Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS

Wyatt Lloyd⋆, Michael J. Freedman⋆, Michael Kaminsky†, and David G. Andersen‡
⋆Princeton University, †Intel Labs, ‡Carnegie Mellon University

ABSTRACT

Geo-replicated, distributed data stores that support complex online applications, such as social networks, must provide an "always-on" experience where operations always complete with low latency. Today's systems often sacrifice strong consistency to achieve these goals, exposing inconsistencies to their clients and necessitating complex application logic. In this paper, we identify and define a consistency model—causal consistency with convergent conflict handling, or *causal+*—that is the strongest achieved under these constraints.

We present the design and implementation of COPS, a key-value store that delivers this consistency model across the wide-area. A key contribution of COPS is its scalability, which can enforce causal dependencies between keys stored across an entire cluster, rather than a single server like previous systems. The central approach in COPS is tracking and explicitly checking whether causal dependencies between keys are satisfied in the local cluster before exposing writes. Further, in COPS-GT, we introduce get transactions in order to obtain a consistent view of multiple keys without locking or blocking. Our evaluation shows that COPS completes operations in less than a millisecond, provides throughput similar to previous systems when using one server per cluster, and scales well as we increase the number of servers in each cluster. It also shows that COPS-GT provides similar latency, throughput, and scaling to COPS for common workloads.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Distributed Systems

General Terms

Design, Experimentation, Performance

Keywords

Key-value storage, causal+ consistency, scalable wide-area replication, ALPS systems, read transactions

1. INTRODUCTION

Distributed data stores are a fundamental building block of modern Internet services. Ideally, these data stores would be strongly consistent, always available for reads and writes, and able to continue operating during network partitions. The CAP Theorem, unfortunately, proves it impossible to create a system that achieves all three [13, 23]. Instead, modern web services have chosen overwhelmingly to embrace availability and partition tolerance at the cost of strong consistency [16, 20, 30]. This is perhaps not surprising, given that this choice also enables these systems to provide low latency for client operations and high scalability. Further, many of the earlier high-scale Internet services, typically focusing on web search, saw little reason for stronger consistency, although this position is changing with the rise of interactive services such as social networking applications [46]. We refer to systems with these four properties—**A**vailability, low **L**atency, **P**artition-tolerance, and high **S**calability—as ALPS systems.

Given that ALPS systems must sacrifice strong consistency (i.e., linearizability), we seek the strongest consistency model that is achievable under these constraints. Stronger consistency is desirable because it makes systems easier for a programmer to reason about. In this paper, we consider *causal consistency with convergent conflict handling*, which we refer to as *causal+ consistency*. Many previous systems believed to implement the weaker causal consistency [10, 41] actually implement the more useful causal+ consistency, though none do so in a scalable manner.

The causal component of causal+ consistency ensures that the data store respects the causal dependencies between operations [31]. Consider a scenario where a user uploads a picture to a web site, the picture is saved, and then a reference to it is added to that user's album. The reference "depends on" the picture being saved. Under causal+ consistency, these dependencies are always satisfied. Programmers never have to deal with the situation where they can get the reference to the picture but not the picture itself, unlike in systems with weaker guarantees, such as eventual consistency.

The convergent conflict handling component of causal+ consistency ensures that replicas never permanently diverge and that conflicting updates to the same key are dealt with identically at all sites. When combined with causal consistency, this property ensures that clients see only progressively newer versions of keys. In comparison, eventually consistent systems may expose versions out of order. By combining causal consistency and convergent conflict handling, causal+ consistency ensures clients see a causally-correct, conflict-free, and always-progressing data store.

Our COPS system (Clusters of Order-Preserving Servers) provides causal+ consistency and is designed to support complex online applications that are hosted from a small number of large-scale datacenters, each of which is composed of front-end servers (clients of

COPS) and back-end key-value data stores. COPS executes all put and get operations in the local datacenter in a linearizable fashion, and it then replicates data across datacenters in a causal+ consistent order in the background.

We detail two versions of our COPS system. The regular version, COPS, provides scalable causal+ consistency between individual items in the data store, even if their causal dependencies are spread across many different machines in the local datacenter. These consistency properties come at low cost: The performance and overhead of COPS is similar to prior systems, such as those based on log exchange [10, 41], even while providing much greater scalability.

We also detail an extended version of the system, COPS-GT, which also provides *get transactions* that give clients a consistent view of multiple keys. Get transactions are needed to obtain a consistent view of multiple keys, even in a fully-linearizable system. Our get transactions require no locks, are non-blocking, and take at most two parallel rounds of intra-datacenter requests. To the best of our knowledge, COPS-GT is the first ALPS system to achieve non-blocking scalable get transactions. These transactions do come at some cost: compared to the regular version of COPS, COPS-GT is less efficient for certain workloads (e.g., write-heavy) and is less robust to long network partitions and datacenter failures.

The scalability requirements for ALPS systems creates the largest distinction between COPS and prior causal+ consistent systems. Previous systems required that all data fit on a single machine [2, 12, 41] or that all data that potentially could be accessed together fit on a single machine [10]. In comparison, data stored in COPS can be spread across an arbitrary-sized datacenter, and dependencies (or get transactions) can stretch across many servers in the datacenter. To the best of our knowledge, COPS is the first scalable system to implement causal+ (and thus causal) consistency.

The contributions in this paper include:

- We explicitly identify four important properties of distributed data stores and use them to define ALPS systems.
- We name and formally define causal+ consistency.
- We present the design and implementation of COPS, a *scalable* system that efficiently realizes the causal+ consistency model.
- We present a non-blocking, lock-free get transaction algorithm in COPS-GT that provides clients with a consistent view of multiple keys in at most two rounds of local operations.
- We show through evaluation that COPS has low latency, high throughput, and scales well for all tested workloads; and that COPS-GT has similar properties for common workloads.

2. ALPS SYSTEMS AND TRADE-OFFS

We are interested in infrastructure that can support many of today's largest Internet services. In contrast with classical distributed storage systems that focused on local-area operation in the small, these services are typically characterized by wide-area deployments across a few to tens of datacenters, as illustrated in Figure 1. Each datacenter includes a set of application-level clients, as well as a back-end data store to which these clients read and write. For many applications—and the setting considered in the paper—data written in one datacenter is replicated to others.

Often, these clients are actually webservers that run code on behalf of remote browsers. Although this paper considers consistency from the perspective of the application client (i.e., the webserver), if the browser accesses a service through a single datacenter, as we expect, it will enjoy similar consistency guarantees.

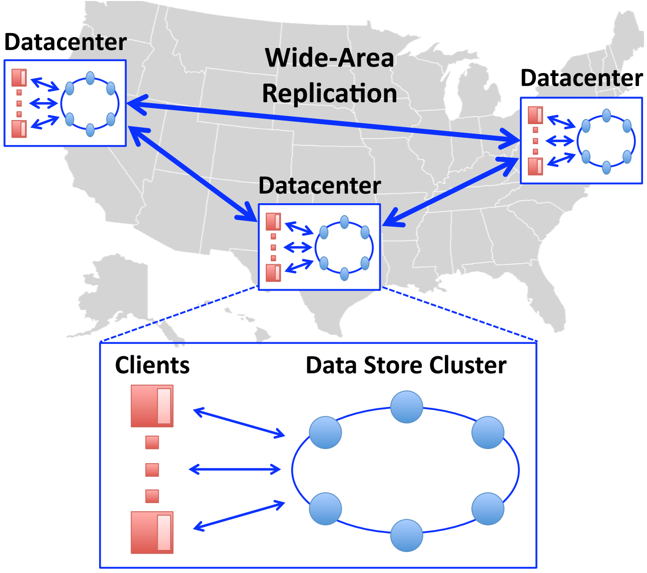


Figure 1: The general architecture of modern web services. Multiple geographically distributed datacenters each have application clients that read and write state from a data store that is distributed across all of the datacenters.

Such a distributed storage system has multiple, sometimes competing, goals: *availability*, *low latency*, and *partition tolerance* to provide an "always on" user experience [16]; *scalability* to adapt to increasing load and storage demands; and a sufficiently strong *consistency* model to simplify programming and provide users with the system behavior that they expect. In slightly more depth, the desirable properties include:

1. Availability. All operations issued to the data store complete successfully. No operation can block indefinitely or return an error signifying that data is unavailable.

2. Low Latency. Client operations complete "quickly." Commercial service-level objectives suggest average performance of a few milliseconds and worse-case performance (i.e., 99.9th percentile) of 10s or 100s of milliseconds [16].

3. Partition Tolerance. The data store continues to operate under network partitions, e.g., one separating datacenters in Asia from the United States.

4. High Scalability. The data store scales out linearly. Adding N resources to the system increases aggregate throughput and storage capacity by O(N).

5. Stronger Consistency. An ideal data store would provide *linearizability*—sometimes informally called *strong consistency*—which dictates that operations appear to take effect across the entire system at a single instance in time between the invocation and completion of the operation [26]. In a data store that provides linearizability, as soon as a client completes a write operation to an object in one datacenter, read operations to the same object in all other datacenters will reflect its newly written state. Linearizability simplifies programming—the distributed system provides a single, consistent image—and users experience the storage behavior they expect. Weaker, eventual consistency models, common in many large distributed systems, are less intuitive: Not only might subsequent reads not reflect the latest value, reads across multiple objects might reflect an incoherent mix of old and new values.

The CAP Theorem proves that a shared-data system that has availability and partition tolerance cannot achieve linearizability [13,

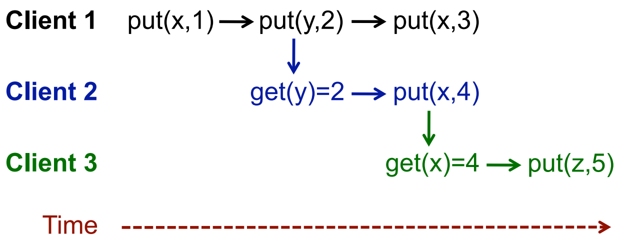


Figure 2: Graph showing the causal relationship between operations at a replica. An edge from *a* to *b* indicates that $a \rightsquigarrow b$, or *b* depends on *a*.

23]. Low latency—defined as latency less than the maximum wide-area delay between replicas—has also been proven incompatible with linearizability [34] and sequential consistency [8]. To balance between the requirements of ALPS systems and programmability we define an intermediate consistency model in the next section.

3. CAUSAL+ CONSISTENCY

To define *causal consistency with convergent conflict handling* (causal+ consistency), we first describe the abstract model over which it operates. We restrict our consideration to a key-value data store, with two basic operations: `put`(key,val) and `get`(key)=val. These are equivalent to write and read operations in a shared-memory system. Values are stored and retrieved from logical *replicas*, each of which hosts the entire key space. In our COPS system, a single *logical* replica corresponds to an entire local cluster of nodes.

An important concept in our model is the notion of *potential causality* [2, 31] between operations. Three rules define potential causality, denoted $\rightsquigarrow$:

1. **Execution Thread.** If a and b are two operations in a *single thread of execution*, then $a \rightsquigarrow b$ if operation a happens before operation b.
2. **Gets From.** If a is a `put` operation and b is a `get` operation that returns the value written by a, then $a \rightsquigarrow b$.
3. **Transitivity.** For operations a, b, and c, if $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.

