LEGOStore: A Linearizable Geo-Distributed Store Combining Replication and Erasure Coding

Hamidreza Zare, Viveck R. Cadambe, Bhuvan Urgaonkar,
Chetan Sharma†, Praneet Soni‡, Nader Alfares, and Arif Merchant♢
The Pennsylvania State University, Microsoft†, Apple‡, Google♢

Abstract—We design and implement LEGOStore, an erasure coding (EC) based linearizable data store over geo-distributed public cloud data centers (DCs). For such a data store, the confluence of the following factors opens up opportunities for EC to be latency-competitive with replication: (a) the necessity of communicating with remote DCs to tolerate entire DC failures and implement linearizability; and (b) the emergence of DCs near most large population centers. LEGOStore employs an optimization framework that, for a given object, carefully chooses among replication and EC, as well as among various DC placements to minimize overall costs. To handle workload dynamism, LEGOStore employs a novel agile reconfiguration protocol. Our evaluation using a LEGOStore prototype spanning 9 Google Cloud Platform DCs demonstrates the efficacy of our ideas. We observe cost savings ranging from moderate (5-20%) to significant (60%) over baselines representing the state of the art while meeting tail latency SLOs. Our reconfiguration protocol is able to transition key placements in 3 to 4 inter-DC RTTs ($<$ 1s in our experiments), allowing for agile adaptation to dynamic conditions.

1 Introduction

Consistent geo-distributed key-value stores are crucial building blocks of modern Internet-scale services including databases and web applications. Strong consistency (i.e., linearizability) is especially preferred by users for the ease of development and testing it offers. As a case in point, the hugely popular S3 store from Amazon Web Services (AWS), in essence a key-value store with a GET/PUT interface, was recently re-designed to switch its consistency model from eventual to linearizable [70]. However, because of the inherent lower bound in [13], linearizable key-value stores inevitably incur significant latency and cost overheads compared to weaker consistency models such as causal and eventual consistency. These drawbacks are particularly pronounced in the geo-distributed setting because of the high inter-data center networking costs, and large network latencies (see Table 2). To further exacerbate the problem, dynamic phenomena such as shifts in arrival rates, appearance of clients in new locations far from where data is stored, increase in network delays, etc., can lead to gaps between predicted and actual performance both in terms of costs and tail latencies.

We design LEGOStore, a geo-distributed key-value store (with the familiar GET/PUT or read-/write API), meant for a global user-base. LEGOStore procures its resources from a public cloud provider's fleet of data centers (DCs) like many storage service providers [28, 55, 14]. Since an entire DC may become unavailable [14, 15, 16, 69], LEGOStore employs redundancy across geo-

distributed DCs to operate despite such events. LEGOStore's goal is to *offer tail latency service-level objectives (SLOs) that are predictable and robust in the face of myriad sources of dynamism at a low cost*. For achieving these properties, LEGOStore is is built upon the following three pillars:

1. Erasure Coding (EC) is a generalization of replication that is more storage-efficient than replication for a given fault tolerance. A long line of research has helped establish EC's efficacy *within a DC* or for weaker consistency needs. However, EC's efficacy in the linearizable geo-distributed setting is relatively less well-understood. Two recent works Giza [23] and Pando [66] demonstrate some aspects of EC's promise in the geo-distributed context. In particular, because EC allows fragmenting the data and storing it in a fault-tolerant manner, it leads to smaller storage costs and (more importantly) smaller inter-DC networking costs. However, the smaller-sized fragments inevitably require contacting more DCs and, therefore, are thought to imply higher latencies in a first-order estimation. By comprehensively exploring a wide gamut of workload features and SLOs, and careful design of the data placement and EC parameters through an optimization framework, LEGOStore brings out the full potential of EC in the geo-distributed setting.

2. LEGOStore adapts **non-blocking, leaderless, quorum-based linearizability protocols** for both EC [21] and replication [71, 11]. When used with carefully optimized quorums, these protocols help realize LEGOStore's goal of predictable performance (i.e., meeting latency SLOs with a very high likelihood as long as workload features match their predicted values considered by our optimization). A second important reason is that, when used in conjunction with a well-designed resource autoscaling strategy, the latency resulting from non-blocking protocols depends primarily on its inter-DC latency and data transfer time components. This is in contrast with the leader-based consensus used in Giza and Pando (see Fig. 5). While these works may be able to offer lower latencies for certain workloads, they are susceptible to severe performance degradation under concurrency-induced contention.

3. To offer robust SLOs in the face of dynamism, LEGOStore implements an **agile reconfiguration mechanism**. LEGOStore continually weighs the pros and cons of changing the *configuration*[1] of a key or a group of keys via a cost-benefit analysis rooted in its cost/performance modeling. If prompted by this analysis, it uses a novel reconfiguration protocol to safely switch the configurations of concerned keys without breaking linearizability. We have designed this reconfiguration protocol carefully and specifically to work alongside our GET/PUT protocols to keep its execution time small which, in turn, allows us to limit the performance degradation experienced by user requests issued during a reconfiguration.

	GCP data center location								
	Tokyo	Sydney	Singapore	Frankfurt	London	Virginia	São Paulo	Los Angeles	Oregon
Storage ($/GB.Month)	0.052	0.054	0.044	0.048	0.048	0.044	0.06	0.048	0.04
Virtual machine ($/hour)	0.0261	0.0283	0.0253	0.0262	0.0262	0.0226	0.0310	0.0248	0.0215

Table 1: Storage and VM prices for the 9 GCP data centers that our prototype spans. We use custom VMs with 1 core and 1 GB of RAM from General-purpose machine type family to run the LEGOStore's servers and Standard provisioned space for its storage [31].

Contributions: We design LEGOStore, a cost-effective key value store with predictable tail latency that adapts to dynamism. We develop an optimization framework that, for a group of keys with similar suitable workload features, takes as input these features, and public cloud characteristics (inter-DC latencies and pricing information) and determines configurations that satisfy

[1] By the configuration of a key, we mean the following aspects of its placement: (i) whether EC or replication is being used ; (ii) the EC parameters or replication degree being used; and (iii) the DCs that comprise various quorums.

		User location																	
		Tokyo		Sydney		Singapore		Frankfurt		London		Virginia		São Paulo		Los Angeles		Oregon	
		n/w price ($/GB)	Latency (ms)	n/w price ($/GB)	Latency (ms)	n/w price ($/GB)	Latency (ms)	n/w price ($/GB)	Latency (ms)	n/w price ($/GB)	Latency (ms)	n/w price ($/GB)	Latency (ms)	n/w price ($/GB)	Latency (ms)	n/w price ($/GB)	Latency (ms)	n/w price ($/GB)	Latency (ms)
GCP data center	Tokyo	-	2	0.15	115	0.12	70	0.12	226	0.12	218	0.12	148	0.12	253	0.12	100	0.12	90
	Sydney	0.15	115	-	2	0.15	94	0.15	289	0.15	277	0.15	204	0.15	291	0.15	139	0.15	162
	Singapore	0.09	72	0.15	94	-	2	0.09	202	0.09	203	0.09	214	0.09	319	0.09	165	0.09	166
	Frankfurt	0.08	229	0.15	289	0.08	201	-	2	0.08	15	0.08	89	0.08	202	0.08	153	0.08	139
	London	0.08	222	0.15	280	0.08	204	0.08	15	-	2	0.08	79	0.08	192	0.08	141	0.08	131
	Virginia	0.08	146	0.15	204	0.08	214	0.08	90	0.08	79	-	2	0.08	116	0.08	68	0.08	58
	São Paulo	0.08	252	0.15	292	0.08	317	0.08	202	0.08	192	0.08	117	-	1	0.08	155	0.08	172
	Los Angeles	0.08	101	0.15	139	0.08	180	0.08	153	0.08	142	0.08	67	0.08	155	-	2	0.08	26
	Oregon	0.08	95	0.15	164	0.08	165	0.08	142	0.08	131	0.08	58	0.08	173	0.08	26	-	2

Table 2: Diverse RTTs and network pricing for 9 chosen GCP data centers. The RTTs are measured between VMs placed within the DCs; they would be higher if one of the VMs were external to GCP but not by enough to change the outcome of our optimizer. For the same general recipient location, the outbound network prices are sometimes higher if the recipient is located outside of GCP (egress pricing) but these prices exhibit a similar geographical diversity [30].

SLOs at minimal cost. The configuration choice involves selecting one from a family of linearizability protocols, and the protocol parameters that can include the degree of redundancy, DC placement, quorum sizes, and EC parameters. In this manner it selectively uses EC for its better storage/networking costs when allowed by latency goals, and uses replication otherwise. We develop a safe reconfiguration protocol, and an accompanying heuristic cost/benefit analysis that allows LEGOStore to keep costs under control by dynamically transforming configurations in response to changes in workload features.

We build a prototype LEGOStore system. We carry out extensive evaluations using our optimizer and prototype spanning 9 Google Cloud Platform (GCP) DCs. We get insights from our evaluation to make suitable modeling and protocol design choices. Because of our design, our prototype has close match with the performance predicted by the optimizer. Potential cost savings over baselines range from moderate (5-20%) to significant (up to 60%). The most significant cost savings emerge from carefully avoiding the use of DCs with high outbound network prices. Our work offers a number of general trends and insights relating workload and infrastructure properties to cost-effective realization of linearizability. Some of our findings are perhaps non-intuitive: (i) smaller EC-fragments do not always lead to lower costs (Sec. 5.3.4); (ii) there exists a significant asymmetry between GETs and PUTs in terms of costs, and read-heavy workloads lead to different choices than write-heavy workloads (Sec. 5.3.3); (iii) we find scenarios where the optimizer is able to exploit the lower costs of EC without a latency penalty (Sec. 5.3.5); and (iv) even when a majority of the requests to a key arise at a particular location, the DC near that location may not necessarily be used for this key in our optimizer's solution.

2 Background

Latency and Pricing Diversity in the Public Cloud: The lower bound of [13, 12] implies that *both* GET and PUT operations in LEGOStore necessarily involve inter-DC latencies and data transfers unlike with weaker consistency models. The latencies between users and various DCs of a public cloud provider span a large range. In Table 2 we depict our measurements of round-trip times (RTTs) between pairs of DCs out of a set of 9 Google Cloud Platform (GCP) data centers we use. The smallest RTTs are 15-20 msec while the largest exceed 300 msec; RTTs between nodes within the same DC are 1-2 msec and pale in comparison. Similarly, the prices for storage, computational, and network resources across DCs also exhibit geographical diversity as shown in Tables 1

and 2. This diversity is the most prominent for data transfers—the cheapest per-byte transfer is \$0.08/GB (e.g., London to Tokyo), the costliest is \$0.15/GB (e.g., Sydney to Tokyo). LEGOStore's data placement strategies and protocol design must carefully navigate these sources of diversity to meet latency SLOs at minimum cost.

Our Choice of Consistent Storage Algorithms: Due to the lower bound mentioned above, in leader-based protocols (e.g., Raft [53]), for the geo-distributed scenario of interest to us, one round trip time to the leader is inevitable. Using such leader-based protocols can drive up latency when the workload is distributed over a wide geographical area, and there is no leader node that is sufficiently close to all DCs so as to satisfy the SLO requirements. Such a design can also place excessive load on DCs that are more centrally located. Therefore, as such, we choose algorithms with leaderless, quorum-based protocols in our design and implementation. We describe these protocols (ABD for replication and CAS for EC) next.

The ABD Algorithm:[2] Let N denote the degree of replication (specifically, spanning N separate DCs) being used for the key under discussion. In the ABD algorithm, for a given key, each of the nodes (i.e., DCs) stores a (tag, value) pair, where the tag is a (logical timestamp, client ID) pair. The node replaces this tuple when it receives a value with a higher tag from a client operation. The PUT operation consists of two phases. The first phase, which involves a logical-time query, requires responses from a quorum of q_1 nodes, and the second phase, which involves sending the new (tag, value) pair requires acknowledgements from a quorum of q_2 nodes. The GET operation also consists of two phases. The first phase again involves logical-time queries from a quorum of q_1 nodes and determining the highest of these tags. The second "write-back" phase sends the (tag, value) pair chosen from the first-phase responses to a quorum of q_2 nodes. If $q_1 + q_2 > N$, then ABD guarantees linearizability. If $q_1, q_2 \leq N - f$, then operations terminate so long as the number of node failures is at most f. Note that this is a stronger liveness guarantee as compared to Paxos; ABD circumvents FLP impossibility [29] because it implements a data type (read/write memory) that is weaker than consensus. In fact, this formal liveness property translates into excellent robustness of operation latency (see Sec. 5.4). A formal algorithm description is in Appendix A.

Whereas the above vanilla ABD requires two phases for all its GET operations, a slight enhancement allows some (potentially many) GET operations to complete in only one phase; we refer to such a scenario as an "Optimized GET." A GET operation becomes an Optimized GET if all the servers return the same (tag, value) pair in the first phase. This scenario occurs if there is no concurrent operation, and some previous operation to the same key has propagated the version to the servers in the quorum that responds to the client. Furthermore, to increase the recurrence of Optimized GET, we enhance the vanilla ABD algorithm to do an asynchronous phase after each PUT operation to propagate (tag, value) pair to servers that do not constitute q_2. The Optimized GET mirrors consensus protocols such as Paxos, where operations can complete in one phase in optimistic circumstances.

Erasure Coding: Erasure coding (EC) is a generalization of replication that is attractive for modern storage systems because of its potential cost savings over replication. An (N, K) Reed Solomon Code stores a value over N nodes, with each node storing a codeword symbol of size $1/K$ of the original value, unlike replication where each node stores the entire value. The value can be decoded from *any* K of the N nodes, so the code tolerates up to $f = N - K$ failures. On the other hand, replication duplicates the data $N = f + 1$ times to tolerate f failures. For a fixed value of N, EC leads to a K-fold reduction in storage cost compared to N-way replication for the same fault-tolerance. It also leads to a K-fold reduction in communication cost for PUTs, which can be

[2]The name "ABD" comes from the authors, Attya, Bar-Noy and Dolev [71, 11].

significant because of the inter-DC network transfer pricing. While this suggests that costs reduce with increasing K, we will see that the actual dependence of costs on K in LEGOStore is far more complex (see Sec. 5.3.4).

The use of EC leads to structural changes in non-blocking linearizable protocols [20, 21, 27, 34, 3]. In EC-based protocols, GET operations require $K > 1$ nodes to respond with codeword symbols corresponding to the *same* version of the key. However, due to asynchrony, different nodes may store different versions at a given time. Reconciling the different versions incurs additional communication overheads for EC-based algorithms as compared with ABD.

