SPANStore: Cost-Effective Geo-Replicated Storage Spanning Multiple Cloud Services

Zhe Wu*, Michael Butkiewicz*, Dorian Perkins*, Ethan Katz-Bassett†, and Harsha V. Madhyastha*
*UC Riverside** *and USC†*

Abstract

By offering storage services in several geographically distributed data centers, cloud computing platforms enable applications to offer low latency access to user data. However, application developers are left to deal with the complexities associated with choosing the storage services at which any object is replicated and maintaining consistency across these replicas.

In this paper, we present *SPANStore*, a key-value store that exports a unified view of storage services in geographically distributed data centers. To minimize an application provider's cost, we combine three key principles. First, *SPANStore* spans multiple cloud providers to increase the geographical density of data centers and to minimize cost by exploiting pricing discrepancies across providers. Second, by estimating application workload at the right granularity, *SPANStore* judiciously trades off greater geo-distributed replication necessary to satisfy latency goals with the higher storage and data propagation costs this entails in order to satisfy fault tolerance and consistency requirements. Finally, *SPANStore* minimizes the use of compute resources to implement tasks such as two-phase locking and data propagation, which are necessary to offer a global view of the storage services that it builds upon. Our evaluation of *SPANStore* shows that it can lower costs by over 10x in several scenarios, in comparison with alternative solutions that either use a single storage provider or replicate every object to every data center from which it is accessed.

1 Introduction

Today, several cloud providers offer storage as a service. Amazon S3 [1], Google Cloud Storage (GCS) [3], and Microsoft Azure [8] are notable examples. All of these services provide storage in several data centers distributed around the world. Customers can store and retrieve data via PUTs and GETs without dealing with the complexities associated with setting up and managing the underlying storage infrastructure.

Ideally, web applications should be able to provide low-latency service to their clients by leveraging the distributed locations for storage offered by these services. For example, a photo sharing webservice deployed across Amazon's data centers may serve every user from the data center closest to the user.

However, a number of realities complicate this goal. First, almost every storage service offers an isolated pool of storage in each of its data centers, leaving replication across data centers to applications. For example, even though Amazon's cloud platform has 8 data centers, customers of its S3 storage service need to read/write data at each data center separately. If a user in Seattle uploads a photo to a photo sharing webservice deployed on all of Amazon's data centers, the application will have to replicate this photo to each data center to ensure low latency access to the photo for users in other locations.

Second, while replicating all objects to all data centers can ensure low latency access [26], that approach is costly and may be inefficient. Some applications may value lower costs over the most stringent latency bounds, different applications may demand different degrees of data consistency, some objects may only be popular in some regions, and some clients may be near to multiple data centers, any of which can serve them quickly. All these parameters mean that no single deployment provides the best fit for all applications and all objects. Since cloud providers do not provide a centralized view of storage with rich semantics, every application needs to reason on its own about where and how to replicate data to satisfy its latency goals and consistency requirements at low cost.

To address this problem, we design and implement *SPANStore* ("Storage Provider Aggregating Networked Store"), a key-value store that presents a unified view of storage services present in several geographically distributed data centers. Unlike existing geo-replicated storage systems [26, 27, 32, 19], our primary focus in developing *SPANStore* is to minimize the cost incurred by latency-sensitive application providers. Three key principles guide our design of *SPANStore* to minimize cost.

First, *SPANStore* spans data centers across multiple cloud providers due to the associated performance and

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522730

cost benefits. On one hand, *SPANStore* can offer lower latencies because the union of data centers across multiple cloud providers results in a geographically denser set of data centers than any single provider's data centers. On the other hand, the cost of storage and networking resources can significantly differ across cloud providers. For example, when an application hosted in the US serves a user in China, storing the user's data in S3's California data center is more expensive ($0.105 per GB) than doing so in GCS ($0.085 per GB), whereas the price for serving data to the user has the opposite trend ($0.12 per GB in S3 vs. $0.21 per GB in GCS). *SPANStore* exploits these pricing discrepancies to drive down the cost incurred in satisfying application providers' latency, consistency, and fault tolerance goals.

Second, to minimize cost, *SPANStore* judiciously determines where to replicate every object and how to perform this replication. Replicating objects to a larger diversity of locations reduces GET latencies by moving copies closer to clients, but this additional replication increases both storage costs and the expenses necessary to pay for the bandwidth required to propagate updates. For every object that it stores, *SPANStore* addresses this trade-off by taking into consideration several factors: the anticipated workload for the object (i.e., how often different clients access it), the latency guarantees specified by the application that stored the object in *SPANStore*, the number of failures that the application wishes to tolerate, the level of data consistency desired by the application (e.g., strong versus eventual), and the pricing models of storage services that *SPANStore* builds upon.

Lastly, *SPANStore* further reduces cost by minimizing the compute resources necessary to offer a global view of storage. These compute resources are used to implement tasks such as two-phase locking while offering strong consistency and propagation of updates when offering eventual consistency. To keep costs low, we ensure that all data is largely exchanged directly between application virtual machines (VMs) and the storage services that *SPANStore* builds upon; VMs provisioned by *SPANStore* itself—rather than by application provider—are predominantly involved only in metadata operations.

We have developed and deployed a prototype of *SPANStore* that spans all data centers in the S3, Azure, and GCS storage services. In comparison to alternative designs for geo-replicated storage (such as using the data centers in a single cloud service or replicating every object in every data center from which it is accessed), we see that *SPANStore* can lower costs by over 10x in a range of scenarios. We have also ported two applications with disparate consistency requirements (a social networking webservice and a collaborative document editing application), and we find that *SPANStore* is able to meet latency goals for both applications.

2 Problem formulation

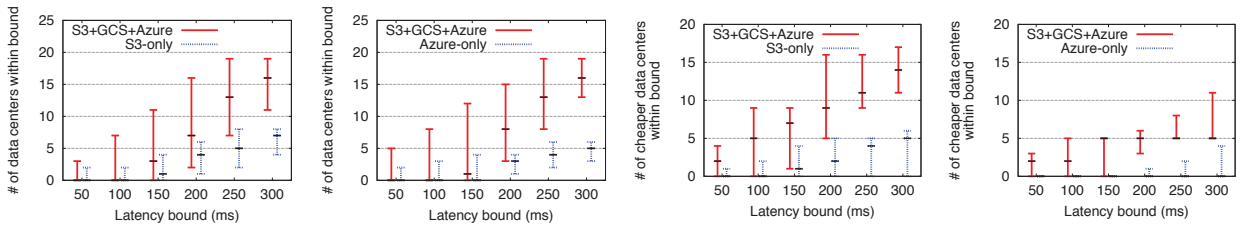
Our overarching goal in developing *SPANStore* is to enable applications to interact with a single storage service, which underneath the covers uses several geographically distributed storage services. Here, we outline our vision for how *SPANStore* simplifies application development and the challenges associated with minimizing cost.

2.1 Setting and utility

We assume an application employing *SPANStore* for data storage uses only the data centers of a single cloud service to host its computing instances, even though (via *SPANStore*) it will use multiple cloud providers for data storage. This is because different cloud computing platforms significantly vary in the abstractions that applications can build upon; an application's implementation will require significant customization in order for it be deployable across multiple cloud computing platforms. For example, applications deployed on Amazon EC2 can utilize a range of services such as Simple Queueing Service, Elastic Beanstalk, and Elastic Load Balancing. Other cloud computing platforms such as Azure and GCE do not offer direct equivalents of these services.

To appreciate the utility of developing *SPANStore*, consider a collaborative document editing webservice (similar to Google Docs) deployed across all of EC2's data centers. Say this application hosts a document that is shared among three users who are in Seattle, China, and Germany. The application has a range of choices as to where this document could be stored. One option is to store copies of the document at EC2's Oregon, Tokyo, and Ireland data centers. While this ensures that GET operations have low latencies, PUTs will incur latencies as high as 560ms since updates need to be applied to all copies of the document in order to preserve the document's consistency. Another option is to maintain only one copy of the document at EC2's Oregon data center. This makes it easier to preserve consistency and also reduces PUT latencies to 170ms, but increases GET latencies to the same value. A third alternative is to store a single copy of the document at Azure's data center on the US west coast. This deployment reduces PUT and GET latencies to below 140ms and may significantly reduce cost, since GET and PUT operations on EC2 cost 4x and 50x, respectively, what they do on Azure.

Thus, every application has a range of replication strategies to choose from, each of which presents a different trade-off between latency and cost. Today, the onus of choosing from these various options on a object-by-object basis is left to individual application developers. By developing *SPANStore*, we seek to simplify the development of distributed applications by presenting a single view to geo-replicated storage and automating the process of navigating this space of replication strategies.



(a) EC2 (b) Azure (c) EC2 (d) Azure

Figure 1: For applications deployed on a single cloud service (EC2 or Azure), a storage service that spans multiple cloud services offers a larger number of data centers (a and b) and more cheaper data centers (c and d) within a latency bound.

2.2 Goals

Four objectives guide our synthesis of geographically distributed storage services into a single key-value store.