These rules establish potential causality between operations within the same execution thread and between operations whose execution threads interacted through the data store. Our model, like many, does not allow threads to communicate directly, requiring instead that all communication occur through the data store.

The example execution in Figure 2 demonstrates all three rules. The *execution thread* rule gives `get`(y)=2 $\rightsquigarrow$ `put`(x,4); the *gets from* rule gives `put`(y,2) $\rightsquigarrow$ `get`(y)=2; and the *transitivity* rule gives `put`(y,2) $\rightsquigarrow$ `put`(x,4). Even though some operations follow `put`(x,3) in real time, no other operations depend on it, as none read the value it wrote nor follow it in the same thread of execution.

3.1 Definition

We define causal+ consistency as a combination of two properties: causal consistency and convergent conflict handling. We present intuitive definitions here and the formal definitions in Appendix A.

Causal consistency requires that values returned from `get` operations at a replica are consistent with the order defined by $\rightsquigarrow$ (causality) [2]. In other words, it must appear the operation that writes a value occurs after all operations that causally precede it. For example, in Figure 2, it must appear `put`(y,2) happened before `put`(x,4), which in turn happened before `put`(z,5). If client 2 saw `get`(x)=4 and then `get`(x)=1, causal consistency would be violated.

Causal consistency does not order concurrent operations. If $a \not\rightsquigarrow b$ and $b \not\rightsquigarrow a$, then a and b are concurrent. Normally, this allows increased efficiency in an implementation: Two unrelated put operations can be replicated in any order, avoiding the need for a serialization point between them. If, however, a and b are both puts to the same key, then they are in *conflict*.

Conflicts are undesirable for two reasons. First, because they are unordered by causal consistency, conflicts allow replicas to diverge forever [2]. For instance, if a is `put`(x,1) and b is `put`(x,2), then causal consistency allows one replica to forever return 1 for x and another replica to forever return 2 for x. Second, conflicts may represent an exceptional condition that requires special handling. For example, in a shopping cart application, if two people logged in to the same account concurrently add items to their cart, the desired result is to end up with both items in the cart.

Convergent conflict handling requires that all conflicting puts be handled in the same manner at all replicas, using a handler function h. This handler function h must be associative and commutative, so that replicas can handle conflicting writes in the order they receive them and that the results of these handlings will converge (e.g., one replica's $h(a, h(b, c))$ and another's $h(c, h(b, a))$ agree).

One common way to handle conflicting writes in a convergent fashion is the last-writer-wins rule (also called Thomas's write rule [50]), which declares one of the conflicting writes as having occurred later and has it overwrite the "earlier" write. Another common way to handle conflicting writes is to mark them as conflicting and require their resolution by some other means, e.g., through direct user intervention as in Coda [28], or through a programmed procedure as in Bayou [41] and Dynamo [16].

All potential forms of convergent conflict handling avoid the first issue—conflicting updates may continually diverge—by ensuring that replicas reach the same result after exchanging operations. On the other hand, the second issue with conflicts—applications may want special handling of conflicts—is only avoided by the use of more explicit *conflict resolution* procedures. These explicit procedures provide greater flexibility for applications, but require additional programmer complexity and/or performance overhead. Although COPS can be configured to detect conflicting updates explicitly and apply some application-defined resolution, the default version of COPS uses the last-writer-wins rule.

3.2 Causal+ vs. Other Consistency Models

The distributed systems literature defines several popular consistency models. In decreasing strength, they include: linearizability (or strong consistency) [26], which maintains a global, real-time ordering; sequential consistency [32], which ensures at least a global ordering; causal consistency [2], which ensures partial orderings between dependent operations; FIFO (PRAM) consistency [34], which only preserves the partial ordering of an execution thread, not between threads; per-key sequential consistency [15], which ensures that, for each individual key, all operations have a global order; and eventual consistency, a "catch-all" term used today suggesting eventual convergence to some type of agreement.

The causal+ consistency we introduce falls between sequential and causal consistency, as shown in Figure 3. It is weaker than sequential consistency, but sequential consistency is provably not achievable in an ALPS system. It is stronger than causal consistency

Linearizability > **Sequential** > Causal+ > Causal > FIFO
> Per-Key Sequential > Eventual

Figure 3: A spectrum of consistency models, with stronger models on the left. Bolded models are provably incompatible with ALPS systems.

and per-key sequential consistency, however, and it *is* achievable for ALPS systems. Mahajan et al. [35] have concurrently defined a similar strengthening of causal consistency; see Section 7 for details.

To illustrate the utility of the causal+ model, consider two examples. First, let Alice try to share a photo with Bob. Alice uploads the photo and then adds the photo to her album. Bob then checks Alice's album expecting to see her photo. Under causal and thus causal+ consistency, if the album has a reference to the photo, then Bob must be able to view the photo. Under per-key sequential and eventual consistency, it is possible for the album to have a reference to a photo that has not been written yet.

Second, consider an example where Carol and Dan both update the starting time for an event. The time was originally set for 9pm, Carol changed it to 8pm, and Dan concurrently changed it to 10pm. Regular causal consistency would allow two different replicas to forever return different times, even after receiving both put operations. Causal+ consistency requires that replicas handle this conflict in a convergent manner. If a last-writer-wins policy is used, then either Dan's 10pm or Carol's 8pm would win. If a more explicit conflict resolution policy is used, the key could be marked as in conflict and future gets on it could return both 8pm and 10pm with instructions to resolve the conflict.

If the data store was sequentially consistent or linearizable, it would still be possible for there to be two simultaneous updates to a key. In these stronger models, however, it is possible to implement mutual exclusion algorithms—such as the one suggested by Lamport in the original sequential consistency paper [32]—that can be used to avoid creating a conflict altogether.

3.3 Causal+ in COPS

We use two abstractions in the COPS system, *versions* and *dependencies*, to help us reason about causal+ consistency. We refer to the different values a key has as the *versions* of a key, which we denote $\text{key}_{\text{version}}$. In COPS, versions are assigned in a manner that ensures that if $x_i \rightsquigarrow y_j$ then $i < j$. Once a replica in COPS returns version i of a key, x_i, causal+ consistency ensures it will then only return that version or a causally later version (note that the handling of a conflict is causally later than the conflicting puts it resolves).[1] Thus, each replica in COPS always returns non-decreasing versions of a key. We refer to this as causal+ consistency's *progressing property*.

Causal consistency dictates that all operations that causally precede a given operations must appear to take effect before it. In other words, if $x_i \rightsquigarrow y_j$, then x_i must be written before y_j. We call these preceding values *dependencies*. More formally, we say y_j ***depends on*** x_i if and only if $\texttt{put}(x_i) \rightsquigarrow \texttt{put}(y_j)$. These dependencies are in essence the reverse of the causal ordering of writes. COPS provides causal+ consistency during replication by writing a version only after writing all of its dependencies.

3.4 Scalable Causality

To our knowledge, this paper is the first to name and formally define causal+ consistency. Interestingly, several previous systems [10, 41] believed to achieve causal consistency in fact achieved the stronger guarantees of causal+ consistency.

These systems were not designed to and do not provide *scalable* causal (or causal+) consistency, however, as they all use a form of log serialization and exchange. All operations at a logical replica are written to a single log in serialized order, commonly marked with a version vector [40]. Different replicas then exchange these logs, using version vectors to establish potential causality and detect concurrency between operations at different replicas.

Log-exchange-based serialization inhibits replica scalability, as it relies on a single serialization point in each replica to establish ordering. Thus, either causal dependencies between keys are limited to the set of keys that can be stored on one node [10, 15, 30, 41], or a single node (or replicated state machine) must provide a commit ordering and log for all operations across a cluster.

As we show below, COPS achieves scalability by taking a different approach. Nodes in each datacenter are responsible for different partitions of the keyspace, but the system can track and enforce dependencies between keys stored on different nodes. COPS explicitly encodes dependencies in metadata associated with each key's version. When keys are replicated remotely, the receiving datacenter performs dependency checks before committing the incoming version.

4. SYSTEM DESIGN OF COPS

COPS is a distributed storage system that realizes causal+ consistency and possesses the desired ALPS properties. There are two distinct versions of the system. The first, which we refer to simply as COPS, provides a data store that is causal+ consistent. The second, called COPS-GT, provides a superset of this functionality by also introducing support for *get transactions*. With get transactions, clients request a set of keys and the data store replies with a consistent snapshot of corresponding values. Because of the additional metadata needed to enforce the consistency properties of get transactions, a given deployment must run exclusively as COPS or COPS-GT.

4.1 Overview of COPS

COPS is a key-value storage system designed to run across a small number of datacenters, as illustrated in Figure 4. Each datacenter has a local COPS *cluster* with a complete replica of the stored data.[2] A *client* of COPS is an application that uses the COPS *client library* to call directly into the COPS key-value store. Clients communicate only with their local COPS cluster running in the same datacenter.

Each local COPS cluster is set up as a linearizable (strongly consistent) key-value store [5, 48]. Linearizable systems can be implemented scalably by partitioning the keyspace into N linearizable partitions (each of which can reside on a single node or a single chain of nodes) and having clients access each partition independently. The composability of linearizability [26] ensures that the resulting system as a whole remains linearizable. Linearizability is acceptable locally because we expect very low latency and no

[1] To see this, consider by contradiction the following scenario: assume a replica first returns x_i and then x_k, where $i \neq k$ and $x_i \not\rightsquigarrow x_k$. Causal consistency ensures that if x_k is returned after x_i, then $x_k \not\rightsquigarrow x_i$, and so x_i and x_k conflict. But, if x_i and x_k conflict, then convergent conflict handling ensures that as soon as both are present at a replica, their handling $h(x_i, x_k)$, which is causally after both, will be returned instead of either x_i or x_k, which contradicts our assumption.

[2] The assumption of full replication simplifies our presentation, though one could imagine clusters that replicate only part of the total data store and sacrifice low latency for the rest (according to configuration rules).

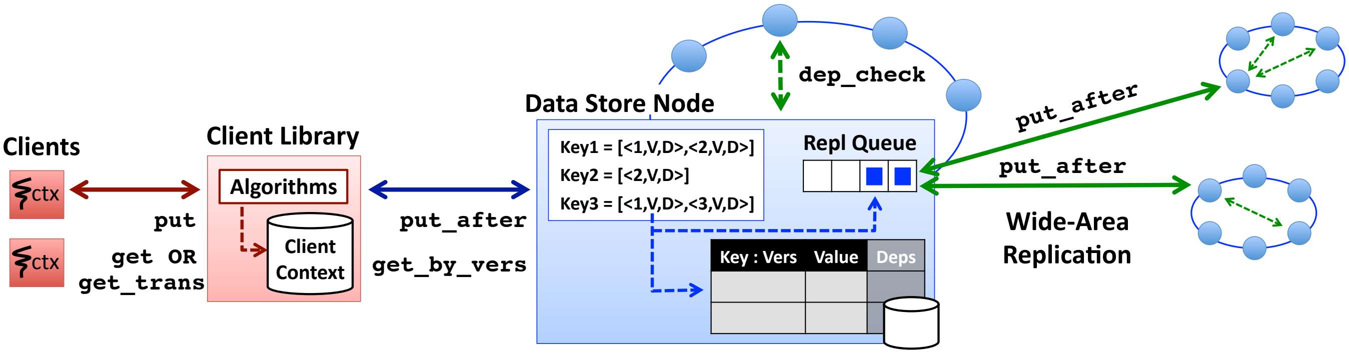


Figure 4: The COPS architecture. A client library exposes a put/get interface to its clients and ensures operations are properly labeled with causal dependencies. A key-value store replicates data between clusters, ensures writes are committed in their local cluster only after their dependencies have been satisfied, and in COPS-GT, stores multiple versions of each key along with dependency metadata.

partitions within a cluster—especially with the trend towards redundant paths in modern datacenter networks [3, 24]—unlike in the wide-area. On the other hand, replication *between* COPS clusters happens asynchronously to ensure low latency for client operations and availability in the face of external partitions.

