The Pennsylvania State University
The Graduate School
College of Engineering

DESIGN AND IMPLEMENTATION OF A COST-EFFECTIVE LINEARIZABLE GEO-DISTRIBUTED KEY-VALUE STORE COMBINING REPLICATION AND ERASURE CODING

A Thesis in
Computer Science and Engineering
by
Chetan Sharma

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2018

The thesis of Chetan Sharma was reviewed and approved* by the following:

Bhuvan Urgaonkar
Associate Professor of Computer Science and Engineering
Thesis Advisor

Viveck Cadambe
Assistant Professor of Electrical Engineering

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

*Signatures are on file in the Graduate School.

Abstract

In this thesis, we present the variants of two representative non-blocking linearizable protocols tailored to geo-distributed environments and discuss the trade-offs between them in terms of cost and latency.

This thesis also presents a cost-effective linearizable geo-distributed key-value store called *GeoStore*. *GeoStore* takes as an input the properties of the workload being considered and predicts a suitable protocol and its configuration by modeling it as a quadratic optimization problem. It also shows that for certain workloads, erasure coding based linearizable protocols perform significantly better in terms of cost as compared to replication based protocol.

As a future direction, this thesis also implements the optimistic version of these protocols which can further optimize them for certain workloads.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor Dr. Bhuvan Urgaonkar first for giving me a chance to work with him, solving my queries even during late nights and motivating me whenever I got stuck. I deeply enjoyed my discussions with him and in fact, that was the best part of my whole journey.

I am thankful to Dr. Viveck Cadambe for all the mind-boggling discussions he conducted to find a suitable research problem for me and assist me in shaping abstract ideas into a well defined problem.

Apart from them, I would like to acknowledge my parents for being supportive in every decision I have taken so far.

I would like to mention my friend Nader Alfares, who has helped me consistently in performing experiments.

None of it would have been possible without these all.

Chapter 1
Introduction

With tremendous increment in applications with global users like social media, ticket reservations, online shared document, banking etc billions of varied size objects are stored for users all across the globe. It becomes deemed necessary for cloud providers to expand their data stores to different geographical regions so as to provide lower latencies to users across the globe.

However, in many such applications like banking, shared document editors when different users concurrently try to access shared data, a notion of "thread-safety" is required in order to maintain consistency. Hence, atomicity (linearizability) [1] becomes very crucial. An operation can be defined as *linearizable* if it always appears to take effect instantaneously at some point between its invocation and its response. Due to the huge implication and significance of it, there are a series of algorithms implemented using replication and erasure coding for linearizability to improve upon either the latency or the storage cost.

Most of the current literature is based on the assumption that latencies are uniform across the servers and network or that the storage is equally priced across the servers. However, in reality, most of these assumptions are void. Latency between data centers and the cost of cloud services vary by region even within one service provider as shown in Table 1.1 and Table 1.2.

This complicates the problem and makes it very difficult for one solution to cover all the required aspects of geo-distributed settings. Hence we state our hypothesis as:

For a given workload, selecting the right combination of protocol, its configuration, and its placement can lead to considerable cost savings in geo-distributed settings.

Location	Tokyo	Seoul	Mumb.	Frankf.	Ireland	Virginia	S. America	Cali.	Oregon
USD (per GB)	0.9	0.8	0.86	0.2	0.2	0.2	0.16	0.2	0.2

Table 1.1. Data transfer cost outside AWS data center

Location	Tokyo	Seoul	Mumb.	Frankf.	Ireland	Virginia	S. America	Cali.	Oregon
Tokyo	0	43	136	247	237	165	293	128	109
Seoul	43	0	169	280	261	186	319	156	144
Mumb.	143	171	0	131	138	194	326	267	236
Frankf.	241	269	177	0	25	90	214	148	162
Ireland	224	251	133	24	0	76	189	138	140
Virginia	161	186	194	92	79	0	126	71	83
S. America	305	332	324	213	188	123	0	203	191
Cali.	122	150	253	149	139	68	197	0	24
Oregon	108	137	229	163	139	82	186	25	0

Table 1.2. Inter-region TCP ping latencies in AWS (in ms) between different regionsprogram

In this thesis, we will formalize the above problem as a quadratic optimization program and build a prototype called *GeoStore*. *GeoStore* is a geo-distributed key-value store which tailors the algorithm, its configurations, and its placement across different data centers for a provided workload to optimize the cost. We will deploy *GeoStore* to nine data centers provided by AWS and verify the correctness of our optimization and hypothesis.

The rest of this thesis is organized as follows. In Chapter 2, we discussed erasure coding and two non-blocking atomic protocols which will be used in our prototype. In Chapter 3, we presented few examples to show the trade-off between different protocols in a geo-distributed environment. In Chapter 4 our optimization is presented followed by prototype design and implementation in Chapter 5. Finally, Chapter 6 and 7 are about about experimental evaluation, results, and discussions.

Chapter 2
Background and Related Work

This chapter briefly covers erasure coding (EC) and introduces two representative non-blocking linearizable protocols which were used in building the prototype.

2.1 Erasure Coding

A (k, m) erasure code encodes k data units and generates additional m number of parity codes such that data can be recovered as long as there are any k available codes among the (k + m) codes [2]. EC has been widely used for fault tolerance in storage systems and is known as generalization of replication [3–7]. It provides similar redundancy as replication with lower storage cost. [8]

2.2 Two Representative Protocols

2.2.1 ABD

A paper by Attiya et al. [9] presented a non-blocking, atomic, single-writer, multi-reader register algorithm (ABD) implemented for unreliable asynchronous systems. It achieves fault tolerance with the help of replication and supports read (GET) and write (PUT) requests. In addition to the above vanilla protocol, we restricted the number of server failures in the system to F. In order to understand the protocol better, we should be aware of the sequence of steps made by the GET and PUT requests.

2.2.1.1 Client Side PUT Protocol

Similar to GET put involves two round trips across the servers as shown in Figure 2.1. The first trip is to fetch the timestamp keeping data transfer cost negligible. However, the second trip is to put values into write quorum servers incurring data transfer to write quorum servers.