- **Minimize cost.** Our primary goal is to minimize costs for applications that use *SPANStore*. For this, we need to minimize the total cost across *SPANStore*'s use of 1) the storage services that it unifies, and 2) the compute resources offered by the corresponding providers.
- **Respect latency SLOs.** We design *SPANStore* to serve latency-sensitive applications that have geographically distributed deployments, and therefore stand to benefit from the geo-replicated storage offered by *SPANStore*. However, the service level objectives (SLOs)[1] for GET/PUT latencies may vary across applications. While minimizing cost for any particular application, *SPANStore* must strive to meet applications' latency goals.
- **Flexible consistency.** Different applications can also vary in their requirements for the consistency of the data they store. For example, a collaborative document editing service requires strong consistency whereas eventual consistency suffices for a social networking service. *SPANStore* should respect requirements for strong consistency and exploit cases where eventual consistency suffices to offer lower latencies.
- **Tolerate failures.** Applications seek to tolerate failures of data centers and Internet paths. However, applications may differ in the cost that they are willing to bear for increased fault tolerance. *SPANStore* should account for an application's fault tolerance requirements while storing objects written by the application.

2.3 Challenges

Satisfying these goals is challenging for several reasons.

Inter-dependencies between goals. To minimize cost, it is critical that *SPANStore jointly* considers an application's latency, consistency, and fault tolerance requirements. For example, if an application desires strongly consistent data, the most cost-effective strategy is for *SPANStore* to store all of the application's data in the cheapest storage service. However, serving all PUTs and GETs from a single replica may violate the application's latency requirements, since this replica may be distant from some of the data centers on which the application is deployed. On the other hand, replicating the application's data at all data centers in order to reduce PUT/GET latencies will increase the cost for *SPANStore* to ensure strong consistency of the data.

Dependence on workload. Even if two applications have the same latency, fault tolerance, and consistency requirements, the most cost-effective solution for storing their data may differ. The lowest cost configuration for replicating any object depends on several properties of an application's workload for that object:

- The set of data centers from which the application accesses the object, e.g., an object accessed only by users within the US can be stored only on data centers in the US, whereas another object accessed by users worldwide needs wider replication.
- The number of PUTs and GETs issued for that object at each of these data centers, e.g., to reduce network transfer costs, it is more cost-effective to replicate the object more (less) if the workload is dominated by GETs (PUTs).
- The temporal variation of the workload for the object, e.g., the object may initially receive a high fraction of PUTs and later be dominated by GETs, thus requiring a change in the replication strategy for the object.

Multi-dimensional pricing. Cost minimization is further complicated by the fact that any storage service prices its use based on several metrics: the amount of data stored, the number of PUTs and GETs issued, and the amount of data transferred out of the data center in which the service is hosted. No single storage service is the cheapest along all dimensions. For example, one storage service may offer cheap storage but charge high prices for network transfers, whereas another may offer cheap network bandwidth but be expensive per PUT and GET to the service. Moreover, some storage services (e.g., GCS) charge for network bandwidth based on the location of the client issuing PUTs and GETs.

[1] We use the term SLO instead of SLA because violations of the latency bounds are not fatal, but need to be minimized.

3 Why multi-cloud?

A key design decision in *SPANStore* is to have it span the data centers of *multiple* cloud service providers. In this section, we motivate this design decision by presenting measurements which demonstrate that deploying across multiple cloud providers can potentially lead to reduced latencies for clients and reduced cost for applications.

3.1 Lower latencies

We first show that using multiple cloud providers can enable *SPANStore* to offer lower GET/PUT latencies. For this, we instantiate VMs in each of the data centers in EC2, Azure, and GCE. From the VM in every data center, we measure GET latencies to the storage service in every other data center once every 5 minutes for a week. We consider the latency between a pair of data centers as the median of the measurements for that pair.

Figure 1 shows how many other data centers are within a given latency bound of each EC2 [1(a)] and Azure [1(b)] data center. These graphs compare the number of nearby data centers if we only consider the single provider to the number if we consider all three providers—Amazon, Google, and Microsoft. For a number of latency bounds, either graph depicts the minimum, median, and maximum (across data centers) of the number of options within the latency bound.

For nearly all latency bounds and data centers, we find that deploying across multiple cloud providers increases the number of nearby options. *SPANStore* can use this greater choice of nearby storage options to meet tighter latency SLOs, or to meet a fixed latency SLO using fewer storage replicas (by picking locations nearby to multiple frontends). Intuitively, this benefit occurs because different providers have data centers in different locations, resulting in a variation in latencies to other data centers and to clients.

3.2 Lower cost

Deploying *SPANStore* across multiple cloud providers also enables it to meet latency SLOs at potentially lower cost due to the discrepancies in pricing across providers. Figures 1(c) and 1(d) show, for each EC2 and Azure data center, the number of other data centers within a given latency bound that are cheaper than the local data center along some dimension (storage, PUT/GET requests, or network bandwidth). For example, nearby Azure data centers have similar pricing, and so, no cheaper options than local storage exist within 150ms for Azure-based services. However, for the majority of Azure-based frontends, deploying across all three providers yields multiple storage options that are cheaper for at least some operations. Thus, by judiciously combining resources from multiple providers, *SPANStore* can use these cheaper options to reduce costs.

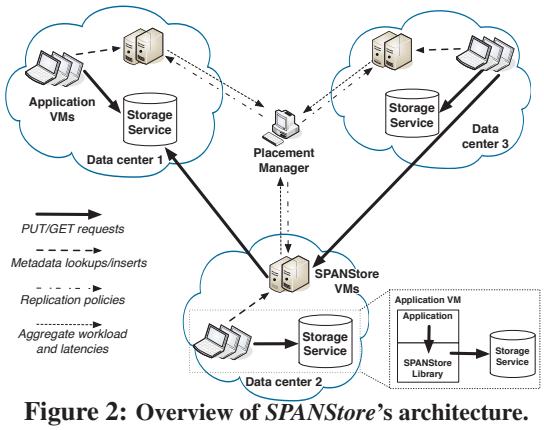


Figure 2: Overview of *SPANStore*'s architecture.

4 Overview

We design *SPANStore* such that every application uses a separate deployment of *SPANStore*. Figure 2 summarizes *SPANStore*'s deployment for any particular application. At every data center in which the application is deployed, the application issues PUT and GET requests for objects to a *SPANStore* library that the application links to. The *SPANStore* library serves these requests by 1) looking up in-memory metadata stored in the *SPANStore*-instantiated VMs in the local data center, and thereafter 2) issuing PUTs and GETs to underlying storage services. To issue PUTs to remote storage services, the *SPANStore* library may choose to relay PUT operations via *SPANStore* VMs in other data centers.

The manner in which *SPANStore* VMs should serve PUT/GET requests for any particular object is dictated by a central PlacementManager (*PMan*). We divide time into fixed-duration epochs; an epoch lasts one hour in our current implementation. At the start of every epoch, all *SPANStore* VMs transmit to *PMan* a summary of the application's workload and latencies to remote data centers measured in the previous epoch. *PMan* then computes the optimal replication policies to be used for the application's objects based on its estimate of the application's workload in the next epoch and the application's latency, consistency, and fault tolerance requirements. In our current implementation, *PMan* estimates the application's workload in a particular epoch to be the same as that observed during the same period in the previous week. *PMan* then communicates the new replication policies to *SPANStore* VMs at all data centers. These replication policies dictate how *SPANStore* should serve the application's PUTs (where to write copies of an object and how to propagate updates) and GETs (where to fetch an object from) in the next epoch.

5 Determining replication policies

In this section, we discuss *PMan*'s determination of the replication policies used in *SPANStore*'s operation. We first describe the inputs required by *PMan* and the format

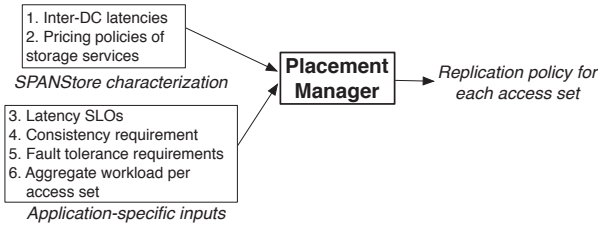


Figure 3: Overview of *PMan*'s inputs and output.

in which it outputs replication policies. We then present the algorithms used by *PMan* in two different data consistency scenarios.

5.1 Inputs and output

As shown in Figure 3, *PMan* requires three types of inputs: 1) a characterization of *SPANStore*'s deployment, 2) the application's latency, fault tolerance, and consistency requirements, and 3) a specification of the application's workload.

Characterization of *SPANStore* deployment. *PMan* requires two pieces of information about *SPANStore*'s deployment. First, it takes as input the distribution of latencies between every pair of data centers on which *SPANStore* is deployed. These latencies include measurements of PUTs, GETs, and pings issued from a VM in one data center to the storage service or a VM in another data center. Second, *PMan* needs the pricing policy for the resources used by *SPANStore*. For each data center, we specify the price per byte of storage, per PUT request, per GET request, and per hour of usage for the type of virtual machine used by *SPANStore* in that data center. We also specify, for each pair of data centers, the price per byte of network transfer from one to the other, which is determined by the upload bandwidth pricing at the source data center.

Application requirements. *PMan* also needs as input the application's latency, data consistency, and fault tolerance requirements. For the latency goals, we let the application separately specify SLOs for latencies incurred by PUT and GET operations. Either SLO is specified by a latency bound and the fraction of requests that should incur a latency less than the specified bound.