System Components. COPS is composed of two main software components:

- *Key-value store.* The basic building block in COPS is a standard key-value store that provides linearizable operations on keys. COPS extends the standard key-value store in two ways, and COPS-GT adds a third extension.
  1. Each key-value pair has associated metadata. In COPS, this metadata is a version number. In COPS-GT, it is both a version number and a list of dependencies (other keys and their respective versions).
  2. The key-value store exports three additional operations as part of its key-value interface: `get_by_version`, `put_after`, and `dep_check`, each described below. These operations enable the COPS client library and an asynchronous replication process that supports causal+ consistency and get transactions.
  3. For COPS-GT, the system keeps around old versions of key-value pairs, not just the most recent `put`, to ensure that it can provide get transactions. Maintaining old versions is discussed further in Section 4.3.
- *Client library.* The client library exports two main operations to applications: reads via `get` (in COPS) or `get_trans` (in COPS-GT), and writes via `put`.[3] The client library also maintains state about a client's current dependencies through a *context* parameter in the client library API.

Goals. The COPS design strives to provide causal+ consistency with resource and performance overhead similar to existing eventually consistent systems. COPS and COPS-GT must therefore:

- *Minimize overhead of consistency-preserving replication.* COPS must ensure that values are replicated between clusters in a causal+ consistent manner. A naive implementation, however, would require checks on all of a value's dependencies. We present a mechanism that requires only a small number of such checks by leveraging the graph structure inherent to causal dependencies.
- *(COPS-GT) Minimize space requirements.* COPS-GT stores (potentially) multiple versions of each key, along with their associated dependency metadata. COPS-GT uses aggressive garbage collection to prune old state (see Section 5.1).
- *(COPS-GT) Ensure fast `get_trans` operations.* The get transactions in COPS-GT ensure that the set of returned values are causal+ consistent (all dependencies are satisfied). A naive algorithm could block and/or take an unbounded number of get rounds to complete. Both situations are incompatible with the availability and low latency goals of ALPS systems; we present an algorithm for `get_trans` that completes in at most two rounds of local `get_by_version` operations.

4.2 The COPS Key-Value Store

Unlike traditional ⟨*key*, *val*⟩–tuple stores, COPS must track the versions of written values, as well as their dependencies in the case of COPS-GT. In COPS, the system stores the most recent version number and value for each key. In COPS-GT, the system maps each key to a list of version entries, each consisting of ⟨*version*, *value*, *deps*⟩. The *deps* field is a list of the version's zero or more dependencies; each dependency is a ⟨*key*, *version*⟩ pair.

Each COPS cluster maintains its own copy of the key-value store. For scalability, our implementation partitions the keyspace across a cluster's nodes using consistent hashing [27], through other techniques (e.g., directory-based approaches [6, 21]) are also possible. For fault tolerance, each key is replicated across a small number of nodes using chain replication [5, 48, 51]. Gets and puts are linearizable across the nodes in the cluster. Operations return to the client library as soon as they execute in the *local* cluster; operations between clusters occur asynchronously.

Every key stored in COPS has one *primary* node in each cluster. We term the set of primary nodes for a key across all clusters as the *equivalent nodes* for that key. In practice, COPS's consistent hashing assigns each node responsibility for a few different key ranges. Key ranges may have different sizes and node mappings in different datacenters, but the total number of equivalent nodes with which a given node needs to communicate is proportional to the number of datacenters (i.e., communication is *not* all-to-all between nodes in different datacenters).

After a write completes locally, the primary node places it in a replication queue, from which it is sent asynchronously to remote equivalent nodes. Those nodes, in turn, wait until the value's dependencies are satisfied in their local cluster before locally committing

[3] This paper uses different fixed-width fonts for client-facing API calls (e.g., `get`) and data store API calls (e.g., `get_by_version`).

```
# Alice's Photo Upload
ctx_id = createContext()   // Alice logs in
put(Photo, "Portuguese Coast", ctx_id)
put(Album, "add &Photo", ctx_id)
deleteContext(ctx_id)      // Alice logs out

# Bob's Photo View
ctx_id = createContext()   // Bob logs in
"&Photo" ← get(Album, ctx_id)
"Portuguese Coast" ← get(Photo, ctx_id)
deleteContext(ctx_id)      // Bob logs out
```

Figure 5: Snippets of pseudocode using the COPS programmer interface for the photo upload scenario from Section 3.2. When using COPS-GT, each `get` would instead be a `get_trans` on a single key.

the value. This dependency checking mechanism ensures writes happen in a causally consistent order and reads never block.

4.3 Client Library and Interface

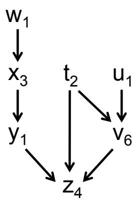
The COPS client library provides a simple and intuitive programming interface. Figure 5 illustrates the use of this interface for the photo upload scenario. The client API consists of four operations:

1. *ctx_id* ← `createContext`()
2. bool ← `deleteContext`(*ctx_id*)
3. bool ← `put` (*key*, *value*, *ctx_id*)
4. *value* ← `get` (*key*, *ctx_id*) [In COPS]
   or
4. ⟨*values*⟩ ← `get_trans` (⟨*keys*⟩, *ctx_id*) [In COPS-GT]

The client API differs from a traditional key-value interface in two ways. First, COPS-GT provides `get_trans`, which returns a consistent view of multiple key-value pairs in a single call. Second, all functions take a context argument, which the library uses internally to track causal dependencies across each client's operations [49]. The context defines the causal+ "thread of execution." A single process may contain many separate threads of execution (e.g., a web server concurrently serving 1000s of independent connections). By separating different threads of execution, COPS avoids false dependencies that would result from intermixing them.

We next describe the state kept by the client library in COPS-GT to enforce consistency in get transactions. We then show how COPS can store significantly less dependency state.

COPS-GT Client Library. The client library in COPS-GT stores the client's context in a table of ⟨*key*, *version*, *deps*⟩ entries. Clients reference their context using a *context ID* (*ctx_id*) in the API.[4] When a client gets a key from the data store, the library adds this key and its causal dependencies to the context. When a client puts a value, the library sets the put's dependencies to the most recent version of each key in the current context. A successful put into the data store returns the version number v assigned to the written value. The client library then adds this new entry, ⟨*key*, v, D⟩, to the context.



Val	Nearest Deps	All Deps
t_2	-	-
u_1	-	-
v_6	t_2,u_1	t_2,u_1
w_1	-	-
x_3	w_1	w_1
y_1	x_3	x_3,w_1
z_4	y_1,v_6	t_2,u_1,v_6,w_1,x_3,y_1

Figure 6: A sample graph of causal dependencies for a client context. Arrows indicate causal relationships (e.g., x_3 depends on w_1). The table lists all dependencies for each value and the "nearest" dependencies used to minimize dependency checks.

The context therefore includes all values previously read or written in the client's session, as well as all of those dependencies' dependencies, as illustrated in Figure 6. This raises two concerns about the potential size of this causality graph: (i) state requirements for storing these dependencies, both in the client library and in the data store, and (ii) the number of potential checks that must occur when replicating writes between clusters, in order to ensure causal consistency. To mitigate the client and data-store state required to track dependencies, COPS-GT provides garbage collection, described in Section 5.1, that removes dependencies once they are committed to all COPS replicas.

To reduce the number of dependency checks during replication, the client library identifies several potential optimizations for servers to use. Consider the graph in Figure 6. y_1 depends on x_3 and, by transitivity, on w_1. If the storage node committing y_1 determines that x_3 has been committed, then it can infer that w_1 has also been committed, and thus, need not check for it explicitly. Similarly, while z_4 depends directly on t_2 and v_6, the committing node needs only check v_6, because v_6 itself depends on t_2.

We term dependencies that *must* be checked the *nearest* dependencies, listed in the table in Figure 6.[5] To enable servers to use these optimizations, the client library first computes the nearest dependencies within the write's dependency list and marks them accordingly when issuing the write.

The nearest dependencies are sufficient for the key-value store to provide causal+ consistency; the full dependency list is only needed to provide `get_trans` operations in COPS-GT.

COPS Client Library. The client library in COPS requires significantly less state and complexity because it only needs to learn the nearest, rather than all, dependencies. Accordingly, it does not store or even retrieve the dependencies of any value it gets: The retrieved value is nearer than any of its dependencies, rendering them unnecessary.

Thus, the COPS client library stores only ⟨*key*, *version*⟩ entries. For a `get` operation, the retrieved ⟨*key*, *version*⟩ is added to the context. For a put operation, the library uses the current context as the nearest dependencies, clears the context, and then repopulates it with only this put. This put depends on all previous key-version pairs and thus is nearer than them.

4.4 Writing Values in COPS and COPS-GT

Building on our description of the client library and key-value store, we now walk through the steps involved in writing a value to COPS. All writes in COPS first go to the client's local cluster and then propagate asynchronously to remote clusters. The key-value store exports a single API call to provide both operations:

⟨bool,*vers*⟩ ← `put_after` (*key*, *val*, [*deps*], *nearest*, *vers*=∅)

[4] Maintaining state in the library and passing in an ID was a design choice; one could also encode the entire context table as an opaque blob and pass it between client and library so that the library is stateless.

[5] In graph-theoretic terms, the nearest dependencies of a value are those in the causality graph with a *longest* path to the value of length one.

Writes to the local cluster. When a client calls put (*key*,*val*,*ctx_id*), the library computes the complete set of dependencies *deps*, and identifies some of those dependency tuples as the value's *nearest* ones. The library then calls put_after without the *version* argument (i.e., it sets *version*=∅). In COPS-GT, the library includes *deps* in the put_after call because dependencies must be stored with the value; in COPS, the library only needs to include *nearest* and does not include *deps*.[6] The key's primary storage node in the local cluster assigns the key a version number and returns it to the client library. We restrict each client to a single outstanding put; this is necessary because later puts must know the version numbers of earlier puts so they may depend on them.

The put_after operation ensures that *val* is committed to each cluster only *after* all of the entries in its dependency list have been written. In the client's local cluster, this property holds automatically, as the local store provides linearizability. (If y depends on x, then put(x) must have been committed before put(y) was issued.) This is not true in remote clusters, however, which we discuss below.