PUT(*key*, *value*):

get-timestamp: Send *get-timestamp* message to the read quorum and wait for all of them to respond with *timestamp*. On receiving response, select the largest timestamp and let its integer component be t.

put-value: Create a new timestamp *timestamp-new* as ($t + 1$, *client-id*) where *client-id* is an unique id associated with each client. Send *put-value* message with (*timestamp-new*, *value*) to the write quorum and wait for all of them to respond.

Figure 2.1. Client Side PUT Protocol for ABD

2.2.1.2 Client Side GET Protocol

The GET request takes as an input the key you would like to fetch the value for. It involves two round-trips across the servers shown in Figure 2.2. The first trip is to fetch the recent timestamp and corresponding value. This trip incurs data transfer as we fetch data from read quorum. The second trip is to send the new value to the write quorum servers. It also requires the client to transfer data into write quorum servers.

GET(*key*):

get-timestamp-value: Send *get-timestamp-value* message to the read quorum and wait for all of them to respond with (*tag*, *value*). On receiving response, select the largest timestamp and its corresponding value (*max-tag*, *max-value*).

put-value: Send *put-value* message with (*max-tag*, *max-value*) to the write quorum and wait for all of them to respond.

Figure 2.2. Client Side GET Protocol for ABD

2.2.1.3 Server Side Protocol

The server receives messages from clients and responds accordingly as shown in Figure 2.3.

Server Side Protocol:
Initial State: Stores the (*timestamp*, *value*) pair for each key.

On receiving get-timestamp: Respond back with locally available *timestamp*.

On receiving get-timestamp-value: Respond back with locally available *timestamp* and *value*.

On receiving put-value: If received timestamp is greater than locally available timestamp, store new (*timestamp*, *value*). Respond back with acknowledgement in any case.

Figure 2.3. Server Side Protocol for ABD

In case of a failure or timeout, these protocols will send the request to all the remaining servers in N and wait until they get a response from the required number of servers.

2.2.1.4 Constraints

For the correctness of the protocol, there are certain constraints on read (R) and and write (W) quorum servers . We have included one additional constraint for the number of server failures system can tolerate(F). We will denote total number of servers i.e. (W R) as N.

- $R + W > N$
- N - $F \geq$ R, W

2.2.2 CAS

A paper by Cadambe et al. presented another protocol named Coded Atomic Storage (CAS) [7]. It can be defined as an atomic, lock-free shared-memory emulation algorithm which uses erasure coding. We will deviate slightly from the generic algorithm for CAS and define our own variant of CAS to make it more

suitable for geo-distributed settings. Our variant will involve four different quorums $Q1$, $Q2$, $Q3$ and $Q4$ where ($Q1 \quad Q2 \quad Q3 \quad Q4$) will be a total number of servers where data can be placed (N). The relationship between the quorums will be discussed in further subsections. We will use K to denote the dimension of the code.

2.2.2.1 Client Side PUT Protocol

The PUT request, shown in Figure 2.4, takes as an input the key and value which need to be updated. It involves three round trips. The first one is to fetch the timestamp, followed by sending the code and last one is to update 'fin' tag. Data transfer is negligible in the first and third trip.

PUT(*key*, *value*):

get-timestamp: Send *get-timestamp* message to the *Q1* quorum servers and wait for all of them to respond with *timestamp*. On receiving response, select the largest timestamp, lets call it's integer component as t. Also, encode the *value* into Q2 number of smaller chunks (c_i,...,c_{Q2}).

put-code: Create a new timestamp *timestamp-new* as ($t + 1$, *server-id*. Send *put-code* message with (*timestamp-new*, c_i, 'null') to *ith* server of Q2 quorum. Wait for all of them to respond.

put-fin: Send *put-fin* message to Q3 quorum with *timestamp-new* and wait for all of them to respond.

Figure 2.4. Client Side PUT Protocol for CAS

2.2.2.2 Client Side GET Protocol

The GET request, shown in Figure 2.5, takes the key as an input. It involves two round trips, the first one is to fetch the timestamp and second is to fetch the data. Data transfer will be involved in the second trip although its negligible in the first one.

GET(*key*):

get-timestamp: Send *get-timestamp* message to the *Q1* quorum servers and wait for all of them to respond with *timestamp*. On receiving response, select the largest timestamp, lets call it *timestamp-max*.

get-code: Send *get-code* message with *timestamp-max* to the Q4 quorum and wait for all of them to respond with their corresponding *codes*. Decode the *value* from collected *codes*.

Figure 2.5. Client Side GET Protocol for CAS

2.2.2.3 Server Side Protocol

Like ABD, server receives messages from clients and responds accordingly, shown in Figure 2.6.

Server Side Protocol for CAS:
Initial State: For each key, it stores (*timestamp*, V, *label*) where V (c_i, 'null') and *timestamp* ('fin', 'null'). c_i is the coded element for server i.

On receiving get-timestamp: Respond with locally known most recent timestamp for the key with the 'fin' tag.

On receiving get-code: If the (*timestamp*, c_i, *) exits for the key where * could be 'null' or 'fin' then send this code else add (*timestamp*, 'null', 'fin') and send 'null'.

On receiving put-code: If the *timestamp* doesn't exits for the key then update (*timestamp*, c_i, 'null') else ignore. Send acknowledgement in any case.

On receiving put-fin: If (*timestamp*, c_i, 'null') exists for the key then update it to (*timestamp*, c_i, 'fin') else insert (*timestamp*, 'null', 'fin'). Send acknowledgement in any case.

Figure 2.6. Server Side Protocol for CAS

In case of a failure or timeout, these protocols will send the request to all the remaining servers in N and wait until they get a response from the required number of servers.

2.2.2.4 Constraints

Similar to ABD, these are the constraints necessary for the correctness of the protocol and for required fault tolerance.

