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.4.2 Programming principles and tools
Due date of deliverable:	October 1, 2015
Actual submission date:	28th September 2015
Start date of project:	October 1, 2013
Duration:	36 months
Organisation name of lead contractor for this deliverable:	UCL
Revision:	1
Dissemination level:	PU

Contents

1 Executive summary

The purpose of WP4 is to design the programming principles and tools that we need to program large-scale applications based on CRDTs. In the second year, this work package has collaborated with all other work packages. WP4 has made significant progress in four tracks, namely programming model, program specification, consistency properties of programs, and program verification. We summarize the progress as follows:

- In the programming model track, we have designed and implemented *Lasp*, a programming model that allows doing functional composition of CRDTs while keeping the good convergence properties and the robust distributed implementation of CRDTs. This allows building complete applications that retain the good properties of CRDTs. We have proved that Lasp is semantically equivalent to *functional programming*, which makes it easy for programmers to use.
- In the consistency track, we have greatly extended the power of the work on explicit consistency by defining a proof system, *CISE*, that allows to efficiently prove data integrity invariants. We have also extended explicit consistency to do conflict resolution using invariant repair instead of using reservations in the case of potential conflicts. This supports a powerful program development methodology where the proof system is able to prove correctness of programs that have been fixed using invariant repair techniques. We are currently implementing the proof system in Z3 and VCC.
- In the specification and verification tracks, we are now focusing on verifying applications written in Antidote, the project platform developed in WP2. For specification, we have formalized part of Antidote and we are using the Isabelle prover to verify the specification of eventually consistent programs using CRDTs. For verification, we are extending the Antidote system to do runtime verification of applications written in Antidote.

In the following paragraphs we explain each of these tracks in more detail. This is followed by five sections in the main body of the report that give more technical information about the tracks. Finally, the appendices give all the papers published in these tracks during year 2.

1.1 Software deliverable (Section 4)

The software deliverable D4.2.1 consists of three software packages: the first is the Lasp programming model and the second and third are tool prototypes, respectively for static and dynamic verification of Antidote applications.

1.2 Programming model (Section 5)

The Lasp programming model extends the Strong Eventual Consistency property of individual CRDT instances to the *composition* of CRDT instances using operators derived from functional and database programming (map, filter, fold, product, union, and intersection). A distributed program written in Lasp runs as a graph of CRDT instances connected by replicated processes. We have proved that the execution of a Lasp program is equivalent to a single sequence of states. This allows reasoning with Lasp programs as functional programs. The resulting model has the following good properties: it is confluent and referentially transparent (like functional programming), it supports non-determinism and nonmonotonicity, and it has an efficient distributed and fault-tolerant implementation.

We currently have two implementations of Lasp, where the same programming model runs with a different distribution model (client/server on a data center using Riak Core, and gossip using the Plumtree epidemic broadcast protocol). The first is a data-center implementation running on a consistent-hashed ring (in Riak Core) with external clients (see Appendix B). The second, called Selective Hearing, is a gossip-based implementation that uses the Plumtree epidemic broadcast algorithm to execute Lasp programs on a set of nodes connected by gossip (see Appendix D). The second model is particularly interesting since it targets *edge computing*, which is a major trend in Internet computing, e.g., for Internet of Things, and which is one of the goals of SyncFree in the third year. A prototype implementation of Lasp is available (see Section 4).

In the third year, we will extend Lasp to support specifying explicit causality and transactions. We intend to use Antidote as a backend since it supports transactions. We will also continue our work on applying Lasp ideas to edge computing.

1.3 Consistency (Section 6)

One of the goals of SyncFree is to understand how different consistency models can live together in the same application. In the second year, we have made progress on this goal by extending SyncFree's work on consistency into a full-fledged programming methodology, combining program proof with conflict resolution. We have defined a formal proof system, CISE (*'Cause I'm Strong Enough*), that can efficiently prove correctness of data integrity invariants in a distributed program. CISE is general enough to cover the use of multiple consistency models in the same program (see Appendix I), including but not limited to causal consistency, sequential consistency (a form of strong consistency), and RedBlue consistency. CISE "operations" as mentioned in Section 6 are in fact full transactions since they can consist of multiple primitive operations. We are currently implementing and evaluating the CISE proof system through implementations done in the Z3 and VCC theorem provers.