The primary storage node uses a Lamport timestamp [31] to assign a unique version number to each update. The node sets the version number's high-order bits to its Lamport clock and the low-order bits to its unique node identifier. Lamport timestamps allow COPS to derive a single global order over all writes for each key. This order implicitly implements the last-writer-wins convergent conflict handling policy. COPS is also capable of explicitly detecting and resolving conflicts, which we discuss in Section 5.3. Note that because Lamport timestamps provide a partial ordering of all distributed events in a way that respects potential causality, this global ordering is compatible with COPS's causal consistency.

Write replication between clusters. After a write commits locally, the primary storage node asynchronously replicates that write to its equivalent nodes in different clusters using a stream of put_after operations; here, however, the primary node includes the key's version number in the put_after call. As with local put_after calls, the *deps* argument is included in COPS-GT, and not included in COPS. This approach scales well and avoids the need for a single serialization point, but requires the remote nodes receiving updates to commit an update only after its dependencies have been committed to the same cluster.

To ensure this property, a node that receives a put_after request from another cluster must determine if the value's *nearest* dependencies have already been satisfied locally. It does so by issuing a check to the local nodes responsible for the those dependencies:

bool ← dep_check (*key*, *version*)

When a node receives a dep_check, it examines its local state to determine if the dependency value has already been written. If so, it immediately responds to the operation. If not, it blocks until the needed version has been written.

If all dep_check operations on the nearest dependencies succeed, the node handling the put_after request commits the written value, making it available to other reads and writes in its local cluster. (If any dep_check operation times out the node handling the put_after reissues it, potentially to a new node if a failure occurred.) The way that *nearest* dependencies are computed ensures that *all* dependencies have been satisfied before the value is committed, which in turn ensures causal consistency.

4.5 Reading Values in COPS

Like writes, reads are satisfied in the local cluster. Clients call the get library function with the appropriate context; the library in turn issues a read to the node responsible for the key in the local cluster:

⟨*value*, *version*, *deps*⟩ ← get_by_version (*key*, *version*=LATEST)

This read can request either the latest version of the key or a specific older one. Requesting the latest version is equivalent to a regular single-key get; requesting a specific version is necessary to enable get transactions. Accordingly, get_by_version operations in COPS always request the latest version. Upon receiving a response, the client library adds the ⟨*key*,*version*[,*deps*]⟩ tuple to the client context, and returns *value* to the calling code. The *deps* are stored only in COPS-GT, not in COPS.

4.6 Get Transactions in COPS-GT

The COPS-GT client library provides a get_trans interface because reading a set of dependent keys using a single-key get interface cannot ensure causal+ consistency, even though the data store itself is causal+ consistent. We demonstrate this problem by extending the photo album example to include access control, whereby Alice first changes her album ACL to "friends only", and then writes a new description of her travels and adds more photos to the album.

A natural (but incorrect!) implementation of code to read Alice's album might (1) fetch the ACL, (2) check permissions, and (3) fetch the album description and photos. This approach contains a straightforward "time-to-check-to-time-to-use" race condition: when Eve accesses the album, her get(ACL) might return the old ACL, which permitted anyone (including Eve) to read it, but her get(album contents) might return the "friends only" version.

One straw-man solution is to require that clients issue single-key get operations in the reverse order of their causal dependencies: The above problem would not have occurred if the client executed get(album) before get(ACL). This solution, however, is *also* incorrect. Imagine that after updating her album, Alice decided that some photographs were too personal, so she (3) deleted those photos and rewrote the description, and then (4) marked the ACL open again. This straw-man has a different time-of-check-to-time-of-use error, where get(album) retrieves the private album, and the subsequent get(ACL) retrieves the "public" ACL. In short, there is no correct canonical ordering of the ACL and the album entries.

Instead, a better programming interface would allow the client to obtain a causal+ consistent view of multiple keys. The standard way to achieve such a guarantee is to read and write all related keys in a transaction; this, however, requires a single serialization point for all grouped keys, which COPS avoids for greater scalability and simplicity. Instead, COPS allows keys to be written independently (with explicit dependencies in metadata), and provides a get_trans operation for retrieving a consistent view of multiple keys.

Get transactions. To retrieve multiple values in a causal+ consistent manner, a client calls get_trans with the desired set of keys, e.g., get_trans(⟨ACL, album⟩). Depending on when and where it was issued, this get transaction can return different combinations of ACL and album, but never ⟨$\text{ACL}_{\text{public}}$, $\text{Album}_{\text{personal}}$⟩.

The COPS client library implements the get transactions algorithm in two rounds, shown in Figure 7. In the first round, the library issues n concurrent get_by_version operations to the local cluster, one for each key the client listed in get_trans. Because

[6] We use bracket notation ([]) to indicate an argument is optional; the optional arguments are used in COPS-GT, but not in COPS.

```
# @param  keys    list of keys
# @param  ctx_id  context id
# @return values  list of values

function get_trans(keys, ctx_id):
  # Get keys in parallel (first round)
  for k in keys
    results[k] = get_by_version(k, LATEST)

  # Calculate causally correct versions (ccv)
  for k in keys
    ccv[k] = max(ccv[k], results[k].vers)
    for dep in results[k].deps
      if dep.key in keys
        ccv[dep.key] = max(ccv[dep.key], dep.vers)

  # Get needed ccvs in parallel (second round)
  for k in keys
    if ccv[k] > results[k].vers
      results[k] = get_by_version(k, ccv[k])

  # Update the metadata stored in the context
  update_context(results, ctx_id)

  # Return only the values to the client
  return extract_values(results)
```

Figure 7: Pseudocode for the `get_trans` algorithm.

COPS-GT commits writes locally, the local data store guarantees that each of these explicitly listed keys' dependencies are already satisfied—that is, they have been written locally and reads on them will immediately return. These explicitly listed, independently retrieved values, however, may not be consistent with one another, as shown above. Each `get_by_version` operation returns a ⟨*value*, *version*, *deps*⟩ tuple, where *deps* is a list of keys and versions. The client library then examines every dependency entry ⟨*key*, *version*⟩. The causal dependencies for that result are satisfied if either the client did not request the dependent key, or if it did, the version it retrieved was ≥ the version in the dependency list.

For all keys that are not satisfied, the library issues a second round of concurrent `get_by_version` operations. The version requested will be the newest version seen in any dependency list from the first round. These versions satisfy all causal dependencies from the first round because they are ≥ the needed versions. In addition, because dependencies are transitive and these second-round versions are all depended on by versions retrieved in the first round, they do not introduce any new dependencies that need to be satisfied. This algorithm allows `get_trans` to return a consistent view of the data store as of the time of the latest timestamp retrieved in first round.

The second round happens only when the client must read newer versions than those retrieved in the first round. This case occurs only if keys involved in the get transaction are updated during the first round. Thus, we expect the second round to be rare. In our example, if Eve issues a `get_trans` concurrent with Alice's writes, the algorithms first round of gets might retrieve the public ACL and the private album. The private album, however, depends on the "friends only" ACL, so the second round would fetch this newer version of the ACL, allowing `get_trans` to return a causal+ consistent set of values to the client.

The causal+ consistency of the data store provides two important properties for the get transaction algorithm's second round. First, the `get_by_version` requests will succeed immediately, as the requested version must already exist in the local cluster. Second, the new `get_by_version` requests will not introduce any new dependencies, as those dependencies were already known in the first round due to transitivity. This second property demonstrates why the get transaction algorithm specifies an explicit version in its second round, rather than just getting the latest: Otherwise, in the face of concurrent writes, a newer version could introduce still newer dependencies, which could continue indefinitely.

5. GARBAGE, FAULTS, AND CONFLICTS

This section describes three important aspects of COPS and COPS-GT: their garbage collection subsystems, which reduce the amount of extra state in the system; their fault tolerant design for client, node, and datacenter failures; and their conflict detection mechanisms.

5.1 Garbage Collection Subsystem

COPS and COPS-GT clients store metadata; COPS-GT servers additionally keeps multiple versions of keys and dependencies. Without intervention, the space footprint of the system would grow without bound as keys are updated and inserted. The COPS garbage collection subsystem deletes unneeded state, keeping the total system size in check. Section 6 evaluates the overhead of maintaining and transmitting this additional metadata.

Version Garbage Collection. (COPS-GT only)
What is stored: COPS-GT stores multiple versions of each key to answer `get_by_version` requests from clients.
Why it can be cleaned: The `get_trans` algorithm limits the number of versions needed to complete a get transaction. The algorithm's second round issues `get_by_version` requests only for versions later than those returned in the first round. To enable prompt garbage collection, COPS-GT limits the total running time of `get_trans` through a configurable parameter, *trans_time* (set to 5 seconds in our implementation). (If the timeout fires, the client library will restart the `get_trans` call and satisfy the transaction with newer versions of the keys; we expect this to happen only if multiple nodes in a cluster crash.)
When it can be cleaned: After a new version of a key is written, COPS-GT only needs to keep the old version around for *trans_time* plus a small delta for clock skew. After this time, no `get_by_version` call will subsequently request the old version, and the garbage collector can remove it.
Space Overhead: The space overhead is bounded by the number of old versions that can be created within the *trans_time*. This number is determined by the maximum write throughput that the node can sustain. Our implementation sustains 105MB/s of write traffic per node, requiring (potentially) a non-prohibitive extra 525MB of buffer space to hold old versions. This overhead is per-machine and does not grow with the cluster size or the number of datacenters.

Dependency Garbage Collection. (COPS-GT only)
What is stored: Dependencies are stored to allow get transactions to obtain a consistent view of the data store.
Why it can be cleaned: COPS-GT can garbage collect these dependencies once the versions associated with old dependencies are no longer needed for correctness in get transaction operations.

To illustrate when get transaction operations no longer need dependencies, consider value z_2 that depends on x_2 and y_2. A get transaction of x, y, and z requires that if z_2 is returned, then $x_{\geq 2}$ and $y_{\geq 2}$ must be returned as well. Causal consistency ensures that x_2 and y_2 are written before z_2 is written. Causal+ consistency's progressing property ensures that once x_2 and y_2 are written, then either they or some later version will always be returned by a get operation. Thus, once z_2 has been written in all datacenters and the *trans_time* has passed, any get transaction returning z_2 will return $x_{\geq 2}$ and $y_{\geq 2}$, and thus z_2's dependencies can be garbage collected.

When it can be cleaned: After *trans_time* seconds after a value has been committed in *all* datacenters, COPS-GT can clean a value's dependencies. (Recall that committed enforces that its dependencies have been satisfied.) Both COPS and COPS-GT can further set the value's *never-depend* flag, discussed below. To clean dependencies each remote datacenter notifies the originating datacenter when the write has committed and the timeout period has elapsed. Once all datacenters confirm, the originating datacenter cleans its own dependencies and informs the others to do likewise. To minimize bandwidth devoted to cleaning dependencies, a replica only notifies the originating datacenter if this version of a key is the newest after *trans_time* seconds; if it is not, there is no need to collect the dependencies because the entire version will be collected.[7]

Space Overhead: Under normal operation, dependencies are garbage collected after *trans_time* plus a round-trip time. Dependencies are only collected on the most recent version of the key; older versions of keys are already garbage collected as described above.