- $Q1 + Q2 > N$
- $Q1 + Q4 > N$
- $Q2 + Q4 \geq N + K$
- $Q4 \geq K$
- $N - F \geq Q1, Q2, Q3, Q4$

2.3 Related Work

Work on developing lock-free algorithms for linearizability is an active area of research since many years. Traditionally these algorithms used replication to bring about redundancy in the system. In recent past, new series of algorithms for linearizability are emerging which use erasure coding for redundancy [7, 10–17] . Most of the current state of art algorithms with erasure coding on linearizability focus on optimizing latencies or storage cost with no consideration for the workload properties and challenges faced with geo-distributed settings. SPANStore [18] tried to optimize the cost in geo-distributed settings but they restricted themselves to replication and use just one blocking protocol for linearizability. EC-Cache [2] is another work which tries to provide better latencies and lower IOPS by using different configurations of erasure coding. However, they did not deal with concurrent objects and restricted themselves to just one data center as shown in Figure 2.7. Giza [19] worked with erasure coding in geo-replicated settings but they are predominantly focused on archival workloads.

Figure 2.7 shows the current state of the art and what we are trying to achieve.

2.4 Linearizabilty Testing with Knossos

Testing linearizability is an NP-Complete problem. Although there has been a lot of related work in this field for developing new algorithms [20–23], we did not come

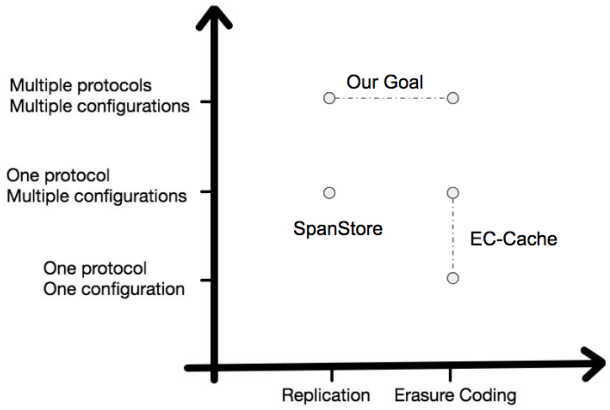

Figure 2.7. Current State of the Art

across any ideal tools to test linearizability of an implemented system in the current literature. Therefore, we took the freedom to chose Knossos [24] for our testing.

Every operation in Knossos is a logical transition from one state to another. Its operations include *invoke* if a request is invoked, *ok* if a request returned successfully, *failure* if a request failed and *info* when the outcome is not clear, helpful in case of delays or timeouts. Knossos takes an interleaved sequence of operations made by a set of clients as input and tries to show that it is NOT linearizable. Since it is a computationally heavy process, it is not feasible to check long traces of operations to test for linearizability. Therefore, we will test with a smaller subset of traces of our workload with just a single key to increase the level of concurrency in our system.

Chapter 3
Trade-o□s in Geo-distributed Settings

In this chapter, we will analyze the trade-offs across the two protocols - ABD, CAS and within each of these protocols.

3.1 Trade-o□ across di□erent protocols

To understand the trade-offs between protocols better, lets consider a simple setup of 5 data centers uniformly distributed across the globe. We make the following assumptions about the setup -

- latency between any two nearest data centers is 1 second
- data transfer cost between any two data centers is constant i.e. 1 dollar/byte
- object size is constant and equals to *ObjSize*
- fault tolerance of at most 1 is tolerated.

3.1.1 ABD PUT Protocol

We used 3-way replication with uniform distribution of objects as shown in Figure 3.1. We have considered the read quorum and write quorum to be of size 2, hence utmost one failure is tolerated.

- *Average Latency* for PUT Request (two round trips):

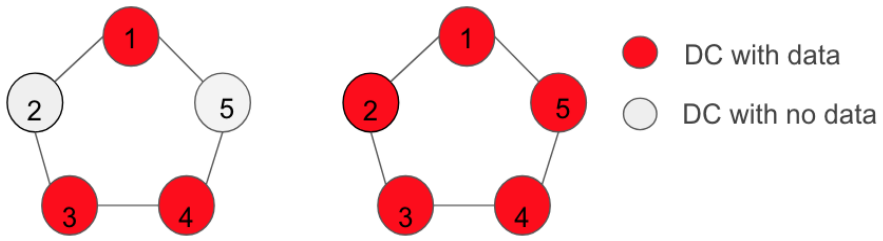

Figure 3.1. (Left) Placement policy for ABD Toy Example and (Right) Placement policy for CAS Toy Example

$$2 \quad \{P\{\text{value in DC}\} \quad (\text{latency of nearest DC with data}) + P\{\text{value not in DC}\} \quad max(\text{latency of two nearest DC with data})\} \tag{3.1}$$

$$2 \quad \{\frac{3}{5} \quad (\frac{2}{3} \quad 1 + \frac{1}{3} \quad 2) + \frac{2}{5} \quad 1\} = \frac{102}{30} \tag{3.2}$$

- *Average data transfer cost* for PUT Request

$$P\{\text{value in DC}\} \quad ObjSize + P\{\text{value not in DC}\} \quad 2 \quad ObjSize \tag{3.3}$$

$$\frac{3}{5} \quad 1 \quad ObjSize + \frac{2}{5} \quad 2 \quad ObjSize = \frac{7}{5} \quad ObjSize \tag{3.4}$$

- *Total storage cost*

$$3 \quad ObjSize \tag{3.5}$$

3.1.2 CAS PUT Protocol

For CAS, the configuration we selected was Q1 = Q3 = 3, Q2 = Q4 = 4 and K = 3 with data placed in all 5 data centers as shown in Figure 3.1. As the maximum quorum size is 4 i.e. we would only need any 4 servers utmost out of 5, hence it can tolerate utmost one failure.

- *Average Latency* for PUT Request (three round trips) can be defined as:

$$max(\text{latency of two nearest DC with data})[\textit{for first round trip}] + max(\text{latency of three nearest DC with data})[\textit{for second round trip}] + max(\text{latency of two nearest DC with data})[\textit{for third round trip}] \tag{3.6}$$

	ABD	CAS
Latency	3.4 (better)	4
Data transfer Cost	1.4 * *ObjSize*	*ObjSize* (better)
Storage Cost	3 * *ObjSize*	1.67**ObjSize* (better)

Table 3.1. Trade-offs between ABD and CAS PUT Request for our toy example

$$1 + 2 + 1 = 4 \tag{3.7}$$

- *Average data transfer cost* for PUT request

$$P\{\text{value in DC}\} \quad \frac{3}{3} \quad ObjSize + P\{\text{value not in DC}\} \quad \frac{4}{3} \quad ObjSize \tag{3.8}$$

$$ObjSize \tag{3.9}$$

- *Total storage cost*

$$\frac{5}{3} \quad ObjSize \tag{3.10}$$

As specified in Table 3.1 while ABD provides better latencies, CAS has lower storage and data transfer costs.