The CAS Algorithm: We use the *coded atomic storage* (CAS) algorithm[3] of [20, 21], described in Appendix B. In CAS, servers store a list of triples, each consisting of a tag, a codeword symbol, and a label that can be 'pre' or 'fin'. The GET protocol operates in two phases like ABD; however, PUT operates in *three* phases. Similar to ABD, the first phase of PUT acquires the latest tag. The second phase sends an encoded value to servers, and servers store this symbol with a 'pre' label. The third phase propagates the 'fin' label to servers, and servers which receive it update the label for that tag. The three phases of PUT require quorums of q_1, q_2, q_3, resp., responses to complete. Servers respond to queries from GETs/PUTs only with latest tag labeled 'fin' in their lists. A GET operates in two phases, the first phase to acquire the highest tag labeled 'fin' and the second to acquire the chunks for that tag, decode the value and do a write-back. The two phases of GET require responses from quorums of size q_1, q_4, resp. In the write-back phase, CAS only sends a 'fin' label with the tag, unlike ABD which sends the entire value. In fact, this translates to lower communication costs for CAS even if $k = 1$ (i.e., replication) is used as compared to ABD, with the penalty of incurring higher write latency due to the additional phase. This variation between ABD and CAS offers LEGOStore further flexibility in adapting to workloads as demonstrated in Sec. 5.3. Similar to ABD, we also employ an "Optimized GET" for CAS that enables some (potentially many) GET operations to complete in only one phase. This optimized GET is based on a client-side cache (note that the client is different from a user, cf. Sec 3) for the value computed in the second phase of GET. LEGOStore exploits these protocol differences to reduce costs based on whether the workload is read- or write-intensive.

On the server-side protocol, unlike ABD where a server simply replaces a value with a higher tagged value, CAS requires servers to store a history of the codeword symbols corresponding to multiple versions, and then garbage collect (GC) older versions at a later point. In theory, the storage overhead of GC grows with the number of concurrent operations to that specific key. However, in practice, the overhead is negligible for the workloads we study (see Sec. 3.5).

3 LEGOStore System Design

3.1 Interface and Components

LEGOStore is a linearizable key-value store spanning a set $\mathcal{D}$ of D DCs of a public cloud provider.[4] Applications using LEGOStore ("users") link the LEGOStore library that offers them an API comprising the following linearizable operations:

- CREATE(k,v): creates the key `k` using default configuration `c` (we will define a configuration

[3]The algorithm in Appendix B is a modification the algorithm in [20, 21] to allow for flexible quorum sizes, which in turn this exposes more cost-saving opportunities.

[4]The D DCs could even be from multiple cloud providers with minor enhancements to our formulations and prototype implementation.

momentarily) if it doesn't already exist and stores `(k,c)` in the local meta-data server (MDS); returns an error if the key already exists.[5]
- GET(k): returns value for `k` if `k` exists; else returns an error.
- PUT(k,v): sets value of `k` to `v`; returns error if `k` doesn't exist.
- DELETE(k): removes `k`; returns error if `k` doesn't exist.

To service these operations, the library issues RPCs to a LEGOStore "client" within a DC in $\mathcal{D}$. A LEGOStore client implements the client-end of LEGOStore's consistency protocols. A user resident within a DC in $\mathcal{D}$ would be assisted by a client within the same DC. For users outside of $\mathcal{D}$, a natural choice would be a client in the nearest DC. The client assisting a user may change over time (e.g., due to user movement) but only across operations. Since the user-client delay is negligible compared to other RTTs involved in request servicing (recall Table 2), we will ignore it in our modeling.

In order to service a `GET` or a `PUT` request for a key `k`, a client first determines the "configuration" for `k` which consists of the following elements: (i) replication or erasure coding to be used (and, correspondingly, ABD or CAS); (ii) the DCs that comprise relevant quorums; and (iii) the identities of the LEGOStore "proxies" within each of these DCs. Having obtained the configuration, a client issues protocol-specific Remote Procedure Calls (RPCs) to proxies in relevant quorums to service the user request. Each DC's proxy serves as the intermediary between the client and the compute/storage servers that (a) implement the server-end of our consistency protocols and (b) store actual data (replicas for ABD, EC chunks for CAS) along with appropriate tags.

It is worth noting that when using replication for a key, unlike some related systems [45], LEGOStore does not replicate the key across all DCs but only stores the copies selectively among carefully chosen subsets. Additionally, when using replication, it does not shard a given key across DCs but stores copies in their entirety within relevant DCs. Finally, LEGOStore has the following two components for each key: (i) the *optimizer* determines a suitable configuration for the key; see Sec. 3.2; and (ii) the *reconfiguration controller* safely moves a key from its old configuration to its newly determined configuration; see Sec. 3.3. Due to the composability property of linearizability and how we model latency, both of these are logically separate across keys. Therefore, LEGOStore does not have a centralized optimizer or reconfiguration controller and, in fact, the location of the reconfiguration controller can be chosen carefully relative to the members of the old and new configuration quorums to help keep reconfiguration time small.

3.2 Finding Cost-Effective Configurations

We develop an optimization that determines cost-effective configurations assuming perfect knowledge of workload and system properties. Since our protocols operate at a per-key granularity due to the composability of linearizability [35]—notice how the ABD and CAS algorithms in Appendix A and B are described for a generic key—we can decompose our datastore-wide optimization into smaller optimization problems, one per key.[6] So, in the following, we will outline our formulation for a generic key $g \in G$.

Inputs (See Table 3): We assume that LEGOStore spans D geo-distributed DCs numbered $1, ..., D$. We assume that the following five workload properties are available at a per-key granularity: (i)

[5] A default configuration uses the nearest DCs for various quorums in terms of their RTTs from the client.

[6] Although we design and implement our optimizer for individual keys, aggregating keys with similar workload features and considering such a "group" of keys in the optimizer may be useful (perhaps even necessary) for LEGOStore to scale to large content sizes. We leave the identification and evaluation of such groups to future work.

Table 3: Input and decision variables used by LEGOStore's optimization.

Input	Interpretation	Type
D	Number of data centers	integer
$\mathcal{D}$	Set of data centers numbered 1, ..., D	set
l_{ij}	Latency from DC i to DC j (RTT/2)	real
B_{ij}	Bandwidth between DC i and DC j	real
$\mathcal{G}$	Set of keys	set
λ_g	Aggregate request arrival rate for key $g \in \mathcal{G}$	integer
ρ_g	Read-write ratio for g	real [0,1]
α_{ig}	Fraction of requests originating at/near DC i for key g	real
o_g	Average object size, *including* per-phase protocol-specific meta data exchanged between a client and a server	integer
o_m	Average per-phase protocol-specific meta-data exchanged between a client and a server	integer
l_{get}	GET latency SLO	real
l_{put}	PUT latency SLO	real
f	Availability requirement (i.e., number of failed DCs to tolerate)	integer
p_i^s	Storage price (per byte per unit time) for DC $i \in \mathcal{D}$	real
p_{ij}^n	Network transfer price (per byte) location i to location j	real
p_i^v	VM price at DC i (simplifying assumption: all VMs of a single size)	real
θ^v	This quantity multiplied by the request arrival rate at DC i captures the VM capacity required at i	real
Var.	**Interpretation**	**Type**
e_g	Protocol (0 for ABD, 1 for CAS) for key g	boolean
m_g	Length of code (replication factor for ABD)	integer
k_g	Dimension of code (equals 1 for replication)	integer
$q_{i,g}$	Quorum size for i^{th} quorum of key g	integer
v_i	Capacity of VMs at DC i	real
$\mathbf{iq}_g^k$	Indicator for data placement for k^{th} quorum of key g. $iq_{ijg}^k = 1$ iff DC j in k^{th} quorum of clients in/near DC i	boolean matrix

overall request arrival rate; (ii) geographical distribution of requests (specifically, fractions of the overall arrival rate emerging in/near each DC); (iii) fraction of requests that are GET operations; (iv) average object size and meta-data[7] size; (v) GET and PUT latency SLOs. We assume that SLOs are in terms of tail latencies (specifically, we use the 99^{th} percentile latency in our evaluation in Sec. 5). We assume that the availability requirement is expressed via the single parameter f¿ 0 applicable to all keys; LEGOStore must continue servicing requests despite up to f DC failures. The system properties considered in our formulation are: (i) inter-DC latencies and prices for network traffic between clients and servers; (ii) storage; (iii) computational resources in the form of virtual machines (VMs).

Decision Variables: Our decision variables, as described at the bottom of Table 3, help capture all aspects of a valid configuration. These include: (i) whether this key would be served using ABD or CAS, and (ii) which DCs constitute various quorums that the chosen algorithm (2 and 4 quorums, resp., for ABD and CAS) requires.

Optimization: Our optimization tries to minimize the cost of operating key $g \in G$ in the next *epoch* – a period of relative stability in workload features. Our objective for key $g \in G$, which is cost per unit time during the epoch being considered, is expressed as:

[7]Meta-data transferred over the network can have non-negligible cost/latency implications and that is what we explicitly capture. On the other hand, the storage of meta-data contributes relatively negligibly to costs and we do not consider those costs.

$$\boxed{\begin{array}{l}\text{minimize} \quad \left(C_{g,get} + C_{g,put} + C_{g,Storage} + C_{g,VM}\right) \\ \text{subject to (9)—(20) in Appendix C;}\end{array}} \tag{1}$$

The first two components of the objective with put and get in their subscripts denote the networking costs per unit time of PUT and GET operations, resp., for key g while the last two denote costs per unit time spent towards storage and computation, resp.

Constraints: Our constraints express various costs and request latency in terms of protocol-specific number of rounds and data+metadata transferred. We describe these in Appendix C. '

3.3 How to Reconfigure?

LEGOStore uses a reconfiguration protocol that transitions chosen keys from their old configurations to their new configurations without violating linearizability. Unlike our approach, consensus-based protocols such as Raft and Viewstamped Replication [54, 44], implement the key-value store as a log of commands applied sequentially to a replicated state machine. These solutions implement reconfiguration by adding it as a special command to this log. Thus, when a reconfiguration request is issued, the commands that are issued before the reconfiguration request are first applied to the state machine before executing the reconfiguration. To execute the reconfiguration, the leader transfers the state to the new configuration. After the transfer, it resumes handling of client commands that are serialized after the reconfiguration request, but replicating the state machine in the new configuration. While our approach to key-value store implementation does not involve a replicated log[8], it is possible to develop an approach that inherits the essential idea of consensus-based reconfiguration as follows:

(i) On receiving a reconfiguration request, wait for all ongoing operations to complete, and pause all new operations;

(ii) Perform the reconfiguration by transferring state from the old to the new configurations;

(iii) Resume all operations.

Under the reasonable assumption that reconfigurations *of a given key are performed relatively infrequently*,[9] our design goal is to ensure that user performance is not degraded in the common case where the configurations remain static. For this, it is crucial that user operations that are not concurrent with reconfigurations follow the baseline static protocols without requiring additional steps/phases (such as contacting a controller). Our protocol does not assume any special relation between the old and new configurations. It can handle all types transitions, including changing of the replication factor, EC parameters, quorum structure, and the protocol itself.

We wish to keep the number of communication phases as well as the number of operations affected as small as possible. Towards this, LEGOStore's reconfiguration protocol improves upon steps (i)-(iii) above. Reconfigurations are conducted by a controller that reads data from the old configuration and transfers it to the new configuration. On detecting a workload or system change

[8]Rather than a replicated log, we simply have a replicated single read/write variable per key, which is updated on receiving new values.

[9]More precisely, we assume that reconfigurations to a key are separated in time by periods that are much longer (several minutes to hours or even longer) than the time it takes to reconfigure (sub-second to a second, see measurements in Sec. 5.5).

(See Sec. 3.4 for details), the controller immediately performs the reconfiguration without having to wait for all ongoing operations to complete, i.e., without having to perform step (i). This enables LEGOStore to adapt quickly to workload changes. Furthermore, steps (ii),(iii) are conducted jointly through a single round of messaging. In particular, LEGOStore's protocol integrates with the underlying protocols of CAS and ABD and piggybacks the reconfiguration requests from the controller along with the actions that read or transfer the data. LEGOStore's algorithm is provably linearizable (Appendix D), and therefore, preserves the correctness of the overall data store. The algorithm blocks certain concurrent operations and then resumes them on completing the reconfiguration.

Algorithm 1: Reconfig - Controller

Inputs: old_config, new_config
Send `reconfig_query` messages to all servers in `old_config`
if *old_config*.e_g == *1* **then** /* CAS */
    Await responses from $\max(\texttt{old\_config}.N - \texttt{old\_config}.q_3 + 1, \texttt{old\_config}.N - \texttt{old\_config}.q_4 + 1)$ servers
    Let t denote the highest logical timestamp received
    Send `reconfig_get`(t) message to all servers in `old_config`
    Wait for `old_config`.q_4 servers to respond
    Decode the corresponding value v
else /* ABD */
    Await responses from `old_config`.$N - q_2 + 1$ servers
    Let t denote the received pair with the highest logical timestamp
    Let v be the value of pair t
end
if *new_config*.$e_g == 1$ **then** /* CAS */
    Encode the value v to $chunks$ using Reed-Solomon code
    Send `reconfig_write`($t, chunk[i],' fin'$) to $server_i$ for all servers in `new_config`
    Wait for $\max($`new_config`.q_2, `new_config`.$q_3)$ servers to respond
else /* ABD */
    Send `reconfig_write`(t, v) pair to all servers in `new_config`
    Wait for `new_config`.q_2 servers to respond
end
Update global metadata (probably local to the controller) to reflect new configuration;
Send a `finish_reconfig`(t, v, `new_config`) message to all servers in `old_config`.

The reconfiguration protocol is described formally in Algorithms 1 and 2 and depicted visually in Fig. 1. We assume that reconfigurations are applied sequentially by the reconfiguration controller (or simply controller). The controller sends a `reconfig_query` message to all the servers in the old configuration. On receiving this message, the servers pause all the ongoing operations and respond with the latest value if the old configuration is performing ABD, or the highest tag labeled 'fin' if it is performing CAS. The controller waits for a quorum to respond from the old configuration. If the old configuration is performing CAS, then the reconfiguration controller sends a `reconfiguration_get` and decodes the value from the responses. The controller then proceeds to write the $(tag, value)$ pair to the new configuration, performing encoding if the new configuration involves CAS. On completing writing the value to the new configuration, the

Algorithm 2: Reconfig - Server and Client

On receiving a `reconfig_query` `/* from controller */`
  Disable all actions for clients for `old_config`
  if $new\_config.e_g == 1$ **then** `/* CAS */`
    Send the highest tag labeled fin to controller
  else `/* ABD */`
    Send stored (t, v) pair to controller
  end
end
On receiving a `reconfig_get`(t) `/* from controller */`
  if *the server has a locally stored tuple* $(t, chunk, *)$ **then**
    Send $(t, chunk)$ to the controller
  else
    Store $(t, NULL, ' fin')$
    Send an ack
  end
end
On receiving a `finish_reconfig`($t, v,$ `new_config`) `/* from controller */`
  Send an `(operation_fail, new_config)` message to all `get_timestamp` queries.
  Unblock other queries and respond as described per protocol to every operation with a tag less than or equal to t.
  Send an `(operation_fail, new_config)` message to all client operations with tag larger than t.
end
On receiving a `reconfig_write`(t, v) **or `reconfig_write`**$(t, chunk, ' fin')$ **tuple** `/* from controller */`
  Store the tuple
  Send an ack `/* To controller */`
end
On receiving `operation_fail, new_config)` during an operation `/* in client */`
  Restart the operation with `new_config`.
end

controller sends a `finish_reconfig` message to the servers in the old configuration. On receiving these messages, the servers complete all operations with tag less than or equal to $t_{highest}$ and send `operation_fail` messages along with the information of the new configuration to the other pending operations that were paused. On receiving `operation_fail` messages, the clients restart the operation in the new configuration.