During a partition, dependencies on the most recent versions of keys cannot be collected. This is a limitation of COPS-GT, although we expect long partitions to be rare. To illustrate why this concession is necessary for get transaction correctness, consider value b_2 that depends on value a_2: if b_2's dependence on a_2 is prematurely collected, some later value c_2 that causally depends on b_2—and thus on a_2—could be written without the explicit dependence on a_2. Then, if a_2, b_2, and c_2 are all replicated to a datacenter in short order, the first round of a get transaction could obtain a_1, an earlier version of a, with c_2, and then return these two values to the client, precisely because it did not know c_2 depends on the newer a_2.

Client Metadata Garbage Collection. (COPS + COPS-GT)

What is Stored: The COPS client library tracks all operations during a client session (single thread of execution) using the *ctx_id* passed with all operation. In contrast to the dependency information discussed above which resides in the key-value store itself, the dependencies discussed here are part of the client metadata and are store in the client library. In both systems, each `get` since the last `put` adds another nearest dependency. Additionally in COPS-GT, all new values and their dependencies returned in `get_trans` operations and all `put` operations add normal dependencies. If a client session lasts for a long time, the number of dependencies attached to updates will grow large, increasing the size of the dependency metadata that COPS needs to store.

Why it can be cleaned: As with the dependency tracking above, clients need to track dependencies only until they are guaranteed to be satisfied everywhere.

When it can be cleaned: COPS reduces the size of this client state (the context) in two ways. First, as noted above, once a `put_after` commits successfully to all datacenters, COPS flags that key version as *never-depend*, in order to indicate that clients need not express a dependence upon it. `get_by_version` results include this flag, and the client library will immediately remove a *never-depend* item from the list of dependencies in the client context. Furthermore, this process is transitive: Anything that a *never-depend* key depended on must have been flagged *never-depend*, so it too can be garbage collected from the context.

Second, the COPS storage nodes remove unnecessary dependencies from `put_after` operations. When a node receives a `put_after`, it checks each item in the dependency list and removes items with version numbers older than a *global checkpoint time*. This checkpoint time is the newest Lamport timestamp that is satisfied at *all* nodes across the entire system. The COPS key-value store returns this checkpoint time to the client library (e.g., in response to a `put_after`), allowing the library to clean these dependencies from the context.[8]

To compute the global checkpoint time, each storage node first determines the oldest Lamport timestamp of any *pending* `put_after` in the key range for which it is primary. (In other words, it determines the timestamp of its oldest key that is not guaranteed to be satisfied at all replicas.) It then contacts its equivalent nodes in other datacenters (those nodes that handle the same key range). The nodes pair-wise exchange their minimum Lamport times, remembering the oldest observed Lamport clock of any of the replicas. At the conclusion of this step, all datacenters have the same information: each node knows the globally oldest Lamport timestamp in its key range. The nodes within a datacenter then gossip around the per-range minimums to find the minimum Lamport timestamp observed by any one of them. This periodic procedure is done 10 times a second in our implementation and has no noticeable impact on performance.

5.2 Fault Tolerance

COPS is resilient to client, node, and datacenter failures. For the following discussion, we assume that failures are fail-stop: components halt in response to a failure instead of operating incorrectly or maliciously, and failures are detectable.

Client Failures. COPS's key-value interface means that each client request (through the library) is handled independently and atomically by the data store. From the storage system's perspective, if a client fails, it simply stops issuing new requests; no recovery is necessary. From a client's perspective, COPS's dependency tracking makes it easier to handle failures of other clients, by ensuring properties such as referential integrity. Consider the photo and album example: If a client fails after writing the photo, but before writing a reference to the photo, the data store will still be in a consistent state. There will never be an instance of the reference to the photo without the photo itself already being written.

Key-Value Node Failures. COPS can use any underlying fault-tolerant linearizable key-value store. We built our system on top of independent clusters of FAWN-KV [5] nodes, which use chain replication [51] *within* a cluster to mask node failures. Accordingly, we describe how COPS can use chain replication to provide tolerance to node failures.

Similar to the design of FAWN-KV, each data item is stored in a chain of R consecutive nodes along the consistent hashing ring. `put_after` operations are sent to the head of the appropriate chain, propagate along the chain, and then commit at the tail, which then acknowledges the operation. `get_by_version` operations are sent to the tail, which responds directly.

Server-issued operations are slightly more involved because they are issued from and processed by different chains of nodes. The tail in the local cluster replicates `put_after` operations to the head in each remote datacenter. The remote heads then send `dep_check` operations, which are essentially read operations, to the appropriate

[7] We are currently investigating if collecting dependencies in this manner provides a significant enough benefit over collecting them after the global checkpoint time (discussed below) to justify its messaging cost.

[8] Because of outstanding reads, clients and servers must also wait *trans_time* seconds before they can use a new global checkpoint time.

tails in their local cluster. Once these return (if the operation does not return, a timeout will fire and the dep_check will be reissued), the remote head propagates the value down the (remote) chain to the remote tail, which commits the value and acknowledges the operation back to the originating datacenter.

Dependency garbage collection follows a similar pattern of interlocking chains, though we omit details for brevity. Version garbage collection is done locally on each node and can operate as in the single node case. Calculation of the global checkpoint time, for client metadata garbage collection, operates normally with each tail updating its corresponding key range minimums.

Datacenter Failures. The partition-tolerant design of COPS also provides resiliency to entire datacenter failures (or partitions). In the face of such failures, COPS continues to operate as normal, with a few key differences.

First, any put_after operations that originated in the failed datacenter, but which were not yet copied out, will be lost. This is an inevitable cost of allowing low-latency local writes that return faster than the propagation delay between datacenters. If the datacenter is only partitioned and has not failed, no writes will be lost. Instead, they will only be delayed until the partition heals.[9]

Second, the storage required for replication queues in the active datacenters will grow. They will be unable to send put_after operations to the failed datacenter, and thus COPS will be unable to garbage collect those dependencies. The system administrator has two options: allow the queues to grow if the partition is likely to heal soon, or reconfigure COPS to no longer use the failed datacenter.

Third, in COPS-GT, dependency garbage collection cannot continue in the face of a datacenter failure, until either the partition is healed or the system is reconfigured to exclude the failed datacenter.

5.3 Conflict Detection

Conflicts occur when there are two "simultaneous" (i.e., not in the same context/thread of execution) writes to a given key. The default COPS system avoids conflict detection using a last-writer-wins strategy. The "last" write is determined by comparing version numbers, and allows us to avoid conflict detection for increased simplicity and efficiency. We believe this behavior is useful for many applications. There are other applications, however, that become simpler to reason about and program with a more explicit conflict-detection scheme. For these applications, COPS can be configured to detect conflicting operations and then invoke some application-specific convergent conflict handler.

COPS with conflict detection, which we refer to as COPS-CD, adds three new components to the system. First, all put operations carry with them *previous version* metadata, which indicates the most recent previous version of the key that was visible at the local cluster at the time of the write (this previous version may be null). Second, all put operations now have an implicit dependency on that previous version, which ensures that a new version will only be written after its previous version. This implicit dependency entails an additional dep_check operation, though this has low overhead and always executes on the local machine. Third, COPS-CD has an application-specified convergent conflict handler that is invoked when a conflict is detected.

System	Causal+	Scalable	Get Trans
LOG	Yes	No	No
COPS	Yes	Yes	No
COPS-GT	Yes	Yes	Yes

Table 1: Summary of three systems under comparison.

COPS-CD follows a simple procedure to determine if a put operation *new* to a key (with previous version *prev*) is in conflict with the key's current visible version *curr*:

$prev \neq curr$ if and only if *new* and *curr* conflict.

We omit a full proof, but present the intuition here. In the forward direction, we know that *prev* must be written before *new*, $prev \neq curr$, and that for *curr* to be visible instead of *prev*, we must have $curr > prev$ by the progressing property of causal+. But because *prev* is the most recent causally previous version of *new*, we can conclude $curr \not\rightsquigarrow new$. Further, because *curr* was written before *new*, it cannot be causally after it, so $new \not\rightsquigarrow curr$ and thus they conflict. In the reverse direction, if *new* and *curr* conflict, then $curr \not\rightsquigarrow new$. By definition, $prev \rightsquigarrow new$, and thus $curr \neq prev$.

6. EVALUATION

This section presents an evaluation of COPS and COPS-GT using microbenchmarks that establish baseline system latency and throughput, a sensitivity analysis that explores the impact of different parameters that characterize a dynamic workload, and larger end-to-end experiments that show scalable causal+ consistency.

6.1 Implementation and Experimental Setup

COPS is approximately 13,000 lines of C++ code. It is built on top of FAWN-KV [5, 18] (~8500 LOC), which provides linearizable key-value storage within a local cluster. COPS uses Apache Thrift [7] for communication between all system components and Google's Snappy [45] for compressing dependency lists. Our current prototype implements all features described in the paper, excluding chain replication for local fault tolerance[10] and conflict detection.

We compare three systems: LOG, COPS, and COPS-GT. LOG uses the COPS code-base but excludes all dependency tracking, making it simulate previous work that uses log exchange to establish causal consistency (e.g., Bayou [41] and PRACTI [10]). Table 1 summarizes these three systems.

Each experiment is run on one cluster from the VICCI testbed [52]. The cluster's 70 servers give users an isolated Linux VServer. Each server has 2x6 core Intel Xeon X5650 CPUs, 48GB RAM, 3x1TB Hard Drives, and 2x1GigE network ports.

For each experiment, we partition the cluster into multiple logical "datacenters" as necessary. We retain full bandwidth between the nodes in different datacenters to reflect the high-bandwidth backbone that often exists between them. All reads and writes in FAWN-KV go to disk, but most operations in our experiments hit the kernel buffer cache.

The results presented are from 60-second trials. Data from the first and last 15s of each trial were elided to avoid experimental artifacts, as well as to allow garbage collection and replication mechanisms to ramp up. We run each trial 15 times and report the median; the minimum and maximum results are almost always within 6% of

[9] It remains an interesting aspect of future work to support flexibility in the number of datacenters required for committing within the causal+ model.

[10] Chain replication was not fully functional in the version of FAWN-KV on which our implementation is built.

System	Operation	Latency (ms)			Throughput (Kops/s)
		50%	99%	99.9%	
Thrift	ping	0.26	3.62	12.25	60
COPS	get_by_version	0.37	3.08	11.29	52
COPS-GT	get_by_version	0.38	3.14	9.52	52
COPS	put_after (1)	0.57	6.91	11.37	30
COPS-GT	put_after (1)	0.91	5.37	7.37	24
COPS-GT	put_after (130)	1.03	7.45	11.54	20

Table 2: Latency (in ms) and throughput (in Kops/s) of various operations for 1B objects in saturated systems. put_after(x) includes metadata for x dependencies.

the median, and we attribute the few trials with larger throughput differences to the shared nature of the VICCI platform.

6.2 Microbenchmarks

We first evaluate the performance characteristics for COPS and COPS-GT in a simple setting: two datacenters, one server per datacenter, and one colocated client machine. The client sends put and get requests to its local server, attempting to saturate the system. The requests are spread over 2^{18} keys and have 1B values—we use 1B values for consistency with later experiments, where small values are the worst case for COPS (see Figure 11(c)). Table 2 shows the median, 99%, and 99.9% latencies and throughput.

