Performance sensitive replication in geo-distributed cloud datastores

Shankaranarayanan P N, Ashiwan Sivakumar, Sanjay Rao, Mohit Tawarmalani
Purdue University
{spuzhava,asivakum,sanjay,mtawarma}@purdue.edu

***Abstract*—Modern web applications face stringent requirements along many dimensions including latency, scalability, and availability. In response, several geo-distributed cloud datastores have emerged in recent years. Customizing datastores to meet application SLAs is challenging given the scale of applications, and their diverse and dynamic workloads. In this paper, we tackle these challenges in the context of quorum-based systems (e.g. Amazon Dynamo, Cassandra), an important class of cloud storage systems. We present models that optimize percentiles of response time under normal operation and under a data-center (DC) failure. Our models consider factors like the geographic spread of users, DC locations, consistency requirements and inter-DC communication costs. We evaluate our models using real-world traces of three applications: *Twitter*, *Wikipedia* and *Gowalla* on a Cassandra cluster deployed in Amazon EC2. Our results confirm the importance and effectiveness of our models, and highlight the benefits of customizing replication in cloud datastores.**

I. Introduction

Interactive web applications face stringent requirements on latency, and availability. Service level agreements (SLAs) often require bounds on the $90th$ (and higher) percentile latencies [33], which must be met while scaling to hundreds of thousands of geographically dispersed users. Applications require 5 9's of availability or higher, and must often be operational despite downtime of an entire DC. Failures of entire DCs may occur due to planned maintenance (e.g. upgrade of power, cooling and network systems), and unplanned failure (e.g. power outages, and natural disasters) [33], [24], [4], [8] (Figure 1). Application latencies and downtime directly impact business revenue [7].

In response to these challenges, a number of systems that replicate data across geographically distributed data-centers (DCs) have emerged in recent years [24], [38], [33], [26], [23], [40], [15], [8]. An important requirement on these systems is the need to support consistent updates on distributed replicas, and ensure both low write and read latencies. This is necessitated given datastores target interactive web applications that involve reads and writes by geographically distributed users (e.g. Facebook timelines, collaborative editing). Consequently, a distinguishing aspect of cloud datastores is the use of algorithms (e.g., quorum protocols [38], [33], Paxos [24], [15], [8]) to maintain consistency across distributed replicas.

Achieving low read and write latencies with cloud datastores while meeting the consistency requirements is a challenge. Meeting these goals requires developers to carefully choose the number of replicas maintained, which DCs contain what data, as well as the underlying consistency parameters (e.g., quorum sizes in a quorum based system). Replica placement techniques in traditional Content Delivery Networks (CDNs) (e.g., [45]) do not apply because consistency has to be maintained with distributed writes while maintaining low latencies. Tailoring cloud datastores to application workloads is especially challenging given the scale of applications (potentially hundreds of thousands of data items), workload diversity across individual data items (e.g. celebrities and normal users in Twitter have very different workload patterns), and workload dynamics (e.g. due to user mobility, changes in social graph etc.)

The problem of customizing replication policies in cloud datastores to application workloads has received limited systematic attention. Some datastores like [38], [33] are based on consistent hashing, which limits their flexibility in placing replicas. Other datastores like [41], [40] assume that all data is replicated everywhere, which may be prohibitively expensive for large applications. While a few datastores can support flexible replication policies [24], [48], they require these replication decisions to be configured manually which is a daunting task.

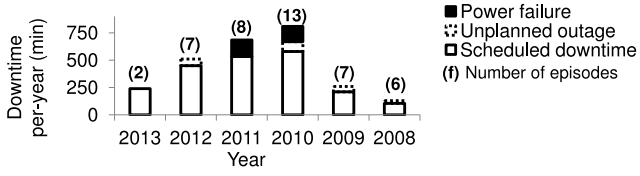


Fig. 1. Downtime and number of failure episodes (aggregated per year) of the Google App Engine data store obtained from [5].

In this paper, we present frameworks that can automatically determine how best to customize the replication configuration of geo-distributed datastores to meet desired application objectives. We focus our work on systems such as Amazon's Dynamo [33], and Cassandra [38] that employ quorum protocols. We focus on quorum-based systems given their wide usage in production [33], [38], the rich body of theoretical work they are based on [30], [28], [50], [43], and given the availability of an open-source quorum system [38]. However, we believe our frameworks can be extended to other classes of cloud storage systems as well.

We focus on optimization frameworks to obtain insights into the fundamental limits on application latency achievable for a given workload while meeting the consistency requirement.

Our models are distinguished from quorum protocols in the theoretical distributed systems community [30], [28], [50], [43], in that we focus on new aspects that arise in the context of geo-distributed cloud datastores. In particular, our models consider the impact of DC failures on datastore latency, and guide designers towards replica placements that ensure good latencies even under failures. Further, we optimize latency percentiles, allow different priorities on read and write traffic, and focus on realistic application workloads in wide-area settings.

We validate our models using traces of three popular applications: *Twitter*, *Wikipedia* and *Gowalla*, and through experiments with a multi-region Cassandra cluster [38] spanning all 8 EC2 geographic regions. While latencies with Cassandra vary widely across different replication configurations, our framework generates configurations which perform very close to predicted optimal on our multi-region EC2 setup. Further, our schemes that explicitly optimize latency under failure are able to out-perform failure-agnostic schemes by as much as 55% under the failure of a DC while incurring only modest penalties under normal operation. Our results also show the importance of choosing configurations differently across data items of a single application given the heterogeneity in workloads. For instance, our *Twitter* trace required 1985 distinct replica configurations across all items, with optimal configurations for some items often performing poorly for other items. Overall the results confirm the importance and effectiveness of our frameworks in customizing geo-distributed datastores to meet the unique requirements of cloud applications.

II. Replication in geo-distributed datastores

A commonly used scheme for geo-replicating data is to use a master-slave system, with master and slave replicas located in different DCs, and data asynchronously copied to the slave [2], [4]. However, slaves may not be completely synchronized with the master when a failure occurs. The system might serve stale data during the failure, and application-level reconciliation may be required once the master recovers [4], [8]. On the other hand, synchronized master-slave systems ensure consistency but face higher write latencies.

To address these limitations with master-slave systems, many geo-distributed cloud storage systems [24], [15], [8], [40], [15], [19], [48], [37], [41], [26] have been developed in the recent years. A distinguishing aspect of cloud datastores is the use of algorithms to maintain consistency across distributed replicas, though they differ in their consistency semantics and algorithms used. Systems like Spanner [24] provide database-like transaction support while other systems like EIGER[40] and COPS[41] offer weaker guarantees, primarily with the goal of achieving lower latency.

Quorum-based datastores: Quorum protocols have been extensively used in the distributed systems community for managing replicated data [30]. Under quorum replication, the datastore writes a data item by sending it to a set of replicas (called a write quorum) and reads a data item by fetching it from a possibly different set of replicas (called a read quorum). While classical quorum protocols [30] guarantee strong consistency, many geo-distributed datastores such as Dynamo [33], and Cassandra [38] employ adapted versions of the quorum protocol, and sacrifice stronger consistency for greater availability [33]. In these systems, reads (or writes) are sent to all replicas, and the read (or write) is deemed successful if acknowledgments are received from a quorum. In case the replicas do not agree on the value of the item on a read, typically, the most recent value is returned to the user [33], [38], and a background process is used to propagate this value to other replicas. Replication in these systems can be configured so as to satisfy the strict quorum property:

$$R + W > N \tag{1}$$

where N is the number of replicas, R and W are the read and write quorum sizes respectively. This ensures that any read and write quorum of a data item intersect. Configuring replication with the strict quorum property in Cassandra and Dynamo guarantees read-your-writes consistency [51]. Further, any read to a data item sees no version older than the last complete successful write for that item (though it may see any later write that is unsuccessful or is partially complete). Finally, note that Dynamo and Cassandra can be explicitly configured with weaker quorum requirements leading to even weaker consistency guarantees [14].

