Bolt-On Global Consistency for the Cloud

Zhe Wu*
Google Inc.
Mountain View, CA
towuzhe@gmail.com

Edward Wijaya
University of Michigan
Ann Arbor, MI
wijaya@umich.edu

Muhammed Uluyol
University of Michigan
Ann Arbor, MI
uluyol@umich.edu

Harsha V. Madhyastha
University of Michigan
Ann Arbor, MI
harshavm@umich.edu

Abstract

Web services that enable users in multiple regions to collaborate can increase availability and decrease latency by replicating data across data centers. If such a service spreads its data across multiple cloud providers—for the associated performance, cost, and reliability benefits—it cannot rely on cloud providers to keep the data globally consistent.

Therefore, in this paper, we present an alternate approach to realizing global consistency in the cloud, which relies on cloud providers to only offer a strongly consistent storage service within each data center. A client library then accesses replicas stored in different data stores in a manner that preserves global consistency. To do so, our key contribution lies in rethinking the Paxos replication protocol to account for the limited interface offered by cloud storage. Compared to approaches not tailored for use in the cloud, our system CRIC can either halve median write latency or lower cost by up to 60%.

CCS Concepts

• **Information systems** → **Cloud based storage**; *Storage replication*;

Keywords

Cloud storage, strong consistency, Paxos

ACM Reference Format:
Zhe Wu, Edward Wijaya, Muhammed Uluyol, and Harsha V. Madhyastha. 2018. Bolt-On Global Consistency for the Cloud. In *Proceedings of SoCC '18: ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 13 pages.
https://doi.org/10.1145/3267809.3267835

*Work done while the author was a Ph.D. student at University of Michigan

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA

ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267835

1 Introduction

Minimizing user-perceived latency is a critical goal for web services, as even a few 100 milliseconds of additional delay can significantly reduce revenue [1]. Key to achieving this goal is to deploy web servers in multiple locations so that every user can be served from a nearby server. Therefore, cloud services such as Azure and AWS are an attractive option for web service deployments, as these platforms offer data centers in tens of locations spread across the globe [5, 7].

To simplify the development of geo-distributed web services, cloud providers have recently begun offering support for replicating data across data centers while exporting consistency semantics as though there were a single global copy of each data item. Google's Cloud Spanner [9] and Azure's Cosmos DB [6] are examples of such storage services. Geo-replicating data, like in these services, is crucial to maximize availability (so as to tolerate data center and Internet path failures) and to achieve low latency (so that web servers can read/write shared data by accessing a subset of nearby copies).

However, reliance on such geo-distributed storage services offered by cloud providers is not an option for applications that choose to spread their data across *multiple* cloud providers. While prior work has shown that multi-cloud web service deployments can offer performance, cost, and fault-tolerance benefits [31, 32, 53, 55], a cloud provider has no incentive to keep data stored on its platform in sync with copies stored on other platforms. As a result, applications are left with the onus of managing the consistency of data that they replicate across the data centers of multiple cloud providers.

To simplify the development of applications that opt for multi-cloud storage, we present CRIC (*Consistent Replication in the Cloud*), a client library which exports single-copy semantics for data replicated across data centers in the cloud. Rather than relying upon a geo-distributed storage service, CRIC builds upon strongly consistent storage services *within* each data center as building blocks. Given multiple data centers across which to replicate any object, a web service can store a copy of the object in the storage service at each of these data centers. CRIC enables a client VM in any data center to then read and update these copies without violating global consistency.

The key challenge in developing CRIC is to ensure that the latency of accessing geo-replicated data using it is comparable to that feasible when using a geo-distributed service. The reason this is challenging is because there is a mismatch between the interface offered by cloud storage services (e.g., PUT/GET) and the interface necessary to reuse existing low latency replication protocols (e.g.,

Accept/Learn to use Fast Paxos [36]). Since application providers have no control over modifying cloud storage's interface, they could resort to using replication protocols [25, 29] that are compatible with this limited interface. Doing so however significantly degrades latencies, as these protocols require additional round-trips of wide-area communication.

In CRIC, we address this challenge in two ways. First, we develop CPaxos, a variant of Paxos optimized for use in the cloud. To cope with the limited storage interface, CPaxos stores the state maintained by Paxos acceptors in cloud storage but moves the logic executed by Paxos acceptors into the client. Unlike prior variants of Paxos [25, 29] designed similarly, CPaxos leverages additional features that strongly consistent cloud storage services export beyond a simple PUT/GET interface. These include the ability to conditionally update an object only if it is unchanged since it was last read, and the option of augmenting every object's data with metadata that can be independently accessed and updated. By exploiting these features, CPaxos can execute both reads and writes with one round-trip of wide-area communication in the common case when conflicts are rare.

Second, to ensure that latencies and throughput are not significantly degraded in settings where conflicts are common, when any client VM attempts to update multiple copies of an object, CRIC makes it likely that the update is successfully applied either to all copies or to none. For this, rather than *concurrently issuing* requests to an object's replicas, CRIC staggers its requests such that they *arrive near-simultaneously* at these replicas. Such staggering of requests is feasible because Internet path latencies between cloud data centers are stable and these latencies dominate any client VM's interactions with remote storage.

We have implemented a prototype of CRIC and deployed applications that use it across AWS and Azure. Compared to existing replication protocols compatible with a limited storage interface, we show that CPaxos can halve write latency and also lower cost by 10–60%.

2 Overview and Motivation

In a geo-distributed web service deployment, user-facing web servers are deployed in multiple data centers, so that any user's request can be served from a nearby location. The latency incurred by a user includes time that the web server waits for data to be read from or written to storage. We focus on geo-replicating data to minimize latency and maximize availability for web services that use key-value stores to store small KB-sized objects [13, 19, 44]. By geo-replicating data, we can enable web servers to read/write a object by accessing a subset of nearby replicas, rather than a centrally located copy of the object.

2.1 Why multi-cloud?

While public cloud services allow tenants to deploy applications in many locations, any tenant must choose from the set of data centers in the cloud platform it uses. If a cloud provider has little or no presence in a particular region, then its tenants will be unable to store any data there, and will instead have to access data stored in a distant location.

Since different cloud providers target different markets, using multiple clouds enables tenants to store data in more geographic

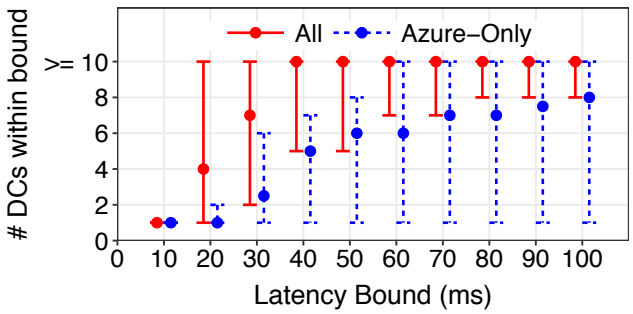

Figure 1: For each Azure data center, we compute the number of other data centers within varying latency bounds. For each latency bound, the min, median, and max value across Azure data centers is shown.

regions. For example, while US-based cloud providers (e.g., AWS and Azure) have a large number of data centers in North America and Europe, Alibaba Cloud and Telefónica target the Asia-Pacific and Latin America regions respectively. While porting applications to multiple clouds is challenging due to differences in the interfaces they offer, modifying applications to use additional providers only for data storage is relatively simpler. For example, in this paper, we consider that a web service's servers will be deployed across the data centers of a single cloud provider, but any data center that has a key-value storage service—irrespective of which provider owns that data center—is a candidate for storing some of the service's data.

To quantify the utility of using multiple cloud providers for data storage, we consider 11 public cloud providers. Besides AWS, Azure, and Google Cloud, we consider the public cloud services offered by IBM, Alibaba Cloud, Telefónica, CityCloud, OVH, ECS, Rackspace, and SAP; all of these providers have data centers in which they offer strongly consistent key-value storage services. We measured network latencies between all pairs of AWS, Azure, and Google Cloud data centers and used this data to compute a linear regression model of latency as a function of geographic distance. Using our best estimates of the physical locations of data centers operated by each cloud provider, we then estimated latencies between all pairs of data centers.

