Project no. 609551
Project acronym: SyncFree
Project title: *Large-scale computation without synchronisation*

# European Seventh Framework Programme

# ICT call 10

| | |
|---|---|
| Deliverable reference number and title: | D.3.1<br>Guarantees in the presence of CRDT composition and transactions |
| Due date of deliverable: | October 1, 2014 |
| Actual submission date: | November 17, 2014 |
| Start date of project: | October 1, 2013 |
| Duration: | 36 months |
| Name and organisation of lead editor for this deliverable: | FCT, Universidade Nova de Lisboa |
| Revision: | 0.1 |
| Dissemination level: | CO |

# Contents

# 1 Executive summary

The SyncFree project aims to enable trustworthy large-scale distributed applications in geo-replicated settings. The core concept are replicated yet consistent data types (CRDTs) which allow information dissemination and sharing without the need for global synchronization.

Within the project, Work Package 3 (WP3) coordinates the work on extending the safety, quality and security guarantees provided by a system that uses minimal synchronisation. To this end, it intends to study the guarantees that can be provided by CRDTs: what are the guarantees that basic CRDTs provide, what are their inherent power and limitations, how can the guarantees be extended without synchronisation, and to identify at what point extra guarantees do require synchronisation.

This goal of the first task, *CRDT object composition and transactions*, was to research issues that arise in the development of applications with complex data structures, namely: how to compose complex data structures from simpler ones; how to access and modify multiple replicated objects with transactional properties; how to guarantee invariant preservations while minimizing coordination.

The specific requirements addressed in our work were driven by both the use cases studied in WP1 and previous experience of the project partners, both industrial and academic, in the development and deployment of large scale systems. We now briefly overview the results achieved during the reporting period.

**Composition** Applications manipulate complex data structures created by composing simpler ones. For applications that use CRDTs, there is also the need to be able to create complex objects from simpler ones, guaranteeing the the composed object maintains the convergence properties of CRDTs. We have produced the following contributions for addressing this challenge.

First, we have developed support for composing CRDTs using Maps [18] – in this approach, the values stored in a Map can be Sets, Maps, Registers, Flags and Counters CRDTs. This allows to create complex data structures for storing rich data from application. while guaranteeing convergence in the presence of concurrent updates. Support for the Map CRDT has been integrated in version 2.0 of Riak database[1] and it will also be included in Antidote, the platform prototype being developed in the context of WP2.

This work is currently being extended to allow storing each object in a different key [35], thus addressing the scalability issues in the initial approach. In a complimentary work, we have proposed techniques for supporting partial replication of large CRDT objects, while preserving convergence properties [17].

Finally, we have studied the problem of general composition. We proposed techniques for creating complex CRDT by composing elementary monotonic elements using a set of pre-defined composition rules that guarantee that the composed object is still monotonic [11]. This work is being extended with techniques for defining the semantics of composed objects.

[1] `http://riak.basho.com`

**Transactions and Replication** In many cases, applications need to access and modify data that does not fit naturally in a large composed object. In such cases, there is a need to group operations in atomic units or transactions that need to be replicated. In this context, we have made several contributions.

We have consolidated previous work on transactional models for weakly consistent settings, including the algorithms for executing mergeable transactions, a form of highly available transaction [6] that provides access to a causally consistent database snapshot and allow concurrent updates that are merged relying on CRDTs [51, 38, 52]. This work has served as the starting point for the development of the algorithms for supporting transactions in Antidote, the platform being developed in the context of WP2.

For supporting efficient replication protocols, we have proposed two general techniques in the context of this work. First, we have introduced delta-mutators as a technique for propagating a set of updates efficiently [2]. Second, we have proposed dotted version vectors sets as a causality tracking technique that keeps metadata small with complexity $O(\#replicas)$ [1]. Basho has adapted these proposals for integration in Riak, with the latter already integrated in version 2.0 of Riak while the former is currently being integrated for a future version. The ideas of dotted version vectors have also been used in Antidote.

**Invariants** Although a large number of application can work correctly under weak consistency models, other applications need to maintain strong invariants that cannot be enforced using such models.

To address this challenge, we have proposed techniques for enforcing invariants while minimizing the required coordination. A first approach addressed numeric invariants only by relying on a new CRDT, the Bounded Counter CRDT, and a middleware that runs on top of Riak [7]. The second approach addresses generic invariants and includes a semi-automatic methodology for deriving a reservation system from the specification of invariants and operations [9, 8]. The reservation system enforces invariants on top of a weakly consistent data store while requiring minimal synchronisation that can usually be executed outside of the critical path of operation execution. This work has been developed in context of both WP3 and WP4, and we expect to integrate the produced results (or part of them) with the work being developed in WP2 during the next year, namely by integrating support for enforcing invariants in Antidote.

# 2 Milestones in the Deliverable

WP3, task 1 has the following milestone, shared with other work packages:

| Mil. no | Mil. name | Date due | Actual date | Lead contractor |
|---|---|---|---|---|
| MS1 | CRDT consolidation in a static environment | M12 | M12 | INRIA |

Task 3.1 has contributed to this milestone (and produced research to be included in following milestones) by focusing on the following goals, as stated in the project proposal:

> This deliverable will report on the mechanisms for enforcing guarantees in the presence of CRDT composition and transactions.
>
> Applications create complex data structures by composing CRDTs under some invariant. (For instance, a graph is the composition of a set of vertices and a set of arcs, under the invariant that the end-points of every arc are in the set of vertices.) Another composition pattern is CRDT transactions to ensure cross-object invariants (e.g., referential integrity). This task studies composition mechanisms and their suitability for different classes of invariants. In particular, we will examine the issues of non-monotonic composition, not addressed in existing work. We will study which transaction properties are required to maintain different classes of invariants. Of particular interest is identifying at what point stronger guarantees require introducing small doses of synchronisation.

# 3 Contractors contributing to the Deliverable

The following contractors contributed to the deliverables

## 3.1 KL

Annette Bieniusa.

## 3.2 INRIA

Marek Zawirski, Mahsa Najafzadeh, Marc Shapiro, Ahmed-Nacer Mehdi, Pascal Urso.

## 3.3 Louvain

Iwan Briquemont, Manuel Bravo, Zhongmiao Li, Peter van Roy.

## 3.4 Nova

Valter Balegas, Sérgio Duarte, Ali Shoker, Carla Ferreira, Alcino Cunha, Paulo Sérgio Almeida, Rodrigo Rodrigues, Carlos Baquero, Nuno Preguiça.

## 3.5 Basho

Russell Brown, Sean Cribbs, Sam Elliot, Chris Meiklejohn.

# 4 Results

This section presents the results obtained in this task, during the reporting period. We organize the results in three groups: *composition*, discussing the approaches developed for creating complex objects by composing multiple CRDTs; *transactions*, describing the algorithms developed for grouping the execution of multiple operations; *invariants*, detailing how to enforce invariant while minimizing the required coordination.

## 4.1 Composition

Applications manipulate complex data that is modelled as a composition of basic data structures, such as sets, maps, lists, etc. CRDT designs proposed in literature [50, 41, 44, 43, 16, 47] provide versions of these basic data structures that can be replicated, allow concurrent updates to modify different replicas, and provide an automatic mechanism that combines these updates in a deterministic way. This simplifies the design of eventually consistent systems, ensuring the convergence of multiple replicas.

Some of the use-cases studied in WP1 and the experience in modelling complex applications by some of the partners have shown that supporting the composition of multiple CRDTs was an important requirement. To address this requirement, we have proposed a limited solution that allows composing multiple CRDTs using a Map CRDT [18]. We have also studied more general composition rules for creating CRDTs from more elementary CRDTs [11].

### 4.1.1 Map CRDT

The Map CRDT [18] allows associating a key to a CRDT, with the current implementation supporting Boolean, Register, Counter, Set and Map. This allows to build complex data models that have a tree structure. The main challenge resided in the definition and implementation of a sensible semantics for merging concurrent updates, in particular the case of a concurrent remove and update in the same sub-tree.

The current implementation is state-based and provides the following update operations: (i) *put(key,obj)*, that links a new CRDT *obj* with key *key* (more precisely, the key to the object is the pair $(key, objtype)$); (ii) *rem(key)*, that unlinks key *key* from the map. The *put* operation allows to create a tree of objects, with the Map CRDTs serving as internal nodes. The effect of *rem(key)* is equivalent to recursively resetting all objects in the tree of objects associated with *key* (or undo the effect of all update operations in such objects).

The *merge* of two replicas of a given map, combines the updates to the map and recursively performs merge on the linked objects. Updates on different keys are merged trivially. Updates on the same key are handled as follows: (i) two concurrent *put* operation linking some key to two different objects results in the merge of the state of the objects. Note that two concurrent *put* operations linking a key to objects of different types results in two different map entries, as the key of the map entry is the pair $(key, objtype)$. (ii) a *rem(key)* concurrent with some

```
Location ahmedMap = new Location(new Namespace("maps", "customers"), "ahmed_info");

MapUpdate purchaseUpdate = new MapUpdate()
        .update("first_purchase", new FlagUpdate().set(true))
        .update("widget_purchases", new CounterUpdate(1));
        .update("amount", new RegisterUpdate().set(BinaryValue.create("1271")))
        .update("items", new SetUpdate().add(BinaryValue.create("large_widget")));
MapUpdate annikaUpdate = new MapUpdate()
        .update("purchase", purchaseUpdate);
MapUpdate ahmedUpdate = new MapUpdate()
        .update("annika_info", annikaUpdate);
UpdateMap update = new UpdateMap.Builder(ahmedMap, ahmedUpdate)
        .withUpdate(ahmedUpdate)
        .build();
client.execute(update);
```

Figure 1: Access to Map CRDT in Java (adapted from [14]).

update in the object mapped by *key* is solved by keeping the tree of Map CRDTs strictly necessary to access the modified object.

For implementing the described merge approach efficiently, each operation and its effects are assigned a timestamp (or dot) [1]. The (outer) Map CRDT keeps a version vector summarizing the updates applied to all CRDTs in the tree. With this information it is possible to know what information has been added and deleted to the Map, using an approach similar to the one used in the optimized OR-Set [16] - when merging two states, the merge procedure detects that replica $r_1$ was updated concurrently with a remove in replica $r_2$ if the dot associated with the new information in $r_1$ is not reflected in the vector of $r_2$. Additionally, a bit of information has been deleted if the dot associated with it in $r_1$ is already reflected in the vector of $r_2$

Basho provides an open-source implementation of the Map CRDT, as well as other CRDTs [43], in Erlang [13]. This implementation is integrated in the Riak Database v. 2.0, released in September of 2014. It includes APIs to access the CRDTs provided by the database in several languages - Figure 1 exemplifies the access to the Map CRDT in Java.

### 4.1.2 General composition

Map CRDT provides a good practical design for storing complex application data. However, the possible compositions are limited, and although it can address a large number of application scenarios, it falls short in some cases. To address this limitation, we have investigated the problem of general composition of CRDTs. The work detailed in a (currently unpublished) technical report [11], is briefly described here.

State-based CRDTs are rooted in mathematical structures called join-semilattices (or simply lattices, in this context). These order structures ensure that the replicated states of the defined data types evolve and increase in a partial order in a sufficiently defined way, so as to ensure that all concurrent evolutions can be merged deterministically.

We started by identifying a small set of primitive lattices that are useful to construct more complex structures by composition, namely, (i) Singleton, that has

a single element; (ii) Boolean, with join the logical $\vee$; (iii) Naturals, with join being the max.

State-based CRDTs can be specified by selecting a given lattice to model the state, and choosing an initial value in the lattice. Operations can only change the state by *inflations* (intuitively, by moving upwards in the lattice) and do not return values. Query operations evaluate an arbitrary function on the state and return a value.

For creating lattices that model the state of CRDTs we consider a number of composition techniques that are known to derive lattices from other lattices or from simpler structure, including: (i) the product of two lattices; (ii) the composition according to a lexicographic order; (iii) the linear sum of two lattices.

We have shown that from the primitive lattices and the composition rules described, it is possible to compose the CRDT proposed in literature and more complex CRDTs.

### 4.1.3 Decomposing CRDTs (for storage)

Creating CRDTs that can be used for storing complex application data may simplify application development, but it can lead to performance problems as the system needs to handle these potentially large data objects. This problem occurs both in the servers, as a small update to a large object may lead to loading and storing large amounts of data from disk, and when transferring CRDT to clients, as large objects may be transferred when only a part of the data is necessary. This problem has been addressed in two complementary works in the context of the project.

**Decomposition in Riak** The Map CRDT implemented in Riak DT library provides the ability of composing CRDTs through embedding, in which the complete tree of CRDT object is stored in a single key. This leads to performance problems, in which a noticeable performance degradation can be observed when accessing object larger than one megabyte. The naive approach of storing objects in different keys cannot be used in Riak, as it does not guarantee causal consistency, which can lead to the observation of reference for objects that are still not available in a given replica.

To address these issues, we have been working on an alternative composition mechanism that does not degrade in performance as size increases, but provides values at read time which observe the lattice properties of state-based CRDTs ensuring conflict-free merges with later states. The key ideas that are being pursued in this on-going work, presented by Meiklejohn [35], are the following.

First, the API for interacting with Riak DT map needs to be extended to support specifying whether the object should be composed by embedding or by reference. When performing a write of an object containing references to other objects, the system generates unique reference identifiers for each referenced object.

Second, when performing a read operation of an object containing references to other objects, the system recursively attempts to retrieve the referenced objects from the data store. For this step to work properly, it is necessary to guarantee that a read operation observes a causally consistent database state. We are currently exploring different alternative to achieve this goal.

**Conflict-free Partially Replicated Data Structures** Replicating large CRDTs in the clients can be a waste of resources, of both storage and bandwidth. For example, in a Facebook-like application, the posts of a user wall can be stored in a set CRDT (or sequence CRDT). If the user is interested in only a small subset of these posts, according to some criterium, storing the complete state of the CRDT is not necessary.

We have proposed a new abstraction, the Conflict-free Partially Replicated Data Structures (CPRDTs) [17], to address this issue. A CPRDT is a CRDT that can be partitioned in multiple *particles*. We define particles as the smallest meaningful elements of a CPRDT. By meaningful we refer to the smallest element that can be used for query and update operations. For instance, a particle in a grow-only set would be any element that can be added or looked up in the set.

When defining a CPRDT, it is necessary to define the particles that compose the CPRDT and the operations that are defined. For each operation, the following functions must be specified:

**required** For an operation *op* with its arguments, $required(op)$ is the set of particles needed by op to be properly executed. This means that, for replica $x_i$, an operation is enabled only if $required(op) \subseteq shard(x_i)$. E.g. for the lookup operation of a set, $required(lookup(e)) = e$ where $e$ is an element of the set. In case $e \notin shard(x_i)$, the replica will not be able to know whether $e$ is in the set because it has not kept a state for it.

**affected** The function $affected(op)$ returns the set of particles that may have their state affected after executing operation *op*.

These functions provide the necessary information for the system to control the access to partially replicated CPRDTs. Each replica of a CPRDT $x_i$ maintains a set of particles, $shard(x_i)$. The replica only knows the state of the particles in $shard(x_i)$; therefore, it can only enable query and update operations that require and affect those particles. Furthermore, the CPRDT replica only needs to receive update operations that affect the particles in $shard(x_i)$ in order to converge.

In the context of this work, we have defined several CPRDT version of existing CRDTs - Figure 2 presents the specification of a G-SSet CPRDT. We have evaluated the impact of using CPRDTs, showing that partially replicating a CPRDT can lead to important performance benefits when manipulating large CRDT objects.

### 4.1.4 Related work

Lattices and lattice composition have been studied in order theory [22]. Our work on composition builds on this work and applies it to the context of replicated data structures.

Partial replication has been addressed in a large number of works. In most of these works, such as Thor [30], PRACTI [15] and SwiftCloud [38, 51], partial replication refers to replicating a subset of the database objects. The research being pursued in this project extends these works by addressing the problem of convergence for partially replicated objects that can be modified concurrently.

1: **particle definition** A possible element of the set.
2: **payload** set $A$
3: **initial** $\varnothing$
4: **query** lookup(element $e$) : boolean $b$
5: **required particles** $\{e\}$
6: **let** $b = e \in A$
7: **update** add(element $e$)
8: **required particles** $\varnothing$
9: **affected particles** $\{e\}$
10: $A := A \cup \{e\}$
11: **merge** ($S$, $T$) : payload $U$
12: **let** $U.A = S.A \cup T.A$
13: **fraction** (particles $Z$) : payload $D$
14: **let** $D.A = A \cap Z\}$

Figure 2: State-based Grow-Only Set (G-set) with Partial Replication

This problem has been addressed by Deftu et. al. [24] in the context of set CRDTs. In contrast to this work, we are proposing a principled solution to the problem that can be used with any CRDT.

### 4.1.5 Summary

The use-cases studied in WP1 and the experience in modelling complex applications by some of the partners have shown that supporting the composition of multiple CRDTs was an important requirement. To address this requirement, we have been investigated two main approaches.

First, we proposed a solution that supports composition relying on a Map CRDT. In this approach, it is possible to add to an existing map a new CRDT (including other maps). This allows to compose CRDTs with the limitation that the structure of the data model must be acyclic. The proposed solution has been integrated in Riak 2.0 release.

Second, we have studied techniques for allowing the general composition of CRDTs. To this end, we identified elementary CRDTs and compositions rules that guarantee that the composed object is still a CRDT.

Large CRDTs can lead to performance problems as the system needs to handle these potentially large data objects. We have addressed these problems in two complementary works. The first proposes a solution for storing the elements of a Map CRDT in different keys, allowing the system to manipulate each of the inner object independently. The second work studies how to partition a large CRDT, by decomposing it in multiple parts that can be manipulated independently by the system.

## 4.2 Transactions and Replication

Creating CRDTs that maintain more complex data is only one part of the solution for addressing the requirements of applications. In some cases, applications need to

access and modify data that does not fit naturally in a large object. In such cases, there is a need to group operations in atomic units or transactions.

In this project, and continuing previous work from team members, we have studied the models and algorithms for executing transactions in the the context of weakly consistent geo-replicated systems. We have additionally proposed two general techniques for supporting efficient replication, namely a mechanism for tracking causality and a new model for efficiently propagating updates on CRDTs. We overview our contribution in the remaining of this section.

### 4.2.1 Foundations for efficient replication

**Dotted Version Vector Sets** Weakly consistent geo-replicated storage systems [23, 31, 32, 3, 27] follow a design where the data store is always writeable and concurrent writes may occur. While some systems use simple repair strategies, such as last-writer-wins, that may lead to lost updates, others merge the effects of all concurrent updates. CRDTs adopt this last strategy.

In such systems, it is important to track causality in an efficient and accurate way. Version vectors [37] are the mostly used technique for comparing pairs of replica versions and detect if they are equivalent, concurrent or if one makes the other obsolete. However, in the presence of multiple clients executing concurrent updates, version vectors lack the ability to accurately represent concurrent values when using one entry per server. Alternatively, version vector can accurately track causality when using one entry per client, but this approach leads to scalability issues.

To address this issue, we have proposed in the past a new and simple causality tracking mechanism, Dotted Version Vectors [1, 39], that overcomes these limitations allowing both scalable and fully accurate causality tracking. A Dotted Version Vector (DVV) is a logical clock which consists of a pair $(d, v)$, where $v$ is a traditional version vector and the dot $d$ is a pair $(i, n)$, with $i$ a node identifier and $n$ an integer. The dot is the version identifier and it represents the globally unique event being described, while the VV represents the causal past.

An event $a$ with DVV $((i_a, n_a), v_a)$ causally precedes an event $b$ with DVV $((i_b, n_b), v_b)$: $a < b$, iff $n_a \leq v_b[i_a]$ (i.e., the event identifier of $a$ is in the causal past of $b$). Two events are concurrent if neither causally precedes the other: $a \parallel b$ iff $n_a > v_b[i_a] \wedge n_b > v_a[i_b]$.

Although DVVs address the problem of tracking causality efficiently, for systems that maintain multiple concurrent object versions, storing a DVV for each one may still represent an important overhead. To address this issue, we have proposed a new container - Dotted Version Vector Sets (DVVSet) [1] - that efficiently compacts a set of concurrent DVVs in a single data structure. The key idea is to factorize common knowledge for the set of DVVs described, keeping only the strictly relevant information in a single data structure. This results in not only a very succinct representation, but also in reduced time complexity of operations: the concurrent values will be indexed and ordered in the data structure, and traversal will be efficient.

Basho has adopted DVV and DVVSets in the latest release of the Riak database. The same idea, of decoupling the update identifier and its causal past has been used

in the transactional protocols described in the next subsection.

**Delta Mutators** State-based CRDTs are preferred to operation-based when causal delivery is not guaranteed by the messaging middlware or when it is necessary to propagate a large number of updates. However, shipping the complete state when a single update has been executed becomes expensive when CRDTs get large.

We have addressed this issue by rethinking the way that state-based CRDTs are designed and synchronized [2], having in mind the problematic shipping of the entire state. Our aim is to ship a representation of the effect of recent update operations on the state, rather than the whole state, while preserving the idempotent nature of join. Thus, this allows unreliable communication, on the contrary to operation-based CRDTs that demand exactly-once delivery and are prone to message replays.

To achieve this, we introduced Delta State-based CRDTs ($\delta$-CRDT): a state is a join-semilattice that results from the join of multiple fine-grained states, i.e., deltas, generated by what we call $\delta$-mutators; these are new versions of the datatype mutators that return the effect of these mutators on the state. In this way, deltas can be retained in a buffer to be shipped individually (or joined in groups) instead of shipping the entire object. The changes to the local state are then incorporated at other replicas by joining the shipped deltas with their own states.

The challenge in this approach is to make sure that decomposing a CRDT into deltas and then joining them into another replica state (after shipping) produces the same effect as if the entire state had been shipped and merged. This is on-going work, but we have already produced a library of $\delta$-CRDTs that is publicly available [10].

### 4.2.2 Transactional Causal+ Consistency

Cloud platforms improve availability and latency by geo-replicating data in several data centers (DCs) across the world [23, 31, 32, 3, 27, 21, 29, 20]. Nevertheless, the closest DC is often still too far away for an optimal user experience. Caching data at client machines can improve latency and availability for many applications, and even allow for a temporary disconnection. While increasingly used, this approach often leads to ad-hoc implementations that integrate poorly with server-side storage and tend to degrade data consistency guarantees.

Although extending geo-replication to the client machine seems natural, it raises two big challenges. The first one is to provide programming guarantees for applications running on client machines, at a reasonable cost at scale and under churn. Recent DC-centric storage systems [47, 31, 32] provide transactions, and combine support for causal consistency with mergeable objects, i.e., CRDTs. Extending these guarantees to the clients is problematic for a number of reasons: standard approaches to support causality in client nodes require vector clocks entries proportional to the number of replicas; seamless access to client and server replicas require careful maintenance of object versions; fast execution in the client requires asynchronous commit. We developed protocols that efficiently address these issues despite failures, by combining a set of novel techniques.

Client-side execution is not always beneficial. For instance, computations that access a lot of data, such as search or recommendations is best done in the DC.

We show how to support server-side execution, without breaking the guarantees of client-side in-cache execution.

The second challenge is to maintain these guarantees when the client-DC connection breaks. Upon reconnection, possibly to a different DC, the outcome of the client's in-flight transactions is unknown, and state of the DC might miss the causal dependencies of the client. We discuss how to address this challenge in the context of WP2 report.

**System model** SwiftCloud is a data storage systems for cloud platforms that spans both client nodes and data center servers (DCs). The core of the system consists of a set of DCs that replicate every object. At the periphery, applications running in client nodes access the system through a local module called scout. A scout caches a subset of the objects.

SwiftCloud provides a straightforward transactional key-object API. An application executes transactions by interactively executing sequences of reads and updates, concluded by either a commit or rollback.

Our transactional model, Transactional Causal+ Consistency, offers the following guarantees: every transaction reads a causally consistent snapshot; updates of a transaction are atomic (all-or-nothing) and isolated (no concurrent transaction observes an intermediate state); and concurrently committed updates do not conflict.

This transactional model allows different clients to observe the same set of concurrent updates applied in different orders, which poses a risk of yielding different operation outcomes on different replicas or at different times. We address this problem by disallowing non-commutative (order-dependent) concurrent updates. Practically, we enforce this property with *Mergeable* transactions.

Mergeable transactions commute with each other and with non-mergeable transactions, which allows to execute them immediately in the cache, commit asynchronously in the background, and remain available in failure scenarios. Mergeable transaction are either read-only transaction or update transactions that modify CRDTs. Next, we present the key ideas for supporting mergeable transactions.

**Algorithms** An application issues a mergeable transaction iteratively through the scout. Reads are served from the local scout; on a cache miss, the scout fetches the data from the DC it is connected to. Updates execute in a local copy. When a mergeable transaction terminates, it is locally committed and updates are applied to the scout cache. Updates are also propagated to a DC for being globally committed. The DC eventually propagates the effects of transactions to other DCs and other scouts as needed.

*Atomicity and Isolation:* For supporting atomicity and isolation, a transaction reads from a database snapshot. Each transaction is assigned a DC timestamp by the DC that received it from the client. Each DC maintains a vector clock with the summary of all transactions that have been executed in that DC, which is updated whenever a transaction completes its execution in that DC. This vector has as $n$ entries, with $n$ the number of DCs. Each scout maintains a vector clock with the version of the objects in the local cache.

When a transaction starts in the client, the current version of the cache is selected as the transaction snapshot. If the transaction accesses an object that is not present in the cache, the appropriate version is fetched from the DC - to this end, DCs maintain recent versions of each object.

*Read your writes:* When a transaction commits in the client, the local cache is updated. The following transactions access a snapshot that includes these locally committed transactions. To this end, each transaction executed in the client is assigned a scout timestamp. The vector that summarizes the transactions reflected in the local cache has $n+1$ entries, with the additional entry being used to summarize locally submitted transactions. This approach guarantees that a client always reads a state that reflects his previous transactions.

*Causality:* The system ensures the invariant that every node (DC or scout) maintains a causally-consistent set of object versions. To this end, a transaction only executes in a DC after its dependencies are satisfied - the dependencies of a transaction, summarized in the transaction snapshot, are propagated both from the client to the initial DC and from one DC to other DCs. The combination of the server timestamp and the transaction dependencies form a Dotted Version Vector, introduced previously.

When a scout caches some object, the DC it is connected to becomes responsible of notifying it with updates to those cached objects. SwiftCloud includes a notification subsystem that guarantees that updates from a committed transaction are propagated atomically and respecting causality. As a result, the cache in the scout is also causally consistent.

We have implemented SwiftCloud, as detailed elsewhere [52, 38, 51]. The evaluation in a cloud environment, using Amazon AWS DCs for running the servers and PlanetLab nodes for running the client shows that extending geo-replication to the client machine leads to a huge latency and throughput benefit for scenarios that exhibit good locality, a property verified in real workloads.

The algorithms being developed in the context of WP2 build on the knowledge and ideas of this work. The new algorithms focus on scaling the execution in the DCs, by avoiding the use of any centralized component. Besides this work, in the future we intend to explore two main directions. First, the design of efficient algorithms for supporting also traditional strong transactions, akin to the model of Walter [47] or Red-Blue [29]. Second, to study the implementation of weaker transactional models that provide only read committed isolation, addressing requirements of some use cases of WP1. In this context, we intend to understand how further relaxing isolation can help improving latency, availability and performance.

### 4.2.3 Related work

Cloud storage systems provide a wide range of consistency models. Some systems [21, 53, 26, 34] provide strong consistency, at the cost of unavailability when a replica is unreachable (network partitions). At the opposite end of the spectrum, some systems [23] provide only eventual consistency (EC), but allow any replica to perform updates even when the network is partitioned. Other systems' consistency model lies between these two extremes or combine both models [47, 29, 48, 20, 46].

Causal consistency strengthens EC with the guarantee that if a write is observed, all previous writes are also observed. [33] show that, in the presence of partitions, this is the strongest possible guarantee in an always-available, one-way convergent system. To cope with concurrent updates, Causal+ Consistency incorporate mergeable data. This is the model of COPS [31], Eiger [32] and ChainReaction [3]. These systems merge by last-writer-wins.

We extended Causal+ Consistency with mergeable transactions. COPS and ChainReaction implement read-only transactions that are non-interactive, i.e, the read set is known from the beginning. Eiger additionally supports non-interactive write-only transactions. Our work extends this work with interactive transactions and support for DC failover. An approach similar to ours, including the study of isolation levels, session guarantees and causality, was proposed by Bailis et. al. [6]. Burckhardt et. al. [19] also provide a model of transactions for EC that uses a centralized main revision, being more suitable for smaller databases.

Systems that support strong consistency often present support for transactions that provide serializable semantics [21, 53] or variants of snapshot isolation [47, 4]. When compared with mergeable transactions, these approaches offer stronger semantics by incurring in higher latency for transaction execution, as multiple replicas need to be contacted before a transaction commit.

### 4.2.4 Summary

Applications often need to access and modify data that does not fit naturally in a large composed object. In these cases, it is necessary to group operations for providing correct behavior. We have proposed the transactional causal+ consistency model and algorithms that allow a transaction to access a causally consistent database snapshot while concurrent updates are merged relying on CRDTs. These algorithms execute in a geo-replicated storage, SwiftCloud, that we have continued to develop during this period. This work has served as the basis for the transaction protocols proposed and implemented in Antidote.

We have also proposed two general techniques for supporting efficient replication. First, a mechanism for efficiently tracking causality in geo-replicated data stores, by keeping metadata of size $O(\#replicas)$. Second, a new model for propagating updates in state-based CRDTs, that allows to propagate deltas instead of the full CRDT state.

## 4.3 Invariants

Systems that adopt weak consistency models have to deal with concurrent operations not seeing the effects of each other. If CRDTs can be used to guarantee eventual convergence in these cases, they cannot be used to guarantee that application invariants are enforced, which can lead to non-intuitive and undesirable semantics.

Semantic anomalies do not occur in systems that offer strong consistency guarantees, namely those that serialize all updates [21, 29, 48]. However, these consistency models require coordination among replicas, which increases latency and decreases availability. An alternative is to try to combine the strengths of both approaches by

supporting both weak and strong consistency for different operations, as supported by SwiftCloud [38, 51] and other systems [47, 29, 48]. However, operations requiring strong consistency still incur in high latency.

To address these issues we have investigated two approaches to maintain application invariants. The first addresses numeric invariants, which accounts for an important class of application invariants, and can be deployed as a middleware on top of existing key-value stores. The second is more general and can address generic application invariants.

### 4.3.1 Middleware for enforcing numeric invariants

Our first work focused on enforcing numeric invariants [7] in the presence of concurrent updates to counter objects, thus helping to address the requirements of the ad counter use case studied in WP1.

In our work, we showed that fast geo-replicated operations on counters can coexist with strong invariants. To this end, we proposed a novel abstract data type called Bounded Counter. This replicated object, like conventional CRDTs, allows for operations to execute locally, automatically merges concurrent updates, and, in contrast to previous counter CRDTs, also enforces numeric invariants while avoiding any coordination in most cases.

This work builds on some ideas previously developed in the context of escrow transactions [36] and adapt them to run efficiently in cloud environments. The basic idea is to consider that the difference between the value of a counter and its bound can be seen as a set of rights to execute operations. For example, in a counter, $n$, with initial value $n = 40$ and invariant $n \geq 10$, there are 30 $(40 - 10)$ rights to execute decrement operations. Executing *dec(5)* consumes 5 of these rights. Executing *inc(5)* creates 5 rights. These rights can be split among the replicas of the counter – e.g. if there are 3 replicas, each replica can be assigned 10 rights. If the rights needed to execute some operation exist in the local replica, the operation can execute safely locally, knowing that the global invariant will not be broken – in the previous example, if the decrements of each replica are less or equal to 10, it follows immediately that the total decrements are at most 30 and the invariant still holds. If not enough rights exist, then either the operation fails or additional rights must be obtained from other replicas.

Unlike previous works that include some central authority [36, 40, 45] and are often based on synchronous interactions between nodes, our approach is completely decentralized and asynchronous, relying on maintaining the necessary information for enforcing the invariant in a new CRDT – the Bounded Counter CRDT. This allows for replicas to synchronize peer-to-peer and asynchronously, thus minimizing the deployment requirements and avoiding situations where the temporary unreachability of the master data center can prevent operations from making progress

For deploying the Bounded Counter in existing cloud infrastructures, we have developed two middleware designs. While the first design is implemented using only a client-side library, the second includes server side components deployed in a distributed hash table. Both designs require only that the underlying cloud store executes operations sequentially in each replica (not necessarily by the same order across replicas) and that it provides a reconciliation mechanism that allows for

merging concurrent updates. This makes our solutions generic and portable, but for achieving performance comparable with accessing directly to the underlying cloud store our middleware had to include a set of techniques to minimize overhead, which are fully described elsewhere [7].

Our evaluation shows that the proposed approach enforces numeric invariants with latency similar to weak consistency systems.

### 4.3.2 Explicit consistency

We have also proposed a general approach for maintaining applications invariants, based on *explicit consistency* [8, 9].

*Explicit consistency* is a novel consistency semantics for replicated systems. The high level idea is to let programmers define the application-specific correctness rules that should be met at all times. These rules are defined as invariants over the database state.

Given the invariants expressed by the programmer, we propose a methodology for enforcing explicit consistency that has three steps: (i) detect the sets of operations that may lead to invariant violation when executed concurrently (we call these sets *I-offender sets*); (ii) select an efficient mechanism for handling *I-offender sets*; (iii) instrument the application code to use the selected mechanism in a weakly consistent database system.

The first step consists of discovering *I-offender sets*. For this analysis, it is necessary to model the effects of operations. This information should be provided by programmers, in the form of annotations specifying how predicates are affected by each operation [2]. Using this information and the invariants, a static analysis process infers the minimal sets of operation invocations that may lead to invariant violation when executed concurrently (*I-offender sets*), and the reason for such violation.

The second step consists in deciding which approach will be used to handle *I-offender sets*. The programmer must select from the two alternative approaches supported: *invariant-repair*, in which operations are allowed to execute concurrently and invariants are enforced by automatic conflict resolution rules; *violation-avoidance*, in which the system restricts the concurrent execution of operations that can lead to invariant violation.

Third, the application code is instrumented to use the conflict-repair and conflict-avoidance mechanisms selected by the programmer. This involves extending operations to call the appropriate API functions of a system that support such mechanisms. We have designed and implemented such system on top of SwiftCloud [51, 38].

In this deliverable we briefly discuss how to avoid invariant violation based on reservations. Other aspects of the proposed methodology are addressed in the deliverable of WP4.1.

**Reservations** For avoiding operations that can lead to invariant violation from executing concurrently, we use the following techniques.

[2] This step could be automated using program analysis techniques, as done for example in [28, 42].

**UID generator:** The system provides a unique identifier generator, which splits the space of identifiers among replicas. This is an important source of invariant violations, as discussed elsewhere [29, 5].

**Escrow reservation:** For numeric invariants of the form $x \geq k$, we include an escrow reservation for allowing decrements to be executed without coordination. Given an initial value for $x = x_0$, there are initially $x_0 - k$ rights to execute decrements. These rights can be split by different replicas. For executing $x.decrement(n)$, the operation must acquire and consume $n$ rights to decrement $x$ in the replica it is submitted. If not enough rights exist in the replica, the system will try to obtain additional rights from other replicas. If this is not possible, the operation will fail. Executing $x.increment(n)$ creates $n$ rights to decrement $n$ initially assigned to the replica in which the operation that executes the increment is submitted.

A similar approach is used for invariants of the form $x \leq k$, with increments consuming rights and decrements creating new rights. For invariants of the form $x+y+\ldots+z \geq k$, a single escrow reservation is used, with decrements to any of the involved variables consuming rights and increments creating rights. If a variable $x$ is involved in more than one invariant, several escrow reservations will be affected by a single increment/decrement operation on $x$.

**Multi-level lock reservation:** When the invariant in risk is not numeric, we use a multi-level lock reservation (or simply multi-level lock) to restrict the concurrent execution of operations that can break invariants. A multi-level lock can provide the following rights: (i) *shared forbid*, giving the shared right to forbid some action to occur; (ii) *shared allow*, giving the shared right to allow some action to occur; (iii) *exclusive allow*, giving the exclusive right to execute some action.

When a replica holds some right, it knows no other replica holds rights of a different type - e.g. if a replica holds a *shared forbid*, it knows no replica has any form of *allow*.

**Multi-level mask reservation:** For invariants of the form $P_1 \vee P_2 \vee \ldots \vee P_n$, the concurrent execution of any pair of operations that makes two different predicates false may lead to an invariant violation if all other predicates were originally false. In our analysis, each of these pairs is an *I-offender set*.

Using simple multi-level locks for each pair of operations is too restrictive, as getting a *shared allow* on one operation would prevent the execution of the other operation in all pairs. In this case, for executing one operation is suffices to guarantee that a single other operation is forbidden (assuming that the predicate associated with the forbidden operation is true).

To this end, we propose multi-level mask reservation that maintains the same rights as multi-level lock regarding a set of $K$ operations. With multi-level mask, when obtaining a *shared allow* right for some operation, it is necessary to obtain (if it does not exist already) a *shared forbid* right on some other operation.

**Indigo system** We have built a prototype named Indigo on top of the SwiftCloud geo-replicated data store, leveraging on the following properties: (i) causal consistency; (ii) support for transactions that access a database snapshot and merge concurrent updates using CRDTs; (iii) linearizable execution of operations for each object in each datacenter.

The details of the implementation are described in Balegas et. al. [8]. The evaluation in a geo-replicated environment shows that the proposed approach can enforce application invariants while most operations complete in the local datacenter, thus providing a much lower latency than solutions requiring coordination among replicas.

In the future, we intend to focus on studying approaches that implement invariant repair across multiple objects. This approach has the potential to further reduce the required synchronization among replicas.

### 4.3.3 Related work

A large number of systems supporting weakly consistent geo-replication emerged in recent years [23, 31, 32, 3, 27]. These systems cannot address the requirements of applications that require (some operations to execute under) strong consistency for correctness.

Other geo-replicated systems provide strong consistency [21, 53, 26, 34], or a combination of weak and strong consistency [47, 29, 48, 20, 46]. Unlike these systems, our proposals enforce application invariants by exploring application semantics to let (most) operations execute in a single datacenter.

A number of systems have been proposed for maintaining application invariants in a distributed way. Escrow transactions [36] offer a mechanism for enforcing numeric invariants under concurrent execution of transactions. By enforcing local invariants in each transaction, they can guarantee that a global invariant is not broken. This idea can be applied to other data types, and it has been explored for supporting disconnected operation in mobile computing [49, 40, 45]. The demarcation protocol [12] is aimed at maintaining invariants in distributed databases. Although its underlying protocols are similar to escrow-based approaches, it focuses on maintaining invariants across different objects. Warranties [25] provide time-limited assertions over the database state, which can improve latency of read operations in cloud storages.

Our work builds on these works, but it is the first to provide an approach that, starting from application invariants expressed in first-order logic leads to the deployment of the appropriate techniques for enforcing such invariants in a geo-replicated weakly consistent data store.

### 4.3.4 Summary

Some applications need to maintain invariants that cannot be enforced in a pure weakly consistent replicated store. To address this issue, we have proposed two approaches to enforce application invariants while minimizing the required coordination and moving the needed coordination outside of the critical path of operation execution.

The first approach addresses numeric invariants, which are an important class of application invariants. The solution uses a new CRDT, the Bounded Counter, and a middleware design to enforce invariants in existing key-value stores.

The second approach allows to enforce generic application invariants, by relying on reservations [40]. We propose a novel methodology that, starting from

application invariants and operation side-effects, helps programmers deploying a reservation system that enforces invariants with minimal coordination.

## 4.4 Final remarks

According to the DOW, the goal of this deliverable was to "report on the mechanisms for enforcing guarantees in the presence of CRDT composition and transactions", by "studying composition mechanisms", "which transaction properties are required to maintain different classes of invariants" and "identifying at what point stronger guarantees require introducing small doses of synchronisation".

The contributions produced during this first year by the project consortium reached the goals of task 3.1. Namely, we have studied different composition mechanisms that still guarantee data convergence, name a composition based on a Map CRDT and a generic composition framework. For providing stronger guarantees and maintaining invariants, we have researched two main complementary mechanisms. Mergeable transactions provide atomicity guarantees and allow accessing a database snapshot. We have shown how to enforce generic application invariants with low latency by moving the required coordination outside of the critical path of operations and by amortizing the cost of coordination over multiple operations.

These works have been described in several papers that are either published or under submission, and several prototypes have been created and are publicly available in the project repository (some works are available in other public repositories, as mentioned in this report).

As discussed throughout this section, some of the works described are still ongoing, either because we are awaiting successful publication or because new directions have been uncover during our work. During this first year, some works produced in the context of this WP have contributed to development of the Antidote prototype, in coordination with WP2. During the next year we intend to continue contributing and integrating some of the developed techniques in Antidote, namely the support for partial replication of large and composed CRDTs and the techniques for enforcing invariants.

The work produced in the task has received as input the requirements identified in the use-cases of WP1. As discussed throughout the report, where appropriate, some works have different aspects that conceptually belong to multiple work packages, namely: (i) the work on composition is related with both WP2 and WP4 - the Map CRDT and techniques for partially replicating large CRDTs are expected to be integrated in the Antidote prototype with adequate programming support; (ii) the work on mergeable transactions served as the basis for the protocols developed in WP2; (iii) the work on invariants is related with WP4 and we expect to integrate our proposals in the prototype being developed in WP2 in the next year.

# 5 Papers and Publications

The work performed in the context of WP3 and in collaboration with other work packages has led to several papers. The following papers have been accepted and published during this period:

- [1] Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno Preguiça, and Victor Fonte. Scalable and accurate causality tracking for eventually consistent stores. In Proceedings of the 14th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems, DAIS 2014, Berlin, Germany, June 3-5, 2014, Proceedings, volume 8460 of Lecture Notes in Computer Science, pages 67–81. Springer, 2014. (appendix A.3)

- [2] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based crdts by decomposition. In Proceedings of the First Workshop on Principles and Practice of Eventual Consistency, PaPEC '14, New York, NY, USA, 2014. ACM. (appendix A.4)

- [9] Valter Balegas, Mahsa Najafzadeh, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Marc Shapiro, and Nuno Preguiça. The Case for Fast and Invariant-Preserving Geo-Replication. In Proceedings of the SRDS Workshop on Planetary-Scale Distributed Systems, October 2014. (appendix A.6)

- [18] Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. Riak dt map: A composable, convergent replicated dictionary. In Proceedings of the First Workshop on Principles and Practice of Eventual Consistency, PaPEC '14, New York, NY, USA, 2014. ACM. (appendix A.1)

- [35] Christopher Meiklejohn. On the composability of the riak dt map: Expanding from embedded to multi-key structures. In Proceedings of the First Workshop on Principles and Practice of Eventual Consistency, PaPEC '14, New York, NY, USA, 2014. ACM. (appendix A.2)

- [38] Nuno Preguiça, Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, and Marc Shapiro. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine (invited talk). In Proceedings of the SRDS Workshop on Planetary-Scale Distributed Systems, October 2014. (appendix A.5)

The following paper are under submission or being prepared for submission.

- [7] Valter Balegas, Mahsa Najafzadeh, Sergio Duarte, Carla Ferreira, Rodrigo Rodrigues, Marc Shapiro, and Nuno Preguiça. Extending Eventually Consistent Cloud Stores for Enforcing Numeric Invariants. Technical report, 2014. (appendix B.4)

- [8] Valter Balegas, Mahsa Najafzadeh, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Marc Shapiro, and Nuno Preguiça. Putting Consistency Back into Eventual Consistency. Submitted to EuroSys'2015, 2014. (appendix B.5)

- [11] Carlos Baquero, Paulo Sérgio Almeia, Alcino Cunha, and Alcin Ferreira. Com- position of state-based CRDTs. Technical report, U. Minho, 2014. (appendix B.1)
- [17] Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict- free partially replicated data types. Submitted to 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2014. (appendix B.2)
- [52] Marek Zawirski, Nuno Preguiça, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. Write Fast, Read in the Past: Causal Consistency for Client-side Applications. Submitted to EuroSys'2015, 2014. (appendix B.3)

# References

[1] Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno M. Preguiça, and Victor Fonte. Scalable and accurate causality tracking for eventually consistent stores. In *Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, volume 8460 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2014.

[2] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based crdts by decomposition. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 3:1–3:2, New York, NY, USA, 2014. ACM.

[3] Sérgio Almeida, Joao Leitão, and Luís Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[4] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems*, SRDS '13, pages 163–172, Washington, DC, USA, 2013. IEEE Computer Society.

[5] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination-avoiding database systems. *CoRR*, abs/1402.2237, 2014.

[6] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Hat, not cap: Towards highly available transactions. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 24–24, Berkeley, CA, USA, 2013. USENIX Association.

[7] Valter Balegas, Mahsa Najafzadeh, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Marc Shapiro, and Nuno Preguiça. Extending Eventually Consistent Cloud Stores for Enforcing Numeric Invariants. Technical Report, 2014.

[8] Valter Balegas, Mahsa Najafzadeh, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Marc Shapiro, and Nuno Preguiça. Putting Consistency Back into Eventual Consistency. Submitted to EuroSys'2015, 2014.

[9] Valter Balegas, Mahsa Najafzadeh, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Marc Shapiro, and Nuno Preguiça. The Case for Fast and Invariant-Preserving Geo-Replication. In *Proceedings of the SRDS Workshop on Planetary-Scale Distributed Systems*, October 2014.

[10] Carlos Baquero. Delta crdt library. `https://github.com/CBaquero/delta-enabled-crdts`, 2014.

[11] Carlos Baquero, Paulo Sérgio Almeia, Alcino Cunha, and Alcin Ferreira. Composition of state-based crdts. Technical report, U. Minho, 2014.

[12] Daniel Barbará-Millá and Hector Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, July 1994.

[13] Basho. Riak dt library. `https://github.com/basho/riak_dt`, 2014.

[14] Basho. Using data types. `http://docs.basho.com/riak/2.0.0/dev/using/data-types/`, 2014.

[15] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association.

[16] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. Rapport de recherche RR-8083, INRIA, October 2012.

[17] Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict-free partially replicated data types. Submitted to 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2014.

[18] Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. Riak dt map: A composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 1:1–1:1, New York, NY, USA, 2014. ACM.

[19] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Mooly Sagiv. Eventually consistent transactions. In *European Symposium on Programming (ESOP)*, Tallinn, Estonia, March 2012.

[20] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[21] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[22] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order (2. ed.)*. Cambridge University Press, 2002.

[23] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[24] Andrei Deftu and Jan Griebsch. A scalable conflict-free replicated set data type. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pages 186–195, Washington, DC, USA, 2013. IEEE Computer Society.

[25] ed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, nsdi'14, Berkeley, CA, USA, 2014. USENIX Association.

[26] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[27] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[28] Cheng Li, J. Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association.

[29] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[30] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 230–257, London, UK, UK, 1999. Springer-Verlag.

[31] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[32] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings*

*of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[33] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. Technical Report UTCS TR-11-22, Dept. of Comp. Sc., The U. of Texas at Austin, Austin, TX, USA, 2011.

[34] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9):661–672, July 2013.

[35] Christopher Meiklejohn. On the composability of the riak dt map: Expanding from embedded to multi-key structures. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 13:1–13:2, New York, NY, USA, 2014. ACM.

[36] Patrick E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, December 1986.

[37] 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. Softw. Eng.*, 9(3):240–247, May 1983.

[38] Nuno Preguiça, Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, and Marc Shapiro. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine (invited talk). In *Proceedings of the SRDS Workshop on Planetary-Scale Distributed Systems*, October 2014.

[39] Nuno Preguiça, Carlos Bauqero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Brief announcement: Efficient causality tracking in distributed storage systems with dotted version vectors. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 335–336, New York, NY, USA, 2012. ACM.

[40] Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, and Henrique Domingos. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 43–56, New York, NY, USA, 2003. ACM.

[41] Nuno Preguiça, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society.

[42] Sudip Roy, Lucja Kot, Nate Foster, Johannes Gehrke, Hossein Hojjat, and Christoph Koch. Writes that fall in the forest and make no sound: Semantics-based adaptive data consistency. *CoRR*, abs/1403.2307, 2014.

[43] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, January 2011.

[44] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and V. Villain, editors, *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes on Computer Science*, pages 386–400, Grenoble, France, October 2011. Springer.

[45] Liuba Shrira, Hong Tian, and Doug Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 42–61, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[46] Swaminathan Sivasubramanian. Amazon dynamodb: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM.

[47] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[48] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[49] G. D. Walborn and P. K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Proceedings of the 14TH Symposium on Reliable Distributed Systems*, SRDS '95, pages 31–, Washington, DC, USA, 1995. IEEE Computer Society.

[50] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 404–412, Washington, DC, USA, 2009. IEEE Computer Society.

[51] Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, Marc Shapiro, and Nuno Preguiça. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. Rapport de recherche RR-8347, INRIA, October 2013.

[52] Marek Zawirski, Nuno Preguiça, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. Write Fast, Read in the Past: Causal Consistency for Client-side Applications. Submitted to EuroSys'2015, 2014.

[53] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, New York, NY, USA, 2013. ACM.

# A Published papers

## A.1 Russell Brown, Sean Cribbs, Sam Elliot, Christopher Meiklejohn. Riak DT Map: A Composable, Convergent Replicated Dictionary. In Proc. PaPEC 14.

# Riak DT Map: A Composable, Convergent Replicated Dictionary

Russell Brown
Basho Technologies, Inc.
russelldb@basho.com

Sean Cribbs
Basho Technologies, Inc.
sean@basho.com

Sam Elliott
Basho Technologies, Inc.
sam.elliott@basho.com

Christopher Meiklejohn
Basho Technologies, Inc.
cmeiklejohn@basho.com

## Abstract

Conflict-Free Replicated Data-Types (CRDTs) [6] provide greater safety properties to eventually-consistent distributed systems without requiring synchronization. CRDTs ensure that concurrent, uncoordinated updates have deterministic outcomes via the properties of bounded join-semilattices.

We discuss the design of a new convergent (state-based) replicated data-type, the Map, as implemented by the Riak DT library [4] and the Riak data store [3]. Like traditional dictionary data structures, the Map associates keys with values, and provides operations to add, remove, and mutate entries. Unlike traditional dictionaries, all values in the Map data structure are also state-based CRDTs and updates to embedded values preserve their convergence semantics via lattice inflations [1] that propagate upward to the top-level. Updates to the Map and its embedded values can also be applied atomically in batches. Metadata required for ensuring convergence is minimized in a manner similar to the optimized OR-set [5].

This design allows greater flexibility to application developers working with semi-structured data, while removing the need for the developer to design custom conflict-resolution routines for each class of application data. We also discuss the experimental validation of the data-type using stateful property-based tests with QuickCheck [2].

***Categories and Subject Descriptors*** C.2.4 [*Distributed Systems*]: Distributed databases; D.3.3 [*Programming Techniques*]: Language Constructs and Features - abstract data types, patterns, control structures; E.1 [*Data Structures*]: Distributed data structures; H.2.4 [*Database Management Systems*]: Distributed databases

***Keywords*** Dynamo, Eventual Consistency, Data Replication, Commutative Operations, Riak, Erlang, Property-based Testing, QuickCheck



*PaPEC '14,* April 13-16, 2014, Amsterdam, Netherlands.

ACM 978-1-4503-2716-9/14/04. . . $15.00.
http://dx.doi.org/10.1145/2596631.2596633

## Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 609551.