To capture consistency needs, we ask the application developer to choose between strong and eventual consistency. In the strong consistency case, we provide linearizability, i.e., all PUTs for a particular object are ordered and any GET returns the data written by the last committed PUT for the object. In contrast, if an application can make do with eventual consistency, *SPANStore* can satisfy lower latency SLOs. Our algorithms for the eventual consistency scenario are extensible to other consistency models such as causal consistency [26] by augmenting data transfers with additional metadata.

In both the eventual consistency and strong consistency scenarios, the application developer can specify the number of failures—either of data centers or of Internet paths between data centers—that *SPANStore* should tolerate. As long as the number of failures is less than the specified number, *SPANStore* should ensure the availability of all GET and PUT operations while also satisfying the application's consistency and latency requirements. When the number of failures exceeds the specified number, *SPANStore* may make certain operations unavailable or violate latency goals in order to ensure that consistency requirements are preserved.

Workload characterization. Lastly, *PMan* accounts for the application's workload in two ways. First, for every object stored by an application, we ask the application to specify the set of data centers from which it will issue PUTs and GETs for the object. We refer to this as the *access set* for the object. An application can determine the access set for an object based on the sharing pattern of that object across users. For example, a collaborative online document editing webservice knows the set of users with whom a particular document has been shared. The access set for the document is then the set of data centers from which the webservice serves these users. In cases where the application itself is unsure which users will access a particular object (e.g., in a file hosting service like Rapidshare), it can specify the access set of an object as comprising all data centers on which the application is deployed; this uncertainty will translate to higher costs. In this work, we consider every object as having a fixed access set over its lifetime. *SPANStore* could account for changes in an object's access set over time, but at the expense of a larger number of latency SLO violations; we defer the consideration of this scenario to future work.

Second, *SPANStore*'s VMs track the GET and PUT requests received from an application to characterize its workload. Since the GET/PUT rates for individual objects can exhibit bursty patterns (e.g., due to flash crowds), it is hard to predict the workload of a particular object in the next epoch based on the GETs and PUTs issued for that object in previous epochs. Therefore, *SPANStore* instead leverages the stationarity that typically exists in an application's aggregate workload, e.g., many applications exhibit diurnal and weekly patterns in their workload [11, 17]. Specifically, at every data center, *SPANStore* VMs group an application's objects based on their access sets. In every epoch, for every access set, the VMs at a data center report to *PMan* 1) the number of objects associated with that access set and the sum of the sizes of these objects, and 2) the aggregate number of PUTs and GETs issued by the application at that data center for all objects with that access set.

To demonstrate the utility of considering aggregate workloads in this manner, we analyze a Twitter dataset

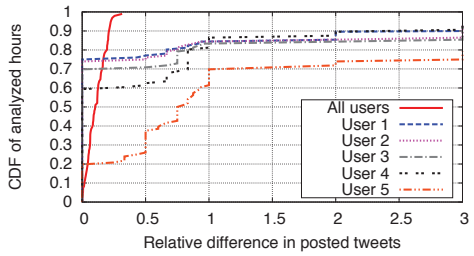


Figure 4: Comparison at different granularities of the stationarity in the number of posted tweets.

that lists the times at which 120K users in the US posted on Twitter over a month [25]. We consider a scenario in which every user is served from the EC2 data center closest to the user, and consider every user's Twitter timeline to represent an object. When a user posts a tweet, this translates to one PUT operation on the user's timeline and one PUT each on the timelines of each of the user's followers. Thus, the access set for a particular user's timeline includes the data centers from which the user's followers are served.

Here, we consider those users whose timelines have their access set as all EC2 data centers in the US. Figure 4 presents the stationarity in the number of PUTs when considering the timelines of all of these users in aggregate and when considering five popular individual users. In either case, we compare across two weeks the number of PUTs issued in the same hour on the same day of the week, i.e., for every hour, we compute the difference between the number of tweets in that hour and the number of tweets in the same hour the previous week, normalized by the latter value. Aggregate across all users, the count for every hour is within 50% of the count for that hour the previous week, whereas individual users often exhibit 2x and greater variability. The greater stationarity in the aggregate workload thus enables more accurate prediction based on historical workload measurements.

Replication policy. Given these inputs, at the beginning of every epoch, *PMan* determines the replication policy to be used in the next epoch. Since we capture workload in aggregate across all objects with the same access set, *PMan* determines the replication policy separately for every access set, and *SPANStore* employs the same replication strategy for all objects with the same access set. For any particular access set, the replication policy output by *PMan* specifies 1) the set of data centers that maintain copies of all objects with that access set, and 2) at each data center in the access set, which of these copies *SPANStore* should read from and write to when an application VM at that data center issues a GET or PUT on an object with that access set.

Thus, the crux of *SPANStore*'s design boils down to: 1) in each epoch, how does *PMan* determine the replication policy for each access set, and 2) how does *SPANStore* enforce *PMan*-mandated replication policies during its operation, accounting for failures and changes in replication policies across epochs? We next describe separately how *SPANStore* addresses the first question in the eventual consistency and strong consistency cases, and then tackle the second question in the next section.

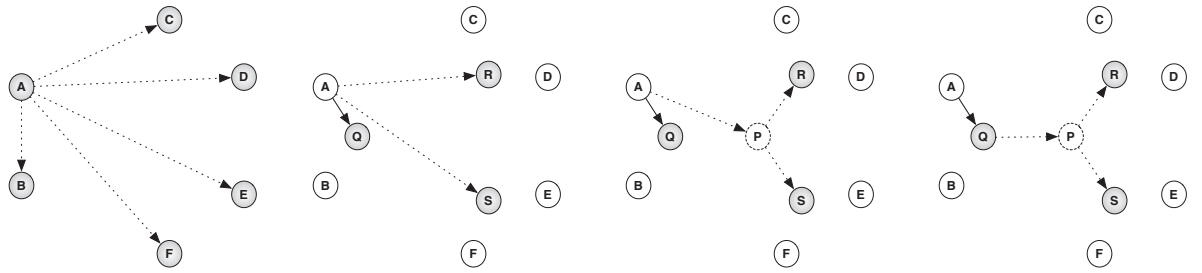
5.2 Eventual consistency

When the application can make do with eventual consistency, *SPANStore* can trade-off costs for storage, PUT/GET requests, and network transfers. To see why this is the case, let us first consider the simple replication policy where *SPANStore* maintains a copy of every object at each data center in that object's access set (as shown in Figure 5(a)). In this case, a GET for any object can be served from the local storage service. Similarly, PUTs can be committed to the local storage service and updates to an object can be propagated to other data centers in the background; *SPANStore* considers a PUT as complete after writing the object to the local storage service because of the durability guarantees offered by storage services. By serving PUTs and GETs from the storage service in the same data center, this replication policy minimizes GET/PUT latencies, the primary benefit of settling for eventual consistency. In addition, serving GETs from local storage ensures that GETs do not incur any network transfer costs.

However, as the size of the access set increases, replicating every object at every data center in the access set can result in high storage costs. Furthermore, as the fraction of PUTs in the workload increase, the costs associated with PUT requests and network transfers increase as more copies need to be kept up-to-date.

To reduce storage costs and PUT request costs, *SPANStore* can store replicas of an object at fewer data centers, such that every data center in the object's access set has a nearby replica that can serve GETs/PUTs from this data center within the application-specified latency SLOs. For example, as shown in Figure 5(b), instead of storing a local copy, data center *A* can issue PUTs and GETs to the nearby replica at *Q*.

However, *SPANStore* may incur unnecessary networking costs if it propagates a PUT at data center *A* by having *A* directly issue a PUT to every replica. Instead, we can capitalize on the discrepancies in pricing across different cloud services (see Section 3) and relay updates to the replicas via another data center that has cheaper pricing for upload bandwidth. For example, in Figure 5(c), *SPANStore* reduces networking costs by having *A* send each of its updates to *P*, which in turn issues a PUT for this update to all the replicas that *A* has not written to directly. In some cases, it may be even more cost-effective to have the replica to which *A*



(a) Full replication minimizes latency

(b) Partial replication reduces costs

(c) Relayed update propagation reduces bandwidth costs

(d) Two-hop relaying may reduce costs further

Figure 5: When eventual consistency suffices, illustration of different replication policies for access set {A, B, C, D, E, F}. In all cases, we show how PUTs from data center *A* are propagated. Shaded circles are data centers that host replicas, and dotted circles represent data centers that propagate updates. Solid arrows correspond to transfers that impact PUT latencies, and dotted arrows represent asynchronous propagation.

commits its PUTs relay updates to a data center that has cheap network pricing, which in turn PUTs the update to all other replicas, e.g., as shown in Figure 5(d).

PMan addresses this trade-off between storage, networking, and PUT/GET request costs by formulating the problem of determining the replication policy for a given access set *AS* as a mixed integer program (shown in Appendix A; for simplicity, we present the formulation without including VM costs). For every data center $i \in AS$, *PMan* chooses $f+1$ data centers (out of all those on which *SPANStore* is deployed) which will serve as the replicas to which i issues PUTs and GETs (*line 27*). *SPANStore* then stores copies of all objects with access set *AS* at all data centers in the union of PUT/GET replica sets (*line 29*).