For any web service hosted on Azure, Figure 1 shows that using all 11 cloud providers for data storage significantly increases the number of data centers near any Azure data center. The median Azure data center has 5 *other* Azure data centers within a 50 ms radius, while the most isolated data center has to suffer over 100 ms latency to contact another Azure data center. In contrast, using our full list of cloud providers, the median Azure data center has over 9 other locations within a 50 ms radius and the most isolated Azure data center has a non-Azure data center just 30 ms away.

To see how this might affect storage performance, we evaluate the latency improvement for two access patterns. The first considers data being accessed from Korea and Singapore, and the second considers data being accessed from Central India and UK; these are locations in which both Azure and AWS have data centers. In either case, we assign data to be replicated across 3 data centers chosen to minimize

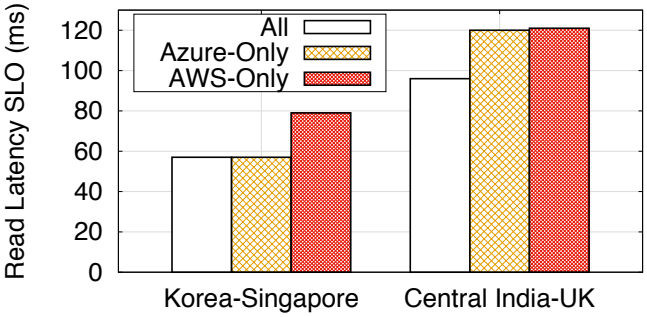

Figure 2: Latency improvement from going multi-cloud. For two different access patterns, we show the minimum read latency SLO feasible when using a single provider for data storage versus when using multiple providers.

the latency for either access location to read data by contacting a majority of the replicas.

Figure 2 shows that using all clouds for storage can provide better read latency than using only Azure or AWS. For data accessed from Korea and Singapore, using all clouds improves read latency compared to AWS-only by 25% while Azure-only suffices in this case for low latency. However, for data accessed from Central India and UK, using all clouds improves read latency compared to both Azure-only and AWS-only storage by 20%.

2.2 Goals

Realizing these latency benefits from multi-cloud data storage is however challenging for applications because they cannot rely on a storage service offered by any provider to manage the geo-replication of data. Therefore, our goal is to enable web service developers to outsource to CRIC the task of managing the consistency of data replicated across data centers. As shown in Figure 3, CRIC's client library stores data in and accesses data from strongly consistent storage services offered by cloud providers within each of their data centers. CRIC can implement operations supported at all the storage services that it builds upon (§5), but we focus primarily on reads, writes, and conditional-writes on individual geo-replicated objects (§3).

Our design of CRIC is guided by four objectives.

- **Strong consistency:** All reads and writes must be linearizable. Strongly consistent replicated storage simplifies application development, enables seamless porting of web services written to use centralized storage, and is essential in many web services, including collaborative document editing (e.g., Google Docs), online auctions (e.g., eBay, stock trading), and multi-player online gaming.
- **Fault-tolerance:** CRIC must ensure that any object continues to be readable/writable as long as storage services in at most f data centers are unavailable.
- **Low latency:** CRIC should ensure low latencies for both reads and writes.

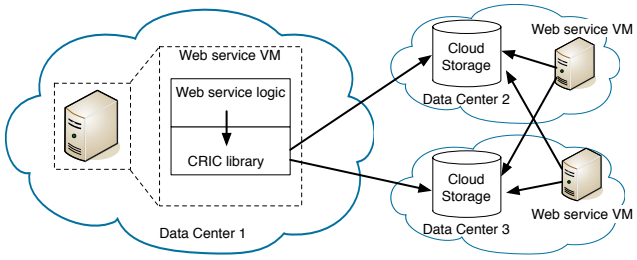

Figure 3: Illustration of CRIC's use in a geo-distributed web service deployed in the cloud.

- **Cost-effectiveness:** CRIC should minimize the cost overhead (i.e., the dollar amount charged by cloud providers) that it imposes on web services. We account for the cost of data transfers, GET/PUT requests, and any VMs necessary to support geo-replication.

3 Global consistency atop limited interface

CRIC's interface matches that offered by strongly consistent intra-data center key-value stores. Here, we describe CRIC's support for three operations: read, write, and conditional-write (a client's conditional-write on an object succeeds only if the object has not been modified since the client last read it); Table 1 summarizes the various techniques we use. These operations are analogues of the GET, PUT, and conditional-PUT operations supported by cloud storage within each data center. In Section 5, we describe how the mechanisms used in CRIC also enable support for other operations offered at all the data centers across which objects are replicated.

3.1 Basics

The need to support conditional-writes requires being able to associate every object's data with monotonically increasing version numbers. Once a write to version i of an object is complete, any attempt by a client that previously read version j ($j < i$) to write version $j + 1$ must fail. This rules out the use of protocols such as ABD [14], which suffice to implement a PUT/GET interface atop replicated data, but are fundamentally incapable of serving as the basis for the read-modify-write semantics necessary to implement conditional-writes [14]; see Appendix B for an example.

Therefore, CRIC instead builds upon Paxos. Each version of an object corresponds to one Paxos instance, and every replica of the object serves as a Paxos acceptor. When a client executes a conditional-write, the CRIC library attempts to write to the version one higher than that last read by the client and returns an error if a quorum of acceptors have already accepted a value for this version. To execute a (unconditional) write, the CRIC library repeatedly retries writing to higher versions until success.

The state stored at each replica of an object comprises a log of increasing versions (Figure 5). For each version, we store the highest proposal number that the acceptor has promised to accept, the highest proposal number accepted, the last accepted value, and a commit bit, which indicates whether the locally accepted value for this version has been globally accepted.

Metric to optimize	Technique	Section
Write latency	Treat cloud storage as a passive acceptor and use conditional-PUTs to update Paxos state, cache acceptors' states, and attempt to write to a super quorum of acceptors	§ 3.2
Read latency	Clients act as learners, asynchronously update Paxos state to mark version as globally accepted	§ 3.3
Storage and data transfer costs	Garbage collect old versions in Accept phase Store Paxos state as object's metadata so that no need to transfer object data in Prepare phase	§ 3.4
Performance under conflicts	Stagger client's requests to replicas such that requests arrive near-simultaneously at either the closest quorum or the closest super quorum	§ 3.5

Table 1: Techniques used in CRIC to minimize latency and cost for strongly consistent reads and writes.

	Write latency (RTTs) in common case		When executing writes, # of copies of object's data transferred per write attempt		Average # of copies of object's data per replica
	Fast path	Slow path	No conflict	n concurrent proposers	
Disk Paxos [25]	2	4	$O(m)$	$O(m)$	m
pPaxos [29]	2	4	3	$O(n)$	≈1
Classic Paxos [35]	1	2	1	1	1
CPaxos	1	2	1	$O(1)$	1

Table 2: Comparison of CPaxos to prior protocols. m is the number of clients. For pPaxos, we assume older versions are garbage collected as soon as a new version is globally accepted.

3.2 Low latency writes

Using Paxos to execute writes requires replicas to participate in the Paxos protocol. For this, one could deploy VMs which proxy all client interactions with cloud storage and augment the storage interface. However, as we show later in Section 4, deploying a sufficient number of proxy VMs so that they do not prove to be a bottleneck can significantly increase the cost that a web service must pay to the cloud provider. Moreover, since write conflicts are rare in typical web service workloads [23, 41], these proxy VMs would largely have to simply relay data between application VMs and cloud storage, without having to perform any concurrency control.

Therefore, to enable client VMs that use CRIC to directly communicate with cloud storage, we design CPaxos (Cloud Paxos), a new variant of Paxos which works with passive Paxos acceptors (i.e., acceptors that only store state but cannot run any computation); see Appendix A for proof of correctness. Like prior variants of Paxos [25, 29] designed for passive acceptors, CPaxos moves the acceptor and learner logic into the proposer. However, prior Paxos variants for passive acceptors inflate latency and cost (see Table 2) because they require two rounds of communication to complete each phase of Paxos: one round to perform conflict-free writes at a quorum of acceptors (e.g., write a new object [25] or perform an atomic append [29]), and another round to read the state of the acceptors to check for success. We use three techniques to execute a write in one round of wide-area communication in the common case, when conflicts are rare [23, 41].