The design decision in COPS to handle client operations locally yields low latency for all operations. The latencies for `get_by_version` operations in COPS and COPS-GT are similar to an end-to-end RPC ping using Thrift. The latencies for `put_after` operations are slightly higher because they are more computationally expensive; they need to update metadata and write values. Nevertheless, the median latency for `put_after` operations, even those with up to 130 dependencies, is around 1 ms.

System throughput follows a similar pattern. `get_by_version` operations achieve high throughput, similar to that of Thrift ping operations (52 vs. 60 Kops/s). A COPS server can process `put_after` operations at 30 Kops/s (such operations are more computationally expensive than gets), while COPS-GT achieves 20% lower throughput when `put_after` operations have 1 dependency (due to the cost of garbage collecting old versions). As the number of dependencies in COP-GT `put_after` operations increases, throughput drops slightly due to the greater size of metadata in each operation (each dependency is ~12B).

6.3 Dynamic Workloads

We model a dynamic workload with interacting clients accessing the COPS system as follows. We set up two datacenters of S servers each and colocate S client machines in one of the two datacenters. The clients access storage servers in the local datacenter, which replicates `put_after` operations to the remote datacenter. We report the *sustainable* throughput in our experiments, which is the maximum throughput that both datacenters can handle. In most cases, COPS becomes CPU-bound at the local datacenter, and that COPS-GT becomes CPU-bound at the remote one.

To better stress the system and more accurately depict real operation, each client machine emulates multiple logical COPS clients. Each time a client performs an operation, it randomly executes a put or get operation, according to a specified put:get ratio. All operations in a given experiment use fixed-size values.

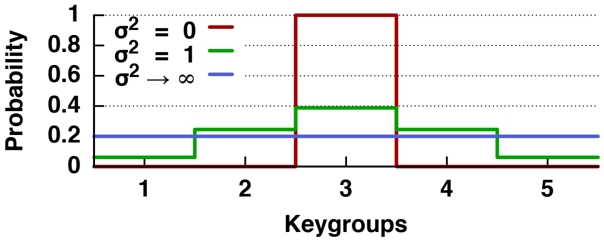


Figure 8: In our experiments, clients choose keys to access by first selecting a keygroup according to some normal distribution, then randomly selecting a key within that group according to a uniform distribution. Figure shows such a stepped normal distribution for differing variances for client #3 (of 5).

The key for each operation is selected to control the amount of dependence between operations (i.e., from fully isolated to fully intermixed). Specifically, given N clients, the full keyspace consists of N keygroups, $R_1 \ldots R_N$, one per client. Each keygroup contains K keys, which are randomly distributed (i.e., they do not all reside on the same server). When clients issue operations, they select keys as follows. First, they pick a keygroup by sampling from a normal distribution defined over the N keygroups, where each keygroup has width 1. Then, they select a key within that keygroup uniformly at random. The result is a distribution over keys with equal likelihood for keys within the same keygroup, and possibly varying likelihood across groups.

Figure 8 illustrates an example, showing the keygroup distribution for client #3 (of 5 total) for variances of 0, 1, and the limit approaching ∞. When the variance is 0, a client will restrict its accesses to its "own" keygroup and never interact with other clients. In contrast, when the variance → ∞, client accesses are distributed uniformly at random over all keys, leading to maximal inter-dependencies between `put_after` operations.

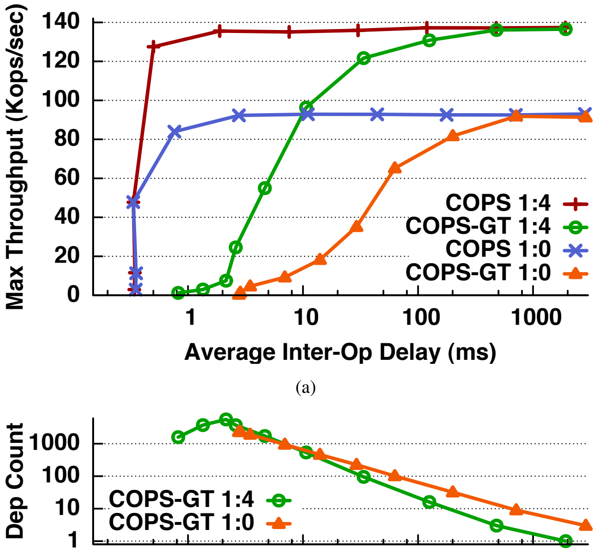
The parameters of the dynamic workload experiments are the following, unless otherwise specified:

Parameter	Default	Parameter	Default
datacenters	2	put:get ratio	1:1 or 1:4
servers / datacenter	4	variance	1
clients / server	1024	value size	1B
keys / keygroup	512		

As the state space of all possible combinations of these variables is large, the following experiments explore parameters individually.

Clients Per Server. We first characterize the system throughput as a function of increasing delay between client operations (for two different put:get ratios).[11] Figure 9(a) shows that when the inter-operation delay is low, COPS significantly outperforms COPS-GT. Conversely, when the inter-operation delay approaches several hundred milliseconds, the maximum throughputs of COPS and COPS-GT converge. Figure 9(b) helps explain this behavior: As the inter-operation delay increases, the number of dependencies per operation *decreases* because of the ongoing garbage collection.

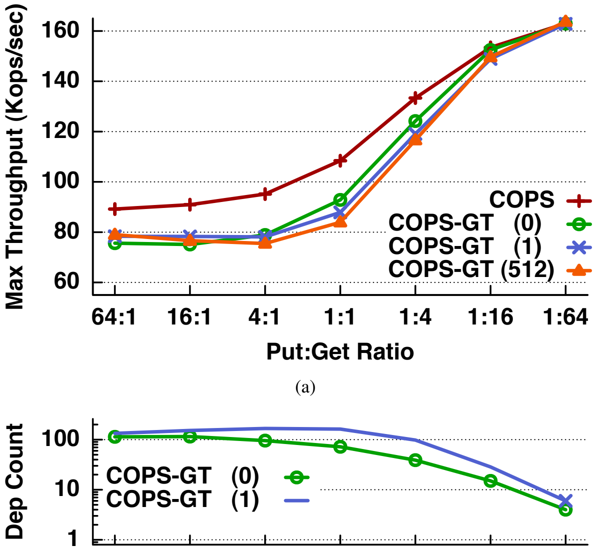
[11] For these experiments, we do not directly control the inter-operation delay. Rather, we increase the number of logical clients running on each of the client machines from 1 to 2^{18}; given a fixed-size thread pool for clients in our test framework, each logical client gets scheduled more infrequently. As each client makes one request before yielding, this leads to higher average inter-op delay (calculated simply as $\frac{\text{throughput}}{\text{\# of clients}}$). Our default setting of 1024 clients/server yields an average inter-op delay of 29 ms for COPS-GT with a 1:0 put:get ratio, 11ms for COPS with 1:0, 11ms for COPS-GT with 1:4, and 8ms for COPS with 1:4.



(a)

(b)

Figure 9: Maximum throughput and the resulting average dependency size of COPS and COPS-GT for a given inter-put delay between consecutive operations by the same logical client. The legend gives the put:get ratio (i.e., 1:0 or 1:4).



(a)

(b)

Figure 10: Maximum throughput and the resulting average dependency size of COPS and COPS-GT for a given put:get ratio. The legend gives the variance (i.e., 0, 1, or 512).

To understand this relationship, consider the following example. If the global-checkpoint-time is 6 seconds behind the current time and a logical client is performing 100 puts/sec (in an all-put workload), each put will have $100 \cdot 6 = 600$ dependencies. Figure 9(b) illustrates this relationship. While COPS will store only the single nearest dependency (not shown), COPS-GT must track all dependencies that have not been garbage collected. These additional dependencies explain the performance of COPS-GT: When the inter-put time is small, there are a large number of dependencies that need to be propagated with each value, and thus each operation is more expensive.

The global-checkpoint-time typically lags ~6 seconds behind the current time because it includes both the *trans_time* delay (per Section 5.1) and the time needed to gossip checkpoints around their local datacenter (nodes gossip once every 100ms). Recall that an agreed-upon *trans_time* delay is needed to ensure that currently executing `get_trans` operations can complete, while storage nodes use gossiping to determine the oldest uncommitted operation (and thus the latest timestamp for which dependencies can be garbage collected). Notably, round-trip-time latency *between* datacenters is only a small component of the lag, and thus performance is not significantly affected by RTT (e.g., a 70ms wide-area RTT is about 1% of a 6s lag for the global-checkpoint-time).

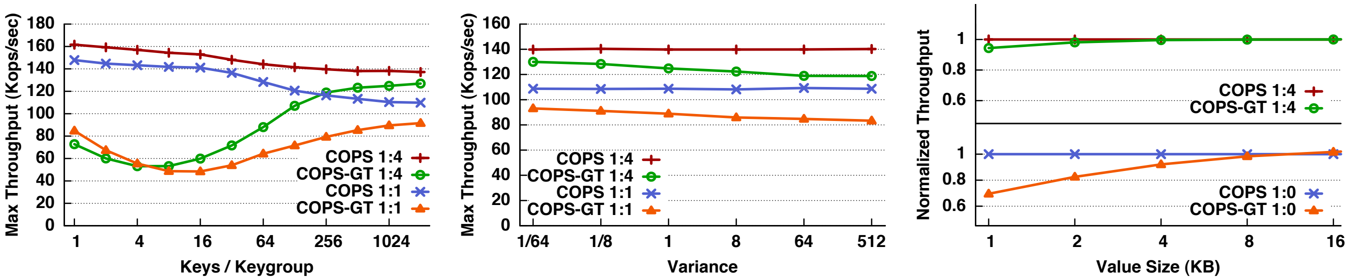
Put:Get Ratio. We next evaluate system performance under varying put:get ratios and key-access distributions. Figure 10(a) shows the throughput of COPS and COPS-GT for put:get ratios from 64:1 to 1:64 and three different distribution variances. We observe that throughput increases for read-heavier workloads (put:get ratios < 1), and that COPS-GT becomes competitive with COPS for read-mostly workloads. While the performance of COPS is identical under different variances, the throughput of COPS-GT *is* affected by variance. We explain both behaviors by characterizing the relationship between put:get ratio and the number of dependencies (Figure 10(b)); fewer dependencies translates to less metadata that needs to be propagated and thus higher throughput.

When different clients access the same keys (variance > 0), we observe two distinct phases in Figure 10(b). First, as the put:get ratio decreases from 64:1 to 1:1, the number of dependencies *increases*. This increase occurs because each get operation increases the likelihood a client will inherit new dependencies by getting a value that has been recently put by another client. For instance, if client_1 puts a value v_1 with dependencies d and client_2 reads that value, then client_2's future put will have dependencies on both v_1 and d. Second, as the put:get ratio then decreases from 1:1 to 1:64, the number of dependencies *decreases* for two reasons: (i) each client is executing fewer put operations and thus each value depends on fewer values previously written by this client; and (ii) because there are fewer put operations, more of the keys have a value that is older than the global-checkpoint-time, and thus getting them introduces no additional dependencies.

When clients access independent keys (variance = 0), the number of dependencies is strictly decreasing with the put:get ratio. This result is expected because each client accesses only values in its own keygroup that it previously wrote and already has a dependency on. Thus, no get causes a client to inherit new dependencies.