3.4 When and What to Reconfigure?

At the heart of our strategy are these questions: (i) is a key configured poorly for its current or upcoming workload? (ii) if yes, should it be stored using a different configuration? While our discussion focuses on workload dynamism, our ideas also apply to changes in system properties such as inter-DC latencies or cloud pricing.

Is a Key Configured Sub-Optimally? Some workload changes can be predicted (e.g., cyclical temporal patterns or domain-specific insights from users) while others can only be determined after they have occurred. Generally speaking, a system such as LEGOStore would employ a combination of predictive and reactive approaches for detecting workload changes [62, 67, 17]. In this paper, we pursue a purely reactive approach. We employ two types of reactive rules that are indicative of a key being configured poorly:

• *SLO violations:* If SLO violations for a key are observed for more than a threshold duration or during a window containing a threshold number of requests, LEGOStore chooses to reconfigure it. Sections 5.5,5.6 show that LEGOStore's reconfiguration occurs within 1 sec; the threshold should be set sufficiently larger than this to avoid harmful oscillatory behavior over-optimizing for transient phenomena. If, in addition to SLO violations, some quorum members are suspected of being slow or having failed, these nodes are removed from consideration when determining the next configuration.

• *Cost sub-optimality:* Alternatively, a key's configuration might meet the SLO but the estimated running cost might exceed the expected cost. We consider such sub-optimality to have occurred if the observed cost exceeds modeled cost by more than a threshold percentage as assessed over a window of a certain duration.[10] Having determined the need for a change, LEGOStore reconfigures the key based on our *cost-benefit analysis* described below.

Should Such a Key be Reconfigured? For the case of SLO violations, LEGOStore will reconfigure as we consider SLO maintenance to be sacrosanct. In the rest of the discussion, we focus on the case of cost sub-optimality. We assume that such a key's workload features can be predicted for the near term. In the absence of such predictability—and this applies more generally to any similar system—LEGOStore's options are limited to using a state-of-the-art latency-oriented optimization (see our baselines ABD Nearest and CAS Nearest in Sec. 5.1) that is likely to be able to meet the SLO; the impact of such a heuristic on cost is examined in Sec. 5.3.

Assuming predictability, LEGOStore computes the new configuration with the updated workload characteristics using the optimization framework from Sec. 3.2. For an illustrative instance of such decision-making, denote this newly computed configuration as c_{new} and the existing configuration as c_{exist}. Let us denote by $\texttt{Cost}(c)$ the per time unit cost incurred when using configuration c. We assume an additional predicted feature T_{new}, the minimum duration for which the predicted workload properties will endure before changing. LEGOStore compares the cost involved in reconfiguring with potential cost savings arising due to it. Reconfiguring a key entails (a) *explicit*

[10] Detailed exploration of the impact of the thresholds on performance is future work.

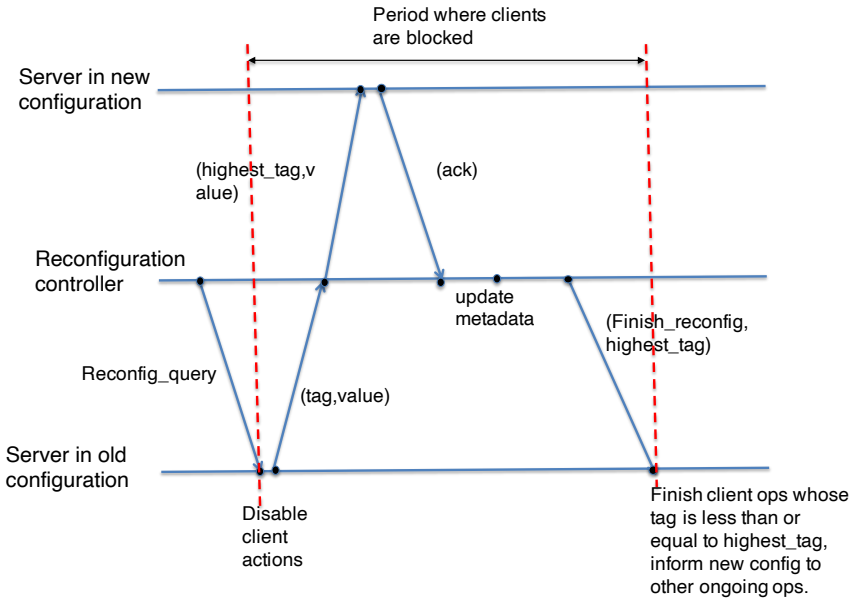


Figure 1: Reconfiguration protocol timeline. The timeline is depicted assuming both configurations are performing replication-based ABD.

costs arising from the addtional data transfer; and (b) *implicit costs* resulting from requests that are slowed down or rejected. An evaluation (see Sec. 5.6 for details) of our reconfiguration protocol suggests that the number of operations experiencing slowdown is small. Therefore, we consider only (a).

LEGOStore's cost-benefit analysis is simple. A reconfiguration to c_{new} is carried out if the potential (minimum) cost savings $T_{new} \cdot \big(\texttt{Cost}(c_{exist}) - \texttt{Cost}(c_{new})\big)$ significantly outweigh the explicit cost of reconfiguration as captured by $\texttt{ReCost}(c_{old}, c_{exist}) \cdot (1+\alpha)$. $\texttt{ReCost}(.,.)$ is the cost of network transfer induced by our reconfiguration and its calculation involves ideas similar to those presented in our optimizer (see Appendix C. The $(1+\alpha)$ multiplier ($\alpha > 0$) serves to capture how aggressive or conservative LEGOStore wishes to be. The efficacy of our heuristics depends on the predictability in workload features and the parameter T_{new}.

Per-Key Controller Placement: Note from Algorithms 1, 2 that the time to complete a reconfiguration (call it T_{re}) and the time during which GET/PUT operations are blocked are largely dictated by the RTTs between the controller and the servers farthest from it in the quorums involved in various phases of the reconfiguration protocol.Fig. 1 highlights this period when both the old and the new configurations use ABD. Careful placement of the controller can lower both of these types of delays. LEGOStore employs a simple heuristic which locates the controller for a key to be reconfigured on a DC to minimize T_{re} (modeling it as a sum of the relevant RTTs).

3.5 Garbage and Faults

To ensure termination of certain concurrent operations, CAS requires each server to store codeword symbols corresponding to older versions, not just the latest version. A garbage collection (GC) procedure is required to remove older values and keep storage costs close to our models. GC does not affect safety, and can only affect termination of concurrent operations to the same key [20]. We use a simple GC heuristic that deletes codeword symbols corresponding to older

versions after a threshold time set larger than the maximum predicted latency of all operations. In our prototype we set this parameter to 5 minutes, an order of magnitude larger than operation latencies (10s-100s of msec). We empirically confirm that this heuristic keeps the storage overheads negligibly small.

LEGOStore continues servicing user requests as long as the number of DC failures does not exceed f. During DC failures, users may experience a degradation in latency or cost depending on the specific DCs that need to be relied upon to compensate for the unavailable DCs. If more than f DCs become unavailable, operations timeout at the client and the user is returned an error. The reconfiguration is used to replace failed nodes with non-failed nodes. If the reconfiguration controller for a key fails in the midst of an ongoing reconfiguration, then all operations to that key can get stalled indefinitely, whereas the functioning of the rest of the data store remains unaffected. LEGOStore's design allows for the reconfiguration controller to be made fault tolerant via state machine replication (e.g., using Paxos), without much change in performance of operations in the common case where there is no concurrent reconfiguration. In our implementation we do not replicate the reconfiguration controller because (i) this is orthogonal to our main interest in this paper and (ii) it does not affect our results in any meaningful way. If the controller fails when no reconfiguration is underway, it can be safely relaunched at an available DC.

4 Salient Implementation Aspects

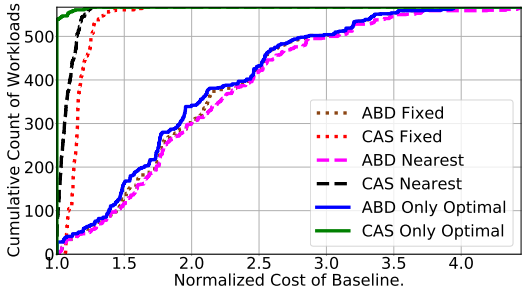
We implement our optimizer using Python3. Since finding the configurations can be done in parallel with each other, we execute up to 48 instances of the optimizer in parallel spanning 8 *e2-standard-4* GCP VMs.[11] We implement our LEGOStore prototype in C++11 using about 5.5K LOC. Our prototype uses the `liberasurecode` library's [43] Reed-Solomon backend for encoding and decoding data in CAS. We refer to LEGOStore's users and clients & servers of our protocols as its *data plane*; we call the per-DC metadata servers, the optimizer and reconfiguration controllers its *control plane*. LEGOStore's source code is available under the Apache license 2.0 at github.com/shahrooz1997/LEGOstore.

Data Plane: A user application (simply user) needs to link the LEGOStore library and ensure it uses LEGOStore's API. The library implements logic for the user to interact with the appropriate (ABD or CAS) client. LEGOStore maintains a pool of *compute nodes* (VMs) (GCP 'custom-1-1024' instance types) that run the client and server protocols and store keys using the RocksDB [1] database engine.

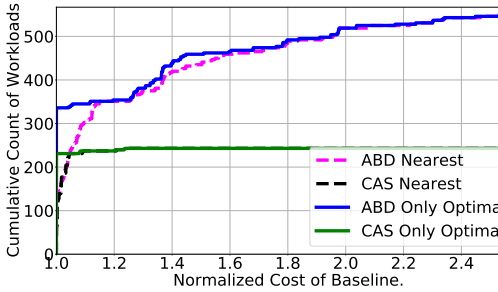
Each DC maintains a meta-data server (MDS) locally that contains configurations for keys that the clients within it have served and that have not yet been deleted. *LEGOStore does not have to have a centralized MDS* with corresponding freedom from concerns related to the failure of such an MDS. When a client does not find a key in the local MDS, it issues queries to other DCs (e.g., in increasing order of RTTs) to learn the key's configuration.

Control Plane: LEGOStore's control plane comprises the two types of elements for each key described in Section 3.1: (i) the optimizer and (ii) the reconfiguration controller. Neither of these components is tied to a particular DC location (e.g., See Sec. 3.4 for discussion regarding controller placement).

[11] Our optimizer takes 610 msec for each workload on average for f=1 and 78 msec for each workload in average for f=2 because with f=2 the search space is smaller.



(a) Latency SLO=1 sec.



(b) Latency SLO=200 msec.

Figure 2: Cumulative count of the normalized cost of our baselines (for our 567 basic workloads) with $f = 1$ and two extreme latency SLOs.

5 Evaluation

We evaluate LEGOStore in terms of its ability to (i) lower costs compared to the state-of-the-art; and (ii) meet latency SLOs. We use Porcupine [10] for verifying the linearizability of a number of execution histories from our prototype.

5.1 Experimental Setup

Prototype Setup: We deploy our LEGOStore prototype across 9 Google Cloud Platform (GCP) DCs with locations, pairwise RTTs, and resource pricing shown in Tables 1 and 2. We locate our users within these data centers as well for the experiments.

Workloads: We employ a custom-built workload[12] generator which emulates a user application with an assumption that it sends requests as per a Poisson process. We explore a large workload space by systematically varying our workload parameters as follows.

- 3 per-key sizes in KB: (i) 1, (ii) 10, and (iii) 100;
- 3 per-key read ratios for high-read (HR), read-write (RW), and high-write (HW) workloads, resp.: (i) 30:1, (ii) 1:1, and (iii) 1:30;
- 3 per-key arrival rates in requests/sec: (i) 50, (ii) 200, and (iii) 500;
- 3 sizes for the overall data: (i) 100 GB, (ii) 1 TB, and (iii) 10 TB.
- 7 different client distributions: (i) Oregon, (ii) Los Angeles, (iii) Tokyo, (iv) Sydney, (v) Los Angeles and Oregon, (vi) Sydney and Singapore, and (vii) Sydney and Tokyo.

This gives us a total of 567 diverse *basic* workloads for a given availability target and latency SLO. Finally, we also vary the availability target (f=1 in this section and f=2 in Appendix F) and latency SLO (in the range 200 ms—1 sec) in our experiments. Additionally, we sometimes use the following customized workloads to explore particular performance related phenomena: (i) a uniform client distribution across all 9 locations; (ii) workloads related to Figs. 4-6; we describe these in the text accompanying these figures. Finally, while our exact metadata size varies slightly between ABD and CAS, we round it up to an overestimated 100 B.

Baselines: We would like to compare LEGOStore's efficacy to the most important state-of-the-art

[12]We do not use an existing workload generator such as YCSB [78] because we wish to explore a wider workload feature space than covered by available tools.

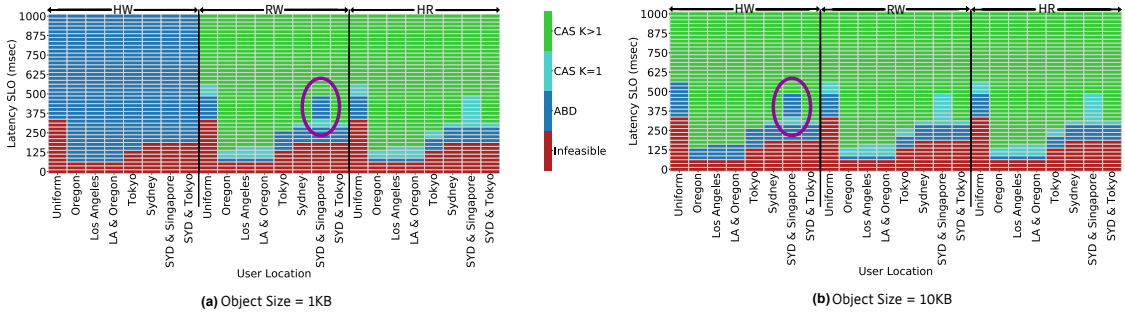


Figure 3: Sensitivity of the optimizer's choice to the latency SLO. We consider 2 object sizes (1KB and 10KB), 8 different client distributions, arrival rate=500 req/sec, and f=1. We consider 3 different read ratios (HW, RW, HR defined in Sec. 5.1).

approaches. To enable such comparison, we construct the following baselines:

• **ABD Fixed and CAS Fixed:** These baselines use only ABD or only CAS, resp. The baseline employs either a fixed degree of replication or a fixed set of CAS parameters. These parameter values (3 for ABD and $(5, 3)$ for CAS) are the ones chosen most frequently by our optimizer across our large set of experiments described in Sec. 5.3. For these fixed parameters, these baselines pick the DCs with the smallest average network prices for their quorums, where the average for a DC i is calculated over the price of transferring data to all user locations. A comparison with these baselines demonstrates that merely knowing the right parameters does not suffice—one must pick the actual DCs making up the quorums judiciously.