## References

[1] P. S. Almeida, C. Baquero, and A. Cunha. Composing Lattices and CRDTs. In B. Kemme, G. Ramalingam, A. Schiper, M. Shapiro, and K. Vaswani, editors, *Consistency in Distributed Systems (Dagstuhl Seminar 13081)*, volume 3, pages 92–126, Dagstuhl, Germany, Feb. 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. . URL `http://www.dagstuhl.de/mat/Files/13/13081/13081.BaqueroCarlos.Slides.pdf`.

[2] T. Arts, L. M. Castro, and J. Hughes. Testing Erlang Data Types with Quviq Quickcheck. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, ERLANG '08, pages 1–8, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-065-4. . URL `http://doi.acm.org/10.1145/1411273.1411275`.

[3] Basho Technologies, Inc. Riak source code repository. `https://github.com/basho/riak`, 2009-2014.

[4] Basho Technologies, Inc. Riak DT source code repository. `https://github.com/basho/riak_dt`, 2012-2014.

[5] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. *ArXiv e-prints*, Oct. 2012.

[6] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, Jan. 2011. URL `http://hal.inria.fr/inria-00555588`.

## A.2 Christopher Meiklejohn. On The Composability of the Riak DT Map: Expanding From Embedded To Multi-Key Structures. In Proc. PaPEC 14.

# On The Composability of the Riak DT Map: Expanding From Embedded To Multi-Key Structures

(Work in progress report)

Christopher Meiklejohn
Basho Technologies, Inc.
cmeiklejohn@basho.com

## Abstract

The Riak DT library [2] provides a composable, convergent replicated dictionary called the Riak DT map, designed for use in the Riak [1] replicated data store. This data type provides the ability for the composition of conflict-free replicated data types (CRDT) [7] through embedding.

Composition by embedding works well when the total object size of the composed CRDTs is small, however suffers a performance penalty as object size increases. The root of this problem is based in how replication is achieved in the Riak data store using Erlang distribution. [4]

We propose a solution for providing an alternative composition mechanism, composition by reference, which provides support for arbitrarily large objects while ensuring predictable performance and high availability. We explore the use of this new composition mechanism by examining a common use case for the Riak data store.

***Categories and Subject Descriptors*** C.2.4 [*Distributed Systems*]: Distributed databases; D.3.3 [*Programming Techniques*]: Language Constructs and Features - abstract data types, patterns, control structures; E.1 [*Data Structures*]: Distributed data structures; H.2.4 [*Database Management Systems*]: Distributed databases

***Keywords*** Dynamo, Eventual Consistency, Data Replication, Commutative Operations, Riak, Erlang

## 1. Introduction

The Riak DT library [2] provides a composable, convergent replicated dictionary called the Riak DT map, designed for use in the Riak [1] replicated data store. This data type provides the ability for the composition of conflict-free replicated data types (CRDT) [7] through embedding.

Composition by embedding works well when the total object size of the composed CRDTs is small, however suffers a performance penalty as object size increases. The root of this problem is based in how replication is achieved in the Riak data store using Erlang distribution. [4]



*PaPEC '14,* April 13-16 2014, Amsterdam, Netherlands.

ACM 978-1-4503-2716-9/14/04. . . $15.00.
http://dx.doi.org/10.1145/2596631.2596635

We propose a solution for providing an alternative composition mechanism, composition by reference, which provides support for arbitrarily large objects while ensuring predictable performance and high availability. We explore the use of this new composition mechanism by examining a common use case for the Riak data store.

## 2. Motivation

Consider a social network application where each user can create events that are visible to other users via a timeline. One way to implement this, that has been used by existing users of the Riak data store [5] [6], has been to model each user's timeline as lists of references to independent objects in the data store, one for each event with a unique key.

We can model this using the Riak DT map by creating a dictionary of entries in the map from timestamps to embedded maps containing the information for each post. Modeling the timeline objects this way has two problems:

- Once objects grow to be over one megabyte, a noticeable degradation in performance can be observed.
- Given Riak does not guarantee causal consistency, it is possible to observe references to objects that are not available. This is known to be true during failure conditions when primary replicas are not available and both read and write operations are handled with sloppy quorums. [3]

Given these limitations, we need a solution for providing an alternative composition mechanism that does not degrade in performance as size increases, but provides values at read time which observe the lattice properties of state-based CRDTs ensuring conflict-free merges with later state.

## 3. Solution

We explore a solution to the limitations of composition by embedding by proposing the following changes to the Riak data store:

- We extend the existing API as provided by Riak for interacting with the Riak DT map, to support the specification during a write of whether the object should be composed by embedding or by reference.
- When performing a write operation of an object containing references to other objects, we generate unique reference identifiers for each referenced object. Using these unique identifiers, we write the referencing objects first followed by the referenced objects in a recursive manner.

- When performing a read operation of an object containing references to other objects, we recursively attempt to retrieve the referenced objects from the data store.

The above changes are sufficient for providing causal consistency of objects when both the referencing and referenced objects are located across the same set of replicas, however we can not make that guarantee when attempting to provide equal distribution and high availability of data through consistent hashing and sloppy quorums, which is a core tenet of the Riak data store.

### 3.1 Sloppy quorums and disjoint replica sets

To support sloppy quorums, and the ability to compose objects by reference that span a disjoint replica set, we also need to provide a solution to handle objects that have been composed by reference when the referenced objects are not available. In the event of a referenced object becoming unavailable during a read operation, we can leverage the type information stored in the Riak DT map about composed objects to return the bottom value for the referenced object's type. This ensures that later read operations, where the previous missing reference is now available, correctly merges with the version where it was not.

## 4. Future work

In this section, we explore work which we believe will improve the performance and viability of this solution.

### 4.1 Parallel retrieval

Providing a mechanism for parallel retrieval of referenced objects in the map would help increase performance as the breadth of referenced objects increases, as we could launch jobs across disjoint replica sets which run in parallel. We believe that the Riak Pipe processing pipeline, which is used to support Riak's scatter-gather query mechanism would be appropriate for providing the substrate for this improvement.

### 4.2 Garbage collection

We are still exploring providing a solution for garbage collection of referenced objects. The major concerns of garbage collection arrive from two cases:

- When deleting objects, we need to ensure a recursive removal of all referenced objects. Given that the unique reference identifiers are known by the referencing objects, scheduling these for removal is not problematic. However, knowing how to properly schedule these removals when referenced objects might incur a concurrent update and removal is still unknown as this operation is not safe until the replicas are merged.
- When dealing with a partial failure, we need to ensure that any objects that have been written before the failure are scheduled for garbage collection. For example, when writing an object with three references, we need to make sure that we schedule both the referenced objects for garbage collection as well as the references.

## 5. Conclusion

In this work, we discuss the challenges involved in implementing this approach and the possible solutions. We explore the drawbacks of composing these values by reference and the problems of garbage collection when dealing with concurrent operations to composed objects.


## Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n$^{o}$ 609551.


## References


[1] Basho Technologies, Inc. Riak source code repository. `https://github.com/basho/riak`, .

[2] Basho Technologies, Inc. Riak DT source code repository. `https://github.com/basho/riak_dt`, .

[3] J. Blomstedt. Absolute consistency. `http://lists.basho.com/pipermail/riak-users_lists.basho.com/2012-January/007157.html`.

[4] Boundary. Incuriosity Killed the Infrastructure: Getting Ahead of Riak Performance and Operations. `http://boundary.com/blog/2012/09/26/incuriosity-killed-the-infrastructur/`.

[5] C. Hale and R. Kennedy. Riak and Scala at Yammer. `http://vimeo.com/21598799`.

[6] W. Moss and T. Douglas. Building A Transaction Logs-based Protocol On Riak. `http://vimeo.com/53550624`.

[7] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, Jan. 2011. URL `http://hal.inria.fr/inria-00555588`.

## A.3 Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno Preguiça, and Victor Fonte. Scalable and Accurate Causality Tracking for Eventually Consistent Stores. In Proc. DAIS 14.

# Scalable and Accurate Causality Tracking for Eventually Consistent Stores

Paulo Sérgio Almeida[1], Carlos Baquero[1],
Ricardo Gonçalves[1], Nuno Preguiça[2], and Victor Fonte[1]

[1] HASLab, INESC Tec & Universidade do Minho
{psa,cbm,tome,vff}@di.uminho.pt
[2] CITI/DI, FCT, Universidade Nova de Lisboa
nuno.preguica@fct.unl.pt

**Abstract.** In cloud computing environments, data storage systems often rely on optimistic replication to provide good performance and availability even in the presence of failures or network partitions. In this scenario, it is important to be able to accurately and efficiently identify updates executed concurrently. Current approaches to causality tracking in optimistic replication have problems with concurrent updates: they either (1) do not scale, as they require replicas to maintain information that grows linearly with the number of writes or unique clients; (2) lose information about causality, either by removing entries from client-id based version vectors or using server-id based version vectors, which cause false conflicts. We propose a new logical clock mechanism and a logical clock framework that together support a traditional key-value store API, while capturing causality in an accurate and scalable way, avoiding false conflicts. It maintains concise information per data replica, only linear on the number of replica servers, and allows data replicas to be compared and merged linear with the number of replica servers and versions.

## 1 Introduction

Amazon's Dynamo system [5] was an important influence to a new generation of databases, such as Cassandra [10] and Riak [9], focusing on partition tolerance, write availability and eventual consistency. The underlying rationale to these systems stems from the observation that when faced with the three concurrent goals of *consistency*, *availability* and *partition-tolerance* only two of those can be achievable in the same system [3,6]. Facing geo-replication operation environments where partitions cannot be ruled out, consistency requirements are inevitably relaxed in order to achieve high availability.

These systems follow a design where the data store is always writable: replicas of the same data item are allowed to temporarily diverge and to be repaired later on. A simple repair approach followed in Cassandra, is to use wall-clock timestamps to know which concurrent updates should prevail. This last writer wins (lww) policy may lead to lost updates. An approach which avoids this, must be able to represent and maintain causally concurrent updates until they can be reconciled.

Accurate tracking of concurrent data updates can be achieved by a careful use of well established causality tracking mechanisms [11,14,20,19,2]. In particular, for data

storage systems, version vectors (vv) [14] enable the system to compare any pair of replica versions and detect if they are equivalent, concurrent or if one makes the other obsolete. However, as we will discuss in Section 3, vv lack the ability to accurately represent concurrent values when used with server ids, or are not scalable when used with client ids.

We present a new and simple causality tracking solution, Dotted Version Vectors (briefly introduced in [16]), that overcomes these limitations allowing both scalable (using server ids) and fully accurate (representing same server concurrent writes) causality tracking. It achieves this by explicitly separating a new write event identifier from its causal past, which has the additional benefit of allowing causality checks between two clocks in constant time (instead of linear with the size of version vectors).

Besides fully describing Dotted Version Vectors (dvv), in this paper we make two novel contributions. First, we propose a new container (DVV Sets or dvvs) that efficiently compacts a set of concurrent dvv's in a single data structure, improving on two dvv limitations: (1) dvvs representation is independent of the number of concurrent values, instead of linear; (2) comparing and synchronizing two replica servers w.r.t. a single key is linear with the number of concurrent values, instead of quadratic.

Our final contribution is a general framework that clearly defines a set of functions that logical clocks need to implement to correctly track causality in eventually consistent systems. We implement both dvv and dvvs using this framework.

The rest of this paper is organized as follows. Section 2 presents the system model for the remaining paper. We survey and compare current mechanisms for causality tracking in Section 3. In Section 4, we present our mechanism dvv, followed by its compact version dvvs, in Section 5. We then propose in Section 6 a general framework for logical clocks and its implementation with both dvv and dvvs. In Section 7 we present the asymptotic complexities for both the current and proposed mechanisms, as well as an evaluation of dvvs. Additional techniques are briefly discussed in Section 8. We conclude in Section 9.

## 2 System Model and Data Store API

We consider a standard Dynamo-like key-value store interface that exposes two operations: get($key$) and put($key, value, context$). get returns a pair $(value(s), context)$, i.e., a value or set of causally concurrent values, and an opaque context that encodes the causal knowledge in the value(s). put submits a single value that supersedes all values associated to the supplied context. This context is either empty if we are writing a new value, or some opaque data structure returned to the client by a previous get, if we are updating a value. This context encodes causal information, and its use in the API serves to generate a *happens-before* [20] relation between a get and a subsequent put.

We assume a distributed system where nodes communicate by asynchronous message passing, with no shared memory. The system is composed by possibly many (e.g., thousands) clients which make concurrent get and put requests to server nodes (in the order of, e.g., hundreds). Each key is replicated in a typically small subset of the server nodes (e.g., 3 nodes), which we call the replica nodes for that key. These different or-

ders of magnitude of clients, servers and replicas play an important role in the design of a scalable causality tracking mechanism.

We assume: no global distributed coordination mechanism, only that nodes can perform internal concurrency control to obtain atomic blocks; no sessions or any form of client-server affinity, so clients are free to read from a replica server node and then write to a different one; no byzantine failures; server nodes have stable storage; nodes can fail without warning and later recover with their last state in the stable storage.

As we do not aim to track causality between different keys, in the remainder we will focus on operations over a single key, which we leave implicit; namely, all data structures in servers that we will describe are per key. Techniques as in [13] can be applied when considering groups of keys and could introduce additional savings; this we leave for future work.

## 3 Current Approaches

To simplify comparisons between different mechanisms, we will introduce a simple execution example between clients Mary and Peter, and a single replica node. In this example, presented in Figure 1, Peter starts by writing a new object version $\mathbf{v_1}$, with an empty context, which results in some server state $A$. He then reads server state $A$, returning current version $\mathbf{v_1}$ and context $\mathsf{ctx}_A$. Meanwhile, Mary writes a new version $\mathbf{v_2}$, with an empty context, resulting in some server state $B$. Since Mary wrote $\mathbf{v_2}$ without reading state $A$, state $B$ should contain both $\mathbf{v_1}$ and $\mathbf{v_2}$ as concurrent versions, if causality is tracked. Finally, Peter updates version $\mathbf{v_1}$ with $\mathbf{v_3}$, using the previous context $\mathsf{ctx}_A$, resulting in some state $C$. If causal relations are correctly represented, state $C$ we should only have $\mathbf{v_2}$ and $\mathbf{v_3}$, since $\mathbf{v_1}$ was superseded by $\mathbf{v_3}$ and $\mathbf{v_2}$ is concurrent with $\mathbf{v_3}$. We now discuss how different causality tracking approaches address this example, which are summarized in Table 1.

**Last Writer Wins (lww)** In systems that enforce a lww policy, such as Cassandra, concurrent updates are not represented in the stored state and only the last update prevails. Under lww, our example would result in the loss of $\mathbf{v_2}$. Although some specific application semantics are compatible with a lww policy, this simplistic approach is not adequate for many other application semantics. In general, a correct tracking of concurrent updates is essential to allow all updates to be considered for conflict resolution.

**Causal Histories (ch)** Causal Histories [20] are simply described by sets of unique write identifiers. These identifiers can be generated with a unique identifier and a monotonic counter. In our example, we used server identifiers $r$, but client identifiers could be used as well. The crucial point is that identifiers have to be globally unique to correctly represent causality. Let $id_n$ be the notation for the $n^{th}$ event of the entity represented by $id$. The partial order of causality can be precisely tracked by comparing these sets under set inclusion. Two ch are concurrent if neither includes the other: $A \parallel B$ iff $A \not\subseteq B$ and $B \not\subseteq A$. ch correctly track causality relations, as can be seen in our example, but have a major drawback: they grow linearly with the number of writes.

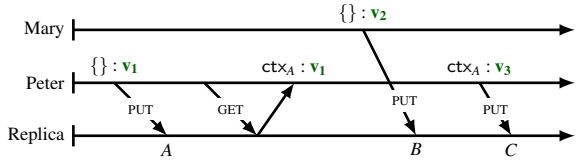



Fig. 1: Example execution for one key: Peter writes a new value $\mathbf{v_1}$ ($A$), then reads from Replica ($\mathsf{ctx}_A$). Next, Mary writes a new value $\mathbf{v_2}$ ($B$) and finally Peter updates $\mathbf{v_1}$ with $\mathbf{v_3}$ ($C$).

| | **lww** | **ch** | **vv$_{\text{client}}$** | **vv$_{\text{server}}$** | **dvv** | **dvvs** |
|---|---|---|---|---|---|---|
| $A$ | 17h00 : $\mathbf{v_1}$ | $\{r_1\} : \mathbf{v_1}$ | $\{(p,1)\} : \mathbf{v_1}$ | $\{(r,1)\} : \{\mathbf{v_1}\}$ | $((r,1),\{\}) : \mathbf{v_1}$ | $\{(r,1,[\mathbf{v_1}])\}$ |
| $\mathsf{ctx}_A$ | $\{\}$ | $\{r_1\}$ | $\{(p,1)\}$ | $\{(r,1)\}$ | $\{(r,1)\}$ | $\{(r,1)\}$ |
| $B$ | 17h03 : $\mathbf{v_2}$ | $\{r_1\} : \mathbf{v_1}$<br>$\{r_2\} : \mathbf{v_2}$ | $\{(p,1)\} : \mathbf{v_1}$<br>$\{(m,1)\} : \mathbf{v_2}$ | $\{(r,2)\} :$<br>$\{\mathbf{v_1},\mathbf{v_2}\}$ | $((r,1),\{\}) : \mathbf{v_1}$<br>$((r,2),\{\}) : \mathbf{v_2}$ | $\{(r,2,[\mathbf{v_2},\mathbf{v_1}])\}$ |
| $C$ | 17h07 : $\mathbf{v_3}$ | $\{r_2\} : \mathbf{v_2}$<br>$\{r_1,r_3\} : \mathbf{v_3}$ | $\{(m,1)\} : \mathbf{v_2}$<br>$\{(p,2)\} : \mathbf{v_3}$ | $\{(r,3)\} :$<br>$\{\mathbf{v_1},\mathbf{v_2},\mathbf{v_3}\}$ | $((r,2),\{\}) : \mathbf{v_2}$<br>$((r,3),\{(r,1)\}) : \mathbf{v_3}$ | $\{(r,3,[\mathbf{v_3},\mathbf{v_2}])\}$ |

Table 1: The table shows the replica (r) state after write from Peter (p) and Mary (m), and the context returned by Peter's read. We use the $metadata : value(s)$ notation, except for dvvs which has its own internal structure.

**Version Vectors (vv)** Version Vectors are an efficient representation of ch, provided that the ch has no gaps in each *id*'s event sequence. A vv is a mapping from identifiers to counters, and can be written as a set of pairs $(id, counter)$; each pair represents a set of ch events for that *id*: $\{id_n \mid 0 < n \leq counter\}$. In terms of partial order, $A \leq B$ iff $\forall (i,c_a) \in A \cdot \exists (i,c_b) \in B \cdot c_a \leq c_b$. Again, $A \parallel B$ iff $A \not\leq B$ and $B \not\leq A$. Whether client or server identifiers are used in vv has major consequences, as we'll see next.

**Version Vectors with Id-per-Client (vv$_{\text{client}}$)** This approach uses vv with clients as unique identifiers. An update is registered in a server by using the client identification issued in a put. This provides enough information to accurately encode the concurrency and causality in the system, since concurrent client writes are represented in the vv$_{\text{client}}$ with different ids. However, it sacrifices scalability, since vv$_{\text{client}}$ will end up storing the ids of all the clients that ever issued writes to that key. Systems like Dynamo try to compensate this by *pruning* entries in vv$_{\text{client}}$ at a specific threshold, but it typically leads to false concurrency and further need for reconciliation. The higher the degree of pruning, the higher is the degree of false concurrency in the system.

**Version Vectors with Id-per-Server (vv$_{\text{server}}$)** If causality is tracked with vv$_{\text{server}}$, i.e., using vv with server identifiers, it is possible to correctly detect concurrent updates that are handled by different server nodes. However, if concurrent updates are handled by the same server, there is no way to *express* the concurrent values — *siblings* — separately. To avoid overwriting siblings and losing information (as in lww), a popular solution to this, is to group all siblings under the same vv$_{\text{server}}$, losing individual causality information. This can easily lead to false concurrency: either a write's context

causally dominates the server $\mathsf{vv}_{\mathsf{server}}$, in which case all siblings are deemed obsolete and replaced by the new value; or this new value must be added to the current siblings, even if some of them were in its causal past.

Using our example, we finish the execution with all three values $\{\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}\}$, when in fact $\mathbf{v_3}$ should have obsoleted $\mathbf{v_1}$, like the other causally correct mechanisms in Table 2 (expect for lww).

With $\mathsf{vv}_{\mathsf{server}}$, false concurrency can arise whenever a client *read-write cycle* is interleaved with another concurrent write on the same server. This can become especially problematic under heavy load with many clients concurrently writing: under high latency, if a read-write cycle cannot be completed without interleaving with another concurrent write, the set of siblings will keep on growing. This will make messages grow larger, the server load heavier, resulting in a positive feedback loop, in what can be called a *sibling explosion*.

## 4 Dotted Version Vectors

We now present an accurate mechanism that can be used as a substitute for classic version vectors (vv) in eventually consistent stores, while still using only one *Id* per replica node. The basic idea of *Dotted Version Vectors* (dvv) is to take a vv and add the possibility of representing an individual causal event — *dot* — separate from the rest of the contiguous events. The dot is kept separate from the causal past and it globally and uniquely identifies a write. This allows representing concurrent writes, on the same server, by having different dots.

In our example from Figure 1, we can see that state $B$ is represented with a unique dot for both $\mathbf{v_1}$ and $\mathbf{v_2}$, even-though they both were written with an equally empty context. This distinction in their dots is what enables the final write by Peter to correctly overwrite $\mathbf{v_1}$, since the context supersedes its dot (and dvv), while maintaining $\mathbf{v_2}$ which has a newer dot than the context. In contrast, $\mathsf{vv}_{\mathsf{server}}$ loses this distinction gained by separating dots by grouping every sibling in one vv and thus cannot know that $\mathbf{v_1}$ is outdated by $\mathbf{v_3}$.

### 4.1 Definition

A dvv consists in a pair $(d,v)$, where $v$ is a traditional vv and the dot $d$ is a pair $(i,n)$, with $i$ as a node identifier and $n$ as an integer. The dot uniquely represents a write and its associated version, while the vv represents the causal past (i.e. its context). The causal events (or dots) represented by a dvv can be generated by a function toch that translates logical clocks to causal histories (ch can be viewed as sets of dots):

$$\begin{aligned} \mathsf{toch}(((i,n),v)) &= \{i_n\} \cup \mathsf{toch}(v), \\ \mathsf{toch}(v) &= \bigcup_{(i,n)\in v} \{i_m \mid 1 \leq m \leq n\}, \end{aligned}$$

where $i_n$ denotes the $n^{th}$ dot generated by node $i$, and $\mathsf{toch}(v)$ is the same function but for traditional vv. With this definition, the ch $\{a_1, b_1, b_2, c_1, c_2, c_4\}$ that cannot be represented by vv, can now be represented by the dvv $((c,4),\{(a,1),(b,2),(c,2)\})$.

### 4.2 Partial Order

The partial order on dvv can be defined in terms of inclusion of ch; i.e.:

$$X \leq Y \iff \mathsf{toch}(X) \subseteq \mathsf{toch}(Y),$$

Given that each dot is generated as a globally unique event — using the notational convenience $v[i] = n$, for $(i,n) \in v$ and $v[i] = 0$ for any non mapped id — the partial order on possible dvv values becomes:

$$((i,n),u) < ((j,m),v) \iff n \leq v[i] \wedge u \leq v,$$

where the traditional point-wise comparison of vv is used: $u \leq v \iff \forall_{(i,n)\in u}.\, n \leq v[i]$.

An important consequence of keeping the dot separate from the causal past is that, if the dot in $X$ is contained in the causal past of $Y$, it means that $Y$ was generated causally after $X$, thus $Y$ also contains the causal past of $X$. This means that there is no need for the comparison of the vv component and the order can be computed as an $O(1)$ operation (assuming access to a map data structure in effectively constant time), simply as:

$$((i,n),u) < ((j,m),v) \iff n \leq v[i].$$

## 5 Dotted Version Vector Sets

Dotted Version Vectors (dvv), as presented in the previous section, allow an accurate representation of causality using server-based ids. Still, a dvv is kept for each concurrent version: $\{(dvv_1, v_1), (dvv_2, v_2), \ldots\}$. We can go further in exploring the fact that operations will mostly handle sets of dvv, and not single instances.

We propose now that the set of $(dvv, version)$ for a given key in a replica node is represented by a single instance of a container data type, a *Dotted Version Vector Set* (dvvs), which describes causality for the whole set. dvvs factorizes out common knowledge for the set of dvv described, and keeps only the strictly relevant information in a single data structure. This results in not only a very succinct representation, but also in reduced time complexity of operations: the concurrent values will be indexed and ordered in the data structure, and traversal will be efficient.

### 5.1 From a Set of Clocks to a Clock for Sets

To obtain a logical clock for a set of versions, we will explore the fact that at each node, the set of dvv as a whole can be represented with a compact vv. Formally this invariant means that, for any set of dvv $S$, for each node id $i$, *all* dots for $i$ in $S$ form a contiguous range up to some dot. Note that we can only assume to have this invariant, if we follow some protocol rules enforced by our framework, described in detail in section 6.3.

Assuming this invariant, we obtain a logical clock for a set of $(dvv, version)$ by performing a two-step transformation of the sets of versions. In the *first step*, we compute a single vv for the whole set — the *top vector* — by the pointwise maximum of the dots and vv in the dvv's; additionally, for each dvv in the set, we discard the vv component. As an example, the following set:

$$\{(((r,4),\{(r,3),(s,5)\}),v_1),(((r,5),\{(r,2),(s,3)\}),v_2),(((s,7),\{(r,2),(s,6)\}),v_3)\},$$

generates the top vector $\{(r,5),(s,7)\}$ and is transformed to a set of *(dot, version)*:

$$\{((r,4),v_1),((r,5),v_2),((s,7),v_3)\}.$$

This first transformation has incurred in a loss of knowledge: the specific causal past of each version. This knowledge is not, however, needed for our purposes. The insight is that, to know whether to discard or not a pair (dot,version) $(d,v)$ from some set when comparing with another set of versions $S$, we do not need to know exactly *which* version in $S$ dominates $d$, but only that *some* version does; if version $v$ is not present in $S$, but its dot $d$ is included in the causal information of the whole $S$ (which is now represented by the top vector), then we know that $v$ was obsolete and can be removed.

In the *second step*, we use the knowledge that all dots for each server id, form a contiguous sequence up to the corresponding top vector entry. Therefore, we can associate a list of versions (siblings) to each entry in the top vector, where each dot is implicitly derived by the corresponding version position in the list. In our example, the whole set is then simply described as:

$$\{(r,5,[v_2,v_1]),(s,7,[v_3])\},$$

where the head of each list corresponds to the more recently generated version at the corresponding node. The first version has the dot corresponding to the maximum of the top vector for that entry, the second version has the maximum minus one, and so on.

### 5.2 Definition

A dvvs is a set of triples $(i,n,l)$, each containing a server id, an integer, and a list of concurrent versions. It describes a set of versions and their dots, implicitly given by the position in the list. It also describes only the knowledge about the collective causal history, as given by the vv derived from the pairs $(i,n)$.

## 6 Using dvv and dvvs in Distributed Key-Value Stores

In this section we show how to use logical clocks — in particular dvv and dvvs— in modern distributed key-value stores, to accurate and efficiently track causality among writes in each key. Our solution consists in a general workflow that a database must use to serve get and put requests. Towards this, we define a kernel of operations over logical clocks, on top of which the workflow is defined. We then instantiate these operations over the logical clocks that we propose, first dvv and then dvvs.

We support both get and put operations, performing possibly several steps, as sketched in Figure 2. Lets first define our kernel operations.

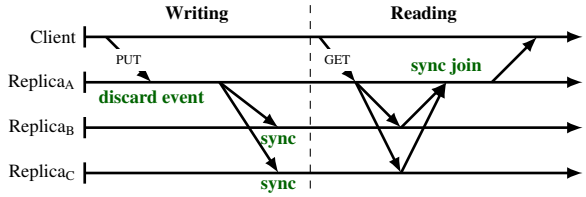



Fig. 2: Generic execution paths for operations get and put.

*Function* sync The function sync takes two sets of clocks, each describing a set of siblings, and returns the set of clocks for the siblings that remain after removing obsolete ones. It can have a general definition only in terms of the partial order on clocks, regardless of their actual representation: Equation 1.

*Function* join The join function takes a set of clocks and returns a single clock that describes the collective causal past of all siblings in the set received. An actual implementation of join is any function that corresponds to performing the union of all the events (dots) in the ch corresponding to the set, i.e., that satisfies Equation 2.

*Function* discard The discard function takes a set of clocks $S$ (representing siblings) and a clock $C$ (representing the context), and discards from $S$ all siblings that are obsolete because they are included in the context $C$. Similar to sync, discard has a simple general definition only in terms of the partial order on clocks: Equation 3.

*Function* event The event function takes a set of clocks $S$ (representing siblings) and a clock $C$ (representing the context) and a replica node identifier $r$; it returns a new clock to represent a new version, given by a new unique event (dot) generated at $r$, and having $C$ in the causal past. An implementation must respect Equation 4.

$$\mathsf{sync}(S_1,S_2) = \{x \in S_1 \mid \nexists y \in S_2 . x < y\} \cup \{x \in S_2 \mid \nexists y \in S_1 . x < y\}. \quad (1)$$

$$\mathsf{toch}(\mathsf{join}(S)) = \bigcup \{\mathsf{toch}(x) \mid x \in S\}. \quad (2)$$

$$\mathsf{discard}(S,C) = \{x \in S \mid x \not\leq C\}. \quad (3)$$

$$\mathsf{toch}(\mathsf{event}(C,S,r)) = \mathsf{toch}(C) \cup \{\mathsf{next}(C,S,r)\}, \quad (4)$$

where next denotes the next new unique event (dot) generated with $r$, which can be deterministically defined given $C$, $S$ and $r$.

### 6.1 Serving a get

Functions sync and join are used to define the get operation: when a server receives a get request, it may ask to a subset of replica nodes for their set of versions and clocks for that key, to be then "merged" by applying sync pairwise; however, the server can skip this phase if it deems it unnecessary for a successful response. Having the necessary

information ready, it is returned to the client both the values stripped from causality information and the context as a result of applying join to the clocks. sync can also be used at other times, such as anti-entropy synchronization between replica nodes.

### 6.2 Serving a put

When a put request is received, the server forwards the request to a replica node for the given key, unless the server is itself a replica node. A non-replica node for the key being written can coordinate a put request using $\mathsf{vv}_{\mathsf{client}}$ for example, because it can use the *client Id* to update the clock and then propagate the result to the replica nodes. However, clocks using *server Ids* like $\mathsf{vv}_{\mathsf{server}}$, dvv and dvvs need the coordinating node to generate an unique event in the clock, using its own *Id*. Not forwarding the request to replica node, would mean that non-replica nodes Ids would be added to clocks, making them linear with the total number of servers (e.g. hundreds) instead of only the replica nodes (e.g. three).

When a replica node $r$, containing the set of clocks $S_r$ for the given key, receives a put request, it starts by removing obsolete versions from $S_r$, using function discard, resulting in $S'_r$; it also generates a new clock $u$ for the new version with event; finally, $u$ is added to the set of non- obsolete versions $S'_r$, resulting in $S''_r$.

The server can then save $S''_r$ locally, propagate it to other replica nodes and successfully inform the client. The order of these three steps depends on the system's durability and replication parameters. Each replica node that receives $S''_r$, uses function sync to apply it against its own local versions.

For each key, the steps at the coordinator (discarding versions, generating a new one and adding it to the non-obsolete set of versions) must be performed atomically when serving a given put. This can be trivially obtained by local concurrency control, and does not prevent full concurrency between local operations on different keys. For operations over the same key, a replica can pipeline the steps of consecutive put for maximizing throughput (note that some steps already need to be serialized, such as writing versions to stable storage).

### 6.3 Maintaining Local Conciseness

As previously stated, both dvv and dvvs have an crucial invariant that servers must maintain, in order to preserve their correctness and conciseness:

**Invariant 1 (Local Clock Conciseness)** *Every key at any server has locally associated with it a set of version(s) and clock(s), that collectively can be logically represented by a contiguous set of causal events (e.g. represented as a vv).*

To enforce this invariant, we made two design choices: *(rule 1)* a server cannot respond to a get with a subset of the versions obtained locally and/or remotely, only the entire set should be sent; *(rule 2)* a coordinator cannot replicate the new version to remote nodes, without also sending all local concurrent versions (siblings).

Without the first rule, clients could update a key by reading and writing back a new value with a context containing arbitrary gaps in its causal history. Neither dvv nor dvvs

would be expressive enough to support this, since dvv only supports one gap (between the contiguous past and the dot) and dvvs does not support any.

Without the second rule, dvvs would clearly not work, since writes can create siblings, which cannot be expressed separately with this clock. It could work with dvv, however it would eventually result in some server not having a local concise representation for a key (e.g. the network lost a previous sibling), which in turn would make this server unable to respond to get without contacting other servers (see *rule 1*); it would degrade latency and in case of partitions, availability could also suffer.

### 6.4 Dotted Version Vectors

Functions sync and discard for dvv can be trivially implemented according to their general definitions, by using the partial order for dvv, already defined in Section 4.2.

We will make use of some two functions: function ids returns the set of identifiers of a pair from a vv, a dvv or a set of dvv; the maxdot function takes a dvv or set of dvv and a server id and returns the maximum sequence number of the events from that server:

$$\begin{aligned}
\mathsf{ids}((i,_)) &= \{i\},\\
\mathsf{ids}(((i,_),v)) &= \{i\} \cup \mathsf{ids}(v),\\
\mathsf{ids}(S) &= \bigcup_{s \in S} \mathsf{ids}(s).\\
\mathsf{maxdot}(r,((i,n),v)) &= \max(\{n \mid i = r\} \cup \{v[r]\}),\\
\mathsf{maxdot}(r,S) &= \max(\{0\} \cup \{\mathsf{maxdot}(r,s) \mid s \in S\}).
\end{aligned}$$

Function join returns a simple vv, which is enough to accurately express the causal information. Function event can be defined as simply generating a new dot and using the context $C$, which is already a vv, for the causal past.

$$\begin{aligned}
\mathsf{join}(S) &= \{(i, \mathsf{maxdot}(i,S)) \mid i \in \mathsf{ids}(S)\}.\\
\mathsf{event}(C,S,r) &= ((r, \max(\mathsf{maxdot}(r,S), C[r]) + 1), C).
\end{aligned}$$

### 6.5 Dotted Version Vector Sets

With dvvs, we need to make slight interface changes: functions now receive a single dvvs, instead of a set of clocks; and event now inserts the newly generated version directly in the dvvs.

For clarity and conciseness, we will assume $R$ to be the complete set of replica nodes ids, and any absent id $i$ in a dvvs, is promoted implicitly to the element $(i,0,[])$. We will make use of the functions: $\mathsf{first}(n,l)$, that returns the first $n$ elements of list $l$ (or the whole list if it has less than $n$ elements, or an empty list for non-positive $n$); $|l|$ for the number of elements in $l$, $[x \mid l]$ to append $x$ at the head of list $l$; and function merge:

$$\mathsf{merge}(n,l,n',l') = \begin{cases} \mathsf{first}(n - n' + |l'|, l), & \text{if } n \geq n',\\ \mathsf{first}(n' - n + |l|, l'), & \text{otherwise.} \end{cases}$$

| | | lww | ch | $\text{vv}_{\text{client}}$ | $\text{vv}_{\text{server}}$ | dvv | dvvs |
|---|---|---|---|---|---|---|---|
| **Space** | | $\tilde{O}(1)$ | $\tilde{O}(U)$ | $\tilde{O}(C\times V)$ | $\tilde{O}(R+V)$ | $\tilde{O}(R\times V)$ | $\tilde{O}(R+V)$ |
| **Time** | **event** | – | $\tilde{O}(1)$ | $\tilde{O}(1)$ | $\tilde{O}(1)$ | $\tilde{O}(V)$ | $\tilde{O}(R)$ |
| | **join** | – | $\tilde{O}(U\times V)$ | $\tilde{O}(C\times V)$ | $\tilde{O}(1)$ | $\tilde{O}(R\times V)$ | $\tilde{O}(R)$ |
| | **discard** | – | $\tilde{O}(U\times V)$ | $\tilde{O}(C\times V)$ | $\tilde{O}(R)$ | $\tilde{O}(V)$ | $\tilde{O}(R+V)$ |
| | **sync** | – | $\tilde{O}(U\times V^2)$ | $\tilde{O}(C\times V^2)$ | $\tilde{O}(R+V)$ | $\tilde{O}(V^2)$ | $\tilde{O}(R+V)$ |
| | **PUT** | $\tilde{O}(1)$ | $\tilde{O}(S_w\times U\times V^2)$ | $\tilde{O}(S_w\times C\times V^2)$ | $\tilde{O}(S_w\times(R+V))$ | $\tilde{O}(S_w\times V^2)$ | $\tilde{O}(S_w\times(R+V))$ |
| | **GET** | $\tilde{O}(1)$ | $\tilde{O}(S_r\times U\times V^2)$ | $\tilde{O}(S_r\times C\times V^2)$ | $\tilde{O}(S_r\times(R+V))$ | $\tilde{O}(R\times V+S_r\times V^2)$ | $\tilde{O}(S_r\times(R+V))$ |
| **Causally Correct** | | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ |

Table 2: Space and time complexity, for different causality tracking mechanisms. $U$: updates; $C$: writing clients; $R$: replica servers; $V$: (concurrent) versions; $S_r$ and $S_w$: number of servers involved in a GET and PUT, respectively.

Function discard takes a dvvs $S$ and a vv $C$, and discards values in $S$ obsoleted by $C$. Similarly, sync takes two dvvs and removes obsolete values. Function join simply returns the top vector, discarding the lists. Function event is now adapted to not only produce a new event, but also to insert the new value, explicitly passed as parameter, in the dvvs. It returns a new dvvs that contains the new value $v$, represented by a new event performed by $r$ and, therefore, appended at the head of the list for $r$. The context is only used to propagate causal information to the top vector, as we no longer keep it per version.

$$
\begin{aligned}
\mathsf{sync}(S,S') &= \{(r,\max(n,n'),\mathsf{merge}(n,l,n',l')) \mid r\in R,(r,n,l)\in S,(r,n',l')\in S'\},\\
\mathsf{join}(C) &= \{(r,n) \mid (r,n,l)\in C\},\\
\mathsf{discard}(S,C) &= \{(r,n,\mathsf{first}(n-C(r),l)) \mid (r,n,l)\in S\},\\
\mathsf{event}(C,S,r,v) &= \{(i,n+1,[v \mid l]) \mid (i,n,l)\in S \mid i=r\}\cup\\
&\quad\ \{(i,\max(n,C(i)),l) \mid (i,n,l)\in S \mid i\neq r\}
\end{aligned}
$$

## 7 Complexity and Evaluation

Table 2 shows space and time complexities of each causality tracking mechanism, for a single key. Lets consider $U$ the number of updates (writes), $C$ the number of writing clients, $R$ the number of replica servers, $V$ the number of concurrent versions (siblings) and $S_w$ and $S_r$ the number of replicas nodes involved in a put and get, respectively. Note that $U$ and $C$ are generally several orders of magnitude larger than $R$ and $V$. The complexity measures presented assume effectively constant time in accessing or updating maps and sets. We also assume ordered maps/sets that allow a pairwise traversal linear on the number of entries.

lww is constant both in time and space, since it does not track causality and ignores siblings. Space-wise, ch and $\text{vv}_{\text{client}}$ do not scale well, because they grow linearly with writes and clients, respectively. dvv scales well given that typically there is little con-

currency per key, but it still needs a dvv per sibling. From the considered clocks, dvvs and $\text{vv}_{\text{server}}$ have the best space complexity, but the latter is not causally accurate.

Following our framework (Section 6), the time complexities are[3]:

- put is $\tilde{O}(discard + event + S_w \times sync)$ and get is $\tilde{O}(join + S_r \times sync)$;
- *event* is effectively $\tilde{O}(1)$ for ch, $\text{vv}_{\text{client}}$ and $\text{vv}_{\text{server}}$; is linear with $V$ for dvv, because it has to check each value's clock; and is $\tilde{O}(R)$ for dvvs because it also merges the context to the local clock;
- *join* is constant for $\text{vv}_{\text{server}}$, since there is already only one clock; for ch, $\text{vv}_{\text{client}}$ and dvv it amounts to merging all their clocks into one; for dvvs, *join* simply extracts the top vector from the clock;
- *discard* is only linear with $V$ in dvv, because it can check the partial order of two clocks in constant time; as for ch and $\text{vv}_{\text{client}}$, they have to compare the context to every version's clock; $\text{vv}_{\text{server}}$ and dvvs always compare the context to a single clock, and in addition, dvvs has to traverse lists of versions;
- *sync* resembles *discard*, but instead of comparing a set of versions to a single context, it compares two sets of versions. Thus, ch, $\text{vv}_{\text{client}}$ and dvv complexities are similar to *discard*, but quadratic with $V$ instead of linear. Since $\text{vv}_{\text{server}}$ and dvvs have only one clock, the complexity of *sync* is linear on $V$.

### 7.1 Evaluation

We implemented both dvv and dvvs in Erlang, and integrated it with our fork of the NoSQL Riak datastore[4]. To evaluate the causality tracking accuracy of dvvs, and its ability to overcome the sibling explosion problem, we setup two equivalent 5 node Riak clusters, one using dvvs and the other $\text{vv}_{\text{server}}$.

We then ran a script[5] equivalent to the following: Peter (P) and Mary (M) write and read 50 times each to the same key, with read-write cycles interleaved (P writes then reads, next M writes then reads, in alternation). Figure 3 shows the growth in the number of siblings with every new write. The cluster with $\text{vv}_{\text{server}}$ had an explosion of false concurrency: 100 concurrent versions after 100 writes. Every time a client wrote with the its latest context, the clock in the server was already modified, thus generating and adding a sibling. However, with dvvs, although each write still conflicted with the latest write from the other client, it detected and removed siblings that were causally older (all the siblings present at the last read by that client). Thus, the cluster with dvvs had only two siblings after the same 100 writes: the last write from each client.

Finally, dvvs has already seen early adoption in the industry, namely in Riak, where it is the default logical clock mechanism in the latest release. As expected, it overcame the sibling explosion problem that was affecting real world Riak deployments, when multiple clients wrote on the same key.

---

[3] For simplicity of notation, we use the big $O$ variant: $\tilde{O}$, that ignores logarithmic factors in the size of integer counters and unique ids.

[4] `https://github.com/ricardobcl/Dotted-Version-Vectors`

[5] `https://gist.github.com/ricardobcl/4992839`

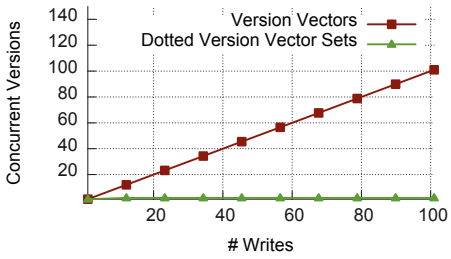



Fig. 3: Results of running two interleaved clients with 50 writes each.

## 8 Related Work

The role of causality in distributed systems was introduced by Lamport [11], establishing the foundation for the subsequent mechanisms and theory [11,14,20,19,2,4]. In Section 3 we discussed the problems of solutions commonly used in eventually consistent stores. In this section, we discuss other related work.

**Variability in the number of entities.** The basic vector based mechanisms can be generalized to deal with a variable number of nodes or replicas. The common strategy is to map identifiers to counters and handle dynamism in the set of identifiers. Additions depend on the generation of unique identifiers. Removals can require communication with several other servers [7], or to a single server [15,1]. While dvv and dvvs avoid identifier assignment to clients, these techniques could support changes in the set of servers.

**Exceptions on conflicts.** Some systems just detect the concurrent PUT operations from different clients and reject the update (e.g. version control systems such as CVS and subversion) or keep the updates but do not allow further accesses until the conflict is solved (e.g. original version of Coda [8]); in these cases, using version vectors (vv) with one entry per server is sufficient. However, these solutions sacrifice write availability which is a key "feature" of modern geo-replicated databases.

**Compacting the representation.** In general, using a format that is more compact than the set of independent entities that can register concurrency, leads to lossy representation of causality [4]. Plausible clocks [21] condense event counting from multiple replicas over the same vector entry, resulting in false concurrency. Several approaches for removing entries that are not necessary have been proposed, some being safe but requiring running consensus (e.g. Roam [18]), and others fast but unsafe (e.g. Dynamo [5]) potentially leading to causality errors.

**Extensions and added expressiveness.** In Depot [12], the vv associated with each update only includes the entries that have changed since the previous update in the same node. However, each node still needs to maintain vv that include entries for all clients and servers; in a similar scenario, the same approach could be used as a complement to our solution. Other systems explore the fact that they manage a large number of objects

to maintain less information for each object. WinFS [13] maintains a base vv for all objects is the file system, and for each object it maintains only the difference for the base in a concise vv. Cimbiosys [17] uses the same technique in a peer-to-peer system. These systems, as they maintain only one entry per server, cannot generate two vv for tagging concurrent updates submitted to the same server from different clients, as discussed in Section 3 with $\mathsf{vv}_{\mathsf{server}}$. WinFS includes a mechanism to deal with disrupted synchronizations that allow to encode non sequential causal histories by registering exceptions to the events registered in vv; e.g. $\{a_1, a_2, b_1, c_1, c_2, c_4, c_7\}$ could be represented by $\{(a,2),(b,1),(c,7)\}$ plus exceptions $\{c_3, c_5, c_6\}$. However, using dvv with its system workflow, at most a single update event that is outside the vv is needed, and thus a single *dot* per version is enough. dvvs goes further, by condensing all causal information in a vv, while being able to keep multiple implicit *dots*. This ensures just enough expressiveness to allow any number of concurrent clients and still avoids the size complexity of encoding a generic non sequential ch. Wang et. al. [22] have proposed a variant of vv with $O(1)$ comparison time (like dvv), but the vv entries must be kept ordered which prevents constant time for other operations. Furthermore, it also incurs in the problems associate with $\mathsf{vv}_{\mathsf{server}}$, which we solved with dvvs.

## 9 Closing Remarks

We have presented in detail Dotted Version Vectors, a novel solution for tracking causality among update events. The base idea is to add an extra isolated event over a causal history. This is sufficiently expressive to capture all causality established among concurrent versions (siblings), while keeping its size linear with the number of replicas.

We then proposed a more compact representation — Dotted Version Vector Sets — which allows for a single data structure to accurately represent causal information for a set of siblings. Its space and time complexity is only linear with the number of replicas plus siblings, better than all current mechanisms that accurately track causality.

Finally, we introduced a general workflow for requests to distributed data stores. It abstracts and factors the essential operations that are necessary for causality tracking mechanisms. We then implemented both our mechanisms using those kernel operations.

**Acknowledgements.** This research was partially supported by FCT/MCT projects PEst-OE/EEI/UI0527/2014 and PTDC/EEI-SCR/1837/2012; by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n$^{o}$ 609551, SyncFree project; by the ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-037281.

## References

1. Almeida, P.S., Baquero, C., Fonte, V.: Interval tree clocks. In: Proceedings of the 12th International Conference on Principles of Distributed Systems. pp. 259–274. OPODIS '08, Springer-Verlag, Berlin, Heidelberg (2008)

2. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM Trans. Comput. Syst. 5(1), 47–76 (Jan 1987)
3. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing. pp. 7–. PODC '00, ACM, New York, NY, USA (2000)
4. Charron-Bost, B.: Concerning the size of logical clocks in distributed systems. Information Processing Letters 39, 11–16 (1991)
5. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: Proceedings of twenty-first ACM SIGOPS SOSP. pp. 205–220. ACM (2007)
6. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. In: In ACM SIGACT News. p. 2002 (2002)
7. Golding, R.A.: A weak-consistency architecture for distributed information services. Computing Systems 5, 5–4 (1992)
8. Kistler, J.J., Satyanarayanan, M.: Disconnected operation in the Coda file system. In: Thirteenth ACM Symposium on Operating Systems Principles. vol. 25, pp. 213–225. Asilomar Conference Center, Pacific Grove, US (1991)
9. Klophaus, R.: Riak core: building distributed applications without shared state. In: ACM SIGPLAN Commercial Users of Functional Programming. pp. 14:1–14:1. CUFP '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1900160.1900176
10. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 35–40 (April 2010)
11. Lamport, L.: Time, clocks and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (Jul 1978)
12. Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., Walfish, M.: Depot: Cloud storage with minimal trust. In: OSDI 2010 (Oct 2010)
13. Malkhi, D., Terry, D.B.: Concise version vectors in winfs. In: Fraigniaud, P. (ed.) DISC. Lecture Notes in Computer Science, vol. 3724, pp. 339–353. Springer (2005)
14. Parker, D.S., Popek, G., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kiser, S., Kline, C.: Detection of mutual inconsistency in distributed systems. Transactions on Software Engineering 9(3), 240–246 (1983)
15. Petersen, K., Spreitzer, M.J., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible update propagation for weakly consistent replication. In: Sixteen ACM Symposium on Operating Systems Principles. Saint Malo, France (Oct 1997)
16. Preguiça, N., Baquero, C., Almeida, P.S., Fonte, V., Gonçalves, R.: Brief announcement: Efficient causality tracking in distributed storage systems with dotted version vectors. In: Proceedings of the 2012 ACM symposium on PODC. pp. 335–336. ACM (2012)
17. Ramasubramanian, V., Rodeheffer, T.L., Terry, D.B., Walraed-Sullivan, M., Wobber, T., Marshall, C.C., Vahdat, A.: Cimbiosys: a platform for content-based partial replication. In: Proceedings of the 6th USENIX symposium on NSDI. pp. 261–276. Berkeley, CA, USA (2009)
18. Ratner, D., Reiher, P.L., Popek, G.J.: Roam: A scalable replication system for mobility. MONET 9(5), 537–544 (2004)
19. Raynal, M., Singhal, M.: Logical time: Capturing causality in distributed systems. IEEE Computer 30, 49–56 (Feb 1996)
20. Schwarz, R., Mattern, F.: Detecting causal relationships in distributed computations: In search of the holy grail. Distributed Computing 3(7), 149–174 (1994)
21. Torres-Rojas, F.J., Ahamad, M.: Plausible clocks: constant size logical clocks for distributed systems. Distributed Computing 12(4), 179–196 (1999)
22. Wang, W., Amza, C.: On optimal concurrency control for optimistic replication. In: Proc. ICDCS. pp. 317–326 (2009)

## A.4 Paulo Sérgio Almeida, Ali Shoker, Carlos Baquero. Efficient State-based CRDTs by Decomposition. In Proc. PaPEC 14.