The integer program used by *PMan* imposes several constraints on the selection of replicas and how updates made by PUT operations propagate. First, whenever an application VM in data center i issues a PUT, *SPANStore* synchronously propagates the update to all the data centers in the replica set for i (*line 31*) and asynchronously propagates the PUT to all other replicas of the object (*line 33*). Second, to minimize networking costs, the integer program used by *PMan* allows for both synchronous and asynchronous propagation of updates to be relayed via other data centers. Synchronous relaying of updates must satisfy the latency SLOs (*lines 13 and 14*), whereas in the case of asynchronous propagation of updates, relaying can optionally be over two hops (*line 15*), as in the example in Figure 5(d). Finally, for every data center i in the access set, *PMan* identifies the paths from data centers j to k along which PUTs from i are transmitted during either synchronous or asynchronous propagation (*lines 35 and 36*).

PMan solves this integer program with the objective of minimizing total cost, which is the sum of storage cost and the cost incurred for serving GETs and PUTs. The storage cost is simply the cost of storing one copy of every object with access set *AS* at each of the replicas

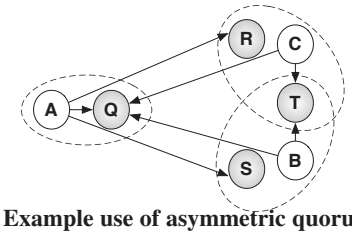


Figure 6: Example use of asymmetric quorum sets. Solid unshaded circles represent data centers in the access set, and shaded circles are data centers that host replicas. Directed edges represent transfers to PUT replica sets, and dashed ovals represent GET replica sets.

chosen for that access set (*line 24*). For every GET operation at data center i, *SPANStore* incurs the price of one GET request at each of i's replicas and the cost of transferring the object over the network from those replicas (*line 20*). In contrast, every PUT operation at any data center i incurs the price of one PUT request each at all the replicas chosen for access set *AS*, and network transfer costs are incurred on every path along which i's PUTs are propagated (*line 22*).

5.3 Strong consistency

When the application using *SPANStore* for geo-replicated storage requires strong consistency of data, we rely on quorum consistency [20]. Quorum consistency imposes two requirements to ensure linearizability. For every data center i in an access set, 1) the subset of data centers to which i commits each of its PUTs—the PUT replica set for i—should intersect with the PUT replica set for every other data center in the access set, and 2) the GET replica set for i should intersect with the PUT replica set for every data center in the access set. The cardinality of these intersections should be greater than the number of failures that the application wants *SPANStore* to tolerate.

In our design, we use asymmetric quorum sets [31] to instantiate quorum consistency as above. With asymmetric quorum sets, the PUT and GET replica sets for

any particular data center can differ. We choose to use asymmetric quorum sets due to the non-uniform geographic distribution of data centers. For example, as seen in Figure 1(a), EC2 data centers have between 2 and 16 other data centers within 200ms of them. Figure 6 shows an example where, due to this non-uniform geographic distribution of data centers, asymmetric quorum sets reduce cost and help meet lower latency SLOs.

The integer program that *PMan* uses for choosing replication policies in the strong consistency setting (shown in Appendix B) mirrors the program for the eventual consistency case in several ways: 1) PUTs can be relayed via other data centers to reduce networking costs (*lines 10 and 11, and 27–30*), 2) storage costs are incurred for maintaining a copy of every object at every data center that is in the union of the GET and PUT replica sets of all data centers in the access set (*lines 7, 18, and 32*), and 3) for every GET operation at data center i, one GET request's price and the price for transferring a copy of the object over the network is incurred at every data center in i's GET replica set (*line 14*).

However, the integer program for the strong consistency setting does differ from the program used in the eventual consistency case in three significant ways. First, for every data center in the access set, the PUT and GET replica sets for that data center may differ (*lines 5 and 6*). Second, *PMan* constrains these replica sets so that every data center's PUT and GET replica sets have an intersection of at least $2f + 1$ data centers with the PUT replica set of every other data center in the access set (*lines 20–26*). Finally, PUT operations at any data center i are propagated only to the data centers in i's PUT replica set, and these updates are propagated via at most one hop (*line 16*).

6 *SPANStore* dynamics

Next, we describe *SPANStore*'s operation in terms of the mechanisms it uses to execute PUTs and GETs, to tolerate failures, and to handle changes in the application's workload across epochs. First, we discuss the metadata stored by *SPANStore* to implement these mechanisms.

6.1 Metadata

At every data center, *SPANStore* stores in-memory metadata across all the VMs that it deploys in that data center. At the beginning of an epoch, *PMan* computes the new replication policy to use in that epoch, and it transmits to every data center the replication policy information needed at that data center. A data center A needs, for every access set AS that contains A, the PUT and GET replica sets to be used by A for objects with access set AS. Whenever the application issues a PUT for a new object at data center A, it needs to specify the access set for that object. *SPANStore* then inserts an (object name → access set) mapping into the in-memory metadata at every data center in the access set.

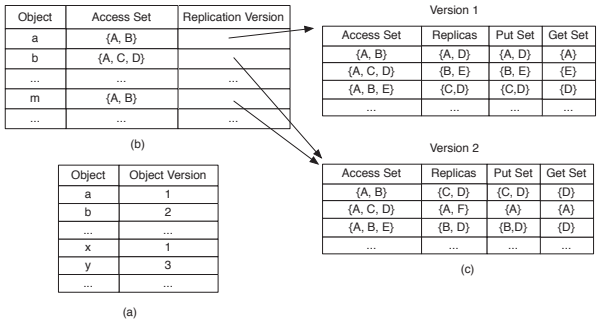


Figure 7: At any data center A, *SPANStore* stores an (a) in-memory version mapping for objects stored at A. If the application is deployed at A, *SPANStore* also stores (b) the access set mapping for objects whose access set includes A, and (c) replication policy versions for different epochs.

As we describe later in Section 6.4, when serving the first operation for an object in a particular epoch, *SPANStore* needs to account for both the replication policy currently in use for that object and the new replication policy computed by *PMan* for the current epoch. Therefore, we store both current and historical versions of the (access set → replica sets) mapping. As shown in Figure 7, the access set mapping for an object includes the replication policy version that currently applies for that object. *SPANStore* eventually destroys an old replication policy when no object is using it.

In addition, at any data center, *SPANStore* also stores an in-memory version mapping for all objects stored in the storage service at that data center. Note that, at any data center A, the set of objects stored at A can differ from the set of objects whose access set includes A.

6.2 Serving PUTs and GETs

Any application uses *SPANStore* by linking to a library that implements *SPANStore*'s protocol for performing PUTs and GETs. If the application configures *SPANStore* to provide eventual consistency, the library looks up the local metadata for the current PUT/GET replica set for the queried object. Upon learning which replicas to use, the *SPANStore* library issues PUT/GET requests to the storage services at those replicas and returns an ACK/the object's data to the application as soon as it receives a response from any one of those replicas.

When using strong consistency, *SPANStore* uses two phase locking (2PL) to execute PUT operations. First, the *SPANStore* library looks up the object's metadata to discover the set of replicas that this data center must replicate the object's PUTs to. The library then acquires locks for the object at all data centers in this replica set. If it fails to acquire any of the locks, it releases the locks that it did acquire, backs off for a random period of time,

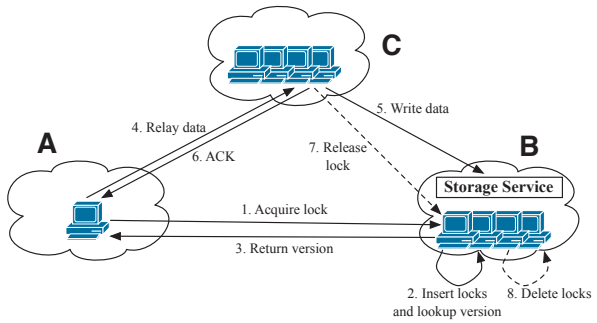


Figure 8: Illustration of *SPANStore*'s two-phase locking protocol. Solid lines impact PUT latency, whereas operations along dashed lines are performed asynchronously.

and then retries. Once the library acquires locks at all data centers in the PUT replica set, it writes the new version of the object to the storage services at those data centers and releases the locks.

The straightforward implementation of this protocol for executing PUTs can be expensive. Consider a VM at data center A performing a PUT operation on a replica set that includes data center B. The VM at A can first send a request to a VM at B to acquire the lock and to obtain the version of the object stored at data center B. The VM at A can then send the data for the object being updated to the VM at B (possibly via another data center C that relays the data). The VM at B can write the data to the local data center, release the lock, and update the in-memory version mapping for the object. However, this requires *SPANStore*'s VMs to receive object data from remote data centers. To meet the bandwidth demands of doing so, we will need to provision a large number of VMs, thus inflating cost.

Instead, we employ a modified 2PL protocol as shown in Figure 8. As before, the VM at A communicates with a VM at B to acquire the lock and obtain the version number of B's copy of the object. To acquire the lock for object o, the VM at B inserts two objects into the local in-memory metadata cluster—L_o^T that times out after 5 seconds, and L_o^U that does not have any timeout. Once it acquires the lock, the VM at A directly issues a PUT to the storage service at data center B, rather than asking the VM at B to perform this PUT. While writing the object, we prepend the version number to the object's data. Once the PUT to the storage service is complete, *SPANStore* lazily requests a VM at B to release the lock by deleting both L_o^T and L_o^U, and to also update the version number stored in memory for the updated object.