• **ABD Nearest and CAS Nearest:** These baselines also use only ABD or only CAS. However, they do not a priori fix the degree of replication or the EC parameters. Instead, we pick the optimized value for each parameter and choose quorums that result in the smallest latencies for the GET/PUT operations ignoring cost concerns. They solve a variant of our optimizer where the objective is latency minimization, expressions involving costs are not considered, and all other constraints are the same. These baselines serve as representatives of existing works (e.g., Volley [6]) that primarily focus on latency reduction. While the baselines are admittedly not as sophisticated as Volley, our results demonstrate that unbridled focus on latency can lead to high costs in the public cloud.

• **ABD Only Optimal and CAS Only Optimal:** These are our most sophisticated baselines meant to represent state-of-the-art approaches. ABD Only Optimal and CAS Only Optimal are representative of works that optimize replication-based systems such as SPANStore [74] or EC-based systems such as Pando [66].

It is instructive to note that our baselines are quite powerful. Our optimizer picks the lower cost feasible solution among ABD Only Optimal and CAS Only Optimal. Yet, we will demonstrate that: (i) these baselines individually perform poorly for many of our workloads; and (ii) the choice of which of the two is better for a particular workload is highly non-trivial.

5.2 Prototype-based Validation

We conduct extensive validation of the efficacy of the latency and cost modeling underlying our optimizer (Appendix F.1).

5.3 Insights from Our Optimization

5.3.1 The Extent and Nature of Cost Savings

In Fig. 2, we express the cost savings our optimizer offers over our baselines via each baseline's *normalized cost* (cost offered by baseline / cost offered by our optimizer). We consider our collection of 567 basic workloads with $f = 1$ and (a) a relaxed SLO of 1 sec and (b) a more stringent SLO of 200 msec. We begin by contrasting the first-order strengths and weaknesses of ABD and CAS as they are understood in conventional wisdom. For the relatively relaxed latency SLO of 1 sec in Fig. 2(a), we find that ABD Only Optimal (and other ABD-based variants) have more than twice the cost of our optimizer for more than 300 (i.e, more than half) of our workloads. On the other hand, CAS Only Optimal closely tracks our optimizer's cost.[13] That is, as widely held, if high latencies are tolerable, EC can save storage and networking costs. Fig. 2(b), with its far more stringent SLO of 200 msec, confirms another aspect of conventional wisdom. CAS is now simply unable to meet the SLO for many workloads (324 out of 567). This is expected given the 3-phase PUT operations and larger quorums to accommodate EC in CAS. (See similar results with $f = 2$ in Fig. 11).

What is surprising, however, is that when we focus on the subset of 243 workloads for which CAS Only Optimal is feasible, it proves to be the cost-effective choice—nearly all workloads for which CAS Only Optimal is feasible have a normalized cost of 1. So, even for stringent SLOs, EC does hold the potential of saving costs. Unlike in Fig. 2(a), however, CAS Fixed or CAS Nearest are nowhere close to being as effective as CAS Only Optimal. That is, while EC can be cost-effective for these workloads, its quorums need to be chosen carefully rather than via simple greedy heuristics.[14] By the same token, replication tends to be the less preferred choice for more relaxed SLOs but, again, there are exceptions.

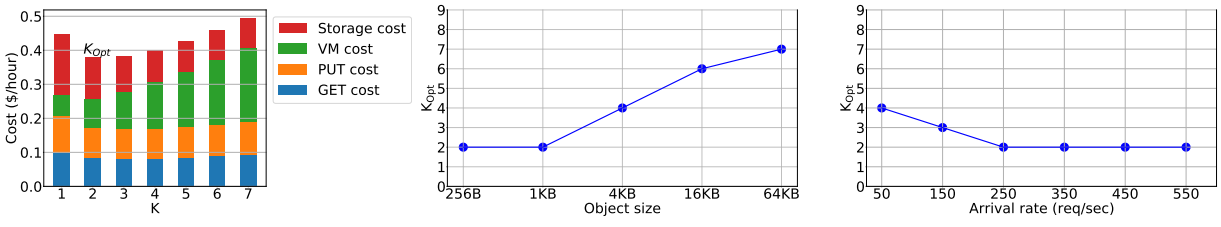
5.3.2 Sensitivity to Latency SLO

We zoom into how the cost-efficacy of ABD vs. CAS depends on the latency SLO by examining the entire range of latencies from 50 msec to 1 sec. Furthermore, we separate out this dependence based on read ratio, availability target and object size. Our selected results are shown in Fig. 3. As expected, as one moves towards more relaxed SLOs, the optimizer's choice tends to shift from ABD to CAS (recall the 3-phase PUTs and larger quorums in CAS). The complexity that the figures bring out is *when* this transition from replication to EC occurs—we see that, depending on workload features, this transition may never occur (e.g., HW in Fig. 3(a)) or may occur at a relatively high latency (e.g., at 575 msec for the uniform user distribution for RW/HR).[15] In particular, more spatially distributed workloads correspond to a tendency to choose replication over EC; for instance, for workloads with uniformly distributed users, SLOs smaller than 300 msec are infeasible due to a natural lower bound implied by the inter-DC latencies. We find that f also has a complex impact on the optimizer's choice; see results with f=2 in Fig. 12, Appendix F.

[13] For this relaxed SLO, even CAS Fixed performs close to the optimizer (recall that this fixed K was chosen as the most frequently suggested value by our optimizer for our basic workloads).

[14] As a further nuance, only 3 workloads out of 243 use CAS with k=1.

[15] The reader might be intrigued by the portions of Fig. 3 highlighted using ovals. Here, our optimizer's choice shifts from ABD to CAS as the latency SLO is relaxed (as expected) but then it shifts back to ABD! We consider this to be a quirk of the heuristics embedded in our optimizer rather than a fundamental property of the optimal solution.



(a) (b) (c)

Figure 4: For CAS-based solutions, cost is non-monotonic in K and K_{opt} has a complex relation with object size and arrival rate. Latency SLO is 1 sec.

5.3.3 Read- vs. Write-Intensive Workloads

One phenomenon that visibly stands out in Fig. 3 is how write-intensive workloads for the relatively small object sizes (HW in Fig. 3(a)) prefer ABD even for the more relaxed SLOs. This preference of ABD over CAS becomes less pronounced when we increase the object size to 10KB in Fig. 3(b). Finally, we also observe that read-intensive/moderate workloads tend to prefer CAS (K=1) over ABD, even when replication is used. To understand this asymmetry, note the following:

• **Reads:** Whereas both ABD and CAS have a "write-back" phase for read operations, ABD's write-back phase carries data, while CAS's only carries metadata, and thereby incurs much lower network cost. Thus, our optimizer tends to prefer CAS for HR workloads.

• **Writes:** For writes, CAS involves 3 phases whereas ABD only requires 2. Since each phase incurs an additional overhead in terms of metadata, the metadata costs for write operations are higher for CAS. Therefore, especially for small object sizes (Fig. 3(a)) and write-heavy workloads, our optimizer will tend to prefer ABD.

Collectively, the results in Fig. 3 convey the significant complexity of choosing between ABD and CAS. Within CAS-friendly workloads, there is further substantial complexity in how the parameter K depends on workload features.

5.3.4 Factors Affecting Optimal Code Dimension K

We illustrate our findings using the representative results in Fig. 4(a)-(c) based on a workload with the following features for which CAS is the cost-effective choice: object size=1KB; datastore size=1TB; arrival rate=200 req/sec; read ratio= RW (50%); user locations are Sydney and Tokyo; latency SLO=1 sec. To understand the effects in Fig. 4(a)-(c), we develop a simple analytical model based on the empirical results (details in Appendix E). Our model relates cost to K, object size (o), arrival rate (λ), and f as follows:

$$cost = \left(c_1 \cdot \lambda \cdot K + c_2 \cdot o \cdot \lambda \cdot \frac{f}{K} + c_3 \cdot o \cdot \frac{2f}{K} + \bar{c}_4\right). \quad (2)$$

Here, c_1, c_2, c_3 are workload and system-specific constants VM cost, network cost, and storage cost, resp.[16] Our model captures and helps understand the *non-monotonicity of cost in K* seen in Fig. 4(a). This behavior emerges because the following cost components move in opposite directions with growing K: network and storage costs decrease due to reduction in object size,

[16] $\bar{c}_4$ is a constant and does not affect K_{opt}.

while VM costs increase due to increase in quorum sizes. Fundamentally, this implies that even under very relaxed latency constraints, the highest value of K is not necessarily optimal. Our model yields the following optimal value of K: $K_{opt} = \sqrt{\frac{o \cdot f \cdot (c_2 \cdot \lambda + 2c_3)}{c_1 \cdot \lambda}}$. Observe that K_{opt} increases with object size[17], which is in agreement with Fig. 4(b). We observe a similar qualitative match between our model-predicted dependence of K_{opt} on arrival rate and that in Fig. 4(c). K_{opt} is a decreasing function of the arrival rate λ, and saturates to a constant K^* when $\lambda \to \infty$, i.e., when the storage cost becomes a negligible component of the overall cost. Interestingly, even for $\lambda \to \infty$, the system does not revert to replication, i.e., K^* is not necessarily 1.

5.3.5 Does EC Necessarily Have Higher Latency Than Replication?

Conventional wisdom dictates that EC has lower costs than replication but suffers from higher latency. We show that perhaps surprisingly, this insight does not always lead to the right choices in the geo-distributed setting. Note that for a linearizable store, requests cannot be local [13], and so even with replication, requests need to contact multiple DCs and the overall latency corresponds to the response time of the farthest DC. Thus, in a geo-distributed scenario where there are multiple DCs at similar distances as the farthest DC in a replication-based system, EC can offer comparable latency at a lower cost. Our optimizer corroborates this insight. Consider a workload where requests to a million objects of 1 KB come from users in Tokyo. The workload is HR (read ratio of 97%) with an arrival rate of 500 req/sec. To tolerate f=1 failure, the lowest GET latency achievable via ABD is 139 msec at a cost of $\$1.057$ per hour, whereas using CAS achieves a GET latency of 160 msec at a cost of $\$0.704$ per hour - a cost saving of 33% for a mere 21 msec of latency gap. To tolerate f=2 failures for the same workload, the lowest GET latency with ABD is 180 msec at a cost of $\$1.254$ per hour, whereas CAS offers a GET latency of 190 msec at a cost of $\$0.773$ per hour - 38% lower cost for a mere 10 msec latency increase.

5.3.6 Are Nearest DCs Always the Right Choice?

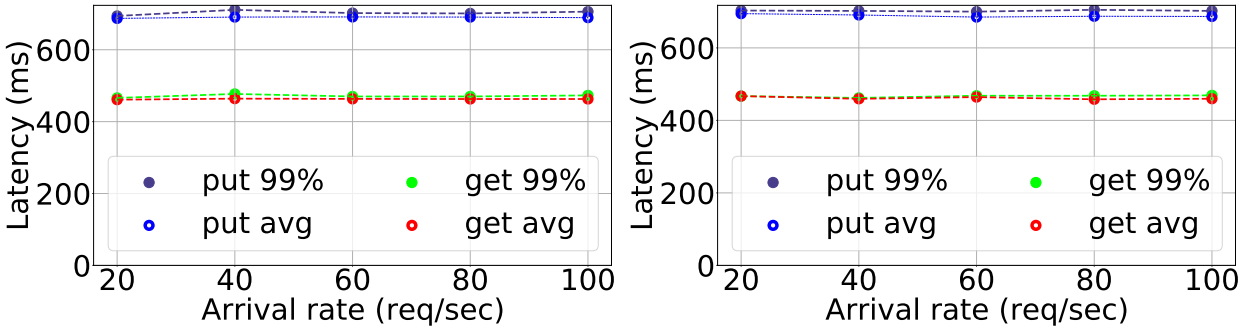
Our optimizer reveals that, perhaps surprisingly, the naturally appealing approach of using DCs nearest to user locations [6] can lead to wasted costs. We describe one such finding in Appendix F.3.

5.4 Scalable Concurrency Handling

A distinguishing feature of LEGOStore is that it is designed to provide reliable tail latency even in the face of highly concurrent access to a key. For consensus-based protocols that apply operations sequentially to a replicated state machines one after another, even in the optimistic case where all operations are issued at a leader that does not fail, the latency is expected to grow linearly with the amount of concurrency. Furthermore, in distributed consensus, due to FLP impossibility [29], concurrent operations may endure several (in theory, unbounded) rounds of communication.

To validate our expectation of robust tail latency even under high concurrency, we increase the arrival rate for the *same key* with object size 1 KB. The object is configured to use CAS(5, 3) with DCs

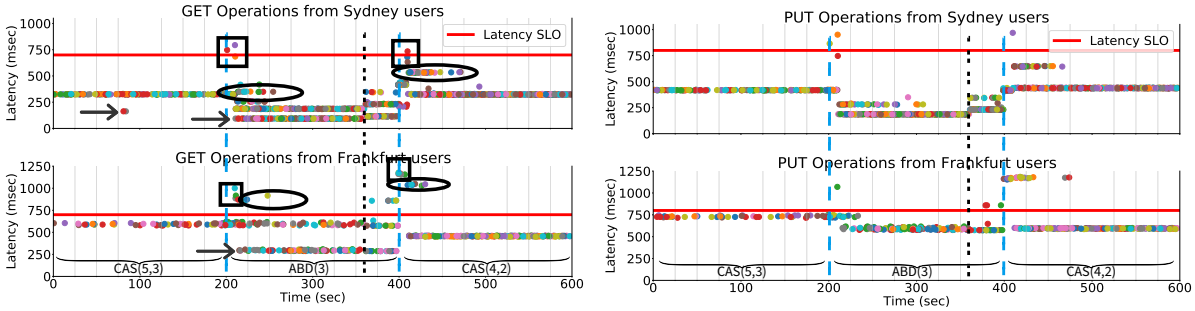
[17] A qualification to note is that the phenomenon is connected to our modeling choice of having VM cost independent of the object size o. E.g., if the VM cost were chosen as an affine function of o, then the dependence of K_{opt} on o would diminish.



(a) Read ratio=50%.

(b) Read ratio=3.2%.

Figure 5: LEGOStore is able to ensure that latency offered to a key is robust even at for highly concurrent accesses. Here we plot the latency experienced by clients at the Tokyo location for arrival rates ranging from 20 req/sec to 100 req/sec.



(a) GET operations.

(b) PUT operations.