The average dependency count for COPS (not shown) is always low, from 1 to 4 dependencies, because COPS needs to track only the nearest (instead of all) dependencies.

Keys Per Keygroup. Figure 11(a) shows the effect of keygroup size on the throughput of COPS and COPS-GT. Recall that clients distribute their requests uniformly over keys in their selected keygroup. The behavior of COPS-GT is nuanced; we explain its varying throughput by considering the likelihood that a get operation will inherit new dependencies, which in turn reduces throughput. With the default variance of 1 and a low number of keys/keygroup, most



(a) (b) (c)

Figure 11: Maximum system throughput (using put:get ratios of 1:4, 1:1, or 1:0) for varied keys/keygroup, variances, and value sizes.

clients access only a small number of keys. Once a value is retrieved and its dependencies inherited, subsequent gets on that same value do not cause a client to inherit any new dependencies. As the number of keys/keygroup begins to increase, however, clients are less likely to get the same value repeatedly, and they begin inheriting additional dependencies. As this number continues to rise, however, garbage collection can begin to have an effect: Fewer gets retrieve a value that was recently written by another client (e.g., after the global checkpoint time), and thus fewer gets return new dependencies. The bowed shape of COPS-GT's performance is likely due to these two contrasting effects.

Variance. Figure 11(b) examines the effect of variance on system performance. As noted earlier, the throughput of COPS is unaffected by different variances: Get operations in COPS never inherit extra dependencies, as the returned value is always "nearer," by definition. COPS-GT has an increased chance of inheriting dependencies as variance increases, however, which results in decreased throughput.

Value Size. Finally, Figure 11(c) shows the effect of value size on system performance. In this experiment, we normalize the systems' maximum throughput against that of COPS (the COPS line at exactly 1.0 is shown only for comparison). As the size of values increases, the relative throughput of COPS-GT approaches that of COPS.

We attribute this to two reasons. First, the relative cost of processing dependencies (which are of fixed size) decreases compared to that of processing the actual values. Second, as processing time per operation increases, the inter-operation delay correspondingly increases, which in turn leads to fewer dependencies.

6.4 Scalability

To evaluate the scalability of COPS and COPS-GT, we compare them to LOG. LOG mimics systems based on log serialization and exchange, which can only provide causal+ consistency with single node replicas. Our implementation of LOG uses the COPS code, but excludes dependency tracking.

Figure 12 shows the throughput of COPS and COPS-GT (running on 1, 2, 4, 8, or 16 servers/datacenter) normalized against LOG (running on 1 server/datacenter). Unless specified otherwise, all experiments use the default settings given in Section 6.3, including a put:get ratio of 1:1. In all experiments, COPS running on a single server/datacenter achieves performance almost identical to LOG. (After all, compared to LOG, COPS needs to track only a small number of dependencies, typically ≤ 4, and any dep_check operations in the remote datacenter can be executed as local function calls.) More importantly, we see that COPS and COPS-GT scale well in all scenarios: as we double the number of servers per datacenter, throughput almost doubles.

In the experiment with all default settings, COPS and COPS-GT scale well relative to themselves, although COPS-GT's throughput is only about two-thirds that of COPS. These results demonstrate that the default parameters were chosen to provide a non-ideal workload for the system. However, under a number of different conditions—and, indeed, a workload more common to Internet services—the performance of COPS and COPS-GT is almost identical.

As one example, the relative throughput of COPS-GT is close to that of COPS when the inter-operation delay is high (achieved by hosting 32K clients per server, as opposed to the default 1024 clients; see Footnote 11). Similarly, a more read-heavy workload (put:get ratio of 1:16 vs. 1:1), a smaller variance in clients' access distributions (1/128 vs. 1), or larger-sized values (16KB vs. 1B)—controlling for all other parameters—all have the similar effect: the throughput of COPS-GT becomes comparable to that of COPS.

Finally, for the "expected workload" experiment, we set the parameters closer to what we might encounter in an Internet service such as social networking. Compared to the default, this workload has a higher inter-operation delay (32K clients/server), larger values (1KB), and a read-heavy distribution (1:16 ratio). Under these settings, the throughput of COPS and COPS-GT are very comparable, and both scale well with the number of servers.

7. RELATED WORK

We divide related work into four categories: ALPS systems, causally consistent systems, linearizable systems, and transactional systems.

ALPS Systems. The increasingly crowded category of ALPS systems includes eventually consistent key-value stores such as Amazon's Dynamo [16], LinkedIn's Project Voldemort [43], and the popular memcached [19]. Facebook's Cassandra [30] can be configured to use eventual consistency to achieve ALPS properties, or can sacrifice ALPS properties to provide linearizability. A key influence for our work was Yahoo!'s PNUTS [15], which provides per-key sequential consistency (although they name this per-record timeline consistency). PNUTS does not provide any consistency between keys, however; achieving such consistency introduces the scalability challenges that COPS addresses.

Causally Consistent Systems. Many previous system designers have recognized the utility of causal consistency. Bayou [41] provides a SQL-like interface to single-machine replicas that achieves causal+ consistency. Bayou handles all reads and writes locally; it does not address the scalability goals we consider.

TACT [53] is a causal+ consistent system that uses order and numeric bounding to limit the divergence of replicas in the system. The ISIS [12] system exploits the concept of virtual synchrony [11] to provide applications with a causal broadcast primitive (CBcast).

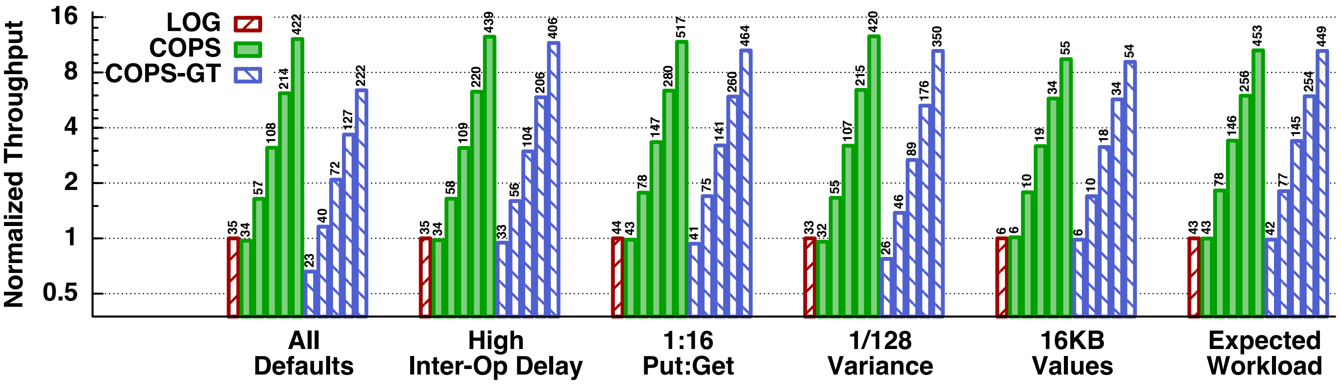


Figure 12: Throughput for LOG with 1 server/datacenter, and COPS and COPS-GT with 1, 2, 4, 8, and 16 servers/datacenter, for a variety of scenarios. Throughput is normalized against LOG for each scenario; raw throughput (in Kops/s) is given above each bar.

CBcast could be used in a straightforward manner to provide a causally consistent key-value store. Replicas that share information via causal memory [2] can also provide a causally consistent ALP key-value store. These systems, however, all require single-machine replicas and thus do not provide scalability.

PRACTI [10] is a causal+ consistent ALP system that supports partial replication, which allows a replica to store only a subset of keys and thus provides some scalability. However, each replica—and thus the set of keys over which causal+ consistency is provided—is still limited to what a single machine can handle.

Lazy replication [29] is closest to COPS's approach. Lazy replication explicitly marks updates with their causal dependencies and waits for those dependencies to be satisfied before applying them at a replica. These dependencies are maintained and attached to updates via a front-end that is an analog to our client library. The design of lazy replication, however, assumes that replicas are limited to a single machine: Each replica requires a single point that can (i) create a sequential log of all replica operations, (ii) gossip that log to other replicas, (iii) merge the log of its operations with those of other replicas, and finally (iv) apply these operations in causal order.

Finally, in concurrent theoretical work, Mahajan et al. [35] define real-time causal (RTC) consistency and prove that it is the strongest achievable in an always-available system. RTC is stronger than causal+ because it enforces a real-time requirement: if causally-concurrent writes do not overlap in real-time, the earlier write may not be ordered after the later write. This real-time requirement helps capture potential causality that is hidden from the system (e.g., out-of-band messaging [14]). In contrast, causal+ does not have a real-time requirement, which allows for more efficient implementations. Notably, COPS's efficient last-writer-wins rule results in a causal+ but not RTC consistent system, while a "return-them-all" conflict handler would provide both properties.

Linearizable Systems. Linearizability can be provided using a single commit point (as in primary-copy systems [4, 39], which may *eagerly* replicate data through two-phase commit protocols [44]) or using distributed agreement (e.g., Paxos [33]). Rather than replicate content everywhere, quorum systems ensure that read and write sets overlap for linearizability [22, 25].

As noted earlier, CAP states that linearizable systems cannot have latency lower than their round-trip inter-datacenter latency; only recently have they been used for wide-area operation, and only when the low latency of ALPS can be sacrificed [9]. CRAQ [48] can complete reads in the local-area when there is little write contention, but otherwise requires wide-area operations to ensure linearizability.

Transactions. Unlike most filesystems or key-value stores, the database community has long considered consistency across multiple keys through the use of read *and* write transactions. In many commercial database systems, a single master executes transactions across keys, then *lazily* sends its transaction log to other replicas, potentially over the wide-area. Typically, these asynchronous replicas are read-only, unlike COPS's write-anywhere replicas. Today's large-scale databases typically partition (or shard) data over multiple DB instances [17, 38, 42], much like in consistent hashing. Transactions are applied only within a single partition, whereas COPS can establish causal dependencies across nodes/partitions.

Several database systems support transactions across partitions and/or datacenters (both of which have been viewed in the database literature as independent *sites*). For example, the R* database [37] uses a tree of processes and two-phase locking for multi-site transactions. This two-phase locking, however, prevents the system from guaranteeing availability, low latency, or partition tolerance. Sinfonia [1] provides "mini" transactions to distributed shared memory via a lightweight two-phase commit protocol, but only considers operations within a single datacenter. Finally, Walter [47], a recent key-value store for the wide-area, provides transactional consistency across keys (including for writes, unlike COPS), and includes optimizations that allow transactions to execute within a single site, under certain scenarios. But while COPS focuses on availability and low-latency, Walter stresses transactional guarantees: ensuring causal relationships between keys can require a two-phase commit across the wide-area. Furthermore, in COPS, scalable datacenters are a first-order design goal, while Walter's sites currently consist of single machines (as a single serialization point for transactions).

8. CONCLUSION

Today's high-scale, wide-area systems provide "always on," low-latency operations for clients, at the cost of weak consistency guarantees and complex application logic. This paper presents COPS, a scalable distributed storage system that provides causal+ consistency without sacrificing ALPS properties. COPS achieves causal+ consistency by tracking and explicitly checking that causal dependencies are satisfied before exposing writes in each cluster. COPS-GT builds upon COPS by introducing get transactions that enable clients to obtain a consistent view of multiple keys; COPS-GT incorporates optimizations to curtail state, minimize multi-round protocols, and reduce replication overhead. Our evaluation demonstrates that COPS and COPS-GT provide low latency, high throughput, and scalability.