Leveraging conditional updates. To write to a specific version of an object, like in classic Paxos, a CPaxos proposer must first gather promises from a majority of acceptors in the Prepare phase and then get its proposal accepted by a majority of acceptors in the Accept phase. For this, as shown in Figure 4, a proposer first gathers the state from a majority of acceptors. For each acceptor, the proposer then runs the same logic that the acceptor would have executed upon receiving the proposer's Prepare message. If the proposer finds that a majority of acceptors would have promised to accept its proposal, it then attempts to update the state of these acceptors using conditional-PUTs; the update of any acceptor's state succeeds only if the state has not already been updated since when the proposer read it.

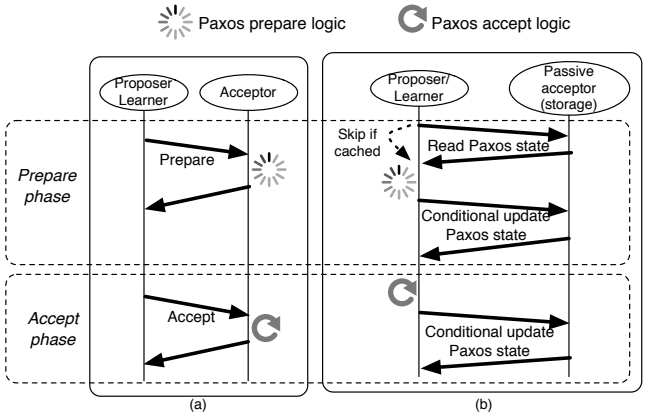

Figure 4: Comparison of (a) classic Paxos and (b) CPaxos.

If conditional-PUTs fail at a majority of acceptors, the proposer re-collects acceptor states, re-runs the logic for the Prepare phase (possibly with a higher proposal number), and tries to update the acceptors' states again. The proposer repeatedly does so until it

either gets a majority of acceptors to promise to accept its proposal, or discovers that a conflicting value has been globally accepted.

The Accept phase is similar to the Prepare phase, except that the proposer need not collect the acceptors' states at the start of the Accept phase; it already knows the state of at least a majority of acceptors after its conditional updates succeed in the Prepare phase.

Thus, after collecting the states of all acceptors, CRIC leverages conditional-PUTs to complete the Prepare and Accept phases of a write in one round of communication each in the common case. In contrast to the concurrency-safe primitives that prior Paxos variants designed for passive acceptors [25, 29] rely upon (write a new object or do an atomic append), conditional-PUTs ensure that at most one among concurrent attempts to update a replica will succeed and that the successful client will discover its success without incurring an additional round-trip to read the acceptor's state.

Caching acceptors' states. The first step of collecting the current states of all acceptors can often be omitted because 1) if an object is being created, it will have no state at any acceptor, and 2) when a client is updating an object, it has likely read that object in the past, enabling the CRIC library to cache acceptors' states. If an object is updated between a client's read and subsequent write, the client's write to the version one higher than what it had read will fail, even with classic Paxos. In workloads where this scenario is common and it is important for such writes to fail quickly, the CRIC library can either periodically refresh cached acceptor states, or first check that the state of the closest acceptor is unchanged before attempting to execute a write.

Enabling one round write. Taking inspiration from Fast Paxos [36], we further reduce write latency to one round of communication. Since conflicts are rare in typical web service workloads [23, 41], any write can start with an attempt to fast accept the update by skipping the Prepare phase and running the Accept phase for *proposal #0*. The value is considered as committed once a super quorum $\left\lceil \frac{3}{2}f + 1 \right\rceil$ of the object's $2f + 1$ replicas accept the value. Requiring acceptance at a super quorum of replicas ensures that, when a reader subsequently reads from any quorum of replicas, a successful write would have been accepted at a majority of that quorum.

Failure of the fast round indicates that there may be conflicting writes, so the proposer must fall back to the regular two-round CPaxos protocol using a higher proposal number. This increases the likelihood that one of the writers will be successful. During the Prepare phase, if a proposer observes multiple values accepted in *proposal #0* and is unable to determine if any of them is globally accepted (e.g., because some replicas are unavailable), like in Fast Paxos, it will pick the value with the highest frequency (because any value committed in a fast round will have been accepted by a majority of any quorum of acceptors) and try to get it globally accepted.

Note that, when objects are geo-replicated, using a fast round may not always lower write latency even when conflicts are rare, because two rounds of communication with a quorum may impose lower latency than one round-trip to a super quorum [33]. Given the placement of an object's replicas, CRIC can identify whether a particular client would benefit from using a fast round to update the object.

3.3 Low latency reads

Since typical web service workloads are dominated by reads [13, 23], it is particularly vital to minimize read latency. Hence, in the common case, CRIC enables any client to read the latest version of an object in one round of communication with the closest majority of the object's replicas. We achieve this property by having clients act as learners.

After a client has its update to an object's data accepted by a quorum of acceptors, the client asynchronously issues conditional-PUTs to update the commit bit for this new version in the object's replicas. The utility of doing so is that, when a client wants to read the latest version of an object, it can issue GET requests to fetch the CPaxos logs for this object from all of the object's replicas but need not wait to hear from *all* replicas in order to identify the highest globally accepted version. Instead, once a reader is done fetching CPaxos logs from any majority of acceptors (typically the quorum of acceptors closest to it), if it finds the commit bit for the highest version enabled at any replica, the reader can safely use the corresponding data.

A client may find that the commit bit for the highest version it sees is not set at any of the majority of replicas that it first hears from, e.g., because this read occurs after a new version has been globally accepted but before the writer updates the commit bit for that version. In this case, the client needs to simply wait to receive CPaxos logs for the object from more replicas to recognize the committed version.

When storage services in some cloud data centers are unavailable, a reader may be unable to confirm that a particular version has been globally accepted even if it reads from a majority of replicas that are available; that version may have been accepted by a different majority of acceptors, some of whom are currently unavailable. In this case, a client identifies the highest version accepted at any of the acceptors from which it has read the CPaxos log. Then, like prior systems [16, 46] that use Paxos, the client re-proposes the data corresponding to the highest accepted proposal number for this version in order to either get this value globally accepted or discover a different value that is already globally accepted. The purpose of a reader performing such a write back is to ensure that, in case an object is in an inconsistent state (e.g., because a client failed in the midst of performing a write), the object is left in a consistent state which reflects the data returned to the application. Since failures in the cloud are rare [2, 3, 10], readers will seldom have to incur the latency overhead of performing write backs.

3.4 Low cost

In comparison with classic Paxos, CPaxos inflates the cost associated with network transfers in two ways: 1) from every acceptor, a reader has to fetch the entire CPaxos log, not just the last accepted version, and 2) to update an acceptor's state via a conditional-PUT, a client has to redundantly transfer all data previously stored in the CPaxos log. We address these sources of cost overhead via efficient garbage collection and by separating data from metadata.

Garbage collection of CPaxos logs. Since readers need to know only the highest version accepted by each acceptor (Section 3.3), we can afford to garbage collect CPaxos state for older versions during the Accept phase of a write. Even if all older versions of an object are garbage collected when writing a new version, this does

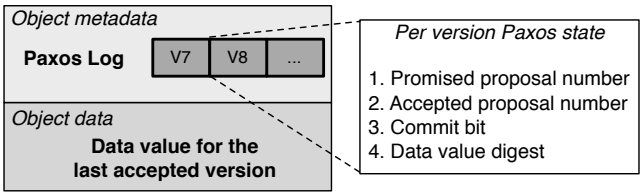

Figure 5: CRIC's per-replica state in cloud storage.

not violate safety; if another client reads this new version before it is globally accepted, the reader will write back a value for this version before returning to the application. In settings where conflicts are known to be not so rare, one can minimize the chances of readers having to perform write backs by configuring CRIC so that writers garbage collect *some*, but not *all*, older versions in the Accept phase.

The key point to note here is that CRIC can safely afford to always store a single copy of an object's data at any replica. In contrast, to not violate safety, prior variants of Paxos designed for passive acceptors [25, 29] require every writer to create an *additional* copy of the object's data at a quorum of replicas, even if every update to an object is a complete overwrite of the object's data. This in turn leads to overheads in network transfers as subsequent readers must necessarily fetch *multiple* copies of the object's data from each replica.