Figure 6: The efficacy and performance impact of two reconfigurations is shown for one of 20 keys with similar workloads. The first reconfiguration occurs in response to a 4-fold increase in request arrival rate at t=200 sec. The second reconfiguration occurs at t=400 sec in response to the Singapore DC failing at t=360 sec. The arrows show the optimized GET operations while the squares and ovals highlight two types of performance degradation associated with reconfiguration, resp.: (i) requests blocked in the old configuration, and (ii) first request issued by a user after the reconfiguration which needs to acquire the new configuration from the controller at LA. The different colors for the latency dots represent different users.

in Singapore, Frankfurt, Virginia, California, and Oregon. In particular, requests from uniformly-distributed user locations come to the single key. We run the experiments for both HW and RW for a period of 1 minute for each arrival rate. We plot the latency experienced by clients at the Tokyo location against arrival rate in Fig. 5. LEGOStore demonstrates a remarkable robustness of the latency of operations. Even for an arrival rate of 100 req/sec to the same key, every operation completes and we see no degradation in performance for the average and tail latencies. We recorded a maximum concurrency of 142 write operations on one key for an arrival rate of 100 req/sec and 30:1 write ratio. Little's law suggests an average concurrency of around 60 operations for this experiment. Note the contrast with consensus-based protocols, where the tail latency is crucially dependent on limited concurrency for a given key. E.g., in [66] Fig. 13, even with somewhat limited concurrency, the latency of only "successful" writes can grow up to 30s without leader fallback, and at least doubles with leader fallback.

The similarity of the latency in Fig. 5(b) which has a HW workload as compared with the RW workload in Fig. 5(a) indicates that our latencies remain robust even if the workload is write heavy. It is also worth noting that Fig. 5 is a further corroboration of the robustness of our modeling in Sec. 3.2. Specifically, our model ignores intra-DC phenomena such as queueing, and

the robustness of latency despite a high arrival rate shows the overwhelming significance of the inter-DC RTTs in determining response times.

5.5 Reconfiguration to Handle Load Change

In this subsection and the next, we explore LEGOStore's ability in performing fast reconfiguration in line with the expectations set in Sec. 3.3.We consider a set of 20 keys with similar workloads, each with an object size of 1 KB and f=1 to which RW (i.e., read ratio of 50%) requests arrive from 4 user locations with the following spatial distribution: Tokyo (30%), Sydney (30%), Singapore (30%), and Frankfurt (10%). Each user performs operations with a Poisson process with rate one request every 2 seconds. Our latency SLOs are 700 msec and 800 msec for GETs and PUTs, resp. As seen in Fig. 6, till t=200 sec, requests arrive at a collective rate of 100 req/sec (i.e., 200 users) from the 4 user locations. LEGOStore employs configurations with CAS(5,3) for our keys with DCs in Tokyo, Sydney, Singapore, Virginia, and Oregon. The figure plots the latency experienced by users at the Sydney and Frankfurt locations; the latency experienced by users at Singapore and Tokyo is similar in terms of SLO adherence. LEGOStore successfully meets SLOs. In fact, a small number of GET requests (highlighted using a right-facing arrow) experience superior performance as they are "optimized" GETs (recall Sec. 2). At t=200 sec, the collective request arrival rate increases 4-fold to 400 req/sec (i.e., 800 users) while all other workload features remain unchanged. We assume that the controller located at LA issues a reconfiguration without delay on detecting this workload change. For the new workload, LEGOStore's optimizer recommends a new configuration performing ABD with replication factor of 3 over DCs in Tokyo, Sydney, and Singapore. Across multiple measurements, we find that reconfiguration concludes in less than 1 sec. The breakdown of overall reconfiguration for a sample instance that takes 717 msec is: (i) reconfig query=68 msec; (ii) reconfig finalize=208 msec; (iii) reconfig write=139 msec; (iv) updating metadata=163; and (v) reconfig finish=139 msec.

We examine user experience during and in the immediate aftermath of reconfiguration. We show the latencies experienced by all the users each at the Sydney and Frankfurt locations to isolate the performance degradation experienced at each user location more clearly.A user request experiences one of two types of degradation which mainly depends on when it arrives in relation to the reconfiguration. **Type (i)** A small number of requests (small due to how quick the reconfiguration is) is blocked at the old configuration servers with the possibility of either getting eventually serviced by these old servers or having to restart in the new configuration (see Sec. 3.3). These are the requests experiencing latencies in the 750 msec - 1 sec range and highlighted using boxes for GET requests. **Type (ii)** A second possibility applies to all other requests that do not get blocked at the old configuration servers. These requests incur an additional delay of about 200 msec (users in Sydney) and 250 msec (Frankfurt) to acquire the new configuration from LA and are shown using an ovals for GET requests. This increase in latency happens because the users do not know that a reconfiguration has occurred and try to do an operation with the old configuration, e.g., see requests at $t \sim 200$ sec experiencing a slight degradation among GET operations from Sydney users.

5.6 Reconfiguration to Handle DC Failure

When a DC in one of the quorums fails, LEGOStore will send the request to all other DCs participating in the configuration that are not in the quorum. This will in general be sub-optimal cost-wise and may also fail to meet the SLO. In Appendix F.2, Figure 10, we show a sample result

where a DC failure results in such SLO degradation. To alleviate this, upon detecting a failure,[18] LEGOStore invokes its optimizer to determine a new cost-effective configuration that discounts the failed DC and then transitions to this new configuration. Fig. 6 depicts a scenario where the Singapore DC, a member of both ABD quorums, fails at t=360 sec. We assume that this failure is detected and remediated via a transition to a new configuration using CAS(4,2) at t=400 sec. Similar to the first reconfiguration, we find that the transition occurs within a second and has a small adverse impact on request latency—again, most requests whose latency exceeds the SLO are of the unavoidable Type (ii).

6 Related Work

EC Based Data Storage: This area has a rich history. Several papers have studied the design of in-memory key-value stores [56, 58, 46, 82, 76, 22, 73, 5, 26]. A significant body of work focuses on minimizing repair costs and encoding/decoding [68, 72, 18, 77, 42, 50, 64, 75, 79, 40, 41, 26]. The cost savings offered by EC have motivated its use particularly in production archival (i.e., write-once/rarely) storage systems [37, 51]. These papers do not focus on consistency aspects that are relevant to workloads with both reads and writes, nor do they study the geo-distributed setting; therefore, the key factors governing their performance are different from us. Strongly consistent EC-based algorithms and key value stores are developed in [3, 34, 27, 25, 57]; however, none of these works study the geo-distributed setting, nor the public cloud.

Strongly Consistent Geo-Distributed Storage: There are several strongly consistent geo-distributed key value stores [24, 74, 23, 66, 33, 81]. SpanStore [74] develops an optimization to minimize costs while satisfying latencies for a strongly consistent geo-distributed store on the public cloud. While there are several technical differences (e.g., SpanStore uses a blocking protocol via locks), the most important advance made by LEGOStore is its integration of EC into the picture. In addition to tuning EC parameters, LEGOStore integrates the constraints of structurally more complex EC-based protocols to enable cost savings.

Most closely related to our work are Giza [23] and Pando [66], which are both strongly consistent EC-based geo-distributed data stores. Both data stores modify consensus protocols (Paxos and Fast Paxos) to utilize EC and minimize latency. The most notable difference between these works and LEGOStore is that LEGOStore is designed to keep *tail* latency predictable and robust and keep costs low in the face of dynamism. Since Giza and Pando are based on consensus, they will tend to have higher latency under concurrent writes, e.g., for hot objects with high arrival rates. Furthermore, neither Pando nor Giza have an explicit reconfiguration protocol. LEGOStore's reconfiguration leverages the GET/PUT protocols and piggybacking of messages to quickly adapt to changes in workloads and system parameters. On the other hand, since Giza and Pando both use consensus, they offer more complex data structures and operations than LEGOStore such as Read-Modify-Writes and versioned objects. A noteworthy comparison point vs. Giza is that it does not operate in the public cloud setting. While Giza involves careful protocol choice and tweaks to improve latency, it does not describe a formal optimization framework for cost minimization.

Reconfiguration: There is a growing body of work that develops *non-blocking* algorithms for reconfiguration [52, 8]. Algorithms in [52, 8] require an additional phase of a client to contact a controller/configuration service in the critical path of *every* operation. In LEGOStore, for the common case of operations that are not concurrent with a reconfiguration, the number of phases (and

[18] LEGOStore can work with any existing approach for failure detection.

therefore the latency, costs) are identical to the baseline static protocol. Our algorithm has a resemblance to an adaptation of [47] in the tutorial [7]. That algorithm works mainly for replication and requires clients to propagate values to the new configuration rather than the controller, which can incur larger costs. Though we block some operations, our algorithm is designed to work with ABD and CAS, and to keep the reconfiguration latency/costs low and predictable.

Several works [60, 6, 9] design heuristics to determine when and which objects to reconfigure. Sharov et. al [60] give a method for optimizing the configuration of quorum-based replication schemes, including the placement of the leaders and replica locations. The paper was similar in concept to this paper, but was limited to replication-based schemes, whereas this paper focuses on erasure-coded schemes. Volley [6] describes techniques for dynamically migrating data among Microsoft's geo-distributed data centers to keep content closer to users and keeping server loads well-balanced.

Data Placement and Optimization for Public Cloud: There is a rich area of data placement and tuning of consistency parameters for replication based geo-distributed stores [65, 9, 74, 38, 59, 4, 63, 49]. These works expose the role of diverse workloads and costs in system design, and our optimization framework is inspired by this body of work. However, most of these references [65, 9, 74, 38, 59] only consider replication. References [4, 63] studied placement and parameter optimization for EC *within* a DC; while some insights are qualitatively similar, our geo-distributed setting along with its diversity makes the salient factors that govern performance different. From an optimization viewpoint, closest is [49] which studies EC over geo-distributed public clouds; however, they do not consider consistency and related algorithmic aspects and costs.

7 Conclusion and Future Directions

We developed LEGOStore, a linearizable geo-distributed key-value store which procured resources from a public cloud provider. LEGOStore's goal was to offer tail latency SLOs that were predictable and robust in the face of dynamism. We focused on salient aspects of EC's benefits for LEGOStore. Several additional aspects of key-value store design constitute interesting future directions.

Throughput and Scalability: A common stress test for key-value stores is throughput—the average number of transactions per second. We tested our prototype for throughputs of up to 2000 requests per second. LEGOStore's algorithms and design allow for unlimited concurrency, and utilization of the system capacity to its maximum extent. Thus, so long as LEGOStore is operated using a well-provisioned system where the compute and IO resources are scaled appropriately for incident workload, we expect that LEGOStore can support much higher throughputs. This is unlike consensus-based systems where high throughput—which results in increased concurrency—can result in severe latency degradation due to FLP impossibility even if the system is well-provisioned.

Workload Change Detection: Many of LEGOStore's benefits depend on careful tuning of placement, EC parameters, quorum-sizes and other parameters based on workload properties. Therefore, LEGOStore's effectiveness depends on a module that detects workload change and then reconfigures based on the detected changes. In our implementation, we simulated an idealized version of this module that perfectly detects changes in the workload. The design of such a workload change detection module is orthogonal to our paper, and can be performed based on a number of methods [67, 61].

Beyond Read/Write: We focused on a read and write operations and have not implemented read/-modify/write (RMW) operations [39]. Our data store can be used to implement a RMW operation by performing a read, followed by write; however, it would not be strongly consistent. It is worth contrasting LEGOStore with CassandraDB, whose read and write operations uses protocols similar to ABD. Cassandra offers tuneable consistency, and therefore can be operated as a strongly consistent data store (See a use-case in [2]). LEGOStore can be interpreted as a cost-effective EC-based alternative to Cassandra, when the latter is operated as strongly consistent data store for applications with read/write operations. However, Cassandra implements RMW operations and transactions via Paxos, although incurring performance degradation. Indeed, the recent paper Gryff [19] designs a protocol that offers the best of both worlds of consensus protocols and ABD—it provides a provably strongly consistent data store with RMW operations and yet provides the favorable tail-latency properties of ABD for read and write operations. Gryff is based on replication, and the development of a similar system that uses EC is an area of future research.

References

[1] Rocksdb: A persistent key-value store for flash and ram storage. `https://github.com/facebook/rocksdb`.

[2] Cassandra BlackRock. `https://www.youtube.com/watch?v=vJVHfqE2mPM`, 2021.

[3] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 59–74, 2005.

[4] M. Abebe, K. Daudjee, B. Glasbergen, and Y. Tian. Ec-store: Bridging the gap between storage and latency in distributed erasure coded systems. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 255–266, July 2018.

[5] Michael Abebe, Khuzaima Daudjee, Brad Glasbergen, and Yuanfeng Tian. Ec-store: Bridging the gap between storage and latency in distributed erasure coded systems. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 255–266. IEEE, 2018.

[6] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, San Jose, CA, April 2010. USENIX Association.

[7] Marcos K Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, Alexander Shraer, et al. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, (102):84–108, 2010.

[8] Marcos K Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *Journal of the ACM (JACM)*, 58(2):7, 2011.

[9] Masoud Saeida Ardekani and Douglas B Terry. A self-configurable geo-replicated cloud storage system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 367–381, 2014.

[10] Anish Athalye. Porcupine: A fast linearizability checker in Go. `https://github.com/anishathalye/porcupine`, 2017.

[11] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995.

[12] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

[13] Hagit Attiya and Jennifer L Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2):91–122, 1994.

[14] Amazon investigating major cloud outage, GitHub and Heroku report issues. `https://www.information-age.com/amazon-investigating-major-cloud-outage-github-and-heroku-report-issues-123459971/`, 2015.

[15] Azure outage: Microsoft working to restore key services after US regional disruption. `https://www.techrepublic.com/article/azure-outage-microsoft-working-to-restore-key-services-after-us-regional-outage/`, 2018.

[16] Microsoft's March 3 Azure East US outage: What went wrong (or right)? `https://www.zdnet.com/article/microsofts-march-3-azure-east-us-outage-what-went-wrong-or-right/`, 2020.

[17] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. Burscale: Using burstable instances for cost-effective autoscaling in the public cloud. In *Proc. ACM SOCC*, 2019.

[18] Yunren Bai, Zihan Xu, Haixia Wang, and Dongsheng Wang. Fast recovery techniques for erasure-coded clusters in non-uniform traffic network. In *Proceedings of the 48th International Conference on Parallel Processing*, page 61. ACM, 2019.

[19] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying consensus and shared registers. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 591–617, 2020.

[20] Viveck R Cadambe, Nancy Lynch, Muriel Medard, and Peter Musial. A coded shared atomic memory algorithm for message passing architectures. In *2014 IEEE 13th International Symposium on Network Computing and Applications (NCA)*, pages 253–260. IEEE, 2014.

[21] Viveck R Cadambe, Nancy Lynch, Muriel Medard, and Peter Musial. A coded shared atomic memory algorithm for message passing architectures. *Distributed Computing*, 30(1):49–73, 2017.

[22] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. Efficient and available in-memory kv-store with hybrid erasure coding and replication. *ACM Transactions on Storage (TOS)*, 13(3):25, 2017.

[23] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. Giza: Erasure coding objects across global data centers. In *Proc. USENIX ATC*, 2017.