1.4 Explicit consistency using invariant repair (Section 7)

The Explicit Consistency approach was originally defined to use static analysis to pinpoint potential conflicting operations so they can be handled correctly at run-time. We have now extended this approach to support *invariant repair*, which uses compensation operations to avoid costly synchronization at run-time. This approach naturally combines with the CISE proof system to support a program development methodology: the CISE proof tool can prove correctness of programs that have been modified with invariant repair.

1.5 Specification and static verification (Section 8)

We have extended the work on specification of eventually consistent applications using CRDTs to use a formal model of part of Antidote and an interactive proof using the

Isabelle/HOL theorem prover, which is able to generate formal proofs of application properties. The Antidote formalization includes causal consistency, eventually consistent transactions, and data type semantics. In the third year we will explore how to combine this work with the CISE proof system, to combine the flexibility of our model with the automatic methods of CISE, and the work on run-time verification, so that developers can test invariants before starting a formal proof. We intend for this to lead to a practical tool that combines useful forms of specification, formal proof, and run-time verification. A prototype of this tool is available (see Section 4).

1.6 Run-time (dynamic) verification (Section 9)

As a continuation of the work on verification, in the first year on CRDTs and now on applications written with CRDTs, we have started work in the second year on a run-time verification tool for Antidote that instruments the Antidote implementation and adds a high-level command interface to support the development and debugging of CRDT-based applications running on Antidote. Since they are written in Antidote, these applications can use causal consistency and transactions. We are exploring early stage bug detection and reproducibility, and support to help programmers write applications that avoid synchronization operations. We expect this work to lead to a practical tool for developers using Antidote. A prototype of this tool is available (see Section 4).

2 Milestones in the Deliverable

Milestone MS2 (M24) is on extended guarantees and composition in a dynamic environment. This concerns work packages WP2, WP3, WP4, and WP5. Task 4.2 has contributed to this milestone by focusing on the following goals, as stated in the description of work:

> *T4.2 Extended programming model (transactions and garbage collection)*
> This task will extend the basic programming model with additional abilities according to application needs. Abilities to be added include transaction support, causality management, stream-based dataflow, and garbage collection. Early experience shows the need for these abilities. For example, it is important to provide efficient causality management for CRDT operations, which depend on knowledge of causality. It is also important to provide garbage collection, to remove no-longer-needed information while maintaining CRDT monotonicity. The research question is how to add these abilities to a CRDT-based framework while maintaining its good properties.

3 Contractors contributing to the Deliverable

3.1 UCLouvain

Peter Van Roy, Seyed Hossein Haeri.

3.2 Basho

Christopher Meiklejohn.

3.3 Koç

Maryam Dabaghchian, Serdar Tasiran.

3.4 KL

Peter Zeller.

3.5 Nova

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

3.6 INRIA

Mahsa Najafzadeh, Marc Shapiro.

4 Software Deliverable D4.2.1

The software deliverable D4.2.1 is presented as the following software package:

- Lasp programming model. Available in the repository:

  `https://github.com/lasp-lang/lasp`

  See Section 5. This repository contains the Riak Core prototype (see Appendix B), the Plumtree (gossip) prototype (see Appendix D), the Ad Counter scenario, as well as numerous tests.

In addition, we present also the current state of two tool prototypes:

- Prototype framework for verifying specifications of Antidote-based applications. Available in the repository:

  ```
  https://softech-git.informatik.uni-kl.de/zeller/
            isabelle_crdt_apps
  ```

  See Section 8. This repository contains the framework for specification and verification, implemented using the Isabelle proof assistant. This framework allows verifying applications built on top of a causally consistent database and CRDTs, which is a formalized subset of Antidote.

- Commander: a run-time verification prototype for Antidote-based applications. Available in the repository:

  ```
  https://github.com/SyncFree/antidote/tree/
            scheduler_added
  ```

  See Section 9. This repository contains the current version of the Commander tool and an example based on a simple wallet application.

5 Programming model

5.1 Overview

A major result for programming models in the second year is the definition and implementation of the *Lasp* language, which allows defining programs that compose multiple CRDTs, while keeping the good convergence properties of CRDTs (see Appendix B and Appendix F).[1] The Derflow$_L$ system of the first year allows to do dataflow programming with single CRDT instances [7]; Lasp extends this by allowing to *combine* CRDT instances using powerful operations inspired by functional and logic programming. Our first implementation of Lasp is freely available as an Erlang library (see Section 4). To show the expressiveness of this library, we have used it to implement the Ad Counter scenario of WP1.

