erasure Coding in Windows Azure Storage," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, USENIX, Boston, MA, pp. 15–26.
URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang

[19] Dimakis, A. G., P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran (2010) "Network Coding for Distributed Storage Systems," *IEEE Transactions on Information Theory*, **56**(9), pp. 4539–4551.

[20] Annamalai, M., K. Ravichandran, H. Srinivas, I. Zinkovsky, L. Pan, T. Savor, D. Nagle, and M. Stumm (2018) "Sharding the shards: managing datastore locality at scale with Akkio," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 445–460.

[21] Ardekani, M. S. and D. B. Terry (2014) "A Self-Configurable Geo-Replicated Cloud Storage System," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, USENIX Association, Broomfield, CO, pp. 367–381.
URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/ardekani

[22] Wu, Z., M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha (2013) "SPANStore: Cost-Effective Geo-Replicated Storage Spanning Multiple Cloud Services," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, Association for Computing Machinery, New York, NY, USA, p. 292–308.
URL https://doi.org/10.1145/2517349.2522730

[23] Sharov, A., A. Shraer, A. Merchant, and M. Stokely (2015) "Take me to your leader! Online Optimization of Distributed Storage Configurations," in *Proceedings of the 41st International Conference on Very Large Data Bases*, pp. 1490–1501.

[24] "CPP LRU Cache," .
URL https://github.com/lamerman/cpp-lru-cache

[25] "RocksDB: A Persistent Key-Value Store for Flash and RAM Storage," .
URL https://github.com/facebook/rocksdb

[26] "High Volume Incoming Connections: Linux Kernel Tuning for High Performance Networking Series," .
URL https://levelup.gitconnected.com/linux-kernel-tuning-for-high-performance-networking-high-volume-incoming-connections-196e863d458a

[27] "Protocol Buffers: A language-neutral, platform-neutral extensible mechanism for serializing structured data," .
URL https://developers.google.com/protocol-buffers/docs/proto

[28] "liberasurecode: An Erasure Code API library written in C with pluggable Erasure Code backends," .
URL https://github.com/openstack/liberasurecode