# Efficient State-based CRDTs by Decomposition

[Work in progress report]

Paulo Sérgio Almeida
HASLab, INESC TEC &
Universidade do Minho
Braga, Portugal
psa@di.uminho.pt

Ali Shoker
HASLab, INESC TEC &
Universidade do Minho
Braga, Portugal
shokerali@di.uminho.pt

Carlos Baquero
HASLab, INESC TEC &
Universidade do Minho
Braga, Portugal
cbm@di.uminho.pt

## ABSTRACT

Eventual consistency is a relaxed consistency model used in large-scale distributed systems that seek better availability when consistency can be delayed. CRDTs are distributed data types that make eventual consistency of a distributed object possible and non ad-hoc. Specifically, state-based CRDTs achieve this through shipping the entire replica state that is, eventually, merged to other replicas ensuring convergence. This imposes a large communication overhead when the replica size or the number of replicas gets larger. In this work, we introduce a decomposable version of state-based CRDTs, called *Delta State-based CRDTs* ($\delta$-CRDT). A $\delta$-CRDT is viewed as a join of multiple fine-grained CRDTs of the same type, called deltas ($\delta$). The deltas are produced by applying $\delta$-mutators, on a replica state, which are modified versions of the original CRDT mutators. This makes it possible to ship small deltas (or batches) instead of shipping the entire state. The challenges are to make the join of deltas equivalent to the join of the entire object in classical state-based CRDTs, and to find a way to derive the $\delta$-mutators. We address this challenge in this work, and we explore the minimal requirements that a communication algorithm must offer according to the guarantees provided by the underlying messaging middleware.

## 1. INTRODUCTION

Eventual consistency [12] has recently got the attention of both research community and industry [5, 1, 11, 6] due to the enormous growth of large-scale distributed systems, and at the same time, the need to ensure availability for users despite outages and partitioning. In fact, the practical experience of leading industry shows that daily server outages and network partitioning in large-scale distributed systems is a norm rather than an exception. Given that partitioning cannot be avoided, the limitations explained by the CAP theorem [7] requires some sacrifice in consistency (by delaying it) for the sake of higher availability only when immediate consistency is not a requirement; a *like/unlike* action in social networks is a concrete example. CRDTs [9, 10] are formal methods to make eventual convergence of distributed datatypes generic and easy. Although they are currently being used in industry [5], CRDTs are still not mature, and many enhancements are still needed on both levels: design and performance. This work addresses some design issues to achieve better performance.

*Conflict-free Replicated Data Types (CRDTs)* [9, 10] are formalized data types designed to ensure the convergence of different replicas of a distributed CRDT object. Traditionally, two types of CRDTs were defined: *operation-based* and *state-based*. In operation-based CRDTs [8, 10], once an operation is invoked on a replica, a *prepare* phase returns a payload message that comprises a derived operation of the original one and possibly other meta-data. The message is sent to other replicas that apply this message via the *effect* phase which, in its turn, makes use of the received meta-data to maintain the causal order of operations. To achieve eventual consistency, this approach assumes a middleware that provides causal delivery of operations and membership management. In state-based CRDTs [2, 10], an invoked operation is applied on the local object state that derives a new state. Occasionally, the new state is sent to other replicas that incorporate the received state with the local state though a *merge*. A merge is designed in such a way to achieve convergence from any two states, being commutative, associative, and idempotent. In mathematical terms, merge is defined as a *join*: a least upper bound over a join-semilattice [2, 10].

State-based CRDTs are preferred to operation-based when causal delivery is not guaranteed by the messaging middlware. However, the state-based approach has two main weaknesses: (1) shipping updates becomes expensive when the distributed object gets large, and (2) a sort of garbage collection is often required. Some recent works [4, 3] addressed the problem of garbage collection; however, to the best of our knowledge, no profound research dealt with reducing the overhead of data shipping as we propose in this work.

The communication overhead of shipping the entire state in state-based CRDTs often grows with the replica state size and the number of replicas. For instance, the state size of a *counter* CRDT increases with the number of replicas, whereas, in a *grow-only Set*, the state size grows as more operations are invoked. Other CRDTs, like the *OR-Set*, impose a similar overhead also (due to shipping the set and its tombstones); although garbage collection can reduce this overhead once used, this is only possible when the invoked


*PaPEC'14* April 13-16 2014, Amsterdam, Netherlands

http://dx.doi.org/10.1145/2596631.259663.

operations that cancel each others are close in time; e.g., an *add* followed by *remove* of the same element must occur before the shipping time is due. These scalability issues limit the use of state-based CRDTs to data-types with conservative payloads (e.g. few megabytes in Dynamo [6]). Recently, calls in the industry started to show up asking for the possibility to consider larger state sizes (e.g., in RIAK [5]).

In this work, we rethink the way that state-based CRDTs should be designed, having in mind the useless redundant shipping of the entire state. Our idea is to decompose a state-based CRDT in such a way to only ship recent updates rather than the whole state. To achieve this goal, we introduce *Delta State-based CRDTs* ($\delta$-CRDT). A $\delta$-CRDT is roughly a union of multiple fine-grained $\delta$-CRDTs of the same type, which is built through multiple invocations of *$\delta$-mutators* which are then merged. A $\delta$-mutator is a derived version of a CRDT mutator that produces a $\delta$ which only comprises the new changes that the original mutator induced on the state. This way, we can retain the deltas, and join these deltas together into batches, to be shipped later instead of shipping the entire object. Once these batches of deltas arrive at the receiving replica, they are joined with its local state.

The challenge in our approach is to make sure that decomposing a CRDT into deltas and then joining them into another replica state (after shipping) produces the same effect as if the entire state had been shipped and merged. In particular, the challenge involves how to derive the $\delta$-mutators from the original CRDT mutators.

In this work, we discuss these challenges, and explore possible solutions. In addition, we discuss the benefits of this approach given the guarantees provided by the messaging middleware, and we propose the basic requirements a distributed algorithm must satisfy towards this goal.

*Acknowledgments.*

Project Norte-01-0124-FEDER-000058 is co-financed by the North Portugal Regional Operational Program (ON.2 -- O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF). Funding from the European Union Seventh Framework Program (FP7/2007-2013) with grant agreement 609551, SyncFree project.

## 2. REFERENCES

[1] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32, Mar. 2013.

[2] C. Baquero and F. Moura. Using structural characteristics for autonomous operation. *Operating Systems Review*, 33(4):90–96, 1999.

[3] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. Rapp. Rech. RR-8083, Institut National de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, Oct. 2012.

[4] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In S. Jagannathan and P. Sewell, editors, *POPL*, pages 271–284. ACM, 2014.

[5] S. Cribbs and R. Brown. Data structures in Riak. In *Riak Conference (RICON)*, San Francisco, CA, USA, oct 2012.

[6] 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 *Symp. on Op. Sys. Principles (SOSP)*, volume 41 of *Operating Systems Review*, pages 205–220, Stevenson, Washington, USA, Oct. 2007. Assoc. for Computing Machinery.

[7] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[8] M. Letia, N. Preguiça, and M. Shapiro. CRDTs: Consistency without concurrency control. Rapp. Rech. RR-6956, Institut National de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, June 2009.

[9] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapp. Rech. 7506, Institut National de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, Jan. 2011.

[10] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, Oct. 2011. Springer-Verlag.

[11] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Symp. on Op. Sys. Principles (SOSP)*, pages 172–182, Copper Mountain, CO, USA, Dec. 1995. ACM SIGOPS, ACM Press.

[12] W. Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, Oct. 2008.

### A.5 Nuno Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, Marc Shapiro. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. In Proc. W-PSDS 14 (SRDS 14).

# SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine

Nuno Preguiça[†]

joint work with:

Marek Zawirski[*], Annette Bieniusa[‡], Sérgio Duarte[†], Valter Balegas[†], Carlos Baquero[§], Marc Shapiro[*]
[*]*Inria & UPMC-LIP6*
[†]*NOVA-LINCS/CITI/U. Nova de Lisboa*
[‡]*U. Kaiserslautern*
[§]*INESC Tec & U. Minho*

***Abstract*—Client-side logic and storage are increasingly used in web and mobile applications to improve response time and availability. Current approaches tend to be ad-hoc and poorly integrated with the server-side logic. We present a principled approach to integrate client- and server-side storage. We support both mergeable and strongly consistent transactions that target either client or server replicas and provide access to causally-consistent snapshots efficiently. In the presence of infrastructure faults, a client-assisted failover solution allows client execution to resume immediately and seamlessly access consistent snapshots without waiting. We implement this approach in SwiftCloud, the first transactional system to bring geo-replication all the way to the client machine.**

**Example applications show that our programming model is useful across a range of application areas. Our experimental evaluation shows that SwiftCloud provides better fault tolerance and at the same time can improve both latency and throughput by up to an order of magnitude, compared to classical geo-replication techniques.**

## I. Introduction

Cloud computing infrastructures support a wide range of services, from social networks and games to collaborative spaces and online shops. Cloud platforms improve availability and latency by geo-replicating data in several data centers (DCs) across the world [1], [2], [3], [4], [5], [6]. Nevertheless, the closest DC is often still too far away for an optimal user experience. For instance, round-trip times to the closest Facebook DC range from several tens to several hundreds of milliseconds, and several round trips per operation are often necessary [7]. Furthermore, mobile clients may be completely disconnected from any DC for an unpredictable period of minutes, hours or days.

Caching data at client machines can improve latency and availability for many applications, and even allow for a temporary disconnection. While increasingly used, this approach often leads to ad-hoc implementations that integrate poorly with server-side storage and tend to degrade data consistency guarantees. To address this issue, we present SwiftCloud, the first system to bring geo-replication all the way to the client machine and to propose a principled approach to access data replicas at client machines and cloud servers.

Although extending geo-replication to the client machine seems natural, it raises two big challenges. The first one is to provide programming guarantees for applications running on client machines, at a reasonable cost at scale and under churn. Recent DC-centric storage systems [5], [6], [4] provide transactions, and combine support for causal consistency with mergeable objects [8]. Extending these guarantees to the clients is problematic for a number of reasons: standard approaches to support causality in client nodes require vector clocks entries proportional to the number of replicas; seamless access to client and server replicas require careful maintenance of object versions; fast execution in the client requires asynchronous commit. We developed protocols that efficiently address these issues despite failures, by combining a set of novel techniques.

Client-side execution is not always beneficial. For instance, computations that access a lot of data, such as search or recommendations, or running strongly consistent transactions, is best done in the DC. SwiftCloud supports server-side execution, without breaking the guarantees of client-side in-cache execution.

The second challenge is to maintain these guarantees when the client-DC connection breaks. Upon reconnection, possibly to a different DC, the outcome of the client's in-flight transactions is unknown, and state of the DC might miss the causal dependencies of the client. Previous cloud storage systems either retract consistency guarantees in similar cases [5], [6], [9], or avoid the issue by waiting for writes to finish at a quorum of servers [4], which incurs high latency and may affect availability.

SwiftCloud provides a novel client-assisted failover protocol that preserves causality cheaply. The insight is that, in addition to its own updates, a client observes a causally-consistent view of stable (i.e., stored at multiple servers) updates from other users. This approach ensures that a client always observes his previous updates and that it can safely

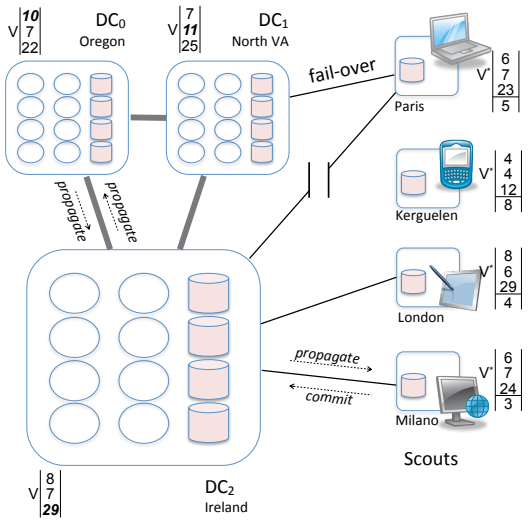



Figure 1. SwiftCloud system structure.

reconnect to other DC, as it can replay its own updates and other observed updates being stable, are already in other DCs.

Experimental evaluation shows that under sufficient access locality, SwiftCloud enjoys order-of-magnitude improvements in both response time and throughput over the classical approach. This is because, not only reads (if they hit in the cache), but also updates commit at the client side without delay; servers only need to store and forward updates asynchronously. Although our fault tolerance approach delays propagation, the proportion of stale reads remains under 1%.

In the remaining of this paper, we briefly overview the key solutions developed in the context of SwiftCloud [10].

## II. System overview

SwiftCloud is a data storage systems for cloud platforms that spans both client nodes and data center servers (DCs), as illustrated in Figure 1. The core of the system consists of a set of *data centers (DCs)* that replicate every object. At the periphery, applications running in *client nodes* access the system through a local module called *scout*. A scout caches a subset of the objects. If the appropriate objects are in cache, responsiveness is improved and a client node supports disconnected operation.

SwiftCloud provides a straightforward transactional key-object API. An application executes transactions by interactively executing sequences of reads and updates, concluded by either a commit or rollback.

Our transactional model, Transactional Causal+ Consistency, offers the following guarantees: every transaction reads a causally consistent snapshot; updates of a transaction are atomic (all-or-nothing) and isolated (no concurrent transaction observes an intermediate state); and concurrently committed updates do not conflict.

This transactional model allows different clients to observe the same set of concurrent updates applied in different orders, which poses a risk of yielding different operation outcomes on different replicas or at different times. We address this problem by disallowing non-commutative (order-dependent) concurrent updates. Practically, we enforce this property with two different types of transactions: *Mergeable* and *Classical, non-mergeable* transaction, akin to the model of Walter [4] or Red-Blue [9]:

Mergeable transactions commute with each other and with non-mergeable transactions, which allows to execute them immediately in the cache, commit asynchronously in the background, and remain available in failure scenarios. Mergeable transaction are either read-only transaction or update transactions that modify Conflict-free Replicated Data Types (CRDT)[8], [11]. CRDTs encapsulate the logic to merge concurrent updates deterministically, independently of the order of execution of updates.

Classical transactions provide the traditional strongly-consistent transaction model, in which non-commuting concurrent updates conflict (as determined by an oracle on pairs of updates) and cannot both commit. These transactions execute completely in the data centers.

## III. Algorithms for Mergeable transactions

We now present the key ideas of the algorithms for executing *mergeable transactions* in a failure-free case. In the next section we address the problems posed by failures.

An application issues a *mergeable transaction* iteratively through the scout. Reads are served from the local scout; on a cache miss, the scout fetches the data from the DC it is connected to. Updates execute in a local copy. When a mergeable transaction terminates, it is locally committed and updates are applied to the scout cache. Updates are also propagated to a data center (DC) for being globally committed. The DC eventually propagates the effects of transactions to other DCs and other scouts scouts as needed.

*Atomicity and Isolation:* For supporting atomicity and isolation, a transaction reads from a database snapshot. Each transaction is assigned a DC timestamp by the DC that received it from the client. Each DC maintains a vector clock with the summary of all transactions that have been executed in that DC, which is updated whenever a transaction completes its execution in that DC. This vector has as $n$ entries, with $n$ the number of DCs. Each scout maintains a vector clock with the version of the objects in the local cache.

When a transaction starts in the client, the current version of the cache is selected as the transaction snapshot. If the transaction accesses an object that is not present in the cache, the appropriate version is fetched from the DC - to this end, DCs maintain recent versions of each object.

*Read your writes:* When a transaction commits in the client, the local cache is updated. The following transactions access a snapshot that includes these locally committed transactions. To this end, each transaction executed in the client is assigned a scout timestamp. The vector that summarizes the transactions reflected in the local cache has $n+1$ entries, with the additional entry being used to summarize locally submitted transactions. This approach guarantees that a client always reads a state that reflects his previous transactions.

*Causality:* The system ensures the invariant that every node (DC or scout) maintains a causally-consistent set of object versions. To this end, a transaction only executes in a DC after its dependencies are satisfied - the dependencies of a transaction, summarized in the transaction snapshot, are propagated both from the client to the initial DC and from one DC to other DCs.

When a scout caches some object, the DC it is connected to becomes responsible of notifying it with updates to those cached objects. SwiftCloud includes a notification subsystem that guarantees that updates from a committed transaction are propagated atomically and respecting causality. As a result, the cache in the scout is also causally consistent.

## IV. Fault-tolerant session and durability

We discuss now how SwiftCloud handles network, DC and client faults, focusing on client-side mergeable transactions. When a scout loses communication with its current DC, due to network or DC failure, the scout may need to switch over to a different DC. The latter's state is likely to be different, and it might have not processed some transactions observed or indirectly observed (via transitive causality) by the scout. In this case, ensuring that the clients' execution satisfies the consistency model and the system remains live is more complex. As we will see, this also creates problems with durability and exactly-once execution.

### A. Causal dependency issue

When a scout switches to a different DC, the state of the new DC may be unsafe, because some of the scout's causal dependencies are missing. Some geo-replication systems avoid creating dangling causal dependencies by making synchronous writes to multiple data centers, at the cost of high update latency [1]. Others remain asynchronous or rely on a single DC, but after failover clients are either blocked or they violate causal consistency [5], [6], [9]. The former systems trade consistency for latency, the latter trade latency for consistency or availability.

An alternative approach would be to store the dependencies on the scout. However, since causal dependencies are transitive, this might include a large part of the causal history and a substantial part of the database.

Our approach is to make scouts co-responsible for the recovery of missing session causal dependencies at the new DC. Since, as explained earlier, a scout cannot keep track of all transitive dependencies, we restrict the set of dependencies. We define a transaction to be $K$*-durable* [12] at a DC, if it is known to be durable in at least $K$ DCs, where $K$ is a configurable threshold. Our protocols let a scout observe only the union of: *(i)* its own updates, in order to ensure the "read-your-writes" session guarantee [13], and *(ii)* the $K$-durable updates made by other scouts, to ensure other session guarantees, hence causal consistency. In other words, the client depends only on updates that the scout itself can send to the new DC, or on ones that are likely to be found in a new DC. When failing over to a new DC, the scout helps out by checking whether the new DC has received its recent updates, and if not, by repeating the commit protocol with the new DC.

SwiftCloud prefers to serve a slightly old but $K$-durable version, instead of a more recent but more risky version. Instead of the consistency and availability vs. latency trade-off of previous systems, SwiftCloud trades availability for staleness.

### B. Durability and exactly-once execution issue

A scout sends each transaction to its DC to be globally-committed. The DC assigns a DC timestamp to the transaction, and eventually transmits it to every replica. If the scout does not receive an acknowledgment, it must retry the global-commit, either with the same or with a different DC. However, the outcome of the initial global-commit remains unknown. If it happens that the global commit succeeded with the first DC, and the second DC assigns a second DC timestamp, the danger is that the transaction's effects could be applied twice under the two identities.

For some data types, this is not a problem, because their updates are idempotent, for instance `put(key,value)` in a last-writer-wins map. For other mergeable data types, however, this is not true: think of executing `increment(10)` on a counter. Systems restricted to idempotent updates can be much simpler [6], but in order to support general mergeable objects with rich merge semantics, SwiftCloud must ensure exactly-once execution.

Our approach separates the concerns of tracking causality and of uniqueness, following by the insight of [14]. Recall that a transaction has both a scout timestamp and a DC timestamp. The scout timestamp identifies a transaction uniquely, whereas the DC timestamp is used when a summary of a set of transactions is needed. Whenever a scout globally-commits a transaction at a DC, and the DC does not have a record of this transaction already, the DC assigns it a new DC timestamp. This approach makes the system available, but may assign several DC timestamp aliases for the same transaction. All alias DC timestamps are equivalent in the sense that, if updates of $T'$ depend on $T$, then $T'$ comes after $T$ in the causality order, no matter what DC timestamp $T'$ uses to refer to $T$.

When a DC processes a commit record for an already-known transaction with a different DC timestamp, it adds the alias DC timestamp to its commit record on durable storage.

To provide a reliable test whether a transaction is already known, each DC maintains durably a map of the last scout timestamp received from each scout. Thanks to causal consistency, this value is monotonically non-decreasing. Thus, a DC knows that a transaction being received for global-commit from a scout has already been processed if the recorded value for that scout is greater or equal to the scout timestamp of the received transaction.

## V. Final remarks

We overview the design of SwiftCloud, the first system that brings geo-replication to the client machine, providing a principled approach for using client and data center replicas. SwiftCloud allows applications to run transactions in the client machine, for common operations that access a limited set of objects, or in the DC, for transactions that require strong consistency or accessing a large number of objects. Our evaluation of the system [10] shows that the latency and throughput benefit can be huge when compared with traditional cloud deployments for scenarios that exhibit good locality, a property verified in real workloads [15].

SwiftCloud also proposes a novel client-assisted failover mechanism that trades latency by a small increase in staleness. Our evaluation shows that our approach helps reducing latency while increasing stale reads by less than 1%.

## Acknowledgment

This research was supported in part by EU FP7 project SyncFreee (grant agreement no 609551), ANR project ConcoRDanT (ANR-10-BLAN 0208), by the Google Europe Fellowship in Distributed Computing awarded to Marek Zawirski, and by Portuguese FCT/MCT projects PEst-OE/EEI/UI0527/2014 and PTDC/EEI-SCR/1837/2012 and Phd scholarship awarded to Valter Balegas (SFRH/BD/87540/2012).

## References

[1] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *OSDI*. Hollywood, CA, USA: Usenix, Oct. 2012, pp. 251–264.

[2] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *SIGMOD*, Scottsdale, AZ, USA, May 2012, pp. 1–12.

[3] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR*, Asilomar, CA, USA, Jan. 2011, pp. 229–240.

[4] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *SOSP*. Cascais, Portugal: Assoc. for Comp. Mach., Oct. 2011, pp. 385–400.

[5] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS," in *SOSP*. Cascais, Portugal: Assoc. for Comp. Mach., Oct. 2011, pp. 401–416.

[6] ——, "Stronger semantics for low-latency geo-replicated storage," in *NSDI*, Lombard, IL, USA, Apr. 2013, pp. 313–328.

[7] M. P. Wittie, V. Pejovic, L. Deek, K. C. Almeroth, and B. Y. Zhao, "Exploiting locality of interest in online social networks." Philadelphia, PA, USA: Assoc. for Comp. Mach., Dec. 2010, pp. 25:1–25:12.

[8] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *SSS*, ser. LNCS, X. Défago, F. Petit, and V. Villain, Eds., vol. 6976. Grenoble, France: Springer Verlag, Oct. 2011, pp. 386–400.

[9] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *OSDI*, Hollywood, CA, USA, Oct. 2012, pp. 265–278.

[10] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. Preguiça, "SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine," INRIA, Rapp. Rech. RR-8347, Aug. 2013.

[11] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Convergent and commutative replicated data types," *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, no. 104, pp. 67–88, Jun. 2011.

[12] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," *TOCS*, vol. 29, no. 4, pp. 12:1–12:38, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2063509.2063512

[13] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch, "Session guarantees for weakly consistent replicated data," in *PDIS*, Austin, Texas, USA, Sep. 1994, pp. 140–149.

[14] P. S. Almeida, C. Baquero, R. Gonçalves, N. M. Preguiça, and V. Fonte, "Scalable and accurate causality tracking for eventually consistent stores," in *Proc. 14th Int. Conf. Distributed Applications and Interoperable Systems (LNCS 8460)*, 2014, pp. 67–81.

[15] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida, "Characterizing user behavior in online social networks," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, ser. IMC '09, 2009, pp. 49–62.

## A.6 Valter Balegas, Mahsa Najafzadeh, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Marc Shapiro, Nuno Preguiça. The Case for Fast and Invariant-Preserving Geo-Replication. In Proc. W-PSDS 14 (SRDS 14)

# The Case for Fast and Invariant-Preserving Geo-Replication

Valter Balegas, Sérgio Duarte,
Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça

CITI/FCT/Universidade Nova de Lisboa

Marc Shapiro
Mahsa Najafzadeh

INRIA / LIP6

***Abstract*—Cloud storage systems showcase a range of consistency models, from weak to strong consistency. Weakly consistent systems enable better performance, but cannot maintain strong application invariants, which strong consistency trivially supports. This paper takes the position that it is possible to both achieve fast operation and maintain application invariants. To that end, we propose the novel abstraction of *invariant-preserving CRDTs*, which are replicated objects that provide invariant-safe automatic merging of concurrent updates. The key technique behind the implementation of these CRDTs is to move replica coordination outside the critical path of operations execution, to enable low normal case latency while retaining the coordination necessary to enforce invariants. In this paper we present ongoing work, where we show different *invariant-preserving CRDTs* designs and evaluate the latency of operations using a counter that never goes negative.**

## I. Introduction

To improve the user experience in services that operate on a global scale, from social networks and multi-player online games to e-commerce applications, the infrastructure that supports those services often resorts to geo-replication [9], [7], [17], [18], [16], [26], [8], i.e., maintains copies of application data and logic in multiple data centers scattered across the globe, providing improved scalability and lower latency. But not always the advantages of geo-replication are exploited by worldwide services, because, when services need to maintain invariants over the data, they have to synchronize with remote data centers in order to execute some operations, which negatively impacts operations' latency. In a geo-replicated scenario, latency may amount to hundreds of milliseconds.

The impact of high latency in the user's experience is well known [22], [11] and has motivated the academia [7], [1], [9] and industry [13], [5], [25] to use weaker consistency models with low-latency operations at the trade of data consistency.

When running applications under such weaker consistency models, applications in different data centers execute operations concurrently over the same set of data leading to temporary divergence between replicas and potentially unintuitive and undesirable user-perceived semantics.

This research is supported in part by European FP7 project 609 551 SyncFree (2013–2016), Fundação para a Ciência e Tecnologia SFRH/BD/87540/2012 and PEst-OE/EEI/UI0527/2014.

Good user-perceived semantics are trivially provided by systems that use strong-consistency models, namely those that serialize all updates, and therefore preclude that two operations execute without seeing the effects of one another [8], [16]. Not all operations require strong guarantees to execute, and some systems provide both strong and weak consistency models for different operations [26], [16].

In this paper, we claim that it is possible to achieve the best of both worlds, i.e., that fast geo-replicated operations can coexist with strong application invariants without impairing the latency of operations. To this end, we propose novel abstract data types called *invariant-preserving CRDTs*. These are replicated objects that, like conventional CRDTs [23], automatically merge concurrent updates, but, in addition, they can maintain application invariants. Furthermore, we show how these CRDTs can be efficiently implemented in a geo-replicated setting by moving the replica coordination that is needed for enforcing invariants outside the critical path of operation execution.

In this paper, we discuss cloud consistency models(§II); present the concept of InvCRDT (§III), abstract data types that offer invariant-safe operations; discuss the implementation of these ADTs (§IV); discuss invariants that span multiple objects (§V-B ); Present the practical benefits of InvCRDTs (§VI) and, finally, we briefly review related work (§VII) and present our conclusions (§VIII).

## II. Decomposing Consistency Requirements

Recent cloud systems [8], [12], [26], [16] have adopted strong consistency models to avoid concurrency anomalies. These models rely on a serializable (or even linearizable) execution order for operations to provide the illusion that a single replica exists. They do so at the expense of lower availability on failures and increased latency for operations - a direct consequence of the CAP theorem [6], which states that there is a trade-off between availability and consistency in systems prone to partitioning.

We argue that enforcing strong consistency is not mandatory for fulfilling the requirements of most applications. We use the example of an e-commerce site to motivate such statement, by identifying three central requirements of this application.

First, users of the application must not observe a past version of any given data item after observing a more recent

one – e.g., after adding some item to her shopping cart, the user does not want to observe a shopping cart where the item is not present. A way to achieve this without per-operation replica synchronization is to support causal consistency, as found in several cloud systems [17], [18].

Second, when concurrent updates exist, data replicas cannot be allowed to diverge permanently. This requires some form of automatic reconciliation that deals with concurrent updates identically in all sites, leading to a consistency model that has been recently coined as causal+ consistency [17] or fork-join-causal consistency [19]. For example, after two users add two different items to a shopping cart, both items should be in the reconciled version of the shopping cart.

Finally, the e-commerce application has crucial integrity constraints that must be preserved despite concurrent updates – e.g., the stock of a product should be greater or equal to zero, thus avoiding that the store sells more items than what it has in stock.

In current systems, invariants as the stock example are usually preserved by running such application (or operations that can break the invariant [26], [16]) under a strong consistency model. Instead, we propose to run such applications under a consistency model that provides the following properties: causal consistency; automatic reconciliation; and invariant preservation. We call this consistency model *causal+invariants* consistency.

It seems straightforward that enforcing invariants usually requires some form of coordination among nodes of the system – e.g., to ensure that a product stock does not go negative, it is necessary that replicas coordinate so that the number of successful sales do not exceed the number of items in stock. However, unlike the solution adopted by strong consistency, in many situations this coordination can be executed outside of the critical execution path of operations. In the previous example, the rights to use the available stock can be split among the replicas, allowing a purchase to proceed without further coordination provided replica where the operation is submitted has enough rights [20], [21].

## III. The case for invariant-preserving CRDTs

Conflict-free replicated data-types (CRDT [23]) are data types that leverage the commutativity of operations to automatically merge concurrent updates in a sensible way. Several CRDT specifications have been proposed for some of the most commonly used data types, such as lists, sets, maps and counters, allowing rapid integration in existing applications. CRDTs provide convergence by design and, when combined with a replication protocol that delivers operations in causal order, they trivially provide causal+ consistency [17], [26].

### A. The concept of InvCRDTs

In this paper, we propose the concept of invariant-preserving CRDT (InvCRDT), a conflict-free data type that maintains a given invariant even in the presence of concurrent operations – the *BoundedCounter* [under submission] implements a counter that cannot be negative.

Some CRDTs already maintain invariants internally by repairing the state – e.g., in the graph CRDT [23], when one user adds an arc between two nodes and other user concurrently removes one of the nodes, the graph CRDT does not show the arc. However, unlike these solutions, InvCRDTs maintain invariants by explicitly disallowing the execution of operations that would lead to the violation of an invariant. By having immediate feedback that an operation cannot be executed, an application can give that feedback to the users – e.g., in an e-commerce application, an order will fail if some product has no stock available, since the operation of decrementing the stock of the product, aborts when implemented with a *BoundedCounter*.

For achieving this functionality, a replica of an InvCRDT includes both the state of the object and information about the rights the replica holds. These rights allow the execution of operations that potentially break invariants without coordination while guaranteeing that the invariants will not be broken. The union of the rights granted to each of the existing replicas guarantees that the invariants defined will be preserved despite any concurrent operation. The set of initial rights will depend on the initial value of the object. For example, in a *BoundedCounter* with initial value 10 and two replicas, each replica has the rights to increment the counter at any moment and the rights to execute five decrement operations.

The rights each replica holds are consumed or extended when an operation is submitted locally – e.g., in the previous example, a decrement will consume the rights to decrement by one, and an increment will increase the local rights to decrement by one. If enough rights exist locally, it is assured that the execution of the operation in other replicas will not break the defined invariant. If not enough rights exist locally, the execution of the method aborts (in our Java-based implementation, by throwing an exception) and it has no side-effects in any replica. Optionally, when not enough rights exist locally, the system may try to obtain additional rights by transferring them from some other replica(s). In this case, the method execution blocks until the necessary communication with other replicas is done. In this case, the overhead of operation execution will tend to be similar to the overhead of providing strong consistency.

This model for InvCRDTs is general enough to allow different implementations, as discussed in the next section. An important property on InvCRDTs that must be highlighted is that InvCRDTs do not eliminate the need of coordination among replicas: they only allow the coordination to be executed outside the critical path of execution of an application request, through the exchange of rights. Next we discuss the common invariants in applications and how they can be addressed using InvCRDTs.

### B. Using InvCRDTs in applications

There are many examples in the literature of applications with integrity constraints that are good candidates for using InvCRDTs.

Li et. al. [16] report that two invariants must be considered

in TPC-W. First, the stock of a product must be non-negative. This can be addressed by the *BoundedCounter* previously mentioned. Second, the system must guarantee that unique identifiers are generated in a number of situations where new data items are created. To address this requirement, the space of possible identifiers could be partitioned among the replicas (for example, using the replica identifier as a suffix). InvCRDT versions of containers (e.g., set, maps) can be created, where each replica maintains rights for assigning new unique identifiers to elements added to the object. The authors also report that similar invariants must be preserved for Rubis.

Cooper et. al. [7] discuss several applications, among them, one that maintains an hierarchical namespace. Although they do not explicitly discuss invariants, it is clear to see that there are two important invariants that should be preserved: no two objects have the same name; and no cycles exist in the presence of renames. For the first invariant, we use rights that preclude two replicas from generating identical names – a replica must acquire rights to generate identifiers with some prefix). Maintaining the second invariant is more complex and requires obtaining the exclusive right to modify the path of directories from the first common ancestor of the original and destination names for supporting renames (section V-A). This can be implemented by extending our graph CRDT [23] with these rights.

Other applications have invariants on the cardinality of containers (e.g., a meeting must have at least K members), on the properties of elements present in containers (e.g., at least one element of each gender), etc. These invariants can also be preserved by having InvCRDT versions of those containers.

More recently, Bailis et al.[2] have studied OLTP systems and summarized typical invariants that show up in applications. Some of them are instantiations of the ones described above, while other require more elaborate mechanisms as discussed in section IV.

## IV. Supporting InvCRDTs

We assume a typical cloud computing environment composed by clients and data centres. Data centres run application servers for handling client requests and a replicated storage system to persist application data. The effects of client requests are persisted by modifying the data stored in the system, represented as InvCRDTs. Finally, a replication protocol that delivers operations in causal order is used to achieve our proposed causal+invariants consistency model.

One possible design would consist of managing the rights associated with InvCRDTs through a centralized server. In this case, each replica would obtain these rights by contacting such central entity (as in [20], [21]). We propose an alternative approach, where the rights associated with an InvCRDT are maintained in a decentralized way, completely inside the InvCRDT.

Our generic solution consists in modelling application data as resources and by keeping the rights of each replica as a vector of (*replicaId* $\Rightarrow$ *value*) entries for each resource type in all InvCRDT replicas. Each operation is modelled as consuming or creating resources. For example, in the *BoundedCounter*, a single resource type exists, and a resource corresponds to one unit in the counter; an increment creates one resource; a decrement consumes one resource. In an InvCRDT that needs to generate unique identifiers, the reserved resources are a subset of the identifiers (e.g., a chunk of consecutive identifiers or a subset of identifiers ended in the reserved suffix).

Operations that modify the rights vector – consume (subtract), extend (add), transfer (atomically subtract from one entry and add to another) – are commutative. Thus, they can be supported in a convergent data-type style, where operations only need to execute in causal order in the different replicas [1]. Consume and extend operations affect the rights of the replica where the operations are initiated. The transfer operation must be initiated in the replica from which the rights are to be transferred from.

This execution model guarantees that in any given replica $i$, the rights that are known to exist for replica $i$ are a conservative view when considering all operations that can have been executed. The reason for this is that all operations that decrement the rights of a given replica, consume and transfer, are submitted locally, while a remote transfer that is not yet known may increase the local rights. This property guarantees the correctness of our approach.

## V. Discussion

### A. InvCRDT data-types

In section III-A we briefly presented the design of the *BoundedCounter* CRDT. We are studying other data-types that can share the same philosophy of maintaining the state of the object as well as the rights to execute operations. The *BoundedCounter* is a fairly simple example to understand, however the same idea can be applied to other data-types.

We give the intuition for a few other data-types and what invariants they can preserve:

**Tree** Each node in a tree has a unique parent node. This invariant can be broken by concurrently moving a node and putting it under two different nodes. A possible solution to prevent the violation of this invariant consists in associating to each node a right to modify its subtree. When a replica acquires rights over a node it automatically acquires the rights to modify any descendent of that node. The replica that holds rights over a portion of the tree may give permission to another replica to modify some subtree, losing the permission itself to modify any node under that subtree. This strategy enforces a replica executing a rename operation to hold rights over the origin and destination names, which prevents any concurrent operation from creating a cycle.

**Graph** To implement a graph that is always consistent, i.e.,an edge always connect to an existing node, without using the automatic convergence mechanism of the graph CRDT, we associate rights to each node, which have to be acquired in order to remove it, or connect an edge. When a new node

[1] As with CRDTs, it is possible to design an equivalent solution based on state propagation.

is created it has rights associated to the replica that created the node. Preventing cycles in a graph is more complex than in trees and we have not addressed that so far.

**Map** Two concurrent puts in a map may end up associating different element to the same key. To prevent this situation, we can associate rights to ranges of keys which have to be acquired in order to execute a put operation. This guarantees that two different replicas cannot execute a conflicting put operation. The strategy of key domain partitioning can be used to provide unique identifiers.

We aim to provide a library of InvCRDTs that support most of the invariants that are common in applications, however we are still investigating an easy way to provide them to programmers.

### B. Multi-object invariants

InvCRDTs enforce invariants in a single object. However, application invariants can often span multiple objects – e.g., a user can only checkout a shopping cart if all items are in stock.

Supporting these invariants requires enforcing some type of operation grouping. Recently, weakly consistent storage systems have provided support for some form of transactions [18], [26]. We could build on this type of support to maintain invariants over multiple objects – e.g., in the previous example, a transaction would succeed only if the data centre where it was submitted holds rights to consume all the necessary stock units of each item.

Some other invariants establish relations that must be maintained among multiple objects – e.g., in a courseware application, a student can only be part of a course student group if he or she is enrolled in the course. This invariant can be maintained either by repairing (e.g., if the students enrolment in the course is cancelled, the membership in the course student group is also cancelled) or avoiding the invariant violation. If it seems clear that these types of invariants can be preserved by restricting concurrent operations in multiple objects (e.g., avoiding the concurrent creation of a group and removal of a student involved). However, we are still studying the best approach to represent them as InvCRDTs. Additionally, it is also not obvious what is the best way to define invariant repairing solutions in these cases. Addressing these issues is also left as future work.

## VI. Preliminary evaluation

We conducted some preliminary experiments to evaluate the latency of InvCRDT operations. We made an Erlang prototype that extends Riak [5] with support for InvCRDTs. Basically the prototype is a middleware component that is stacked between the application server and the storage system. The middleware's main function is to exchange rights between replicas, so that when operation are executed rights are available locally and the operation succeed without contacting any remote data center.

We implemented a micro-benchmark that simulates the manipulation of items' stock on purchases in an e-commerce application: Decrement operations are submitted to a counter in multiple data centers and the value of the counter cannot go negative, regardless the operations propagation frequency between data centers.

We implemented the BoundedCounter and the policies to exchange rights between replicas. These exchange of rights occur in the background and try to prevent rights from being exhausted locally. When a replica runs out of rights and executes a decrement, it tries to fetch the rights from a remote data center, which potentially has high latency.

We compare the solution using InvCRDT (BCounter) against an weak consistency (WeakC) solution that uses a convergent counter and a solution that provides strong consistency (StrongC) by executing all operations on the same data center. Riak natively support these features: the convergent counter is an implementation of the PN-Counter CRDT [23] and the strong consistency solution uses a consensus algorithm based on the Paxos algorithm [14].

We did not implemented true causality in our prototype, instead the middleware provides key-linearizability, which is sufficient because in the experiments all operations are executed in a single-object. Key-linearizibility is necessary to avoid concurrent requests to use the same rights within the same data center.

### A. Experimental Setup

Our experiments comprised 3 Amazon EC2 data centers distributed across the globe. We installed a Riak data store in each EC2 availability zone (US-East, US-West, EU). Each Riak cluster is composed by three m1.large machines, with 2 vCPUs, producing 4 ECU[2] units of computational power, and with 7.5GB of memory available. We use Riak 2.0.0pre5 version.

*a) Operations latency:* Figure 1 details these results by showing the CDF of latency for operation execution. As expected, the results show that for *StrongC*, remote clients experience high latency for operation execution. This latency is close to the RTT latency between the client and the DC holding the data. For *StrongC*, each step in the line consists mostly of operations issued in different DCs.

Both *BCounter* and *WeakC* experience very low latency. In a counter-intuitive way, the latency of *BCounter* is sometimes even better than the latency of *WeakC*. This happens because our middleware caches the counters, requiring only one access to Riak for processing an update operation when compared with two accesses in *WeakC* (one for reading the value of the counter and another for updating the value if it is positive).

Figure 2 furthers details the behaviour of our middleware, by presenting the latency of operations over time. The results show that most operations take low latency, with a few peak of high latency when a replica runs out of rights and needs to ask for additional rights from other data centers. The number of peaks is small because most of the time the pro-active

[2]1 ECU corresponds is a relative metric used to compare instance types in the AWS platform

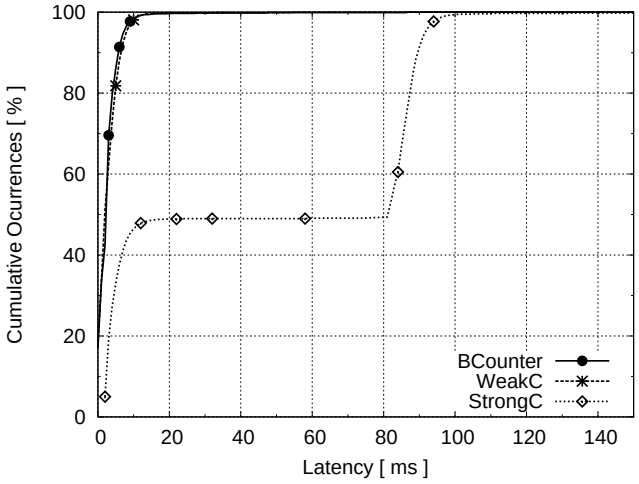



Fig. 1. CDF of latency with a single counter.

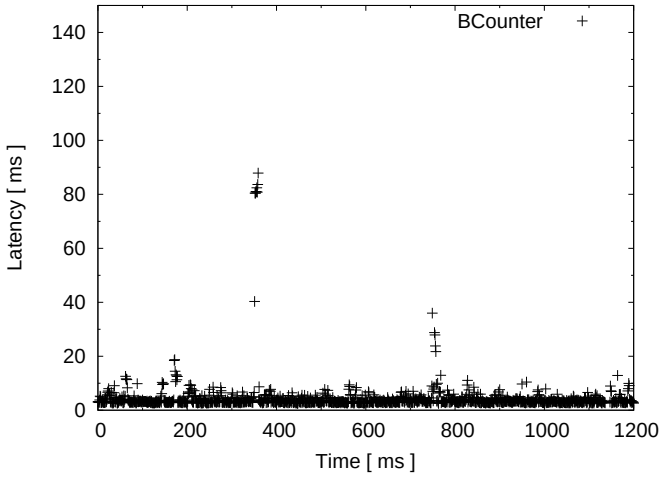



Fig. 2. Latency measured over time.

mechanism for exchanging rights is able to provision a replica with enough rights before all rights are used.

## VII. Related Work

A large number of cloud storage systems supporting geo-replication have been developed in recent years. Some of these systems [9], [17], [18], [1], [13] provide variants of eventual consistency, where operations return immediately after being executed in a single data center. This approach has the lowest latency possible for end-users, but since the guarantees they provide are so weak, a handful of other systems try to provide better semantics for the user and still avoid cross data center coordination, such as those that provide causal consistency [17], [1], [10], [3]. We target to provide similar ordering guarantees of messages but improve over these systems by maintaining applications invariants that require some form of coordination.

Systems that provide strong consistency[8] incur in coordination overhead that increases latency of operations. Some systems tried to combine the benefits of weak and strong consistency models by supporting both models. In Walter [26] and Gemini [16], transactions that can execute under weak consistency run fast, without needing to coordinate with other data centers.

More recently, Sieve [15] automates the decision between executing some operation in weak or strong consistency. Bailis et al. [2] have also studied when it is possible to avoid coordination in database systems, while maintaining application invariants. Our work is complimentary, by providing solutions that can be used when coordination cannot be avoided.

Escrow transactions [20] have been proposed as a mechanism for enforcing numeric invariants while allowing concurrent execution of transactions. The key idea is to enforce local invariants in each transaction that guarantee that the global invariant is not broken. The original escrow model is agnostic to the underlying storage system and in practice was mainly used to support disconnected operations [24], [21] in mobile computing environments, using a centralized solution to handle reservations.

The demarcation protocol [4] is an alternative that has been proposed to maintain invariants in distributed databases and recently applied to optimize strong-consistency protocols [12]. Although the underlying protocol are similar to escrow-based solutions, the demarcation protocol focus on maintaining invariants across different objects.

We aim to combine these different mechanism to provide an unified framework that programmers can use to improve the consistency of applications given the same assumptions as in weak consistency systems.

## VIII. Conclusion

This paper presents a weak consistency model, extended with invariant preservation for supporting geo-replicated services. For supporting the causal+invariants consistency model, we propose a novel abstraction called invariant-preserving CRDTs, which are replicated objects that provide both sensible merge of concurrent updates and invariant preservation in the presence of concurrent updates. We outline the design of InvCRDTs that can be deployed on top of systems providing causal+ consistency only. Our approach provides low latency for most operations by moving the necessary coordination among nodes outside of the critical path of operation execution.

The next steps in our work are to build a library of CRDTs that programmers can use to maintain application invariants as well as providing a programming model that ease the use of these data-types in applications. One possibility would be to categorize invariants and have specific data-types to preserve each of them with low-latency. We are also still studying how to maintain invariants that span multiple objects and what guarantees does the replication model must provide in order

to maintain them.

The preliminary evaluation showed that it is possible to maintain invariants under weak consistency by relying on a proactive rights exchange mechanism to transfer rights between replicas.

## REFERENCES


[1] S. Almeida, J. a. Leitão, and L. Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[2] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination-avoiding database systems. *CoRR*, abs/1402.2237, 2014.

[3] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[4] D. Barbará-Millá and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, July 1994.

[5] Basho. Riak. http://basho.com/riak/, 2014. Accessed Jan/2014.

[6] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[7] 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. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[9] 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 *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[10] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[11] T. Hoff. Latency is everywhere and it costs you sales - how to crush it. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it, 2009.

[12] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[13] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[14] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[15] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, June 2014. USENIX Association.

[16] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[17] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[18] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[19] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. Comput. Syst.*, 29(4):12:1–12:38, Dec. 2011.

[20] P. E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, Dec. 1986.

[21] N. Preguiça, J. L. Martins, M. Cunha, and H. Domingos. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 43–56, New York, NY, USA, 2003. ACM.

[22] E. Schurman and J. Brutlag. Performance related changes and their user impact. Presented at velocity web performance and operations conference, 2009.

[23] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] L. Shrira, H. Tian, and D. Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 42–61, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[25] S. Sivasubramanian. Amazon dynamodb: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM.

[26] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

# B Papers under submission and technical reports

## B.1 Carlos Baquero, Paulo Sérgio Almeia, Alcino Cunha, Carla Ferreira. Composition of state-based CRDTs. Internal techinal report

# Composition of state-based CRDTs

Carlos Baquero, Paulo Srgio Almeida, Alcino Cunha, Carla Ferreira

October 21, 2014

## 1 Introduction

State-based CRDTs are rooted in mathematical structures called join-semilattices (ore simply lattices, in this context). These order structures ensure that the replicated states of the defined data types evolve and increase in a partial order in a sufficiently defined way, so as to ensure that all concurrent evolutions can be merged deterministically. In order to build, or understand the building principles, of state-based CRDTs it is necessary to understand the basic building blocks of the support lattices and how lattices can be composed.

## 2 From Sets to Lattices

In this context the most basic structure to define is a set of distinct values. An example is the set of vowels that can defined by extension as $\mathsf{vowels} \doteq \{a, e, i, o, u\}$. Elements in a set have no specific order and they only need to be distinguishable, i.e. by defining $=$.

Having a set we can define partial orders by defining a poset over a support set and an order relation $\sqsubseteq$. This relation can be any binary relation that is reflexive, transitive and anti-symmetric. Given elements $o, p, q$ in a set.

- (reflexive) $o \sqsubseteq o$
- (transitive) $o \sqsubseteq p \wedge p \sqsubseteq q \Rightarrow o \sqsubseteq q$
- (anti-symmetric) $o \sqsubseteq p \wedge p \sqsubseteq o \Rightarrow o = p$

Since sets already define $=$ it is possible to create posets transitively by enumerating the element pairs related by $\sqsubset$. As an example, we can build a poset with a total order on the set of vowels by defining $\langle \mathsf{vowels}, \{(a, e), (e, i), (i, o), (o, u)\}\rangle$ In this example we ordered all elements and thus created a chain, with $a \sqsubset e \sqsubset i \sqsubset o \sqsubset u$, i.e. given any two elements $o, p$ either $o \sqsubseteq p$ or $p \sqsubseteq o$.

If some elements were left unordered we could have concurrent elements.

- (concurrent) $o \parallel p \iff \neg(o \sqsubseteq p \vee p \sqsubseteq o)$

In the extreme case we could have left all elements unordered and defined a poset that depicted an *antichain*, where any two elements are always concurrent. E.g. $\langle \mathsf{vowels}, \{\}\rangle$. Having a poset we also have the properties of a set.

$$\frac{A : \mathsf{poset}}{A : \mathsf{set}}$$

For a given poset to be a join-semilattice there must be a *least-upper-bound* for any subset of the support set. Given a pair of elements $o, p$, their least-upper-bound can be derived by the result of a binary join operator, by $o \sqcup p$. Since the binary join is commutative and associative it can be iterated over the elements of any subset to derive the least-upper-bound of the subset. Some properties of join are listed bellow.

- (idempotent) $o \sqcup o = o$
- (commutative) $o \sqcup p = p \sqcup o$
- (associative) $o \sqcup (p \sqcup q) = (o \sqcup p) \sqcup q$

And properties of least-upper-bounds.

- (upper-bound) $o \sqsubseteq o \sqcup p$
- (least-upper-bound) $o \sqsubseteq q \wedge o \sqsubseteq q \Rightarrow o \sqsubseteq p \sqcup q$

A general example of a poset with a join is obtained from any set by selecting the order to be set inclusion and the join to be set union. In our running example this would be the lattice defined by $\langle \mathsf{vowels}, \subseteq, \cup\rangle$. Another simple lattice can be obtained by taking the maximum in a total order (or dually the minimum), for naturals we can derive $\mathsf{maxint} \doteq \langle \mathbb{N}, \leq_{\mathbb{N}}, \mathbf{max}\rangle$.

Having a lattice we also have the properties of a poset.

$$\frac{A : \mathsf{lattice}}{A : \mathsf{poset}}$$

A chain (a special case of a poset) always derives a lattice.

$$\frac{A : \mathsf{chain}}{A : \mathsf{lattice}}$$

Notice that although some specific partial orders always derive lattices, as is the case for *chains*, in general we can have partial orders that are not lattices. An example is the prefix ordering on bit strings that can produce concurrent elements, $010 \parallel 100$, and is not a lattice.

We will see in latter sections that in some cases it is useful to have a special element in the lattice that is the bottom element $\bot$. Some properties are.

- (bottom) $\bot \sqsubseteq o$
- (identity) $\bot \sqcup o = o$

The lattice formed by set inclusion has the empty set as bottom, $\langle \mathsf{vowels}, \subseteq, \cup, \emptyset \rangle$. Not all lattices have a "natural" bottom, but it is always possible to add an extra element as bottom to an existing lattice. We will address this construction when talking about lattice composition by linear sums. As expected, lattices with bottom also have the lattice properties.

$$\frac{A : \mathsf{lattice}_\bot}{A : \mathsf{lattice}}$$

### 2.1 Primitive Lattices

We now introduce a small set of lattices, that will be later useful to construct more complex structures by composition.