5.2 Motivation and basic principles

Traditional approaches to synchronization increasingly have problems when clients become geographically distributed and more numerous. CRDTs solve the problem for the case of single data structures. However, this is insufficient for distributed applications, which consist of many data structures connected together, running in a distributed setting. What is needed is the ability to write these applications while keeping the good convergence properties of CRDTs. It has been shown that the arbitrary composition of CRDTs is nontrivial [1, 13].

The Lasp programming model solves the composition problem for several important CRDTs, in particular our first implementation uses Observed-Remove sets (OR-set), PN Counters, and simpler versions of these CRDTs. The OR-set is powerful enough to express many realistic programming scenarios. Lasp allows deterministically composing instances of OR-sets and other CRDTs into larger computations that observe the Strong Eventual Consistency property. Available compositions include the higher-order combinators from functional programming, namely map, filter, and fold, and set-theoretic operations from logic programming, namely product, union, and intersection.

Lasp programs do computation with data structures whose values appear nonmonotonic externally, while computing internally with the data structures' monotonic metadata. We recall that CRDTs are based on lattice operations and also do monotonic computations internally, even though the external values may be nonmonotonic. The Lasp model connects CRDTs through active processes that compute with the CRDTs internal metadata. We have made a prototype implementation of Lasp that leverages the CRDT functionality of Basho Technologies, namely the *riak_dt* library. We have made two distributed implementations of Lasp. The first uses the Riak Core distributed systems framework to give a data-center implementation of Lasp (published in PPDP 2015; see Appendix B). The second uses the Plumtree epidemic broadcast protocol to give a completely decentralized gossip-based implementation of Lasp (published in WPSDS 2015; see Appendix D).

[1]The name *Lasp* is inspired by the etymology of Lisp, whose name comes from the phrase *List Processing*. Since our fundamental data structure is a lattice, we derive our name from the phrase *Lattice Processing*.

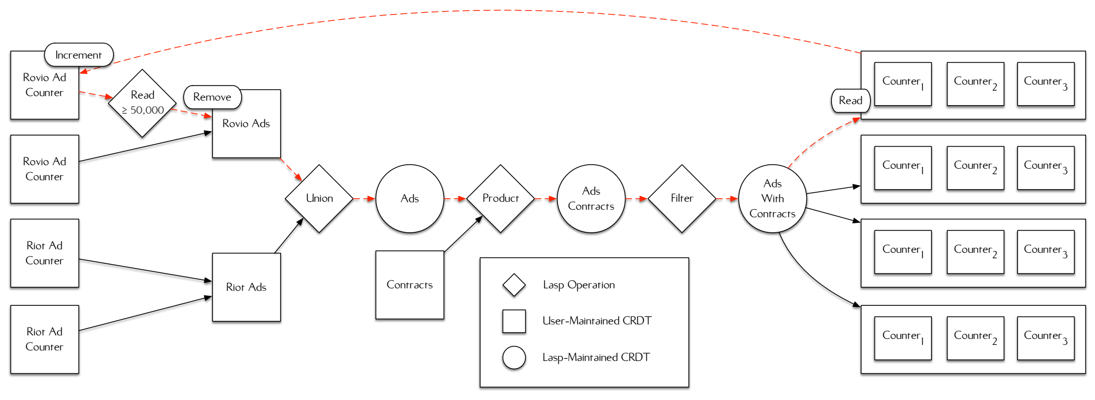


Figure 1: *Eventually consistent advertisement counter. The dotted line represents the monotonic flow of information for one counter.*

5.3 Application scenario: Ad counter

As a first test case for Lasp, we implemented the Ad Counter scenario from Work Package 1. Imagine a provider of mobile games that sells advertisement space within their games. In this scenario, the correctness criteria are twofold:

- Clients will go offline: consider mobile devices such as cellular phones that experience periods without connectivity. When the client is offline, advertisements should still be displayable.
- Advertisements need to be displayed a minimum number of times. Additional impressions are not problematic.