Acknowledgments. We owe a particular debt both to the SOSP program committee and to our shepherd, Mike Dahlin, for their extensive comments and Mike's thoughtful interaction that substantially improved both the presentation of and, indeed, our own view of, this work. Jeff Terrace, Erik Nordström, and David Shue provided useful comments on this work; Vijay Vasudevan offered helpful assistance with FAWN-KV; and Sapan Bhatia and Andy Bavier helped us run experiments on the VICCI testbed. This work was supported by NSF funding (CAREER CSR-0953197 and CCF-0964474), VICCI (NSF Award MRI-1040123), a gift from Google, and the Intel Science and Technology Center for Cloud Computing.

REFERENCES

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM TOCS*, 27(3), 2009.

[2] M. Ahamad, G. Neiger, P. Kohli, J. Burns, and P. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1), 1995.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, Aug. 2008.

[4] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Conf. Software Engineering*, Oct. 1976.

[5] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP*, Oct. 2009.

[6] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM TOCS*, 14(1), 1996.

[7] Apache Thrift. `http://thrift.apache.org/`, 2011.

[8] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM TOCS*, 12(2), 1994.

[9] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, Jan. 2011.

[10] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI*, May 2006.

[11] K. P. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP*, Nov. 1987.

[12] K. P. Birman and R. V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Comp. Soc. Press, 1994.

[13] E. Brewer. Towards robust distributed systems. PODC Keynote, July 2000.

[14] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *SOSP*, Dec. 1993.

[15] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, Aug. 2008.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, Oct. 2007.

[17] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *Knowledge and Data Engineering*, 2(1), 1990.

[18] FAWN-KV. `https://github.com/vrv/FAWN-KV`, 2011.

[19] B. Fitzpatrick. Memcached: a distributed memory object caching system. `http://memcached.org/`, 2011.

[20] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *SOSP*, Oct. 1997.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, Oct. 2003.

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

[23] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), 2002.

[24] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, Aug. 2009.

[25] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM TOCS*, 4(1), Feb. 1986.

[26] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[27] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, May 1997.

[28] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM TOCS*, 10(3), Feb. 1992.

[29] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM TOCS*, 10(4), 1992.

[30] A. Lakshman and P. Malik. Cassandra – a decentralized structured storage system. In *LADIS*, Oct. 2009.

[31] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7), 1978.

[32] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computer*, 28(9), 1979.

[33] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2), 1998.

[34] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report TR-180-88, Princeton Univ., Dept. Comp. Sci., 1988.

[35] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. Technical Report TR-11-22, Univ. Texas at Austin, Dept. Comp. Sci., 2011.

[36] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM TOPLAS*, 8(1), Jan. 1986.

[37] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Trans. Database Sys.*, 11(4), 1986.

[38] MySQL. `http://www.mysql.com/`, 2011.

[39] B. M. Oki and B. H. Liskov. Viewstamped replication: A general primary copy. In *PODC*, Aug. 1988.

[40] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Software Eng.*, 9(3), 1983.

[41] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, Oct. 1997.

[42] PostgresSQL. `http://www.postgresql.org/`, 2011.

[43] Project Voldemort. `http://project-voldemort.com/`, 2011.

[44] D. Skeen. A formal model of crash recovery in a distributed system. *IEEE Trans. Software Engineering*, 9(3), May 1983.

[45] Snappy. `http://code.google.com/p/snappy/`, 2011.

[46] J. Sobel. Scaling out. Engineering at Facebook blog, Aug. 20 2008.

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

[48] J. Terrace and M. J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *USENIX ATC*, June 2009.

[49] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *Conf. Parallel Distributed Info. Sys.*, Sept. 1994.

[50] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Sys.*, 4(2), 1979.

[51] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, Dec. 2004.

[52] VICCI. `http://vicci.org/`, 2011.

[53] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *OSDI*, Oct. 2000.

A. FORMAL DEFINITION OF CAUSAL+

We first present causal consistency with convergent conflict handling (causal+ consistency) for a system with only get and put operations (reads and writes), and we then introduce get transactions. We use a model closely derived from Ahamad et al. [2], which in turn was derived from those used by Herlihy and Wing [26] and Misra [36].

Original Model of Causal Consistency [2] with terminology modified to match this paper's definitions:

A system is a finite set of threads of execution, also called threads, that interact via a key-value store that consists of a finite set of keys. Let $T = \{t_1, t_2, \ldots, t_n\}$ be the set of threads. The local history L_i of a thread i is a sequence of get and put operations. If operation σ_1 precedes σ_2 in L_i, we write $\sigma_1 \underset{i}{\rightarrow} \sigma_2$. A history $H = \langle L_1, L_2, \ldots, L_n \rangle$ is the collection of local histories for all threads of execution. A serialization S of H is a linear sequence of all operations in H in which each get on a key returns its most recent preceding put (or $\perp$ if there does not exist any preceding put). The serialization S respects an order $\rightarrow$ if, for any operation σ_1 and σ_2 in S, $\sigma_1 \rightarrow \sigma_2$ implies σ_1 precedes σ_2 in S.

The *puts-into* order associates a put operation, `put`(k,v), with each get operation, `get`(k)=v. Because there may be multiple puts of a value to a key, there may be more than one puts-into order.[12] A puts-into order $\mapsto$ on H is any relation with the following properties:

- If $\sigma_1 \mapsto \sigma_2$, then there is a key k and value v such that operation σ_1 := `put`(k,v) and σ_2 := `get`(k)=v.
- For any operation σ_2, there exists at most one σ_1 for which $\sigma_1 \mapsto \sigma_2$.
- If σ_2 := `get`(k)=v for some k,v and there exists no σ_1 such that $\sigma_1 \mapsto \sigma_2$, then v = $\perp$. That is, a get with no preceding put must retrieve the initial value.

Two operations, σ_1 and σ_2, are related by a causal order $\leadsto$ if and only if one of the following holds:

- $\sigma_1 \underset{i}{\rightarrow} \sigma_2$ for some t_i (σ_1 precedes σ_2 in L_i);
- $\sigma_1 \mapsto \sigma_2$ (σ_2 gets the value put by σ_1); or
- There is some other operation σ' such that $\sigma_1 \leadsto \sigma' \leadsto \sigma_2$.

Incorporating Convergent Conflict Handling. Two operations on the same key, σ_1:=`put`(k,v$_1$) and σ_2:=`put`(k,v$_2$), are in *conflict* if they are not related by causality: $\sigma_1 \not\leadsto \sigma_2$ and $\sigma_2 \not\leadsto \sigma_1$.

A *convergent conflict handling function* is an associative, commutative function that operates on a set of conflicting operations on a key to eventually produce one (possibly new) final value for that key. The function must produce the same final value independent of the order in which it observes the conflicting updates. In this way, once every replica has observed the conflicting updates for a key, they will all independently agree on the same final value.

We model convergent conflict handling as a set of *handler* threads that are distinct from normal client threads. The handlers operate on a pair of conflicting values (v_1, v_2) to produce a new value *newval* = $h(v_1, v_2)$. By commutativity, $h(v_1, v_2) = h(v_2, v_1)$. To produce the new value, the handler thread had to read both v_1 and v_2 before putting the new value, and so *newval* is causally ordered after both original values: $v_1 \leadsto$ *newval* and $v_2 \leadsto$ *newval*.

With more than two conflicting updates, there will be multiple invocations of handler threads. For three values, there are several possible orders for resolving the conflicting updates in pairs:

$$\begin{tikzcd}
\sigma_1 \arrow[r] \arrow[dr] & \sigma_{1,2} \arrow[dr] & \\
\sigma_2 \arrow[ur] \arrow[dr] & \sigma_{1,3} \arrow[r] & \sigma_{1,2,3} \\
\sigma_3 \arrow[r] \arrow[ur] & \sigma_{2,3} \arrow[ur] &
\end{tikzcd}$$

The commutativity and associativity of the handler function ensures that regardless of the order, the final output will be identical. Further, it will be causally ordered after all of the original conflicting writes, as well as any intermediate values generated by the application of the handler function. If the handler observes multiple pairs of conflicting updates that produce the same output value (e.g., the final output in the figure above), it must output only one value, not multiple instances of the same value.

To prevent a client from seeing and having to reason about multiple, conflicting values, we restrict the put set for each **client** thread to be conflict free, denoted p_{cf_i}. A put set is *conflict free* if $\forall \sigma_j, \sigma_k \in p_{cf_i}$, σ_j and σ_k are not in conflict; that is, either they are puts to different keys or causally-related puts to the same key. For example, in the three conflicting put example, p_{cf_i} might include σ_1, $\sigma_{1,2}$, and $\sigma_{1,2,3}$, but not σ_2, σ_3, $\sigma_{1,3}$, or $\sigma_{2,3}$. The conflict-free property applies to client threads and not handler threads purposefully. Handler threads must be able to get values from conflicting puts so they may reason about and resolve them; client threads should not see conflicts so they do not have to reason about them.

Adding handler threads models the new functionality that convergent conflict handling provides. Restricting the put set strengthens consistency from causal to causal+. There are causal executions that are not causal+: for example, if σ_1 and σ_2 conflict, a client may get the value put by σ_1 and then the value put by σ_2 in a causal, but not causal+, system. On the other hand, there are no causal+ executions that are not causal, because causal+ only introduces an additional restriction (a smaller put set) to causal consistency.

If H is a history and t_i is a thread, let $A^H_{i+p_{cf_i}}$ comprise all operations in the local history of t_i, and a conflict-free set of puts in H, p_{cf_i}. A history H is *causally consistent with convergent conflict handling* (causal+) if it has a causal order $\leadsto$, such that

Causal+: For each *client* thread of execution t_i, there is a serialization S_i of $A^H_{i+p_{cf_i}}$ that respects $\leadsto$.

A data store is *causal+ consistent* if it admits only causal+ histories.

Introducing Get Transactions. To add get transactions to the model, we redefine the puts-into order so that it associates N put operations, `put`(k,v), with each get transaction of N values, `get_trans`([$k_1, \ldots, k_N$])=[$v_1, \ldots, v_N$]. Now, a puts-into order $\mapsto$ on H is any relation with the following properties:

- If $\sigma_1 \mapsto \sigma_2$, then there is a k and v such that σ_1 := `put`(k,v) and σ_2 := `get_trans`([...,k,...])=[...,v,...]. That is, for each component of a get transaction, there exists a preceding put.
- For each component of a get transaction σ_2, there exists at most one σ_1 for which $\sigma_1 \mapsto \sigma_2$.
- If σ_2 := `get_trans`([...,k,...])=[...,v,...] for some k,v and there exists no σ_1 such that $\sigma_1 \mapsto \sigma_2$, then v=$\perp$. That is, a get with no preceding put must retrieve the initial value.

[12] The COPS system uniquely identifies values with version numbers so there is only one puts-into order, but this is not necessarily true for causal+ consistency in general.