**Singleton** A single element, $\bot$.

$$\frac{}{\mathbb{1} : \mathsf{lattice}_\bot}$$

$$\bot \sqsubseteq \bot \qquad \bot \sqcup \bot = \bot$$

**Boolean** Two elements $\mathbb{B} = \{\mathsf{False}, \mathsf{True}\}$ in a chain, join is logical $\vee$.

$$\frac{}{\mathbb{B} : \mathsf{lattice}_\bot}$$

$$\mathsf{False} \sqsubseteq \mathsf{True} \qquad x \sqcup y = x \vee y \qquad \bot = \mathsf{False}$$

**Naturals** Natural numbers. We include the 0, thus $\mathbb{N} = \{0, 1, \ldots\}$.

$$\frac{}{\mathbb{N} : \mathsf{lattice}_\bot}$$

$$n \sqsubseteq m = n \leq m \qquad n \sqcup m = \max(n, m) \qquad \bot = 0$$

## 3 Inflations make CRDTs

State-based CRDTs can be specified by selecting a given lattice to model the state, and choosing an initial value in the lattice, usually the $\bot$. Mutation operations can only change the state by *inflations* and do not return values. Query operations evaluate an arbitrary function on the state and return a value.

An inflation is an endo-function on the lattice type that picks a value $x$ among the set of valid lattice states $a$ and produces a new value state such that:

- (inflation) $x \sqsubseteq f(x)$

Inflations can be further classified as non-strict and strict inflations, where a strict inflation is such that:

- (strict inflation) $x \sqsubset f(x)$

We can now classify inflations.

$$\frac{\forall x \in a \cdot x \sqsubseteq f(x)}{f : A \xrightarrow{\sqsubseteq} A}$$

$$\frac{\forall x \in a \cdot x \sqsubset f(x)}{f : A \xrightarrow{\sqsubset} A}$$

$$\frac{f : A \xrightarrow{\sqsubset} A}{f : A \xrightarrow{\sqsubseteq} A}$$

A state that is only updated as a result of an inflation over its current value, is immutable under joins with copies of past states.

Notice that an inflation is not the same as a monotonic function, $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. Example, the function $f(x) = \frac{x}{2}$ on positive reals is monotonic and is not an inflation.

## 3.1 Primitive Inflations

Building on the primitive lattices introduced above we can build some inflations.

$$\mathsf{id}(x) = x \qquad \frac{}{\mathsf{id} : A \xrightarrow{\sqsubseteq} A}$$

$$\underline{\mathsf{True}}(x) = \mathsf{True} \qquad \frac{}{\underline{\mathsf{True}} : \mathbb{B} \xrightarrow{\sqsubseteq} \mathbb{B}}$$

$$\mathsf{succ}(x) = x + 1 \qquad \frac{}{\mathsf{succ} : \mathbb{N} \xrightarrow{\sqsubset} \mathbb{N}}$$

## 3.2 Sequential Composition

Inflations can be composed sequentially. As long as there is at least one strict inflation in the composition, we are sure to also have a strict composition.

$$(f \bullet g)(x) = f(g(x))$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : A \xrightarrow{\sqsubseteq} A}{f \bullet g : A \xrightarrow{\sqsubseteq} A}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : A \xrightarrow{\sqsubset} A}{f \bullet g : A \xrightarrow{\sqsubset} A} \qquad \frac{f : A \xrightarrow{\sqsubset} A \quad g : A \xrightarrow{\sqsubseteq} A}{f \bullet g : A \xrightarrow{\sqsubset} A}$$

# 4 Lattice Compositions

Since we are interested in creating lattices we consider a few composition techniques that are known to derive lattices. While in some cases they build from other lattices, in others they can derive lattices from simpler structures.

## 4.1 Product

The product $\times$, or pair construction, derives a lattice formed by pairs of other lattices. It can be applied recursively and derive a composition from a sequence of lattices, where operations are applied in point-wise order.

$$\frac{A : \mathsf{lattice} \qquad B : \mathsf{lattice}}{A \times B : \mathsf{lattice}}$$

$$(x_1, y_1) \sqsubseteq (x_2, y_2) = x_1 \sqsubseteq x_2 \wedge y_1 \sqsubseteq y_2$$

$$(x_1, y_1) \sqcup (x_2, y_2) = (x_1 \sqcup x_2, y_1 \sqcup y_2)$$

The construction also extends to $\mathsf{lattice}_\perp$ when all sources are also $\mathsf{lattice}_\perp$.

$$\frac{A : \mathsf{lattice}_\perp \qquad B : \mathsf{lattice}_\perp}{A \times B : \mathsf{lattice}_\perp}$$

$$\perp = (\perp, \perp)$$

As an example, the underlying lattice structure of a version vector among three replica nodes is composable by $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ with $\perp = (0, 0, 0)$.

Bellow are the properties of inflations over products. A strict inflation on one of the components leads to an overall strict inflation.

$$(f \times g)(x, y) = (f(x), g(y))$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \times g : A \times B \xrightarrow{\sqsubseteq} A \times B}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubset} B}{f \times g : A \times B \xrightarrow{\sqsubset} A \times B} \qquad \frac{f : A \xrightarrow{\sqsubset} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \times g : A \times B \xrightarrow{\sqsubset} A \times B}$$

## 4.2 Lexicographic Product

The $\boxtimes$ construct builds a lexicographic order from its source lattices. Components to the left are more significant and unless they are equal they filter out further comparisons to the right side.

$$\frac{A : \mathsf{lattice} \qquad B : \mathsf{lattice}_\perp}{A \boxtimes B : \mathsf{lattice}} \qquad \frac{A : \mathsf{lattice}_\perp \qquad B : \mathsf{lattice}_\perp}{A \boxtimes B : \mathsf{lattice}_\perp}$$

$$(x_1, y_1) \sqsubseteq (x_2, y_2) = x_1 \sqsubseteq x_2 \vee (x_1 = x_2 \wedge y_1 \sqsubseteq y_2)$$

$$(x_1, y_1) \sqcup (x_2, y_2) = \begin{cases} (x_1, y_1) & \textbf{if } x_2 \sqsubset x_1 \\ (x_2, y_2) & \textbf{if } x_1 \sqsubset x_2 \\ (x_1, y_1 \sqcup y_2) & \textbf{if } x_1 = x_2 \\ (x_1 \sqcup x_2, \bot) & \textbf{otherwise} \end{cases}$$

$$\bot = (\bot, \bot)$$

In the join definition we can observe that the $\bot$ value is used only when the left components can have concurrent values. If the left component is a chain, often the case in practical uses, then the right one can be a simple lattice (without $\bot$) and the fourth clause of the join definition is not used.

$$\frac{A : \mathsf{chain} \qquad B : \mathsf{lattice}}{A \boxtimes B : \mathsf{lattice}}$$

And, if the right component is also a chain the composition is a chain.

$$\frac{A : \mathsf{chain} \qquad B : \mathsf{chain}}{A \boxtimes B : \mathsf{chain}}$$

Properties of inflations.

$$(f \boxtimes g)(x, y) = (f(x), g(y))$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \boxtimes g : A \boxtimes B \xrightarrow{\sqsubseteq} A \boxtimes B}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubset} B}{f \boxtimes g : A \boxtimes B \xrightarrow{\sqsubset} A \boxtimes B} \qquad \frac{f : A \xrightarrow{\sqsubset} A \quad g : B \longrightarrow B}{f \boxtimes g : A \boxtimes B \xrightarrow{\sqsubset} A \boxtimes B}$$

Notice that if we apply a strict inflation to the left component, then the right can be transformed by any (endo-)function even if non inflationary. In practice this allows resetting the right component after strictly inflating the left.

## 4.3 Linear Sum

The next composition, linear sum $\oplus$, picks two lattices, left and right, and creates a new lattice where any element from the left lattice is always lower that any element in the right lattice. In the resulting set the elements are tagged with a label that identifies from which source lattice they came form. i.e. Left $a$ means that element $a$ came from the left lattice and is now named Left $a$. Tagging also ensures that the sets supporting each lattice could have had elements in common.

$$\frac{A : \mathsf{lattice} \qquad B : \mathsf{lattice}}{A \oplus B : \mathsf{lattice}} \qquad \frac{A : \mathsf{lattice}_\bot \qquad B : \mathsf{lattice}}{A \oplus B : \mathsf{lattice}_\bot}$$

$$\begin{array}{ll} \mathsf{Left}\ x \sqsubseteq \mathsf{Left}\ y = x \sqsubseteq y & \mathsf{Left}\ x \sqcup \mathsf{Left}\ y = \mathsf{Left}\ (x \sqcup y) \\ \mathsf{Right}\ x \sqsubseteq \mathsf{Right}\ y = x \sqsubseteq y & \mathsf{Right}\ x \sqcup \mathsf{Right}\ y = \mathsf{Right}\ (x \sqcup y) \\ \mathsf{Left}\ x \sqsubseteq \mathsf{Right}\ y = \mathsf{True} & \mathsf{Left}\ x \sqcup \mathsf{Right}\ y = \mathsf{Right}\ y \\ \mathsf{Right}\ x \sqsubseteq \mathsf{Left}\ y = \mathsf{False} & \mathsf{Right}\ x \sqcup \mathsf{Left}\ y = \mathsf{Right}\ x \end{array}$$

$$\bot = \mathsf{Left}\ \bot$$

A possible use of this construction is to add a $\bot$ to a lattice that didn't had one. For instance $\mathbb{1} \oplus \mathbb{R}$ can add a special element, e.g. nil, that is ordered as lower than any real number. The same construction can also be used to add a top element $\top$ to a lattice, that can act as a tombstone that stops lattice evolution. Notice that for any state $x$, $x \sqcup \top = \top$.

Properties of inflations.

$$\begin{array}{l} (f \oplus g)(\mathsf{Left}\ x) = \mathsf{Left}\ f(x) \\ (f \oplus g)(\mathsf{Right}\ x) = \mathsf{Right}\ g(x) \end{array}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \oplus g : A \oplus B \xrightarrow{\sqsubseteq} A \oplus B}$$

$$\frac{f : A \xrightarrow{\sqsubset} A \quad g : B \xrightarrow{\sqsubset} B}{f \oplus g : A \oplus B \xrightarrow{\sqsubset} A \oplus B}$$

## 4.4 Function and Map

A total function $\to$ is obtained by combining a set with a lattice. This construction does keywise comparison and joins.

$$\frac{A : \mathsf{set} \qquad B : \mathsf{lattice}}{A \to B : \mathsf{lattice}} \qquad \frac{A : \mathsf{set} \qquad B : \mathsf{lattice}_\bot}{A \to B : \mathsf{lattice}_\bot}$$

$$f \sqsubseteq g = \forall x \in A \cdot f(x) \sqsubseteq g(x) \qquad (f \sqcup g)(x) = f(x) \sqcup g(x)$$

$$\bot(x) = \bot$$

A map $\hookrightarrow$ can be obtained from a function by assigning a bottom to keys that are not present in a given map, and then using the function definitions. The linear sum construction is used to assign a distinguished bottom to any lattice $V$ in the co-domain.

$$K \hookrightarrow V \cong K \to \mathbb{1} \oplus V$$

$$\frac{K : \mathsf{set} \qquad V : \mathsf{lattice}}{K \hookrightarrow V : \mathsf{lattice}_\bot}$$

For example, we can define a map of $\mathsf{vowels}$ keys to integer counters $\mathsf{vowels} \hookrightarrow \mathbb{N}$ by using a total function $\mathsf{vowels} \to \mathbb{1} \oplus \mathbb{N}$. Where the map state $\{a \mapsto 3, i \mapsto 5\}$ would be the same as the function state $\{a \mapsto 3, e \mapsto \bot, i \mapsto 5, o \mapsto \bot, u \mapsto \bot\}$.

We define some inflations over maps. The first inflation applies an inflation to all values in the co-domain and thus inflates the map composition.

$$\mathsf{map}(f)(m) = \{(k, f(v)) \mid (k, v) \in m\}$$

$$\frac{f : V \xrightarrow{\sqsubseteq} V}{\mathsf{map}(f) : (K \hookrightarrow V) \xrightarrow{\sqsubseteq} (K \hookrightarrow V)}$$

The second inflation transforms the value on a given key, and if the key is missing applies it to $\bot$. This allows a strict inflation in the co-domain lattice to imply a strict inflation in the composition.

$$\mathsf{apply}_k(f)(m) = \begin{cases} m\{k \mapsto f(v)\} & \textbf{if } (k, v) \in m \\ m\{k \mapsto f(\bot)\} & \textbf{otherwise} \end{cases}$$

$$\frac{f : V \xrightarrow{\sqsubseteq} V}{\mathsf{apply}_k(f) : (K \hookrightarrow V) \xrightarrow{\sqsubseteq} (K \hookrightarrow V)}$$

$$\frac{f : V \xrightarrow{\sqsubset} V}{\mathsf{apply}_k(f) : (K \hookrightarrow V) \xrightarrow{\sqsubset} (K \hookrightarrow V)}$$

## 4.5 Sets and Multisets

Given any $\mathsf{set}$ $A$ it is possible to derive a $\mathsf{lattice}_\bot$ by using the set of all possible subsets, the *powerset* $\mathcal{P}(A)$.

For example, $\mathcal{P}(\{x, y, z\}) = \{\{\}, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$.

$$\frac{A : \mathsf{set}}{\mathcal{P}(A) : \mathsf{lattice}_\bot}$$

$$\mathcal{P}(A) \cong A \to \mathbb{B}$$

$$a \sqsubseteq b = a \subseteq b \qquad a \sqcup b = a \cup b \qquad \bot = \{\}$$

The *powerset* can also be represented by a function composition that maps each set element to a boolean that states its presence in the subset.

This composition is very general since it can produce a $\mathsf{lattice}_\bot$ from any $\mathsf{set}$.

A natural extension is to represent *mutisets* by mapping the domain set to naturals, instead of booleans.

$$\frac{A : \mathsf{set}}{\mathcal{M}(A) : \mathsf{lattice}_\bot}$$

$$\mathcal{M}(A) \cong A \to \mathbb{N}$$

$$a \sqsubseteq b = a \subseteq b \qquad a \sqcup b = a \cup b \qquad \bot = \{\}$$

The generic inflations defined for functions when used here show that adding elements is inflationary. For sets represented by $A \to \mathbb{B}$ with a given state $s$ we can define how to add an element $e$.

$$\mathsf{add}(e)(s) = \mathsf{apply}_e(\underline{\mathsf{True}})(s)$$

Likewise, when adding on multisets $A \to \mathbb{N}$ one increments the element count, having a strict inflation.

$$\mathsf{add}(e)(s) = \mathsf{apply}_e(\mathsf{succ})(s)$$

### 4.6 Antichain of Maximal Elements

Starting from a poset this construction produces a $\mathsf{lattice}_\bot$ by keeping an antichain of maximal elements, given the base poset order. Upon join, all elements that are concurrent are kept, but any element that is present together with a higher element is removed.

$$\frac{A : \mathsf{poset}}{\mathcal{A}(A) : \mathsf{lattice}_\bot}$$

$$\mathcal{A}(A) = \{\mathsf{maximal}(a) \mid a \in \mathcal{P}(A)\}$$

$$\mathsf{maximal}(a) = \{x \in a \mid \nexists y \in a \cdot x \sqsubset y\}$$

$$a \sqsubseteq b = \forall x \in a \cdot \exists y \in b \cdot x \sqsubseteq y$$

$$a \sqcup b = \mathsf{maximal}(a \cup b)$$

$$\bot = \{\}$$

## 5 Abridged Catalog

In order to exemplify the composition constructs we present a small set of example CRDTs. Simple query functions are included and all mutators are inflations.

Notice that join does not need to be defined as it follows from the composition rules that were introduced.

## 5.1 Positive Counter

This simple form of counter can only increase. Replica nodes must have access to unique ids among a set $I$ and can only increment its position in a map of ids to integers. While increment mutators are parametrized by id $i$ the query is anonymous and simply inspects the state.

$$\mathsf{PCounter}(I) = I \hookrightarrow \mathbb{N}$$

$$\begin{aligned} \mathsf{inc}_i(a) &= \mathsf{apply}_i(\mathsf{succ})(a) \\ \mathsf{value}(a) &= \sum\{v \mid (c, v) \in a\} \end{aligned}$$

Notice that if a given node does not yet have an entry in the map and increments, then $\mathsf{succ}$ applies over $\bot$, which for $\mathbb{N}$ was defined to be 0.

## 5.2 Positive and Negative Counter

This variation allows for both increments and decrements. A solution is to pair two positive counters and consider the right side as negative. We use the standard functions $\mathsf{fst}()$ and $\mathsf{snd}()$ to respectively access the left and right elements of a pair.

$$\mathsf{PNCounter}(I) = I \hookrightarrow \mathbb{N} \times I \hookrightarrow \mathbb{N}$$

$$\begin{aligned} \mathsf{inc}_i(a) &= \mathsf{apply}_i(\mathsf{succ})(\mathsf{fst}(a)) \\ \mathsf{dec}_i(a) &= \mathsf{apply}_i(\mathsf{succ})(\mathsf{snd}(a)) \\ \mathsf{value}(a) &= \sum\{v \mid (c, v) \in \mathsf{fst}(a)\} - \sum\{v \mid (c, v) \in \mathsf{snd}(a)\} \end{aligned}$$

An alternative way to obtain a similar result (the difference being that this counter is non-negative but can lose decrements) is to use a lexicographic pair and have the first element incremented when one needs to update the count on the second element.

$$\mathsf{PNCCounter}(I) = I \hookrightarrow \mathbb{N} \boxtimes \mathbb{N}$$

$$\begin{aligned} \mathsf{inc}_i(a) &= \mathsf{apply}_i(\mathsf{succ} \boxtimes \mathsf{succ})(a) \\ \mathsf{dec}_i(a) &= \mathsf{apply}_i(\mathsf{succ} \boxtimes \mathsf{pred})(a) \\ \mathsf{value}(a) &= \sum\{\mathsf{snd}(v) \mid (c, v) \in a\} \end{aligned}$$

$$\mathsf{pred}(x) = \mathbf{max}(x - 1, 0)$$

### 5.3 Observed-remove Add-wins Set

An observed-remove set with add-wins semantics can be derived by creating unique tokens whenever a new element is inserted, using for that a grow only counter per replica, and canceling this tokens, by increasing a boolean to $\mathsf{True}$, upon removal. Only elements supported by non-canceled tokens are considered to be in the set.

$$\mathsf{ORSet}^{+}(E, I) = E \hookrightarrow I \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}$$

$$\begin{aligned} \mathsf{add}_{e,i}(a) &= \mathsf{apply}_e(\mathsf{apply}_i(\mathsf{succ} \boxtimes \underline{\mathsf{False}}))(a) \\ \mathsf{rmv}_e(a) &= \mathsf{apply}_e(\mathsf{map}(\mathsf{id} \boxtimes \underline{\mathsf{True}}))(a) \end{aligned}$$

$$\mathsf{member}_e(a) = \exists (e, m) \in a \cdot \exists i, n \cdot (n, \mathsf{False}) \in m(i)$$

### 5.4 Observed-remove Remove-wins Set

An observed-remove set with remove-wins semantics is derived by a dual construction to the previous one, while sharing the same state lattice. Removal creates unique tokens, and additions need to cancel all remove tokens that are visible in the state.

$$\mathsf{ORSet}^{-}(E, I) = E \hookrightarrow I \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}$$

$$\begin{aligned} \mathsf{rmv}_{e,i}(a) &= \mathsf{apply}_e(\mathsf{apply}_i(\mathsf{succ} \boxtimes \underline{\mathsf{False}}))(a) \\ \mathsf{add}_e(a) &= \mathsf{apply}_e(\mathsf{map}(\mathsf{id} \boxtimes \underline{\mathsf{True}}))(a) \end{aligned}$$

$$\mathsf{member}_e(a) = \exists (e, m) \in a \cdot \nexists i, n \cdot (n, \mathsf{False}) \in m(i)$$

### 5.5 Multi-value Register

A non-optimized multi-value register can be derived by lexicographic coupling of a version vector clock $C$ with a payload value $V$. When a new value $v$ is to be assigned, a new clock, greater than all visible clocks in the state, is created and coupled with the value. These pairs are kept in a antichain of maximal elements. Thus, upon merge, concurrently assigned values will be collected, but any subsequent assignment will again reduce the state to a single pair value.

$$
\begin{aligned}
\mathsf{MVReg}(V, I) &= \mathcal{A}(C \boxtimes V) \\
C &= I \hookrightarrow \mathbb{N}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{assign}_{v,i}(a) &= \{\mathsf{apply}_i(\mathsf{succ})(\bigsqcup\{c \mid (c, v') \in a\}) \boxtimes v\} \\
\mathsf{values}(a) &= \{v \mid (c, v) \in a\}
\end{aligned}
$$

Notice that the value is never updated without creating a new clock. Thus, lexicographic comparison (needed for the operation of the antichain join) is always decided by the first component, and in practice $V$ can be any opaque payload without need to define a partial order on its values.

# 6 Closing Remarks

This report collects several composition techniques for lattices, adopts the notion of inflation and how it applies to the specification of state based CRDTs over lattices. Most of the lattice compositions are very standard techniques from order theory [5]. An early version of this work was presented at Schloss Dagstuhl under the title *Composition of Lattices and CRDTs* and the summary of the presentation is available at [6]. Most of the CRDT constructions used here are influenced by work in [8, 7, 2, 4, 3, 1].

# References


[1] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based CRDTs operation-based. In *Proceedings of Distributed Applications and Interoperable Systems: 14th IFIP WG 6.1 International Conference*. Springer, 2014.

[2] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In Marcos K. Aguilera, editor, *Int. Symp. on Dist. Comp. (DISC)*, volume 7611 of *Lecture Notes in Comp. Sc.*, pages 441–442, Salvador, Bahia, Brazil, October 2012. Springer-Verlag.

[3] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In Suresh Jagannathan and Peter Sewell, editors, *POPL*, pages 271–284. ACM, 2014.

[4] Neil Conway, William R Marczak, Peter Alvaro, Joseph M Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM, 2012.

[5] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order (2. ed.)*. Cambridge University Press, 2002.

[6] Bettina Kemme, André Schiper, G. Ramalingam, and Marc Shapiro. Dagstuhl seminar review: Consistency in distributed systems. *SIGACT News*, 45(1):67–89, March 2014.

[7] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapp. Rech. 7506, Institut National de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, January 2011.

[8] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and V. Villain, editors, *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, October 2011. Springer-Verlag.

## B.2 Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict-free Partially Replicated Data Types. Submitted to PPoPP 15.

# Conflict-free Partially Replicated Data Types


## Abstract

Designers of large user-oriented distributed applications, such as social networks and mobile applications, have adopted measures to improve the responsiveness of their applications. Latency is a major concern as people are very sensitive to it. Geo-replication is a commonly used mechanism to bring the data closer to the clients. Nevertheless, reaching the closest datacenter can still be considerably slow. Thus, in order to further reduce the access latency, mobile and web applications may be forced to replicate data at the client-side. Nevertheless, fully replicating large data structures may still be a waste of resources, specially for thin-clients.

We propose a replication mechanism built upon conflict-free replicated data types (CRDT) to seamlessly replicate parts of large data structures. We define partial replication and give an approach to keep the strong eventual consistency properties of CRDTs with partial replicas. We integrate our mechanism into SwiftCloud, a transactional system that brings geo-replication to the client. We evaluate the solution with a content-sharing application. Our results show improvements in bandwidth, memory, and latency usage over both classical geo-replication and the existing SwiftCloud solution.



***Categories and Subject Descriptors*** E.1 [*Data Structures*]: Distributed Data Structures

***Keywords*** Partial replication; CRDTs


## 1. Introduction

Globally accessible web applications, such as social networks, aim to provide low-latency access to their services. Thus, data locality is a fundamental property of their systems. Geo-replication is a common solution where the data is replicated in multiple datacenters [9, 11, 12]. In this scenario, user requests are forwarded to the closest datacenter. Therefore, the latency is reduced. Unfortunately, the latency, even when accessing the closest datacenter, may still be considerable. It has been proved that clients are sensitive to even small increases of latency [10, 18].

Systems such as [17, 21] use caching techniques to yet reduce latency even more. However, this can be challenging and expensive. For instance, one could simply use the client caches for reading purposes. Nevertheless, in order to keep some consistency guarantees and freshness of the data, mechanisms, such as cache invalidation, need to be used. Scaling these kinds of techniques is difficult and directly affects the performance. Moreover, one could let clients apply write operations locally and eventually propagate them. However, this can cause conflicts between replicas and potential rollback situations.

The recently formalized CRDTs [19, 20] can serve to diminish the impact of some of the previously mentioned problems. Due to CRDTs semantics, rollback situations can not occur. Moreover, these data structures are conflict-free by default; therefore, no conflict resolution mechanisms need to be written by application developers. SwiftCloud [25], a geo-replicated storage system that ensures causal consistency, benefits from the CRDTs semantics. It replicates CRDTs not only across datacenters, but it also replicates them in clients. It allows read and write operations to be directly executed in clients caches. In consequence, SwiftCloud reduces latency, and enables off-line mode during disconnection periods.

The current specifications of CRDTs do not allow them to be partitioned. Thus, a CRDT replica is assumed to contain the full data structure. We believe CPRDTs can be effectively used to address two relevant issues of current systems:

- CRDTs can easily become heavy data structures. For instance, a CRDT that contains the posts of a user wall in a Facebook-like application. In many cases, the user is simply interested in the most relevant posts, according to some criterium. For instance, one may only be interested in reading the top-ten most voted posts of a Reddit-like application. Thus, replicating the whole CRDT is a waste of resources, of both storage and bandwidth. The former can be critical when thin devices, such as smartphones, are considered as clients. These types of clients have limited memory resources; therefore, it is convenient to avoid storing unnecessary data. On the other hand, bandwidth is one of the most costly resources offered by cloud providers such as Amazon S3 [1], Google Cloud Storage (GCS) [3], and Microsoft Azure [2]; therefore, it is beneficial to use it efficiently.
- The full replication of CRDTs in clients arises security concerns. By partitioning the CRDTs, applications could precisely decide which data each client stores. This could keep malicious clients from storing sensitive data.

In this paper, we propose a new set of CRDTs that allow partitioning. We call them "Conflict-free Partially Replicated Data Structures" (hereafter CPRDTs). We study how partitions of the same CRDT can interact among each other and still maintain its consistency guarantees. Furthermore, we revise previously defined CRDT specifications and propose new specifications that consider partitioning.

The major contributions of this paper are the following:

- The definition of the new CPRDTs. This includes revisiting the specifications of previously defined CRDTs.
- Extension of SwiftCloud to integrate CPRDTs.
- Extensive evaluation of the performance improvements of CPRDTs in SwiftCloud. This includes the creation of a Reddit-like [15, 16] application, called SwiftLinks, on top of SwiftCloud.

The remainder of the paper is organized as follows: Section 2 introduces previous work that we consider relevant to understand this paper; Section 3 presents a formal definition of the partitioned CRDTs and the specifications of some of them; Section 4 discusses how CPRDTs could be practically used; Section 5 presents an extensive evaluation of the SwiftCloud extension that includes CPRDTs; Section 6 briefly describes preceding related work; finally, Section 7 discusses future work and concludes the paper.

## 2. Background

### 2.1 CRDTs

Conflict-free Replicated Data Types (CRDTs) are a set of concurrent data structures that allow replicas to be updated concurrently and guarantee all replicas will eventually converge to the same state [19, 20]. While traditional approaches require user interference [21] or last-write-wins [24] to resolve possible conflicts, CRDTs avoid conflicts by leveraging simple mathematical properties, i.e. commutativity of operations and monotonicity of the state. Based on this, there are two types of CRDTs, namely operation-based and state-based, which differ in their propagation model.

Operation-based CRDTs require concurrent operations to be commutative. Update operations are performed at one replica which asynchronously propagates them to the rest. Causal delivery is usually required. Nevertheless, the order in which concurrent updates are delivered does not affect the convergence of the replicas. On the other hand, State-based CRDTs require its internal state to grow monotonically. Replicas send their internal states to the rest of replicas. Upon receiving a state, the replica has to merge it with its local state.

For both kinds of CRDTs, replicas will converge to the same state after they have seen the same set of operations.

### 2.2 SwiftCloud

SwiftCloud is a geo-replicated cloud storage system that stores CRDTs and caches data at clients [25]. It consists of several datacenters that fully replicate data. Clients communicate with the closest data center and cache accessed data in local cache.

SwiftCloud supports causal+ mergeable transactions for CRDTs. When a transaction only involves CRDTs, it is mergeable since it can not have conflict with any other concurrent transactions. Therefore, a transaction is firstly executed only in client cache, unless there is a cache miss so the transaction has to fetch data from the data center. After a transaction has committed, its effect is only visible locally but not to datacenters and other clients. An update transaction will be asynchronously propagated to a datacenter. Eventually the transaction will be applied to all data centers and will be visible to other clients.

Causality for each transaction is tracked by checking its start timestamp and apply them in causal order. When clients start a transaction, it assigns the transaction a vector clock that summarizes its causal dependency. The transaction reads the largest versions of objects that are less than or equal to the start timestamp, in order to read from a causally consistent snapshot. When a transaction arrives at a data center, the data center first checks if it satisfies the transaction's dependencies. If yes, the data center will assign the transaction a global transaction id and make it globally visible after it is applied to all data centers.

## 3. Conflict-free Partially Replicated Data Types

In this section we present the Conflict-free Partially Replicated Data Types (CPRDTs). These new data types are CRDTs that can be partitioned. We believe that partitioning permits a more efficient usage of resources such as memory and bandwidth. This may be critical when thin clients, such as mobile devices or embedded computers, cache the data structures. On the other hand, we believe CPRDTs have other applications. For instance, CPRDTs can be used to enforce fine-grained security policies. Furthermore, they can also be used to provide a way to support data with multiple fidelity requirements to accommodate resource-thin devices while keeping consistency between the fidelity levels [23]. This can be achieved by not replicating less important information on mobile devices.

This poses new challenges: all operations are not enabled on partial replicas, which means new preconditions must be added to ensure correct usage. However, these conditions must not interfere with the convergence of the replicas. Care must be taken as a partial replica may change over time. A partial replica could change the parts it keeps, by becoming interested in more parts. This has to be carefully done without loosing data and still achieving convergence between replicas.

### 3.1 Example of use

Let's use an example to illustrate the advantages of CPRDTs: the user wall of a social network. We can model a user's wall with an OR-set CRDT. In this example, there are four users that interact: Alice, Bob, Charlie and an anonymous user. Bob is a friend of Alice, while Charlie is a friend of Bob, but not of Alice. Participating users may want to read or post something in Alice's wall. We make two assumptions:

- Users maintain a full replica of their wall.
- A user $X$ that reads or posts in user's $Y$ wall replicates user's $Y$ wall locally.

Each post contains a date, an author, a message and a privacy setting. The privacy setting restricts who is allowed to read the posts. We can assume there are three security levels: public, friends of friends, and friends. Then, Alice and Bob can read all the posts of Alice's wall. Charlie can only read public and Bob's posts. Finally, any other user can only read public posts.

CPRDTs have two applications in this scenario: (i) limiting the size of the wall to be replicated, which can lead to a better usage of memory and bandwidth; and (ii) enforcing security policies.

We can assume that Alice has been using the social network for a few years and there are a considerable number of posts on her wall. It seems natural that a user should not have to replicate the whole wall to simply read the latest posts. Nevertheless, this is what presumably may occur in a fully-replicated scenario, where the data structures cannot be partitioned and we still want to replicate data in clients-side.

One solution is to manually split the data structure according to some criteria (e.g. by date, author or privacy setting). However, developers should anticipate how users will use the application. While possible in some cases, it makes the application more cumbersome to write. Furthermore, it would be difficult to achieve optimal results since each client may behave differently.

On the other hand, CPRDTs abstract the partitioning from the application. Thus, from the point of view of the programmer, there will only be one logical data structure per wall. We strongly believe this ease developers task. Moreover, this allows a more efficient and fine-grained partitioning adapted to the needs of a particular client in a specific point of time. For instance, Bob might want to look at the posts that Alice and himself made during the last week. On the other hand, Charlie may want to see all the posts of the last two years. This two request will end up with completely different parts of the same CRDT. Only with CPRDTs, optimal results can be achieved.

The second application of CPRDTs is related to the enforcement of security policies. Due to the security setting field of a post, we want users to only replicate posts which they are allowed to see. For

instance, an anonymous user should only replicate public posts. On the other hand, Charlie can also replicate "friends of friends" posts. This will keep malicious users from storing sensitive data locally.

### 3.2 Definitions

Before defining CPRDTs, we have to clarify some concepts that we will use throughout the paper.

An *object* is a named instance of a CRDT or CPRDT in our case. Each participating process replicates a set of objects. The objects can be read using *query* operations and modified using *update* operations. The query operations return the external state of the object, that we call the *data* of the object. Nevertheless, additional data, which we refer as *metadata*, is kept internally to ensure convergence.

An update operation can have preconditions that capture its safety requirements. In consequence, an operation is said to be *enabled* at a replica, if it satisfies its preconditions. For instance, the remove operation of a set is enabled only if the element to be removed is present in the set.

Previous definitions fit into both CRDTs and CPRDTs. Nevertheless, for CPRDTs, we further consider that a process might replicate an object partially: it only has access to the part of the data that is relevant for the client, and the process only keeps the metadata required for that given part. Intuitively, this means that only part of the data structure is replicated: some elements of a set, a subgraph of a graph, or a slice of a sequence. The part replicated is defined by the specifications of the CPRDT.

***particle*** We define *particles* as the smallest meaningful elements of a CPRDT. By meaningful we refer to the smallest element that can be used for query and update operations. For instance, a particle in a grow-only set would be any element that can be added or looked up in the set. The set of all particles of a CPRDT is denoted by $\pi$. In many cases, such as unbound counters and sets, the set of particles of a CRDT is infinite.

Apart from the definition of particles, we need to introduce three functions to understand CPRDTs: *shard*, *required*, and *affected*.

***shard*** Each replica of a CPRDT $x_i$ has associated a set of particles. The set of particles is defined by $\mathrm{shard}(x_i)$, by analogy to the databases concept. The replica only knows the state of the particles in $\mathrm{shard}(x_i)$; therefore, it can only enable query and update operations that require and affect those particles. Furthermore, the CPRDT replica only needs to receive update operations that affect the particles in $\mathrm{shard}(x_i)$ in order to converge. For now, we assume that $\mathrm{shard}(x_i)$ is chosen when $x_i$ is created. We will relax this assumption in Section 4.3.

There are two special cases: a *full* replica and a *hollow* replica. When $\mathrm{shard}(x_i) = \pi$ then we say that $x_i$ is a *full* replica, and it is equivalent to a normal CRDT. On the other hand, when $\mathrm{shard}(x_i) = \varnothing$, then $x_i$ is a *hollow* replica (as named in [13]). A *hollow* replica does not maintain any state. Nevertheless, it can still handle updates, as explained in section 3.3.2.

***required*** For an operation $op$ with its arguments, $\mathrm{required}(op)$ is the set of particles needed by $op$ to be properly executed. This means that, for replica $x_i$, an operation is enabled only if $\mathrm{required}(op) \subseteq \mathrm{shard}(x_i)$. E.g. for the lookup operation of a set, $\mathrm{required}(lookup(e)) = \{e\}$ where $e$ is an element of the set. In case $e \notin shard(x_i)$, the replica will not be able to know whether $e$ is in the set because it has not kept a state for it. This implies that updates affecting $e$ have not been necessarily seen by $x_i$.

***affected*** The function $\mathrm{affected}(op)$ tells us the set of particles that may have their state affected after executing an update operation.

### 3.3 Replication

As for the original CRDTs, we consider two equivalent replication techniques: state-based and operation-based. Allowing partitioning introduces changes in the way these replication techniques work. Furthermore, concepts such as causal history and convergence have to be revisited.

First, we need to define when two replicas are equivalent.

**Definition 1** (Equivalence between replicas)**.** *$x_i$ and $x_j$ have equivalent common abstract states if all `query` operations $q$, for which $\mathrm{required}(q) \subseteq (\mathrm{shard}(x_i) \cap \mathrm{shard}(x_j))$, return the same values.*

One requirement for replicas to converge is that they apply, directly or indirectly, the same update operations. We can informally define the causal history of a replica ($x_i$) as the applied update operations ($C(x_i)$). Later in the section, we will formally define it.

Now, we are ready to formally define convergence in the context of CPRDTs:

**Definition 2** (Eventual Convergence of Partial Replicas)**.** *Two partial replicas $x_i$ and $x_j$ of an object $x$ converge eventually if the following conditions are met:*

- *Safety: $\forall i, j : C(x_i) = C(x_j)$ implies that the abstract states of $i$ and $j$ are equivalent on their common particles.*
- *Liveness: $\forall i, j : f \in C(x_i)$ implies that, eventually, if $\mathrm{affected}(f) \cap \mathrm{shard}(x_j) \neq \varnothing$, then $f \in C(x_j)$.*

#### 3.3.1 State-based partial replication

This form of replication is interesting if the size of the state is relatively small compared to the size and number of updates, as only the state must be sent over the network. CPRDTs can optimize this technique since only parts of the state need to be sent and received.

We define the causal history of a replica for state-based replication as follows:

**Definition 3** (Causal History on Partial Replicas - state-based)**.** *For any replica $x_i$ of $x$:*

- *Initially, $C(x_i) = \varnothing$.*
- *Before executing `update` operation $f$,*
  *if $\mathrm{affected}(f) \cap \mathrm{shard}(x_i) \neq \varnothing$ then execute $f$ and $C(f(x_i)) = C(x_i) \cup \{f\}$,*
  *otherwise $C(f(x_i)) = C(x_i)$.*
- *After executing merge against states $x_i$, $x_j$, $C(x_i \bullet \mathrm{merge}(x_j)) = C(x_i) \cup \{f \in C(x_j) | \, \mathrm{affected}(f) \cap \mathrm{shard}(x_i) \neq \varnothing$*

The *merge* method used by a replica must only merge the state of its particles with the remote replica and ignore the others, so that $\mathrm{shard}(x_i) = \mathrm{shard}(x_i \bullet \mathrm{merge}(x_j))$.

To achieve convergence with state-based replication on partial replicas, an additional condition is needed for an update to be enabled at a replica. Since updates are indirectly replicated through the state, an operation cannot be applied if it affects a particle that is not in that replica's shard, this would violate the liveness property of convergence as that update might not be added to the causal history of another replica when merging. Thus, an operation $f$ is disabled if $\mathrm{affected}(f) \nsubseteq \mathrm{shard}(x_j)$.

Since the replicas only converge on their common parts, a replica $x_i$ just needs to send to another, $x_j$, the state of the intersection of their shards ($\mathrm{shard}(x_i) \cap \mathrm{shard}(x_j)$).

#### 3.3.2 Operation-based partial replication

As with classic CRDTs, the update operations are divided into two phases: *prepare* and *downstream* phase. The former is done

at the source replica and does not have any side-effect. The latter is applied at all replicas and it affects the state of the replica.

In contrast to CRDTs, CPRDTs only have to broadcast updates to the replicas interested in the particles affected by the update. Therefore, an update $u$ is broadcasted to $x_i$ if $\text{affected}(u) \cap \text{shard}(x_i) \neq \varnothing$.

This poses an interesting situation. A CPRDT replica can complete the first phase of the update operation without necessarily complete the second phase. For instance, a replica $x_i$, whose $\text{shard}(x_i)$ are particles $a$ and $b$, receives an update operation that affects particle $c$. In this situation $x_i$ can complete the prepare phase, broadcast the downstream operation to the interested replicas, and discard it locally. We named this scenario as *blind updates*. It is important to highlight that this cannot happen in state-based replication. Hollow replicas, which have an empty shard, can only do blind updates.

**Definition 4** (Causal History on Partial Replicas - op-based)**.** *For any replica $x_i$ of $x$:*

- *Initially, $C(x_i) = \varnothing$.*
- *After executing the `downstream` phase of operation $f$ at replica $x_i$,*
  *if* $\text{affected}(f) \cap \text{shard}(x_i) \neq \varnothing$ *then* $C(f(x_i)) = C(x_i) \cup \{f\}$,
  *otherwise* $C(f(x_i)) = C(x_i)$.

### 3.4 Specification

In this section, we extend the CRDT specification models.

#### 3.4.1 Creation of a new partial replica

The creation of new replicas in CRDTs is rather straightforward. The CRDT can simply be copied in its entirety. Nevertheless, in the context of CPRDTs, we want to choose which particles to copy.

In order to solve this problem, we propose a new operation, called $\text{fraction}$, that allows us to create new partial replicas from a subset of a given replica. The subset we want to copy in the new replica is defined by a set of particles. More formally, $\text{fraction}$ can be defined as follows:
$x_j = \text{fraction}(x_i, Z)$, where $Z$ is the set of particles we want to take. The operations ensures that $\text{shard}(x_j) = \text{shard}(x_i) \cap Z$.

Please notice that using a set of particle is the canonical form to define the subset. In practice, it can be defined by using a more high-level query language. For instance, an application could issue a query in the form of "give me the first 10 elements of your sorted set", which can then be transformed into a set of particles. We further discuss this in Section 4.1.

This operation is also useful to simplify the specifications of state-based CRDTs: when merging two partial states, we only want to merge the state of the common particles since the rest can be ignored. However, putting this in the specification is cumbersome.

Instead, we assume that the merge operation merges the complete payloads, regardless of their shard. We can then limit the growth of the replica to its own shard as such: if replica $x_j$ receives the payload of replica $x_i$, $x_j$ should do:
$x_k = \text{fraction}(\text{merge}(x_i, x_j), \text{shard}(x_j))$
Thus $\text{shard}(x_k) = \text{shard}(x_j)$ and, in consequence, the replica does not grow. In practice the fraction operation can be applied before sending the payload to another replica (to save bandwidth), but to keep the convergence property, the fraction taken from replicas $x_i$ and sent to $x_j$ must have at least the particles $\text{shard}(x_i) \cap \text{shard}(x_j)$. Otherwise, $x_j$ may miss some updates.

#### 3.4.2 Specification model

The specifications are similar to the CRDT specifications, with some added notations. Each operation must define which particles it involves (*required* particles and *affected* particles). Note that the conditions given in section 3.2 ($\text{required}(op) \subseteq \text{shard}(x_i)$, and $\text{affected}(op) \subseteq \text{shard}(op(x_i))$ for state-based replication), regarding whether an operation is enabled or not, are not explicitly included in the specification. Nevertheless, it must be enforced.

Specification 1 and Specification 2 show the template of specifications for state-based and operation-based CPRDTs respectively.

**Specification 1** State-based object specification with Partial Replication

**particle definition** Informal definition of what is a particle
**payload** type
    **initial** *Initial value*
**query** *query(arguments)* : *returns*
    **required particles** *Set of required particles*
    **pre** *Precondition*
    **let** *Evaluate synchronously, no side effects*
**update** *update(arguments)* : returns
    **required particles** *Set of required particles*
    **affected particles** *Set of particles on which there can be an effect*
    **pre** *Precondition*
    **let** *Evaluate at source, synchronously*
**merge** ($value1$, $value2$) : payload $mergedValues$
    *Least Upper Bound merge of* $value1$ *and* $value2$
    $\text{shard}(mergedValues) = \text{shard}(value1) \cup \text{shard}(value2)$ *must be true*
**fraction** (particles $selection$) : payload $partialReplica$
    *Copies the particles* $selection$ *into* $partialReplica$ *so that* $\text{shard}(partialReplica) = selection \cap \text{shard}(self)$ *(*$self$ *is the replica on which* $fraction$ *is applied to).*

**Specification 2** Op-based specification model with Partial Replication

**particle definition** Informal definition of what is a particle
**query** *query(arguments)* : *returns*
    **required particles** *Set of required particles*
    **pre** *Precondition*
    **let** *Evaluate synchronously, no side effects*
**update** *Global update(arguments)* : returns
    **prepare** (*arguments*) : *intermediate value(s) to pass downstream*
        **required particles** *Set of required particles to prepare the update*
        **pre** *Precondition*
        **let** *1st phase: synchronous, at source, no side effects*
    **effect** (*arguments passed downstream*)
        **required particles** *Set of required particles when applying the update*
        **affected particles** *Set of particles which might be affected when applying the update*
        **pre** *Precondition against downstream state*
        **let** *2nd phase: asynchronous, side effects to downstream state*
**fraction** (particles $selection$) : payload $partialReplica$
    *Copies the particles* $selection$ *into* $partialReplica$ *so that* $\text{shard}(partialReplica) = selection \cap \text{shard}(self)$ *(*$self$ *is the replica on which* $fraction$ *is applied to).*

### 3.5 CPRDT examples

In this section we propose the specifications for some CPRDTs. We mostly adapt the CRDT specifications proposed by Shapiro

et al ([19, 20]). We also introduce a tree CPRDT. To the best of our knowledge, a tree CRDT has never been formally specified before. Due to space restrictions, we are forced to only present few CPDRTs; nevertheless, more CPRDTs specifications can be found in [7].

***Grow-Only set*** Specification 3 gives a simple state-based grow only set (which only support the add operation).

**Specification 3** State-based Grow-Only Set (G-set) with Partial Replication

**particle definition** A possible element of the set.
**payload** set $A$
  **initial** $\varnothing$
**query** lookup(element $e$) : boolean $b$
  **required particles** $\{e\}$
  **let** $b = e \in A$
**update** add(element $e$)
  **required particles** $\varnothing$
  **affected particles** $\{e\}$
  $A := A \cup \{e\}$
**merge** $(S, T)$ : payload $U$
  **let** $U.A = S.A \cup T.A$
**fraction** (particles $Z$) : payload $D$
  **let** $D.A = A \cap Z\}$

***Observed-Removed set*** In the Specification 4, we show the CPDRT specification of an Observed-Removed set. It is an operation-based specification that assumes causal delivery of its operations to optimise the payload. A particle is defined as an element of the set.

Notice that the add operation can be a blind update: it does not require any particle in the prepare phase, and it can thus be prepared by a replica which does not have the element to be added in its shard. The remove operation does require the particle of the element it removes, as it needs to send the added $(e, u)$ pairs it observed to the other replicas.

***Grow-only tree*** A state-based grow-only tree is specified in Specification 5. A node is defined by its path and its content in a recursive way, which is noted as $(parent, nodeContent)$, where $parent$ is defined similarly. The root is represented by empty: $()$. For instance, a node $(((), 1), 2)$ has content 2 and parent $((), 1)$. This allows to make a grow-only tree that is very similar to a set, with only the added precondition that the parent must exist when adding a node. It also means that adding nodes which have the same value and the same parent result in one node in the tree.

The particles for this tree are the nodes (with their parent, as defined).

**Specification 4** Op-based Observed-Remove Set (OR-set) with Partial Replication

**particle definition** A possible element of the set.
**payload** set $S$
  **initial** $\varnothing$
**query** lookup(element $e$) : boolean $b$
  **required particles** $\{e\}$
  **let** $b = \exists u : (e, u) \in S$
**update** add(element $e$)
  **prepare** $(e) : \alpha$
    **let** $\alpha = unique()$
  **effect** $(e, \alpha)$
    **affected particles** $\{e\}$
    $S := S \cup \{e, \alpha\}$
**update** remove(element $e$)
  **prepare** $(e)$ : R
    **required particles** $\{e\}$
    **pre** lookup($e$)
    **let** $R = \{(e, u) | \exists u : (e, u) \in S\}$
  **effect** $(R)$
    **affected particles** $\{e\}$
    **pre** $\forall (e, u) \in R : add(e, u)$ has been delivered
    $S := S \setminus R$
**fraction** (particles $Z$) : payload $D$
  **let** $D.S = \{(e, u) \in S | e \in Z\}$
**add** (payload $U$)
  **let** $S = S \cup U.S$

**Specification 5** State-based Grow-Only Tree (G-tree) with Partial Replication.

**particle definition** A node of the tree.
**payload** set $A$
  **initial** $\varnothing$
**query** lookup(node $n$) : boolean $b$
  **required particles** $\{n\}$
  **let** $b = n \in A$
**update** add(node $(parent, content)$)
  **required particles** $\{parent\}$ (if parent is not the root)
  **affected particles** $\{(parent, content)\}$
  **pre** $parent \in A$
  $A := A \cup \{(parent, content)\}$
**merge** $(S, T)$ : payload $U$
  **let** $U.A = S.A \cup T.A$
**fraction** (particles $Z$) : payload $D$
  **let** $D.A = A \cap Z\}$

## 4. Practical usage

In this section, we explain how CPRDTs can be used in practice. This takes us to discuss three things: (i) how shard can be practically defined, (ii) how replicas of CPRDTs are created and modified through *shard queries*, and (iii) how the shard can be managed in a real system. The last part of the section discusses a centralized system model aim to simplify and ease the integration of CPRDTs.

### 4.1 Shard definition

In Section 3, we defined the shard of a replica as a set of particles. This set can be infinite; therefore, all elements of the set are not explicitly kept in practice as we only need to know whether a particle is in the shard or not. A shard can be then defined as a range of particles. For instance, on a set of integers, we can define it as $[0, 2]$ for particles $\{0, 1, 2\}$, or even $(0, +\infty)$ for strictly positive integers. Similarly, it can be defined as all the particles that satisfy a specific property. For example, only the odd integers.

### 4.2 Shard query

A *shard query* defines the set of particles that satisfy a particular criterium. Thus, in practice, *shard queries* can be used for two reasons: (i) creation of new CPRDTs from the returned set of particles, and (ii) shrinking or lengthening of the *shard set*. The latter is discussed in more detailed in 4.3.

*Shard queries* bridge the gap between the application semantics and the function fraction, introduced in 3.4.1. Thus, it adds expressiveness to the usage of CPRDTs.

We identify two types of *shard queries*: state-independent and state-dependent queries. The former only depends on the properties

of the particles, and not in the state of CPRDT. In contrast, the latter depends on the current version of the CPRDT. For instance, a state-independent query over a set of integers could be "integers greater than 0". On the other hand, a state-dependent query could be "10 highest integers in the set". The state-independent query does not depend on the state of the CPRDT, and the result of the query will always be the set $(0, +\infty)$. Nevertheless, the state-dependent query will have a different result depending on which elements have been already added, and removed, on the version considered.

State-independent queries are easier to work with: they are comparable. One could determine which query is more specific without having to know the state of the CPDRT they apply to. While with state-dependent queries, one can only compare queries if they apply to the same version of the object. Nevertheless, we believe both types are needed in order to make CPRDTs usable. In 4.4, we describe a system model that can simplify the integration of both types of queries.

### 4.3 Dynamic shard set

Dynamic *shard set* refers to the capability of a partial replica to modify, either shrink or lengthen, its *shard set*. We believe this capability is very useful in practice. For instance, a client may become interested on new parts. Having dynamic *shard set*, the replica does not need to be re-created, only the missing state needs to be grabbed.

Nevertheless, maintaining convergence in some scenarios can become challenging. On one hand, a partial replica can easily shrink its *shard set* without compromising convergence in the operation-based scenario. The replica only needs to take into consideration two things: (i) updates prepared locally have been already broadcasted, and (ii) the data to be dropped is replicated by some other replica; therefore, data do not disappear. On the other hand, lengthening a partial replica is more tricky. For instance, in an operation-based scenario, the following situation can easily occur:

- A replica's ($x_i$) *shard set* is $a, c$.
- $x_i$ did not receive updates that affect $b$ for a while.
- Suddenly, $x_i$ becomes interested in $b$ and starts accepting updates on $b$.
- Unfortunately, the replica will not converge since updates have been missed.

In previous scenario, extra communication between replicas would be needed in order to recover dropped updates. This is clearly not easy to achieve.

In state-based replication, shrinking or lengthening the *shard set* is simpler. On one hand, a replica only needs to broadcast its state before shrinking its *shard set*. On the other hand, a replica that wants to lengthen its *shard set* only needs to merge its current state with the state of a replica that contains the new particles.