Figure 1 presents our initial design for an eventually consistent ad counter written in Lasp. In this example, squares represent primitive CRDTs and circles represent CRDTs that are maintained using Lasp operations. Additionally, Lasp operations are represented as diamonds and edges represent the monotonic flow of information. Our advertisement counter operates as follows:

- Advertisement counters are grouped by vendor.
- All advertisement groups are combined into one list of advertisements using a union operation.
- Advertisements are joined with active “contracts” into a list of displayable advertisements using both the product and filter operations.
- Each client selects an advertisement to display from the list of active advertisements.
- For each advertisement displayed, each client updates its local copy of the advertisement counter.
- Periodically, advertisement counters are merged upstream.
- When a counter hits at least 50,000 advertisement impressions, the advertisement is “disabled” by removing it from the list of advertisements.

The implementation of this advertisement counter is completely monotonic and synchronization-free. Adding and removing ads, adding and removing contracts, and disabling ads when their contractual number of views is achieved are all modeled as the monotonic growth of state in CRDTs connected by active processes. Programmer-visible nonmonotonicity is represented by monotonic metadata in the CRDTs.

The full implementation of the advertisement counter is available in the Lasp source code repository and consists of 213 LOC. In this example, transparent distribution and failure handling is supported by the runtime environment, and not exposed to the developer. For brevity, we provide only two code samples: the advertisement counter "server" process, that is responsible for disable advertisements when their threshold is reached, and example use of the product and filter operations used for composing the advertisements with their contracts.

5.4 Implementation

We have built two prototype implementations of Lasp based on quite different distribution models. The first implementation is based on a consistent-hashed ring hosted in a data center (see Section 5.4.1). The second implementation is based on gossip: an epidemic broadcast algorithm on a dynamic network of nodes (see Section 5.4.2). It is remarkable that the same programming model is able to support successfully such different distribution structures. We consider this as evidence for the usefulness of the combination of Lasp's weak synchronization model and strong convergence properties, both inherited from CRDTs. We give here an overview of both implementations; more information can be found in Appendix B and Appendix D, and in the Lasp repository (see Section 4).

5.4.1 Data center implementation

The data center implementation distributes data using the Riak Core distributed systems framework. The Riak Core library provides a framework for building applications in the style of the original Dynamo system. Riak Core provides library functions for cluster management, dynamic membership and failure detection.

Dynamo-style partitioning and hashing Lasp uses Dynamo-style partitioning of CRDTs: consistent hashing and hash-space partitioning are used to distribute copies of CRDTs across nodes in a cluster to ensure high availability and fault tolerance. Replication of each CRDT is performed between adjacent nodes in a cluster. While the partitioning mechanism and implementation is nuanced, it is sufficient to realize the collection of CRDTs as a series of disjoint replica sets, of which the data is sharded across, with full replication between the nodes in any given replica set.

Anti-Entropy Protocol We provide an active anti-entropy protocol built on top of Riak Core that is responsible for ensuring all replicas are up-to-date. Periodically, a process is used to notify replicas that contain CRDT replicas with the value of a CRDT from a neighboring replica.

Quorum system operations In Section 5.5, we outline the three properties of our system: crash-stop failures, anti-entropy, and correctness. While these properties are sufficient to ensure confluence of computations, they do not guarantee that all updates will be observed if a given replica of a CRDT fails before communicating its state to a peer replica. Therefore, to guarantee safety and be tolerant to failures, both read and update operations are performed against a quorum of replicas. This ensures fault tolerance: by performing read and write operations against a majority, the system is tolerant to failures. The system remains safe and does not make progress when the majority is not available. Additionally, quorum operations can be used to increase liveness in the system: by writing back the merged value of the majority, we can passively repair objects during normal system operation, improving anti-entropy.

Replication and Execution of Operations Given replication of the objects themselves, to ensure fault-tolerance and high availability, our functional programming operations and set-theoretic operations must be replicated as well. To achieve this, quorum replication is used to contact a majority of replicas near the output CRDT, which are responsible for reading the input CRDT and performing the transformation. We spawn processes at a majority of the output replicas of a CRDT which reads from an input CRDT.

To ensure forward progress of these computations, each of our operations uses the strict version of the monotonic read operation to prevent from executing over stale values when talking to replicas which are out-of-date. In the map example, the transformation is performed for a given observation in the stream of updates to variable S1 with the output written into the stream for variable S2, at which the process tail-recursively executes and wait to observe a causally greater value than the previously observed S1 before proceeding. This prevents duplication of already computed work and ensure forward progress at each replica. Additionally, we can apply read repair and anti-entropy techniques to repair the value of the output CRDT if it falls very far behind instead of relying on applying operations from the input CRDT in order.