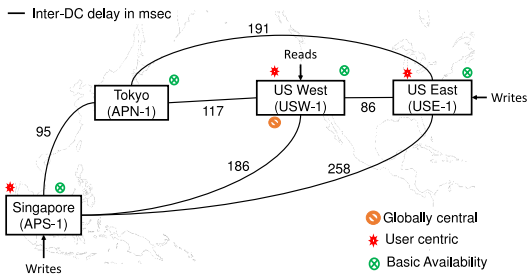


Fig. 2. Replica configuration across schemes for a set of *Twitter* data items. Reads/writes are mapped to the nearest Amazon EC2 DC. While all 8 EC2 regions (and 21 Availability Zones) were used to compute the configurations for all schemes, only DCs that appear in at least one solution are shown. For clarity, placement with *N-1C* is not shown.

III. Motivating example

In using cloud storage systems, application developers must judiciously choose several parameters such as the number of replicas (N), their location, and read(R) and write(W) quorum sizes. In this section, we illustrate the complexity in the problem using a real example, and highlight the need for a systematic framework to guide these choices. The example is from a real *Twitter* trace (Section VII-A), and represents a set of users in the West Coast who seldom tweet but actively follow friends in Asia and the East Coast.

Figure 2 depicts the placement with multiple replica configuration schemes. The DC locations and inter-DC delays were based on Amazon EC2, and we required that at most one replica may be placed in any EC2 Availability Zone (AZ). Table I summarizes the performance of the schemes. Our primary performance metric is the quorum latency, which

for the purpose of this example is the maximum of the read and write latency from any DC. The read (write) latency in a quorum datastore is the time to get responses from as many replicas as the read (write) quorum size. Our frameworks are more general and can generate configurations optimized for different priorities on read and write latencies. We discuss possible schemes:

User centric: This scheme is representative of traditional CDN approaches and aims to place replicas as close to users as possible with no regard to quorum requirements. In the limit, replicas are placed at all DCs from which accesses to the data item arrive (USW-1, APS-1, and USE-1 in our example). It may be verified that for this choice of replicas, the best quorum latency achievable is 186 msec, obtained with read and write quorum sizes of 2. Note that this placement would also be generated by the classical *Facility Location* problem when facilities may be opened with zero cost.

Globally central: This scheme seeks to place replicas at a DC which is centrally located with respect to all users by minimizing the maximum latency from all DCs with read/write requests. In our example, this scheme places a replica at USW-1. Note that for resiliency, replicas could be placed in additional availability zones of the US West region, but the quorum latency would still remain 186 msec.

***Basic Availability*:** This is our model (Section VI), which optimizes quorum latencies under normal conditions (all DCs are operational) while ensuring the system is functional under the failure of a single DC. This scheme chooses 4 replicas, one at each of the DCs, as shown in Figure 2, with $R = 3$ and $W = 2$. This configuration has a quorum latency of 117msec - a gain of 69 msec over other schemes. Intuitively, the benefit comes from our scheme's ability to exploit the asymmetry in read and write locations, increasing the number of replicas and appropriately tuning the quorum sizes.

***N-1 Contingency*:** While the Basic Availability scheme guarantees operations under any single DC failure, latencies could be poor. For e.g., on the failure of APN-1, the write latency from USE-1 increases to 258msec. Our *N-1 Contingency* scheme (Section VI) suggests configurations that guarantee optimal performance even under the failure of an entire DC. In our example, the *N-1 Contingency* scheme configures 6 replicas (3 in APN-1, 2 in USE-1 and 1 in USW-1) with $R = 5$ and $W = 2$. This configuration ensures the quorum latency remains 117 msec even under any single DC failure. Note that this configuration has the same performance as the *BA* scheme under normal conditions as well.

Overall, these results indicate the need and benefits for a systematic approach to configure replication policies in cloud datastores. Further, while our example only considers a subset of items, applications may contain tens of thousands of groups of items with different workload characteristics. Manually making decisions at this scale is not feasible.

IV. System Overview

Figure 3 shows the overview of our system. The datastore is deployed in multiple geographically distributed DCs (or availability zones), with each data item replicated in a subset of these DCs. Since our focus is on geo-replication, we consider scenarios where each DC hosts exactly one replica of each item, though our work may be easily extended to allow multiple replicas.

TABLE I
Comparing performance of schemes

Scheme	Quorum latency (msec)		N,R,W
	Normal	Failure	
Globally central	186	186	$3, 2, 2$
User centric	186	258	$3, 2, 2$
Basic Availability	117	191	$4, 3, 2$
N-1 Contingency	117	117	$6, 5, 2$

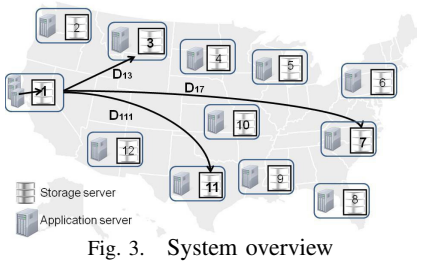


Fig. 3. System overview

Applications consist of front-end application servers and back-end storage servers. To read/write data items, an application server contacts a "coordinator" node in the storage layer which is typically co-located in the same DC. The coordinator determines where the item is replicated (e.g. using consistent hashing or explicit directories), fetches/updates the item using a quorum protocol, and responds to the application server.

We use the term "requests" to denote read/write accesses from application servers to the storage service, and we consider the request to "originate" from the DC where the application server is located. We model "request latency" as the time taken from when an application server issues a read/write request to when it gets a response from the storage service. It is possible that the application issues a single API call to the storage service that accesses multiple data items. (e.g. a multi-get call in Cassandra with multiple keys). We treat such a call as separate requests to each data item.

Users are mapped to application servers in DCs nearest to them through traditional DNS redirection mechanisms [49]. While application servers typically contact a coordinator in the same DC, a coordinator in a nearby DC may be contacted if a DC level storage service failure occurs (Section VI).

V. Latency optimized replication

In this section, we present a model that can help application developers optimize the latency seen by their applications with a quorum-based datastore. Our overall goal is to determine the replication parameters for each group of related data items. These include (i) the number, and location of DCs in which the data items must be replicated; and (ii) the read and write quorum sizes.

We expect our formulations to be applied over classes of items that see similar access patterns. For e.g., while access patterns for *Wikipedia* vary across languages, documents

TABLE II
PARAMETERS AND INPUTS TO THE MODEL

Term	Meaning
M	Number of available DCs.
D_{ij}	Access latency between DCs i and j.
C_i	Cost of outgoing traffic at DC i.
N_i^l	Number of reads/writes from DC i.
T^l	Read/Write Latency Threshold.
p^l	Fraction of requests to be satisfied within T^l.
x_i	Whether DC i hosts a replica.
q_{ij}^l	Whether i's requests use replica in j to meet quorum.
Q^l	Quorum size.
Y_i^l	Whether requests from i are satisfied within T^l.
Y_{ik}^l	Whether requests from i are satisfied within T^l on failure of replica in k.
n_{ij}	Whether reads from i fetch the full data item from j.
l	$l \in r, w$ indicates if term refers to reads/writes.

within a language see accesses from the same geographic regions, and could be grouped together. Systems like Spanner [24] require applications to bucket items into "directories", and items in the bucket see the same replica configuration. Our formulations would be applied at the granularity of directories.

In this section, we focus on latency under normal operation. In Sections VI and VI-B, we show how our models may be extended to consider latency under failure, and incorporate communication costs.

A. Meeting SLA targets under normal operation

We consider settings where the datastore is deployed in up to M geographically distributed DCs. D_{ij} denotes the time to transfer a data item from DC j to DC i. For the applications we consider, the size of objects is typically small (e.g., tweets, meta-data, small text files etc.), and hence data transmission times are typically dominated by propagation delays rather than the bandwidth between the DCs. Therefore, the D_{ij} parameter in our formulations (and evaluation) are based on the round trip times between the DCs. For applications dealing with large data objects, the measured D_{ij} values would capture the impact of data size and bandwidth as well.