### 4.4 Centralised system model

We have not specified a system model up to now. We have only said that processes storing objects propagate either states or operations to reach convergence. CPRDTs are not biased to any specific system model. Nevertheless, assuming a centralised system model considerable simplifies the management of the partial replicas.

A centralised system model assumes that there is a logically centralised entity (authority) holding a full replica and distributing the updates, or sharing the new states, of the other replicas, stored in clients. The centralised entity does not need to be a unique server, it can perfectly be a datacenter.

This model poses several advantages in comparison to an ad-hoc architecture where no authority is assumed. Firstly, it makes the model scalable, letting data structures to be replicated in clients at will. Secondly, clients can discard their (partial) replicas at will as long as their updates have been reliably sent to the authority. Thirdly, a client can request any fraction to the authority in order to either get a new partial replica, or to lengthen its own *shard set*. Finally, the authority could store which particles each partial replica has in his *shard set*. Thus, it could only propagate operations to the interested replicas, saving bandwidth.

## 5. Evaluation

In this section, we report the results of our experimental evaluation. This study aim at evaluating the benefits of CPRDTs in terms of memory, bandwidth and latency. Efficient resources usage positively impacts the performance. In our study we compare three approaches: (i) classic geo-replicated system where data is exclusively stored in datacenters, (ii) SwiftCloud, and (iii) our modified version of SwiftCloud that integrates CPRDTs.

***SwiftLinks*** In order to compare the three systems, we implemented a new application, namely SwiftLinks, on top of SwiftCloud. SwiftLinks is a vote-based content-sharing application based on Reddit. In few words, the application allows users to create forums where they can publish post. Then, users can vote positively or negatively the posts. As a consequence, posts get ranked according to the votes and some other criteria. In addition, users can add comments to posts and to other comments. Users can also vote comments, and consequently the comments get ranked. For more information [15, 16].

There are three main types of data structures in SwiftLinks: posts, comments and votes. We use an OR-Set to store all the posts of a forum, i.e. each forum is represented by a OR-Set. We proposed a novel CRDT, namely Remove-once Tree, to store the comments of a post, which naturally form a tree-like structure. Finally, votes, which represent the vote of a single user, are modelled with LWW-Registers. Thus, each post and comment have associated a set of votes.

***Warm-up*** We used Reddit's API to fetch data to warm up our system. For each benchmark, we create 10000 posts over 20 forums (so an average of 500 posts per forum). Each post has 20 comments on average. Moreover, posts have an average of 170 votes, while comments an average of 13 votes.

***Workload*** Our workloads are composed by read and update operations. Read operations are executed over posts and comments. On the other hand, there are three types of update operation: (i) new post, (ii) new comment, and (iii) new vote.

For most of the experiments, 20% of the operation are updates and 80% are read operations. Furthermore, 90% of the operations are biased to previously accessed objects. This means that they are likely to hit the cache. The rest (10%) is done on randomly selected posts and comments.

### 5.1 Experimental setup

SwiftLinks was evaluated using three Amazon EC2 servers as datacenters: one in Ireland and two in the USA (east and west coast). The EC2 instances are equivalent to a single core 64-bit 2.8 GHz Intel Xeon virtual processor (4 ECUs) with 7.5 GB of RAM. The clients run in 15 PlanetLab nodes located near the DCs. These nodes have heterogeneous configurations with varying processing power and RAM. We set up five SwiftLinks users running concurrently per node. Each client performs an operation per second.

There are three main configurations for clients to run the application: *cloud*, *non-lazy*, and *lazy*.

In the *cloud* configuration, operations are applied synchronously at one datacenter and replicated asynchronously to the rest of dat-

acenters. This simulates a typical geo-replication system. In this case, the client does not cache any data.

The other two configurations adopt the SwiftCloud approach of caching data on clients side. We limit the capacity of the cache in our experiments, using 64MB as default size. If the cache size exceeds this limit, the least recently used object is dropped. This simulates memory restrictions on thin clients. In this configurations, *non-lazy* and *lazy*, if the cache contains the required data, the operations are run locally at the clients, and propagated asynchronously to the closest datacenter.

The difference between *non-lazy* and *lazy* is that the latter benefits from the partial replication mechanism described in the paper. This means that objects are fetched in parts as needed, so the cache can hold only parts of an object. On the other hand, for the *non-lazy* configuration, the objects are only fully replicated in clients side, as the original SwiftCloud.

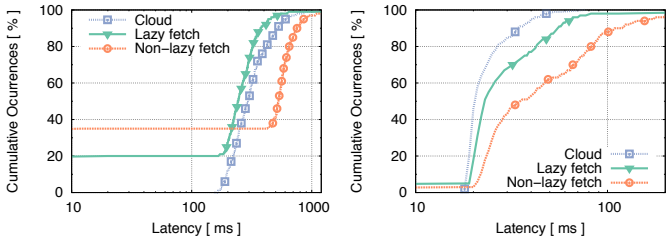
### 5.2 Latency

We evaluated the perceived latency for various operations with and without partial object replication. Figure 1 shows the cumulative distribution functions of different operations' latency with a 64MB cache size limit. These results are obtained after a warm-up phase for the cache. This means that the cache is pre-filled with objects that will be used by the operations present in the workload. For the *non-lazy* and *lazy* mode, there are always a percentage of operations with a very reduced latency. We can conclude that it is the percentage of operations that hit the cache.

***Read operations*** Figure 1a shows that the *non-lazy* mode has greater cache hit rate (35%) than the *lazy* mode. Nevertheless, the hit rate is not optimal due to the limit in the cache size: the cache cannot hold full replicas of all the forums and thus sometimes need to fetch them again. Figure 2 shows the results of a similar experiment but without any cache size limit. In that case, the cache hit rate, for the *non-lazy* mode, is 90%, which corresponds to our ratio of biased operations, and it confirms the previous results with a social network application of the SwiftCloud paper [25]. On the other hand, in *lazy* mode, the cache hit rate is lower, with only 20% in both experiments (figures 1a and 2), because the cache only holds partial replicas which gives it less chance of having all the parts needed for hitting the cache in subsequent operations. However, it has the advantage of a lower maximum latency: if an operation does not hit the cache, it only needs to fetch some parts, instead of the full object. In that scenario, it induces a delay similar to the cloud solution, around 200 to 300 ms, while without lazy fetching, the delay is increased to around 500 to 700 ms by having to replicate a full object. This poses a trade-off between the cache hit rate and the maximum latency. While fully replicating an object will provide more cache hits, a cache miss is more costly.

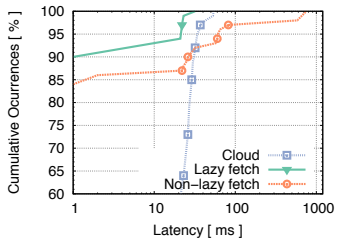
For the latency of reading comments of a post, shown in Figure 1b, the situation is a bit different. Clients are less likely to read the same comment tree multiple times; therefore, this affects the cache hit ratio. As the figure shows, the hit ratio is less than 5% in both *lazy* and *non-lazy* fetching. But again, *lazy* fetching has the advantage of reducing the impact of a cache miss as it only replicates the comments required by the operation instead of the full comment tree. In consequence, the *lazy* approach has a slightly better latency, close to the *cloud* mode. The *cloud* mode performs better because it does never need to fetch any data, which means the returned messages are considerable smaller. Notice that the difference between *non-lazy* and *lazy* mode has been reduced in this experiment because the involved objects are smaller.

***Update operations*** Caching modes (*lazy* and *non-lazy*) are more beneficial with update operations. The reason is that update operations are typically applied on objects, or parts of objects, that




(a) Reads of pages of links

(b) Reads of comments of a link




(c) Updates: posting a link, commenting, voting a link, and voting a comment

Figure 1: Perceived latency of SwiftLinks at one site with medium (64MB) cache size limit and a warmed up cache.

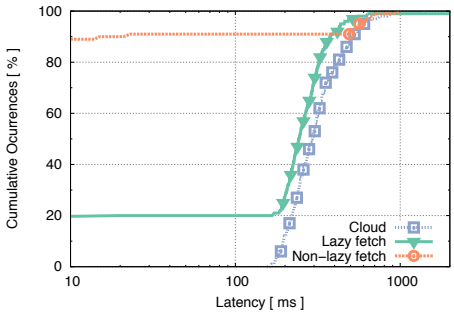



Figure 2: Reads of pages of links with unlimited cache which is already warmed up.

have already been read by the client. In addition, the update operations only use state-independent queries to fetch their missing part, which substantially simplifies the comparison of partial objects in the cache. Figure 1c proves experimentally our reasoning. While the cloud mode has an almost constant latency for all operations of a round-trip time, with caching modes, most of the operations (almost 90%) have no latency. Again, the *lazy* mode has the advantage of reducing the latency when the cache is not hit, as it only needs to fetch the part of the object that needs to be updated, instead of the full object. Moreover, some updates can be done blindly, therefore, they are completed locally.

In particular, Figure 3 shows the benefit of updates when posting comments, which almost always only requires particles already present in the cache. One can see that with lazy fetching, all the operations have almost no latency, as they can be done completely asynchronously. In contrast, in *non-lazy* mode, there can be a large delay when the tree of comments is not in the cache, as it needs

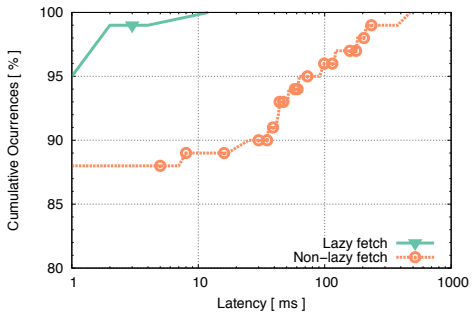



Figure 3: Perceived latency of commenting on a post, at all sites, with medium (64MB) cache size limit.

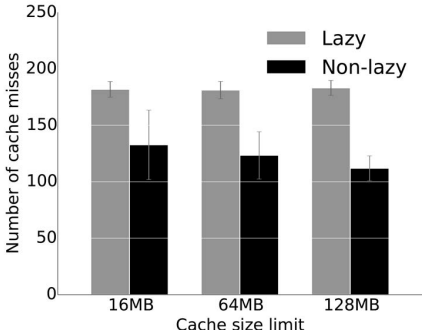



Figure 4: Number of cache misses with different cache size limits.

to be fetched from the store. As in previous scenarios, even if an operation cannot be done completely locally in *lazy* mode, the client only has to fetch part of the tree to complete the update.

### 5.3 Impact of cache size limit

In this section we look at how the application performance changes with various cache size limits (16MB, 64MB, and 128MB).

#### 5.3.1 Impact on latency

We have already proved that the *non-lazy* mode performs better without cache limit when reading links. We run the same experiments showed in Figure 1 setting the cache size limit to 16MB and 128MB. We do not show the plot due to space restrictions.

The experiments show that a smaller cache size limit has a big latency impact on reading links and updates in *non-lazy* mode. Nevertheless, its impact is considerable smaller in *lazy* mode. With a small cache, the cache hit rate of *non-lazy* mode of reading links becomes worse than in *lazy* mode. This is caused because only few objects can fit in the cache at a given time; therefore, clients need to fetch objects more frequently. This results in a lower fraction of operations having no latency, about 5% against the 35% obtained with a 64MB cache. There is also an impact for the *lazy* mode, but it is considerable lower: it only drops to 13% from 20%. The same is applies for update operations.

Reads of comments are almost not impacted by the cache size limit: the operations have a low cache locality, so most operations need to fetch an object from the datacenter.

With a 128MB cache size limit, the *non-lazy* mode has a large portion of zero latency operations when reading links, as more link sets can be kept in the cache. It however still performs worse than Lazy fetch for operations that do not hit the cache. The latency of update operations is also improved for the *non-lazy* mode with a bigger cache, but the *lazy* mode still outperforms it for the same reasons.

#### 5.3.2 Impact on cache miss rate

The size limit imposed on the cache has an impact on the cache hit rate. Figure 4 shows that the *lazy* mode is less impacted by the cache size limit than the *non-lazy* mode. With the three cache limits, the *lazy* mode registers a rather stable number of cache misses, about 180. Nevertheless, this does not apply to the *non-lazy* mode, where the number of caches misses increases as the cache size limit is reduced. As in previous experiments, the number of cache misses is always greater in the *lazy* mode. Nevertheless, we have already proved that the latency in emphlazy mode, is always smaller in average.

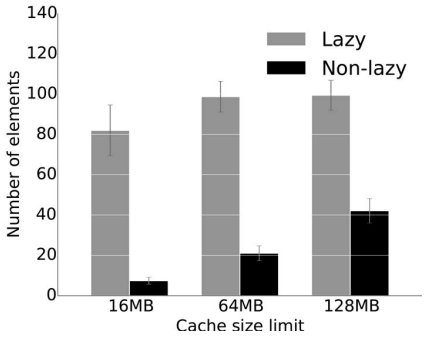



Figure 5: Number of objects kept in the cache during a benchmark with or without lazy fetch. In lazy mode objects can be *partial*, which in non-lazy mode all objects are *full* replicas.

#### 5.3.3 Impact on number of objects in the cache

Another impact of the cache size limit is the number of objects that can be kept in the cache. Notice that for partial replication, only one object is counted even if multiple parts of it have been fetched over time.

Figure 5 shows the difference between both modes: *lazy* and *non-lazy*. In the *lazy* mode, many more objects can fit in the cache at any moment, since only parts of the object are kept. 64MB is enough to keep all the objects needed by the application, while in the *non-lazy* mode, even 128MB is not enough. This leads us to determine that the *lazy* mode makes a more intelligent use of the cache, allowing more object to coexist at the same time.

### 5.4 Bandwidth usage

In order to measure the bandwidth usage, we measure the average bandwidth usage of one client over one minute, with the cache already warmed up. Figure 6 compares the *lazy* and the *non-lazy* modes. As the figure shows, the *lazy* reduces significantly the bandwidth used by a client.

### 5.5 Cache warm up

The results shown previously are taken with the cache warm. In practice, this will not always be the case. The following experiments compare both *lazy* and *non-lazy* modes latencies when the cache is still cold, i.e. no objects are stores in the client side.

Figure 7 shows the latency of operations during the first 10 seconds of running the application, with a cold cache. In this case, the *lazy* mode produces lower latencies as it does not need to replicate the full object. The difference is more noticeable for links

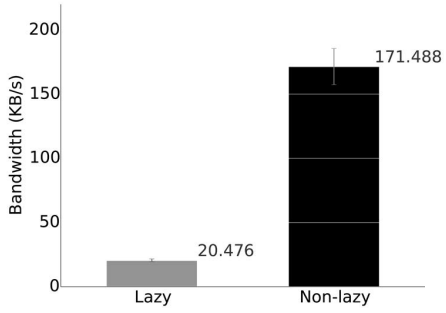
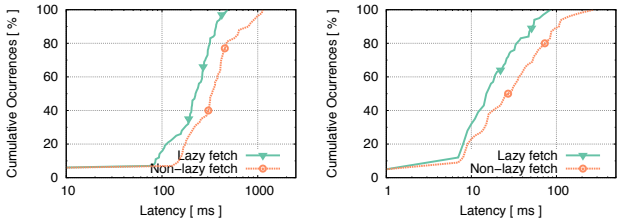



Figure 6: Average bandwidth usage to fetch objects with a 128MB cache limit, with the cache already warmed up.




(a) Reads of pages of links (b) Reads of comments of a link

Figure 7: Perceived latency of SwiftLinks at one site during cache warm up.

reading operations, as shown in Figure 7a, as the set of links are large objects. But even for smaller objects, such as comment trees, the lazy mode outperforms the *non-lazy* one (Figure 7b). It is important to notice that the cache size limit is not impacting these experiments, since after 10 seconds, the cache does not get full.

### 5.6 Discussion

We have seen that lazy fetching has advantages over full replication of objects. It puts a upper bound on the latency of operations by limiting the size that is fetched from the store.

Blind update operations gain the additional benefit of being applied locally even if the object is not in the cache.

It also limits the memory usage of the cache, which allows more objects to be kept locally even with a small cache size limit. This is useful for memory-thin devices, and to work on very large data structures with a low memory usage.

Partial replication also allows to reduce the bandwidth usage of the application by a factor of 8, which can be especially valuable on mobile wireless connections, such as EDGE or 3G.

The last advantage is a lower cost of filling the cache when starting the application. When the cache is empty all operations induce a cache miss, which is especially costly if a large object has to be fetched. Lazy fetching limits this issue by only replicating the parts of the object that are actually needed.

Unfortunately, lazy fetching has one main drawback, it limits the cache hit rate, as an object is not fully replicated right away, and non-replicated parts may be needed by subsequent operations. Therefore, the *lazy* mode should be used when the cost of a cache miss with full replication outweighs the cost of reduced number of cache hits. Nevertheless, a trade-off is possible between the two: instead of only fetching the parts needed by the operations, we could fetch more parts of the object in order to improve the cache hit rate. This would however increase bandwidth and cache size utilisation. Latency could be kept low by doing this additional fetch asynchronously, when the user is not doing any operation.

## 6. Related work

***Optimizing memory and bandwidth usage for CRDTs*** Bandwidth and space usage of CRDTs is a concern in the research community. Burckhardt et al. [8] formally calculate the space requirements for different replicated data types, such a state-based counter and a state-based set.

On the other hand, Bieniusa et al. proposed an optimization for CRDT sets that can avoid the use of tombstone by using vector clock to capture causal history [6]. Thus, the state kept by the CRDT is considerably reduce.

Finally, Almeida et al. proposed Delta-state conflict-free replicated data types [4] that allows state-based CRDT to only propagate partial states that represent recent local update instead of the whole state. While this approach improves bandwidth usage, it does not reduce the storage space for CRDTs.

***Partial replication*** There exists several prior works for partial replication. They primarily differ in the granularity of partial replication and replication criteria.

PRACTI [5] allows clients to select a subset of objects to replicate. Clients only receive updates on objects of their selected subset. However, clients are forced to keep some metadata about objects that they are not interested.

Polyjuz [22] stores objects consisting of a set of fields. Clients can decide which fields of each object to replicate. Each subset of fields is denoted as fidelity level. Clients can select different fidelity levels according to the space or network limitations of the device where the objects are replicated. Polyjuz transparently handles the replication of an object in different fidelity levels.

In Cimbiosys [14], objects are grouped into collection. Users can use filter expressions to only replicate objects that satisfy some criteria. For example, a user can group his emails in a collection and choose only to replicate emails from his university in his phone. While in the first two systems, users choose the object or fields to replicate based on their name or type, in Cimbiosys user can define replication criteria based on the value of some properties of objects.

## 7. Conclusion and future work

We have introduced and formalized a new set of CRDTs called Conflict-free Partially Replicated Data Types, an extension of CRDTs which allows replicas to hold parts of data structures. We have explained how state-based and operation-based replication mechanisms should be adapted to support partial replicas. We have also shown how to specify CPRDTs by building upon previous work. Moreover, we have given examples of CPRDTs such as a state-based grow-only set and a grow-only tree.

In order to evaluate our solution, we have integrated CPRDT into SwiftCloud, a geo-replicated storage system that replicates CRDTs on client-side in order to reduce latency. We have also implemented a Reddit-like application, called SwiftLinks, on top of the modified version of SwiftCloud. In our evaluation, we have compared three scenarios: geo-replicated storage system without caching on client-side, SwiftCloud with CRDTs and SwiftCloud with CPRDTs.

Our extensive evaluation has shown that CPRDTs can improve the bandwidth and memory usage of replicas by only replicating elements needed by clients, specially in the presence of large data structures. The experimental study has also shown that CPRDTs reduce the latency in average in comparison to the full replication scenario. However, CPRDTs have a negative impact on the cache

hit rate, which has to be weighted against the upper bound on the latency it provides. This is planned to be addressed in the future.

We plan to extend this work in several directions. Firstly, we want to evaluate CPRDTs in different scenarios. This would imply implementing different kind of applications on top. This would help us to get an even better view of its benefits and drawbacks. Secondly, as we already mentioned in the introduction of the paper, partial replication can be used as a security mechanism to avoid replicating sensitive data by restricting access with finely grained rules. We believe is an interesting way of exploiting CPRDTs. Finally, we want to study how predictive caching techniques could still improve bandwidth usage and consequently reduce latency even more.

## Acknowledgments

We thank Marek Zawirsky for his help integrating CPRDTs into SwiftCloud. This work was partially funded by the XXXX project in the European Seventh Framework Programme (FP7/2007-2013) under Grant Agreement no XXXX and by the XXXX under Grant Agreement 2012-0030.

## References

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

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

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

[4] P. S. Almeida, A. Shoker, and C. Baquero. Efficient state-based crdts by decomposition. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 3:1–3:2, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2716-9. . URL http://doi.acm.org/10.1145/2596631.2596634.

[5] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267680.1267685.

[6] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. *ArXiv e-prints*, Oct. 2012.

[7] I. Briquemont. Optimising client-side geo-replication with partially replicated data structures, Sept. 2014.

[8] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *41st Symposium on Principles of Programming Languages (POPL)*. ACM SIGPLAN, January 2014.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google’s globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, Aug. 2013. ISSN 0734-2071. . URL http://doi.acm.org/10.1145/2491245.

[10] C. Jay, M. Glencross, and R. Hubbold. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Trans. Comput.-Hum. Interact.*, 14(2), Aug. 2007. ISSN 1073-0516. . URL http://doi.acm.org/10.1145/1275511.1275514.

[11] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010. ISSN 0163-5980. . URL http://doi.acm.org/10.1145/1773912.1773922.

[12] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. . URL http://doi.acm.org/10.1145/2043556.2043593.

[13] D. Navalho, S. Duarte, N. Preguiça, and M. Shapiro. Incremental stream processing using computational conflict-free replicated data types. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, CloudDP '13, pages 31–36, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2075-7. . URL http://doi.acm.org/10.1145/2460756.2460762.

[14] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 261–276, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1558977.1558995.

[15] reddit inc. About reddit. http://www.reddit.com/about/, . Accessed: 2014-06-02.

[16] reddit inc. reddit source code. https://github.com/reddit/reddit, . Accessed: 2014-04-08.

[17] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4): 447459, Apr 1990. ISSN 00189340. .

[18] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*, June 2009.

[19] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, Jan. 2011. URL http://hal.inria.fr/inria-00555588.

[20] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Dfago, F. Petit, and V. Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-24549-7. . URL http://dx.doi.org/10.1007/978-3-642-24550-3_29.

[21] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. *Managing update conflicts in Bayou, a weakly connected replicated storage system*, volume 29. ACM, 1995. URL http://dl.acm.org/citation.cfm?id=224070.

[22] K. Veeraraghavan, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber. Fidelity-aware replication for mobile devices. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, MobiSys '09, pages 83–94, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-566-6. . URL http://doi.acm.org/10.1145/1555816.1555826.

[23] K. Veeraraghavan, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber. Fidelity-aware replication for mobile devices. In *Mobisys 2009: Proceedings of the 7th international conference on Mobile systems, applications, and services*. Association for Computing Machinery, Inc., June 2009.

[24] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009. ISSN 0001-0782. . URL http://doi.acm.org/10.1145/1435417.1435432.

[25] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. M. Preguiça. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. *CoRR*, abs/1310.3107, 2013.

### B.3 Marek Zawirski, Nuno Preguiça, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, Marc Shapiro. Write Fast, Read in the Past: Causal Consistency for Client-side Applications. Submitted EuroSys 15.

# Write Fast, Read in the Past: Causal Consistency for Client-side Applications

EuroSys 2015 Paper #174 Total length: 14 pages


## Abstract

Client-side (e.g., mobile or in-browser) apps need local access to shared cloud data, but current technologies either do not provide fault-tolerant consistency guarantees, or do not scale to high numbers of unreliable and resource-poor clients, or both. Addressing this issue, we describe the Brie distributed object database, which supports high numbers of client-side partial replicas. Brie offers fast reads and writes from a causally-consistent client-side cache. It is scalable, thanks to small and bounded metadata, and available, tolerating faults and intermittent connectivity by switching between data centres. The price to pay is a modest amount of staleness. This paper present the Brie algorithms, design, and experimental evaluation, which shows that client-side apps enjoy the same guarantees as a cloud data store, at a small cost.


## 1. Introduction

Client-side applications, such as in-browser and mobile apps, are poorly supported by the current technology for sharing mutable data over the wide-area. Existing client-side systems either make only limited consistency guarantees, or do not scale to large numbers of client devices, or both. App developers may resort to implementing their own ad-hoc application-level cache, in order to avoid slow, costly and sometimes unavailable round-trips to a data centre, but they cannot solve system issues such as fault tolerance or session guarantees [36]. Recent application frameworks such as Google Drive Realtime API [14], TouchDevelop [12] or Mobius [15] support client-side access at a small scale, but do not provide system-wide consistency and/or fault tolerance guarantees. Algorithms for geo-replication [5, 6, 19, 25, 26] or for managing database replicas on clients [10, 28] ensure some of the right properties, but were not designed to support high numbers of client replicas.

Our thesis is that the system should be responsible for ensuring correct and scalable database access to client-side applications. It should address the (somewhat conflicting) requirements of consistency, availability, and convergence [27], at least as well as geo-replication systems. Concurrent updates (which are unavoidable if updates are to be always available) should not be lost, nor cause the database to diverge permanently. Under these requirements, the strongest possible consistency model is *causal consistency* where concurrent updates to objects *converge* [25, 27].

Supporting thousands or millions of client-side replicas challenges classical assumptions. To track causality precisely, per client, creates unacceptably fat metadata; but the more compact server-side metadata management has fault-tolerance issues. Full replication at high numbers of resource-poor devices would be unacceptable [10]; but partial replication of data and metadata could cause anomalous message delivery or unavailability. It is not possible to assume, like many previous systems, that fault tolerance or consistency is solved by locating the application is located inside the data centre (DC), or has a sticky session to a single DC [7, 36].

This work addresses these challenges. We present the algorithms, design, and evaluation of Brie, the first distributed object store designed for a high number of replicas. It efficiently ensures consistent, available, and convergent access to client nodes, tolerating failures. To enable both small metadata and fault tolerance, Brie uses a flexible client-server topology, and decouples reads from writes. The client *writes fast* into the local cache, and *reads in the past* (also fast) data that is consistent, but occasionally stale. The novel aspects of our approach include:

**Cloud-backed support for partial replicas** (§3) A DC serves a consistent view of the database to the client, which the client merges with its own updates. In some failure situations, a client may connect to a DC that happens to be inconsistent with its previous DC. Because it does not have a full replica, the client cannot fix the issue on its own. We leverage "reading in the past" to avoid this situation in the common case, and provide control over the inherent trade-off between staleness and unavailability. A client observes a remote update only if it is stored in some number $K \geq 1$ of DCs [28]. The higher the value of $K$, the more likely that a $K$*-stable* version is in both DCs, but the higher the staleness.

**Protocols with decoupled, bounded metadata** (§4) Thanks to funnelling communication through DCs and to "reading in the past," our metadata design decouples *tracking causality*, which uses small vectors assigned in the background by DCs, from *unique identification*, based on client-assigned scalar timestamps. This ensures that the size of

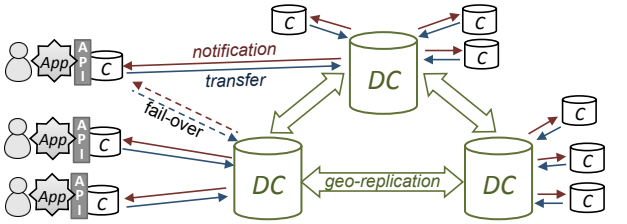



**Figure 1.** System components (Application processes, Clients, Data Centres), and their interfaces.

metadata is small and bounded. Furthermore, a DC can prune its log independently of clients, ensuring safety by storing a local summary of delivered updates.

We implement Brie and demonstrate experimentally that our design reaches its objective, at a modest staleness cost. We evaluate Brie in Amazon EC2, against a port of WaltSocial [35] and against YCSB [16]. When data is cached, response time is two orders of magnitude lower than for server-based protocols with similar availability guarantees. With three DC servers, the system can accommodate thousands of client replicas. Metadata size does not depend on the number of clients, the number of failures, or the size of the database, and increases only slightly with the number of DCs: on average, 15 bytes of metadata overhead per update, compared to kilobytes for previous algorithms with similar safety guarantees. Throughput is comparable to server-side replication, and improved for high locality workloads. When a DC fails, its clients switch to a new DC in under 1000 ms, and remain consistent. Under normal conditions, 2-stability causes fewer than 1% stale reads.

## 2. Problem overview

We consider support for a variety of client-side applications, sharing a database of **objects** that the client can `read` and `update`. We aim to scale to thousands of clients, spanning the whole internet, and to a database of arbitrary size.

Fig. 1 illustrates our system model. A cloud infrastructure connects a small set (say, tens) of geo-replicated data centres, and a large set (thousands) of clients. A DC has abundant computational, storage and network resources. Similarly to Sovran et al. [35], we abstract a DC as a powerful sequential process that hosts a **full replica** of the database.[1] DCs communicate in a peer-to-peer way. A DC may fail and recover with its persistent memory intact.

Clients do not communicate directly, but only via DCs. Normally, a client connects to a single DC; in case of failure or roaming, to zero or more. A client may fail and recover (e.g., disconnection during a flight) or permanently (e.g., destroyed phone) without prior warning. We consider only non-byzantine failures.

[1] We refer to prior work for the somewhat orthogonal issues of parallelism and fault-tolerance within a DC [5, 19, 25, 26].

Client-side apps require high **availability** and **responsiveness**, i.e., to be able to read and update data quickly and at all times. This can be achieved by replicating data locally, and by synchronising updates in the background. However, a client has limited resources; therefore, it hosts a **cache** that contains only the small subset of the database of current interest to the local app. It should not have to receive messages relative to objects that it does not currently replicate [32]. Finally, control messages and piggy-backed metadata should have small and bounded size.

Since a client replica is only *partial*, there cannot be a guarantee of complete availability. The best we can expect is **partial availability**, whereby an operation returns without remote communication if the requested data is cached; and after retrieving the data from a remote node (DC) otherwise. If the data is not there and the network is down, the operation may be unavailable, i.e., it either blocks or returns an error.

### 2.1 Consistency with convergence

Application programmers wish to observe a consistent view of the global database. However, with availability as a requirement, consistency options are limited [21, 27].

**Causal consistency** The strongest available and convergent model is causal consistency [3, 27].

Informally, under causal consistency, every process observes a *monotonically non-decreasing set of updates that includes its own updates, in an order that respects the causality between operations*.[2] Specifically, if an application process reads `x`, and later reads `y`, and if the state of `x` causally-depends on some update `u` to `y`, then the state of `y` that it reads will include update `u`. When the application requests `y`, we say there is a **causal gap** if the local replica has not yet received `u`. The system must detect such a gap, and wait until `u` is delivered before returning `y`, or avoid it in the first place. Otherwise, reads with a causal gap expose both application programmers and users to anomalies [25, 26].

We consider a transactional variant of causal consistency to support multi-object operations: all the reads of a **causal transaction** come from a same database snapshot, and either all its updates are visible as a group, or none is [8, 25, 26].

**Convergence** Another requirement is **convergence**, which consists of two properties: *(i)* **At-least-once delivery** (liveness): an update that is delivered (i.e., is visible by the app) at some node, is delivered to all (interested) nodes after a finite number of message exchanges; *(ii)* **Confluence** (safety): two nodes that delivered the same set of updates read the same value.

Causal consistency is not sufficient to guarantee confluence, as two replicas might receive the same updates in different orders. Therefore, we rely on CRDTs, high-level

[2] This subsumes the well-known session guarantees [13].

data types that guarantee confluence and have rich semantics [13, 34]. An update on a high-level object is not just an assignment, but is a high-level method associated with the object's type. For instance, a Set object supports add(element) and remove(element); a Counter supports increment() and decrement().

CRDTs include primitive last-writer-wins register (LWW) and multi-value register (MVR) [1, 18, 22], but also higher level types such as Sets, Lists, Maps, Graphs, Counters, etc. [2, 33–35]. The implementation of high-level objects is eased by adequate support from the system. For instance, an object's value may be defined not just by the last update, but also depend on earlier updates; causal consistency is helpful, by ensuring that they are not lost or delivered out of order. As high-level updates are often not idempotent (consider for instance increment()), safety also demands **at-most-once delivery**.

Although each of these requirements may seem familiar or simple in isolation, the combination with scalability to high numbers of nodes and database size is a novel challenge.

### 2.2 Metadata design

*Metadata* serves to identify updates and to ensure correctness. Metadata is piggy-backed on update messages, increasing the cost of communication.

One common metadata design assigns each update a timestamp as soon as it is generated on some originating node. The causality data structures tend to grow "fat." For instance, dependency lists [25] grow with the number of updates [19, 26, §3.3], whereas version vectors [10, 28] grow with the number of clients. (Indeed, our experiments hereafter show that their size becomes unreasonable). We call this the **Client-Assigned, Safe but Fat** approach.

An alternative delegates timestamping to a small number of DC servers [5, 19, 26]. This enables the use of small vectors, at the cost of losing some parallelism. However, this is not fault tolerant if the client does not reside in a DC. For instance, it may violate at-most-once delivery. Consider a client transmitting update $\mathsf{u}$ to be timestamped by DC1. If it does not receive an acknowledgement, it retries, say with DC2 (fail-over). This may result in $\mathsf{u}$ receiving two distinct timestamps, and being delivered twice. Duplicate delivery violates safety for many confluent types, or otherwise complicates their implementation considerably [4, 13, 26]. We call this the **Server-Assigned, Lean but Unsafe** approach.

Clearly, neither "fat" nor "unsafe" is satisfactory.

### 2.3 Causal consistency with partial replication is hard

Since a partial replica receives only a subset of the updates, and hence of metadata, it could miss some causal dependencies [10]. Consider the following example: Alice posts a photo on her wall (update $a$). Bob sees the photo and mentions in a message to Charles (update $b$), who in turn mentions it to David (update $c$). When David looks at Alice's wall, he expects to observe update $a$ and view the photo. However, if David's machine does not cache Charles' inbox, it cannot observe the causal chain $a \rightarrow b \rightarrow c$ and might incorrectly deliver $c$ without $a$. Metadata design should protect from such causal gaps, caused by transitive dependency over absent objects.

Failures complicate the picture even more. Suppose David sees Alice's photo, and posts a comment to Alice's wall (update $d$). Now a failure occurs, and David's machine fails over to a new DC. Unfortunately, the new DC has not yet received Bob's update $b$, on which comment $d$ causally depends. Therefore, it cannot deliver the comment, i.e., fulfill convergence, without violating causal consistency. David cannot read new objects from the DC for the same reason.[3]

Finally, a DC logs an individual update for only a limited amount of time, but clients may be unavailable for unlimited periods. Suppose that David's comment $d$ is accepted by the DC, but David's machine disconnects before receiving the acknowledgement. Much later, after $d$ has been executed and purged away, David's machine comes back, only to retry $d$. This could violate at-most-once delivery; some previous systems avoid this with fat version vectors [10, 28].

## 3. The Brie approach

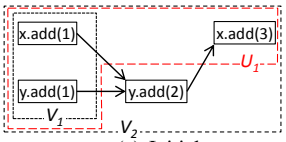
We now describe a design that addresses the above challenges, first in the failure-free case, and next, how we support DC failure.

### 3.1 Causal consistency at full DC replicas

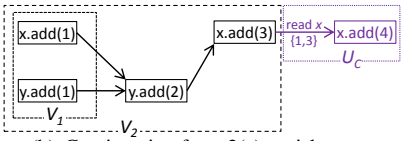
Ensuring causal consistency at fully-replicated DCs is a well-known problem [3, 19, 25, 26]. Our design is a hybrid between state-based (storing and transmitting a whole object states, called **checkpoint**) and log-based (sending and transmitting operations incrementally) [10, 30]. Hereafter, we focus on the log-based angle, and discuss checkpoints only where relevant.

A **database version**, noted $U$, is any subset of updates, ordered by causality. A version maps object identifiers to values (via the read API), by applying the relevant subsequence of the log. We say that a version $U$ has a **causal gap**, or is **inconsistent** if it is not causally-closed, i.e., if $\exists \mathsf{u}, \mathsf{u}' : \mathsf{u} \rightarrow \mathsf{u}' \wedge \mathsf{u} \notin U \wedge \mathsf{u}' \in U$. As we illustrate shortly, reading from an inconsistent version should be avoided, because, otherwise, subsequent accesses might violate causality. On the other hand, waiting for the gap to be filled would increase latency and decrease availability. To side-step this

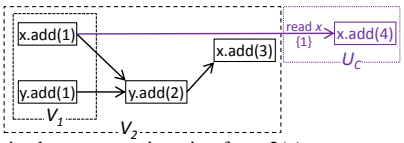
[3] Note that David can still perform updates, but they cannot be delivered. From David's perspective, writes remain available. However, the system as a whole does not converge.




(a) Initial state




(b) Continuation from 2(a) to risky state




(c) Read-in-the-past: continuation from 2(a) to conservative state

**Figure 2.** Example evolution of states for two DCs, and a client. $x$ and $y$ are Sets; box = update; arrow = causal dependence (an optional text indicates the source of dependency); dashed box = named database version/state.

conundrum, we adopt the approach of "reading in the past" [3, 25]. Thus, a DC exposes a gapless but possibly delayed state, noted $V$.

To illustrate, consider the example of Fig. 2(a). Objects $x$ and $y$ are of type Set. $DC_1$ is in state $U_1$ that includes version $V_1 \subseteq U_1$, and $DC_2$ in a later state $V_2$. Versions $V_1$ with value $[x \mapsto \{1\}, y \mapsto \{1\}]$ and $V_2$ with value $[x \mapsto \{1,3\}, y \mapsto \{1,2\}]$ are both gapless. However, version $U_1$, with value $[x \mapsto \{1,3\}, y \mapsto \{1\}]$ has a gap, missing update $\mathsf{y.add(2)}$. When a client requests to read $x$ at $DC_1$ in state $U_1$, the DC could return the most recent version, $x = \{1,3\}$. However, if the application later requests $y$, to return a safe value of $y$ requires to wait for the missing update from $DC_2$. By "reading in the past" instead, the same replica exposes the older but gapless version $V_1$, reading $x = \{1\}$. Then, the second read will be satisfied immediately with $y = \{1\}$. Once the missing update is received from $DC_2$, $DC_1$ may advance from version $V_1$ to $V_2$.

A gapless algorithm maintains a *causally-consistent, monotonically non-decreasing progression* of replica states [3]. Given an update $\mathsf{u}$, let us note $\mathsf{u}.deps$ its set of causal predecessors, called its **dependency set**. If a full replica, in some consistent state $V$, receives $\mathsf{u}$, and its dependencies are satisfied, i.e., $\mathsf{u}.deps \subseteq V$, then it applies $\mathsf{u}$. The new state is $V' = V \oplus \{\mathsf{u}\}$, where we note by $\oplus$ a **log merge operator** that filters out duplicates, further discussed in 4.1. State $V'$ is consistent, and monotonicity is respected, since $V \subseteq V'$.

If the dependencies are not met, the replica buffers $\mathsf{u}$ until the causal gap is filled.

### 3.2 Causal consistency at partial client replicas

As a client replica contains only part of the database and its metadata, this complicates consistency [10]. To avoid the complexity, we leverage the DC's full replicas to manage gapless versions for the clients.

Given some **interest set** of objects the client is interested in, its initial state consists of the projection of a DC state onto the interest set. This is a causally-consistent state, as shown in the previous section.

Client state can change either because of an update generated by the client itself, called an **internal update**, or because of one received from a DC, called **external**. An internal update obviously maintains causal consistency. If an external update arrives, without gaps, from the same DC as the previous one, it also also maintains causal consistency.

More formally, consider some recent DC state, which we will call the **base version** of the client, noted $V_{DC}$. The interest set of client $C$ is noted $O \subseteq x, y, \ldots$. The client state, noted $V_C$, is restricted to these objects. It consists of two parts. One is the projection of base version $V_{DC}$ onto its interest set, noted $V_{DC}|_O$. The other is the log of internal updates, noted $U_C$. The client state is their merge $V_C = V_{DC}|_O \oplus U_C|_O$. On cache miss, the client adds the missing object to its interest set, and fetches the object from base version $V_{DC}$, thereby extending the projection.

Base version $V_{DC}$ is a monotonically non-decreasing causal version (it might be slightly behind the actual current state of the DC due to propagation delays). By induction, internal updates can causally depend, only on internal updates, or on updates taken from the base version. Therefore, a hypothetical full version $V_{DC} \oplus U_C$ would be causally consistent. Its projection is equivalent to the client state: $(V_{DC} \oplus U_C)|_O = V_{DC}|_O \oplus U_C|_O = V_C$.

This approach ensures partial availability. If a version is in the cache, it is guaranteed causally consistent, although possibly slightly stale. If it misses in the cache, the DC returns a consistent version immediately. Furthermore, the client replica can *write fast*, because it does not wait to commit updates, but transfers them to its DC in the background.

Convergence is ensured, because the client's base version is maintained up to date by the DC, in the background.

### 3.3 Failing over: the issue with transitive causal dependency

The approach described so far assumes that a client connects to a single DC. In fact, a client can switch to a new DC at any time, in particular in response to a failure. Although each DC's state is consistent, an update that is delivered to one is not necessarily delivered in the other (because geo-replication is asynchronous, to ensure DC availability and for performance [9]), which may create a causal gap in the client.

To illustrate the problem, return to the example of Fig. 2(a). Consider two DCs: $DC_1$ is in (consistent) state $V_1$, and $DC_2$ in (consistent) state $V_2$; $DC_1$ does not include two recent updates of $V_2$. Client $C$, connected to $DC_2$, replicates object $x$ only; its state is $V_2|_{\{x\}}$. Suppose that the client reads the Set $x = \{1, 3\}$, and performs update $\mathsf{u} = \mathsf{add}(4)$, transitioning to the state shown in Fig. 2(b).

If this client now fails over to $DC_1$, and the two DCs cannot communicate, the system is not live:

(1) *Reads are not available*: $DC_1$ cannot satisfy a request for $y$, since the version read by the client is newer than the $DC_1$ version, $V_2 \not\subseteq V_1$.
(2) *Updates cannot be delivered (divergence)*: $DC_1$ cannot deliver $\mathsf{u}$, due to a missing dependency: $\mathsf{u}.deps \not\subseteq V_1$.

Therefore, $DC_1$ must reject the client to avoid creating the gap in state $V_1 \oplus U_C$.

#### 3.3.1 Conservative read: possibly stale, but safe

To avoid such gaps that cannot be satisfied, the insight is to depend on updates that are likely to be present in the failover DC, called $K$**-stable** updates.

A version $V$ is $K$-stable if every one of its updates is replicated in at least $K$ DCs, i.e., $|\{i \in \mathcal{DC} \mid V \subseteq V_i\}| \geq K$, where $K \geq 1$ is a threshold configured w.r.t. failures model. To this effect, our system maintains a consistent $K$**-stable version** $V_i^K \subseteq V_i$, which contains the updates for which $DC_i$ has received acknowledgements from at least $K - 1$ distinct other DCs.

A client's base version must be $K$-stable, i.e., $V_C = V_i^K|_O \oplus U_C|_O$, to support failover. In this way, the client depends, either on external updates that are likely to be found in any DC ($V_i^K$), or internal ones, which the client can always transfer to the new DC ($U_C$).

To illustrate, let us return to Fig. 2(a), and consider the conservative progression to Fig. 2(c), assuming $K = 2$. The client's read of $x$ returns the 2-stable version $\{1\}$, avoiding the dangerous dependency via an update on $y$. If $DC_2$ is unavailable, the client can fail over to $DC_1$, reading $y$ and propagating its update remain both live.

By the same arguments as in §3.2, a DC version $V_i^K$ is causally consistent and monotonically non-decreasing, and hence the client's version as well. Note that a client observes his internal updates immediately, even if not $K$-stable.

Parameter $K$ can be adjusted dynamically. Decreasing it has immediate effect without impacting correctness. Increasing $K$ has effect only for future updates, in order to not violate montonicity.

#### 3.3.2 Causal consistency and partial replication: discussion

The source of the problem is an indirect causal dependency on an update that the two replicas do not both know about ($\mathsf{y}.\mathsf{add}(2)$ in our example). As this is an inherent issue, we conjecture a general impossibility result, stating that genuine partial replication, causal consistency, partial availability and timely at-least-once delivery (convergence) are incompatible. Accordingly, some requirements must be relaxed.

Note that in many previous systems, this impossibility translates to a trade-off between consistency and availability on the one hand, and performance on the other [17, 25, 35] By "reading in the past," we displace this to a trade-off between freshness and availability, controlled by adjusting $K$. A higher $K$ increases availability, but updates take longer to be delivered;[4] in the limit, $K = N$ ensures complete availability, but no client can transfer a new update when some DC is unavailable.A lower $K$ improves freshness, but increases the probability that a client will not be able to fail over, and that it will block until its original DC recovers. In the limit, $K = 1$ is identical to the basic protocol from §3.2, and is similar to previous blocking session-guarantee protocols [36].

$K = 2$ is a good compromise for deplyoments with three or more DCs that covers common scenarios of a DC failure or disconnection [17, 23]. Our evaluation with $K = 2$ shows that it incurs a negligible staleness.

**Network partitions** Client failover between DCs is safe and generally live, except when the original set of $K$ DCs were partitioned away from both other DCs and the client, shortly after they delivered a version to the client. In this case, the client blocks. To side-step this unavoidable possibility, we provide an unsafe API to read inconsistent data.

When a set of fewer than $K$ DCs is partitioned from other DCs, the clients that connect to them do not deliver their updates until the partition heals. To improve liveness in this scenario, Brie supports two heuristics: *(i)* a partitioned DC announces its "isolated" status, automatically recommending clients to use another DC, and *(ii)* clients who cannot reach another DC that satisfies their dependencies can use the isolated DCs with $K$ temporarily lowered, risking unavailability if another DC fails.

## 4. Implementation

We now describe a metadata and concrete protocols implementing the abstract design.

### 4.1 Timestamps, vectors and log merge

The Brie approach requires metadata: *(1)* to *uniquely identify an update*; *(2)* to *encode its causal dependencies*; *(3)* to *identify* and *compare versions*; *(4)* and to *identify all the updates of a transaction*. We now propose a new type of metadata, which fulfils the requirements and has a low cost. It

[4] The increased number of concurrent updates that this causes is not a problem, thanks to confluent types.

combines the strengths of the two approaches outlined in Section 2.3 and is both *lean and safe*.

A timestamp is a pair $(i, k) \in (\mathcal{DC} \cup \mathcal{C}) \times \mathbb{N}$, where $i$ identifies the node that assigned the timestamp (either a DC or a client) and $k$ is a sequence number. The metadata assigned to some update u combines both: *(i)* a single **client-assigned timestamp** $\mathsf{u}.t_{\mathcal{C}}$ that uniquely identifies the update, and *(ii)* a set of zero or more **DC-assigned timestamps** $\mathsf{u}.T_{\mathcal{DC}}$. Before being delivered to a DC, the update has no DC timestamp; it has one thereafter; it may have more than one in case of delivery to multiple DCs (on failover, §3.3.1). The updates in a transaction all have the same timestamp(s), to ensure all-or-nothing delivery [35]. Our approach provides the flexibility to refer to an update via any of its timestamps, which is handy during failover.

We represent a version or a dependency as a **version vector** [29]. A vector is a partial map from node ID to integer, e.g., $VV = [DC_1 \mapsto 1, DC_2 \mapsto 2]$, which we interpret as a set of timestamps. For example, when $VV$ is used as a dependency for some update u, it means that u causally depends on $\{(DC_1, 1), (DC_2, 1), (DC_2, 2)\}$. In Brie protocols, every vector has at most one client entry, and multiple DC entries; thus, its size is bounded by the number of DCs, limiting network overhead. In contrast to a dependence graph, a vector compactly represents transitive dependencies and can be evaluated locally by any node.

Formally, the timestamps represented by a vector $VV$ are given by a function $\mathcal{T}$:

$$\mathcal{T}(VV) = \{(i, k) \in \mathsf{dom}(VV) \times \mathbb{N} \mid k \leq VV(i)\}$$

Similarly, the version decoding function $\mathcal{V}$ of vector $VV$ on a state $U$ (defined for states $U$ that cover all timestamps of $VV$) selects every update in state $U$ that matches the vector:

$$\mathcal{V}(VV, U) = \{\mathsf{u} \in U \mid (\mathsf{u}.T_{\mathcal{DC}} \cup \{\mathsf{u}.t_{\mathcal{C}}\}) \cap \mathcal{T}(VV) \neq \emptyset\}$$

For the purpose of the decoding function $\mathcal{V}$, a given update can be referred equivalently through any of its timestamps. Moreover, $\mathcal{V}$ is stable with growing state $U$.

The log merge operator $U_1 \oplus U_2$, which eliminates duplicates, is defined using client timestamps. Two updates $\mathsf{u}_1 \in U_1, \mathsf{u}_2 \in U_2$ are identical if $\mathsf{u}_1.t_{\mathcal{C}} = \mathsf{u}_2.t_{\mathcal{C}}$. The merge operator merges their DC timestamps into $\mathsf{u} \in U_1 \oplus U_2$, such that $\mathsf{u}.T_{\mathcal{DC}} = \mathsf{u}_1.T_{\mathcal{DC}} \cup \mathsf{u}_2.T_{\mathcal{DC}}$.

### 4.2 Protocols

We now describe the protocols of Brie by following the lifetime of an update, and with reference to the names in Fig. 1.

**State** A DC replica maintains its state $U_{DC}$ in durable storage. The state respects causality and atomicity for each individual object, but due to internal concurrency, this may not be true across objects. Therefore, the DC also has a vector $VV_{DC}$ that identifies a safe, monotonically non-decreasing causal version in the local state, which we note $V_{DC} = \mathcal{V}(VV_{DC}, U_{DC})$.

A client replica stores the commit log of its own updates $U_C$, and the projection of the base version from the DC, restricted to its interest set $O$, $V_{DC}|_O$, as described previously in §3.2. It also stores a copy of vector $VV_{DC}$ that describes the base version.

**Client-side execution** When the application starts a transaction $\tau$ at client $C$, the client replica initialises it with an empty buffer of updates $\tau.U = \emptyset$ and a **snapshot vector** of the current base version $\tau.depsVV = VV_{DC}$; the base version can be updated concurrently with the transaction execution. A read in transaction $\tau$ is answered from the version identified by the snapshot vector, merged with recent internal updates, $\tau.V = \mathcal{V}(\tau.depsVV, V_{DC}|_O) \oplus U_C|_O \oplus \tau.U$. If the requested object is not in the client's interest set, $o \notin O$, the clients extends its interest set, and returns the value once the DC updates the base version projection.

When the application issues internal update u, it is appended to the transaction buffer $\tau.U \leftarrow \tau.U \oplus \{\mathsf{u}\}$, and included in any later read. The transaction commits locally at the client and never fails [26].[5] If the transaction made update $\mathsf{u} \in \tau.U$, the client replica commits it locally as follows: *(1)* assign it client timestamp $\mathsf{u}.t_{\mathcal{C}} = (C, k)$, where $k$ counts the number of updates at the client; *(2)* assign it a **dependency vector** initialised with the transaction snapshot vector $\mathsf{u}.depsVV = \tau.depsVV$; *(3)* append it to the commit log of local updates on stable storage $U_C \leftarrow U_C \oplus \{\mathsf{u}\}$. This terminates the transaction; the client is now free to start a new one, which will observe the committed updates.