5.4.2 Gossip implementation

The gossip implementation consists of a set of nodes supporting Lasp operations that are implemented using epidemic broadcast. Each node is uniquely identified and tracks a monotonic counter that is incremented with each operation. Nodes can join or leave at any time. Nodes fail by crashing and all messages in the system are eventually delivered to all correct nodes by the epidemic broadcast protocol (reliable broadcast). Crashed nodes disappear from the system; whenever a node recovers it chooses a new identifier and reinitializes its monotonic counter at zero. We can therefore summarize the implementation model as consisting of two layers, which we call the Lasp layer and the gossip layer. The Lasp layer implements Lasp processes as recursive functions that use the Lasp operations provided by that layer on top of the epidemic broadcast.

The gossip layer implements the Plumtree epidemic broadcast protocol. Plumtree is an efficient reliable broadcast protocol that combines the efficiency of a deterministic tree-based broadcast with the resilience of a gossip algorithm [22]. It efficiently implements the broadcast operations required by the Lasp layer. The broadcast operations are not ordered, i.e., a node may receive broadcasts in any order and different nodes may receive them in different orders. Because of the Strong Eventual Consistency property of CRDTs, this does not affect correctness. Furthermore, this allows an important optimization that reduces the computations needed to implement the bind operation. Bind operations initiated on each node are numbered consecutively via a node-level monotonic counter. Since each variable's successive values are inflations of a lattice, a bind operation that is delivered on a node does not have to invoke local computation if another bind with a greater value has already been delivered.

The Plumtree protocol relies on three properties for fault-tolerant message delivery: (1) Each message can be uniquely identified: given the lazy push phase of the protocol broadcasts only message identifiers, a node is required to know whether that message has been received or not. (2) Nodes must store a history of all messages received. (3) When receiving an identifier for a message, a node must be able to determine if it has already been subsumed by a previous one.

To meet these requirements, we maintain a monotonic clock at each node and store a version vector for each CRDT. This version vector is used to uniquely identify the message when broadcast and allows us to identify messages that have been subsumed by other messages without comparison of payload. By leveraging a per object version vector that is incremented as mutations are performed to each object, we can store a history of all messages received with vector as wide as the number of participating actors in the system.

5.5 Fundamental theorem

How easy is programming in Lasp? Can it be as easy as programming in a non-distributed language? Is it possible to ignore the replica-to-replica communication, distribution, and failures of CRDTs? Because of the strong semantic properties of CRDTs, it turns out that this is indeed possible. In Appendix B, we have formalized the distributed execution of a Lasp program and proved that there is a centralized execution, i.e., a single sequence of states, that produces the same result as the distributed execution. This allows us to use the same reasoning and programming techniques as centralized programs. We have also proved that failures have limited effect: in the worst case, a CRDT update is simply ignored.

The programmer can reason about instances of CRDTs as monotonic data structures linked by monotonic functions, which is a form of deterministic dataflow programming. This model has the good properties of functional programming (e.g., confluence and referential transparency) in a concurrent setting.

5.5.1 Assumptions

For our formalization, we assume the following properties.

Property 5.1. Fault model and repair We assume the following three conditions:

- **Crash-stop failures:** replicas fail by crashing and any replica may fail at any time.
- **Anti-entropy:** after every crash, a fresh replica is eventually created with state copied from any correct replica.
- **Correctness:** at least one replica is correct at any instant.

The first condition is imposed by the environment. The second condition is the repair action done by every CRDT when one of its replicas crashes. The third condition is what must hold globally for the CRDT to continue operating correctly.

Property 5.2. Weak synchronization For any execution of a CRDT instance, it is always true that eventually every replica will successfully send a message to each other replica.[2]

Property 5.3. Determinism Given two executions of a CRDT instance with the same sequence of updates but a different merge schedule, i.e., a different sequence of replica-to-replica communication, replicas in the first execution that have delivered the same updates as replicas in the second execution have equal state.

Since we intend Lasp programming to be similar to functional programming, it is important that computations are deterministic. We remark that SEC by itself is not enough to guarantee this. To correctly use an OR-Set in Lasp, it is important to impose conditions that ensure determinism. The following two conditions are sufficient to guarantee determinism for all merge schedules:

- A $remove(v)$ is only allowed if an $add(v)$ with the same value has been done previously at the same replica.
- An $add(v)$ with the same value of v may not be done at two different replicas.

5.5.2 Fundamental theorem

We present our main formal result without proof; a proof can be found in Appendix B.