Separation of object data and Paxos state. During the Prepare phase of a write, only the CPaxos state needs to be updated but not the object's data. To enable this, we store an object's data separately from the CPaxos state at each of the object's replicas (see Figure 5). In services that offer blob storage, we leverage support for storing with any key a limited amount of metadata which can be modified independently of the key's value [8]; storing the CPaxos state for an object in a separate key would make multi-key transactions necessary, which key-value stores often do not support. In services that offer tabular storage (e.g., DynamoDB), we store metadata for each key in a separate column.

We also store within the metadata a *data digest*: a hash of the data last accepted at that replica. In low conflict workloads where replicas are typically in sync, CRIC reduces cost by having a client read an object's data only from the closest replica and read only the metadata from the other replicas to confirm that they have the same data for the object.

3.5 Performance under conflict

In comparison with classic Paxos, performance with CPaxos degrades more rapidly if conflicts become common. This is because, when a client's conditional-PUT to update a replica fails, the client must incur an additional round-trip to read the acceptor's state at that replica before it can retry its proposal. In contrast, when an active acceptor rejects a proposal, its response includes the information necessary for the proposer to retry.

Rather than addressing this difference between active and passive acceptors, we tackle the root cause for conflicts: no client will be able to complete its write if every client's proposal is accepted by less than a majority in a slow round or by less than a super quorum in a fast round. As shown in Figure 6(a), this occurs because a client's requests can arrive at different replicas at different times.

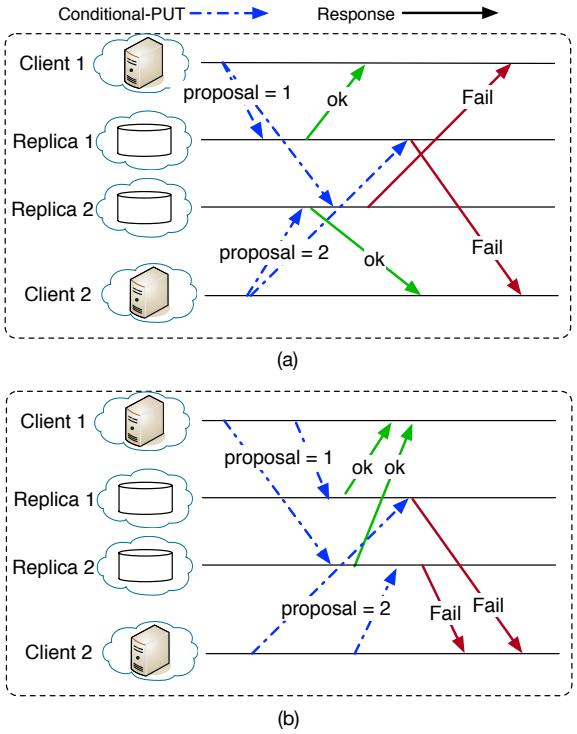

Figure 6: Illustration of staggered requests: (a) When clients issue requests to all replicas simultaneously, update may be successful at some replicas but not others. (b) When every client staggers its requests so that they arrive at the replicas near-simultaneously, either all replicas will be updated or none.

Though both replicas are in the same state to begin with, by the time Client1's request arrives at Replica2, that replica has already been updated by Client2; similarly, Client1 updates Replica1 before that replica receives Client2's request. When data is geo-replicated, such a scenario is likely to occur when there are concurrent writers, because the latency from a client can vary significantly to different replicas.

To address this cause for performance degradation when there are concurrent writers, CRIC does not send out requests to all replicas simultaneously. Instead, it staggers its requests such that they *arrive* near-simultaneously at the subset of replicas that the client needs to update for its write to be successful. For example, in the fast round, after the client first sends out its request to the farthest replica R in the closest super quorum, the client delays its request to any other replica R' in the super quorum by an amount equal to the difference in the client's latencies to R and R'. A client can similarly stagger its requests in a slow round so that its requests arrive at the closest majority of replicas near-simultaneously. As shown in Figure 6(b), the likely outcome of such staggering of requests will be that either all the desired replicas are updated or none.

Staggering requests in this manner works for two reasons. First, when data is geo-replicated, wide-area network latencies dominate the latency of interaction between client VMs and storage in remote data centers. Second, network latencies between cloud data centers

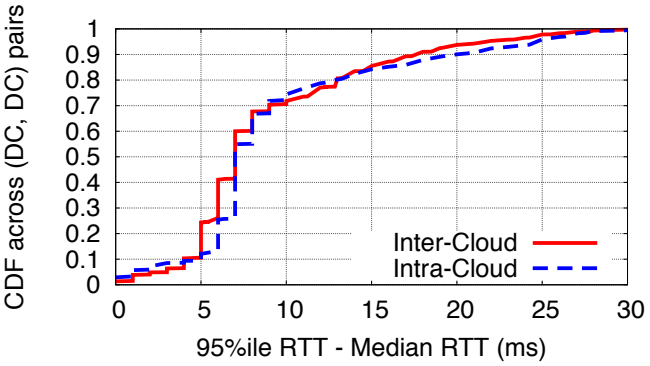

Figure 7: Low variance of wide-area network latencies for Azure, AWS, and Google Cloud. 10,000 latency samples were collected for every pair of data centers.

exhibit low variance. Figure 7 shows that this is true both for Internet paths between data centers in the same cloud platform and between data centers in different platforms.

Note that staggering requests as described also helps when executing reads. A reader's staggered requests to the closest majority of replicas helps increase the likelihood that the client sees the effects of the same write at all replicas.

4 Evaluation

We evaluate CRIC in three parts. First, in a prototype deployment of CRIC, we evaluate the latency and throughput performance under different conflict rates, in comparison with alternative designs not optimized for the cloud. Second, we conduct a simulation-based evaluation to demonstrate CRIC's cost-effectiveness. Finally, we showcase the utility of CRIC on a real-world web service's workload. The primary takeaways from our evaluation are:

- CRIC's performance is comparable to approaches that require proxy VMs to augment the storage interface, but at 20–50% lower dollar cost.
- Compared to existing replication protocols compatible with passive Paxos acceptors, CPaxos can halve write latency and also lower dollar cost by 10–60%.
- Under high conflict scenarios, CRIC's use of staggered requests helps reduce median write latency by over 50%.

4.1 Prototype Evaluation

Implementation. Our prototype implementation of CRIC is roughly 5400 lines of Java code. We use Thrift [48] for RPCs between VMs, and interact with every cloud storage service using the official client libraries. Our prototype is compatible with Microsoft Azure and Amazon AWS.

To compare CRIC with existing solutions, we also implemented two comparison systems: **Fast Paxos** and **pPaxos***. Note that we do not directly compare CRIC with cloud storage services which support geo-replication of data for two reasons. First, at the time of writing, Cosmos DB [6] does not yet support strongly consistent replication of data across Azure regions for cloud customers. Therefore, an experimental performance comparison between CRIC

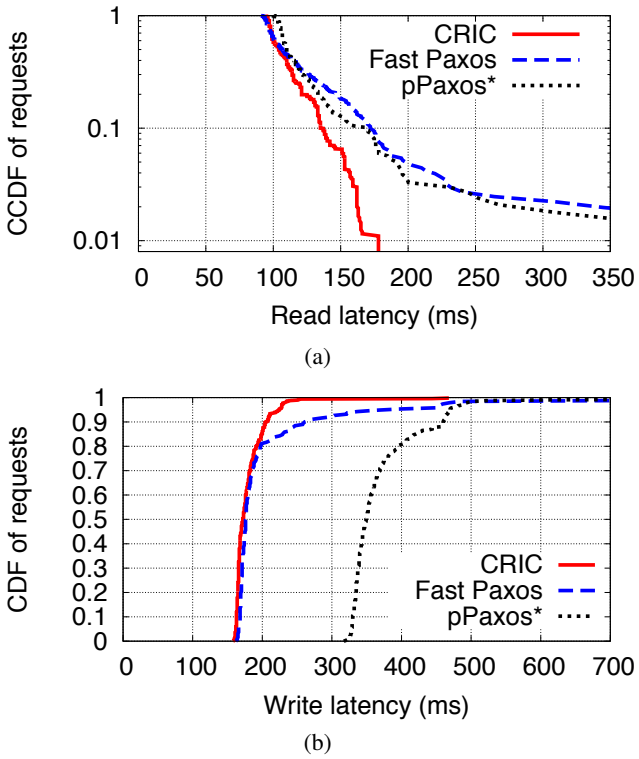

Figure 8: (a) Read and (b) write latencies under low conflicts (Zipfian coefficient = 0.5).

and Cosmos DB is currently infeasible. Second, instead of directly comparing with Cloud Spanner [9], we compare CRIC with the state of the art Paxos protocol, which is the core replication protocol used in Spanner [23].