3.2 Trade-o within a protocol

In this section, we demonstrate the trade-offs within a protocol by varying the value of K (i.e. 2 or 3). Just like the previous example, lets consider a simple setup of 5 data centers uniformly distributed across the globe. We make the following assumptions about this setup -

- latency between any two nearest data centers is 1 second
- data transfer cost between any two data centers is constant i.e. 1 dollar/byte
- object size is constant and equals to *ObjSize*
- fault tolerance of at most 1 is tolerated.

3.2.1 CAS GET Protocol (K=3)

For CAS, the configuration we selected was Q1 = Q3 = 3, Q2 = Q4 = 4 and K = 3 with data placed in all 5 data centers. As maximum quorum size is 4, it can

	CAS(K=3)	CAS(K=2)
Latency	3	2 (better)
Data transfer Cost	*ObjSize*	*ObjSize*
Storage Cost	1.67**ObjSize* (better)	2.5**ObjSize*

Table 3.2. Trade-offs between CAS(K=3) and CAS(K=2) GET Request for our toy example

tolerate utmost one failure.

- *Average Latency* for GET Request (two round trips):

$$max(\text{latency of two nearest DC with data})[\textit{for first round trip}]+ \\ max(\text{latency of three nearest DC with data})[\textit{for second round trip}] \quad (3.11)$$

$$1 + 2 = 3 \quad (3.12)$$

- *Average data transfer cost* for GET Request:

$$P\{\text{value in DC}\} \quad \frac{3}{3} \quad ObjSize + P\{\text{value not in DC}\} \quad \frac{4}{3} \quad ObjSize \quad (3.13)$$

$$ObjSize \quad (3.14)$$

- *Total storage cost*

$$\frac{5}{3} \quad ObjSize \quad (3.15)$$

3.2.2 CAS GET Protocol (K=2)

For CAS, the configuration we selected was Q1 = Q3 = Q4 = 3, Q2 = 4 and K = 2 with data placed in all 5 data centers. As the maximum quorum size is 4 out of 5 where data is placed, it can tolerate utmost one failure.

- *Average Latency* for GET Request (two round trips):

$$max(\text{latency of two nearest DC with data})[\textit{for first round trip}]+ \\ max(\text{latency of two nearest DC with data})[\textit{for second round trip}] \quad (3.16)$$

$$1 + 1 = 2 \quad (3.17)$$

- *Average data transfer cost* for GET Request:

$$P\{\text{value in DC}\} \quad \frac{2}{2} \quad ObjSize + P\{\text{value not in DC}\} \quad \frac{3}{2} \quad ObjSize \tag{3.18}$$

$$ObjSize \tag{3.19}$$

- *Total storage cost*

$$\frac{5}{2} \quad ObjSize \tag{3.20}$$

As shown in Table 3.2, while the latency is higher for CAS with K=3 it provides efficient storage as compared to CAS with K=2.

Example shown in Table 3.2 and **??**3.1, shows that even in much simpler uniform settings different protocols may suit the requirements of different workloads. However, in reality, the problem is much more complicated. Most of these assumptions we made for our toy examples are void. Latency between data centers and the cost of cloud services vary by region even within one service provider as shown in Table 1.1 and 1.2. Hence, there is a need for an optimizer to find the relevant protocol for the given workload.

Chapter 4
Modeling and Optimization

In this chapter, we will discuss about our implementation of an optimizer, its inputs, constraints and objectives. We will also introduce a heuristic based algorithm used for optimization.

4.1 Quradratic Optimization Modeling

We formalized our optimizer as a quadratic optimization program with integer components. We divided our workload into smaller sub-workloads where each sub-workload is a group of keys with identical properties. The reason to consider sub-workloads instead of individual keys is to support scalability as the number of keys stored may be prohibitively large. We will call these sub-workloads as *groups*.

4.1.1 Inputs

The inputs to the program are:

- number of data centers: N
- latency between any pair of data centers i and j denoted as l_{ij}
- For each group G:
  - arrival rate: λ_G
  - number of objects: n_G
  - read/write ratio: RW_G

  - for each DC i, fraction of traffic originating at i denoted as f_{iG}
  - average object size O_G
  - SLO target for PUT and GET requests
  - target data center's failure tolerance denoted as F

- storage cost (per byte per unit time) denoted as C_{store}
- network transfer cost between any two DCs i and j (per byte) denoted as $C_{Net_{ij}}$

4.1.2 Decision Variables

For each *group*, we will following have decision variables:

- boolean variable E_G denoting which protocol will be used - ABD or CAS.
- protocol configuration which includes number of servers (m_G) and dimension of code (k_G) (Note: k_G=1 is a special case for replication)
- various quorum sizes which are dependent on the protocol. $Q1_G$, $Q2_G$, $Q3_G$, $Q4_G$ values are used for CAS protocol. ABD, on the other hand, uses $Q1_G$ and $Q2_G$ as its read and write quorum values respectively.
- boolean variable $iQ1_{ij_G}$ where $iQ1_{ij_G} = 1$ iff DC j is in $Q1$ quorum for requests originating at DC i. Similar values exist for the $Q2$, $Q3$, $Q4$ quorums.
- $iQ12_{i_G} = 1$ if DC i is in some $Q1_G$ or $Q2_G$ for the *group*
  - $iQ12_{i_G} = 1$ if there exist atleast one j s.t. $iQ1_{ij_G} = 1$ or $iQ2_{ij_G} = 1$

4.1.3 Constraints and Objective

Constraints are protocol specific as specified in sections 2.2.1.4 and 2.2.2.4. Most of the constraints were straightforward and easy to capture. However, quorum related constraints were more complicated to model. Hence, we broke them down into simpler constraints and captured them individually.