Our focus is on regimes where the load on the storage node is moderate, and the primary component of the access latency is the network delay. Hence, we do not model the processing delays at the datastore node which are not as critical in the context of geo-replication.

We do not model details specific to implementation – e.g., on a read operation, the Cassandra system retrieves the full item from only the closest replica, and digests from the others. If a replica besides the closest has a more recent value, additional latency is incurred to fetch the actual item from that replica. We do not model this additional latency since the probability that a digest has the latest value is difficult to estimate and small in practice. Our experimental results in Section VIII demonstrate that, despite this assumption, our models work well in practice.

Let x_i be a binary indicator variable which is 1 iff DC i holds a replica of the data item. Let Q^r and Q^w be the read and write quorum sizes, and T^r and T^w respectively denote the latency thresholds within which all read and write accesses to the data item must successfully complete. Let q_{ij}^r and q_{ij}^w respectively be indicator variables that are 1 if read and write accesses originating from DC i use a replica in location j to meet their quorum requirements.

Typical SLAs require bounds on the delays seen by a pre-specified percentage of requests. Let p^r and p^w denote the fraction of read and write requests respectively that must have latencies within the desired thresholds. A key observation is that, given the replica locations, all read and, similarly all write requests, that originate from a given DC encounter the same delay. Thus, it suffices that the model chooses a set of DCs so that the read (resp. write) requests originating at these DCs experience a latency no more than T^r (resp. T^w) and these DCs account for a fraction p^r (resp. p^w) of read (resp. write) requests. Let N_i^r (resp. N_i^w) denote the number of read (write) requests originating from DC i. Let Y_i^r (resp. Y_i^w) denote indicator variables which are 1 iff reads (resp. writes) from DC i meet the delay thresholds. Then, we have :

$$q_{ij}^l \leq x_j \ \forall i, j \ l \in \{r, w\} \tag{2}$$

$$D_{ij} q_{ij}^l \leq T^l \ \forall i, j \ l \in \{r, w\} \tag{3}$$

$$\sum_j q_{ij}^l \geq Q^l Y_i^l \quad \forall i; \ l \in \{r, w\} \tag{4}$$

$$\sum_i N_i^l Y_i^l \geq p^l \sum_i N_i^l \quad \forall i; \ l \in \{r, w\} \tag{5}$$

Equations (2) and (3) require that DC i can use a replica in DC j to meet its quorum only if (i) there exists a replica in DC j; and (ii) DC j is within the desired latency threshold from DC i. Equation (4) ensures that, within i's quorum set, there are sufficiently many replicas that meet the above feasibility constraints for the selected DCs. Equation (5) ensures the selected DCs account for the desired percentage of requests.

To determine the lowest latency threshold for which a feasible placement exists, we treat T^r and T^w as variables of optimization, and minimize the maximum of the two variables. We allow weights a^r and a^w on read and write delay thresholds to enable an application designer to prioritize reads over writes (or vice-versa). In summary, we have the ***Latency Only*(LAT)** model:

$$\begin{array}{lll} \text{(LAT)} & \min & \mathrm{T} \\ & \text{subject to} & T \geq a^l T^l, \quad l \in \{r, w\} \\ & & Q^r + Q^w = \sum_j x_j + 1 \\ & & \text{Quorum constraints (2), (3), (4)} \\ & & \text{Percentile constraints (5)} \\ & & Q^l \in \mathbb{Z}, \quad l \in \{r, w\} \\ & & q_{ij}^l, x_j, Y_i^l \in \{0, 1\}, \quad \forall i, j; \ l \in \{r, w\} \end{array}$$

Note that the constraint on quorum sizes captures the strict quorum requirement (Section II) that each read sees the action of the last write. Also, when $p^r = p^l = 1$, (LAT) minimizes the delay of all requests and we refer to this special case as (LATM). Finally, while (4) is not linear, it may be easily linearized as we show in [47]. Hence, our model can be solved using ILP solvers like CPLEX [6].

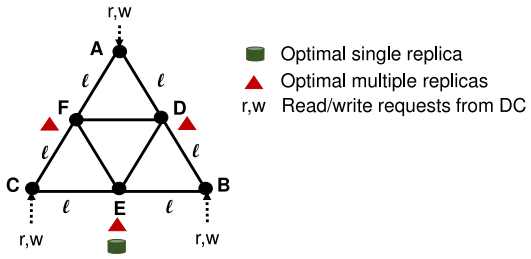


Fig. 4. An optimal multi replica solution with $Q^r = 2$, $Q^w = 2$ ensures a latency threshold of l, while an optimal single replica solution increases it to $\sqrt{3}l$

B. How much can replication lower latency?

Given the consistency requirement of quorum datastores, can replication lower latency, and, if so, by how much? In this section, we present examples to show that replication can lower latency, and provide bounds on the *replication benefit* (ratio of optimal latency without and with replication). In assessing the benefits of replication, two key factors are (i) *symmetric/asymmetric spread:* whether read and write requests originate from an identical or different set of DCs; and (ii) *symmetric/asymmetric weights:* whether the weights attached to read and write latency thresholds (a^r, a^w) are identical or different.

Figure 4 shows an example where spread and weights are symmetric and the *replication benefit* is $\sqrt{3} \approx 1.732$. When replicas can be placed arbitrarily on a Euclidean plane, it can be shown via an application of Helly's theorem [17] that the *replication benefit* is bounded by $\frac{2}{\sqrt{3}} \approx 1.155$. The setup of Figure 4 shows that this is a tight bound since replication achieves this benefit over single placement at the centroid of the triangle. Replication benefit can be even higher with asymmetric weights as seen in the observation below.

Observation 1: With asymmetric spreads and metric delays, the replication benefit for (LATM) and (LAT) is at most $4\frac{\max(a^r,a^w)}{\min(a^r,a^w)}$.

The proof can be found in our technical report [47].

VI. Achieving latency SLAs despite failures

So far, we have focused on replication strategies that can optimize latency under normal conditions. In this section we discuss failures that may impact entire DCs, and present strategies resilient to such failures.

A. Failure resilient replication strategies

While several techniques exist to protect against individual failures in a DC [27], geo-distributed DCs are primarily motivated by failures that impact entire DCs. While failures within a DC have been studied [27], [32], there are few studies on failures across DCs to the best of our knowledge. Discussions with practitioners suggests that while DC level failures are not uncommon (Figure 1), correlated failures of multiple geographically distributed DCs are relatively rare (though feasible). Operators strive to minimize simultaneous downtime of multiple DCs through careful scheduling of maintenance periods and gradual roll-out of software upgrades.

While a sufficiently replicated geo-distributed cloud datastore may be available despite a DC failure, the latency are likely negatively impacted. We present replication strategies that are resilient to such failures. Pragmatically, we first focus on the common case scenario of single DC failures. Then, in Section VI-B, we show how our models easily extend to more complex failure modes. Our models are:

Basic Availability Model (BA): This model simply optimizes latency using (LAT) with the additional constraints that the read and write quorum sizes are at least 2 (and hence the number of replicas is at least 3). Clearly, read and write requests can still achieve quorum when one DC is down and basic availability is maintained. This model does not explicitly consider latency under failure and our evaluations in Section VIII indicate that the scheme may perform poorly under failures – for e.g., the 90^{th} percentile request latency for English *Wikipedia* documents increased from 200msec to 280msec when one replica was unavailable.