[24] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[25] Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. PoWerStore: proofs of writing for efficient and robust storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications security*, pages 285–298. ACM, 2013.

[26] Shaohua Duan, Pradeep Subedi, Keita Teranishi, Philip Davis, Hemanth Kolla, Marc Gamell, and Manish Parashar. Scalable data resilience for in-memory data staging. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 105–115. IEEE, 2018.

[27] Partha Dutta, Rachid Guerraoui, and Ron R Levy. Optimistic erasure-coded distributed storage. In *Distributed Computing*, pages 182–196. Springer, 2008.

[28] Evernote. https://evernote.com.

[29] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one fault process. Technical report, YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE, 1982.

[30] Vm instances pricing — compute engine documentation — google cloud. `https://cloud.google.com/vpc/network-pricing`, 2021.

[31] Vm instances pricing — compute engine documentation — google cloud. `https://cloud.google.com/compute/vm-instance-pricing`, 2021.

[32] Y. Guo, A. L. Stolyar, and A. Walid. Online vm auto-scaling algorithms for application hosting in a cloud. *IEEE Transactions on Cloud Computing*, 8(03):889–898, jul 2020.

[33] Harshit Gupta and Umakishore Ramachandran. Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, pages 148–159, 2018.

[34] James Hendricks, Gregory R Ganger, and Michael K Reiter. Low-overhead byzantine fault-tolerant storage. *ACM SIGOPS Operating Systems Review*, 41(6):73–86, 2007.

[35] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[36] Eben Hewitt. *Cassandra: the definitive guide*. " O'Reilly Media, Inc.", 2010.

[37] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[38] A. Jonathan, M. Uluyol, A. Chandra, and J. Weissman. Ensuring reliability in geo-distributed edge cloud. In *2017 Resilience Week (RWS)*, pages 127–132, Sep. 2017.

[39] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *Proc. ACM ASPLOS*, 2020.

[40] Huiba Li, Yiming Zhang, Zhiming Zhang, Shengyun Liu, Dongsheng Li, Xiaohui Liu, and Yuxing Peng. {PARIX}: Speculative partial writes in erasure-coded systems. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 581–587, 2017.

[41] Shenglong Li, Quanlu Zhang, Zhi Yang, and Yafei Dai. Bcstore: Bandwidth-efficient in-memory kv-store with batch coding. *Proc. of IEEE MSST*, 2017.

[42] Xiaolu Li, Runhui Li, Patrick PC Lee, and Yuchong Hu. Openec: Toward unified and configurable erasure coding management in distributed storage systems. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 331–344, 2019.

[43] Erasure code api library written in c with pluggable erasure code backends. `https://github.com/openstack/liberasurecode`.

[44] Barbara Liskov and James Cowling. Viewstamped replication revisited. 2012.

[45] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[46] Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K Panda. Scalable and distributed key-value store-based data management using rdma-memcached. *IEEE Data Eng. Bull.*, 40(1):50–61, 2017.

[47] Nancy Lynch and Alex A Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *International Symposium on Distributed Computing*, pages 173–190. Springer, 2002.

[48] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[49] J. Matt, P. Waibel, and S. Schulte. Cost- and latency-efficient redundant data storage in the cloud. In *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*, pages 164–172, Nov 2017.

[50] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. Partial-parallel-repair (ppr): a distributed technique for repairing erasure coded storage. In *Proceedings of the eleventh European conference on computer systems*, page 30. ACM, 2016.

[51] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. f4: Facebook's warm blob storage system. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 383–398. USENIX Association, 2014.

[52] Nicolas Nicolaou, Viveck Cadambe, N Prakash, Kishori Konwar, Muriel Medard, and Nancy Lynch. Ares: Adaptive, reconfigurable, erasure coded, atomic storage. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 2195–2205. IEEE, 2019.

[53] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX ATC*, 2014.

[54] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proc. ACM SOSP*, 2013.

[55] Overleaf. https://www.overleaf.com.

[56] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417. USENIX Association, 2016.

[57] Yasushi Saito, Svend Frølund, Alistair Veitch, Arif Merchant, and Susan Spence. Fab: building distributed enterprise disk arrays from commodity components. In *ACM SIGARCH Computer Architecture News*, volume 32, pages 48–58. ACM, 2004.

[58] Dipti Shankar, Xiaoyi Lu, and Dhabaleswar K Panda. High-performance and resilient key-value store with online erasure coding for big data workloads. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 527–537. IEEE, 2017.

[59] PN Shankaranarayanan, Ashiwan Sivakumar, Sanjay Rao, and Mohit Tawarmalani. Performance sensitive replication in geo-distributed cloud datastores. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 240–251. IEEE, 2014.

[60] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. Take me to your leader!: Online optimization of distributed storage configurations. *Proc. VLDB Endow.*, 8(12):1490–1501, August 2015.

[61] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems. In *Proc. ACM SOCC*, Cascais, Portugal, 2011.

[62] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proc. ACM SOCC*, 2011.

[63] Maomeng Su, Lei Zhang, Yongwei Wu, Kang Chen, and Keqin Li. Systematic data placement optimization in multi-cloud storage for complex requirements. *IEEE Transactions on Computers*, 65(6):1964–1977, 2016.

[64] Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 39:1–39:14, New York, NY, USA, 2018. ACM.

[65] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 309–324. ACM, 2013.

[66] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. Near-optimal latency versus cost tradeoffs in geo-distributed storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 157–180, Santa Clara, CA, February 2020. USENIX Association.

[67] Bhuvan Urgaonkar, Prashant J. Shenoy, Abhishek Chandra, and Pawan Goyal. Dynamic provisioning of multi-tier internet applications. In *Proc. ICAC*, 2005.

[68] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P. Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, Syed Hussain, and Siddhartha Nandi. Clay codes: Moulding MDS codes to yield an MSR code. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 139–154, Oakland, CA, 2018. USENIX Association.

[69] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, Yee Jiun Song, and Tianyin Xu. Maelstrom: Mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. In *Proc. USENIX OSDI*, 2018.

[70] Werner Vogels. Diving Deep on S3 Consistency. `https://www.allthingsdistributed.com/2021/04/s3-strong-consistency.html`, 2021.

[71] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, pages 363–375, New York, NY, USA, 1990. ACM.

[72] Fang Wang, Yingjie Tang, Yanwen Xie, and Xuehai Tang. Xorinc: Optimizing data repair and update for erasure-coded systems with xor-based in-network computation. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 244–256. IEEE, 2019.

[73] Shuang Wang, Jianzhong Huang, Xiao Qin, Qiang Cao, and Changsheng Xie. Wps: A workload-aware placement scheme for erasure-coded in-memory stores. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–10. IEEE, 2017.

[74] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proc. ACM SOSP*, 2013.

[75] Jie Xia, Jianzhong Huang, Xiao Qin, Qiang Cao, and Changsheng Xie. Revisiting updating schemes for erasure-coded in-memory stores. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–6. IEEE, 2017.

[76] Yu Xiang, Tian Lan, Vaneet Aggarwal, Yih-Farn R Chen, Yu Xiang, Tian Lan, Vaneet Aggarwal, and Yih-Farn R Chen. Joint latency and cost optimization for erasure-coded data center storage. *IEEE/ACM Transactions on Networking (TON)*, 24(4):2443–2457, 2016.

[77] Xin Xie, Chentao Wu, Junqing Gu, Han Qiu, Jie Li, Minyi Guo, Xubin He, Yuanyuan Dong, and Yafei Zhao. Az-code: An efficient availability zone level erasure code to provide high fault tolerance in cloud storage systems. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 230–243. IEEE, 2019.

[78] Yahoo Cloud Serving Benchmark (YCSB. `https://research.yahoo.com/news/yahoo-cloud-serving-benchmark`.

[79] Matt MT Yiu, Helen HW Chan, and Patrick PC Lee. Erasure coding for small objects in in-memory kv storage. In *Proceedings of the 10th ACM International Systems and Storage Conference*, page 14. ACM, 2017.

[80] Peter Zaitsev. Machine types — compute engine documentation — google cloud. `https://cloud.google.com/compute/docs/machine-types`, 2021.

[81] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291, 2013.

[82] Dongfang Zhao, Ke Wang, Kan Qiao, Tonglin Li, Iman Sadooghi, and Ioan Raicu. Toward high-performance key-value stores through gpu encoding and locality-aware encoding. *Journal of Parallel and Distributed Computing*, 96:27–37, 2016.

client:
write$(value)$
write-query: Send query messages to all servers asking for their tags; await responses from q_1 servers. Select the largest tag; let its integer component be z. Form a new tag t as $(z+1, client\_id)$.
write-value: Send the pair $(t, value)$ to all servers; await acknowledgment from q_2 servers; return to the client.

read
read-query: Send query request to all servers asking for their tags and values; await responses from q_1 servers. Select a tuple with the largest tag, let it be (t, v).
read-writeback: Send (t, v) to all servers; await acknowledgment from q_2 servers; return the value v to the client.

server:
state variable: A tuple $(tag, value)$.
initial state: Store the default $(tag, value)$ pair (t_0, v_0).
On receipt of *read-query* message: Respond with the locally stored $(tag, value)$ pair.
On receipt of *write-query* message: Respond with the locally stored tag in $(tag, value)$ pair.
On receipt of *write-value* or a *read-writeback* message: If the tag of the message is higher than the locally stored tag, store the $(tag, value)$ pair of the message at the server. In any case, send an acknowledgment.

Figure 7: Client and server protocols for ABD. Note that the key is implicit.

A The ABD algorithm

We describe the ABD algorithm in Figure 7. The algorithm here actually refers to a multi-writer variant presented in [48]. In the algorithm description, we assume that every client has a unique identifier, and every tag is of the form $(integer, client\_id)$. The $client\_id$ field is used to break ties. ABD employs 2 quorums of servers Q_1 and Q_2. It uses Q_1 to gather timestamps and data for GET operations and only timestamps for PUT operations. Q_2 is used to write the data into. We use the notation: $q_1 = |Q_1|$ and $q_2 = |Q_2|$. The algorithm implements a linearizable object when the quorums satisfy $q_1 + q_2 \geq N$, and $q_1, q_2 \leq N - f$ where N is the number of DCs and f is the number of failures can be tolerated. To save costs, we only send requests to the servers in the set Q_1 in phase 1 (or Q_2 in phase 2) and only approach additional servers if one or more servers within this set does not respond within a timeout.

We also consider an optimized version of ABD, called $ABD\text{-}Opt$ wherein the *read-writeback* of read operations can sometimes be omitted. The bigger the $read\_ratio$ is, the more useful this modification is. To enable the $ABD\text{-}Opt$ algorithm, the *read-query* phase of read operations in Fig. 7 should be replaced with:
read-query-opt: Send query request to all servers asking for their tags and values; await responses from $max(q_1, q_2)$ servers. Let (t, v) be the pair with the maximum tag. If at least q_2 number of responses have the tag t, return the value v to the client. Otherwise, go to the *read-writeback* phase.

B The CAS Protocol

We present the CAS protocol in Figures 8 and 9. The *tag* refers to the logical timestamp of the form (integer,client-id), where each client has a unique client-id. The set of tags is denoted by $\mathcal{T}$, the set of codeword symbols is denoted by $\mathcal{W}$, and the set of nodes is denoted by $\mathcal{N}$. CAS employs 4

client:
write$(value)$
query: Send query messages to all nodes asking for the highest tag with label 'fin'; await responses from q_1 servers.

pre-write: Select the largest tag from the *query* phase; let its integer component be z. Form a new tag t as $(z+1, \text{'id'})$, where 'id' is the identifier of the client performing the operation. Apply the (n, k) MDS code to the value to obtain coded elements $w_1, w_2, \ldots, w_n$. Send $(t, w_s, \text{'pre'})$ to node s for every $\in \mathcal{N}$. Await responses from q_2 servers.

finalize: Send a *finalize* message $(t, \text{'null'}, \text{'fin'})$ to all nodes. Terminate after receiving responses from q_3 servers.

read
query: As in the write protocol.

finalize: Send a *finalize* message with tag t to all the nodes requesting the associated coded elements. Await responses from q_4 servers. If at least k nodes include their locally stored coded elements in their responses, then obtain the $value$ from these coded elements by decoding and terminate by returning $value$.

Figure 8: Client protocol for CAS. The key is implicit.

quorums: Q_1 to query for the highest timestamp; Q_2 to send the coded elements to; Q_3 to send the finalized tag to; and, finally, Q_4 to request the coded elements. We use the notation $q_i = |Q_i|, \forall i$, and the protocol satisfies safety and liveness despite f failures if the following constraints hold:

$$q_1 + q_3 > N \tag{3}$$
$$q_1 + q_4 > N \tag{4}$$
$$q_2 + q_4 \geq N + K \tag{5}$$
$$q_4 \geq K \tag{6}$$
$$\forall i \quad q_i \leq N - f \tag{7}$$

Relation between N, K, and f: By substituting q_2 and q_4 with $N - f$ from (7), we have:

$$2N + 2f \geq N + k \Rightarrow N - k \geq 2f \tag{8}$$

Like ABD, we present an optimized version of CAS, called CAS-Opt where the *read-finalize* phase of read operations can sometimes be omitted. To enable such optimization, we replace *read-query* phase with the following:
read-query-opt: Send query request to all servers asking for their tags and values; await responses from $max(q_1, q_2)$ servers. Let (t, v) be the pair with the maximum tag. If at least q_4 number of responses have the tag t, return the value v to the client. Otherwise, go to the *read-finalize* phase.

Unlike several other erasure coding based protocols [27, 34], CAS only sends one codeword symbol per operation, therefore, the number of bits sent across DCs will be significantly less than replication (and other erasure coding based algorithms). A 'fin' label indicates that the value has been transferred to a sufficient number of nodes, and can be exposed to reads. Propagation of the 'fin' label is indeed the reason for the additional phase of write operations.

server:

state variable: A variable that is a subset of $\mathcal{T} \times (\mathcal{W} \cup \{\text{'null'}\}) \times \{\text{'pre'}, \text{'fin'}\}$.

initial state: Store $(t_0, w_{0,s}, \text{'fin'})$ where s denotes the server and $w_{0,s}$ is the coded element corresponding to server node s obtained by encoding the initial value v_0.

On receipt of *query* message: Respond with the highest locally known tag that has a label 'fin', i.e., the highest tag t such that the triple $(t, *, \text{'fin'})$ is at the node, where $*$ can be a coded element or 'null'.

On receipt of *pre-write* message: If there is no record of the tag of the message in the list of triples stored at the node, then add the incoming triple in the message to the list of stored triples; otherwise ignore. Send acknowledgment.

On receipt of *finalize* from a write: Let t be the tag of the message. If a triple of the form $(t, w_s, \text{'pre'})$ exists in the list of stored triples, then update it to $(t, w_s, \text{'fin'})$. Otherwise add $(t, \text{'null'}, \text{'fin'})$ to list of stored triples[16]. Send acknowledgment.