For example, $Q1_G + Q2_G > m_G$ can be specified as:

- For a group G, summation of the sizes of $Q1$ and $Q2$ quorums should be greater than m_G, the total number of servers
  - $i \quad N$ where $f_{iG} > 0$; $\sum_{j=1}^{N} \text{iQ1}_{ij_G} + \sum_{j=1}^{N} \text{iQ2}_{ij_G} > m_G$
- For a group G, union of $Q1$ and $Q2$ should be equal to m_G

  $\sum_{i=1}^{N} \text{iQ12}_{i_G} = m_G$

Our ultimate objective is to minimize:

Data transfer cost for GETs + Data transfer cost for PUTs + Storage Cost

4.2 Heuristic Based Modeling

As seen in section 4.1, optimizing the cost with mentioned constraints is called as quadratic optimization. It is computationally heavy and consumes significant resources as also mentioned by SPANStore [18]. Currently, our optimizer is under progress and hence as an initial step, we opted to use a brute force approach where we generate all possible combinations of protocols, their configurations and possible placements. The quorums for a client are chosen to be the nearest data center servers where the data is placed. This in fact might be a good approach for relatively lower SLOs where every client has less flexibility to choose quorums and is bound to talk to its nearest neighbours.

Algorithm 1 provides the pseudo code for our heuristic based approach. The variable *input* is a composite variable which includes workload properties, total data center list and latencies between the DCs. Initially, the method *allPossibleProtoclCombos* generates a list of all the possible protocols and their configurations (total number of servers, dimension of code, size of different quorums etc) which can meet the required fault tolerance. After that, for each combination of protocol and its configuration we generate all the possible placements using the *choose* method. Finally, for each placement we check if the required SLOs can be met for all the data centers from where incoming traffic is generated using the method *check* and pick the one with minimal cost.

Algorithm 1 Optimizer Algorithm

```
1: procedure OPTIMIZE(input)
2:     init optPlacement ← list()
3:     init optProtocol
4:     init optConfig
5:     init optTotalCost ← MAX
6:     possibleProtocolCombos ← allPossibleProtocolCombos(input)
7:     for protocol, config in possibleProtocolCombos do:
8:         placements ← choose(input.allDatacentersList, config.totalServers)
9:         for placement in placements do:
10:            placementPossible ← TRUE
11:            for dc in input.clientDatacentersList do:
12:                latencyMetSLO ← check(input, protocol, config, placement, dc)
13:                if not latencyMetSLO then
14:                    placementPossible ← FALSE
15:                    break
16:            tempTotalCost ← calculateCost(input, config, placement, dc)
17:            if tempTotalCost < optTotalCost and placementPossible then
18:                optPlacement ← placement
19:                optProtocol ← protocol
20:                optConfig ← config
21:    return optProtocol, optConfig, optPlacement
```

Chapter 5
Design and Implementation

In order to test our hypothesis we implemented our idea by deploying a prototype using Amazon AWS with data centers spanning across the globe. We have implemented the prototype of *GeoStore* in python owing to its simplicity in coding and its vast set of supportive libraries. Also, in a geo-distributed setting, the latency between distant servers is of major concern and language is just an communication interface between data servers and client library. Therefore, the choice of language will not affect our results.

5.1 Prototype Design and Implementation

GeoStore consists of four major components -

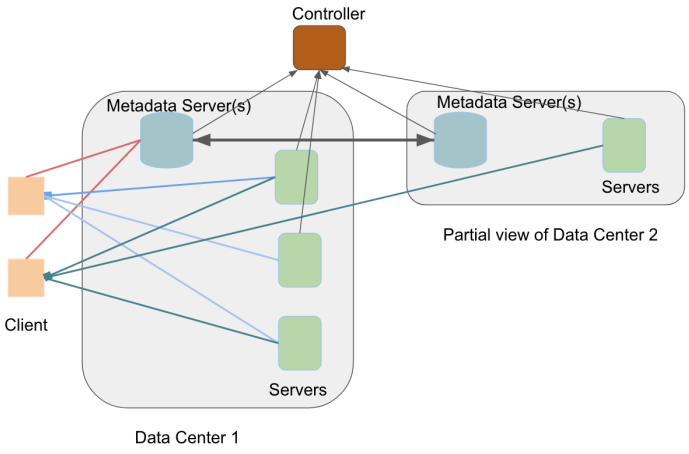

Figure 5.1. Overview of *GeoStore*

1. *Client library* implements the relevant protocols
2. *Controller* implements the optimization library and identifies the relevant protocol, its configuration and placement policy
3. *Metadata server* stores the metadata of each key
4. *Data server* implements the server side Get and Put protocols

In this section, we will look closely into each of these components and what role they play.

5.1.0.1 Client Library

The Client library implements GET, PUT and INSERT functionality for each of the protocols mentioned in section 3. The client initiates the process by sending the following estimated workload properties to the optimizer -

- Object size distribution
- Read-Write ratio
- Arrival rate distribution
- Popularity distribution
- Incoming traffic distribution across all regions
- The required Service Level Objective (SLO) for GET and PUT request
- Fault tolerance required

The optimizer in turn uses the above information to decipher the most relevant protocol, its configuration and its placement policies.

After the initial bootstrapping is complete, the client is ready to send requests to the data servers. It uses the GET and PUT requests to read from and write unto the data servers respectively. The INSERT operation is used to write a key to the data servers for the very first time. We assumed that in our workload only one client will perform an insert for a particular key. During an INSERT operation, the client first makes a PUT request to the data servers based on the protocol and

placement policy it received from the optimizer. It then inserts this information into the nearest metadata server to aid in procuring information for the subsequent GET or PUT requests on that key. The Client also possesses an in-memory cache where it stores the metadata for the most popular keys so as to avoid repeated trips to the metadata server.