Definition 5.1. Simple Lasp program A simple Lasp program consists of either:

- A single CRDT instance, or
- A Lasp process with m inputs that are simple Lasp programs and one output CRDT instance.

Theorem 5.1. *A simple Lasp program can be reduced to a single state execution.*

As a consequence of this theorem, it suffices to reason about single state executions. Since Lasp processes are connected through functional operations, this reasoning corresponds to the reasoning of functional programming. We note that this result holds for both the data center implementation and the gossip implementation, as long as the fault properties hold. Given the Fault and Repair Model (Property 5.1), we have also proved that any update to a CRDT in a simple Lasp program is eventually delivered to all replicas or to no replicas. This means that the worst effect of faults is that an update may be ignored; no other erroneous execution will occur.

5.6 Conclusions and future work

We introduced the Lasp programming model for large-scale synchronization-free computation over replicated data. We have implemented two distribution models for the programming model, namely a data-center-based model using Riak Core, and a gossip-based model using Plumtree. We have implemented several scenarios, of which the largest is the Ad Counter scenario from workpackage 1, to show the expressiveness of Lasp.

In the future, we intend to extend Lasp into a full-fledged language and system. We will identify optimizations for more efficient state propagation, explore stronger consistency models, and optimize distribution and replica placement for better fault tolerance

[2]The content of this message depends on the definition of the CRDT.

and reduced latency. We also intend to extend Lasp to add synchronization where needed, in particular causal consistency and transactions. For these extensions, we intend to use the Antidote system as a backend, since it has both these abilities. We also intend to add higher-order operations and abstractions for long-lived applications (deployment, reconfiguration, and software rejuvenation), and to do realistic evaluations, in particular for applications on edge computing. In the short term, we are working on extending Lasp for rolling-window aggregation in a large sensor network, which is a typical edge computing application.

6 Consistency

In the work on consistency we present the first proof rule for establishing that a particular choice of consistency over a replicated database is enough to ensure the preservation of a data integrity invariant (see Appendix I). We call the resulting proof system CISE, an acrynoym for 'Cause I'm Strong Enough.

6.1 Overview

Recent research [5, 23, 31, 33] and commercial [2, 6, 26] databases provide *hybrid* consistency models that allow the programmer to request stronger consistency for certain operations, while other operations may execute without any synchronisation. For example, a consistency model may execute some operations under causal consistency, and some under strong consistency [23]. For example, to preserve the integrity invariant in a banking application when using this model, only withdrawal operations need to use strong consistency, and hence, synchronise to ensure that the account is not overdrawn; deposit operations may use causal consistency and hence proceed without synchronisation. More recently, the *Explicit Consistency* model [5] allows the programmer to use sophisticated multi-level locks and reservations to fine-tune consistency level of each operation (see Section 7 for the current status of this work). Using this consistency model effectively is far from trivial. Requesting stronger consistency in too many places may hurt performance and availability, and requesting it in too few places may violate correctness. Striking the right balance requires the programmer to reason about the application behaviour on the subtle semantics of the consistency model, understanding which anomalies are disallowed by a particular consistency strengthening and whether disallowing these anomalies is enough to ensure correctness. This difficulty is compounded by the perennial challenge of reasoning about concurrency, present even with strong consistency—having to consider the huge number of possible interactions between concurrently executing operations.

To help programmers exploit hybrid consistency models, we proposed the first proof rule and tool for proving integrity invariants of applications using replicated databases with a range of hybrid models [19]. In more detail, we defined a generic hybrid consistency model that is flexible enough to encode a variety of consistency models for replicated databases proposed in the literature [5, 23, 25, 31]. It guarantees causal consistency by default and allows the programmer to additionally specify which pairs of operations may not execute without synchronisation by means of a special *conflict relation*.

Our key technical contribution is a proof rule for showing that a set of operations preserves a given integrity invariant when executed on our consistency model with a given choice of conflict relation. To avoid explicit reasoning about all possible interactions between operations, our proof rule is *modular*: it allows us to reason about the behaviour of every operation separately under some assumption on the behaviour of other operations, which takes into account the conflict relation. In this way, our proof rule allows the programmer to reason precisely about how strengthening or weakening consistency of certain operations affects correctness.

The modular nature of our proof rule allows it to reason in terms of states of a single database copy, just like in proof rules for strongly consistent shared-memory concurrency. In [19] we have proved that this simple reasoning is sound, despite the weakness of the consistency model.