- **Fast Paxos:** We implement Fast Paxos to represent a range of existing solutions, such as MDCC [34] and TAPIR [57], that would require proxy VMs to replicate data in the cloud. Although these systems provide richer functionality such as transactions, using them to offer a globally consistent view of key-value storage would offer performance similar to the Fast Paxos protocol. Our implementation of Fast Paxos contains two components: a client library that implements a Fast Paxos proposer, and code that runs on VMs which proxy requests to cloud storage and mimic Fast Paxos acceptors and learners.
- **pPaxos*:** Among prior Paxos-based replication protocols that work with passive acceptors, pPaxos [29] is strictly better than Disk Paxos [25] in terms of performance and cost. Therefore, we compare CRIC only with pPaxos. Our implementation, pPaxos*, improves pPaxos's efficiency by using fast round writes and optimal garbage collection.

  Not all cloud storage services support Append operations as needed by pPaxos. Even storage services that do support Append may impose restrictions, e.g., though Azure Storage supports Append Blobs, updating and deleting existing blocks in a Blob is not supported, thus preempting garbage collection. We

therefore implement Appends via proxy VMs that interpose on requests to storage but ignore the dollar cost of these VMs in our comparisons.

Deployment setting. We deploy clients in A2 instances (2 cores, 3.5GB RAM) at 5 Azure data centers: East US, West US, Japan, West Europe, and Southeast Asia. In the blob storage service at each of these data centers, we store a copy of one million objects. When proxy VMs are necessary, we deploy a sufficient number of them for these VMs to not be a bottleneck.

Clients issue reads and writes using YCSB [22], a key value store benchmark which emulates cloud workloads. We set the number of clients such that client VMs are fully utilized. We vary the read-to-write ratio, average object size, and Zipfian distribution that reflects how popularity varies across objects. The higher the Zipfian coefficient, the higher the rate of conflicts.

Performance under low conflict rates. First, when conflicts are rare, we show that CRIC 1) offers performance comparable to approaches that would require proxy VMs for deployment in the cloud, and 2) outperforms prior replication protocols compatible with passive acceptors. In this experiment, we use 1KB objects, set the read-to-write ratio to 30, and set the Zipfian coefficient to 0.5.

Figure 8 shows the distribution of read and write latencies observed by clients in the Azure East US region. In terms of read latency, Fast Paxos, pPaxos*, and CRIC all offer similar latencies since all of them require one round trip to a majority of replicas to execute reads in the common case. Whereas, for writes, latencies with pPaxos* are double that with CRIC and Fast Paxos. This is because, after appending to acceptors' logs, a pPaxos* proposer has to read back those logs to check whether its request was accepted. Tail latencies for both reads and writes are lower with CRIC because its use of staggered requests reduces the need for writers to retry and for readers to write-back.

Performance under high conflict rate. Next, we showcase CRIC's ability to minimize performance degradation even if conflicts are common.

Under high conflict rates, a client may require multiple rounds of communication to complete a read or write, worsening both latency and throughput. Write-write conflicts cause the fast round to fail, and multiple proposers compete in one CPaxos instance. For read-write conflicts, readers may see an inconsistent state due to ongoing writes and perform write backs, which may then interfere with ongoing writes.

Write-write conflicts. To experiment with a scenario with a high conflict rate, we emulate a scenario where a number of clients are concurrently editing the same document (a la Google Docs or ShareLaTeX). We vary the number of clients at each of the five data centers, and we have every client repeatedly write to a shared key, waiting 2 seconds between consecutive writes.

For different conflict scenarios (more the number of clients per region, higher the rate of conflicts), Figure 9(a) compares the latencies for successful writes incurred by clients in the East US region with and without CRIC's use of staggered requests. When there is little to no conflict (i.e., 1 or 2 clients per region), it is best to not use staggered requests because waiting before issuing requests degrades latencies due to cloud storage's latency variance. When there are 4 or 5 clients per region, median write latency significantly

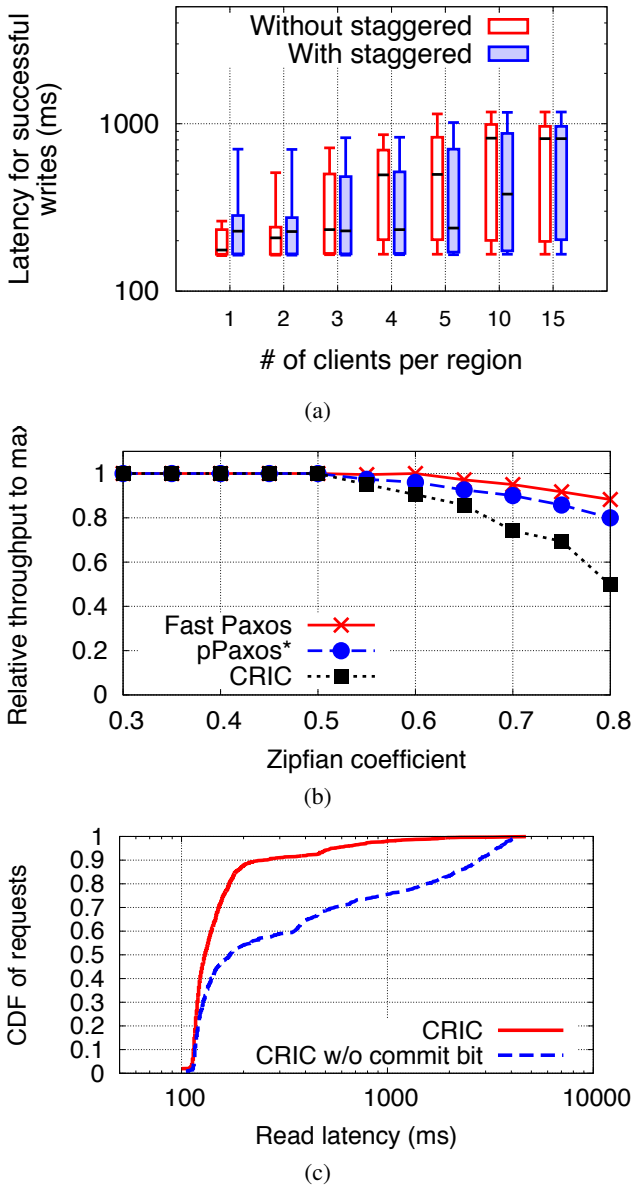

Figure 9: (a) Utility of staggered requests in reducing write latency under different conflict rates; y-axis is logscale, and 10th, 25th, 50th, 75th, and 95th percentile latency are shown. (b) Degradation in write throughput under different conflict rates. (c) Client perceived read latency distribution when conflict rate is high (Zipfian coefficient = 0.8).

degrades when requests to all replicas are issued simultaneously but remains unchanged when requests are staggered. At very high conflict rates (e.g., 15 clients per region), staggered requests no longer help; pessimistic concurrency control would be a better choice in such cases.