On receipt of *finalize* from a read: Let t be the tag of the message. If a triple of the form $(t, w_s, *)$ exists in the list of stored triples where $*$ can be 'pre' or 'fin', then update it to $(t, w_s, \text{'fin'})$ and send (t, w_s) to the client. Otherwise add $(t, \text{'null'}, \text{'fin'})$ to the list of triples at the node and send an acknowledgment.

Figure 9: Server protocol for CAS. The key is implicit.

C LEGOStore's Optimizer

A Detailed Look at our Objective: The networking cost per unit time of PUTs for key g must be represented differently based on whether ABD or CAS is used for g. We use the boolean variable e_g to capture this (0 for ABD and 1 for CAS).

$$C_{g,put} = \underbrace{e_g \cdot C_{g,put,CAS}}_{\text{n/w cost if CAS chosen}} + \underbrace{(1 - e_g) \cdot C_{g,put,ABD}}_{\text{n/w cost if ABD chosen}}. \tag{9}$$

The terms $C_{g,put,ABD}$ and $C_{g,put,CAS}$ are designed to capture the intensity of network traffic exchanged between clients and the various quorums of servers they need to interact with per the concerned protocol's idiosyncrasies. Note the role played by the key boolean decision variable iq^k_{ijg} whose interpretation is: iq^k_{ijg}=1 iff data center j is in the k^{th} quorum for clients in/near data center i. We have:

$$C_{g,put,ABD} = (1 - \rho_g) \cdot \lambda_g \sum_{i=1}^{D} \alpha_{ig} \Big(\underbrace{o_m \sum_{j=1}^{D} p^n_{ji} \cdot iq^1_{ijg}}_{\text{n/w cost for phase 1}} + \underbrace{o_g \sum_{k=1}^{D} p^n_{ik} \cdot iq^2_{ikg}}_{\text{n/w cost for phase 2}} \Big). \tag{10}$$

Here, $(1-\rho_g) \cdot \lambda_g \cdot \alpha_{ig}$ captures the PUT request rate arising at/near data center i and the o_m and o_g multipliers convert this into bytes per unit time. The terms within the braces model the per-byte network transfer prices and are worthy of some elaboration. The first term represents network transfer prices that apply to the first phase of the ABD PUT protocol whereas the second term

does the same for ABD PUT's second phase. The term $p^n_{ji} \cdot iq^1_{ijg}$ should be understood as follows: since ABD's first phase involves clients in/near data center i sending relatively small-sized *write-query* messages to all servers in their "read quorum" (i.e., quorum 1, hence the 1 in the superscript of iq) followed by these servers responding with their (tag, value) pairs, the subscript in p^n_{ji} is selected to denote the price of data transfer from j (for the server at data center j) to i (for clients located in/near data center i); the multiplication with iq^1_{ijg} achieves the effect of considering these prices only for data centers that are in the read quorum for the clients being considered. The networking cost for CAS PUT is as follows:

$$C_{g,put,CAS} = (1-\rho_g) \cdot \lambda_g \sum_{i=1}^{D} \alpha_{ig} \Big\{ o_m \Big(\underbrace{\sum_{j=1}^{D} p^n_{ji} \cdot iq^1_{ijg}}_{\text{phase 1}} + \underbrace{\sum_{k=1}^{D} p^n_{ik} \cdot iq^3_{ikg}}_{phase3} \Big) + \underbrace{\frac{o_g}{k_g} \sum_{m=1}^{D} p^n_{im} \cdot iq^2_{img}}_{phase2} \Big\}. \tag{11}$$

It is instructive to compare the network cost per unit time for ABD in (10) with that for CAS in (11). First, notice how the number of terms within braces correspond to the number of phases involved in the PUT operation for each protocol (2 for ABD and 3 for CAS). Second, notice the nature of network cost savings offered by CAS: terms in (11) involve either meta-data transfer (o_m in the first 2 terms) or only a fraction of the data size ($\frac{o_g}{k_g}$ in the 3rd term) as opposed to (10) both of whose terms involve the entire data size o_g.

Modeling the cost per unit time incurred towards storage for key g is relatively straightforward (recall from Table 3 that we use m_g for length of our code which, for ABD, is the degree of replication):

$$C_{g,store} = p^s \cdot \big(e_g \cdot m_g \cdot \frac{o_g}{k_g} + (1-e_g) \cdot m_g \cdot o_g \big). \tag{12}$$

Finally, we consider the VM costs per unit time for key g (to meet the computational need of our protocol processing). We make the following simplifying yet reasonable assumptions. First, VM capacity may be procured at a relatively fine granularity. We consider this reasonable given that cloud providers offer small-sized VMs with fractional CPU capacities and a few 100 MBs of DRAM (e.g., f1-micro in GCP [80]). Consequently, our decision variable v_i (number of VMs at data center i) is a non-negative real number. Second, across diverse VM sizes (within a VM "class"), VM price tends to be proportional to CPU capacity and DRAM sizes [80].[19] Third, we assume that the extensive existing work on VM autoscaling [17, 32] can be leveraged to ensure satisfactory provisioning of VM capacity at each data center (i.e., neither excessive and wasteful over-provisioning nor under-provisioning that may degrade performance). Furthermore, we assume that this suitable VM capacity chosen by such an autoscaling policy is proportional to the total request arrival rate at data center i for key g. With these assumptions, the VM cost for key g at data center i is:

[19] How price relates to CPU/DRAM capacity may vary across classes, e.g., CPU- vs memory- vs. IO-optimized VMs or on-demand vs. reserved vs. spot vs. burstable VMs. We keep LEGOStore's VM procurement simple in the sense of confining it to a single such VM class.

$$C_{g,VM} = \theta^v \cdot \sum_{j=1}^{D} p_j^v \cdot \lambda_g \sum_{i=1}^{D} \alpha_{ig} \sum_{k=1}^{4} iq_{ijg}^k \tag{13}$$

where θ^v is an empirically determined multiplier that estimates VM capacity needed to serve the computational needs of the request rate arriving at data center j for g.

Constraints: Our optimization needs to capture the 3 types of constraints related to: (i) ensuring linearizability; (ii) meeting availability guarantees corresponding to the parameter f; and (iii) meeting latency SLOs. Recall that our SLOs are based on tail latency. The key modeling choices we make are: (i) to use worst-case latency as a "proxy" for tail latency; and (ii) ignore latency contributors within a data center other than data transfer time (e.g., various sources of queuing in the storage or compute layers, encoding and decoding time). We make a few observations related to these modeling choices. For our particular problem setting, worst-case latency turns out to be an effective proxy for tail latency because inter-DC latencies—by far the largest components of GET/PUT latencies—tend to be fairly stable over long timescales. Latency contributors within a data center tend to exhibit higher variation but tend to be much smaller than inter-DC latencies. Our choice to ignore queuing effects aligns with our earlier assumption of a well-designed autoscaling policy. Coincidentally, constraints based on worst-case latencies enable our decomposition of datastore-wide optimization into per-key optimization formulations (recall that the other enabler of such decomposition was the composability property of linearizability). As a counterpoint, SLOs based on cross-key average latencies would require constraints that combine latencies of various keys.

With these assumptions, our latency constraints take the following form. For GET operations, we have:

$$\begin{aligned} &\forall i, j, k \in \mathcal{D}, \\ &\qquad \underbrace{iq_{ijg}^1 \cdot \left(l_{ij} + l_{ji} + \frac{o_m}{B_{ji}} \right)}_{\text{Latency of first phase of GET}} + \\ &\qquad \underbrace{iq_{ikg}^4 \cdot \left(l_{ik} + \frac{o_m}{B_{ik}} + l_{ki} + \frac{o_g/k_g}{B_{ki}} \right)}_{\text{Latency of second phase of GET}} \leq l_{get}. \end{aligned} \tag{14}$$

$$\begin{aligned} &\forall i, j, k \in \mathcal{D}, \\ &\qquad \underbrace{iq_{ijg}^1 \cdot \left(l_{ij} + l_{ji} + \frac{o_m}{B_{ji}} \right)}_{\text{Latency of first phase of PUT}} + \underbrace{iq_{img}^2 \cdot \left(l_{im} + \frac{o_g/k_g}{B_{im}} + l_{mi} \right)}_{\text{Latency of second phase of PUT}} \quad + \\ &\qquad \underbrace{iq_{ikg}^3 \cdot \left(l_{ik} + \frac{o_m}{B_{ik}} + l_{ki} \right)}_{\text{Latency of third phase of PUT}} \leq l_{put}. \end{aligned} \tag{15}$$

While the LHS expressions of (14) and (15) capture worst-case latencies, they do not employ an explicit `max` operator—forcing the latencies of *all* servers in the relevant quorums accomplishes

this automatically. It is useful to compare the LHS expressions of (14) and (15) alongside the CAS protocol in Appendix B. GET operations require 2 phases with the first involving quorum Q_1 and the second involving quorum Q_4; PUT operations require 3 phases involving the quorums Q_1, Q_2, and Q_3, respectively.

The two following equations show the same constraint for GET and PUT in ABD respectively.

$$\forall i, j, k \in \mathcal{D},$$
$$\underbrace{iq^1_{ijg} \cdot \left(l_{ij} + l_{ji} + \frac{o_m}{B_{ji}} + \frac{o_g}{B_{ji}}\right)}_{\text{Latency of first phase of GET}} + \underbrace{iq^2_{ikg} \cdot \left(l_{ik} + l_{ki} + \frac{o_m}{B_{ik}} + \frac{o_g}{B_{ik}}\right)}_{\text{Latency of second phase of GET}} \leq l_{get}. \tag{16}$$

$$\forall i, j, k, m \in \mathcal{D},$$
$$\underbrace{iq^1_{ijg} \cdot \left(l_{ij} + l_{ji} + \frac{o_m}{B_{ji}}\right)}_{\text{Latency of first phase of PUT}} + \underbrace{iq^2_{ikg} \cdot \left(l_{ik} + l_{ki} + \frac{o_g}{B_{ik}}\right)}_{\text{Latency of second phase of PUT}} \tag{17}$$

Please compare the LHS expressions of (16) and (17) alongside the ABD protocol in Appendix A, and recall GET and PUT operations need 2 phases involving both Q_1 and Q_2 in each phase.

The Case of "Optimized" GETs: While our modeling assumes that GET operations have two phases, the ABD protocol can complete some GET operations in one phase; we refer to such a scenario as an "Optimized GET." A GET operation becomes an Optimized GET if all the servers return the same (tag,value) pair in the first phase. This scenario occurs if there is no concurrent operation, and some previous operation to the same key has propagated the version to the servers in the quorum that responds to the client. This optimization mirrors consensus protocols such as Paxos, where operations can complete in one phase in optimistic circumstances. Our paper also involves an optimization for CAS that enables some GET operations to complete in one phase. Since our optimization focuses on worst-case latencies, our modeling makes a conservative estimate that no operation performs an Optimized GET. A minor extension of our work can allow nodes to gossip the versions in the background (after the operation is complete) to increase the fraction of Optimized GETs for ABD protocol at an increased communication cost (See, e.g., [66] that performs such an optimization).

Discussion: We conclude with a discussion of a few important aspects of our optimization, especially paying attention to their implications for LEGOStore's operational feasibility and efficacy. A key concern for a practitioner would be the obviously high computational complexity of our formulation—notice the presence of several discrete variables and quadratic terms in our constraints. Instead of searching for the optimal configuration in all possible combinations of data centers in each quorum, we use a heuristic to reduce the search scope for the optimal configuration and, consequently, time to find one. Our heuristic will sort the data centers based on their network price coming into the client. For example, to select quorums for client i, we pick the data centers with the lowest network price to client i. (See Table 2 for price discrepancies.)

In our evaluation, individual runs of the optimizer for selecting among 9 data centers concluded within 6.5 minutes on 8 *e2-standard-4* machines from GCP which we consider adequate for the scale of our experiments. Papers describing commercial-grade systems (e.g., Volley [6]) exploit techniques (including parallelization) to scale such optimization formulations to the much larger input sizes these systems must deal with. Our optimizer is also highly parallelizable—parallelization can happen in granularity of keys as well as computing the cost within each key.

While our work only considers two protocols (ABD and CAS) to pick from for a key, it is an easy conceptual enhancement to incorporate other protocols (e.g., underlying protocols used in Cassandra [36] or Google Spanner [24]). The key enhancements needed would consist of (i) generalizing e_g to a vector of booleans of size equal to the number of considered protocols; (ii) generalizing its use to express the exclusive use of one of the considered protocols; and (ii) expressing cost and latency for each of the considered protocols similarly to what was described above.

An important concern would be the data granularity at which to perform reconfigurations. We restrict our attention to a key granularity. More generally, one could identify groups of keys with similar predicted workload properties and compute the new configuration for such a group using a single execution of the optimizer. We consider these concerns as representing relatively minor difficulty beyond our current work and leave these to future work.

The GETs are modeled similar to PUTs for both ABD and CAS:

$$C_{g,get} = \underbrace{e_g \cdot C_{g,get,CAS}}_{\text{n/w cost if CAS chosen}} + \underbrace{(1-e_g) \cdot C_{g,get,ABD}}_{\text{n/w cost if ABD chosen}}. \tag{18}$$

$$C_{g,get,ABD} = \rho_g \cdot \lambda_g \cdot o_g \sum_{i=1}^{D} \alpha_{ig} \Big(\underbrace{\Sigma_{j=1}^{D} p_{ji}^n \cdot iq_{ijg}^1}_{\text{ABD phase 1}} + \underbrace{\Sigma_{k=1}^{D} p_{ik}^n \cdot iq_{ikg}^2}_{\text{ABD phase 2}} \Big). \tag{19}$$

$$C_{g,get,CAS} = \rho_g \cdot \lambda_g \sum_{i=1}^{D} \alpha_{ig} \Big\{ o_m \Big(\sum_{j=1}^{D} p_{ji}^n \cdot iq_{ijg}^1 + \sum_{k=1}^{D} p_{ik}^n \cdot iq_{ikg}^4 \Big) + \frac{o_g}{k_g} \sum_{k=1}^{D} p_{ki}^n \cdot iq_{ikg}^4 \Big\}. \tag{20}$$

Discussion: We conclude with a discussion of a few important aspects of our optimization, especially paying attention to their implications for LEGOStore's operational feasibility and efficacy. A key concern for a practitioner would be the obviously high computational complexity of our formulation—notice the presence of several discrete variables and quadratic terms in our constraints. Instead of searching for the optimal configuration in all possible combinations of data centers in each quorum, we use a heuristic to reduce the search scope for the optimal configuration and, consequently, time to find one. Our heuristic will sort the data centers based on their network price coming into the client. For example, to select quorums for client i, we pick the data centers with the lowest network price to client i. (See Table 2 for price discrepancies.)

In our evaluation, individual runs of the optimizer for selecting among 9 data centers concluded within 6.5 minutes on 8 *e2-standard-4* machines from GCP which we consider adequate for the scale of our experiments. Papers describing commercial-grade systems (e.g., Volley [6]) exploit techniques (including parallelization) to scale such optimization formulations to the much larger input sizes these systems must deal with. Our optimizer is also highly parallelizable—parallelization can happen in granularity of keys as well as computing the cost within each key.