**Transfer protocol: Client to DC** The transfer protocol transmits committed updates from a client to its current DC, in the background. It repeatedly picks the first unacknowledged committed update u from the log. If any of u's internal dependencies has recently been assigned a DC timestamp, it merges this timestamp into the dependency vector. Then, the client sends a copy of u to its current DC. The client expects to receive an acknowledgement from the DC, containing the timestamp $T$ that the DC assigned to update u. If so, the client records the DC timestamp(s) in the original update record $\mathsf{u}.T_{\mathcal{DC}} \leftarrow T$.

It may now iterate with the next update in the log.

A transfer request may fail for three reasons:

(a) Timeout: the DC is suspected unavailable; the client connects to another DC (failover) and repeats the protocol.

[5] To simplify the notation, and without loss of generality, we assume hereafter that a transaction performs at most one update. This is easily extended to multiple updates, by assigning the same timestamp to all the updates of the same transaction, ensuring the all-or-nothing property [35].

(b) The DC reports a *missing internal dependency*, i.e., it has not received some update of the client, as a result of a previous failover. The client recovers by marking as unacknowledged all internal updates starting from the oldest missing dependency, and restarting the transfer protocol from that point.
(c) The DC reports a *missing external dependency*; this is also an effect of failover. In this case, the client tries yet another DC. The approach from §3.3.1 avoids repeated failures.

Upon receiving update $\mathsf{u}$, the DC verifies if it dependencies are satisfied, i.e., if $\mathcal{T}(\mathsf{u}.depsVV) \subseteq \mathcal{T}(VV_{DC})$. (If this check fails, it reports an error to the client, indicating either case (b) or (c)). If the DC has not received this update previously, i.e., $\forall \mathsf{u}' \in U_{DC} : \mathsf{u}'.t_{\mathcal{C}} \neq \mathsf{u}.t_{\mathcal{C}}$, the DC does the following: *(1)* Assign it a DC timestamp $\mathsf{u}.T_{\mathcal{DC}} \leftarrow \{(DC, VV_{DC}(DC) + 1))\}$, *(2)* store it in its durable state $U_{DC} \oplus \{\mathsf{u}\}$, *(3)* make the update visible in the DC version $V_{DC}$, by incorporating its timestamp(s) into $VV_{DC}$. This last step makes $\mathsf{u}$ available to the geo-replication and notification protocols, described hereafter. If the update has been received before, the DC looks up its previously-assigned DC timestamps, $\mathsf{u}.T_{\mathcal{DC}}$. In either case, the DC acknowledges the transfer to the client with the DC timestamp(s). Note that steps (1)–(2) can be parallelised between transfer requests received from different client replicas.

**Geo-replication protocol: DC to DC** The geo-replication protocol consists of a uniform reliable broadcast across DCs. An update enters the geo-replication protocol when a DC accepts a fresh update during the transfer protocol. The accepting DC broadcasts it to all other DCs. A DC that receives a broadcast message containing $\mathsf{u}$ does the following: (1) If the dependencies of $\mathsf{u}$ are not met, i.e., if $\mathcal{T}(\mathsf{u}.depsVV) \not\subseteq \mathcal{T}(VV_{DC})$, buffer it until they are; and (2) incorporate $\mathsf{u}$ into durable state $U_{DC} \oplus \{\mathsf{u}\}$ (if $\mathsf{u}$ is not fresh, the duplicate-resilient log merge safely unions all timestamps), and incorporate its timestamp(s) into the DC version vector $VV_{DC}$. This last step makes it available to the notification protocol. The $K$-stable version $V^K_{DC}$ is computed similarly.

**Notification protocol: DC to Client** A DC maintains a best-effort *notification* session, over a FIFO channel, to each of its connected clients. The soft state of a session includes a copy of the client's interest set $O$ and the last known base version vector used by the client, $VV_{DC}{}'$. The DC accepts a new session only if its own state is consistent with the base version of the client, i.e., if $\mathcal{T}(VV_{DC}{}') \subseteq \mathcal{T}(VV_{DC})$. Otherwise, the DC would cause a causal gap with the client's state; in this case, the client is redirected to another DC (see §3.3.1).

The DC sends over each channel a causal stream of update notifications.[6] Notifications are batched according to either time or to rate [10]. A notification packet consists of a new base version vector $VV_{DC}$, and a sequence of log of all the updates $U_\Delta$ to the objects of the interest set, between the client's previous base vector $VV_{DC}{}'$ and the new one. Formally, $U_\Delta = \{\mathsf{u} \in U_{DC}|_O \mid \mathsf{u}.T_{\mathcal{DC}} \cap (\mathcal{T}(VV_{DC}) \setminus \mathcal{T}(VV_{DC}{}')) \neq \emptyset\}$. The client applies the newly-received updates to its local state, described by the old base version: $V_{DC}|_O \leftarrow V_{DC}|_O \oplus U_\Delta$, and assumes the new vector $VV_{DC}$. If any of received updates is a duplicate w.r.t. to the old version or to a local update, the log merge operator handles it safely.

When the client detects a broken channel, it reinitiates the session, possibly on a new DC.

The interest set can change dynamically. When an object is evicted from the cache, the notifications are lazily unsubscribed to save resources. When it is extended with object $o$, the DC responds with the current version of $o$, which includes all updates to $o$ up to the base version vector. To avoid races, a notification includes a hash of the interest set, which the client checks.

### 4.3 Object checkpoints and log pruning

Update logs contribute to substantial storage and, to smaller extent, network costs. To avoid unbounded growth, **pruning protocol** prediocially replaces the prefix of a log and by a *checkpoint*. In the common case, a checkpoint is more compact than the corresponding log of updates; for instance, a log containing one thousand increments to a Counter object and their timestamps, can be replaced by a checkpoint containing just the number 1000 and a version vector.

#### 4.3.1 Log pruning in the DC

The log at a DC provides *(a)* unique timestamp identification of each update, which serves to filter out duplicates by $\oplus$ operator, as explained earlier, and *(b)* the capability to compute different versions, for application processes reading at different causal times. Update $\mathsf{u}$ is expendable once all of its duplicates have been filtered out, and once $\mathsf{u}$ has been delivered to all interested application processes. However, evaluating expendability precisely would require access to the client replica states.

In practice, we need to prune *aggressively*, but still avoid the above issues, as we explain next.

In order to reduce the risk of pruning a version not yet delivered to an interested application (which could force it to restart an ongoing transaction), we prune only a **delayed version** $VV_i^\Delta$, where $\Delta$ is a real-time delay [25, 26].

To avoid duplicates, we extend our DC local metadata as follows. $DC_i$ maintains an **at-most-once guard** $G_i$, which

[6] Alternatively, the client can ask for invalidations instead, trading responsiveness for lower bandwidth utilization and higher DC throughput.

| | **YCSB** [16] | **SocialApp** [35] |
|---|---|---|
| Type of objects | LWW Map | Set, Counter, Register |
| Object payload | $10 \times 100$ bytes | variable |
| Read txns | read fields (A: 50% / B: 95%) | read wall (80%) see friends (8%) |
| Update txns | update field (A: 50% / B:5%) | message (5%) post status (5%) add friend (2%) |
| Objects / txn | 1 (non-txnal) | 2–5 |
| Database size | 50,000 objects | 400,000 objects |
| Object popularity | uniform / zipfian | uniform |
| Session locality | 40% (low) / 80% (high) | |

**Table 1.** Characteristics of applications/workloads.

records the sequence number of each client's last pruned update $G_i : \mathcal{C} \to \mathbb{N}$. Whenever the DC receives a transfer request for update u with timestamp $(C, k) = \mathsf{u}.t_{\mathcal{C}}$ and cannot find it in its log, it checks the at-most-once guard entry whether u is contained in the checkpoint. If the update was already pruned away ($G_i(C) \geq k$), the update is ignored; the DC discarded information about the exact set of update's DC timestamps in this case; therefore, in transfer reply, they are overapproximated by vector $VV_i^{\Delta}$. Similarly, on a client cache miss, the DC sends object state that consists of the most recent checkpoint of the object together the client's guard entry, so that the client can merge it with his updates safely. Note that a guard is local to and shared at a DC. It is *never* fully transmitted.

#### 4.3.2 Pruning the client's log

Managing the log at a client is comparatively simpler. A client logs his own updates $U_C$, which may include updates to object that is currently out of his interest set. This enables the client to read its own updates, and to propagate them lazily to a DC when connected. An update u can be discarded as soon as it appears in $K$-stable base version $V_i^K$, i.e., when the client becomes dependent on the presence of u at a DC.

## 5. Evaluation

We implement Brie and evaluate it experimentally, in comparison to other protocols. In particular, we show that Brie provides: *(i)* fast response, under 1 ms for both reads and writes to cached objects (§5.3); *(ii)* scalability of throughput with the number of DCs, and small metadata size, linear in the number of DCs (§5.4); *(iii)* fault-tolerance w.r.t. client churn (§5.5) and DC failures (§5.6); and *(iv)* modest staleness cost, under 3% of stale reads (§5.7).

### 5.1 Implementation and applications

Brie and the benchmark applications are implemented in Java. Brie uses a library of CRDT types, BerkeleyDB for durable storage (turned off in the present experiments), and Kryo for data marshalling. A client cache has a fixed size and uses an LRU eviction policy.

Our client API resembles modern object stores, such as Riak 2.0, Redis, or COPS [2, 25, 31]:

| | |
|---|---|
| `begin_transaction ()` | `read (object) : value` |
| `commit_transaction ()` | `update(object, method(args...))` |

Along the lines of previous studies of weak consistency [5, 6, 26, 35], we use two different benchmarks, YCSB and SocialApp, summarized in Table 1.

YCSB [16] serves as a kind of micro-benchmark, with simple requirements, measuring baseline costs and specific system properties in isolation. It has a simple key-field-value object model, implemented as a LWW Map type, using a default payload size of ten fields of 100 bytes each. YCSB issues single-object reads and writes. We use two of the standard YCSB workloads: update-heavy Workload A, and read-dominated Workload B. The object access pattern can be set to either uniform or Zipfian. YCSB does not rely on transactional semantics or high-level data types.

SocialApp is a social network application modelled closely after WaltSocial [35]. It employs high-level data types such as Sets, for friends and posts, LWW Register for profile information, Counter for counting profile visits, and inter-object references. SocialApp accesses multiple objects in a causal transaction to ensure that operations such as reading a wall page and profile information behave consistently. The SocialApp workload is read-dominated, but the ostensibly read-only operation of visiting a wall actually increments the wall visit counter. The access distribution is uniform.

In order to model the locality behaviour of a client, both YCSB and SocialApp are augmented with a facility to control locality, mimicking social network access patterns [11]. Within a client session, the application draws draws uniformly from a pool of session-specific objects with either 40% (*low locality*) or 80% (*high locality*) probability. Objects not drawn from this local pool are drawn from the global (uniform or zipfian) distribution described above. The size of the pool is smaller than the size of cache.

### 5.2 Experimental setup

We run three DCs in geographically distributed Amazon EC2 availability zones (Europe, Virginia, and Oregon), and a pool of distributed clients. Round-Trip Times (RTTs) between nodes are as follows:

| | Oregon DC | Virginia DC | Europe DC |
|---|---|---|---|
| nearby clients | 60–80 ms | 60–80 ms | 60–80 ms |
| Europe DC | 177 ms | 80 ms | |
| Virginia DC | 60 ms | | |

Each DC runs on a single m3.m EC2 instance, equivalent to a single core 64-bit 2.0 GHz Intel Xeon virtual processor (2 ECUs) with 3.75 GB of RAM, and OpenJDK7 on Linux 3.2. Objects are pruned at random intervals between 60–120 s, to avoid bursts of pruning activity. We deploy 500–

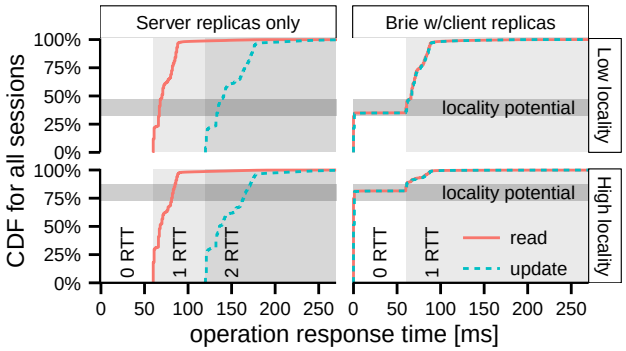



**Figure 3.** Response time for YCSB operations (workload A, zipfian object popularity) under different system and workload locality configurations.

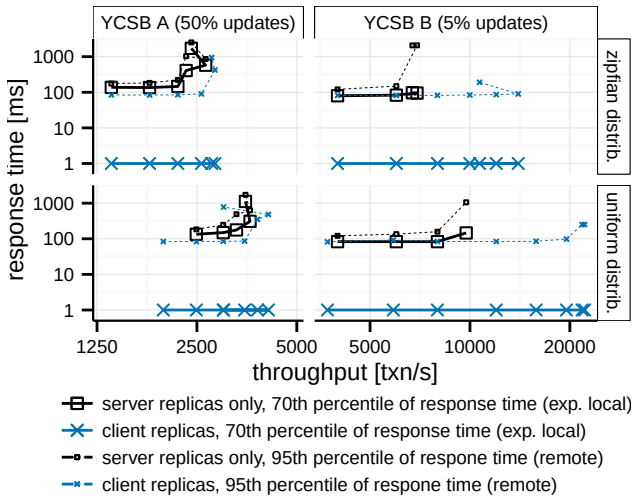



**Figure 4.** Throughput vs. response time for different system configurations running variants of YCSB.

2,500 clients on a separate pool of 90 m3.m EC2 instances. Clients load DCs uniformly and use the closest DC by default, with a client-DC RTT ranging in 60–80 ms.

For comparison, we provide three protocol modes based on the Brie implementation: *(i) Brie mode* (default) with client cache replicas of 256 objects, and refreshed with notifications at a rate $\leq$1 s by default; *(ii) Safe But Fat metadata mode* with cache, but with client-assigned metadata (similarly to PRACTI, or to Depot without cryptography), *(iii) server-side replication mode* without client caches. In this mode, an update incurs two RTTs to a DC, modelling the cost of a synchronous writes to a quorum of servers to ensure fault-tolerance comparable to Brie.

### 5.3 Response time and throughput

We run several experiments to compare Brie's client-side caching, with server-only geo-replication.

Fig. 3 shows response times for YCSB, comparing server-only (left side) with client replication (right side), under low (top) and high locality (bottom). Recall that in server-only replication, a read incurs a RTT to the DC, whereas an update incurs 2 RTTs. We expect Brie to provide much faster response, at least for cached data. Indeed, the figure shows that a significant fraction of operations respond immediately in Brie mode, and this fraction tracks the locality of the workload (marked "locality potential" on the figure), within a $\pm$7.5 percentage-point margin, attributable to caching artefacts. The remaining operations require one round-trip to the DC, indicated as 1 RTT. As our measurements for SocialApp show the same message, we do not report them here. These results demonstrate that the consistency guarantees and the rich programming interface of Brie do not affect responsiveness of read and update caching.

In terms of throughput, client-side replication is a mixed blessing: it lets client replicas absorb read requests that would otherwise reach the DC, but also puts extra load of maintaining client replicas on DCs. In another experiment (not plotted), we saturate the system to determine its maximum throughput. Brie's client-side replication consistently improving throughput for high-locality workloads, by 7% up to 128%. It is especially beneficial to read-heavy workloads. In contrast, low-locality workloads show no clear trend; depending on the workload, throughput either increases by up to 38%, or decrease by up to 11% with Brie.

Our next experiment studies how response times vary with server load and with the staleness settings. The results show that, as expected, cached objects respond immediately and are always available, but the responsiveness of cache misses depends on server load. For this study, Fig. 4 plots throughput vs. response time, for YCSB A (left side) and B (right side), both for the Zipfian (top) and uniform (bottom) distributions. Each point represents the aggregated throughput and latency for a given transaction incoming rate, which we increase until reaching the saturation point.

The curves report two percentiles of response time: the lower (70 th percentile) line represents the response time for requests that hit in the cache (the session locality level is 80%), whereas the higher (95 th percentile) line represents misses, i.e., requests served by a DC.

As expected, the lower (cached) percentile consistently outperforms the server-side baseline, for all workloads and transaction rates. A separate analysis, not reported in detail here, reveals that a saturated DC slows down its rate of notifications, increasing staleness, but this does not impact response time, as desired. In contrast, the higher percentile follows the trend of server-side replication response time, increasing remote access time.

Varying the target notification rate (not plotted) between 500 ms and 1000 ms, reveals the same trend: response time is not affected by the increased staleness. At a lower refresh rate, notification batches are less frequent but larger. This increases throughput for the update-heavy Workload A (up to tens of percent points), but has no effect on the throughput

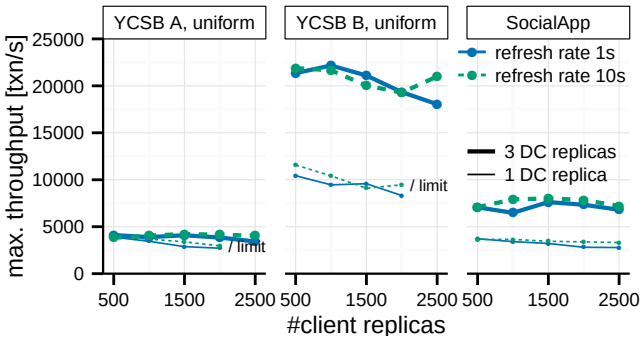



**Figure 5.** Maximum system throughput for a variable number of client and server (DC) replicas.

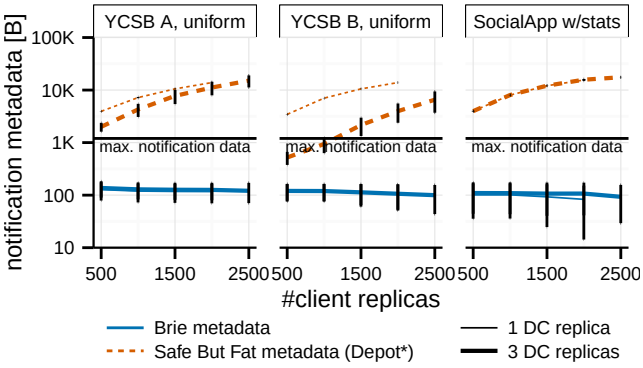



**Figure 6.** Size of metadata in notification message for a variable number of replicas, mean and standard error. Normalised to a notification of 10 updates.

of read-heavy Workload B. However, we expect the impact of refresh rate to be amplified for workloads with lower rate of notification updates.

### 5.4 Scalability

Next, we measure how well Brie scales with increasing numbers of DC and of client replicas. Of course, performance is expected to increase with more DCs, but most importantly, the size of metadata should be small, should increase only marginally with the number of DCs, and should not depend on the number of clients. Our results support these expectations.

In this experiment, we run execute Brie with a variable number of client (500–2500) and server (1–3) replicas. We report only on the uniform object distribution, because under the Zipfian distribution different numbers of clients skew the load differently, making any comparison meaningless. To control staleness, we run Brie with two different notification rates (every 1 s and every 10 s).

Fig. 5 shows the maximum system throughput on the Y axis, increasing the number of replicas along the X axis. The thin lines are for a single DC, the bold ones for three DCs. Solid lines represent the fast notification rate, dashed lines the slow one. The figure shows, left to right, YCSB Workload A, YCSB Workload B, and SocialApp.

The capacity of a single DC in our hardware configuration peaks at 2,000 active client replicas for YCSB, and 2,500 for SocialApp.

As to be expected, additional DC replicas increase the system capacity for operations that can be performed at only one replica such as read operations or sending notification messages. Whereas a single Brie DC supports at most 2,000 clients. With three DCs Brie supports at least 2,500 clients for all workloads. Unfortunately, as we ran out of resources for client machines at this point, we cannot report an upper bound.

For some fixed number of DCs, adding client replicas increases the aggregated system throughput, until a point where the cost of maintaining client replicas up to date saturates the DCs, and further clients do not absorb enough reads to overcome these costs. Note that the lower refresh rate can reduce the load at a DC by 5 to 15%.

In the same experiment, Fig. 6 presents the distribution of metadata size notification messages. (Notifications are the most common and the most costly messages sent over the network.) We plot the size of metadata (in bytes) on the Y axis, varying the number of clients along the X axis. Left to right, the same workloads as in the previous figure. Thin lines are for one DC, thick lines for three DCs. A solid line represents Brie "Lean and Safe" metadata, and dotted lines the classical "Safe But Fat" approach. Note that our Safe-but-Fat implementation includes the optimisation of sending vector deltas rather than the full vector [28]. Vertical bars represent standard error. As notifications are batched, we normalise metadata size to a message carrying exactly 10 updates, corresponding to under approx. 1 KB of data.

This plot confirms that the Brie metadata is small and constant, at 100–150 bytes/notification (10–15 bytes per update); data plus metadata together fit inside a single standard network packet. It is independent both from the number of client replicas and from the workload. Increasing the number of DC replicas from one to three causes a negligible increase in metadata size, of under 10 bytes.

In contrast, the classical metadata grows linearly with the number of clients and exhibits higher variability. Its size reaches approx. 1 KB for 1,000 clients in all workloads, and 10 KB for 2,500 clients. Clearly, metadata being up to 10× larger than the actual data this represents a substantial overhead.

### 5.5 Tolerating client churn

We now turn to fault tolerance. In the next experiment, we evaluate Brie under client churn, by periodically disconnecting client replicas and replacing them with a new set of active clients. At any point in time, there are 500 active clients and a variable number of disconnected clients, up to 5000.

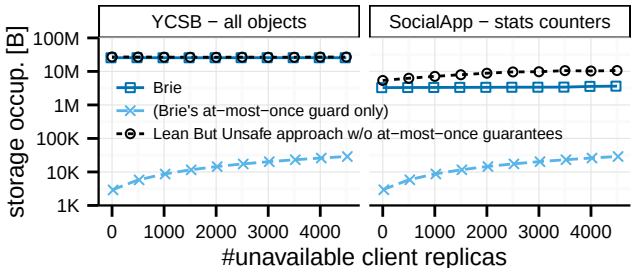



**Figure 7.** Storage occupation at a single DC in reaction to client churn for Brie and Lean-but-Unsafe alternative.

Fig. 7 illustrates the storage occupation of a DC for representative workloads. We compare Brie's log pruning protocol to a protocol without at-most-once delivery guarantees (Lean But Unsafe).

Brie storage size is approximately constant. This is safe thanks to the at-most-once guard table per DC. Although the size of the guard (bottom curve) grows with the number of clients, it requires orders of less storage than the actual database itself.

A protocol without at-most-once delivery guarantees can use Lean-but-Unsafe metadata, without Brie's at-most-once guard. However this requires more complexity in each object's implementation, to protect itself from duplicates. This increases the size of objects, impacting both storage and network costs. As is visible in the figure, the cost depends on the object type: none for idempotent YCSB's LWW-Map, which is naturally idempotent, vs. linear in the number of clients for SocialApp's Counter objects.

### 5.6 Tolerating DC failures

This experiment studies the behaviour of Brie when a DC disconnects. The scatterplot in Fig. 8 shows the response time of a SocialApp client application as the client switches between DCs. Starting with a cold cache, response times quickly drops to near zero for transactions hitting in the cache, and to around 110 ms for misses. Some 33 s into the experiment, the current DC disconnects, and the client is diverted to another DC in a different continent. Thanks to $K$-stability the fail-over succeeds, and the client continues with the new DC. Its response time pattern reflects the higher RTT to the new DC. At 64 s, the client switches back the initial DC, and performance smoothly recovers to the initial pattern.

### 5.7 Staleness cost

The price to pay for our read-in-the-past approach is an increase in staleness, which our next experiment measures. A read is considered *stale* if a version more recent (but not $K$-stable) than the one it returns exists at the current DC of a client that performed the read. A transaction is stale if any of its reads is stale. In the experiments so far, we observed a

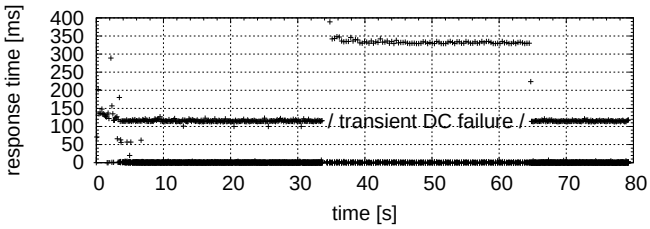



**Figure 8.** Response time for a client that hands over between DCs during a 30 s failure of a DC.

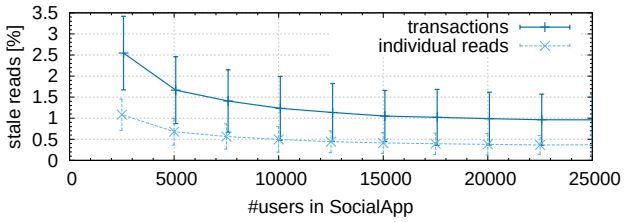



**Figure 9.** $K$-stability staleness overhead.

negligible number of stale reads. The reason is that the window of vulnerability (the time it takes for an update to become $K$-stable) is approximately the RTT to the closest DC. For this experiment, we artificially increase the probability of staleness by various means. We run the SocialApp benchmark with 1000 clients in Europe connected to the Ireland DC and replicated in the Oregon DC.

Fig. 9 shows that stale reads and stale transactions remain under $1\%$ and $2.5\%$ respectively. This shows that even under high contention, accessing a slightly stale snapshot has very little impact on the data read by transactions.

## 6. Related work

We now discuss a number of systems that support consistent, available and convergent data access, at different scales. In particular, Table 2 presents the approach to metadata of *causally* consistent systems. Each row groups some systems that share a similar metadata approach. The columns indicate: *(i)* Which replicas assign timestamps; *(ii)* the guaranteed (worst-case) size of metadata summarising a dependency or a version; *(iii)* whether it ensures at-most-once delivery; *(iv)* whether it supports general confluent types.

**Client-side replication** PRACTI is a seminal work on causal consistency under partial replication [10]. PRACTI uses Safe-but-Fat client-assigned metadata and an ingenious log-exchange protocol that supports an arbitrary communication topology. While such a full generality has advantages, it is not viable for large-scale client-side app deployments backed by the cloud: *(i)* Its fat metadata approach (version vectors sized as the number of clients) is prohibitively expensive (see Fig. 6), and *(ii)* any replica can easily make another unavailable, because of the indirect dependence issue discussed in §3.3.2.

| Representative system | Timestamp assignment | Summary metadata max. size, $O$(#entries) | At-most-once delivery | Support for confluent types |
|---|---|---|---|---|
| PRACTI [10], Depot [28], COPS [25] | client/any replica | #replicas $\approx$ 1 000 000 | yes | weak (COPS) / medium (rest) |
| Eiger [26], Orbe [19], Bolt-on [6] | DC server (shard) | #servers $\approx$ 100–1 000 | no | weak |
| Walter [35], ChainReaction [5] | DC (full replica) | #DCs $\approx$ 5–10 | no | weak |
| Brie | DC (full replica)<br>client replica | #DCs $\approx$ 5–10<br>+ 1 client entry | no<br>yes | strong |

**Table 2.** Analytical comparison of different classes of metadata used by causally consistent systems.

Our design is strongly inspired by Depot, a (fork-join) causally consistent system that provides a reliable storage on top of untrusted cloud [28]. Depot tolerates Byzantine clients, which our current implementation does not address. Their assumption of Byzantine cloud behaviour requires fat metadata to support direct client-to-client communication. Furthermore, Depot is at at odds with genuine partial replication. It requires every replica to process the metadata of every update, and puts the burden of computing a $K$-stable version on the client. In the case of extensive DC partitions, it floods all updates to the client. In contrast, Brie relies on DCs to provide $K$-stable and consistent versions, and uses lean metadata. In the event of failure, Brie provides the flexibility to decrease $K$ dynamically rather than to flood clients.

Both Practi and Depot systems use Safe-but-Fat metadata. They support only LWW Registers, but extension to other confluent types appears feasible.

Recent web and mobile application frameworks, such as TouchDevelop [12], Google Drive Realtime API [14], or Mobius [15] support replication for in-browser or mobile applications. These systems are designed for small objects [14], database that fits on a mobile device [12], or a database of independent objects [15]. It is unknown if/how they support multiple DCs and fault tolerance. This is in contrast with Brie's support for large consistent database, and fault tolerance. TouchDevelop provides a form of object composition, and offers integration with strong consistency [12]. We are looking into ways of adapting similar mechanisms.

**Server-side replication** A number of geo-replicated systems offer available causally consistent data access inside a DC with excellent scale-out by sharding [5, 6, 19, 25, 26].

Table 2 shows that server-side systems use variety of types of metadata. COPS assigns metadata directly at database clients, and uses explicit dependencies (a graph) [25]. Later publications show that this approach is costly [19, 26]. Consequently, later systems assign metadata at partition replicas [19, 26], or on a designated node in the DC [5, 35]. The location of assignment directly impacts the size of causality metadata. In most systems, it varies with the number of reads, with the number of dependencies, and with the stability conditions in the system. When fewer nodes assign metadata, it tends to be smaller (as in Brie), but this may limit throughput.

Previous designs are not directly applicable to client-side replication, because: *(i)* their protocols do not tolerate client or server failures; *(ii)* as they assume that data is updated by overwriting, implementing high-level confluent data types is complex and costly (see Fig. 7); *(iii)* the size of their metadata can grow uncontrollably.

Du et al. [20] make use of full stability, a special kase of $K$-stability, to remove the need for dependency metadata in messages, thereby improving throughput.

**Integration with strong consistency** Some operations or objects of application may require stronger consistency, which requires synchronous protocols [21]. For instance, we observe that our social network application port would benefit from strongly consistent support for user registration or a password change. Prior work demonstrates that combining strong and weak consistency is possible on shared data [24, 35]. We speculate that these techniques are applicable to Brie, grounded on preliminary experience.

**Theoretical limits** Mahajan et al. [27] prove that causal consistency is the strongest achievable model in an available, convergent, *full* replication system. We conjecture that these properties are not simultaneously achievable under partial replication, and demonstrate how to weaken one of the liveness properties. Bailis et al. [7] give an argument for a similar result for a client switching server replicas, but do not take into account the capabilities of a client replica.

## 7. Conclusion

We presented the design of Brie, the first system that offers client-side apps a local access to partial database replica with the guarantees of geo-replicated systems.

Our experiments confirm that Brie is able to provide immediate and consistent response on reads and updates on local objects, and maintain the throughput of a server-side replication system, or better. The novel form of metadata allows the system to scale to thousands of clients with constant size objects and metadata, independent of the number of available and unavailable clients. Our fault-tolerant protocols handle failures nearly transparently.

Many of these properties are due to a common principle demonstrated by Brie design: client buffering and controlled staleness can absorb the cost of scalability, availability, and consistency. Staleness cost is moderate and well separated.

## References


[1] Riak, 2010. http://basho.com/riak/.

[2] Introducing Riak 2.0: Data types, strong consistency, full-text search, and much more, Oct. 2013. http://basho.com/introducing-riak-2-0/.

[3] M. Ahamad, J. E. Burns, P. W. Hutto, et al. Causal memory. In *Proc. 5th Int. Workshop on Distributed Algorithms*, pp. 9–30, Delphi, Greece, Oct. 1991.

[4] P. S. Almeida and C. Baquero. Scalable eventually consistent counters over unreliable networks. Number 1307.3207, July 2013.

[5] S. Almeida, J. Leitão, and L. Rodrigues. ChainReaction: a causal+ consistent datastore based on Chain Replication. In *Euro. Conf. on Comp. Sys. (EuroSys)*, Apr. 2013.

[6] P. Bailis, A. Ghodsi, J. M. Hellerstein, et al. Bolt-on causal consistency. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pp. 761–772, New York, NY, USA, 2013.

[7] P. Bailis, A. Davidson, A. Fekete, et al. Highly Available Transactions: Virtues and limitations. In *Int. Conf. on Very Large Data Bases (VLDB)*, Riva del Garda, Trento, Italy, 2014.

[8] P. Bailis, A. Fekete, A. Ghodsi, et al. Scalable atomic visibility with RAMP transactions. In *ACM SIGMOD Conference*, 2014.

[9] P. Bailis, A. Fekete, M. J. Franklin, et al. Coordination avoidance in database systems. In *Int. Conf. on Very Large Data Bases (VLDB)*, Kohala Coast, Hawaii, 2015. To appear.

[10] N. Belaramani, M. Dahlin, L. Gao, et al. PRACTI replication. In *Networked Sys. Design and Implem. (NSDI)*, pp. 59–72, San Jose, CA, USA, May 2006.

[11] F. Benevenuto, T. Rodrigues, M. Cha, et al. Characterizing user behavior in online social networks. In *Internet Measurement Conference (IMC)*, 2009.

[12] S. Burckhardt. Bringing TouchDevelop to the cloud. Inside Microsoft Research Blog, Oct. 2013. http://blogs.technet.com/b/inside_microsoft_research/archive/2013/10/28/bringing-touchdevelop-to-the-cloud.aspx.

[13] S. Burckhardt, A. Gotsman, H. Yang, et al. Replicated data types: Specification, verification, optimality. In *Symp. on Principles of Prog. Lang. (POPL)*, pp. 271–284, San Diego, CA, USA, Jan. 2014.

[14] B. Cairns. Build collaborative apps with Google Drive Realtime API. Google Apps Developers Blog, Mar. 2013. http://googleappsdeveloper.blogspot.com/2013/03/build-collaborative-apps-with-google.html.

[15] B.-G. Chun, C. Curino, R. Sears, et al. Mobius: Unified messaging and data serving for mobile apps. In *Int. Conf. on Mobile Sys., Apps. and Services (MobiSys)*, pp. 141–154, New York, NY, USA, 2012.

[16] B. F. Cooper, A. Silberstein, E. Tam, et al. Benchmarking cloud serving systems with YCSB. In *Symp. on Cloud Computing*, pp. 143–154, Indianapolis, IN, USA, 2010.

[17] J. C. Corbett, J. Dean, M. Epstein, et al. Spanner: Google's globally-distributed database. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pp. 251–264, Hollywood, CA, USA, Oct. 2012.

[18] G. DeCandia, D. Hastorun, M. Jampani, et al. Dynamo: Amazon's highly available key-value store. In *Symp. on Op. Sys. Principles (SOSP)*, volume 41 of *Operating Systems Review*, pp. 205–220, Stevenson, Washington, USA, Oct. 2007.

[19] J. Du, S. Elnikety, A. Roy, et al. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Symp. on Cloud Computing*, pp. 11:1–11:14, Santa Clara, CA, USA, Oct. 2013.

[20] J. Du, C. Iorgulescu, A. Roy, et al. Closing the performance gap between causal consistency and eventual consistency. In *Workshop on Principles and Practice of Eventual Consistency (PaPEC)*, Amsterdam, Netherland, 2014.

[21] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. ISSN 0163-5700.

[22] P. R. Johnson and R. H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, Jan. 1976.

[23] A. Kansal, B. Urgaonkar, and S. Govindan. Using dark fiber to displace diesel generators. In *Hot Topics in Operating Systems*, Santa Ana Pueblo, NM, USA, 2013.

[24] C. Li, D. Porto, A. Clement, et al. Making geo-replicated systems fast as possible, consistent when necessary. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pp. 265–278, Hollywood, CA, USA, Oct. 2012.

[25] W. Lloyd, M. J. Freedman, M. Kaminsky, et al. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Symp. on Op. Sys. Principles (SOSP)*, pp. 401–416, Cascais, Portugal, Oct. 2011.

[26] W. Lloyd, M. J. Freedman, M. Kaminsky, et al. Stronger semantics for low-latency geo-replicated storage. In *Networked Sys. Design and Implem. (NSDI)*, pp. 313–328, Lombard, IL, USA, Apr. 2013.

[27] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. Technical Report UTCS TR-11-22, Dept. of Comp. Sc., The U. of Texas at Austin, Austin, TX, USA, 2011.

[28] P. Mahajan, S. Setty, S. Lee, et al. Depot: Cloud storage with minimal trust. *Trans. on Computer Systems*, 29(4):12:1–12:38, Dec. 2011.

[29] J. Parker, D.S., G. J. Popek, G. Rudisin, et al. Detection of mutual inconsistency in distributed systems. *IEEE Trans. on Soft. Engin.*, SE-9(3):240–247, May 1983.

[30] K. Petersen, M. J. Spreitzer, D. B. Terry, et al. Flexible update propagation for weakly consistent replication. In *Symp. on Op. Sys. Principles (SOSP)*, pp. 288–301, Saint Malo, Oct. 1997.

[31] Redis. Redis is an open source, BSD licensed, advanced key-value store. http://redis.io, May 2014.

[32] N. Schiper, P. Sutra, and F. Pedone. P-Store: Genuine partial replication in wide area networks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pp. 214–224, New Dehli, India, Oct. 2010.

[33] M. Shapiro, N. Preguiça, C. Baquero, et al. A comprehensive study of Convergent and Commutative Replicated Data Types. Number 7506, Rocquencourt, France, Jan. 2011.

[34] M. Shapiro, N. Preguiça, C. Baquero, et al. Conflict-free replicated data types. In *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pp. 386–400, Grenoble, France, Oct. 2011.

[35] Y. Sovran, R. Power, M. K. Aguilera, et al. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*, pp. 385–400, Cascais, Portugal, Oct. 2011.

[36] D. B. Terry, A. J. Demers, K. Petersen, et al. Session guarantees for weakly consistent replicated data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*, pp. 140–149, Austin, Texas, USA, Sept. 1994.

### B.4 Valter Balegas, Mahsa Najafzadeh, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Marc Shapiro, Nuno Preguiça. Extending Eventually Consistent Cloud Stores for Enforcing Numeric Invariants. Internal technical report.

# Extending Eventually Consistent Cloud Stores for Enforcing Numeric Invariants

(Research paper / primary author is a student)

Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça
CITI/FCT/Universidade Nova de Lisboa

Marc Shapiro, Mahsa Najafzadeh
INRIA / LIP6

## ABSTRACT

Geo-replication based on eventually consistent data stores is a widely used mechanism for improving the user experience in Internet services. Furthermore, recent work on commutative data types allows for these storage systems to provide seamless reconciliation for special purpose data types, such as counters. However, important limitations of eventual consistency still need to be addressed by the applications themselves, namely maintaining numeric invariants across all replicas.

In this paper, we present a solution to support numeric invariants in geo-replicated databases under eventual consistency guarantees, and discuss alternative middleware designs to extend existing cloud stores with support for the enforcement of numeric invariants. Our approach borrows ideas from escrow transactions, but through several novel concepts, we are able to make them decentralized, fault-tolerant and fast. The solution is supported by a new CRDT, the bounded counter, that maintains the necessary information for enforcing numeric invariants in eventual consistent data stores, and a middleware that can be layered on top of existing systems, enriching them with numeric invariants. We used Riak, a production data store, as the use case to test the feasibility of our solution.

Our evaluation shows that our designs can enforce numeric invariants with lower latency and higher scalability that existing solutions than rely on some form of strong consistency and successfully reduces the tension between consistency and availability.

## 1 Introduction

Scalable storage systems with a simple interface providing an extended version of a key/value store have emerged as the platform of choice for providing online services that operate on a global scale, such as Facebook, Amazon, or Yahoo! [9, 11, 15].

In this context, a common technique for improving the user experience is geo-replication [9, 11, 16, 18, 19, 26], i.e., maintaining copies of application data and logic in multiple data centers scattered across the globe. This decreases the latency for handling user requests by routing them to nearby data centers, but at the expense of resorting to weaker data consistency guarantees, which avoid costly replica coordination for executing operations. When executing under such weaker consistency models, applications have to deal with concurrent operations executing without being aware of each other, which implies that a merge strategy is required for reconciling concurrent updates. A common approach is to rely on a *last-writer-wins* strategy [15, 18, 19], but this strategy is not appropriate in all situations. A prominent example is the proper handling of counters, which are not only increasingly part of the interface of widely used storage systems [2, 3], but also a useful abstraction for implementing features such as *like* buttons, votes and ad and page views. In the case of counters, using a *last-writer-wins* strategy would lead to lost updates, and therefore breaking the intended semantics. To address this limitation, various systems added support for counters with a merge strategy specific to this data type. In particular, Cassandra now supports counters [2], DynamoDB has native support for atomic counters, and Riak (an open-source NoSQL storage system used by 25% of the Furtune 50 [1]) recently introduced support for conflict-free data types (CRDT) [23], including counters [3] and sets.

Even though these approaches provide a principled handling of concurrent updates to counter objects, they fall short on supporting the enforcement of crucial invariants or database integrity constraints, which are often required for maintaining correct operation [16]. To give a real-world example, our collaboration with a large game development company, whose games were downloaded over a billion times, led us to understand that they require a precise limit on the number of times an ad is impressed [private communication]. However, since their systems are built on top of Riak, which only supports weakly consistent counters, they are not able to directly enforce that condition. This is because counter updates can occur concurrently, and therefore it is not possible to detect if the limit is exceeded before the operation concludes.

Maintaining this type of invariants would be trivial in systems that offer strong consistency guarantees, namely those that serialize all updates, and therefore preclude that two operations execute without seeing the effects of one another [10, 16]. The problem with these systems is that they require coordination among replicas, leading to an increased latency, which, in a geo-replicated scenario, may amount to hundreds of milliseconds, with the consequent impact on application usability [12, 22].

In this paper we show that it is possible to achieve the best of both worlds, i.e., that fast geo-replicated operations on counters can coexist with strong invariants. To this end, we propose a novel abstract data type called a *Bounded Counter*. This replicated ob-

ject, like conventional CRDTs [23], allows for operations to be executed locally, automatically merges concurrent updates, and, in contrast to previous CRDTs, also enforces numeric invariants while avoiding any coordination in most cases. Implementing *Bounded Counter* in fast and portable way required overcoming a series of challenges, which form the main technical contributions of this work.

First, we extend some of the ideas behind escrow transactions [20], which partition the difference between the current value of a counter and the limit to be enforced among existing replicas, who can locally execute operations that do not exceed their allocated part. Unlike previous solutions that include some central authority [20, 21, 24] and are often based on synchronous interactions between nodes, our approach is completely decentralized and asynchronous, relying on maintaining the necessary information for enforcing the invariant in a new CRDT – the *Bounded Counter* CRDT. This allows for replicas to synchronize peer-to-peer and asynchronously, thus minimizing the deployment requirements and avoiding situations where the temporary unreachability of the master data center can prevent operations from making progress

Second, we present two middleware designs for extending existing cloud stores with support for enforcing numeric invariants using the *Bounded Counter* CRDT. While the first design is implemented using only a client-side library, the second includes server side components deployed in a distributed hash table. Both designs require only that the underlying cloud store executes operations sequentially in each replica (not necessarily by the same order across replicas) and that it provides a reconciliation mechanism that allows for merging concurrent updates. This makes our solutions generic and portable, but raise significant challenges in terms of their design, mostly for achieving performance comparable with accessing directly to the underlying cloud store. We discuss how to deploy our middleware designs on eventually consistent cloud stores and present and evaluate two prototypes that run on top of Riak.

The evaluation of our prototypes shows that: 1. when compared to using weak consistency, our approach exhibits similar latency, while guaranteeing that invariants are not broken; 2. when compared to using a strong consistency model, our approach can enforce invariants without incurring in long latency for coordination among replicas; 3. the client library design performs well under low contention, but does not scale when contention on the same counter is large. 4. the server based middleware design scales well horizontally, providing higher throughput than weak consistency by relying in a set of techniques to minimize the number of operations executed in the underlying storage system.

The remaining of the paper is organized as follows: Section 2 presents the proposed model; Section 3 introduces the *Bounded Counter* CRDT; Section 4 discusses general requirements for using *Bounded Counters* and Section 5 presents two middleware designs that extend Riak with numeric invariant preservation; Section 6 discuss extensions to the proposed solution and Section 7 evaluates our prototypes; Section 8 discusses related work; and Section 9 concludes the paper with some final remarks.

# 2 System Overview

In this section we present an overview of the solution proposed in this paper for providing bounded counters to application servers running in geo-replicated settings.

```
% Regular data operations
get( key): object | fail
put( key, object): ok | fail

% Bounded Counters operations
create(key, type, bound): ok | error
read(key): integer | error
inc(key, delta, flag): ok | fail | retry
dec(key, delta, flag): ok | fail | retry
```

**Figure 1: System API.**

## 2.1 Assumptions

We assume a typical geo-replicated scenario, with copies of application data and logic maintained in multiple data centers scattered across the globe. End clients contact the closest data center for executing application operations in the application server running in that data center. The execution of this application logic leads to issuing a sequence of operations on the data storage system, where application data resides.

The design of *Bounded Counter* only requires very weak assumptions for its correctness to hold. In particular, we consider that system processes (or nodes) are connected by an asynchronous network (i.e., subject to arbitrary delays, including partitions). We assume a finite set $\Pi = p_0, p_1, \ldots, p_{n-1}$ of processes who may fail by crashing. A crashed process may remain crashed forever, or may recover with its persistent memory intact. A non-crashed process is said to be *correct*.

*Bounded Counters* can by layered on top of any weakly consistent storage system, with the only requirement that each replica serializes all operations that it receives, though different replicas can serialize the operations in a different order. Furthermore, the underlying storage system must provide a reconciliation mechanism that allows for merging concurrent updates.

For simplicity, our presentation considers a single data object replicated in all processes of $\Pi$, with $r_i$ representing the replica of the object at process $p_i$. The model trivially generalizes to the case where multiple data objects exist – in such case, for each object $o$, there is a set $\Pi^o$ of the processes that replicate $o$. For each object $o$, we need to consider only the set $\Pi^o$. The main challenge in this case is how to handle invariants involving the sum of various counters. We discuss how to solve this in Section 6.

## 2.2 System API

Our middleware system is built on top of a key-value store. Figure 1 summarizes the programming interface of the system, with the usual *get* and *put* operations for accessing regular data, and additional operation for creating a new *Bounded Counter*, reading its current state, and incrementing or decrementing its value. As any other data, bounded counters are identified in all operations by an application-defined opaque key. Our goal is to ensure that these counters are able to maintain a numeric invariant, while also allowing operations to execute by contacting a single single (local) data center.

The *create* operation creates a new bounded counter. The *type* argument specifies if it is an *upper-* or a *lower-* bounded counter, and the *bound* argument provides the global invariant limit to be maintained – e.g., *create( "X", upper, 1000)* creates a bounded counter that maintains the invariant that the value must be *smaller or equal to 1000*. The counter is initialized to the value of the bound.

The *read* operation returns the current value of the given counter. Since the returned value is computed based on local information

with respect to the underlying data store, it may not be globally accurate. To update a counter, the application submits *inc* or *dec* operations. These operations execute on the local replica and, if they succeed, it is guaranteed that the numeric global invariant of the counter is preserved and its value remains within the allowed bounds. Conversely, *inc* and *dec* operations fail when the global invariant forbids the local execution of the offending operation. In such cases, the runtime provides a hint to the application regarding the possibility of sucessfully executing the operation if other replicas are contacted. These operations also include a *flag* that allows applications to request that the system contacts other replicas to try to successfully execute the operation before reporting a failure.

## 2.3 Consistency Guarantees

The proposed solution provides an extended eventual consistency model that guarantees invariant preservation for counters.

In particular, the eventual consistency guarantee means that the outcome of each operation reflects the effects of only a subset of the operations that all clients have previously invoked – these are the operations that have already been executed by the replica that the client has contacted. However, for each operation that successfully returns at a client, there is a point in time after which its effect becomes visible to every operation that is invoked after that time, i.e., operations are eventually executed by all replicas.

In terms of the invariant preservation guarantee, this means precisely that the bounds on the counter value are never violated, neither *locally* nor *globally*. By locally, this means that the bounds must be obeyed when taking into account the subset of the operations reflected by an operation that client invokes. In other words, the subset of operations seen by the replica where each operation executes must obey the following equation:

$$\text{lower bound} \leq \text{initial value} + \sum inc - \sum dec \leq \text{upper bound}.$$

By globally, this means that, at any instant in the execution of the system, when considering the union of all the operations that have been executed at all sites, the same bound must hold.

Note that the notion of causality is orthogonal to our design and guarantees, in the sense that if the underlying storage system that we build upon offered causal consistency, then we would also provide numeric invariant-preserving causal consistency.

## 2.4 Solution Overview

To implement the API defined in the previous subsection, our solution borrows ideas from the escrow transactional model [20]. The key idea of this model is to consider that the difference between the value of a counter and its bound can be seen as a set of rights to execute operations. For example, in a counter, $n$, with initial value $n = 40$ and invariant $n \geq 10$, there are 30 $(40-10)$ rights to execute decrement operations. Executing *dec(5)* consumes 5 of these rights. Executing *inc(5)* creates 5 rights. These rights can be split among the replicas of the counter – e.g. if there are 3 replicas, each replica can be assigned 10 rights. If the rights needed to execute some operation exist in the local replica, the operation can execute safely locally, knowing that the global invariant will not be broken – in the previous example, if the decrements of each replica are less or equal to 10, it follows immediately that the total decrements are at most 30 and the invariant still holds. If not enough rights exist, then either the operation fails or additional rights must be obtained from other replicas.

Our solution encompasses two components that work together to achieve the goal of our system: a novel data structure, the *Bounded Counter* CRDT, to maintain the necessary information for locally verifying whether it is safe to execute an operation or not; and a middleware layer to store and update instances of this data structure in the underlying eventually consistent cloud store. The first component is detailed in section 3, while alternative designs to the second part are detailed in section 4.

# 3 Design of Bounded Counter CRDT

This section discusses the design of *Bounded Counter*, a CRDT that enforces numeric invariants without requiring coordination during most operation executions. Instead, coordination is normally executed outside of the normal execution flow of an operation.

## 3.1 CRDT basics

Conflict-free replicated data types (CRDTs) [23] are a class of distributed data types that allow replicas to be modified without coordination while guaranteeing that replicas converge to the same correct value after all updates are propagated and executed in all replicas.

Two types of CRDTs have been defined: *operation-based CRDTs*, where modifications are propagated as operations (or patches) and executed on every replica; and *state-based CRDTs*, where modifications are propagated as states, and merged with the state of every replica.

In the design of CRDTs, a client of the object may invoke an operation at some replica of its choice, which is called the *source* replica of the operation. Operations are split into *queries* and *updates*. A query reads the state of the object and executes entirely at the source. In turn, an update is split into two functions: a *prepare* and a *downstream* function (similarly to the generator and shadow operations in RedBlue consistency [16]). The *prepare* function has no side-effects and executes only at the source replica. Its goal is to identify the changes that must be performed based on the current CRDT state, which are encapsulated in the *downstream* function. This function applies the identified changes to the CRDT state. At the source replica, the *prepare* and *downstream* functions execute atomically in isolation with respect to other operations.

Given this basic design, the operation and state-based designs can be distinguished as follows. In the operation-based mode, the *downstream* function is propagated and eventually executed in all replicas, whereas in the state-based mode, the *downstream* only executes at the source replica, and its replication in implicitly achieved through pairwise replica synchronization: replica $r_i$ incorporates the effects of all operations executed by some other replica $r_j$ by merging its state with the state of the remote replica (by executing the *merge* function).

It has been proven that a sufficient condition for guaranteeing the convergence of an operation-based CRDT is that all replicas execute all operations and that all operations commute [23].