5.1.0.2 Controller

The Controller is the 'brain of the whole unit' i.e it takes decisions based on inputs from the client and also performs heartbeat checks to ensure the liveliness of the system. The Controller mainly comprises of the Optimizer library explained in Section 3. During the bootstrap phase, the Client sends the predicted workload properties to the controller. The Controller divides this workload into smaller sub-group(s) wherein each of the sub-groups has the following properties:

- estimated number of objects
- average Read-Write ratio
- average fraction of traffic originating at each data center
- the required Service Level Objective (SLO) for GET and PUT request
- the required fault tolerance

This is where the Optimizer kicks in. Optimization is applied to each sub-group based on its workload properties, latency between DCs and the cost of data transfer originating at each DC. This helps in minimizing the cost to achieve the ideal placement policy ensuring low latency. After the optimization is complete, the Controller returns the suitable consistency protocol, its configuration and the placement policy for each sub-group to the client.

Further, the Controller also holds the responsibility to collect logs from all the metadata servers, data servers and aggregate them at a single place so as to ease in debugging. It simultaneously performs heartbeat checks on these servers to monitor them on a timely basis.

5.1.0.3 Metadata server

As the name suggests, the Metadata server stores the metadata information for a given key. This Metadata information includes -

- placement policy for the key i.e. a list of all the data centers and data server in each of these DCs where the key will be stored
- protocol chosen for the key
- configuration of the protocol

As mentioned in 4.0.1.1, the client contacts its nearest metadata server for information on a key. It is possible that this particular metadata server does not store that key. In such cases, this server takes the onus to contact the rest of the metadata servers and waits for any one of them to handover this information which it then passes along to the client.

The Metadata servers also periodically ping each other and gather the latency times between themselves. This information is then updated to the controller which uses it as the latency between data centers in different regions.

5.1.0.4 Data Server

Data servers take up the server side role for a given protocol. When the client contacts them through the GET, PUT and INSERT requests, the data servers perform the necessary operations and reply back to the client. On a PUT request, the data server writes the data to the storage (both in-memory and persistent storage). We have implemented our own version of LRU dict for cache [25] and used Rocksdb [26] for persistent storage.

Chapter 6
Experimental evaluation

In this chapter we will briefly discuss about our methodology, experimental setup and our findings.

6.1 Methodology

We ran the prototype with two different protocols and their preferred configuration settings, namely:

- ABD
- CAS with Replication (K=1)
- CAS with Erasure Coding

We picked ABD and CAS with Replication because collating the results for these two can help in identifying the trade-offs across these protocols. On the other hand, CAS with Erasure Coding provides more insights for improvement in configuration within a protocol and can help in understanding the significance of erasure coding. We ran the optimizer with the options mentioned above to obtain the ideal placements and configurations. Finally, the data transfer cost, storage cost incurred and latencies observed in each scenario were compared for a given synthetic workload.

6.2 Experimental Setup

For this experiment, we chose t2.xlarge EC2 instances of Amazon AWS as data severs and positioned them in nine different regions spanning across the globe.

The regions include Tokyo, Seoul, Mumbai, Frankfurt, Ireland, California, South America, California and Oregon. An EBS General Purpose SSD was used for storage purposes. We have considered a workload with constant arrival rate of 200 requests/second with a GET:PUT ratio of 9:1 (read skewed). One million objects each of size 10 KB were used to create an incoming traffic uniformly distributed across all the clients operating in the chosen 9 regions. The aim was to create a system which would meet SLOs of maximum 500 ms (tail latencies) and average 300 ms for GET and maximum of 750 ms and average of 450 ms for PUT with a failure tolerance of two data centers.

6.2.1 Prototype Testing

There are two major components in testing the protocol, one is linearizability testing and the other is failure testing.

For linearizability we tested our prototype using varying inputs of concurrent requests made for a single key on multiple data centers and tested the merged output traces with Knossos. We also did stress testing on our university cluster where latencies between servers are close to zero with multiple clients from different servers concurrently writing on a single key.

For failure testing, we intentionally stopped all servers in few data centers and verified that the system runs upto the required fault tolerance. In our current design, we initially send the request to only required number of servers spanning across data centers, but in case of a failure or timeout on any of these servers, the subsequent requests are sent to all the other servers within the workload's placement policy.

6.3 Observation and Results

As mentioned in Section 6.1, we evaluated our results based on performance improvements across the protocols i.e *ABD*, *CAS* and improvements within *CAS* protocol by varying code dimensions in erasure coding.

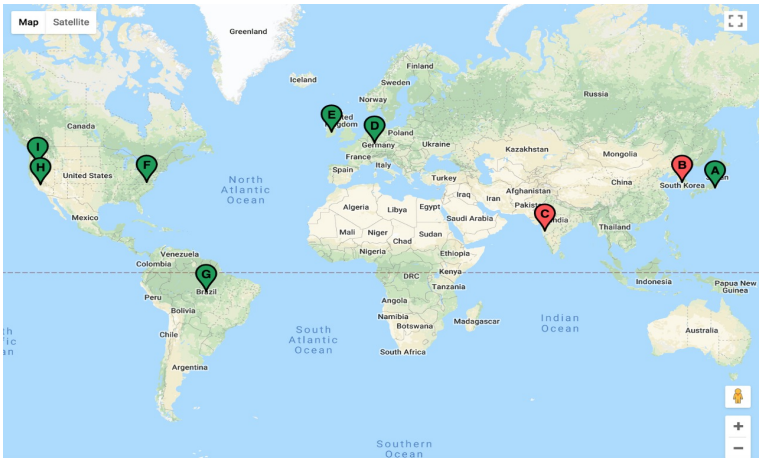

Figure 6.1. Placement policy by Optimizer for ABD and CAS (K = 1). The servers where placement is possible are indicated in green

Figure 6.2. Placement policy by Optimizer for CAS. The servers where placement is possible are indicated in green

6.3.1 Comparison across the protocols

The protocols *ABD* and *CAS with replication (K=1)* are compared in this section. From Figure 6.5 we can deduce that for a given set of workload properties, one protocol can clearly dominate the other with respect to optimal cost.