N-1 Contingency Model (N-1C): This model minimizes the maximum latency across a pre-specified percentile of reads and writes allowing at most one DC to be unavailable at any given time. The model is motivated by contingency analysis techniques commonly employed in power transmission systems [36] to assess the ability of a grid to withstand a single component failure. Although this model is similar in structure to (LAT), there are two important distinctions. First, the quorum requirements must be met not just under normal conditions, but under all possible single DC failures. Second, the desired fraction of requests serviced within a latency threshold, could be met by considering requests from different DCs under different failure scenarios.

Formally, let p_f^r (resp. p_f^w) be the fraction of reads (resp. writes) that must meet the delay thresholds when a replica in any DC is unavailable. Note that the SLA requirement on failures may be more relaxed, possibly requiring a smaller fraction of requests to meet a delay threshold. Let Y_{ik}^r (resp. Y_{ik}^w) be indicator variables that are 1 if read (resp. write) requests from DC i are served within the latency threshold when the replica in DC k is unavailable. Then, we replace (5) and (4) with the following:

$$\sum_i Q_i^l Y_{ik}^l \geq p_f^l \sum_i N_i^l \ \forall i \forall k \tag{6}$$

$$\sum_{j,j\neq k} q_{ij}^l \geq Q^l Y_{ik}^l \ \forall i,k \ \ l \in \{r,w\} \tag{7}$$

The first constraint ensures that sufficient requests are serviced within the latency threshold no matter which DC fails. The index k for the Y variables allows the set of requests satisfied within the latency threshold to depend on the DC that fails. The second constraint ensures that the quorum requirements are met when DC k fails with the caveat that DC k cannot be used to meet quorum requirements. We remark that (7) may be linearized in a manner similar to (4). Putting everything

together, we have:

$$\begin{array}{lll} \text{(N-1C)} & \min & T_f \\ & \text{subject to} & T_f \geq a^l T^l, \quad l \in \{r, w\} \\ & & Q^r + Q^w = \sum_j x_j + 1 \\ & & \text{Quorum constraints (2), (3), (7)} \\ & & \text{Percentile constraints (6)} \\ & & Q^l \in \mathbb{Z}, \quad l \in \{r, w\} \\ & & q^l_{ij}, x_j, Y^l_{ik} \in \{0, 1\}, \forall i, j, k;\ l \in \{r, w\}. \end{array}$$

B. Model Enhancements

We discuss enhancements to the *N-1 Contingency* model:

Cost-sensitive replication: When datastores are deployed on public clouds, it is important to consider dollar costs in addition to latency and availability. We focus on wide-area communication costs since (i) this is known to be a dominant component of costs in geo-replicated settings [34]; (ii) best practices involve storing data in local instance storage with periodic backups to persistent storage [25] - the costs of such backups are independent of our replication policy decision; and (iii) instance costs are comparable to a single DC deployment with the same number of replicas. Most cloud providers today charge for out-bound bandwidth transfers at a flat rate per byte (in-bound transfers are typically free), though the rate itself depends on the location of the DC. Let C_i be the cost per byte of out-bound bandwidth transfer from DC i. Consider an operation that originates in DC i and involves writing a data item whose size is S bytes. Then, the total cost associated with all write operations is $\sum_i N^w_i SC_i \sum_j X_j$. However, read operations in Cassandra retrieve the full data item only from its nearest neighbor but receives digest from everyone. Let n_{ij} denote an indicator variable, which is 1 if the full data item is fetched from DC j. The size of the digest is assumed negligibly small. The total cost associated with all read operations is: $\sum_i \sum_j N^r_i n_{ij} SC_j$. It is now straightforward to modify (N-1C) to optimize costs subject to a delay constraint. This may be done by making threshold (T) a fixed parameter rather than a variable of optimization and adding additional constraints on n_{ij}.

Jointly considering normal operation and failures: Formulation (N-1C) finds replication strategies that reduce latency under failure. In practice, a designer prefers strategies that work well in normal conditions as well as under failure. This is achieved by combining the constraints in (LAT) and (N-1C), with an objective function that is a weighted sum of latency under normal conditions T and under failures T_f. The weights are chosen to capture the desired preferences.

Failures of multiple DCs: While we expect simultaneous failures of multiple DCs to be relatively uncommon, it is easy to extend our formulations to consider such scenarios. Let K be a set whose each element is a set of indices of DCs which may fail simultaneously and we are interested in guarding the performance against such a failure. We then employ (N-1C) but with k iterating over elements of K instead of the set of DCs. A naive approach may exhaustively enumerate all possible combination of DC failures, could be computationally expensive, and may result in schemes optimized for unlikely events at the expense of more typical occurrences. A more practical approach would involve explicit operator specifications of correlated failure scenarios of interest. For e.g., DCs that share the same network PoP are more likely to fail together, and thus of practical interest to operators.

Network partitions: In general, it is impossible to guarantee availability with network partition tolerance given the strict quorum requirement [31]. For more common network outages that partition one DC from others, our *N-1C* model ensures that requests from all other DCs can still be served with low latency. To handle more complex network partitions, an interesting future direction is to consider weaker quorum requirements subject to bounds on data staleness [14].

TABLE III
TRACE CHARACTERISTICS

Application	# of keys/classes	Span
Twitter[39]	3,000,000	2006-2011
Wikipedia[11]	196[1]	2009-2012
Gowalla[22]	196,591	Feb 2009-Oct 2010

VII. Evaluation Methodology

We evaluate our replication strategies *Latency Only* (LAT), *Basic Availability* (*BA*), and *N-1 Contingency* (*N-1C*) with a view to exploring several aspects such as:

- Accuracy of our model in predicting performance
- Limits on latency achievable given consistency constraints
- Benefits and costs of optimizing latency under failures
- Importance of employing heterogeneous configurations for different groups of data items within an application
- Robustness to variations in network delays and workloads

We explore these questions using experiments on a real wide-area Cassandra cluster deployed across all the 8 regions (and 21 availability zones) of Amazon EC2 and using trace-driven simulations from three real-world applications: *Twitter*, *Wikipedia* and *Gowalla*. Our EC2 experiments enable us to validate our models, and to evaluate the benefits of our approach in practice. Simulation studies enable us to evaluate our strategies on a larger scale (hundreds of thousands of data items), and to explore the impact of workload characteristics and model parameters on performance. We use GAMS [18] (a modeling system for optimization problems) and solve the models using the CPLEX optimizer.

A. Application workloads

The applications we choose are widely used, have geographically dispersed users who edit and read data, and fit naturally into a key-value model. We note that both *Twitter* and *Gowalla* are already known to use Cassandra [10]. We discuss details of the traces below (see table III for summary):

Twitter: We obtained *Twitter* traces [39] which included a user friendship graph, a list of user locations, and public tweets sent by users (along with timestamp) over a 5 year period. We analyzed Twissandra, an open-source twitter-like

[1]Aggregating all articles per language (e.g. 4 million articles in English *Wikipedia* are aggregated.)

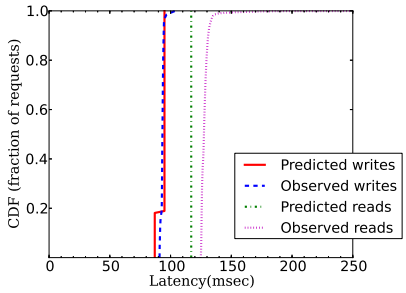


Fig. 5. Validating the accuracy of models.

application, and found three types of data items: users, tweets and timelines. We focus our evaluations on timeline objects which are pre-materialized views that map each user to a list of tweets sent by the user and her friends. Writes to a timeline occur when the associated user or her friends post a tweet, and can be obtained directly from the trace. Since the traces do not include reads, we model reads by assuming each user reads her own timeline periodically (every 10 min), and reads her friend's timeline with some probability (0.1) each time the friend posts a tweet.