In the case where the Internet path from A to B fails after the new version of the object has been written to B or if the VM at A that is performing the PUT fails before it releases the locks, the VM at A cannot explicitly delete L_o^T and L_o^U at B, yet L_o^T will timeout. When a VM at B receives a request to lock object o in the future and finds that L_o^T is absent but L_o^U is present, it issues a GET for the object to the local storage service and updates the in-memory version mapping for o to the version prepended to the object's data.

This modified 2PL protocol eliminates the need for *SPANStore*'s VMs to send or receive object data, other than when PUTs are relayed via another data center. As a result, our 2PL protocol is significantly more cost-effective than the strawman version, e.g., a small VM on EC2 can handle 105 locking operations per second, but can only receive and write to the local storage service 30 100KB objects per second.

In the strong consistency setting, serving GETs is simpler than serving PUTs. When an application VM at a particular data center issues a GET, the *SPANStore* library on that VM looks up the GET replica set for that object in the local in-memory metadata, and it then fetches the copy of the requested object from every data center in that set. From the retrieved copies of the object, the library then returns the latest version of the object to the application. We could reduce networking costs by first querying the in-memory metadata at every replica for the current version of the object at that data center, and then fetching a copy of the object only from the nearest data center which has the latest version. However, for small objects whose data can fit into one IP packet, querying the version first and then fetching the object will double the wide-area RTT overhead.

6.3 Fault tolerance

SPANStore needs to respect the application's fault-tolerance needs, which *PMan* accounts for when it determines replication policies. In the eventual consistency case, every data center in the access set is associated with $f+1$ replicas, and *SPANStore* considers a GET/PUT as complete once the operation successfully completes on any one of the replicas chosen by *PMan*. It suffices for *SPANStore* to consider a PUT as complete even after writing the update to a single replica because of the durability guarantees offered by the storage services that *SPANStore* builds upon. Every storage service replicates objects across servers within a data center to ensure that it is very unlikely to lose an update committed by a PUT.

When configured for strong consistency, *SPANStore* relies on the fault tolerance offered by our use of quorum sets. An intersection of at least $2f+1$ data centers between the PUT replica set of every data center and the PUT and GET replica sets of every other data center in the access set enables *SPANStore* to be resilient to up to f failures [30]. This is because, even if every data center is unable to reach a different set of f replicas, the intersection larger than $2f+1$ between PUT-PUT replica set pairs and GET-PUT replica set pairs ensures that every pair of data centers in the access set has at least one

common replica that they can both reach.

Thus, in the strong consistency setting, the *SPANStore* library can tolerate failures as follows when executing PUT and GET operations. At any data center A, the library initiates a PUT operation by attempting to acquire the lock for the specified object at all the data centers in A's PUT replica set for this object. If the library fails to acquire the lock at some of these data centers, for every other data center B in the object's access set, the library checks whether it failed to acquire the lock at at most f replicas in B's PUT and GET replica sets for this object. If this condition is true, the library considers the object to be successfully locked and writes the new data for the object. If not, the PUT operation cannot be performed and the library releases all acquired locks.

The *SPANStore* library executes GETs in the strong consistency setting similarly. To serve a GET issued at data center A, the library attempts to fetch a copy of the object from every data center in A's GET replica set for the specified object. If the library is unsuccessful in fetching copies of the object from a subset S of A's replica set, it checks to see whether S has an intersection of size greater than f with the PUT replica set of any other data center in the object's access set. If yes, the library determines that the GET operation cannot be performed and returns an error to the application.

6.4 Handling workload changes

The replication policy for an access set can change when there is a significant change in the aggregate workload estimated for objects with that access set. When *PMan* mandates a new replication policy for a particular access set at the start of a new epoch, *SPANStore* switches the configuration for an object with that access set at the time of serving the first GET or PUT request for that object in the new epoch. *SPANStore* can identify the first operation on an object in the new epoch based on a version mismatch between the replication policy associated with the object and the latest replication policy.

In a new epoch, irrespective of whether the first operation on an object is a GET or a PUT, the *SPANStore* library on the VM issuing the request attempts to acquire locks for the object at all data centers in the object's access set. In the case of a PUT, *SPANStore* commits the PUT to the new PUT replica set associated with the object. In the case of a GET, *SPANStore* reads copies of the object from the current GET set and writes the latest version among these copies to the new PUT set. *SPANStore* then switches the replication policy for the object to the current version at all data centers in its access set. Thereafter, all the PUTs and GETs issued for the object can be served based on the new replication policy.

This procedure for switching between replication policies leads to latency SLO violations and cost overhead (due to the additional PUTs incurred when the first request for an object in the new epoch is a GET). However, we expect the SLO violations to be rare and the cost overhead to be low since only the first operation on an object in a new epoch is affected.

7 Implementation

We have implemented and deployed a prototype of *SPANStore* that spans all the data centers in Amazon S3, Microsoft Azure, and Google Cloud Storage. Our implementation has three components—1) *PMan*, 2) a client library that applications can link to, and 3) an XMLRPC server that is run in every VM run by *SPANStore*. In addition, in every data center, we run a memcached cluster across all *SPANStore* instances in that data center to store *SPANStore*'s in-memory metadata.

PMan initially bootstraps its state by reading in a configuration file that specifies the application's latency, consistency, and fault tolerance requirements as well as the parameters (latency distribution between data centers, and prices for resources at these data centers) that characterize *SPANStore*'s deployment. To determine optimal replication policies, it then periodically invokes the CPLEX solver to solve the formulation (Appendix A or Appendix B) appropriate for the application's consistency needs. *PMan* also exports an XMLRPC interface to receive workload and latency information from every data center at the end of every epoch.

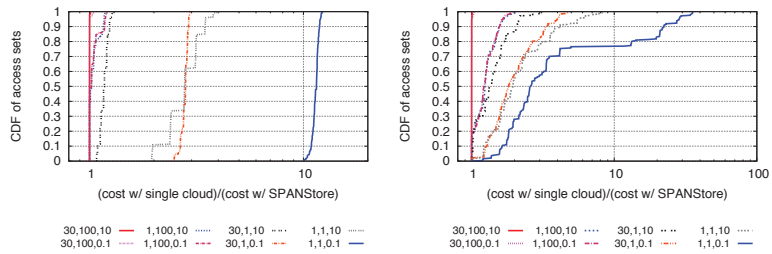
The client library exports two methods: *GET(key)* and *PUT(key, value, [access_set])*. The library implements these methods as per the protocols described in Section 6. To lookup the metadata necessary to serve GET and PUT requests, the library uses DNS to discover the local memcached cluster.

The XMLRPC server exports three interfaces. First, it exports a *LOCK(key)* RPC for the client library to acquire object-specific locks. Second, its *RELAY(key, data, dst)* enables the library or a *SPANStore* VM to indirectly relay a PUT in order to reduce network bandwidth costs. Lastly, the XMLRPC server receives replication policy updates from *PMan*.

In addition, the XMLRPC server 1) gathers statistics about the application's workload and reports this information to *PMan* at the end of every epoch, and 2) exchanges heartbeats and failure information with *SPANStore*'s VMs in other data centers. Both the client library and the XMLRPC server leverage open-source libraries for issuing PUT and GET requests to the S3, Azure, and GCS storage services.

8 Evaluation

We evaluate *SPANStore* from four perspectives: the cost savings that it enables, the cost-optimality of its replication policies, the cost necessary for increased fault toler-



(a) Strong, 90%ile G-SLO=250ms (b) Strong, 90%ile G-SLO=100ms

Figure 9: Cost with *SPANStore* compared to that possible using the data centers of a single cloud service. Legend indicates GET:PUT ratio, average object size (in KB), and overall data size (in TB).

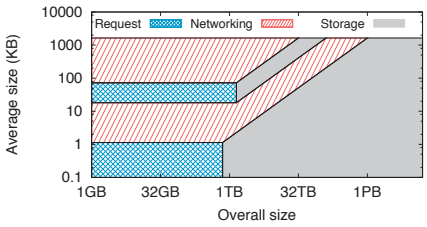


Figure 10: Variation in the dominant component of *SPANStore*'s cost as a function of the application's workload.

ance, and the scalability of *PMan*. Here, we show results for the case where the application is deployed across EC2's data centers, and *SPANStore* is deployed across the storage services offered by S3, Azure, and GCS. Our results are qualitatively similar when we consider application deployments on Azure or GCE.

8.1 Cost savings

Workload and SLOs. To evaluate the cost savings enabled by *SPANStore*, we consider all 18 combinations of a) GET:PUT ratios of 1, 10, and 30, b) average object sizes of 1 KB and 100 KB, and c) aggregate data size of 0.1 TB, 1 TB and 10 TB. Our choice of these workload parameters is informed by the GET:PUT ratio of 30:1 and objects typically smaller than 1 KB seen in Facebook's workload [14]. For brevity, we omit here the results for the intermediate cases where GET:PUT ratio is 10 and overall data size is 1 TB. In all workload settings, we fix the number of GETs at 100M and compute cost over a 30 day period.