Although CRIC works well under low conflict rates, when the conflict rate is high, throughput with CRIC does suffer more compared to other replication approaches. Figure 9(b) shows how write

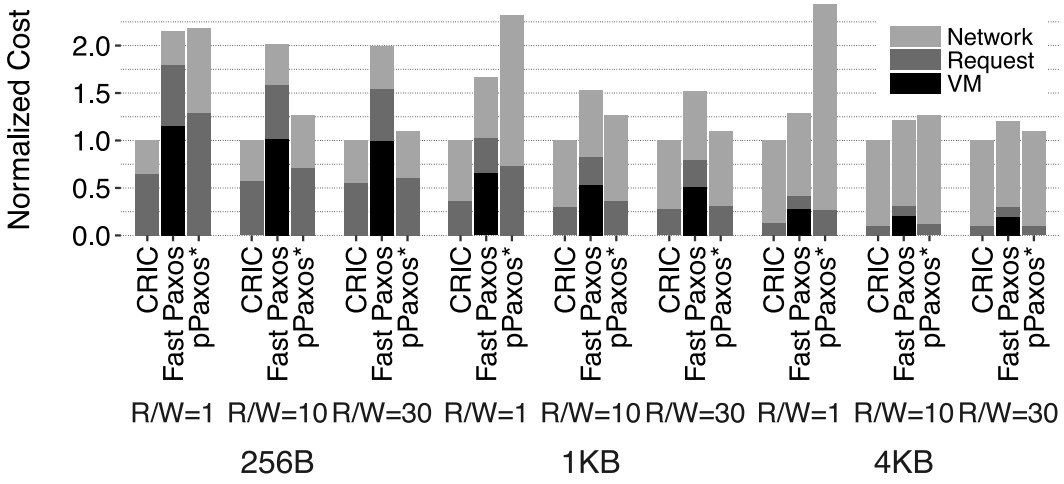

Figure 10: Comparison of dollar cost to execute reads and writes on geo-replicated data; R/W is read-to-write ratio.

throughput degrades in CRIC, Fast paxos, and pPaxos* under different conflict rates. As conflicts increase, CRIC's throughput degrades at a faster rate than Fast Paxos and pPaxos*. This is because in both Fast Paxos and pPaxos*, the proposer can learn the outcome of a Paxos phase in exactly one (Fast Paxos) or two (pPaxos*) rounds. In contrast, in CRIC, it may take an arbitrary number of rounds for a proposer to learn the outcome of a Paxos phase. This difference between CRIC and prior approaches arises because a failed conditional-PUT in CRIC requires the proposer to read the object again and possibly retry using the same proposal number; this process may have to repeated several times until the proposer finally knows the outcome of the Paxos phase. However, under such high conflict rates, it would be more prudent to use pessimistic concurrency control.

Read-write conflicts. In CRIC, when a write to an object is globally accepted but has not yet been applied to all replicas, clients who read that object may have to perform a write back. CRIC's use of the commit bit reduces the incidence of such write backs.

In our experimental setup, Figure 9(c) shows the read latency distribution observed at Azure's Japan data center when the Zipfian coefficient is set to 0.8 and read-to-write ratio is 1. We see that having writers asynchronously mark their completed write as committed greatly improves read latencies; as long as the commit bit is set at any replica with the highest version, a reader need not perform write backs.

4.2 Cost

While we have shown thus far that latencies and throughput with CRIC are comparable to those with Fast Paxos when conflicts are rare, using Fast Paxos incurs significant cost overhead due to the need for proxy VMs to augment the storage interface. Figure 10 compares CRIC against Fast Paxos and pPaxos* in terms of the dollar cost necessary to sustain any specific throughput. Note that there are three main sources of the cost incurred in serving users: the amount of computation resources (VMs) rented, the number of PUTs and GETs issued, and the amount of data transferred out of the data center in which the data is stored. We consider three read-to-write ratios (30, 10, and 1), three object sizes (256 bytes, 1KB, and 100KB), and a Zipfian coefficient of 0.5. These parameter choices are informed by prior studies of web service workloads [13, 19, 23].

We see that CRIC reduces dollar cost by 20–50% compared to Fast Paxos and pPaxos* in our target setting: conflicts are rare and objects are small [13, 19]. In such workloads, CRIC's cost savings over Fast Paxos stem from eliminating the need for VMs that relay transfers and enrich the cloud storage interface. The savings compared to pPaxos* are due to efficient use of storage and network resources: the use of conditional-PUTs and CRIC's more efficient garbage collection mechanism allow CRIC to transfer much less data to fulfill each read and write. Only if both objects are large and the read-to-write ratio is low, do CRIC's cost benefits reduce. Note that, even though, like CRIC, pPaxos* too does not require proxy VMs, its use may result in higher cost than Fast Paxos when objects are large. This is because a pPaxos* proposer has to always *append* a new copy of an object's data to any acceptor's log and has to therefore transfer multiple versions when reading the log to check the status of its proposal.

4.3 Application case study

In the final part of our evaluation, we examine CRIC's performance in a geo-distributed deployment of ShareLaTeX, a collaborative document editor. ShareLaTeX stores multiple versions of any particular document in a MongoDB instance to allow users to view the changes to a document.

Operations. Instead of storing a single copy of its data in MongoDB, we ported ShareLaTeX to use CRIC to geo-replicate its data across Azure and AWS data centers. Our port of ShareLaTeX supports two operations: updating a document (UpdateDoc) and reading a document (GetDoc). GetDoc first fetches the latest copy of the

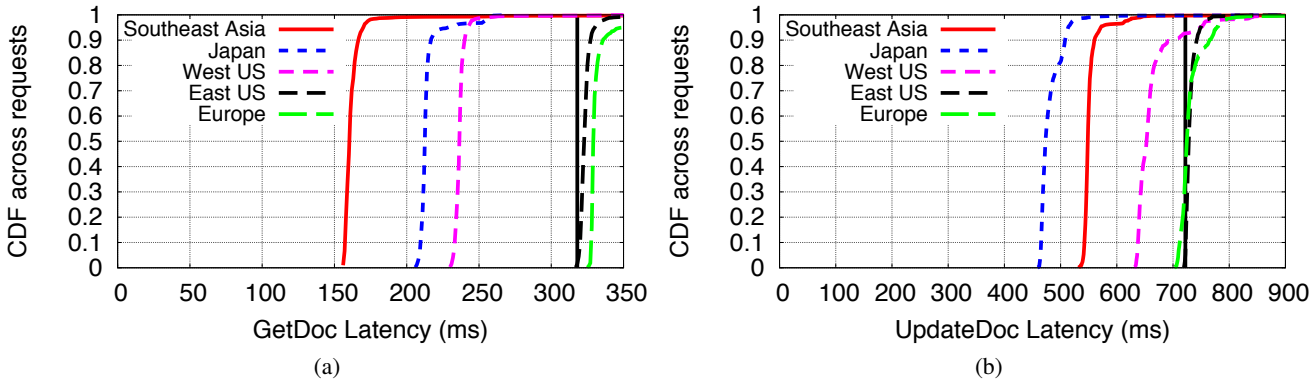

Figure 11: Latencies for operations in ShareLaTeX when using CRIC to store replicas in 5 data centers across Azure and AWS. Solid vertical line represents the latency bound estimated when placing replicas.

specified document before fetching the current version number of that document, and then returns both results to the caller. UpdateDoc gets the latest copy and current version number of the specified document to check if there are any changes, and then updates the document and version number; all operations are executed serially. While it may be possible to parallelize some of these operations, we focus on evaluating CRIC's performance on ShareLaTeX as is, not on improving the application itself.

Setup. We deploy ShareLaTeX in East US, West US, Japan, West Europe, and Southeast Asia as before. We use a mixed-integer-program solver and network latency data to select which data centers should store data. By identifying the nearest majority and super quorum for each application data center, we select a set of replicas that first minimizes read latency for the poorest-performing data center and then minimizes write latency. To tolerate $f = 2$ failures, we store 5 replicas in 3 Azure data centers (Southeast Asia, West Central US, West US 2) and 2 AWS data centers (Tokyo and Mumbai).

We preload our ShareLaTeX deployment with 10 projects each containing 100 1KB documents. We perform 1000 UpdateDoc operations using randomly generated values, followed by 1000 GetDoc operations. In both cases we select projects and documents at random and have 1 client thread running in each data center.