Wikipedia: We obtained statistics regarding *Wikipedia* usage from [11], which lists the total as well as the breakdown of the number of views and edits by geographic region for each language and collaborative project. The data spans a 3 year period with trends shown on quarterly basis. Our model for the *Wikipedia* application consists of article objects with the document id as a key and the content along with its meta data (timestamps, version information, etc). Article page views are modeled as reads while page edits are modeled as writes. Since per article access data is not available, we model all articles of the same language and project as seeing similar access patterns since access patterns are likely dominated by the location of native speakers of the language.

Gowalla: *Gowalla* is a (now disabled) geo-social networking application where users "check-in" at various locations they visit and friends receive all their check-in messages. The traces [9] contained user friendship relationships, and a list of all check-ins sent over a two year period. Since the application workflows are similar, we model *Gowalla* in a similar fashion to *Twitter*. Check-ins represent writes to user timelines from the location of the check-in, and reads to timelines were modeled like with *Twitter*.

VIII. Experimental Validation

In this section, we present results from our experiments using Cassandra deployed on Amazon EC2.

A. Implementation

Off-the-shelf, Cassandra employs a random partitioner that implements consistent hashing to distribute load across multiple storage nodes in the cluster. The output range of a hash function is treated as a fixed circular space and each data item is assigned to a node by hashing its key to yield its position on the ring. Nodes assume responsibility for the region in the ring between itself and its predecessor, with immediately adjacent nodes in the ring hosting replicas of the data item. Cassandra allows applications to express replication policies at the granularity of keyspaces (partitions of data). We modified the applications to treat groups of data items as separate keyspaces and configure distinct replication policy for each keyspace. Keyspace creation is a one-time process and does not affect the application performance. The mapping from data object to the keyspace is maintained in a separate directory service. We implemented the directory service as an independent Cassandra cluster deployed in each of the DCs and configured its replication such that lookups(reads) are served locally within a DC (e.g. $R = 1, W = N$).

B. Experimental platform on EC2

We performed our experiments and model validations using Cassandra deployed on medium size instances on Amazon EC2. Our datastore cluster comprises of nodes deployed in each of the 21 distinct availability zones (AZ) across all the 8 regions of EC2 (9 in US, 3 in Europe, 5 in Asia, 2 in South America and 2 in Australia). We treat availability zones (AZs) as distinct DC in all our experiments. The inter-DC delays ($21 * 21$ pairs) were simultaneously measured for a period of 24 hours using medium instances deployed on all the 21 AZs and the median delays values (MED) were used as input to our models. We mapped users from their locations to the nearest DC. Since the locations are free-text fields in our traces, we make use of geocoding services [3] to obtain the user's geographical co-ordinates.

C. Accuracy and model validation

We validate the accuracy of our models with experiments on our EC2 Cassandra cluster described above. We use the example from our *Twitter* trace (Figure 2) for this experiment. Replica configurations were generated with the MED delay values measured earlier and read/write requests to Cassandra cluster were generated from application servers deployed at the corresponding DCs as per the trace data. The duration of the entire experiment was about 6 hours.

Figure 5 shows the CDFs of the *observed* and *predicted* latencies for read and write requests for the *BA* configuration. The CDFs almost overlap for write requests, while we observe a delay of approximately 9 msec evenly for all read requests. This constant delay difference in the reads can be attributed to the processing overhead of read requests in Cassandra which includes reconciling the response of multiple replicas to ensure consistency of the read data. Overall, our results validate the accuracy of our models. They also show that our solutions are fairly robust to the natural delay variations present in real cloud platforms.

D. Benefits of performance sensitive replication

We first evaluate the benefits of flexible replication policy over a fixed replication policy on the EC2 Cassandra cluster described above. For this experiment, we use a month long trace from *Twitter* consisting of $524,759$ objects corresponding to user timelines in *Twitter*. The replica configurations

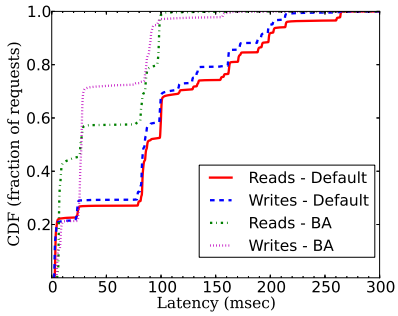
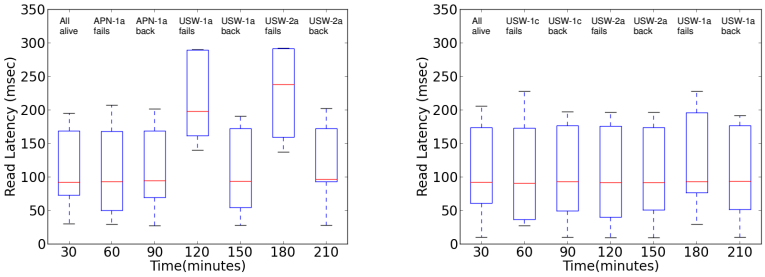


Fig. 6. Comparing the performance of *BA* scheme with Cassandra's default random partitioner.



(a) *Basic Availability* (b) *N-1 Contingency*

Fig. 7. Boxplot showing the distribution of read latency with *BA* and *N-1C* models for every half hour period. Whiskers show the 10th and 90th percentiles.

were generated for each timeline object using the *BA* model and the corresponding directory entries were created in all the regions. Reads and writes were initiated as per the traces from the Twissandra application servers deployed in each of the EC2 regions. While the duration of the entire experiment was scaled to 16 hours, care was taken to ensure that the fraction of requests to all objects from each DC was proportional to what was observed in the trace data.

Figure 6 shows the CDF comparing the read and write latency observed with our *BA* scheme and Cassandra's random partitioner. The Y-Axis shows the CDF of the fraction of all requests seen in the system (approx 6 million each for *BA* and Random) while the X-Axis shows the observed per request latency in msec. To ensure a fair comparison of schemes, the observed latency values for *BA* includes directory lookup latency as well. From the figure, we see that our flexible replication scheme is able to outperform the default replication scheme by 50 msec (factor of 3) at the $50th\%$ile and by 100msec at the $90th\%$ile (factor of 2). A keen observer might note that Random performs marginally better than (approx $3-8$msec) *BA* at the initial percentiles due to the latency overhead incurred for the directory lookup.

E. Availability and performance under failures

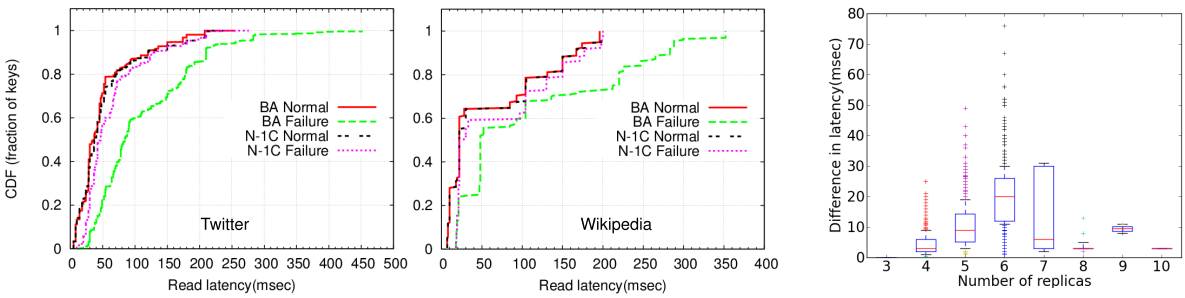
In this section, we study the performance of the *BA* and *N-1C* schemes under the failures of different DCs using our multi-region Cassandra cluster on EC2. We perform this study using the trace data from *Wikipedia* for the English wiki articles for which the accesses arrive from all the 8 EC2 regions including 50% from the US, 23% from Europe, 10% from Singapore, 5% from Sydney and the rest from South America and Tokyo. Failures were created by terminating the Cassandra process in a DC and redirecting requests from the application to the Cassandra process in the closest DC. The duration of the experiment was approximately 9 hours.