When analyzing the eventual consistency setting, we consider two SLOs for the 90^{th} percentile values of GET and PUT latencies—100 ms and 250 ms; 250 ms is the minimum SLO possible with a single replica for every object and 100 ms is less than half of that. In the strong consistency case, we consider two SLO combinations for the 90^{th} percentile GET and PUT latencies—1) 100ms and 830ms, and 2) 250 ms and 830 ms; 830 ms is the minimum PUT SLO if every object was replicated at all data centers in its access set. Since the cost savings enabled by *SPANStore* follow similar trends in the eventual consistency and strong consistency settings, we show results only for the latter scenario for brevity.

Comparison with single-cloud deployment. First, we compare the cost with *SPANStore* with the minimum cost required if we used only Amazon S3's data centers for storage. Figure 9 shows that *SPANStore*'s use of multiple cloud services consistently offers significant cost savings when the workload includes 1 KB objects. The small object size makes networking cost negligible in comparison to PUT/GET requests costs, and hence, *SPANStore*'s multi-cloud deployment helps because PUT and GET requests are priced 50x and 4x cheaper on Azure as compared to on S3. When the average object size is 100KB, *SPANStore* still offers cost benefits for a sizeable fraction of access sets when the PUT/GET ratio is 1 and overall size size is small. In this case, since half of the workload (i.e., all PUT operations) require propagation of updates to all replicas, *SPANStore* enables cost savings by exploiting discrepancies in network bandwidth pricing across cloud services. Furthermore, when the total data size is 10 TB, *SPANStore* reduces storage costs by storing fewer copies of every object.

Comparison with fixed replication policies. We also compare the cost incurred when using *SPANStore* with that imposed by two fixed replication policies: *Everywhere* and *Single*. With the *Everywhere* policy, every object is replicated at every data center in the object's access set. With the *Single* replication policy, any object is stored at one data center that minimizes cost among all single replica alternatives that satisfy the PUT and GET latency SLOs. We consider the same workloads as before, but ignore the cases that set the SLO for the 90^{th} percentile GET latency to 100ms since that SLO cannot be satisfied when using a single replica.

In Figure 11(left), we see that *SPANStore* significantly outdoes *Everywhere* in all cases except when GET:PUT ratio is 30 and average object size is 100KB. On the other hand, in Figure 11(right), we observe a bi-modal distribution in the cost savings as compared to *Single* when the object size is small. We find that this is because, for all access sets that do not include EC2's Sydney data center, using a single replica (at some data center on Azure) proves to be cost-optimal; this is again because the lower PUT/GET costs on Azure compensate for the increased network bandwidth costs. When the GET:PUT ratio is 1 and the average object size is 100KB, *SPANStore* saves cost compared to *Single* by judiciously combining the use of multiple replicas.

Dominant cost analysis. Finally, we analyze how the dominant component of *SPANStore*'s cost varies based on the input workload. For one particular access set,

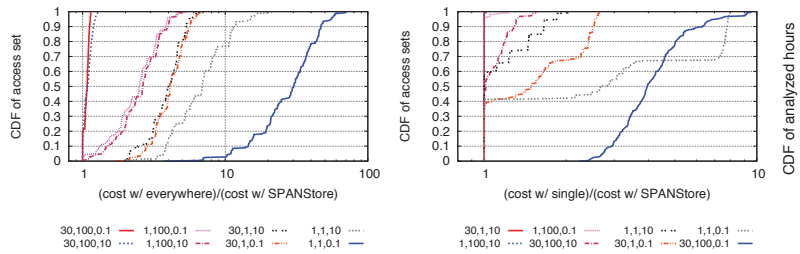


Figure 11: Cost savings enabled by *SPANStore* compared to *Everywhere* (left) and *Single* (right) replication policies. Legend indicates GET:PUT ratio, average object size (in KB), and overall data size (in TB).

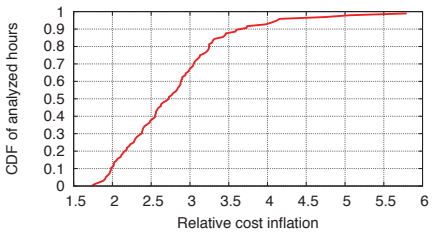


Figure 12: Cost inflation when predicting workload using individual objects compared with aggregate workload prediction.

Figure 10 shows which among network, storage, and request cost dominates the total cost when varying average object size from 0.1 KB to 1 MB and total data size from 1 GB to 1 PB. Here, we use a GET:PUT ratio of 30 and set GET and PUT SLOs as 250ms and 830ms with the need for strong consistency, but the takeaways are similar in other scenarios.

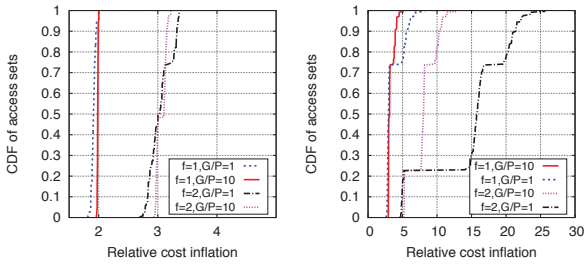
When both the average object size and the total data size are small, costs for PUT and GET requests initially dominate, but network transfer costs increase as the average object size increases. However, as the average object size increases further, *SPANStore* transitions to storing data locally at every data center in the access set. This eliminates the need to use the network when serving GETs, thus making request costs the dominant component of cost again. However, network transfers cannot be completely eliminated due to the need to propagate updates to all replicas. Therefore, network transfer costs again begin to dominate for large object sizes.

When the total data size is large, *SPANStore* stores a single copy of all data when the average object size is small and storage cost is dominant. As the average object size increases, network transfer costs initially exceed the storage cost. However, as network costs continue to increase, *SPANStore* begins to store multiple replicas of every object so that many GETs can be served from the local data center. This reduces network transfer costs and makes storage cost dominant again. Eventually, as the average object size increases further, even if *SPANStore* stores a replica of every object at every data center in its access set, the need to synchronize replicas results in networking costs exceeding storage costs.

8.2 Impact of aggregation of objects

SPANStore's cost-effectiveness critically depends on its ability to estimate the application's workload. As discussed previously in Section 5, we choose to characterize workload in aggregate across all objects with the same access set due to the significantly greater stationarity that exists in aggregate workloads as compared to the workloads of individual objects. Here, we quantify the cost benefits enabled by this design decision.

From the Twitter dataset previously described in Section 5, we randomly choose 10K users. Since the dataset only specifies the times at which these users post tweets, we generate the times at which they check their Twitter timelines based on Twitter's usage statistics [4]. Considering hour-long epochs, we estimate the workload in a particular hour as being the same as that in the same hour in the previous week. We consider workload estimation at two granularities: 1) per-object, and 2) in aggregate across all objects with the same access set.



(a) Eventual,P-SLO=250ms (b) Strong,P-SLO=830ms

Figure 13: Cost inflation when tolerating f failures compared to the cost with $f = 0$. SLOs are on 90^{th} percentile latencies and G-SLO=250ms.

Figure 12 shows the cost inflation arising from estimating workloads on a per-object granularity as compared to the cost when estimating aggregate workloads. The high inflation is due to the significant variation seen in any individual user's usage of Twitter. Since most users rarely post or access Twitter, the use of the per-object workload estimator causes *SPANStore* to typically choose EC2's data centers as replicas since they have lower storage cost. However, this often turns out to be a mis-prediction of the workload, and when a user does post or access her timeline, *SPANStore* has to incur greater request costs that necessary by serving these requests from EC2's data centers. The greater accuracy of estimating workloads in aggregate enables *SPANStore* to replicate data in a more cost-effective manner.

8.3 Cost for fault tolerance

To tolerate failures, *SPANStore* provisions more data centers to serve as replicas. As expected, this results in higher cost. In Figure 13, we show the cost inflation for

various levels of fault-tolerance as compared to the cost when *SPANStore* is provisioned to not tolerate any failures. For most access sets, the cost inflation is roughly proportional to $f+1$ in the eventual consistency scenario and proportional to $2f+1$ in the strong consistency case. However, the need for fault tolerance increases cost by a factor greater than $f+1/2f+1$ for many other access sets. This is because, as f increases, the need to pick a greater number of replicas within the latency SLO for every data center in the access set requires *SPANStore* to use as replicas data centers that have greater prices for GET/PUT requests.

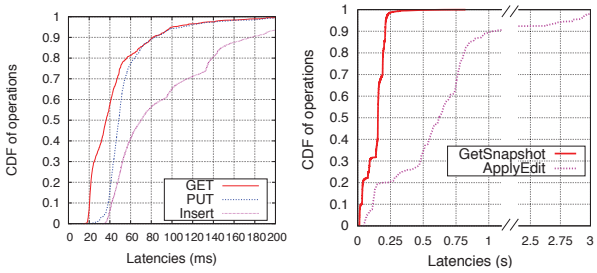
In addition, in both the eventual consistency and strong consistency scenarios, the need to tolerate failures results in higher cost inflation when the GET:PUT ratio is low as compared to when the GET:PUT ratio is high. This is because, when the GET:PUT ratio is low, *SPANStore* can more aggressively reduce costs when $f=0$ by indirectly propagating updates to exploit discrepancies in network bandwidth pricing.