While our work only considers two protocols (ABD and CAS) to pick from for a key, it is an easy conceptual enhancement to incorporate other protocols (e.g., underlying protocols used in Cassandra [36] or Google Spanner [24]). The key enhancements needed would consist of (i) generalizing e_g to a vector of booleans of size equal to the number of considered protocols; (ii) generalizing its use to express the exclusive use of one of the considered protocols; and (ii) expressing cost and latency for each of the considered protocols similarly to what was described above.

An important concern would be the data granularity at which to perform reconfigurations. We restrict our attention to a key granularity. More generally, one could identify groups of keys with similar predicted workload properties and compute the new configuration for such a group using a single execution of the optimizer. We consider these concerns as representing relatively minor difficulty beyond our current work and leave these to future work.

D Proof Sketch of the Linearizability of Our Reconfiguration Protocol

We show that the system is linearizable despite reconfigurations. Our proof assumes that reconfigurations are issued sequentially by the controller (i.e., reconfiguration requests are not concurrent).

The same proof works for both CAS and ABD algorithms; we focus on the CAS algorithm here for brevity. Consider any execution of the algorithm where every operation terminates[20]. With our reconfiguration protocol, note that an operation can span multiple configurations, in particular, if it starts in one configuration and then has to restart on receiving a `operation_fail, new_configuration` message. For any operation that terminates in the execution, we refer to the configuration of that operation as the configuration in which it *terminates*. The *tag* of an operation is defined as the tag associated with it in the configuration where it terminates; for a read, it is the tag acquired in its query phase, and for a write, it is the tag sent along with the value/codeword symbol.

To ensure linearizability, the main invariant we require to show (See [48]) is the following: if operation π begins after operation ϕ ends, then, in the configuration where it terminates, π gets a tag at least as large as the tag of ϕ in its query phase. To show this for our protocol, we consider three cases.

- Case (1), ϕ and π are in the same configuration;
- Case (2), ϕ completes in an old configuration and π in a new configuration;
- Case (3) ϕ completes in a new configuration and π in an old configuration.

For Case (1), the identical proofs as [20] apply.

We show that Case (3) is impossible. For a configuration, we define the point it is *enabled* as the first point where the controller has received the acknowledgment of all the servers in the configuration. Suppose ϕ completes in configuration c_{new} and π completes in configuration c_{old}. Since π completes in the old configuration c_{old}, we infer that the point of the enabling of c_{new} is

[20]For linearizability, it suffices to consider executions where every operation terminates [48]

after the point of invocation of π. This is because, based on the protocol, an operation that begins after the enabling of a new configuration does not contact an old configuration. Since ϕ is an operation in c_{new}, the point of completion of ϕ is after the point of enabling of c_{new} and therefore after the point of invocation of π. This contradicts the assumption that ϕ completes before π begins. Therefore Case (3) is impossible.

Now, we consider Case (2). In this case, it suffices to argue that the controller read a value with a tag at least as large as the tag of ϕ. This is because π acquires a tag at least as large as the tag acquired by the controller. Since ϕ is in an old configuration, the controller issues a read r as a part of the reconfiguration. It suffices to show that the read obtains a value with a tag that is as large as the tag of ϕ. There are two possibilities, (a) at least one server receives a message from the last phase of ϕ and then responds to the first message of r, and (b) there is no server that responds to the first message of r after the last message of ϕ. In case (a) the proof is similar to the proofs of [20]; effectively, since ϕ completes, at least one server that received the fin label from ϕ responds to r. Therefore, r obtains a tag at least as large as the tag of π.

Case (b) is the critical case. Now, since ϕ completes, that means there are $n - f$ servers that responded to the final phase of ϕ. Furthermore, because we are operating in case (b), each of these servers responded to the first message of r before sending the response. From the protocol note that these servers halt responding to operations after receiving the query message from r until they get a $finalize\_reconfig$. Since these $n - f$ servers responded to ϕ, and they responded after the query message from r, we conclude from the protocol that they responded after receiving the $finalize\_reconfig$ message. The fact that they responded implies that the tag of ϕ is at most as large as the tag acquired by the read r of the controller. This completes the proof.

E An Analytical Model Relating Cost to K for CAS-Based Configurations

As K increases, the network and storage costs decrease inversely proportional as K. However, the VM costs increase because the quorum sizes increase to accommodate a larger quorum intersection, indirectly increasing the effective arrival rate for each participating DC. Further the object size increases, network and storage costs increase, whereas, in our model, the VM costs remains constant. Similarly, as the arrival-rate increases, the network and VM costs increase, whereas the storage size remains constant. Thus, overall, the total cost can be modeled as:

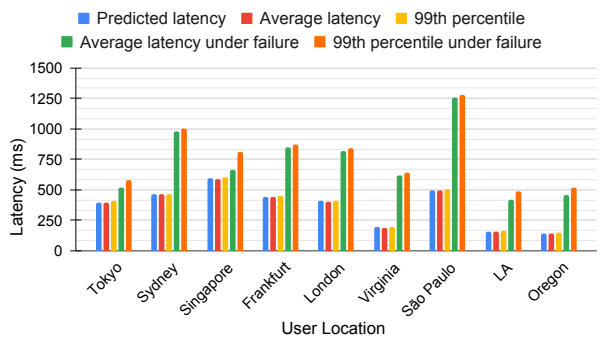
$$c_1 \cdot \lambda \cdot \frac{N+K}{2} + c_2 \cdot \lambda \cdot o\frac{N+K}{2K} + c_3 \cdot o \cdot \frac{N}{K}$$

if we neglect the metadata costs, the spatial variations in the pricing and assume that each quorum is of size $\frac{N+K}{2}$. Note here that $c_1, c_2, c3$ respectively capture the dependence on vm cost, network and storage costs. If we fix the failure tolerance as f and choose $N = K + 2f$ - the most frequent choice of the optimizer, then the overall cost can be written as:

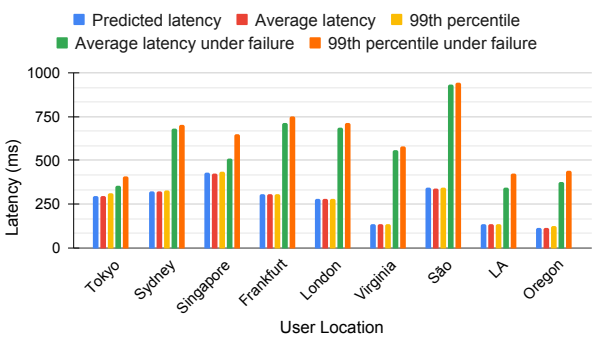
$$c_1 \cdot \lambda \cdot K + c_2 \cdot o \cdot \lambda \cdot \frac{f}{K} + c_3 \cdot o \cdot \frac{2f}{K} + \overline{c}_4$$

where $\overline{c}_4$ is a constant that does not depend on K.

F Additional Results



(a) Latency of PUT operations



(b) Latency of GET operations

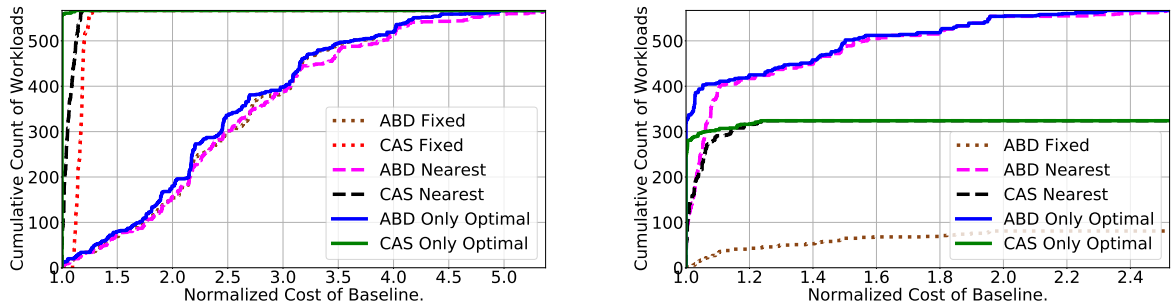
Figure 10: A sample comparison of our models for PUT and GET latencies vs. observations on our prototype. The workload chosen has a uniform user distribution and we show the latency numbers for requests arising at all 9 locations. The two rightmost bars for each location depict observed latencies when the server in LA has failed. Please note that this server is available in all quorums, so it has the worst effect on the latency. If a server that does not participate in any quorums fail, there will be no change in the latencies.

F.1 Prototype-Based Validation

We conduct extensive validation of the efficacy of the latency and cost modeling underlying our optimizer. As a representative result, in Figure 10 we compare the modeled latencies against those observed on our prototype during a 10 minute long run of the following workload for which our optimizer chooses CAS(4,2): (i) Uniform client distribution; object size=1KB; datastore size=1 TB; request rate=200 req/s; read ratio=HW; latency SLO=1 sec. Besides the excellent match between modeled and observed latencies, we also note that the observed tail latency is not much higher than the average. This is because the latency's dominant component are inter-DC RTTs that we have found to be largely stable over long periods. This dominance of inter-DC RTTs also implies that the effect of stragglers on latency and the benefits of EC over replication in mitigating it as recently noted in [56] in the context of a datastore confined to a single data center is not an important concern for us. In several experiments, we have also observed that for some workloads, depending on the arrival rate and read ratio, our average GET latencies could be much smaller than the predicted latency because a fraction of the reads return in one phase due to optimized GET operations (See Section A and B).

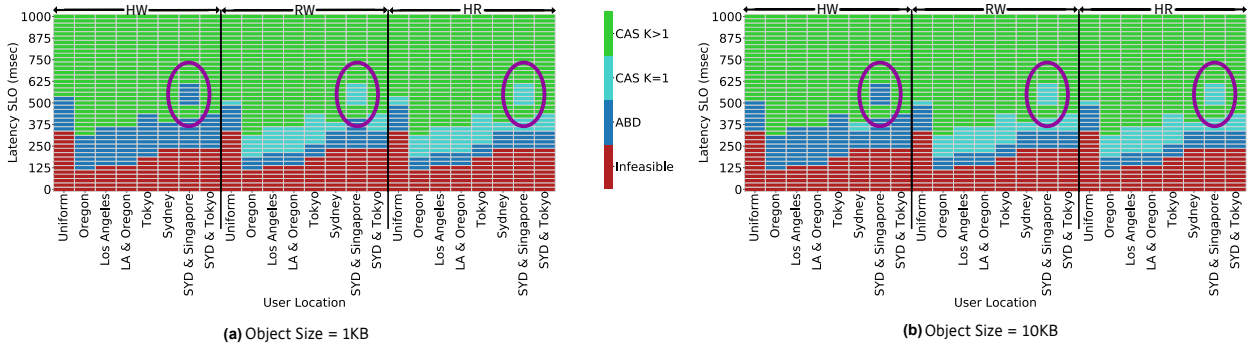
F.2 Performance Under Failure

We consider workloads that tolerate $f = 1$ failure and study the effect of a node failure on performance. Specifically, in Fig. 10, we show the change in latency when the server participating in all the quorums fails. Essentially, if the failed server is not in the quorum, there will be no degradation and we can simply replace the failed server. The degradation in Fig. 10 happens because once a client does not receive a response from at least one server in its optimized quorum, it will retry the operation by sending the request to all the servers, and once it has enough number of response to maintain linearizability, it will return with success.



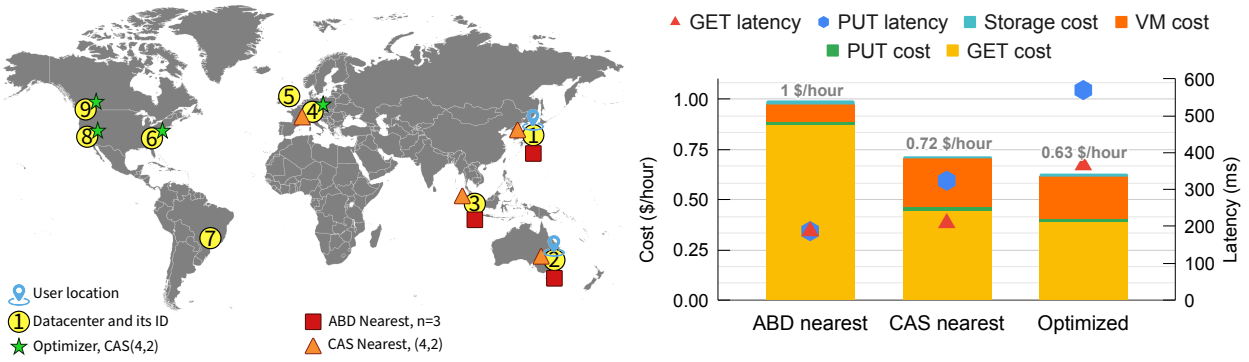
(a) Latency SLO=1 sec. **(b)** Latency SLO=300 msec.

Figure 11: Cumulative count of the normalized cost of our baselines (for our 567 basic workloads) with $f = 2$ and two extreme latency SLOs. Note that no configuration was feasible for CAS Fixed when the latency SLO is 300 msec.



(a) Object Size = 1KB **(b)** Object Size = 10KB

Figure 12: Sensitivity of the optimizer's choice to the latency SLO. We consider 2 object sizes (1KB and 10KB), 8 different client distributions, arrival rate=500 req/sec, and f=2. We consider 3 different read ratios (HW, RW, HR defined in Section 5.1).



(a) DC and user locations; DCs chosen by different solutions

(b) Cost and latency

Figure 13: An experiment demonstrating that choosing the nearest data centers may be sub-optimal. In (a), we show our client locations and the data enters chosen by ABD Nearest, CAS Nearest, and our optimizer. In (b), we compare the component-wise costs ($/hour) and GET/PUT latency offered by these three approaches.

F.3 Are Nearest DCs Always the Right Choice?

Our optimizer reveals that, perhaps surprisingly, the naturally appealing approach of using DCs nearest to user locations [6] can lead to wasted costs. To demonstrate an extreme case of this, we consider an HR workload with 50% of the requests each coming from Sydney and Tokyo with a latency SLO of 1 sec, f=1, and object size of 1 KB. The set of DCs that our optimizer chooses includes neither Tokyo nor Sydney—see Figure 13a. Our optimizer chooses CAS with parameters (N=4,K=2) with four DCs spread across North America and Europe. This is because the network prices for the Singapore and Sydney data centers are nearly twice as expensive (recall Table 2) as the others. We compare the component-wise costs (in $/hour) as well as latencies offered by our nearness-oriented baselines with our optimizer in Figure 13(b). Notice that GET networking costs are the dominant component (since the read ratio is 30:1). The use of EC by the optimizer as well as CAS Nearest significantly reduces this component while incurring higher VM costs because EC needs to use more data centers to maintain $N - K > 2f$ (see Equation 8). In the balance, however, EC offers better costs. Although CAS Nearest uses the *right* EC parameters, its choices of the specific data centers is poorer causing its cost to be about 14% higher than our optimizer's.