As the total data size is relatively small, storage cost doesn't play a significant role in this comparison. When it comes to data transfer costs, CAS with replication incurrs approximately half the cost of ABD implying that it involves lower amounts of data transfer. We can attribute this price difference to the additional write-back call that ABD invokes on every read, thereby increasing the overall data transfer

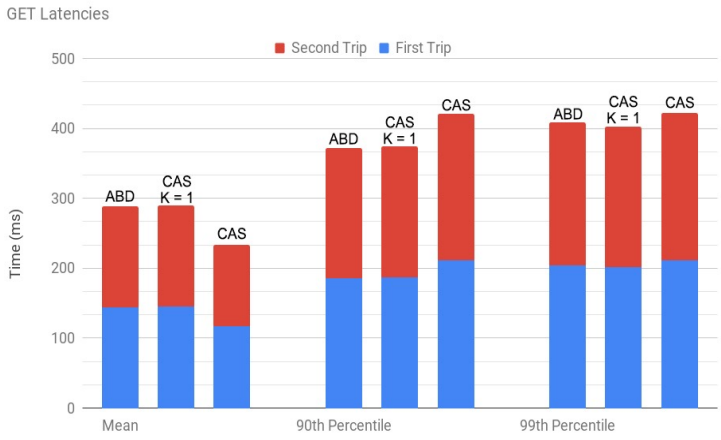

Figure 6.3. Get Latencies comparison for different Protocols

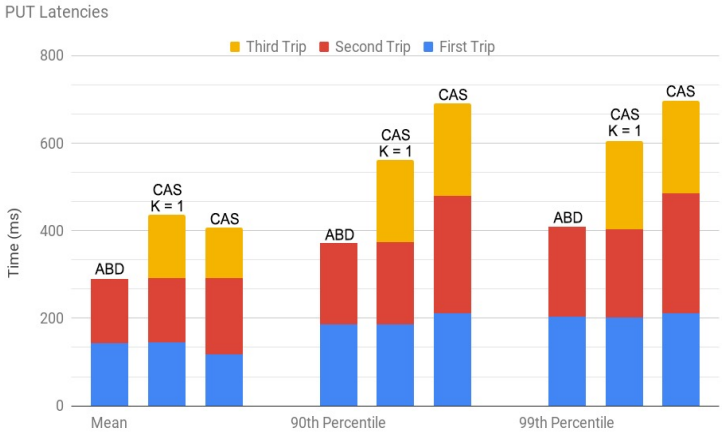

Figure 6.4. Put Latencies comparison for different Protocols

cost.

In terms of latencies, ABD PUT request incurrs lower latencies than its counterpart as the former involves two round trips across the data center whereas CAS involves three as shown in Figure 6.3 and 6.4. As expected, the GET latencies are comparable since both of them have a similar placement policy as shown in Fig 6.1

6.3.2 Comparison within a protocol

In this scenario, we compared the optimal placement policy shown in Figure 6.1 and 6.2 as per the optimizer by keeping the protocol fixed to CAS and varying the options as *replication* or *erasure coding*. Figure 6.3, 6.4 and 6.5 show the cost

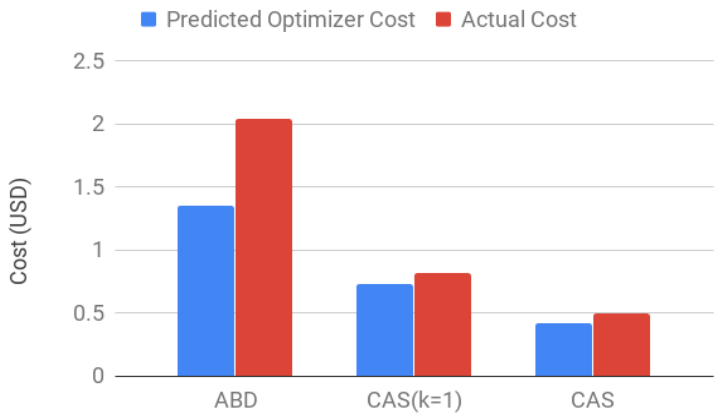

Figure 6.5. Cost comparison for different Protocols

incurred and latency observed for the different configurations. Again, as mentioned above, the total data size is relatively small forcing the storage cost to play a negligible role here. As the cost incurred for CAS with erasure coding is lesser than that of replication, we can safely conclude that CAS with erasure coding observes relatively less data transfer overhead as was expected.

On the contrary, latency observed for CAS with erasure coding is on the higher side for tail latencies (90th percentile) due to the overhead of fetching data from more number of servers.

Chapter 7
Conclusions and Future Work

After taking a look at the experiments detailed in the previous chapters and understanding the trade-offs in each method, we came up with certain conclusions and probable future directions for the work. This chapter also briefly states the impact of our work and how this could be a game-changer in solving the linearizability problems in geo-distributed settings to achieve optimal cost.

In Chapter 6, we have proved that our hypothesis holds true. Even a small change in selecting the characteristics of the workload and its corresponding protocol, placement policy might lead to a significant reduction in the operational costs of a geo-distributed system. To be specific, we have seen that in certain workloads, our optimizer reduces the per request cost by almost 50% with erasure coding than when compared to replication based protocols. Not only that, we can observe that using erasure coding brings in a certain amount of flexibility in choosing the configuration.

7.1 Future Work

Our current work revolves around the vanilla ABD and CAS protocols. These protocols can be further optimized for geo-distributed settings based on the workload properties. For example, in the case of read-heavy or low concurrency, we can optimize the GET calls for ABD and CAS as described in the following sections.

7.1.1 Optimistic ABD

7.1.1.1 Client Side GET Protocol

In the optimistic case, ABD GET protocol can avoid the second trip to the data center all together thus saving upon the latencies and write-back data transfer cost as shown in Fig 7.1.

GET(*key*):

get-timestamp-value: Send *get-timestamp-value* message to the read quorum and wait for all of them to respond with (*tag*, *value*). On receiving the response, select the largest timestamp and its corresponding value (*max-tag*, *tag-value*). If number of received messages with (*max-tag*, *tag-value*) are greater than write quorum, then return the *tag-value* and EXIT, else go to next step.

put-value: Send *put-value* message with (*max-tag*, *tag-value*) to the write quorum and wait for all of them to respond.

Figure 7.1. Client Side GET Protocol for Optimistic ABD

.