To define similar conditions for state-based CRDTs, we need to introduce some definitions. A join semi-lattice (or just semi-lattice) is a partial order $\leq$ equipped with a *least upper bound* (LUB) $\sqcup$ for all pairs: $m = x \sqcup y$ is a Least Upper Bound of $\{x,y\}$ under $\leq$ iff $x \leq m \wedge y \leq m \wedge \forall m', x \leq m' \wedge y \leq m' \Rightarrow m \leq m'$.

Given these definitions, a sufficient condition for guaranteeing that all replicas of a state-based CRDT converge is that the object conforms the properties of a monotonic semi-lattice object [23], in which: *(i)* The set $S$ of possible states forms a semi-lattice ordered by $\leq$. *(ii)* The result of merging state $s$ with remote state $s'$ is the result of computing the LUB of the two states in the semi-lattice of state, i.e., $merge(s,s') = s \sqcup s'$. *(iii)* State is monotonically non-decreasing across updates, i.e., for any update $u$, $s \leq u(s)$.

As a large number of cloud stores synchronize their replicas by propagating the state of the database objects, it was natural to design *Bounded Counter* as a state-based CRDT. However, when compared with state-based synchronization, the biggest benefit of operation-based synchronization is that the communication cost for synchronizing the state when one operation is executed might be smaller – depending on the size of the operation and the object state. In our case, as the state of a *Bounded Counter* is $O(n^2)$ (where $n$ is the number of data centers), and additionally supporting operation-based synchronization involved no relevant complexity, we decided to bring together both synchronization models and design *Bounded Counter* as a mixed state- and operation-based CRDT.

For an object in the mixed state- and object-based model to converge, the sufficient conditions encompass not only both the conditions for the state- and the operation-based models, but are also augmented with the requisite of idempotence of operations. Even though the formalization of the mixed model and of these sufficient conditions is outside of the scope of this paper, we provide an intuition for the reason why these conditions are sufficient. When considering the semi-lattice formed by the possible states of a CRDT, executing a new operation does an inflation by moving the current state of a replica to a new state higher in the semi-lattice. Thus, each node of the semi-lattice corresponds to a state that reflects the execution of a set of operations. In the state-based model, the merging of states reflecting concurrent operations can be done in any order, with the state resulting from all merges being the single LUB of all states. For achieving the same state when executing operations, concurrent operations need to commute, to guarantee that the same state is achieved independently of the execution order. Finally, idempotence for operation execution is necessary as executing an operation to a state that already reflects that operation must have no side-effects – this follows immediately from the fact that the LUB of two values where one of the values is greater than the other is the greatest value.

## 3.2 Bounded Counter CRDT

Next, we detail the design of a *Bounded Counter* for maintaining the invariant *larger or equal to K*. The pseudocode for this design is presented in Figure 2.

*Bounded Counter state.* The *Bounded Counter* must maintain the necessary information to verify whether it is safe to locally execute operations or not. As our approach is inspired in the escrow transactional model [20], as discussed in section 2.4, this information consists of the rights that each replica holds.

To maintain this information in a way that makes it simple to merge the state of two replicas, each replica maintains two data structures: $R$, with information about the available rights; and $U$, with the used rights. Given that $n$ replicas exist, $R$ is a matrix of $n$ lines by $n$ columns, with one line and one column for each replica; $U$ is a vector with $n$ lines, i.e., one entry for each replica.

The line for replica $r_i$ maintains the following information: $R[i][i]$ records the *increments* executed at source replica $r_i$, which define an equal number of rights initially assigned to replica $r_i$; $R[i][j]$ records the rights transfered from replica $r_i$ to replica $r_j$; $U[i]$ records the successful *decrements* executed at source replica $r_i$, which consume an equal number of rights.

*Operations.* When a counter is created, we assume that the initial value of the counter is equal to the minimum value allowed by the invariant, $K$. Thus, no rights are assigned to any replica and both $R$ and $U$ are initialized with all entries being equal to 0. To overcome the limiting assumption of the initial value being $K$, we can immediately execute *increment* operations in the freshly created *Bounded Counter*. Figure 3 shows an example of the state of a *Bounded Counter* for maintaining the invariant *larger or equal to 10*, with initial value 40. This initial value led to the creation of 30 rights assigned to replica $r_0$ – this value is recorded in $R[0][0]$.

The *increment* prepare function records the identifier of the source replica and the new value for the sum of increments of that replica. The downstream function just updates the respective entry of the $R$ matrix with the new value. The use of *max* in the downstream function trivially guarantees both the commutativity and idempotence of the operation, which is required for convergence. In the example of Figure 3, the value of $R[1][1]$ is 1, which is the result of incrementing by 1 the counter in replica $r_1$.

For decrementing the counter, it is necessary to verify if the source replica $r_{id}$ has enough rights to execute the operation. This is achieved by computing the local rights (*localRights*) by adding to the increment operations executed in the local replica, $R[id][id]$, the rights transferred from other replicas to the source replica, $R[i][id], \forall i \neq id$, and subtracting the rights transferred by the source replica to other replicas, $R[id][i], \forall i \neq id$.

The *decrement* operation fails if not enough local rights exist –

payload integer[$n$][$n$] $R$, integer[$n$] $U$, integer *min*
  initial [[0,0,...,0], ..., [0,0,...,0]], [0,0,...,0], $K$

query *value* () : integer $v$
  let $v = min + \sum_{i \in Ids} R[i][i] - \sum_{i \in Ids} U[i]$

query *localRights* () : integer $v$
  let $id = repId()$ **%Id of the local replica**
  let $v = R[id][id] + \sum_{i \neq id} R[i][id] - \sum_{i \neq id} R[id][i] - U[id]$

update *increment* (integer $n$)
  prepare ($n$)
    let $id = repId()$
    let $nv := R[id][id] + n$
  effect ($id, nv$)
    let $R[id][id] := max(R[id][id], nv)$

update *decrement* (integer $n$)
  pre $localRights() \geq n$
  prepare ($n$)
    let $id = repId()$
    let $nu := U[id] + n$
  effect ($id, nu$)
    let $U[id] := max(U[id], nu)$

update *transfer* (integer $n$, replicaId *to*): boolean $b$
  pre $localRights() \geq n$
  prepare ($n$)
    let $from = repId()$
    let $nv := R[from][to] + n$
  effect ($from, to, nv$)
    let $R[from][to] := max(R[from][to], nv)$

update *merge* (X,Y): payload $Z$
  let $Z.P[i][j] = max(X.P[i][j], Y.P[i][j]), \forall i, j \in Ids$
  let $Z.U[i] = max(X.R[i], Y.R[i]), \forall i \in Ids$

**Figure 2: *Bounded Counter* CRDT for maintaining the invariant *larger or equal to* $K$.**

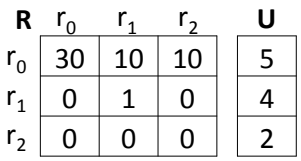



**Figure 3: Example of the state of *Bounded Counter* for maintaining the invariant *larger or equal to 10*.**

we discuss later how this failure can be handled. Otherwise, the prepare records the identifier of the source replica and the new value for the sum of decrement operations of that replica. The downstream operation just updates the respective entry of the $R$ matrix with the new value. In the example of Figure 3, the values of $U$ reflect the execution of 5, 4 and 2 decrements in replicas $r_0$, $r_1$ and $r_2$, respectively.

The operation to retrieve the current *value* consists of adding to the minimum value, $K$, the sum of the increment operations, recorded in $R[i][i], \forall i$, and subtracting the sum of the decrement operations, recorded in $U[i], \forall i$. In the example of Figure 3, the current value is 30 (obtained from $10+(30+1)-(5+4+2)$).

One replica, $r_i$, may *transfer* to another replica, $r_j$, rights to execute decrements. This is achieved by recording that information in the $R$ matrix, namely by updating the entry $R[i][j]$ – as in *increment* and *decrement*, the prepare operation records the necessary information and the downstream updates the data. In the example of Figure 3, transfers of 10 rights from $r_0$ to each of $r_1$ and $r_2$ are recorded in the values of $R[0][1]$ and $R[0][2]$

*Replica synchronization.* Our choice of a mixed operation and state-based design implies that replicas can be updated both by executing downstream functions or by synchronizing its state with other replicas, by using the *merge* operation. This dual synchronization model allows the system to use the most appropriate synchronization mechanism at each moment. For example, for keeping replicas closely synchronized, replicas may be updated by propagating downstream operations using some best-effort communication mechanism. However, when a fault is detected, or when a replica becomes partitioned for a while, the replica state is synchronized from other replica by doing state-based synchronization, which is more efficient when replicas have diverged significantly.

*Correctness.* For showing the correctness of *Bounded Counter*, it is necessary to show that all replicas of *Bounded Counter* eventually converge to the same state and that the execution of concurrent operations will not break the invariant. We now informally show that these properties are satisfied.

For showing that replicas eventually converge to the same state, we can show that the specification satisfies the requirements for the mixed state- and operation-based CRDTs. Regarding the operation-based part, it is necessary to guarantee that concurrent downstream functions commute and are idempotent. The use of *max* when updating the elements of $R$ (in *increment* and *merge*) or $U$ (in *decrement*) trivially guarantees both commutativity and idempotence, as the maximum value will be stored after applying all operations in any order. The use of *max* even allows for optimizing operation propagation, as only the latest operation that modifies some element needs to be propagated.

Regarding state, it is necessary to prove that the specification is a monotonic semi-lattice object. As the elements of $R$ and $U$ are monotonically increasing (since operations never decrement the value of these variables), the semi-lattice properties are immediately satisfied – two states, $s_0, s_1$, are related by a partial order relation, $s_0 \leq s_1$, whenever all values of $R$ and $U$ in $s_1$ are greater or equal to the corresponding values in $s_0$ (i.e., $\forall i, j, R_0[i][j] \leq R_1[i][j] \wedge U_0[i] \leq U_1[i]$).

To guarantee that the invariant is not broken, it is necessary to guarantee that a replica does not execute an operation (*decrement* or *transfer*) without holding enough rights to do it. As operations verify if the local replica holds enough rights before execution, it is necessary to prove that if a replica believe it has $N$ rights, it owns at least $N$ rights. The operations guarantee that line $i$ of $R$ and $U$ is only updated by operations with source replica $r_i$. As the downstream function executes immediately at replica $r_i$, replica $r_i$ necessarily has the most recent value for line $i$ of both $R$ and $U$. As rights of replica $r_i$ are consumed by *decrement* operations, recorded in $U[i]$, and transfer operations, recorded in $R[i][j]$, it follows immediately that replica $r_i$ knows of all rights it has consumed. Thus, when computing the local rights, the value computed locally is always conservative (as replica $r_i$ may not know yet of some transfer to $r_i$ executed by some other replica). This guarantees that the invariant is not broken when operations execute locally in a single replica.

*Extensions.* It is possible to define a *Bounded Counter* that enforces an invariant of the form *smaller or equal to K* by using a similar approach, where rights represent the possibility of executing *increment* operations instead of *decrement* operations. The specification would be similar to the one presented in Figure 2, with the necessary adaptations to the different meaning of the rights.

A *Bounded Counter* that can maintain an invariant of the form *larger or equal to* $K_0$ and *smaller or equal to* $K_1$ can be created by combining the information of two *Bounded Counter*s, one for each invariant, and updating both on each operation.

## 3.3 Transferring Rights

For being able to guarantee the local execution of an operation that may violate an invariant in a given replica, it is necessary that the replica has enough rights. Given that it is impossible to anticipate the rights needed by each operation, it is necessary to provide a mechanism for exchanging rights between replicas. Our *Bounded Counter* provides the *transfer* operation for this purpose.

The are two possible strategies for executing this operation: it can be used to exchange rights proactively, to maintain a similar level (or an expected distribution) of rights among replicas; or on-demand, by fetching rights whenever they are necessary at some replica.

In our prototype, we have implemented both strategies for exchanging rights. However, the use of on-demand transfers is optional and controlled by the programmer: these can be activated by setting the *flag* parameter in the *decrement* operation, as shown in the system API presented in Figure 1. When the flag is set, and if the local replica $r_i$ does not hold enough rights to guarantee the execution of a *decrement* operation, the system tries to obtain additional rights from other replicas. To this end, replica $r_i$ contacts some other replica $r_j$ to request the execution of a *transfer* operation at $r_j$. If the transfer operation executes successfully, replica $r_i$ also synchronizes with replica $r_j$. When enough rights are gathered by replica $r_i$, the *decrement* operation executes locally. If not enough rights can be gathered, an error is reported to the appli-

cation – either *fail*, if, according to the local state, there are not enough rights in all replicas to execute the operation; or *retry*, otherwise.

For deciding which replica to contact, we can leverage the state of *Bounded Counter*. Although the information about rights of other replicas is not precise, by targeting the replicas that are believed to have more available rights, we expect a high probability of having a successful transfer.

A property of the way *transfer* is implemented is that it does not require any strong synchronization between the replica asking for rights and the one providing the rights. Thus, the request for a transfer and synchronization of the information about transferred values can be done completely asynchronously, which simplifies the system design.

# 4 Middleware Requirements

Next, we discuss how to layer *Bounded Counter* on top of an existing storage system. In our design, we can achieve this while only requiring two properties from the underlying cloud store. The first is to be able to execute operations in isolation at each replica. The second requirement is to support a replication model with no lost updates, either by relying on the execution of all operations in all replicas, or by synchronizing replicas through the execution of a merge procedure. We analyze each requirement independently next.

## 4.1 Isolation for operation execution

Our design requires, both at the source replica (which executes prepare and downstream functions) and at downstream replicas (which execute the downstream function), that operations execute atomically under strong isolation. This means that a function can never observe an intermediate state of other functions, i.e., it either sees all the effects of another function or none at all. A possible way to achieve this is to execute operations sequentially at each replica (even if they execute in a different order at different replicas).

Some storage systems provide an API that allows applications to define arbitrary operations that execute in isolation in the replicas – e.g., Gemini [16] and Bayou [27]. Such systems satisfy immediately the isolation requirement.

Other systems provide a conditional write operation where a write fails depending on some condition that is evaluated when the write executes – e.g., PNUTS [9], Walter [26], DynamoDB [11] and Riak [8]. This functionality can be used to provide the isolation requirement necessary for the *Bounded Counter* by reading the counter object and writing a modified version only if the counter has not been modified since it has been read.

## 4.2 Replication with no lost updates

A large number of cloud storage systems provide replication solutions with no lost updates. In systems that propagate operations among replicas (e.g., Gemini, Bayou) this is immediate – in such systems, we could deploy *Bounded Counters* by executing the downstream function in all replicas.

In systems that propagate the state of objects among replicas, some of them support only a *last writer wins* policy that can lead to lost updates – e.g., Cassandra [15]. For such systems, it is not possible to easily deploy *Bounded Counters*. However, most cloud stores provide support for merging concurrent updates, either by exposing the concurrent versions to the applications – e.g. Dynamo [11], Riak [8] – or by automatically applying application-defined merge procedures when concurrent updates are detected – e.g. COPS [18]. In such systems, deploying *Bounded Counter* is immediate, since it allows the merge operation defined for the Bounded Counter CRDT to be used for merging concurrent versions.

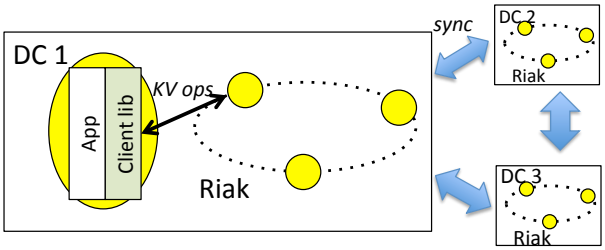



**Figure 4: Client-based middleware for deploying *Bounded Counters*.**

# 5 Extending Riak with Bounded Counters

We now discuss how we have extended the Riak cloud database to include *Bounded Counters* using a middleware solution. We start with an overview of the functionalities of Riak that are relevant for the deployment of *Bounded Counters* and then discuss two alternative designs for the deployment. We conclude this subsection by discussing how the proposed solutions could be used with other cloud stores.

## 5.1 Overview of Riak 2.0

Riak is a key/value store inspired in Dynamo [11], built on top of a distributed hash table (DHT). It provides an API supporting a read and a write operation. A write associates a new value with a key, and a read returns the value(s) associated with the key.

Riak supports geo-replication in its Enterprise Edition by deploying a Riak DHT in each of the data centers. Data centers are kept synchronized using two mechanisms: (1) a continuous synchronization mechanism that propagates entries (i.e., key/value pairs) modified in one data center to the other data centers; (2) a periodic synchronization mechanism that synchronizes all entries of one data center with some other data center [1]. Riak handles concurrent updates by keeping multiple versions and exposing concurrent writes in read operations (similarly to how write/write conflicts are handled in a single data center).

The latest version of Riak introduces a conditional writing mode where a write fails if a concurrent update has been executed. This operation mode, dubbed strong consistency, is currently implemented using a primary/backup solution and works only in a single data center. Riak also includes native support for storing CRDTs, dubbed Riak data types. We have not relied on this mechanism for deploying *Bounded Counter*, as *Bounded Counter* would require modfying Riak to combine its data types with conditional writes, which was not supported in the version we were using. Thus, we have implemented our solution as a middleware layer between the client application and the Riak database.

## 5.2 Alternative 1: Client-based middleware

Our first design is based on a client-side middleware, as depicted in Figure 4. Supporting operations on *Bounded Counters* is fairly

[1]The periodic synchronization mechanism relies on computing Merkle trees for computing a delta between the state of the two sites efficiently.

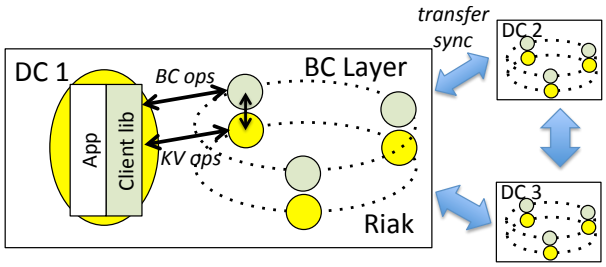



**Figure 5: Server-based middleware for deploying *Bounded Counters*.**

simple, given the functionality provided by Riak.

The state of a *Bounded Counter* CRDT is stored as an opaque object in the Riak database. Rights for executing operations in a *Bounded Counter* are associated with each data center, i.e., each data center is considered as a single replica for a *Bounded Counter*. An increment (resp. decrement) executes in the client library by first reading the current value of the counter (executing a *get* operation in Riak), then executing the increment (resp. decrement) operation in the *Bounded Counter* CRDT and writing the new value of the counter back into the database using conditional writing. If the operation in the CRDT fails, the client can try to obtain additional rights by requesting the execution of a *transfer* operation from another data center. If the operation in the CRDT succeeds but the conditional write fails, the operation must be re-executed until it succeeds.

Given that updates are serialized in each data center through the conditional writing mechanism, concurrent updates to the same *Bounded Counter* can only appear due to geo-replication. If this is the case, then concurrent versions can be merged by the client library when reading the counter.

Any solution that propagates the updated values among data centers could be used for geo-replication, since the only requirement is that the solution detects and exposes concurrent versions. As the current version of Riak does not support multi-data center replication for objects that use strong consistency, we had to implement a custom synchronization mechanism for *Bounded Counters* (while other objects rely on normal Riak replication). This custom synchronization mechanism forwards counters to other data centers periodically. When a counter is received in the remote data center, its value is merged with the local version. Thus, when using this custom synchronization mechanism, we do not rely on the fact that Riak exposes the versions of concurrent updates, as in each data center all updates are serialized and remote updates are merged before writing the new state to Riak.

This deployment strategy has an important limitation: the conditional writing mechanism for serializing operation execution works well under low load, but leads to an increased number of failed writes when the load increases. To address this issue, we present a deployment strategy based on a server-based middleware.

## 5.3 Alternative 2: Server-based middleware

Our server-based middleware for deploying *Bounded Counters* addresses the above mentioned limitation of the client-based solution by serializing in the middleware all operations executed in each data center for each counter. To this end, the middleware is built combining a client library and a server-based middleware deployed using a DHT – our prototype uses the riak_core DHT [13]. The DHT is deployed in the same nodes used by the Riak database, as depicted in Figure 5. Each DHT node is responsible for handling all requests for a subset of the counters, i.e., the client library calls the DHT node when executing *Bounded Counter* operations. For operations on regular objects, the client library calls directly Riak (without contacting DHT nodes).

When an application wants to execute an operation in a counter, the operation is sent to the DHT node responsible for that counter. The DHT node executes the operation running the steps described in the previous deployment strategy (it reads the counter from Riak, executes the operation in the CRDT and writes back the new value using conditional write). As a single node executes all operations for each counter, no concurrent writes will typically exist and conditional writes will tend to succeed.

When a new nodes enters the DHT or some node fails, the DHT is automatically reconfigured. During these reconfiguration periods, it is possible that two nodes process two different messages concurrently. To guarantee correctness in this case, our middleware uses conditional writes when writing the modified *Bounded Counter* CRDT back to the Riak database. Thus, if two nodes concurrently try to update the same *Bounded Counter* CRDT, one of the operations fails.

As in the previous design, our middleware could use the built-in geo-replication synchronization solutions available in Riak provided they worked with conditional writes. Since in the version we were using this was not the case, we had to implement a custom replication mechanism as in the previous design. For *Bounded Counters*, each DHT node periodically propagates entries to the DHT nodes in other data centers – with this approach, each synchronization can include the effects of a sequence of operations, thus reducing the communication overhead. For other objects, we rely on normal built-in Riak multi-data center replication. As in the previous version, our design does not need to rely on the fact that Riak exposes concurrent udpates, as concurrent updates may occur only across data centers and the synchronization mechanism automatically merges the local and remote state of counters.

*Optimizations.* Our prototype includes a number of optimization to improve its efficiency. The first optimization is to cache *Bounded Counter* CRDTs. This allows us to reduce the number of Riak operations necessary for processing each update on a *Bounded Counter* from two to one – only the write is necessary.

Under high contention in a *Bounded Counter*, the simple approach described is not very efficient, as one operation must complete before the next operation starts being processed. As processing an update requires writing the modified *Bounded Counter* CRDT back in the Riak database, which involves contacting remote nodes, each operation can take a few miliseconds to complete. To improve throughput, while a remote write to Riak is being done, the operations that were received are executed in the local copy of the *Bounded Counter* CRDT. If the operation fails when it is executed in the CRDT, the result is immediately returned to the client. Otherwise, no result is immediately returned and the operation becomes pending. When the previous write to the Riak database completes, the local version of the *Bounded Counter* CRDT is written in the Riak database – this version includes the effects of all pending operations. If the conditional write succeeds, all pending operations complete by returning success to the clients. Otherwise, clients are notified of the failure.

# 6 Extensions

In this section we discuss extensions to our middleware designs.

## 6.1 Supporting Other Cloud Stores

As discussed in Section 4, *Bounded Counter* CRDTs can be immediately used in a system that provides isolation for update execution and that supports replication with no lost updates. Our middleware designs, with custom synchronization among data centers, enable waiving the second requirement as discussed before. Thus, our middleware design could be used by any other cloud database that provides isolation for executing update operations on counters. For example, we could easily replace Riak by DynamoDB [11], which also supports conditional writes.

For cloud databases that do not support conditional writes (or that use another approach to serialize operation execution), our server based middleware design would also work, provided that the middleware DHT guarantees that operations for a given key are executed in sequence, even in the case of failures.

We now present an alternative design for guaranteeing that a single DHT node executes operations. The main idea is to nominate a DHT node as responsible for handling requests for each counter, and record this information in the *Bounded Counter* CRDT in the cloud store. When a DHT node receives an operation for some *Bounded Counter*, it tries to nominate itself as the the node responsible for executing operations on that counter. TBC **isto não é nada trivial**

## 6.2 Supporting Other Invariants

*Multiple numeric invariants.* In some cases, it might be interesting to have a counter involved in more than one numeric invariant – e.g. we may want to have $x \geq 0 \wedge y \geq 0 \wedge x + y \geq K$. In such cases, the invariant $x + y \geq K$ can be maintained by a *Bounded Counter* that represents the value of $x + y$. When updating the value of $x$ (or $y$), it is necessary to update both the *Bounded Counter* for $x$ and for $x + y$, with an operation succeeding if both execute with success. For maintaining invariants, this needs to be done atomically but not in isolation, i.e., either both *Bounded Counters* are updated or none, but an application might observe a state where only one of the *Bounded Counters* has been updated.

Without considering failures, this allows for a simple implementation where if one *Bounded Counter* operation fails, the operation in the other *Bounded Counter* is compensated [**?** ] by executing the inverse operation. When considering failures, it is necessary to include some transactional mechanism for guaranteeing that either both updates execute or none – recently, eventual consistent cloud databases started to support such features [18, 19].

*Other invariants.* A number of other invariants can be encoded as numeric invariants, as it has been discussed by Barbará-Milla and Garcia-Molina [7]. We now show how to adapt the proposed ideas and extend them to be able to enforce other invariants when using *Bounded Counters*.

An invariant that establishes a limit on the number of objects that satisfy some given condition can be implemented using a *Bounded Counter* with the appropriate limit – e.g., for guaranteeing that at least one object satisfies some condition, we would have a *Bounded Counter* with an invariant *larger or equal to one*; for guaranteeing that at most one object satisfies some condition, we would have a *Bounded Counter* with an invariant *smaller or equal to one*. Adding a new object that satisfies the condition can proceed without any rights, while removing the object would require the origin replica to hold rights to decrement the counter.

Referential integrity can be enforced by using a counter to count the number of references that exist. For removing the referenced object, the reference count must be zero and a dynamic invariant of *smaller or equal to zero* must be enforced. Adding such dynamic invariant can be implemented by removing all rights to execute *increments*.

# 7 Evaluation

We have implemented our middleware designs for extending Riak with support for numeric invariants and evaluated experimentally the prototypes. This evaluation tries to address the following main questions. (i) What is the performance of the proposed middleware designs when compared with alternative solutions? (ii) What is the horizontal scalability of the proposed middleware designs?

In our prototypes, clients execute operations on regular objects directly on Riak, using the Riak client library. Thus, our middleware has no impact on such operations. Given this, our experiments focus on the performance of *Bounded Counters*, using micro-benchmarks with different workloads. These benchmarks use counters with the invariant *greater or equal to zero*, which model the exhaustion of a budget for ad impressions or a product stock, for example.

## 7.1 Alternative solutions

In our experiments, we have compared the following solutions:

**Weakly Consistent Counters (WeakC)** This solution leverages Riak's native counters operating under weak consistency. Before issuing a decrement operation, clients read the current counter value and issue a decrement only if the current value is positive. While this approach is expected to be fast due to local DC execution, concurrent decrements have a chance to drive the counter past zero, into negative ground. The severity of the non-negativity invariant violation will depend on the level of concurrency, which depends strongly on the inter-DC synchronization frequency. In that regard, this solution uses the Riak's built-in continuous synchronization mode to try to minimize concurrency.

**Bounded Counters - server-side middleware (BCsrv)** This is our server-based middleware, as described in Section 5.3.

**Strongly Consistent Counters (StrongC)** This solution leverages Riak's strong consistency using a single DC to store and manage counters, and having clients in all data centers. A counter is updated by (1) reading its value; (2) updating the counter state; and (3) writing back the new state using a conditional write (that fails if the counter has been modified since it has been read). Executing this logic in clients that do not run in the DC where data resides leads to a high abort rate, as the conditional write will often fail due to concurrent updates (from the DC where data resides). To address this problem, we use a middleware layer to serialize the execution of this logic in the DC that holds the data, as in our server-side middleware. Thus, clients propagate inc/dec operations to the middleware, possibly over a wide area network. As a result, the steps for executing inc/dec operations become

local to the data center holding the counter data. This solution improves overall fairness, increasing the success rate of remote clients.

In this section we do not show detailed results for our client-based middleware, as our evaluation showed that when contention increases, the abort rate for operations increases very fast. The same effect occurs for a strong consistency solution that does not use a middleware to serialize the execution of updates. As our tests stress scenarios with significant contention, the performance of this systems degrades very quickly, showing that these solutions should only be used in deployments with low contention.

## 7.2 Experimental Setup

Our experiments comprised 3 Amazon EC2 data centers distributed across the globe. The latency between each data center is shown in Table 1. We installed a Riak data store in each EC2 availability zone (US-East, US-West, EU).

Each Riak ring is composed by three m1.large machines, with 2 vCPUs, producing 4 ECU[2] units of computational power, and with 7.5GB of memory available. We use Riak 2.0.0pre5 version.

We make use of Riak core 1.4.3 as the basis for the middleware used in BCsrv and StrongC. The DHT in the middleware is configured to ensure the physical mapping of DHT keys to physical nodes matches that of the Riak data store.

In StrongC data is stored in a single DC (US East). We selected this DC to store data to minimize latency from remote clients. In other solutions, data is fully geo-replicated in all data centers. Clients execute in 3 m1.large machines in each DC and connect to the Riak/middleware running in the same DC, with the exception of StrongC that connects to the middleware running in US-East.

| RTT (ms) | US-E | US-W | EU |
|---|---|---|---|
| US-East | - | 80 | 96 |
| US-West | 83 | - | 163 |
| EU | 93 | 161 | - |

**Table 1: RTT Latency between Data Centers in Amazon EC2.**

## 7.3 Single Counter

We start our evaluation with a micro-benchmark that uses a single counter. The counter is initialized to a large value and clients concurrently issue operations to decrement the value of the counter, while maintaining the invariant that the counter must remain larger or equal to zero. This experiment intends to measure the overhead of the different solutions and how they scale in a scenario of high contention.

*Throughput vs. latency.* Figure 6 presents the throughput vs. latency graph when using a single counter. Results show that our server-based middleware design performs better than the strong consistency solution in both latency and throughput. When compared with a solution that uses weak consistency, the scalability of our solution is worse. The reason for this is that in our solution (and in *StrongC*) a single node handles all requests in the middleware and acts as the primary of the Riak's conditional writing mechanism used. The throughput of our middleware is about three times better than the throughput of the strongly consistent solution, since

[2]1 ECU corresponds is a relative metric used to compare instance types in the AWS platform

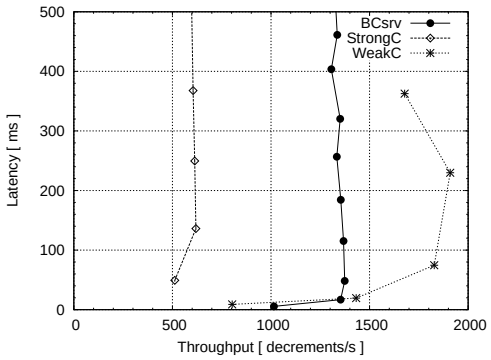



**Figure 6: Throughput vs. latency with a single counter.**

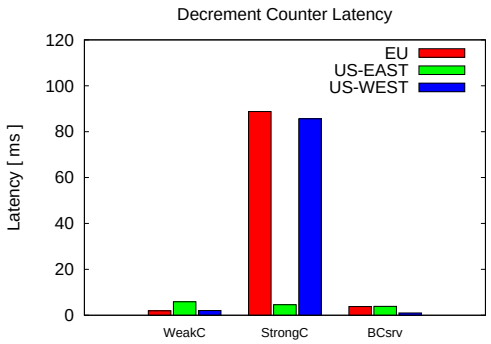



**Figure 7: Median latency with a single counter, per region of clients.**

we execute operation is three DCs instead of one DC for strong consistency.

*Latency under low load.* Figure 7 presents the median latency experienced by clients in different regions when load is low (5 threads in each client machine with a think time of 100 ms between two consecutive requests). As expected, the results show that for *StrongC*, remote clients experience high latency for operation execution. This latency is close to the RTT latency between the client and the DC holding the data.

Both *BCsrv* and *WeakC* experience very low latency. In a counter-intuitive way, the latency of *BCsrv* is sometimes even better than the latency of *WeakC*. This happens because our middleware caches the counters, requiring only one access to Riak for processing an update operation when compared with two accesses in *WeakC* (one for reading the value of the counter and another for updating the value if it is positive).

Figure 8 details these results by showing the CDF of latency for operation execution. The results allow to show that for *BCsrv* and *WeakC* only a few percent of operations experience high latency. For *StrongC*, each step in the line consists mostly of operations issued in different DCs.

Figure 9 furthers details the behavior of our middleware, by presenting the latency of operations over time. The results show that most operations take low latency, with a few peak of high latency when a replica runs out of rights and needs to ask for additional rights from other data centers. The number of peaks is small because most of the time the pro-active mechanism for exchanging

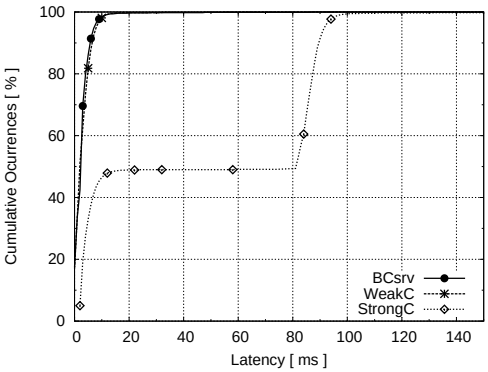



**Figure 8: CDF of latency with a single counter.**

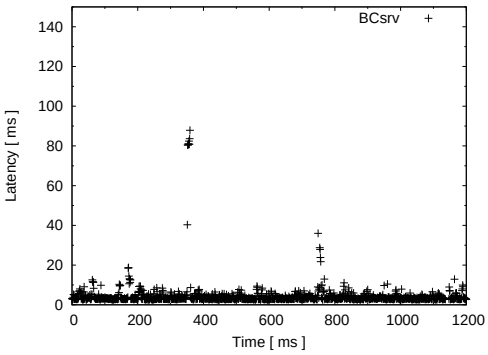



**Figure 9: Latency measured over time.**

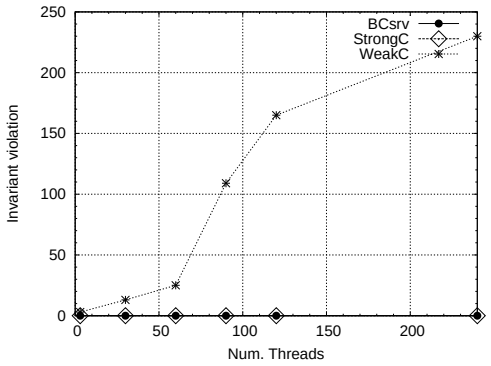



**Figure 10: Decrements executed in excess, violating invariant.**

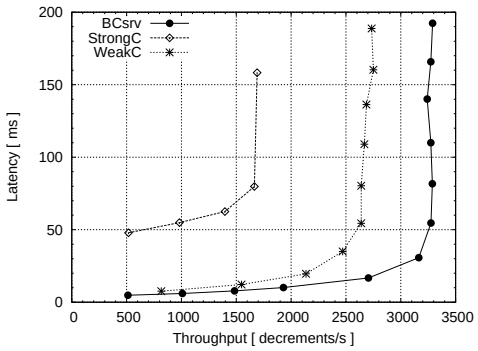



**Figure 11: Throughput vs. latency with multiple counters.**

rights is able to provision a replica with enough rights before all rights are used.

*Invariant Preservation.* We have also evaluated the severity of the risk of invariant violation. To this end, we have computed how may decrements in excess were executed with success in the different solutions. Figure 10 presents the obtained results. As expected, both *BCsrv* and *StrongC* do not break the invariant, but in the *WeakC* the invariant has been broken. The number of operations executed in excess increases as the number of clients increase. This is expected as when reaching a value close to zero, clients executing concurrently will all read that the limit has not been reached, but when all decrements execute, the limit is exceeded – e.g. if $N$ clients concurrently read that the value of a counter is 1 and they all concurrently decrement the counter, the final value will exceed the limit by $N-1$. This problem is made worse by geo-replication, as updates from a remote data center may take hundreds of milliseconds before being integrated, thus increasing the error on the local view of the counter. This shows that a system based on weak consistency cannot maintain strict invariants and that the problem gets worse as the load of the system increases.

## 7.4 Multiple Counters

To evaluate how the different solutions behave when data is distributed in all servers, we have run an experiment with 100 counters. Increasing the number of counters contributes to spreading the load among servers, but due to different reasons in different solutions. For those that use conditional writes (BCsrv and StrongC), each server machine will hold a relatively even subset of the primaries used by the primary-backup replication scheme in Riak. For WeakC, using a larger number of counters helps in achieving a more even load among all servers, since it uses preference lists for choosing which nodes to be included in quorums, and this helps populating these lists more evenly.

Figure 11 presents the throughput vs. latency graph when using multiple counters. When comparing these results with a single counter versus multiple counters, we can observe that our middleware scales better than weak consistency when the number of counters increases, leading to an even better throughput. This is because with a single counter, weak consistency already achieved some load balancing through the use of quorums, which allow for some flexibility in the choice of server nodes used in quorums for each operation. In contrast, with a single counter, the primary node for that counter represented a bottleneck in our design. As such, increasing the number of counters allows for spreading the load across nodes, which is more relevant in our case since it overcomes the more prominent bottleneck.

Achieving a better throughput than with weak consistency is only possible due to the techniques implemented in the middleware to minimize the number of operations executed in the underlying storage system.

# 8 Related work

A large number of cloud storage systems supporting geo-replication have been developed in recent years. Some of these systems [4, 8,

11, 15, 18, 19, 25] provide variants of eventual consistency, where operations return immediately after being executed in a single data center. This approach is very popular, as it allows low latency for end-users, by having data centers in multiple locations scattered across the globe and executing users' operations in the closest data center. Different variants of eventual consistency address different requirements, such as: reading a causally consistent view of the database [4, 18]; supporting a restricted form of transactions where a set of updates are made visible atomically [19]; supporting application-specific or type-specific reconciliation with no lost updates [8, 11, 18, 25, 26], etc. Our solution supports a complementary requirement – having counters that do not break a numeric invariant.

Although these systems can support a large range of applications, some applications require strong consistency (at least for a subset of its operations) in order to ensure correctness. Several systems support strong consistency. Spanner [10] provides strong consistency for the complete database, at the cost of incurring in the necessary coordination overhead for all updates. Transaction chains [29] support transaction serializability with latency proportional to the latency to the first replica accessed. Other systems, such as Walter [26] and Gemini [16], support both weak and strong consistency (snapshot isolation in Walter), which allows operations that can execute under weak consistency to run fast. PNUTS [9], DynamoDB [25] and Riak [8] also combine weak consistency with some form of per-object strong consistency relying on conditional writes – where a write fails if a concurrent write exists. Megastore [6] also combines strong consistency inside a partition with weak consistency accross partitions. Our work allows for maintaining the correctness of applications with numeric invariants, while allowing (most) operations to execute in a single replica (data center). Thus, it can be seen as an extension of some form of eventual consistency with numeric invariant preservation. Although it does not provide a general strong consistency model, it also does not incur in the overhead of such systems when it is only needed to maintain numeric invariants.

Bailis et al. [5] have studied when it is possible to avoid coordination in database systems, while maintaining application invariants. Our work is complimentary, by providing a solution for maintaining numeric invariants when coordination cannot be avoided. In such cases, (many) operations can still be executed without coordination because coordination has been moved outside the critical path of operation execution, by obtaining the necessary rights before start executing operations.

Our solution is inspired in escrow transactions [20]. This approach, initially proposed for increasing the concurrency of transaction in a single database has been used for supporting disconnected operation in mobile computing environments either relying on centralized [21, 28] or peer-to-peer [24] protocols for escrow distribution. We build upon the ideas of these systems and combine them with convergent data-types [23] to provide a decentralized solution that enforces both automatic convergence and invariant-preservation with no central authority. Additionally, we proposed, implemented and evaluated two middleware designs for integrating such solution with existing eventual consistent cloud stores.

Warranties [17] provide time-limited assertions over the state of the database and have been used for improving latency of read operations in cloud storages. While the goal of warranties is to support linearisability efficiently, our goal is to permit concurrent updates while enforcing invariants.

The demarcation protocol [7] has been proposed to maintain invariants in distributed databases. Although the underlying protocols are similar to escrow-based solutions, the demarcation protocol focus on maintaining invariants across different objects. MDCC [14] has recently proposed a variant of this protocol for enforcing data invariants in quorum systems. In section 6 we also discussed how to support other invariants with out approach, but other ideas from these paper could also be integrated with our work.

# 9 Final remarks

This paper proposes two middleware designs for extending eventually consistent cloud stores with the enforcement of numeric invariants. Our designs allow most operations to complete within a single data center by moving the necessary coordination outside of the critical path of operation execution, combining the benefits of eventual consistency – low latency, high availability – with those of strong consistency – easily enforcing global invariants. The resulting consistency model addresses the requirements of a large number of applications – e.g., Li et. al. [16] have shown that numeric invariants are one of the main sources that require web applications to resort to strong consistency models.

Our solution, inspired in escrow transactions, relies on a new CRDT [23], *Bounded Counter*, which maintains the necessary information to know when it is safe to execute operations locally. This CRDT can be used in any system that satisfies very weak assumptions – all updates are serialized in each replica (but can execute in different orders in different replicas) and the system includes a mechanism to merge concurrent updates. We propose two middleware designs for using *Bounded Counter* in existing cloud stores. The evaluation shows that our client-based middleware does not scale when contention is high. Our server-based middleware is scalable and exhibits latency comparable to solutions with weak consistency where invariants can be compromised, to a degree that increases with the load of the system.

As future work, we intend to address other invariants, and also include inter-object invariants.

# References


[1] http://gigaom.com/2013/02/21/basho-technologies-takes-aim-at-more-enterprises-with-upgrades/.

[2] Cassandra counters. http://wiki.apache.org/cassandra/Counters.

[3] Counters in riak 1.4. http://basho.com/counters-in-riak-1-4/.

[4] ALMEIDA, S., LEITÃO, J. A., AND RODRIGUES, L. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 85–98.

[5] BAILIS, P., FEKETE, A., FRANKLIN, M. J., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Coordination-avoiding database systems. *CoRR abs/1402.2237* (2014).

[6] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J. J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research* (2011), pp. 223–234.

[7] BARBARÁ-MILLÁ, D., AND GARCIA-MOLINA, H. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal 3*, 3 (July 1994), 325–353.

[8] BASHO. Riak. http://basho.com/riak/, 2014. Accessed Jan/2014.

[9] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1277–1288.

[10] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[11] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[12] HOFF, T. Latency is everywhere and it costs you sales - how to crush it. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[13] KLOPHAUS, R. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming* (New York, NY, USA, 2010), CUFP '10, ACM, pp. 14:1–14:1.

[14] KRASKA, T., PANG, G., FRANKLIN, M. J., MADDEN, S., AND FEKETE, A. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 113–126.

[15] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[16] LI, C., PORTO, D., CLEMENT, A., GEHRKE, J., PREGUIÇA, N., AND RODRIGUES, R. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 265–278.

[17] LIU, J., MAGRINO, T., ARDEN, O., GEORGE, M. D., AND MYERS, A. C. Warranties for faster strong consistency. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), nsdi'14, USENIX Association.

[18] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 401–416.

[19] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 313–328.

[20] O'NEIL, P. E. The escrow transactional method. *ACM Trans. Database Syst. 11*, 4 (Dec. 1986), 405–430.

[21] PREGUIÇA, N., MARTINS, J. L., CUNHA, M., AND DOMINGOS, H. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services* (New York, NY, USA, 2003), MobiSys '03, ACM, pp. 43–56.

[22] SCHURMAN, E., AND BRUTLAG, J. Performance related-changes and their user impact. Presented at velocity web performance and operations conference, 2009.

[23] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems* (Berlin, Heidelberg, 2011), SSS'11, Springer-Verlag, pp. 386–400.

[24] SHRIRA, L., TIAN, H., AND TERRY, D. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware* (New York, NY, USA, 2008), Middleware '08, Springer-Verlag New York, Inc., pp. 42–61.

[25] SIVASUBRAMANIAN, S. Amazon dynamodb: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 729–730.

[26] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 385–400.

[27] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 172–182.

[28] WALBORN, G. D., AND CHRYSANTHIS, P. K. Supporting semantics-based transaction processing in mobile database applications. In *Proceedings of the 14TH Symposium on Reliable Distributed Systems* (Washington, DC, USA, 1995), SRDS '95, IEEE Computer Society, pp. 31–.

[29] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 276–291.

## B.5 Valter Balegas, Mahsa Najafzadeh, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Marc Shapiro, Nuno Preguiça. Putting Consistency Back into Eventual Consistency. Submitted to EuroSys'15.

# Putting Consistency Back into Eventual Consistency

submission #51 - 15 pages


## Abstract

Geo-replicated storage systems are at the core of current Internet services. The designers of the replication protocols for these systems have to choose between either supporting low latency, eventually consistent operations, or supporting strong consistency for ensuring application correctness. We propose an alternative consistency model, *explicit consistency*, that strengthens eventual consistency with a guarantee to preserve specific invariants defined by the applications. Given these application-specific invariants, a system that supports explicit consistency must identify which operations are unsafe under concurrent execution, and help programmers to select either violation-avoidance or invariant-repair techniques. We show how to achieve the former while allowing most of operations to complete locally, by relying on a reservation system that moves replica coordination off the critical path of operation execution. The latter, in turn, allow operations to execute without restriction, and restore invariants by applying a repair operation to the database state. We present the design and evaluation of Indigo, a middleware that provides Explicit Consistency on top of a causally-consistent data store. Indigo guarantees strong application invariants while providing latency similar to an eventually consistent system.


## 1. Introduction

To improve the user experience in services that operate on a global scale, from social networks and multi-player online games to e-commerce applications, the infrastructure that supports these services often resorts to geo-replication [8, 9, 11, 22, 24, 25, 38], i.e., maintains copies of application data and logic in multiple datacenters scattered across the globe. This ensures low latency, by routing requests to the closest datacenter, but only when the request does not require cross-datacenter synchronization. Executing update operations without cross-datacenter synchronization is normally achieved through weak consistency. The downside of weak consistency models is that applications have to deal with concurrent operations not seeing the effects of each other, which can lead to non-intuitive and undesirable semantics.

Semantic anomalies do not occur in systems that offer strong consistency guarantees, namely those that serialize all updates [9, 22, 40]. However, these consistency models require coordination among replicas, which increases latency and decreases availability. A promising alternative is to try to combine the strengths of both approaches by supporting both weak and strong consistency for different operations [22, 38, 40]. However, operations requiring strong consistency still incur in high latency. Additionally, these systems make it harder to design applications, as operations need to be correctly classified to guarantee the correctness of the application.

In this paper, we propose *explicit consistency* as an alternative consistency model, in which applications define the consistency rules that the system must maintain as a set of invariants. Unlike models defined in terms of execution orders, explicit consistency is defined in terms of application properties: the system is free to reorder the execution of operations at different replicas, provided that application invariants are maintained.

In addition to proposing explicit consistency, we show that it is possible to implement it while mostly avoid cross-datacenter coordination, even for critical operations that potentially break invariants. To this end, we propose a methodology that, starting from the set of application invariants helps in the deployment of a modified version of the application that includes a set of techniques for precluding invariant violation under concurrency (or, alternatively, use a set of invariant repair actions that recover the service to a desired state). The methodology we propose is composed of the following three steps.

First, based on static analysis, we infer which operations can be safely executed without coordination. Second, for the remaining operations, we provide the programmer with a choice of automatic repair [35] or avoidance techniques. The latter extend escrow and reservation approaches [14, 30, 32, 36], in which a replica reserves the permission to execute a number of operations without coordinating with other replicas. This way we amortize the cost of coordina-

tion over multiple requests and move it outside the critical path. Third, after the potentially conflicting operations are identified and the strategy to handle them is chosen, the application code is instrumented with the appropriate calls to our middleware library.

Finally, we present the design of Indigo, a middleware for explicit consistency built on top of a geo-replicated key-value store. Indigo requires the underlying store to provide only properties that have been shown to be efficient to implement, namely per-key linearizability for replicas in each datacenter, causal consistency, and transactions with weak semantics [1, 24, 25].

In summary, this paper makes the following contributions:

- We propose explicit consistency as a new consistency model for application correctness, centered on the application behavior instead of the the order of the execution of operations;
- A methodology that, starting with an application and a set of associated invariants, derives an efficient reservation system to enforce explicit consistency;
- Indigo, a middleware system ensuring explicit consistency on top of a weakly consistent geo-replicated key-value store.

The remaining of the paper is organized as follows: Section 2 introduces explicit consistency; Section 3 gives an overview on the proposed approach to enforce explicit consistency; Section 4 details the analysis for detecting unsafe concurrent operations and Section 5 details the techniques for handling these operations; Section 6 discusses the implementation of Indigo and Section 7 presents an evaluation of the system; related work is discussed in Section 8 and Section 9 concludes the paper with some final remarks.

## 2. Explicit Consistency

In this section we define precisely the consistency guarantees that Indigo provides. To explain these, we start by defining the system model, and then how explicit consistency restricts the set of behaviors allowed by the model.

To illustrate the concepts, we use as running example the management of tournaments in a distributed multi-player game. The game maintains information about players and tournaments. Players can register and de-register from the game. Players compete in tournaments, for which they can enroll and disenroll. A set of matches occurs for each tournament. A tournament has a maximum capacity. In some cases – e.g., when there are not enough participants – a tournament can be canceled before it starts. Otherwise a tournament's lifecycle is creation, start, and end.

### 2.1 System model and definitions

We consider a database composed of a set of objects in a typical cloud deployment, where data is fully replicated in multiple datacenters, and partitioned inside each datacenter. For simplicity we assume that the goal of replication is performance, and not fault tolerance. As such, we can assume that replicas do not fail. However, it would be straightforward to handle faults by replacing each machine at a given datacenter with a replica group running a protocol like Paxos [16].

Applications access and modify the database by issuing high-level operations. These operations include a sequence of *read* and *write* operations enclosed in transactions.

We define a database snapshot, $S_n$, as the value of the database after executing the writes of a sequence of transactions $t_1, \ldots, t_n$ in the initial database state, $S_{init}$, i.e., $S_n = t_n(\ldots(t_1(S_{init})))$, with $t_i(S)$ the state after applying the write operations of $t_i$ to $S$. The state of a replica is the database snapshot that results from executing all committed transactions received in the replica - both local and remote. An application submits a transaction in a replica, with reads and writes executing in a private copy of the replica state. The application may decide to commit or abort the transaction. In the former case, writes are immediately applied in the local replica and asynchronously propagated to remote replicas. In the latter case, the transaction has no side-effect.

The snapshot set $T(S)$ of a database snapshot $S$ is the set of transactions used for computing $S$ - e.g. $T(S_n) = \{t_1, \ldots, t_n\}$. We say a transaction $t_{i+1}$ executing in a database snapshot $S_i$ happened-before $t_{j+1}$ executing in $S_j$, $t_{i+1} \prec t_{j+1}$, iff $T(S_i) \subsetneq T(S_j)$. Two transactions $t_{i+1}$ and $t_{j+1}$ are concurrent, $t_i \parallel t_j$, iff $t_{i+1} \not\prec t_{j+1} \wedge t_{j+1} \not\prec t_{i+1}$ [21].

Happens-before relation defines a partial order among transactions, $O = (T, \prec)$. We say $O_i = (T, <)$ is a valid serialization of $O = (T, \prec)$ if $O_i$ is a linear extension of $O$, i.e., $<$ is a total order compatible with $\prec$.

Our approach allows transactions to execute concurrently. Each replica can execute transactions according to a different valid serialization. We assume the system guarantees state convergence, i.e., for a given set of transactions $T$, all valid serializations of $(T, \prec)$ lead to the same database state. Different techniques can be used to this end, from a simple *last-writer-wins* strategy to more complex approaches based on conflict-free replicated data types (CRDTs) [35, 38].