For the English wiki articles, our *BA* scheme placed two replicas in the west coast (USW-1a and USW-2a) and the 3rd replica in Tokyo (APN-1a) with $R = 2$ and $W = 2$. This is reasonable since nodes in the US West are reasonably equidistant from Asia, Australia, Europe and US East while placing the 3rd replica in Asia also reduces the $90\%ile$ latency under normal operation. Figure 7(a) shows the performance of the *BA* scheme under failure of different DCs. The corresponding events for every half hour period is marked at the top of the plots. From the figure, we see that the $90\%ile$ latency increases significantly from 200msec (under normal operation) to 280msec when the west coast DCs fail (40% increase), while the failure of Tokyo DC (APN-1a) has only a marginal impact on the performance.

In contrast, the *N-1C* scheme explicitly optimizes for latency under a failure and places the 3rd replica in USW-1a instead of Tokyo. Figure 7(b) shows the performance of the *N-1C* scheme under failures of different DCs. The figure shows that our *N-1C* scheme performs similar to the *BA* scheme (median of 90msec and $90\%ile$ of $200ms$) during normal operation. However, unlike the *BA* configuration, the $90\%ile$ latency remains largely unaffected under all failures. Our results highlight the need to explicitly optimize for performance under failure and show the benefits of *N-1C* over the *BA* scheme. Further, the median and $90\%ile$ latencies from our experiments were found to be very close to our model predictions under normal and failure conditions for both the models, thereby validating our models.

IX. Large Scale Evaluation

We adopt a trace driven simulation approach for our large scale evaluation on the three application traces, where we consider the datastore cluster to comprise of nodes from each of 27 distinct DCs world-wide, whose locations were obtained from AWS Global Infrastructure [1]. Inter-DC delays were measured between Planet-lab nodes close to each DC and delay measurements were collected simultaneously between all pairs of locations over a few hours and median delays were considered. Users were mapped to the closest DCs as in our EC2 experiments. We pick this extended set of DCs as the EC2 regions are limited in number. For example, EC2 has no regions in the Mid-west US, but AWS Global Infrastructure provides multiple DCs in those areas. Moreover, we expect these DCs to be expanded to offer more services in the future. Experiments in this section use traces of one month (Dec 2010) in *Twitter*, one month (Oct 2010) in *Gowalla* and one quarter (Q4 2011) in *Wikipedia*.



(a) Performance of BA and N-1C under failure and normal conditions

(b) Benefits of heterogeneous replication

Fig. 8. Trace driven study with all keys in the application.

A. Performance of our optimal schemes

Figure 8(a) shows the CDF of the observed read latency across both schemes for all keys in *Twitter* and *Wikipedia* traces under normal and failure conditions. For each key, we plot the read latency under normal conditions (all replicas are alive) and when the most critical replica (replica whose failure results in the worst latency) for that key fails. From the figure, we see that the read latency observed by the *BA* scheme deteriorates drastically under failure for almost all keys in both the applications. For instance, more than 40% of the keys in *Twitter* observed an increase of $50+$ msec (more than 20% of the keys observed an increase of $100+$ msec in *Wikipedia*) under failure conditions. However, read latency for *N-1C* observed only a marginal variation under failure (most keys in *Twitter* observed less than 30msec increase in latency on its replica failures). Surprisingly, we find that the *N-1C* scheme incurs an almost negligible penalty in its latency under normal conditions despite optimizing the replica configuration explicitly for the failure of a replica. Further, we found that *BA* was often able to optimize latency with two of the chosen replicas and the third choice did not significantly impact performance. In contrast, the *N-1C* scheme carefully selects the 3rd replica ensuring good performance even under failures. Overall, our results clearly show the benefit of explicitly optimizing the replication for failure conditions.

B. Need for heterogeneous configuration policy

In this section, we highlight the importance of allowing heterogeneous replica configurations in datastores and show why a uniform replication configuration for all data in the application can often have poor performance. We analyzed the configurations generated by *N-1C* for all keys in the *Twitter* trace. From our analysis we find that there were as many as 1985 distinct configurations (combination of replica location, N, R, W) that were used in the optimal solutions.

Interestingly, we find that the benefits are not only due to optimizing the location of replicas but also due to careful configuration of the replication parameters - N, R and W. To isolate such cases we consider a variant of our *N-1C* model that we call $3-2-2$ which has fixed replication parameters $N = 3$, $R = 2$ and $W = 2$, but allows flexibility in the location of the replicas. Figure 8(b) shows the difference in the access latency between the $3-2-2$ and *N-1C* schemes for *Twitter*. The X-axis has the various replication factors observed in the optimal solutions and each corresponding box plot shows the 25th, median and 75th percentiles (whiskers showing the 90th percentile) of the difference in access latency between the two schemes. Our results clearly show that a uniform configuration policy for all data in the application can be sub-optimal and allowing heterogeneity in replica configuration can greatly lower the latency (as much as 70msec in some cases).

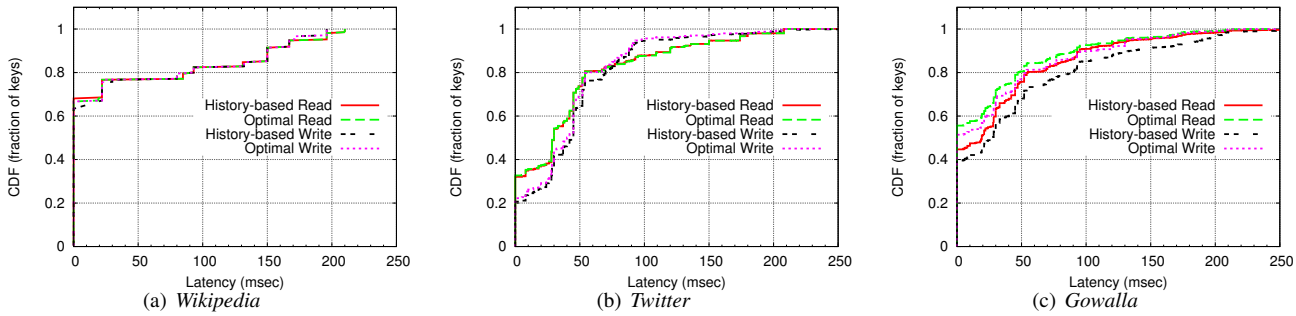
C. History-based vs Optimal

So far, we had assumed that the workloads for the applications are known. However, in practice, this may need to be obtained from historical data. In this section, we analyze this gap by comparing the performance of our schemes using historical and actual workloads for all three applications.

Figure 9(a) shows the CDF comparing the performance of *Wikipedia* during the first quarter of 2012 when using the history-based and the optimal replication configuration. The curves labeled history-based correspond to the read and write latency observed when using the replica configuration predicted from the fourth quarter of 2011. The curves labeled optimal correspond to the read and write latency observed when using the optimal replica configuration for the first quarter of 2012. Figures 9(b) and 9(c) show similar graphs for *Twitter* and *Gowalla*. These figures show that history-based configuration performs close to optimal for *Wikipedia* and *Twitter*, while showing some deviation from optimal performance for *Gowalla*. This is because users in *Gowalla* often move across geographical regions resulting in abrupt workload shifts. For such abrupt shifts, explicit hints from the user when she moves to a new location or automatically detecting change in the workload and rerunning the optimization are potential approaches for improving the performance.

D. Robustness to delay variations

Our experiments on EC2 (Section VIII) show that our strategies are fairly robust to natural delay variations across cloud DCs. In this section, we extend our analysis over a larger set of keys. We compute about 1800 time snapshots of the entire 27*27 inter-DCs delays for our extended DC set. All delay values in the snapshot were measured approximately at the same time. Next, we computed the optimal replica configurations (using our *BA* and *N-1C* schemes) for 500 random keys from the *Twitter* trace for each of 1800 snapshots.