7.1.2 Optimistic CAS

Similar to ABD, CAS can also be altered to save the data transfer cost in geo-distributed settings.

7.1.2.1 Client Side GET Protocol

In contrast to the regular CAS server-side operations, optimistic CAS requests only K servers to provide a value for the given key so as to reduce the data transfer cost.

7.1.2.2 Server Side Protocol

Apart from the operations mentioned in section 2.2.2.3, the server will support one additional operation as shown in Figure 7.3.

GET(*key*):

get-timestamp: Send *get-timestamp* message to $Q1$ quorum servers and wait for all of them to respond with a *timestamp*. On receiving the response, select the largest timestamp and call it *timestamp-max*.

get-code: Send *get-code* message with *timestamp-max* to K servers and *get-code-mock* with *timestamp-max* to the $Q4$ - K servers. Wait for all K to respond with their corresponding *codes* and $Q4$ - K for acknowledgement. Decode the *value* from collected *codes*.

Figure 7.2. Client Side GET Protocol for Optimistic CAS

Server Side Protocol for Optimistic CAS:

get-code-mock: On receiving *get-code-mock* message with a *timestamp*, if (*timestamp*, *value*, 'null') exists then update it to (*timestamp*, *value*, 'fin') else insert (*timestamp*, null, 'fin'). Respond with acknowledgement in any case.

Figure 7.3. Server Side Protocol for Optimistic CAS

We plan to incorporate these changes to the ABD and CAS protocols in *GeoStore* as a part of future work.

Bibliography

[1] Herlihy, M. P. and J. M. Wing (1990) "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.*, **12**(3), pp. 463–492.
URL http://doi.acm.org/10.1145/78969.78972

[2] 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.

[3] Cassuto, Y. (2013) "What Can Coding Theory Do for Storage Systems?" *SIGACT News*, **44**(1), pp. 80–88.
URL http://doi.acm.org/10.1145/2447712.2447734

[4] Datta, A. and F. Oggier (2013) "An Overview of Codes Tailor-made for Better Repairability in Networked Distributed Storage Systems," *SIGACT News*, **44**(1), pp. 89–105.
URL http://doi.acm.org/10.1145/2447712.2447735

[5] Lin, S. and D. J. Costello (2004) *Error Control Coding, Second Edition*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[6] Roth, R. (2006) *Introduction to Coding Theory*, Cambridge University Press, New York, NY, USA.

[7] Cadambe, V. R., N. Lynch, M. Médard, and P. Musial (2014) "A Coded Shared Atomic Memory Algorithm for Message Passing Architectures," in *2014 IEEE 13th International Symposium on Network Computing and Applications*, pp. 253–260.

[8] 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,

London, UK, UK, pp. 328–338.
URL http://dl.acm.org/citation.cfm?id=646334.687814

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

[10] Abd-El-Malek, M., G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie (2005) "Fault-scalable Byzantine Fault-tolerant Services," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, ACM, New York, NY, USA, pp. 59–74.
URL http://doi.acm.org/10.1145/1095810.1095817

[11] Agrawal, A. and P. Jalote (1995) "Coding-based replication schemes for distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, **6**(3), pp. 240–251.

[12] Androulaki, E., C. Cachin, D. Dobre, and M. Vukolic (2014) "Erasure-Coded Byzantine Storage with Separate Metadata," *CoRR*, **abs/1402.4958**, 1402.4958.
URL http://arxiv.org/abs/1402.4958

[13] Cachin, C. and S. Tessaro (2006) "Optimal Resilience for Erasure-Coded Byzantine Distributed Storage," in *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, IEEE Computer Society, Washington, DC, USA, pp. 115–124.
URL http://dx.doi.org/10.1109/DSN.2006.56

[14] Dobre, D., G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolić (2013) "PoWerStore: Proofs of Writing for Efficient and Robust Storage," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, ACM, New York, NY, USA, pp. 285–298.
URL http://doi.acm.org/10.1145/2508859.2516750

[15] Dutta, P., R. Guerraoui, and R. R. Levy (2008) "Optimistic Erasure-Coded Distributed Storage," in *DISC*, pp. 182–196.

[16] Goodson, G. R., J. J. Wylie, G. R. Ganger, and M. K. Reiter (2004) "Efficient Byzantine-tolerant erasure-coded storage," in *International Conference on Dependable Systems and Networks, 2004*, pp. 135–144.

[17] Hendricks, J., G. R. Ganger, and M. K. Reiter (2007) "Low-overhead Byzantine Fault-tolerant Storage," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, ACM, New York, NY, USA, pp. 73–86.
URL http://doi.acm.org/10.1145/1294261.1294269

[18] 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, ACM, New York, NY, USA, pp. 292–308.
URL http://doi.acm.org/10.1145/2517349.2522730

[19] Chen, Y. L., S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips (2017) "Giza: Erasure Coding Objects across Global Data Centers," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, USENIX Association, Santa Clara, CA, pp. 539–551.

[20] Lowe, G. (2017) "Testing for linearizability," *Concurrency and Computation: Practice and Experience*, **29**(4).
URL https://doi.org/10.1002/cpe.3928

[21] Horn, A. and D. Kroening (2015) "Faster Linearizability Checking via P-Compositionality," in *Formal Techniques for Distributed Objects, Components, and Systems* (S. Graf and M. Viswanathan, eds.), Springer International Publishing, Cham, pp. 50–65.

[22] Bouajjani, A., M. Emmi, C. Enea, and J. Hamza (2013) "Verifying Concurrent Programs against Sequential Specifications," in *Programming Languages and Systems* (M. Felleisen and P. Gardner, eds.), Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 290–309.

[23] Farzan, A. and P. Madhusudan (2008) "Monitoring Atomicity in Concurrent Programs," in *Computer Aided Verification* (A. Gupta and S. Malik, eds.), Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 52–65.

[24] jepsen io (2018), "Knossos," https://github.com/jepsen-io/knossos.

[25] amitdev (2018), "LRU Dict," https://github.com/amitdev/lru-dict.

[26] facebook (2018), "RocksDB: A Persistent Key-Value Store for Flash and RAM Storage," https://github.com/facebook/rocksdb.