Performance. Figure 11 shows the latency achieved compared to the latency bound estimated when placing replicas (computed as the median network latency necessary to contact the nearest majority for reads and the nearest super quorum for writes). Because the chosen replicas are in Western US and East Asia, latencies for clients in those regions are significantly lower than the bounds. Our other two clients are within 4% of the bounds at the median and 6% at the 90th percentile. Interestingly, Japan and Southeast Asia swap positions in the relative ordering based on latencies for GetDoc and UpdateDoc operations. This is because read operations need only contact a majority of replicas while writes have to wait for a super quorum. While Southeast Asia is closer to a majority of replicas than Japan, Japan is more quickly able to contact a super quorum of replicas.

5 Discussion

Extensibility of API. The mechanisms used in CRIC to support reads, writes, and conditional-writes enable it to also support several other globally consistent operations on geo-replicated data. In general, CRIC can be extended to implement operations that are supported at all the data centers in which it stores data. This is because strongly consistent intra-data center cloud key-value stores support the conditional execution of all operations that either update or create objects. Here, we show evidence of CRIC's extensibility with a few examples. Specifically, we describe how CRIC can use the following operations offered by cloud storage to implement those same operations on geo-replicated data.

- **DELETE.** To delete an object, the CRIC library can first issue conditional-PUTs to update the metadata at a quorum of replicas to mark the object as deleted, and then asynchronously issue conditional-DELETEs at all replicas of the object.
- **LIST.** The two-step process for deletions is necessary in order to implement the LIST operation, which returns all objects in the account of the tenant issuing the operation. CRIC can implement LIST by invoking this operation at all data centers. For any object that exists at fewer data centers than the number of replicas stored for every object, the client must determine if this object is currently in the process of being created or deleted. The client can resolve this ambiguity by reading the object's metadata from the data centers where a replica of the object does exist; similar to a reader performing a write back, the client can then complete the creation/deletion of the object before returning the results of LIST to the client.
- **COPY and LEASE.** To create a copy of an object or to obtain a lease on it, a client must first read the metadata from the object's replicas, and perform write-backs if a quorum of replicas do not have the latest version. Thereafter, the client can issue a conditional-COPY or conditional-LEASE at each replica. The client must repeat these two steps until its conditional operations succeed at a quorum of replicas.
- **Transactions.** CRIC can also be used to execute multi-key transactions on geo-replicated data if the cloud storage service within

each data center offers support for transactions. For example, on Azure, CRIC can leverage the Table service's support for batched updates [11] to execute a transaction as a batch of conditional updates; on any particular replica, the batch is applied only if all the conditional updates within the batch succeed. In the absence of transaction support from cloud storage, CRIC can still be used to execute transactions with an additional round of latency to acquire locks (stored in the metadata of each replica) at a quorum of replicas.

Other benefits of approach. Though the primary motivation for our work is to help applications seamlessly replicate data across multiple storage providers with the same interface, CRIC's approach of bolting on [15] global consistency offers several other benefits.

- Even if an application replicates its data across the data centers of a single provider, CRIC promises higher availability than a geo-distributed storage service offered by that provider; failure or overload of one component of a geo-distributed storage service could cascade to other components [26–28], rendering the service unavailable, whereas data replicated with CRIC is available as long as the storage services in a majority of data centers are available.
- On cloud platforms that lack a geo-distributed storage service (such as AWS), CRIC enables web services to achieve global consistency semantics on geo-replicated data without waiting for the cloud provider to fill this void.
- From the perspective of cloud providers, CRIC simplifies the task of offering global consistency semantics. Rather than developing a new geo-distributed storage service, cloud providers who already offer a strongly consistent key-value store in each data center would merely need to distribute a new client library for applications to use.

6 Related work

Replication protocols. Paxos [35] and its variants [20, 36, 42] are well studied for transactional storage. Lynx [58] and Azure RTable [4] use chain replication to achieve serializable transactions. In contrast to chain replication, which sacrifices write latency for read latency, CRIC's use of Paxos replication permits both read and write operations to complete while waiting for responses from only the nearest quorum of nodes.

Many advancements in reducing commit latency stem from the development of new protocols that reduce network communication in the common case [23, 24, 34, 36, 38, 42, 43, 46, 57]. Building upon these protocols, CRIC's contribution lies in optimizing read and write latency while minimizing—in the common case, even eliminating—the need for intermediate virtual machines to enrich the interface to cloud storage.

Replicated storage. Like CRIC, other replicated systems too separate data and metadata, but with different goals. Gnothi [52] uses a greater replication factor for metadata than for data to improve fault-tolerance, whereas Giza [21] replicates only metadata and erasure codes data to reduce storage overhead.

Many geo-distributed storage systems [37, 39, 40, 50] opt for weaker consistency guarantees in exchange for low latency. In contrast, CRIC guarantees strong consistency to provide the simplest storage semantics.

Leveraging conditional updates. Giza [21] is similar to CRIC in its use of conditional-PUTs to cope with cloud storage's limited interface, but it relies on proxy servers to relay operations to storage; as we have shown, this can significantly inflate cost. Giza also does not address latency variance or throughput degradation at high conflict rates. Olive [47] takes advantage of conditional writes in cloud storage for a different purpose: to provide exactly-once semantics for application logic in the presence of failures.

Redesigning cloud services. To eliminate performance degradation caused by resource sharing in cloud services, several recent efforts propose redesigning storage systems [49, 51], data center networks [18, 30, 45, 54, 56], or both [12, 17]. Rather than wait for cloud services to adopt these more complex architectures, many of which can only offer predictability in terms of bandwidth but not latency, CRIC mitigates latency variance on legacy cloud services. Moreover, CRIC enables functionality—synchronization of replicas across the data centers of multiple cloud providers—that no cloud provider has the incentive to offer.

7 Conclusions

Replicating data across the data centers of multiple cloud providers for better cost-vs-performance tradeoffs comes at the expense of losing the luxury of being able to rely on a geo-distributed storage service to geo-replicate data. With CRIC, we have demonstrated that it is feasible to address this challenge efficiently with only a client-side library that requires no changes either to applications or to cloud services. Despite having to cope with cloud storage's limited unmodifiable interface, CRIC is able to efficiently and cost-effectively preserve the global consistency of geo-replicated data.

8 Acknowledgments

We thank Manos Kapritsos and Amy Tai for their feedback on earlier drafts of this paper. This work was supported in part by the National Science Foundation via awards CNS-1563849 and CNS-1463126, and by Amazon Web Services via their Cloud Credits for Research program.

Appendices

A Proof of correctness for CPaxos

THEOREM 1. *Only one value can be written for a given version, and committed value cannot be changed.*

PROOF SKETCH. Since all updates to an acceptor's state are via conditional-PUTs, a proposer can successfully update an acceptor only if it has the most up-to-date copy of that acceptor's state. Therefore, once a value is accepted by a quorum (super quorum) of acceptors in a slow (fast) round, the intersection between quorums guarantees that another proposer who proposes a different value in a slow round will discover the committed value in the Prepare phase and only propose the value that it discovers but not its own value in the Accept phase. Whereas, if a subsequent proposer uses a fast round with proposal #0, it will not succeed at a quorum, and it will revert to proposing in a slow round. □

LEMMA 1. *For a given version, once a value is globally committed, the highest accepted proposal must have proposed this globally committed value among a quorum (slow round) or super quorum (fast round) of acceptors.*

PROOF SKETCH. A value is globally committed indicates a quorum (a super quorum if fast round) of acceptors accepts a proposal with proposal number greater than or equal to the highest proposal number in them. Future proposals with lower proposal numbers will not succeed, which leads the proposers to re-propose using higher proposal numbers. For future proposals with higher proposal numbers, those proposals will discover the committed value in the Prepare phase and use the same value in their proposals. Therefore, the globally committed value must be associated with the highest accepted proposal number. □

THEOREM 2. *Read always returns the last committed write value.*

PROOF SKETCH. The moment a write is globally committed is when a quorum/super quorum of acceptors accept a proposal. First, we show that right after a value is globally committed (i.e., after the moment that a quorum/super quorum of acceptors accept this proposal), read is guaranteed to return this value.