8.4 Scalability of PlacementManager

Finally, we evaluate the scalability of *PMan*, the one central component in *SPANStore*. At the start of every epoch, *PMan* needs to compute the replication policy for all access sets; there are 2^N access sets for an application deployed across N data centers. Though the maximum number of data centers in any one cloud service is currently 8 (in EC2), we test the scalability of *PMan* in the extreme case where we consider all the data centers in EC2, Azure, and GCE as being in the same cloud service on which the application is deployed. On a cluster of 16 servers, each with two quad-core hyperthreaded CPUs, we find that we can compute the replication policy within an hour for roughly 33K access sets. Therefore, as long as the application can estimate its workload for the next epoch an hour before the start of the epoch (which is the case with our current way of estimation based on the workload in the same hour in the previous week), our *PMan* implementation can tackle cases where the application is deployed on 15 or fewer data centers. For more geographically distributed application deployments, further analysis is necessary to determine when the new aggregate workload for a access set will not cause a significant change in the optimal replication policy for that set. This will enable *PMan* to only recompute the replication policy for a subset of access sets.

9 Case studies

We have used our *SPANStore* prototype as the backend storage for two applications that can benefit from geo-replicated storage: 1) Retwis [5] is a clone of the Twitter social networking service, which can make do with eventual consistency, and 2) ShareJS [6] is a collaborative document editing webservice, which requires strongly consistent data. To support both applications, we add a few operations that are wrappers over PUTs and GETs, such as "append to a set", "get i^{th} element from a set", and "increment a global counter". We deploy both applications across all of EC2's data centers.[2]



(a) (b)

Figure 14: CDF of operation latencies in (a) Retwis and (b) ShareJS applications.

Retwis. At each EC2 data center, we run emulated Retwis clients which repeatedly make two types of operations: *Post* operations represent a user posting a tweet, and *GetRange* operations fetch the last 100 tweets in a user's timeline (the set of tweets posted by those that the user follows). We set the ratio of number of *Post* operations to number of *GetRange* operations as 0.1, i.e., on average, every user makes a post every 10 times that she checks her timeline. A GetRange operation issues 1) a GET to fetch the user's timeline object, and then 2) GETs for the post IDs in the specified range in the timeline. A Post operation executes the following sequence: 1) a PUT to store the post, 2) a GET to fetch the list of the user's followers, and 3) an Insert operation to append the post's ID to the timeline object of each of the user's followers. The Insert operation on an object fetches the object by issuing GET, modifies it locally, and then writes back the updated object with a PUT.

We run Retwis with a randomly generated social network graph comprising 10K users, where every user follows a randomly chosen subset of 200 users [2]. For every user, we assign the data center from which the user is served at random. We run this workload on *SPANStore* configured with the latency SLOs specifying that the 90^{th} percentile PUT and GET latencies should be less than 100ms. Figure 14(a) shows that *SPANStore* satisfies these latency goals as over 90% of all operations are within the specified SLOs.

ShareJS. We run the ShareJS application with a similar setup; we consider 1K documents, each of which is associated with a randomly chosen access set. At each EC2 data center, we then run a ShareJS client which iteratively issues *GetSnapshot* and *ApplyEdit* operations on

[2]Though ShareJS is not amenable to distributed deployment, we modify it suitably so as to enable every user to be served from her closest EC2 data center.

randomly chosen documents whose access set includes the local data center. These operations correspond to the client fetching a document and applying an edit to a document, respectively. Since we need strong consistency in this case, we use SLOs on the 90^{th} percentile PUT and GET latencies as 830ms and 250ms. Note that the GetSnapshot operation directly maps to a GET, but the ApplyEdit operation requires the application to issue a GET for the latest version of the document and then issue a PUT to incorporate the edit.

Figure 14(b) shows the distribution of latencies incurred for the GetSnapshot and ApplyEdit operations. We can see that more than 95% of GetSnapshot operations satisfy the latency SLO, and a vast majority of ApplyEdit operations are within the SLO, given that an ApplyEdit includes a GET followed by a PUT. The small minority of operations that exceed the latency bounds are due to contention between concurrent updates to the same document; when a writer fails on the two-phase locking operation, it retries after 2 seconds.

10 Related work

Evaluating benefits of cloud deployments. Some recent work addresses when to use cloud services, in particular examining how and when to migrate applications from the application provider's data center to a cloud service [21, 33, 36]. While these efforts consider issues such as cost and wide-area latency like we do, none of them seek to provide a unified view to geo-replicated storage. Some others compare the performance offered by various cloud services [23, 7]. However, these efforts do not consider issues such as cost and consistency.

Using multiple cloud services. Several previously developed systems (e.g., RACS [9], SafeStore [22], DEPSKY [16], and MetaStorage [15]) have considered the use of multiple service providers for storing data. However, all of these systems focus on issues pertaining to availability, durability, vendor lock-in, performance, and consistency. Unlike *SPANStore*, none of these systems seek to minimize cost by exploiting pricing discrepancies across providers.

Other complementary efforts have focused on utilizing compute resources from multiple cloud providers. AppScale enables portability of applications across cloud services [18], but without any attempt to minimize cost. Conductor [34] orchestrates the execution of MapReduce jobs across multiple cloud providers in a manner that minimizes cost. In contrast to these systems, *SPANStore* only focuses on unifying the use of storage resources across multiple providers.

Optimizing costs and scalable storage. Minerva [12], Hippodrome [13], scc [28] and Rome [35] automate the provisioning of cost-effective storage configurations while accounting for workload characterizations. Though these systems share a similar goal as ours, their setting is restricted to storage clusters deployed within a data center. *SPANStore* provides geo-replicated storage, and so its deployment strategies must account for inter-data center latencies and multiple administrative domains. Farsite [10] provides scalable storage in an untrusted environment and shares some techniques with *SPANStore*, e.g., lazily propagating updates. However, Farsite does not focus on optimizing cost.

Low-latency geo-replicated storage with improved consistency. Numerous systems strive to provide fast performance and stronger-than-eventual consistency across the wide area. Recent examples include Walter [32], Spanner [19], Gemini [24], COPS [26], and Eiger [27]. Given that wide-area storage systems cannot simultaneously guarantee both the strongest forms of consistency and low latency [29], these systems strive to push the envelope along one or both of those dimensions. However, none of these systems focus on minimizing cost while meeting performance goals, which is our primary goal. In fact, most of these systems replicate all data to all data centers, and all data centers are assumed to be under one administrative domain. We believe that *SPANStore* can be adapted to achieve the consistency models and performance that these systems offer at the lower costs that *SPANStore* provides.

11 Conclusions

Though the number of cloud storage services available across the world continues to increase, the onus is on application developers to replicate data across these services. We develop *SPANStore* to export a unified view of geographically distributed storage services to applications and to automate the process of trading off cost and latency, while satisfying consistency and fault-tolerance requirements. Our design of *SPANStore* achieves this goal by spanning data centers of multiple cloud providers, by judiciously determining replication policies based on workload properties, and by minimizing the use of compute resources. We have deployed *SPANStore* across Amazon's, Microsoft's, and Google's cloud services and find that it can offer significant cost benefits compared to simple replication policies.

Acknowledgments

We thank the anonymous reviewers and our shepherd Lidong Zhou for their valuable feedback on earlier drafts of this paper. This work was supported in part by a NetApp Faculty Fellowship and by the National Science Foundation under grant CNS-1150219.

References

[1] Amazon S3. http://aws.amazon.com/s3.

[2] By the numbers: 31 amazing Twitter stats. http://expandedramblings.com/index.php/march-2013-by-the-numbers-a-few-amazing-twitter-stats.

[3] Google cloud storage. http://cloud.google.com/storage.

[4] Infographic: Who is using Twitter, how often, and why? http://www.theatlantic.com/technology/archive/2011/07/infographic-who-is-using-twitter-how-often-and-why/241407/.

[5] Retwis. http://retwis.antirez.com.

[6] ShareJS. https://github.com/josephg/ShareJS/.

[7] VMware vFabric Hyperic. http://www.vmware.com/products/datacenter-virtualization/vfabric-hyperic/.

[8] Windows Azure. http://www.microsoft.com/windowsazure.

[9] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A case for cloud storage diversity. In *SOCC*, 2010.

[10] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, 2002.

[11] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, and A. Wolman. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, 2010.

[12] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. A. Becker-Szendy, R. A. Golding, A. Merchant, M. Spasojevic, A. C. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM ToCS*, 2001.

[13] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. C. Veitch. Hippodrome: Running circles around storage administration. In *FAST*, 2002.

[14] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[15] D. Bermbach, M. Klems, S. Tai, and M. Menzel.

[16] A. Bessani, M. Correia, B. Quaresma, F. Andre, and P. Sousa. DEPSKY: Dependable and secure storage in a cloud-of-clouds. In *EuroSys*, 2011.

[17] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *SoCC*, 2010.

[18] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. AppScale: Scalable and open AppEngine application development and deployment. In *CloudComp*, 2009.

[19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

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

[21] M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani. Cloudward bound: Planning for beneficial migration of enterprise applications to the cloud. In *SIGCOMM*, 2010.