(a) *Wikipedia* (b) *Twitter* (c) *Gowalla*

Fig. 9. Optimal performance vs performance using replica placements from the previous period.

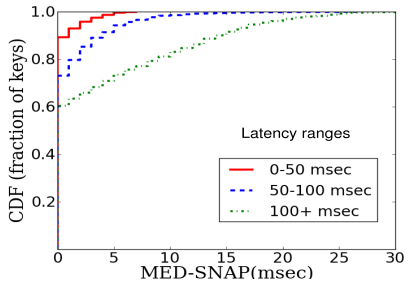


Fig. 10. Comparing SNAP and MED performance.

We call these the *SNAP* configurations. Similarly, replica configurations are computed using the median delay values of the 1800 snapshots. We call these the *MED* configurations. We then compare the performance of the *MED* configuration using delays observed at each snapshot with the performance of the optimal *SNAP* configuration at the same snapshot.

Figure 10 shows the CDF of the difference in access latency between the *MED* and *SNAP* configurations. Each curve in the figure corresponds to a range of latencies observed using the *SNAP* configurations. For *SNAP* latencies less than 100msec, and for over 90% of snapshots, MED only incurs less than 5msec additional latency. Also, for almost 80% of all the *SNAPs*, the corresponding *MED* configuration was optimal. While the penalty is higher for *SNAP* latencies over 100 msec, we believe they are still acceptable (less than 15msec for 90% of the cases) given the relatively higher *SNAP* latencies. Overall, the results further confirm our EC2 results and show that delay variation impacts placement modestly.

E. Asymmetric read and write thresholds

Thus far, we assumed that read and write latencies are equally desirable to optimize. However, in practice, some applications may prioritize read latencies, and others might prioritize writes. We have explored solutions generated by our approach when our models are modified to explicitly constrain the read and write thresholds. For *Twitter*, we found that a bound of 100msec on the write latency has no noticeable impact on the read latency, though the tail was more pronounced. Interestingly, we also found that the bound of 50msec increases the read latency by less than 20msec for 60% of the keys. We found that constraints on write latency resulted in configurations that had a significantly higher replication factor and higher read quorum sizes. This is expected because our models tries to minimize the latency by moving the replica closer to the write locations in order to meet the constraint. We omit results for lack of space.

X. Related Work

SPAR [44] presents a middle-ware for social networks which co-locates data related to each user within the same DC to minimize access latency. [44] achieves this by having a master-slave arrangement for each data item, creating enough slave replicas, and updating them in an eventually consistent fashion. However, master-slave solutions are susceptible to issues related to data loss, and temporary downtime (see Section II). In contrast, we consider a strict quorum requirement, and allow updates on any replica.

Owing to consistency constraints, quorum placement is different from facility location (FL) problems, and known variants [45]. The classical version of FL seeks to pick a subset of facilities (DCs) that would minimize the distance costs (sum of distances from each demand point to its nearest facility), plus the opening costs of the facilities. Without opening cost or capacity constraints, FL is trivial (a replica is introduced at each demand point) – however quorum placement is still complex. For e.g., in Figure 4, the optimal FL solution places 3 replicas at the triangle vertices which is twice the quorum latency of our solution. Increasing the number of replicas can hurt quorum latencies owing to consistency requirements, but does not increase distance costs with FL.

Volley [12] addresses the problem of placing data considering both user locations and data inter-dependencies. However, [12] does not address replication in depth, simply treating replicas as different items that communicate frequently. [12] does not model consistency requirements, a key focus of our work. Also, unlike [12], our models automatically determine the number of replicas and quorum parameters while considering important practical aspects like latency percentiles and performance under failures.

While systems like Spanner [24] and Walter [48] support flexible replication policies, they require these policies to be manually configured by administrators. In contrast, our formulations enable quorum based datastores to make these replica configuration decisions in an automated and optimal fashion. Recent works like Vivace [21] suggest novel read/write algorithms that employ network prioritization which enable geo-replicated datastores adapt to network congestion. Unlike these

systems, we focus on the more general and important problem of automatically configuring the replication parameters including the number of replicas, location of replicas and quorum sizes. SPANStore[52] focuses on placing replicas across multiple cloud providers with the primary aim of minimizing costs exploiting differential provider pricing. In contrast, we focus on supporting flexible replication policies at different granularities that can be tuned to a variety of objectives such as minimizing latencies under failure. Also, the quorum protocol implemented by SPANStore is different from the ones used in quorum based systems like Cassandra, and hence our model formulations are different. [46] proposes algorithms extending scalable deferred update replication (SDUR) in the context of geographically replicated systems. In contrast, we focus on the orthogonal problem of configuring optimal replication policies for geo-distributed datastores.

While there has been much theoretical analysis of quorum protocols, our work is distinguished by our focus on widely used quorum datastores, and issues unique to datastore settings. Prior work has considered communication delays with quorum protocols [28], [50], [43]. In particular, [28], [50] consider problems that minimize the maximum node delays. However, none of these works optimize latency percentiles, latency under failures, or consider different priorities for read and write traffic. To our knowledge, our framework is the first to consider these factors, all of which are key considerations for geo-distributed datastores. We also note that [28], [50], [43] are in the context of coteries [29], and do not immediately apply to cloud datastores which are adapted from weighted voting-based quorum protocols [30].

Several works have examined availability in quorum construction [16], [13], [35], [42], [20]. Most of these works do not consider the impact of failures on latency. Recent work [42] has considered how to dynamically adapt quorums to changes in network delays. Given that systems like Cassandra and Dynamo contact all replicas and not just the quorum, we focus on the orthogonal problem of replica selection so that failure of one DC does not impact latency. Several early works [16], [13] assume independent identically distributed (IID) failures, though non-IID failures are beginning to receive attention [35]. Instead, we focus on choosing replication strategies that are resilient and low-latency under failures of a single DC, or a small subset of DCs which are prone to correlated failures (Section VI-B).

XI. Discussion and Implications

We discuss the implications of our findings:

Implications for datastore design: Our results in Section IX-B show the importance of diverse replica configurations for the same application given heterogeneity in workloads for different groups of items – 1985 distinct replica configurations were required for *Twitter*. Many geo-replicated datastores are not explicitly designed with this requirement in mind and may need to revisit their design decisions. For e.g., Eiger [40] replicates all data items in the same set of DCs. Cassandra [38] and Dynamo [33] use consistent hashing which makes it difficult to flexibly map replicas to desirable DCs (we effectively bypass consistent hashing with multiple keyspaces in Section VIII). In contrast, Spanner [24] explicitly maintain directories that list locations of each group of items, and is thus better positioned to support heterogeneous replication policies.

Delay variation: Our multi-region EC2 evaluations (Section VIII) and simulation results (Section IX-D) show that placements based on median delays observed over several hours of measurement are fairly robust to short-term delay variations. We believe delay variation impacts placement modestly since links with lower median delay also tend to see smaller variations. These results indicate that the benefits of explicitly modeling stochasticity in delay is likely small, and these benefits must be weighed against the fact that stochastic delay values are hard to quantify in practice especially when not independent. Further, we note that placements from our *N-1C* model can tolerate congestion close to any DC. Finally, more persistent variations in delay over longer time-scales are best handled by recomputing placements on a periodic basis or on a prolonged change in network delays.

Workload variation: Section IX-C shows that for many applications, the optimal solution based on historical access patterns performs well compared to the solution obtained with perfect information of future access patterns. Consider the case where workloads exhibit seasonal patterns (for e.g. diurnal effects) and data-migration costs over short time-scales are large enough that one chooses to maintain same replicas across the seasons. Then, our models optimize placement assuming a percentage of total requests across seasons are satisfied within the specified latency. Instead, if one wants to have a certain service level for each season, our models may be extended by replicating the model for each season and imposing the constraint that placement decisions are season independent. Finally, we also evaluated our models with placement recomputations performed at different time granularities. We found that daily, weekly and monthly recomputations perform similarly, while hourly recomputation benefits a modest fraction(15%) of requests but incurs higher migration overheads. Hence, recomputation at coarser granularities seems to be the more appropriate choice.