### 2.2 Explicit consistency

We now define *explicit consistency*, a novel consistency semantics for replicated systems. The high level idea is to let programmers define the application-specific correctness rules that should be met at all times. These rules are defined as invariants over the database state.

In our tournament application, one invariant states that the cardinality of the set of enrolled players in a tournament cannot exceed its capacity. Another invariant is that the enrollment relation must bind players and tournaments that exist - this type of invariant is known as referential integrity in databases. Even if invariants are checked when an operation is executed, in the presence of concurrent operations

these invariants can be broken – e.g., if two replicas concurrently enroll players to the same tournament, and the merge function takes the union of the two sets of participants, the capacity of the tournament can be exceeded.

**Specifying restrictions over the state:** To define explicit consistency, we use first-order logic for specifying invariants as conditions over the state of database. For example, for specifying that the enrollment relation must bind players and tournaments that exist, we could define three predicates: $player(P)$, $tournament(T)$ and $enrolled(P,T)$ to specify that a player $P$ exists, a tournament $T$ exists and that player $P$ is enrolled in tournament $T$ respectively. The condition would then be specified by the following formula: $\forall P,T, enrolled(P,T) \Rightarrow player(P) \wedge tournament(T)$.

**Specifying rules over state transitions:** In addition to conditions over the current state, we support some forms of temporal specifications by specifying restrictions over state transitions. In our example, we can specify, for instance, that players cannot enroll or drop from a tournament between the start and the end of the tournament.

Such temporal specification can be turned into an invariant defined over the state of the database, by having the application store information that allows for such verification. In our example, when a tournament starts the application can store the list of participants for later checking against the list of enrollments. The rules that forbids enrollment/disenrollment of players can then be specified as $\forall P,T, participant(P,T) \Leftrightarrow enrolled(P,T)$, with the new predicate $participant(P,T)$ specifying that player $P$ participates in active tournament $T$.

The alternative to this approach would have been to use temporal logics that can specify rules over time [21, 31]. Such approaches would require more complex specification for programmers and a more complex analysis. As our experience has shown that this simpler approach was sufficient for specifying most common application invariants, we have decided to rely on this approach.

**Correctness conditions** We can now formally define explicit consistency, starting with the helper definition of an invariant $I$ as a logical condition applied over the state of the database. We say that $I$ holds in state $S$ iff $I(S) = true$.

**Definition 2.1** (I-valid serialization)**.** For a given set of transactions $T$, we say that $O_i = (T, <)$ is a I-valid serialization of $O = (T, \prec)$ iff $O_i$ is a valid serialization of $O$ and I holds in the state that results from executing any prefix of $O_i$.

A system is correct, providing explicit consistency, iff all serializations of $O = (T, \prec)$ are I-valid serializations.

## 3. Overview

Given the invariants expressed by the programmer, our approach for enforcing explicit consistency has three steps: (i) detect the sets of operations that may lead to invariant violation when executed concurrently (we call these sets *I-offender sets*); (ii) select an efficient mechanism for handling *I-offender sets*; (iii) instrument the application code to use the selected mechanism in a weakly consistent database system.

The first step consists of discovering *I-offender sets*. For this analysis, it is necessary to model the effects of operations. This information should be provided by programmers, in the form of annotations specifying how predicates are affected by each operation [1]. Using this information and the invariants, a static analysis process infers the minimal sets of operation invocations that may lead to invariant violation when executed concurrently (*I-offender sets*), and the reason for such violation. Conceptually, the analysis considers all valid database states and, for each valid database state, all sets of operation invocations that can execute in that state, and checks if executing all these sets in the same state is valid or not. Obviously, exhaustively considering all database states and operation sets would be impossible in practice, which required the use of the efficient verification techniques detailed in section 4.

The second step consists in deciding which approach will be used to handle *I-offender sets*. The programmer must select from the two alternative approaches supported: *invariant-repair*, in which operations are allowed to execute concurrently and invariants are enforced by automatic conflict resolution rules; *violation-avoidance*, in which the system restricts the concurrent execution of operations that can lead to invariant violation.

In the *invariant-repair* approach, the system automatically guarantees that invariants hold when merging operations executed concurrently, by including the necessary code for restoring invariants in the operations. This is achieved by relying on CRDTs, such as sets, trees and graphs. For example, concurrent changes to a tree can lead to cycles that can be broken using different repair strategies [28].

In the *violation-avoidance* approach, the system uses a set of techniques to control when it is possible and impossible to execute an operation in a datacenter without coordinating with others. For example, to guarantee that an enrollment can only bind a player and a tournament that exist, enrollments can execute in any replica without coordination by forbidding the deletion of players and tournaments. A datacenter can reserve the right to forbid the deletion for a subset of players and tournaments, which gives it the ability to execute enrollments for those players and tournaments without coordinating with other datacenters. Our reservation mechanisms supports such functionality with reservations tailored to the different types of invariants, as detailed in section 5.

Third, the application code is instrumented to use the conflict-repair and conflict-avoidance mechanisms selected

[1] This step could be automated using program analysis techniques, as done for example in [23, 34].

by the programmer. This involves extending operations to call the appropriate API functions defined in Indigo.

## 4. Detecting *I-offender sets*

The language for specifying application invariants is first-order logic formulas containing user-defined predicates and numeric functions. More formally, we assume the invariant is an universally quantified formula in prenex normal form[2]

$$\forall x_1, \cdots, x_n, \varphi(x_1, \cdots, x_n).$$

First-order logic formulas can express a wide variety of consistency constraints, as we exemplify in Section 4.1.

We have already seen that an invariant can use predicates, such as $player(P)$ or $enrolled(P, T)$. Numeric restrictions can be expressed through the use of functions. For example, function $nrPlayers(T)$ that returns the number of players in tournament $T$, can be used to express that tournaments must have at most five players enrolled: $\forall T, nrPlayers(T) \leq 5$. Invariants can be combined to define the global invariant of an application. For instance, we can have:

$$\begin{aligned} I = \forall P, T, enrolled(P, T) &\Rightarrow player(P) \wedge tournament(T) \\ &\wedge \\ &nrPlayers(T) \leq 5 \end{aligned}$$

The programmer does not need to provide an interpretation for the predicates and functions used in the invariant - she just has to write the application invariant and the effects of each operation over the terms of the invariant.

***Defining operation postconditions*** To express the effects of operations we use its side-effects, or postconditions, stating what properties are ensured after execution of the operation. Moreover, we take the postcondition to be the conjunction of all side-effects. There are two types of side-effect clauses: predicate clauses, which describe a truth assignment for a predicate (stating whether the predicate is true or false after execution of the operation); and function clauses, which define the relation between the initial and final function values. For example, operation $remPlayer(P)$, which removes player $P$, has a postcondition with predicate clause $\neg player(P)$, stating that predicate $player$ is false for player $P$. Operation $enroll(P, T)$, which enrolls player $P$ into tournament $T$, has a postcondition with two clauses, $enrolled(P, T) \wedge nrPlayers(T) = nrPlayers(T) + 1$. The second clause can be interpreted as a variable assignment, where $nrPlayers(T)$ is increased by one.

The syntax for postconditions is given by the grammar:

$$\begin{array}{lcl} post & ::= & clause_1 \wedge clause_2 \wedge \cdots \wedge clause_k \\ clause & ::= & pclause \mid fclause \\ pclause & ::= & p(o_1, o_2, \cdots, o_n) \mid \neg p(o_1, o_2, \cdots, o_n) \\ fclause & ::= & f(o_1, o_2, \cdots, o_n) = exp \oplus exp \\ exp & ::= & n \mid f(o_1, o_2, \cdots, o_n) \\ \oplus & ::= & + \mid - \mid * \end{array}$$

where $p$ and $f$ are predicates and functions respectively, over objects $o_1, o_2, \cdots, o_n$.

[2] Formula $\forall x, \varphi(x)$ is in prenex normal form if clause $\varphi$ is quantifier-free. Every first-order logic formula has an equivalent prenex normal form.

Although we imposed that a postcondition is a conjunction, it is possible to deal with operations that have alternative side-effects, by splitting the alternatives between multiple dummy operations. For example, an operation $\varphi$ with postcondition $\varphi_1 \vee \varphi_2$ could be replaced by operations $op_1$ and $op_2$ with postconditions $\varphi_1$ and $\varphi_2$, respectively.

The fact that postconditions are conjunctions of simple expressions and that predicates and functions are uninterpreted (no interpretation is given), imposes limits on the properties that can be expressed in this setting. For example, it not possible to express reachability properties and other properties over recursive data structures. Nevertheless, the next section shows it is possible to express a wide variety of database consistency properties.

***Existential quantifiers*** So far, the invariants have been formulated as universally quantified formulas. However, some properties require existential quantifiers. For example, to state that tournaments must have at least one player enrolled: $\forall T, tournament(T) \Rightarrow (\exists P, enrolled(P, T))$. In practice the existential quantifier can be replaced by a function, using a technique called skolemization. For this example at hand, we may use function $nrPlayers$ as such: $\forall T, tournament(T) \Rightarrow nrPlayers(T) \geq 1$.

### 4.1 Expressing Application Invariants

The intrinsic complexity of general invariants makes it difficult to build a comprehensive invariant model. We decided to use a simple model for defining invariants and predicates that still can express significant classes of invariants. This models allows programmers to express invariants in a rather straightforward way, as we exemplify for the following types of invariants.

***Uniqueness*** The uniqueness constraint can be used to express different correctness properties required by applications - e.g. uniqueness of identifiers within a collection. This invariant can be defined using a function that counts the number of elements with a given identifier. For example, the formula $\forall P, player(P) \Rightarrow nrPlayerId(P) = 1$, states that $P$ must have a unique player identifier. A different example of an uniqueness constraint is the existence of a single leader in a collection: $\forall T, tournament(T) \Rightarrow nrLeaders(T) = 1$.

***Numeric constraints*** Numeric constraints refer to numeric properties of the application and set lower or upper-bounds to data values (equality and inequality are special cases). Usually these constraints control the use or access to a limited resource, such as the limited capacity of a tournament exemplified before. Ensuring that a player does not overspend its (virtual) budget can be expressed as: $\forall P, player(P) \Rightarrow budget(P) \geq 0$. Ensuring experienced players cannot participate in beginner's tournaments can be expressed as: $\forall T, P, enrolled(P, T) \wedge beginners(T) \Rightarrow score(P) \leq 30$.

***Integrity constraints*** This type of constraints describes relationships between different objects, known as foreign keys constraints in databases, such as the fact that the enrollment must refer to existing players and tournaments, as exemplified in the beginning of this section. If the tournament application had a score table for players, another integrity constraint would be that every table entry must belong to an existing player: $\forall P, hasScore(P) \Rightarrow player(P)$.

### 4.2 Determining *I-offender sets*

To detect the sets of concurrent operation invocations that may lead to an invariant violation, we perform a static analysis of the operation's postconditions against invariants. Starting from a valid state, where the invariant is true, if the preconditions hold, the sequential execution of operations always preserve the invariant. However, concurrently executing operations in different replicas may cause a conflict, leading to an invariant violation.

We start by intuitively explaining the process of detecting *I-offender sets*. The process starts by checking operations with opposing postconditions (e.g. $p(x)$ and $\neg p(x)$). Take operations $addPlayer(P)$ with effect $player(P)$ and $remPlayer(P)$ with effect $\neg player(P)$. If these two operations are concurrently executed it is unclear whether player $P$ exists or not in the database. This is an implicit invariant and can be usually addressed choosing a resolution policy (as add-wins).

The process continues by considering, for each invariant, the effects of concurrent executions of multiple operations that affect the invariant: first pairs, then triples, and so forth until all operations are considered or a conflict arises.

To illustrate this process, we use our tournament application and the invariant $I$ presented in the beginning of section 4. For simplicity of presentation, we consider each of the conditions defined in invariant $I$ independently. The first invariant is a numeric restrictions: $\forall T, nrPlayers(T) \leq 5$. In this case, we have to take into account operation $enroll(P,T)$ that affects function $nrPlayers$ and determine if concurrent executions of $enroll(P,T)$ may break the invariant. For that, we substitute in invariant $I$ the operation's effects over function $nrPlayers$. Under the assumption that $nrPlayers(T) < 5$, the weakest precondition ensuring the invariant is not locally broken, we substitute and check whether this results in a valid formula (notation $I\{f\}$ describes the application of formula $f$ in invariant $I$):

$$\begin{array}{r} I\,\{nrPlayers(T) \leftarrow nrPlayers(T)+1\} \\ \{nrPlayers(T) \leftarrow nrPlayers(T)+1\} \\ nrPlayers(T) \leq 5\,\{nrPlayers(T) \leftarrow nrPlayers(T)+1\} \\ \{nrPlayers(T) \leftarrow nrPlayers(T)+1\} \\ nrPlayers(T)+1 \leq 5\,\{nrPlayers(T) \leftarrow nrPlayers(T)+1\} \\ nrPlayers(T)+1+1 \leq 5 \end{array}$$

The assumption $nrPlayers(T) < 5$ does not ensure the resulting inequality. So, it can be concluded that concurrent executions of operation $enroll(P,T)$ can lead to an invariant violation. For this operation, ensuring locally the (weakest) preconditions does not ensure the invariant will hold globally.

The second invariant of $I$ is $\forall P, T, enrolled(P,T) \Rightarrow player(P)$. In this case we need to detect whether $enroll(P,T)$ and $remPlayer(P)$ lead to an invariant violation. To this end, we substitute the effects of these operations in the invariant and check whether the resulting formula is valid.

$$\begin{array}{r} I\,\{enrolled(P,T) \leftarrow true\}\,\{player(P) \leftarrow false\} \\ true \Rightarrow false \wedge Tournament(T) \\ false \end{array}$$

As the resulting formula is not valid, a set of *I-offenders* is identified: $\{enroll, remPlayer(P)\}$.

We now systematically present the algorithm used to detected *I-offender sets*.

**Lemma 4.1** (Conflicting operations)**.** Operations $op_1$, $op_2$, $\cdots$, $op_n$ *conflict* with respect to invariant $I$ iff, assuming that $I$ is initially true and preconditions of $op_i$ are initially true ($1 \leq i \leq n$), the result of substituting the postconditions into the invariant is not a valid formula.

Algorithm 1 statically determines the minimal subsets of conflicting (or unsafe) operations. The core of the algorithm is function $conflict(I, s)$ which determines whether the set of operations $s$ break invariant $I$. This function uses the satisfiability modulo theory (SMT) solver Z3 [10] to verify the validity of the logical formulas used in Definition 4.1. The function checks first if the operations in $s$ have opposing postconditions (as $addPlayer$ and $remPlayer$). If that check fails, the next step is to submit to the solver a formula obtained by substituting all operations post-conditions in the invariant, and determine its validity.

Algorithm 1 has an initial loop (line 4) to determine which non-idempotent operations cause conflicts over numeric restrictions. The main loop (line 10) iteratively checks if adding a new operation into every possible subset of non-conflicting operations raises a conflict. Each step of the iteration increases the numbers of operations in the subset considered. It starts by determining which pairs of operations conflict. If a conflict is detected, it adds a new operation into the set of unsafe operations. Otherwise, in the next step, it checks whether joining another operation raises any conflict, and so forth. Although not expressed in the algorithm, the operation to be added should affect predicates still not instantiated in the invariant (line 10). The overall complexity of the algorithm is exponential on the number of operations, but this could be improved. Each *I-offender set* determined by the algorithm can be seen as an assignment to the predicates in the invariant that results in a non-valid (invariant) formula. Therefore, we could adapt an (efficient) algorithm for satisfiability module theories, as the ones overviewed in [29].

## 5. Handling *I-offender sets*

The previous step identifies *I-offender sets*. These sets are reported to the programmer that decides how each situation

**Algorithm 1** Algorithm for detecting unsafe operations.

**Require:** $I$ : invariant; $O$ : operations.
1: $C \leftarrow \emptyset$ {subsets of unsafe operations}
2: $N \leftarrow \emptyset$ {set of non-idempotent unsafe operations}
3: $S \leftarrow \emptyset$ {subsets of non-conflicting operations}
4: **for** $op \in O$ **do**
5: **if** $conflict(I, \{op\})$ **then**
6: $N \leftarrow N \cup \{\{op\}\}$
7: $S \leftarrow S \cup \{\{op\}\}$
8: $i \leftarrow 1$
9: **for** $s \in S$ **and** $\#s = i$ **and** $i < \#O$ **do**
10: **for** $op \in O - s$ **and** $s \cup \{op\} \notin C$ **do**
11: **if** $conflict(I, s \cup \{op\})$ **then**
12: $C \leftarrow C \cup \{s \cup \{op\}\}$
13: **else**
14: $S \leftarrow S \cup \{s \cup \{op\}\}$
15: $i \leftarrow i + 1$
16: **return** $C \cup N$

should be addressed. We now discuss the techniques that are available to the programmer in Indigo.

### 5.1 Invariant repairing

The first approach that can be used is to allow operations to execute concurrently and repair invariant violation after operations are executed. Indigo has limited support for this approach, which can only address invariants defined in the context of a single database object (which can be as complex as a tree or a graph). To this end, Indigo provides a library of objects that repair invariants automatically with techniques proposed in literature - e.g. sets, maps, graphs, trees with different conflict resolution policies [28, 35].

The programmer still has the opportunity to extend these objects for supporting additional invariants - e.g. it is possible to extend a general set to implement a set with limited capacity $n$ by modifying queries to consider that only $n$ elements exist selected deterministically from all elements in the underlying set [27].

### 5.2 Invariant-violation avoidance

The alternative approach is to avoid the concurrent execution of operations that would lead to an invariant violation when combining their effects. Indigo provides a set of basic techniques for achieving this.

#### 5.2.1 Reservations

We now discuss the high-level semantics of techniques used to restrict concurrent execution of updates - implementation in weakly consistent stores is addressed in the next section.

**UID generator:** One important source of potential invariant violations come from the concurrent creation of the same identifier in situations where these identifiers must be unique - e.g. identifier of objects in sets [3, 22]. This problem can be easily solved by splitting the space of identifiers that can be created in each replica. Indigo provides a service that generates unique identifiers by appending to a locally generated identifier a replica-specific suffix. Applications must use this service to generate unique identifiers that are used in operations.

**Escrow reservation:** For numeric invariants of the form $x \geq k$, we include an escrow reservation for allowing decrements to be executed without coordination. Given an initial value for $x = x_0$, there are initially $x_0 - k$ rights to execute decrements. These rights can be split by different replicas. For executing $x.decrement(n)$, the operation must acquire and consume $n$ rights to decrement $x$ in the replica it is submitted. If not enough rights exist in the replica, the system will try to obtain additional rights from other replicas. If this is not possible, the operation will fail. Executing $x.increment(n)$ creates $n$ rights to decrement $n$ initially assigned to the replica in which the operation that executes the increment is submitted.

A similar approach is used for invariants of the form $x \leq k$, with increments consuming rights and decrements creating new rights. For invariants of the form $x+y+\ldots+z \geq k$, a single escrow reservation is used, with decrements to any of the involved variables consuming rights and increments creating rights. If a variable $x$ is involved in more than one invariant, several escrow reservations will be affected by a single increment/decrement operation on $x$.

**Multi-level lock reservation:** When the invariant in risk is not numeric, we use a multi-level lock reservation (or simply multi-level lock) to restrict the concurrent execution of operations that can break invariants. A multi-level lock can provide the following rights: (i) *shared forbid*, giving the shared right to forbid some action to occur; (ii) *shared allow*, giving the shared right to allow some action to occur; (iii) *exclusive allow*, giving the exclusive right to execute some action.

When a replica holds some right, it knows no other replica holds rights of a different type - e.g. if a replica holds a *shared forbid*, it knows no replica has any form of *allow*. We now show how to use this knowledge to control the execution of *I-offender sets*.

In the tournament example, $\{enroll(P, T), remPlayer(P)\}$ is an *I-offender set*. We can associate a multi-level lock to one of the operations, for specific values of the parameters. For example, we can have a multi-level lock associated with $remPlayer(P)$, for each value of $P$. For executing $remPlayer(P)$, it is necessary to obtain the right *shared allow* on the reservation for $remPlayer(P)$. For executing $enroll(P, T)$, it is necessary to obtain the right *shared forbid* on the reservation for $remPlayer(P)$. This guarantees that enrolling some player will not execute concurrently with deleting the player, but concurrent enrolls or concurrent deletes can occur. If all replicas hold the *shared forbid* right on removing players, the most frequent enroll operation can execute in any replica without coordination with other replicas.

The *exclusive allow* right is necessary when an operation is incompatible with itself, i.e., when executing concurrently the same operation may lead to an invariant violation.

**Multi-level mask reservation:** For invariants of the form $P_1 \vee P_2 \vee \ldots \vee P_n$, the concurrent execution of any pair of operations that makes two different predicates false may lead to an invariant violation if all other predicates were originally false. In our analysis, each of these pairs is an *I-offender set*.

Using simple multi-level locks for each pair of operations is too restrictive, as getting a *shared allow* on one operation would prevent the execution of the other operation in all pairs. In this case, for executing one operation is suffices to guarantee that a single other operation is forbidden (assuming that the predicate associated with the forbidden operation is true).

To this end, Indigo includes a multi-level mask reservation that maintains the same rights as multi-level lock regarding a set of $K$ operations. With multi-level mask, when obtaining a *shared allow* right for some operation, it is necessary to obtain (if it does not exist already) a *shared forbid* right on some other operation. These operations are executed atomically by our system.

#### 5.2.2 Using Reservations

Our analysis outputs *I-offender sets* and the invariant that can be broken if operations execute concurrently. For each *I-offender set*, the programmer must select the type of reservation to be used - based on the invariant type that can be broken, a suggested reservation type is generated.

Even when using the same type of reservations for each *I-offender set*, it is possible to prevent the concurrent execution of *I-offender sets* using different sets of reservations - we call this a reservation system. For example, consider our tournament example with the following two *I-offender sets*:

$$\{enroll(P,T), remPlayer(P)\}$$
$$\{enroll(P,T), remTournament(P)\}$$

Given these *I-offender sets*, two different reservation systems can be used. The first system includes a single multi-level lock associated with $enroll(P,T)$, with $enroll(P,T)$ having to obtain a *shared allow* right to execute, while both $remPlayer(P)$ and $remTournament(T)$ would have to obtain the *shared forbid* right to execute. The second system includes two multi-level lock associated with $remPlayer(P)$ and $remTournament(T)$, with enroll having to obtain the *shared forbid* right in both to execute.

Indigo runs a simple optimization process to decide which reservation system to use. As generating all possible systems may take too long, this process starts by generating a small number of systems using the following heuristic algorithm: (i) select a random *I-offender set*; (ii) decide the reservation to control the concurrent execution of operations in the set, and associate the reservation with the operation: if a reservation already exists for some of the operations, use the same reservation; otherwise, generate a new reservation from the type previously selected by the user; (iii) select the remaining *I-offender set*, if any, that has more operations controlled by existing reservations and repeat the previous step.

For each generated reservations system, Indigo computes the expected frequency of reservation operations needed using as input the expected frequency of operations. The optimization process tries to minimize this expected frequency of reservation operations.

After deciding which reservation system will be used, each operation is extended to acquire and release the necessary rights before and after executing the code of the operation. For escrow locks, an operation that consumes rights will acquire rights before its execution and these rights will not be released in the end. Conversely, an operation that creates rights will create these rights after its execution.

## 6. Implementation

In this section, we discuss the implementation of Indigo as a middleware running on top of a causally consistent store. We first explain the implementation of reservations and how they are used to enforce explicit consistency. We conclude by explaining how Indigo is implemented on top of an existing geo-replicated store.

### 6.1 Reservations

Indigo maintains information about reservations as objects stored in the underlying causally consistent storage system. For each type of reservation, a specific object class exists. Each reservation instance maintains information about the rights assigned to each of the replicas - in Indigo, each datacenter is considered a single replica, as explained later.

The escrow lock object maintains the rights currently assigned to each replica. The following operations can be submitted to modify the state of the object: *escrow_consume* depletes rights assigned to the local replica; *escrow_generate* generates new rights in the local replica; *escrow_transfer* transfers rights from the local replica to some given replica. For example, for an invariant $x \geq K$, *escrow_consume* must be used by an operation that decrements $x$ and *escrow_generate* by operations that increment $x$.

When an operation executes in the replica where it is submitted, if insufficient rights are assigned to the local replica, the operation fails and has no side-effects. Otherwise, the state of the replica is updated accordingly and the side-effects are asynchronously propagated to the other replicas, using the normal replication mechanisms of the underlying storage system. As operations only deplete rights of the replica where they are submitted, it is guaranteed that every replica has a conservative view of the rights assigned to it - all operations that have consumed rights are known, but any operations that transferred new rights from some other replica may still have to be received. Given that the execution of operations is linearizable in a replica, this approach

guarantees the correctness of the system in the presence of any number of concurrent updates in different replicas and asynchronous replication, as no replica will ever consume more rights than those assigned to it.

The multi-level lock object maintains which right (exclusive allow, shared allow, shared forbid) is assigned to each replica, if any. Rights are obtained for executing operations with some given parameters - e.g. in the tournament example, for removing player $P$ the replica needs a *shadow allow* right for player $P$. Thus, a multi-level lock object manages the rights for the different parameters independently - a replica can have a given right for a specific value of the parameters or a subset of the parameter values. For simplicity, in our description, we assume that a single parameter exists.

The following operations can be submitted to modify the state of the multi-level lock object: *mll_giveRight* gives a right to some other replica - a replica with a shared right can give the same right to some other replica; a replica that is the only one with some right can change the right type and give it to itself or to some other replica; *mll_freeRight* revokes a right assigned to the local replica. As a replica can have been given rights by multiple concurrent *mll_giveRight* operations executed in different replicas, *mll_freeRight* internally encodes which *mll_giveRight* operations are being revoked. This is necessary to guarantee that all replicas converge to the same state.

As with escrow lock objects, each replica has a conservative view of the rights assigned to it, as all operations that revoke the local rights are always executed initially in the local replica. Additionally, assuming causal consistency, if the local replica shows that it is the only replica with some right, that information is correct system-wide. This condition holds despite concurrent operations and asynchronous propagation of updates, as any *mll_giveRight* executed in some replica is always propagated before a *mll_freeRight* in that replica. Thus, if the local replica shows that no other replica holds any right that is because no *mll_giveRight* has been executed (without being revoked).

The multi-level mask object maintains the information needed for a multi-level mask reservation by combining several multi-level lock objects. The operation *mlm_giveRight* allows to give rights for one of the specified multi-level locks.

### 6.2 Indigo middleware

We have built a prototype of Indigo on top of a geo-replicated data store with the following properties: (i) causal consistency; (ii) support for transactions that access a database snapshot and merge concurrent updates using CRDTs [35]; (iii) linearizable execution of operations for each object in each datacenter. It has been shown that all these properties can be implemented efficiently in geo-replicated stores and at least two systems support all these functionalities: SwiftCloud [43] and Walter [38]. Given that SwiftCloud has a more extensive support for CRDTs, which are fundamental for invariant-repair, we decided to build Indigo prototype on top of SwiftCloud.

Reservation objects are stored in the underlying storage system and they are replicated in all datacenters. Reservation rights are assigned to datacenters individually, which keeps the information small. As discussed in the previous section, the execution of operations in reservation objects must be linearizable (to guarantee that two concurrent transactions do not consume the same rights).

The execution of an operation in the replica where it is submitted has three phases: i) the reservation rights needed for executing the operation are obtained - if not all rights can be obtained, the operation fails; ii) the operation executes, reading and writing the objects of the database; iii) the used rights are released. For escrow reservations, rights consumed are not released; new rights are created in this phase. The side-effects of the operation in the data and reservation objects are propagated and executed in other replicas asynchronously and atomically.

Reservations guarantee that operations that can lead to invariant violation do not execute concurrently. However, operations need to check if the preconditions for operation execution hold before execution[3]. In our tournament example, an operation to remove a tournament cannot execute before removing all enrolled players. Reservations do not guarantee that this is the case, but only that a remove tournament will not execute concurrently with an enrollment.

An operation needs to access a database snapshot compatible with the used reservation rights, i.e., a snapshot that reflects the updates executed before the replica has acquired the rights being used. In our example, for removing a tournament it is necessary to obtain the right that allows such operation. This precludes the execution of concurrent enroll operations for that tournament. After the tournament has been deleted, an enroll operation can obtain a forbid right on tournament removal. For correctness, it is necessary that the operation observes the tournament as deleted, which is achieved by enforcing that updates of an operation are atomic and that the read snapshot is causally consistent (obtaining the forbid right necessarily happens after revoking the allow right, which happens after deleting the tournament). These properties are guaranteed in Indigo directly by the underlying storage system.

***Obtaining reservation rights*** The first and last phases of operation execution obtain and free the rights needed for operation execution. Indigo provides API functions for obtaining and releasing a list of rights. Indigo tries to obtain the necessary rights locally using ordered locking to avoid deadlocks. If other datacenters need to be contacted for obtaining some reservation rights, this process is executed before start obtaining rights locally. Unlike the process for obtain-

[3] This step could be automated by inferring preconditions from invariants and operation side-effects, given that the programmer specifies the code for computing the value of predicates

ing rights in the local datacenter, Indigo tries to obtain the needed rights from remote datacenters in parallel for minimizing latency. This approach is prone to deadlocks - if some remote right cannot be obtained, we use an exponential backoff approach that frees all rights and tries to obtain them again after an increasing amount of time.

When it is necessary to contact other datacenters to obtain some right, latency of operation execution is severely affected. In Indigo, reservation rights are obtained pro-actively using the following strategy. Escrow lock rights are divided among datacenters, with a datacenter asking for additional rights to the datacenter it believes has more rights (based on local information). Multi-level lock and multi-level mask rights are pre-allocated to allow executing the most common operations (based on the expected frequency of operations), with shared allow and forbid rights being shared among all datacenters. In the tournament example, *shared forbid* for removing tournaments and players can be owned in all datacenters, allowing the most frequent enroll to execute locally.

The middleware maintains a cache of reservation objects and allows concurrent operations to use the same shared (allow or forbid) right. While some ongoing operation is using a shared or exclusive right, the right cannot be revoked.

### 6.3 Fault-tolerance

Indigo builds on the fault-tolerance of the underlying storage system. In a typical geo-replicated store, data is replicated inside a datacenter using quorums or relying on a state-machine replication algorithm. Thus, the failure of a machine inside a datacenter does not lead to any data loss.

If a datacenter (fails or) gets partitioned from other datacenters, it is impossible to transfer rights from and to the partitioned datacenter. In each partition, operations that only require rights available in the partition can execute normally. Operations requiring rights not available in the partition will fail. When the partition is repaired (or the datacenter recovers with its state intact), normal operation is resumed.

In the event that a datacenter fails losing its internal state, the rights held by that datacenter are lost. As reservation objects maintain the rights held by all replicas, the procedure to recover the rights lost by the datacenter failure is greatly simplified - it is only necessary to guarantee that recovery is executed only once with a state that reflects all updates received from the failed datacenter.

## 7. Evaluation

This section presents an evaluation of Indigo. The main question our evaluation tries to answer is how does explicit consistency compares against *causal consistency* and *strong consistency* in terms of latency and throughput with different workloads. Additionally, we try to answer the following questions:

- Can the algorithm for detecting *I-offender sets* be used with realistic applications?
- What is the impact of an increasing the amount of contention in objects and reservations?
- What is the impact of using an increasing number of reservations in each operation?
- What is the behavior when coordination is necessary for obtaining reservations?

### 7.1 Applications

To evaluate Indigo, we used the two following applications.

***Ad counter*** The ad counter application models the information maintained by a system that manages ad impressions in online applications. This information needs to be geo-replicated for allowing fast delivery of ads. For maximizing revenue, an ad should be impressed exactly the number of times the advertiser is willing to pay for. This invariant can be easily expressed as $nrImpressions(A_i) \leq K_i$, with $K_i$ the maximum number of times ad $A_i$ should be impressed and the predicate $nrImpressions(A_i)$ returning the number of times it has been impressed. In a real system, when a client application asks for a new ad to be impressed, some complex logic will decide which ad should be impressed.

Advertisers will typically require ads to be impressed a minimum number of times in some countries - e.g. ad A should be impressed 10.000 times, including 4.000 times in US and 4.000 times in EU. This example is modeled by having the following additional invariants for specifying the limits on the number of impressions (impressions in excess in Europe and US can be accounted in $nrImpressionsOther$):

$$nrImpressionsEU(A) \leq 4000$$
$$nrImpressionsUS(A) \leq 4000$$
$$nrImpressionsOther(A) \leq 2000$$

We modeled this application by having independent counters for each ad and region. Invariants were defined with the limits stored in database objects:

$$nrImpressions((region, ad)) \leq targetImpressions((region, ad))$$

A single update operation that increments the ad tally was defined - this operation updates the predicate $nrImpressions$. Our analysis shows that the increment operation conflicts with itself for any given counter, but increments on different counters are independent. Invariants can be enforced by relying on escrow lock reservations for each ad.

Our experiments used workloads with a mix of: a read only operation that returns the value of a set of counters selected randomly; an operation that reads and increments a randomly selected counter. Our default workload included only increment operations.

***Tournament management*** This a version of the application for managing tournaments described in section 2 (and used throughout the paper as our running example), extended with read operations for browsing tournaments. The operations defined in this application are similar to operations that one would find in other management applications such as courseware management.

As detailed throughout the paper, this application has a rich set of invariants, including uniqueness rules for assigning ids; generic referential integrity rules for enrollments; and order relations for specifying the capacity of each tournament. This leads to a reservation system that uses both escrow lock and multi-level lock reservation objects. Three operations do not require any right to execute - add player, add tournament and disenroll tournament - although the latter access the escrow lock object associated with the capacity of the tournament. The other update operations involve acquiring rights before they can execute.

In our experiments we have run a workload with $82\%$ of read operations (a value similar to the TPC-W shopping workload), $4\%$ of update operations requiring no right for executing, and $14\%$ of update operations requiring rights ($8\%$ of the operations are enrollment and disenrollments).

#### 7.1.1 Performance of the Analysis

We have implemented the algorithm described in Section 4 for detecting *I-offender sets* in Java, relying on the satisfiability modulo theory (SMT) solver Z3 [10] for verifying invariants. The algorithm was able to find the existing *I-offender sets* in the applications. The average running time of this process in a recent MacBook Pro laptop was 19 ms for the ad counter applications and 2892 ms for the more complex tournament application.

We have also modeled TPC-W - the invariants in this benchmark are a subset of those of the tournament application. The average running time for detecting *I-offender sets* was 937 ms. These results show that the running time increases with the number of invariants and operations, but that our algorithm can process realistic applications.

### 7.2 Experimental Setup

We compare Indigo against three alternative approaches:

**Causal Consistency (Causal)** As our system was built on top of causally consistent SwiftCloud system[43], we have used unmodified SwiftCloud as representative of a system providing causal consistency. We note that this system cannot enforce invariants. This comparison allows us to measure the overhead introduced by Indigo.

**Strong Consistency (Strong)** We have emulated a strongly consistent system by running Indigo in a single DC and forwarding all operations to that DC. We note that this approach allows more concurrency than a typical strong consistency system as it allows updates on the same objects to proceed concurrently and be merged if they do not violate invariants.

**Red-Blue consistency (RedBlue)** We have emulated a system with Red-Blue consistency [22] by running Indigo in all DCs and having red operations (those that may violate invariants and require reservations) execute in a master DC, while blue operations execute in the closest DC respecting causal dependencies.

Our experiments comprised 3 Amazon EC2 datacenters - US-East, US-West and EU - with inter-datacenter latency presented in Table 1. In each DC, Indigo servers run in a single m3.xlarge virtual machine with 4 vCPUs and 8 ECUs of computational power, and 15GB of memory available. Clients that issue transactions run in up to three m3.xlarge machines. Where appropriate, we placed the master DC in US-East datacenter to minimize the communication latency and have those configurations perform optimally.

| RTT (ms) | US-E | US-W |
|---|---|---|
| US-West | 81 | - |
| EU | 93 | 161 |

**Table 1.** RTT Latency among Datacenters in Amazon EC2

### 7.3 Latency and throughput

We start by comparing the latency and throughput of Indigo with alternative deployments for both applications.

We have run the ad counter application with $1000$ ads and a single invariant for each ad. The limit on the number of impressions was set sufficiently high to guarantee that the limit is not reached. The workload included only update operations for incrementing the counter. This allows us to measure the peak throughput when operations are able to obtain reservations in advance. The results are presented in Figure 1, and show that Indigo achieves throughput and latency similar to a causally consistent system. Strong and RedBlue results are similar, as all update operations are red and execute in the master DC in both configurations.

Figure 2 presents the results when running the tournament application with the default workload. As before, results show that Indigo achieves throughput and latency similar to a causally consistent system. In this case, as most operations are read-only or can be classified as blue and execute in the local datacenter, RedBlue throughput is only slightly worse than that of Indigo.

Figure 3 details these results presenting latency per operation type (for selected operations) in a run with throughput close to the peak value. The results show that Indigo exhibits lower latency than RedBlue for red operations. These operation can execute in the local DC in Indigo, as they require either no reservation or reservations that can be shared and are typically locally available.

Two other results deserve some discussion. *Remove tournament* requires canceling shared forbid rights acquired by other DCs before being able to acquire the shared allow right for removing the tournament, which explain the high latency. Sometimes latency is extremely high (as shown by the line with the maximum value) - this is a result of the asynchronous algorithms implemented and the approach for requesting remote DCs to cancel their rights, which can fail when a right is being used. This could be improved by running a more elaborate protocol based on Paxos. *Add player* has a surprisingly high latency in all configurations. Analyzing the situation, we found out that the reason for this lies

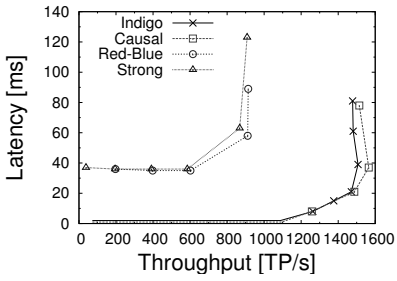



**Figure 1.** Peak throughput (ad counter application).

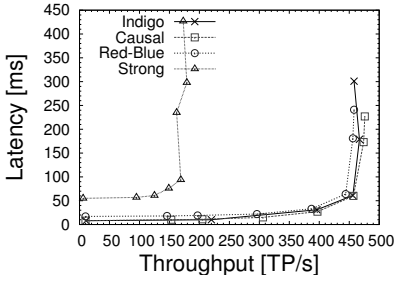



**Figure 2.** Peak throughput (tournament application).

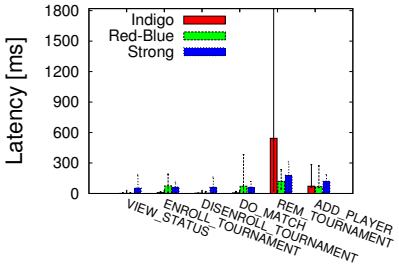



**Figure 3.** Average latency per op. type - Indigo (tournament app.).

in the fact that this operation manipulates very large objects used to maintain indexes - all configurations have a fix overhead due to this manipulation.

### 7.4 Micro-benchmarks

Next, we examine the impact of key parameters.

***Increasing contention*** Figure 4 shows the throughput of the system with increasing contention in the ad counter application, by varying the number of counters in the experiment. As expected, the throughput of Indigo decreases when contention increases as several steps require executing operations sequentially. Our middleware introduces additional contention when accessing the cache. As the underlying storage system also implements linearizability per-object, it is also possible to observe its throughput also decreases with increased contention, although more slowly.

***Increasing number of invariants*** Figure 5 presents the results of ad counter application with an increasing number of invariants - from one to three. In this case, the results show that the peak throughput with Indigo decreases while latency keeps constant. The reason for this is that for escrow locks, each invariant has an associated reservation object - thus, when increasing the number of invariants the number of updated objects also increases, with impact on the operations that each datacenter needs to execute. To verify our explanation, we have run a workload with operations that access the same number of counters in the weak consistency configuration - the presented results show the same pattern for decreased throughput.

***Behaviour when transferring reservations*** Figure 6 shows the latency of individual operations executed in US-W datacenter in the ad counter application for a workload where increments reach the invariant limit for multiple counters. When rights do not exist locally, Indigo cannot mask the latency imposed by coordination - in this case, for obtaining additional rights from the remote datacenters.

In Figure 3 we have shown the impact of obtaining a multi-level lock shared right that requires revoking rights present in all other replicas. We have discussed this problem and a possible solution in section 7.3. Nevertheless, it is important to note that such big impact in latency is only experienced when it is necessary to revoke shared forbid rights in all replicas before acquiring the needed shared allow right. The positive consequence of this approach is that enroll operations requiring the shared forbid right that was shared by all replicas execute with latency close to zero. The maximum latency line in enroll operation shows the maximum latency experienced when a replica acquires a shared forbid right from a replica already holding such right.

## 8. Related work

**Geo-replicated storage systems** Many cloud storage systems supporting geo-replication emerged in recent years. Some [1, 11, 20, 24, 25] offer variants of eventual consistency, where operations return right after being executed in a single datacenter, usually the closest one to the end-user to improve response times. These variants target different requirements, such as: reading a causally consistent view of the database (causal consistency) [1, 2, 13, 24]; supporting limited transactions where a set of updates are made visible atomically [4, 25]; supporting application-specific or type-specific reconciliation with no lost updates [6, 11, 24, 38], etc. Indigo is built on top of a geo-replicated store supporting causal consistency, a restricted form of transactions and automatic reconciliation; it extends those properties by enforcing application invariants.

Eventual consistency is insufficient for some applications that require (some operations to execute under) strong consistency for correctness. Spanner [9] provides strong consistency for the whole database, at the cost of incurring coordination overhead for all updates. Transaction chains [44] support transaction serializability with latency proportional to the latency to the first replica accessed. MDCC [19] and Replicated Commit [26] propose optimized approaches for executing transactions but still incur in intra-datacenter latency for committing transactions.

Some systems tried to combine the benefits of weak and strong consistency models by supporting both. In Walter [38] and Gemini [22], transactions that can execute under weak consistency run fast, without needing to coordinate with other datacenters. Bayou [39] and Pileus [40] allow operations to read data with different consistency levels, from strong to eventual consistency. PNUTS [8] and DynamoDB

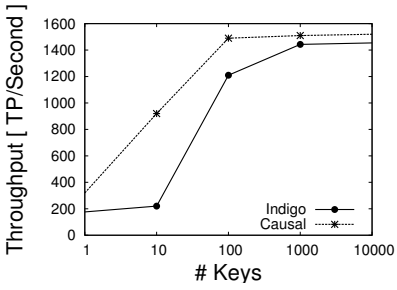



**Figure 4.** Peak throughput with increasing contention (ad counter application).

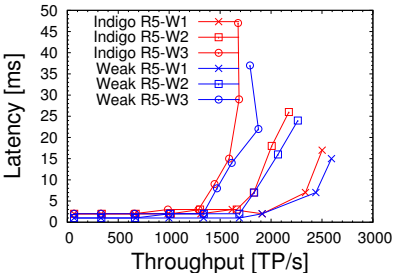



**Figure 5.** Peak throughput with an increasing number of invariants (ad counter application).

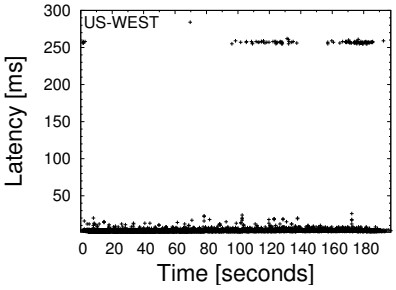



**Figure 6.** Latency of individual operations of US-W datacenter (ad counter application).

[37] also combine weak consistency with per-object strong consistency relying on conditional writes, where a write fails in the presence of concurrent writes. Indigo enforces explicit consistency rules, exploring application semantics to let (most) operations execute in a single datacenter.

**Exploring application semantics** Several works have explored the semantics of applications (and data types) for improving concurrent execution. Semantic types [15] have been used for building non serializable schedules that preserve consistency in distributed databases. Conflict-free replicated data types [35] explore commutativity for enabling the automatic merge of concurrent updates, which Walter [38], Gemini [22] and SwiftCloud [43] use as the basis for providing eventual consistency. Indigo goes further by exploring application semantics to enforce application invariants that can span multiple objects.

Escrow transactions [30] offer a mechanism for enforcing numeric invariants under concurrent execution of transactions. By enforcing local invariants in each transaction, they can guarantee that a global invariant is not broken. This idea can be applied to other data types, and it has been explored for supporting disconnected operation in mobile computing [32, 36, 41]. The demarcation protocol [5] is aimed at maintaining invariants in distributed databases. Although its underlying protocols are similar to escrow-based approaches, it focuses on maintaining invariants across different objects. Warranties [14] provide time-limited assertions over the database state, which can improve latency of read operations in cloud storages.

Indigo builds on these works, but it is the first to provide an approach that, starting from application invariants expressed in first-order logic leads to the deployment of the appropriate techniques for enforcing such invariants in a geo-replicated weakly consistent data store.

**Other related work** Bailis et al. [3] studied the possibility of avoiding coordination in database systems and still maintain application invariants. Our work complements that, addressing the cases that cannot entirely avoid coordination, yet allow operations to execute immediately by obtaining the required reservations in bulk and anticipation.

Others have tried to reduce the need for coordination by bounding the degree of divergence among replicas. Epsilon-serializability [33] and TACT [42] use deterministic algorithms for bounding the amount of divergence observed by an application using different metrics - numerical error, order error and staleness. Consistency rationing [18] uses a statistical model to predict the evolution of replicas state and allows applications to switch from weak to strong consistency on the likelihood of invariant violation. In contrast to these works, Indigo focuses on enforcing invariants efficiently.

The static analysis of code is a standard technique used extensively for various purposes [7, 12, 17], including in a context similar to ours. Sieve [23] combines static and dynamic analysis to infer which operations should use strong consistency and which operations should use weak consistency in a Red-Blue system [22]. In [34], the authors present an analysis algorithm that describes the semantics of transactions. These works are complementary to ours, and the proposed techniques could be used to automatically infer application side-effects. The latter work also proposes an algorithm to allow replicas to execute transactions independently by defining conditions that must be met in each replica. Whenever an operation cannot commit locally, a new set of conditions is computed and installed in all replicas using two-phase commit. In Indigo, replicas can exchange rights peer-to-peer.

## 9. Conclusions

This paper proposes an application-centric consistency model for geo-replicated services - explicit consistency - where programmers specify the consistency rules that the system must maintain as a set of invariants. We describe a methodology that helps programmers decide which invariant-repair and violation-avoidance techniques to use to enforce explicit consistency, extending existing applications. We also present the design of Indigo, a middleware that can enforce explicit consistency on top of a causally consistent store. The results show that the modified applications have performance similar to weak consistency for most operations, while being able to enforce application invariants. Some rare operations that require intricate rights transfers exhibit high latency. As future work, we intend to improve the algorithms for exchanging reservation rights on those situations.

## References


[1] S. Almeida, J. Leitão, and L. Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. . URL http://doi.acm.org/10.1145/2465351.2465361.

[2] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. . URL http://doi.acm.org/10.1145/2463676.2465279.

[3] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination-avoiding database systems. *CoRR*, abs/1402.2237, 2014.

[4] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *ACM SIGMOD Conference*, 2014.

[5] D. Barbará-Millá and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, July 1994. ISSN 1066-8888. . URL http://dx.doi.org/10.1007/BF01232643.

[6] Basho. Riak. http://basho.com/riak/, 2014. Accessed Oct/2014.

[7] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[8] 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. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=1454159.1454167.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387905.

[10] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '08, pages 337–340. Springer, 2008.

[11] 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 *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. . URL http://doi.acm.org/10.1145/1294261.1294281.

[12] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, 12 1998.

[13] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1. . URL http://doi.acm.org/10.1145/2523616.2523628.

[14] ed Liu, T. Magrino, O. Arden, M. D. George, and A. C. Myers. Warranties for faster strong consistency. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, nsdi'14, Berkeley, CA, USA, 2014. USENIX Association.

[15] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, June 1983. ISSN 0362-5915. . URL http://doi.acm.org/10.1145/319983.319985.

[16] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, Mar. 2006. ISSN 0362-5915. . URL http://doi.acm.org/10.1145/1132863.1132867.

[17] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA Formal Methods*, pages 41–55. Springer, 2011.

[18] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, Aug. 2009. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=1687627.1687657.

[19] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. . URL http://doi.acm.org/10.1145/2465351.2465363.

[20] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010. ISSN 0163-5980. . URL http://doi.acm.org/10.1145/1773912.1773922.

[21] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994. ISSN 0164-0925. . URL http://doi.acm.org/10.1145/177492.177726.

[22] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387906.

[23] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2. URL http://dl.acm.org/citation.cfm?id=2643634.2643664.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. . URL http://doi.acm.org/10.1145/2043556.2043593.

[25] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2482626.2482657.

[26] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9):661–672, July 2013. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=2536360.2536366.

[27] S. Martin, M. Ahmed-Nacer, and P. Urso. Controlled conflict resolution for replicated document. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2012 8th International Conference on*, pages 471–480, Oct 2012.

[28] S. Martin, M. Ahmed-Nacer, and P. Urso. Abstract unordered and ordered trees crdt. *CoRR*, abs/1201.1784, 2012.

[29] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to DPLL(T). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[30] P. E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, Dec. 1986. ISSN 0362-5915. . URL http://doi.acm.org/10.1145/7239.7265.

[31] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, FOCS, pages 46–57. IEEE, 1977.

[32] N. Preguiça, J. L. Martins, M. Cunha, and H. Domingos. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 43–56, New York, NY, USA, 2003. ACM. . URL http://doi.acm.org/10.1145/1066116.1189038.

[33] K. Ramamritham and C. Pu. A formal characterization of epsilon serializability. *IEEE Trans. on Knowl. and Data Eng.*, 7(6):997–1007, Dec. 1995. ISSN 1041-4347. . URL http://dx.doi.org/10.1109/69.476504.

[34] S. Roy, L. Kot, N. Foster, J. Gehrke, H. Hojjat, and C. Koch. Writes that fall in the forest and make no sound: Semantics-based adaptive data consistency. *CoRR*, abs/1403.2307, 2014. URL http://arxiv.org/abs/1403.2307.

[35] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24549-7. URL http://dl.acm.org/citation.cfm?id=2050613.2050642.

[36] L. Shrira, H. Tian, and D. Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 42–61, New York, NY, USA, 2008. Springer-Verlag New York, Inc. ISBN 3-540-89855-7. URL http://dl.acm.org/citation.cfm?id=1496950.1496954.

[37] S. Sivasubramanian. Amazon dynamodb: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1247-9. . URL http://doi.acm.org/10.1145/2213836.2213945.

[38] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. . URL http://doi.acm.org/10.1145/2043556.2043592.

[39] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4. . URL http://doi.acm.org/10.1145/224056.224070.

[40] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. . URL http://doi.acm.org/10.1145/2517349.2522731.

[41] G. D. Walborn and P. K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Proceedings of the 14TH Symposium on Reliable Distributed Systems*, SRDS '95, pages 31–, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7153-X. URL http://dl.acm.org/citation.cfm?id=829520.830874.

[42] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251229.1251250.

[43] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. M. Preguiça. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. *CoRR*, abs/1310.3107, 2013.

[44] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. . URL `http://doi.acm.org/10.1145/2517349.2522729`.