[22] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A durable and practical storage system. In *USENIX ATC*, 2007.

[23] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing public cloud providers. In *IMC*, 2010.

[24] C. Li, D. Porto, A. Clement, R. Rodrigues, N. Preguia, and J. Gehrke. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, 2012.

[25] R. Li, S. Wang, H. Deng, R. Wang, and K. C.-C. Chang. Towards social user profiling: Unified and discriminative influence model for inferring home locations. In *KDD*, 2012.

[26] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[27] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, 2013.

[28] H. V. Madhyastha, J. C. McCullough, G. Porter, R. Kapoor, S. Savage, A. C. Snoeren, and A. Vahdat. scc: Cluster storage provisioning informed

by application characteristics and SLAs. In *FAST*, 2012.

[29] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, convergence. Technical report, Univ. of Texas, 2011.

[30] D. Malkhi and M. Reiter. Byzantine quorum systems. In *STOC*, 1997.

[31] J.-P. Martin, L. Alvisi, and M. Dahlin. Small byzantine quorum systems. In *DSN*, 2002.

[32] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[33] B. C. Tak, B. Urgaonkar, and A. Sivasubramaniam. To move or not to move: The economics of cloud computing. In *HotCloud*, 2011.

[34] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with Conductor. In *NSDI*, 2012.

[35] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *IWQoS*, 2001.

[36] T. Wood, E. Cecchet, K. Ramakrishnan, P. Shenoy, J. van der Merwe, and A. Venkataramani. Disaster recovery as a cloud service: Economic benefits & deployment challenges. In *HotCloud*, 2010.

A Replication policy selection for eventual consistency

Inputs:

T = Duration of epoch

AS = Set of data centers that issue PUTs and GETs

f = Number of failures that *SPANStore* should tolerate

SLO, p = SLO on p^{th} percentile of PUT/GET latencies

$L^C_{ij} = p^{th}$ percentile latency between VMs in data centers i and j

$L^S_{ij} = p^{th}$ percentile latency between a VM in data center i and the storage service in data center j

$PUTs_i, GETs_i$ = Total no. of PUTs and GETs issued at data center i across all objects with access set AS

$Size^{avg}, Size^{total}$ = Avg. and total size of objects with access set AS

$Price^{GET}_i, Price^{PUT}_i, Price^{Storage}_i$ = Prices at data center i per GET, per PUT, and per byte per hour of storage

$Price^{Net}_{ij}$ = Price per byte of network transfer from data center i to j

Variables:

$\forall i \in AS, j$ s.t. $L^S_{ij} \leq SLO : R_{ij}$ *// whether j is a replica to which i issues PUTs and GETs; only permitted if a VM at i can complete a GET/PUT on the storage service at j within the SLO*

$\forall i \in AS, j, k$ s.t. p^{th} percentile of $L^C_{ij} + L^S_{jk} \leq SLO : P^S_{ijk}$ *// whether i synchronously forwards its PUTs to k via j; only permitted if a VM at i can forward data to the storage service at j via a VM at k within the SLO*

$\forall i \in AS, j, k, m : P^A_{ijkm}$ *// whether i's PUTs are asynchronously forwarded to m via j and k*

$\forall i \in AS, j, k$ s.t. $j \neq k : F_{ijk}$ *// whether PUTs from i are relayed to k via j*

$\forall j : C_j$ *//whether j is a replica*

Objective: Minimize (Cost for GETs + Cost for PUTs + Storage cost)

// GETs issued at i fetch data only from i's replicas

Cost for GETs = $\sum_i GETs_i \cdot (\sum_j (R_{ij} \cdot (Price^{GET}_j + Price^{Net}_{ji} \cdot Size^{avg})))$

// Every PUT is propagated to all replicas

Cost for PUTs = $\sum_i PUTs_i \cdot (\sum_j (C_j \cdot Price^{PUT}_j) + \sum_{j,k} (F_{ijk} \cdot Price^{Net}_{jk} \cdot Size^{Avg}))$

// every replica stores one copy of every object

Storage cost = $\sum_j (C_j \cdot Price^{Storage}_j \cdot Size^{Total} \cdot T)$

Constraints:

// Every data center in the access set has $f+1$ GET/PUT replicas

$\forall i \in AS : \sum_j R_{ij} = f + 1$

// j is a replica if it is a GET/PUT replica for any i in the access set

$\forall j : (C_j = 1)$ iff $(\sum_{i \in AS} R_{ij} > 0)$ [3]

// i's PUTs must be synchronously forwarded to k iff k is one of i's replicas

$\forall i \in AS, k : (R_{ik} = 1)$ iff $(\sum_j P^S_{ijk} > 0)$

// For every data center in access set, its PUTs must reach every replica

$\forall i \in AS, m : C_m = \sum_j (P^S_{ijm} + \sum_k P^A_{ijkm})$

// PUTs from i can be forwarded over the path from j to k as part of either synchronous or asynchronous forwarding

$\forall i, j, k$ s.t. $i \neq j : (F_{ijk} = 1)$ iff $(P^S_{ijk} + \sum_m (P^A_{ijkm} + P^A_{imjk}) > 0)$

$\forall i, k : (F_{iik} = 1)$ iff $(P^S_{iik} + \sum_m P^S_{ikm} + \sum_{m,n} P^A_{ikmn} > 0)$

B Replication policy selection for strong consistency

Inputs:

Same as the inputs in the eventual consistency scenario, except

SLO_{GET}, SLO_{PUT} = SLOs on GET and PUT latencies

Variables:

$\forall i \in AS, j : PR_{ij}$ *// whether j is in i's PUT replica set*

[3] We implement constraints of the form $(X = 1)$ *iff* $(Y > 0)$ as $X \leq Y \leq max(Y) \cdot X$.

6: $\forall i \in AS, j$ s.t. $L^S_{ij} \leq SLO_{GET} : GR_{ij}$ *// whether j is in i's GET replica set; only permitted if a VM at i can complete a GET on the storage service j within the SLO*

7: $\forall j : C_j$ *// whether j is a replica*

8: $\forall i, j \in AS, k : U^P_{ijk}$ *// whether k is in the union of i's and j's PUT replica sets*

9: $\forall i, j \in AS, k : U^G_{ijk}$ *// whether k is in the union of i's GET replica set and j's PUT replica set*

10: $\forall i \in AS, j, k$ s.t. p^{th} percentile of $L^C_{ij} + L^C_{ik} + L^S_{kj} \leq SLO_{PUT} : F_{ikj}$ *// whether i forwards its PUTs to j via k; only permitted if a VM at i can acquire the lock on a VM at j and then forward data to the storage service at j via a VM at k within the SLO*

11: $\forall i \in AS, k : R_{ik}$ *// whether k serves as a relay from i to any of i's PUT replicas*

12: **Objective:** Minimize (Cost for GETs + Cost for PUTs + Storage cost)

13: *// at each of i's GET replicas, every GET from i incurs one GET request's cost and the network cost of transferring the object from the replica to i*

14: Cost for GETs = $\sum_i GETs_i \cdot (\sum_j (GR_{ij} \cdot (Price^{GET}_j + Price^{Net}_{ji} \cdot Size^{avg})))$

15: *// every PUT from i incurs one PUT request's cost at each of i's PUT replicas and the network cost of transferring the object to these replicas*

16: Cost for PUTs = $\sum_i PUTs_i \cdot \sum_k (R_{ik} \cdot Price^{Net}_{ik} \cdot Size^{avg} + \sum_j (F_{ikj} \cdot (Price^{PUT}_j + Price^{Net}_{kj} \cdot Size^{avg})))$

17: *// every replica stores one copy of every object*

18: Storage cost = $\sum_j (C_j \cdot Price^{Storage}_j \cdot Size^{total} \cdot T)$

19: **Constraints:**

20: *// k is in the union of i's and j's PUT replica sets if it is in either set*

21: $\forall i, j \in AS, k : (U^P_{ijk} = 1)$ iff $(PR_{ik} + PR_{jk} > 0)$

22: *// for the PUT replica sets of any pair of data centers in access set, the sum of their cardinalities should exceed the cardinality of their union by $2f$*

23: $\forall i, j \in AS : \sum_k (PR_{ik} + PR_{jk}) > \sum_k U^P_{ijk} + 2f$

24: *// for any pair of data centers in access set, GET replica set of one must have intersection larger than $2f$ with PUT replica set of the other*

25: $\forall i, j \in AS, k : (U^G_{ijk} = 1)$ iff $(GR_{ik} + PR_{jk} > 0)$

26: $\forall i, j \in AS : \sum_k (GR_{ik} + PR_{jk}) > \sum_k U^G_{ijk} + 2f$

27: *// a PUT from i is relayed to k iff k is used to propagate i's PUT to any of i's PUT replicas*

28: $\forall i \in AS, k : (R_{ik} = 1)$ iff $(\sum_j F_{ikj} > 0)$

29: *// some k must forward i's PUTs to j iff j is in i's PUT replica set*

30: $\forall i \in AS, j : PR_{ij} = \sum_k F_{ikj}$

31: *// a data center is a replica if it is either a PUT replica or a GET replica for any data center in access set*

32: $\forall j : (C_j = 1)$ iff $(\sum_{i \in AS} (GR_{ij} + PR_{ij}) > 0)$