Computational Complexity: Our optimization framework allows a systematic approach to analyzing replication strategies in cloud datastores, and delivers insights on the best latency achievable for a given workload with consistency constraints. With our prototype implementation LAT, *BA*, and *N-1C* models solve within 0.16, 0.17 and 0.41 seconds respectively using a single core on a 4 core, $3GHz$, $8GB$ RAM machines. While already promising, we note that (i) our implementation is not optimized. Many opportunities (heuristics, valid cuts, modeling interface) exist for better efficiency; (ii) systems like Spanner [24] require applications to bucket items, and computations would be performed at coarser bucket granularities; (iii) our per-bucket formulations are embarrassingly parallel; and (iv) our placements are stable over days (Sec IX-C) and placement recomputations are not frequent.

XII. Conclusions

In this paper, we make several contributions. First, we have developed a systematic framework for modeling geo-replicated quorum datastores in a manner that captures their latency, availability and consistency requirements. Our frameworks capture requirements on both read and write latencies, and their relative priority. Second, we have demonstrated the feasibility and importance of tailoring geo-distributed cloud datastores to meet the unique workloads of groups of items in individual applications, so latency SLA requirements (expressed in percentiles) can be met during normal operations and on the failure of a DC. Third, we explore the limits on latency achievable with geo-replicated storage systems for three real applications under strict quorum requirement. Our evaluations on a multi-region EC2 test-bed, and longitudinal workloads of three widely deployed applications validate our models, and confirm their importance.

XIII. Acknowledgments

This work was supported in part by National Science Foundation (NSF) Career Award No. 0953622 and Award No. 1162333, Google and NetApp. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, Google or NetApp. We would also like to thank Anis Elgabli for his inputs and help with the experiments and evaluation.

References

[1] Aws edge locations. http://aws.amazon.com/about-aws/globalinfrastructure/.

[2] Facebook's master slave data storage. http://www.facebook.com/note.php?note_id=23844338919.

[3] Geocoding in ArcGIS. http://geocode.arcgis.com/arcgis/index.html.

[4] Google app engine - transactions across datacenters. http://www.google.com/events/io/2009/sessions/TransactionsAcrossDatacenters.html.

[5] Google groups for App Engine Downtime Notification. https://groups.google.com/forum/?fromgroups=#!forum/google-appengine-downtime-notify.

[6] IBM ILOG CPLEX. http://www-01.ibm.com/software/integration/optimization/cplex/.

[7] Latency - it costs you. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[8] More 9s please: Under the covers of the high replication datastore. http://www.google.com/events/io/2011/sessions/more-9s-please-under-the-covers-of-the-high-replication-datastore.html.

[9] Stanford large network dataset collection. http://snap.stanford.edu/data/loc-gowalla.html.

[10] Twitter application uses key-value data store. http://engineering.twitter.com/2010/07/cassandra-at-twitter-today.html.

[11] Wikimedia statistics. http://stats.wikimedia.org/.

[12] S. Agarwal et al. Volley:automated data placement for geo-distributed cloud services. In *Proc. NSDI*, 2010.

[13] Y. Amir and A. Wool. Evaluating quorum systems over the internet. In *Proc. of FTCS*, pages 26–35, 1996.

[14] P. Bailis et al. Probabilistically bounded staleness for practical partial quorums. In *Proc. VLDB*, 2012.

[15] J. Baker et al. Megastore:providing scalable, highly available storage for interactive services. In *Proc. CIDR*, 2011.

[16] D. Barbara and H. Garcia-Molina. The reliability of voting mechanisms. *IEEE Transactions on Computers*, 36(10), October 1987.

[17] A. I. Barvinok. *A course in convexity*. American Mathematical Society, 2002.

[18] J. Bisschop et al. On the development of a general algebraic modeling system in a strategic planning environment. *Applications*, 1982.

[19] N. Bronson et al. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC*, 2013.

[20] S. Y. Cheung et al. Optimizing vote and quorum assignments for reading and writing replicated data. In *Proc. of the ICDE*, 1989.

[21] B. Cho and M. K. Aguilera. Surviving congestion in geo-distributed storage systems. In *Proc. of USENIX ATC*, 2012.

[22] E. Cho et al. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the SIGKDD*, 2011.

[23] B. F. Cooper et al. Pnuts: Yahoo!'s hosted data serving platform. In *Proceedings of the VLDB*, 2008.

[24] J. C. Corbett et al. Spanner:google's globally-distributed database. In *Proceedings of the OSDI*, 2012.

[25] DataStax. Planning an Amazon EC2 cluster. http://www.datastax.com/documentation/cassandra/1.2/webhelp/cassandra/architecture/architecturePlanningEC2_c.html.

[26] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex:a searchable distributed key-value store. In *Proc. SIGCOMM*, 2012.

[27] D. Ford et al. Availability in globally distributed storage systems. In *Proc. of OSDI*, 2010.

[28] A. W. Fu. Delay-optimal quorum consensus for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(1), 1997.

[29] H. Garcia-molina and D. Barbara. How to assign votes in a distributed system. *Journal of the Association for Computing Machinery*, 1985.

[30] D. K. Gifford. Weighted voting for replicated data. In *Proc. SOSP*, 1979.

[31] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT Newsletter*, 2002.

[32] P. Gill et al. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proc. of SIGCOMM*, 2011.

[33] D. Hastorun et al. Dynamo: amazons highly available key-value store. In *Proc. SOSP*, 2007.

[34] James Hamilton. Inter-Datacenter Replication and Geo-Redundancy. http://perspectives.mvdirona.com/2010/05/10/InterDatacenterReplicationGeoRedundancy.aspx.

[35] F. Junqueira and K. Marzullo. Coterie availability in sites. In *Proc. DISC*, 2005.

[36] U. G. Knight. *Power Systems in Emergencies: From Contingency Planning to Crisis Management*. John Wiley & Sons, LTD, 2001.

[37] T. Kraska et al. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013.

[38] A. Lakshman and P. Malik. Cassandra:a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44:35–40, 2010.

[39] R. Li et al. Towards social user profiling: unified and discriminative influence model for inferring home locations. In *KDD*, 2012.

[40] W. Lloyd et al. Stronger semantics for low-latency geo-replicated storage. *NSDI 2013*.

[41] W. Lloyd et al. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proc. SOSP*, 2011.

[42] M. M.G., F. Oprea, and M. Reiter. When and how to change quorums on wide area networks. In *Proc. SRDS*, 2009.

[43] F. Oprea and M. K. Reiter. Minimizing response time for quorum-system protocols over wide-area networks. In *Proc. DSN*, 2007.

[44] J. M. Pujol et al. The little engine(s) that could: Scaling online social networks. In *Proc. SIGCOMM*, 2010.

[45] L. Qiu et al. On the placement of web server replicas. In *Proceedings of IEEE INFOCOM 2001*.

[46] D. Sciascia and F. Pedone. Geo-replicated storage with scalable deferred update replication. In *IEEE/IFIP DSN*, 2013.

[47] P. N. Shankaranarayanan et al. Balancing latency and availability in geo-distributed cloud data stores. *Purdue University ECE Technical Reports TR-ECE-13-03*, 2013.

[48] Y. Sovran et al. Transactional storage for geo-replicated systems. In *Proc. of SOSP*. ACM, 2011.

[49] A. Su et al. Drafting behind Akamai. *SIGCOMM 2006*.

[50] T. Tsuchiya et al. Minimizing the maximum delay for reaching consensus in quorum-based mutual exclusion schemes. *IEEE TPDS*, 1999.

[51] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[52] Z. Wu et al. Cost-effective geo-replicated storage spanning multiple cloud services. In *Proc. of SOSP*, 2013.