When a read operation reads data from a quorum of replicas, there are 2 cases:

- At least one of the replicas has its commit bit set. In this case, read is guaranteed to return this value since it is indicated by the commit bit that the value associated with the highest accepted proposal number across all responses is committed (Lemma 1).
- No replicas have its commit bit set. There are 2 more cases in this scenario:
  - The highest proposal number is a slow round, and all the returned replicas have the same highest accepted proposal number and data. In this case, the reader knows this is the committed value, so it will return this value.
  - The highest accepted proposal number is a slow round and the discovered highest proposal number does not appear at a quorum of replicas, or the highest accepted proposal number is a fast round. In this case, the reader needs to re-propose the value with the highest accepted proposal number if the proposal number is for a slow round, or the most common value if the proposal number is for a fast round. Because of Lemma 1, if a write prior to this read has committed, the value associated with the highest accepted proposal number if it is a slow round or the most common value associated with the highest accepted proposal number if it is a fast round, must be the globally committed value. Therefore, the reader re-proposes the globally committed value, which will succeed, and then returns this value.

Since any read which happens right after the commit moment can return the just committed value by reading any quorum, following the same logic, other reads after this read will also always return the same value if no value is committed to any higher version. □

THEOREM 3. *Reader-write-back ensures linearizability.*

PROOF SKETCH. Linearizability requires that:

- Reader returns the last committed write value.
- Writes happen atomically (once a reader reads version i of the data, future reads will always return version i or higher version of the data).

So proving reader-write-back does not break linearizability requires proving these two properties hold.

Since Theorem 2 already proves A in the case of write back, here we only need to prove A. Moreover, if reads and writes happen disjointly (no overlap between the execution of requests), this case is also included in Theorem 2. So, all we need to prove is that when a read overlaps with a write, and if that read returns the value of that write, subsequent reads should always return that value (assuming no more writes after the write).

When the responses from a quorum of replicas do not contain the same highest version with the same highest accepted proposal number, the reader starts a new round of CPaxos to propose the value associated with the highest accepted proposal number, with a proposal number that is higher than all the proposal numbers seen from the response. The consequences of this re-propose are:

- The reader successfully gets it committed.
- The reader fails and it either finds there is a committed value, or it needs to re-propose again, until success.

In either case, when this read finishes, the value this read returns is a committed value. And because of Theorem 2, all future reads are guaranteed to return this value.

□

B Example of ABD's inability to support conditional updates

Augmenting the ABD [14] protocol to support conditional-writes on replicated data is fundamentally impossible because ABD is incapable of differentiating between successful and unsuccessful writes. The original ABD paper says as much: that it cannot be used as the basis for read-modify-write.

Consider an object with replicas *R1*, *R2*, and *R3*, all initially storing copies with logical timestamp 1.1 (version.clientID).

- Clients *C1* and *C2* first read from a quorum of replicas.
- *C1*'s conditional-write succeeds updating *R1* and *R2* to copies with timestamp 2.1.
- *C2*'s conditional-write fails but updates *R3* to a copy with timestamp 2.2.
- Now, if *C3* reads from *R2* and *R3*, it will choose to write back the copy with higher timestamp 2.2, incorrectly committing *C2*'s failed conditional-write.

References

[1] 2009. Latency Is Everywhere And It Costs You Sales - How To Crush It. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[2] 2014. Which cloud providers had the best uptime last year? http://www.networkworld.com/article/2866950/cloud-computing/which-cloud-providers-had-the-best-uptime-last-year.html.

[3] 2015. And the cloud provider with the best uptime in 2015 is ... http://www.networkworld.com/article/3020235/cloud-computing/and-the-cloud-provider-with-the-best-uptime-in-2015-is.html.

[4] 2016. Azure Replicated Table Library. http://github.com/Azure/rtable.

[5] 2017. AWS Global Infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/.

[6] 2017. Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/.

[7] 2017. Azure Regions. https://azure.microsoft.com/en-us/regions/.

[8] 2017. Azure Set Blob Metadata. http://docs.microsoft.com/en-us/rest/api/storageservices/fileservices/set-blob-metadata.

[9] 2017. Cloud Spanner. https://cloud.google.com/spanner/.

[10] 2017. CloudSquare Service Status. https://cloudharmony.com/status-1year-group-by-regions-and-provider.

[11] 2017. Performing Entity Group Transactions. https://docs.microsoft.com/en-us/rest/api/storageservices/fileservices/performing-entity-group-transactions.

[12] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-end Performance Isolation through Virtual Datacenters. In *OSDI*.

[13] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *SIGMETRICS*.

[14] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)* 42, 1 (1995), 124–142.

[15] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2013. Bolt-on causal consistency. In *SIGMOD*.

[16] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services.. In *CIDR*.

[17] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards Predictable Datacenter Networks. In *SIGCOMM*.

[18] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. 2013. Chatty Tenants and the Cloud Network Sharing Problem. In *NSDI*.

[19] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX ATC*.

[20] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: An engineering perspective. In *PODC*.

[21] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. 2017. Giza: Erasure Coding Objects across Global Data Centers. In *USENIX ATC*.

[22] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*.

[23] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*.

[24] Robert Escriva and Robbert van Renesse. 2016. Consus: Taming the Paxi. *CoRR* (2016).

[25] Eli Gafni and Leslie Lamport. 2003. Disk paxos. *Distributed Computing* 16, 1 (2003), 1–20.

[26] Haryadi S Gunawi, Thanh Do, Joseph M Hellerstein, Ion Stoica, Dhruba Borthakur, and Jesse Robbins. 2011. *Failure as a service (FaaS): A cloud service for large-scale, online failure drills*. Technical Report UCB/EECS-2011-87.

[27] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *SoCC*.

[28] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. 2016. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages.. In *SoCC*.

[29] Seungyeop Han, Haichen Shen, Taesoo Kim, Arvind Krishnamurthy, Thomas E Anderson, and David Wetherall. 2015. MetaSync: File Synchronization Across Multiple Untrusted Storage Services.. In *USENIX ATC*.

[30] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. 2012. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM*.

[31] Zi Hu, Liang Zhu, Calvin Ardi, Ethan Katz-Bassett, Harsha V. Madhyastha, John Heidemann, and Minlan Yu. 2014. The Need for End-to-End Evaluation of Cloud Availability. In *PAM*.

[32] Qin Jia, Zhiming Shen, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. 2015. Supercloud: Opportunities and Challenges. *ACM SIGOPS Operating Systems Review* 49, 1 (2015), 137–141.

[33] Flavio Junqueira, Yanhua Mao, and Keith Marzullo. 2007. Classic Paxos vs. fast Paxos: Caveat emptor. In *HotDep*.

[34] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data center consistency. In *EuroSys*.

[35] Leslie Lamport. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[36] Leslie Lamport. 2006. Fast paxos. *Distributed Computing* 19, 2 (2006), 79–103.

[37] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary.. In *OSDI*.

[38] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. 2016. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *OSDI*.

[39] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *SOSP*.

[40] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage.. In *NSDI*.

[41] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *SOSP*.

[42] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *SOSP*.

[43] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. 2015. Minimizing commit latency of transactions in geo-replicated data stores. In *SIGMOD*.

[44] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *NSDI*.

[45] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. 2012. FairCloud: Sharing The Network In Cloud Computing. In *SIGCOMM*.

[46] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. 2015. Designing distributed systems using approximate synchrony in data center networks. In *NSDI*.

[47] Srinath Setty, Chunzhi Su, Jacob R. Lorch, Lidong Zhou, Hao Chen, Parveen Patel, and Jinglei Ren. 2016. Realizing the Fault-tolerance Promise of Cloud Storage Using Locks with Intent. In *OSDI*.

[48] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook White Paper* 5, 8 (2007).

[49] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *NSDI*.

[50] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based service level agreements for cloud storage. In *SOSP*.

[51] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. IOFlow: A Software-Defined Storage Architecture. In *SOSP*.

[52] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. 2012. Gnothi: Separating Data and Metadata for Efficient and Available Storage Replication.. In *USENIX ATC*.

[53] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. 2012. The Xen-Blanket: Virtualize Once, Run Everywhere. In *EuroSys*.

[54] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowstron. 2011. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*.

[55] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. 2013. SPANStore: Cost-Effective Geo-Replicated Storage Spanning Multiple Cloud Services. In *SOSP*.

[56] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. 2012. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *SIGCOMM*.

[57] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *SOSP*.

[58] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. 2013. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *SOSP*.