We have also developed a tool that automates our proof rule by reducing checking its obligations to SMT queries. Using the tool, we have verified several example applications that require strengthening consistency in nontrivial ways. These include a banking application, an online auction service and a course registration system. In particular, we were able to handle applications using *replicated data types* (aka CRDTs [29]), which encapsulate policies for automatically merging the effects of operations performed without synchronisation at different replicas. The fact that we can reduce checking the correctness properties of complex computations in our examples to querying off-the-shelf SMT tools demonstrates the simplicity of reasoning required by our approach.

6.2 Consistency model

Even though this model is not implemented in its full generality by an existing database, it can encode a variety of models that have in fact been implemented. In this section we present the programming interface of our consistency model and describe its semantics informally, from an operational perspective.

6.2.1 Causal consistency and its implementation

Our hybrid model guarantees at least *causal consistency* [25]. Therefore we start by presenting informally how a typical implementation of a causally consistent database operates. Let State be the set of possible states of the data managed by the database system. We denote states by σ and let σ_{init} be a distinguished initial state. Applications define a set of operations $\mathsf{Op} = \{o, \dots\}$ on the data and interact with the database by issuing these operations. For simplicity, we assume that an operation always terminates and returns a single value from a set Val. We use a value $\bot \in \mathsf{Val}$ to model operations that return no value. We do not consider operation parameters, since these can be part of the operation name.

The database implementation consists of a set of replicas, each maintaining a complete copy of the database state; we identify the replicas by $r_1, r_2, \dots$ For the purposes of the informal explanation, we assume that replicas never fail. A client operation is initially executed at a single replica, which we refer to as its *origin* replica. At this replica, the execution of the operation is not interleaved with that of others. This execution updates the replica state deterministically, and immediately returns a value to the client. After this, the replica sends a message to all other replicas containing the *effect* of the operation, which describes the updates done by the operation to the database state. The replicas are guaranteed to receive the message at most once. Upon receipt, the replicas apply the effect to their state.

We assume the semantics of operations is given by a function

$$\mathcal{F} \in \mathsf{Op} \to (\mathsf{State} \to (\mathsf{Val} \times (\mathsf{State} \to \mathsf{State}))). \tag{1}$$

To aid readability, for $o \in \mathsf{Op}$ we write $\mathcal{F}_o$ instead of $\mathcal{F}(o)$ and let

$$\forall o, \sigma.\ \mathcal{F}_o(\sigma) = (\mathcal{F}_o^{\mathsf{val}}(\sigma), \mathcal{F}_o^{\mathsf{eff}}(\sigma)).$$

Given a state σ of o's origin replica, $\mathcal{F}_o^{\mathsf{val}}(\sigma) \in \mathsf{Val}$ determines the return value of the operation and $\mathcal{F}_o^{\mathsf{eff}}(\sigma) \in \mathsf{State} \to \mathsf{State}$ its effect. The latter is a function, to be applied

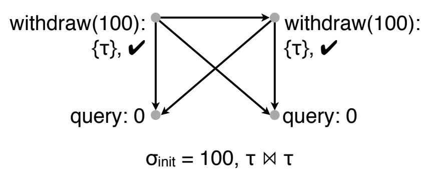


Figure 2: An illustration of an execution of Definition 1.

by every replica to its state to incorporate the operation's effect: immediately at the origin replica, and after receiving the corresponding message at all other replicas.

For example, consider a banking application where states are integers, representing the account balance: $\mathsf{State} = \mathbb{Z}$. We define the semantics of operations for depositing an amount $a > 0$, accruing a 5% interest and querying the balance:

$$\begin{aligned} \mathcal{F}_{\mathsf{deposit}(a)}(\sigma) &= (\bot, (\lambda\sigma'.\, \sigma' + a)); \\ \mathcal{F}_{\mathsf{interest}}(\sigma) &= (\bot, (\lambda\sigma'.\, \sigma' + 0.05 * \sigma)); \\ \mathcal{F}_{\mathsf{query}}(\sigma) &= (\sigma, \mathsf{skip}), \end{aligned} \tag{2}$$

where $\mathsf{skip} = (\lambda\sigma'.\, \sigma')$.

To ensure convergence, for now we require that the effects of all operations commute (we relax this condition for conflicting operations). For example, commutativity holds of the effects defined by (2). The requirement of commutativity can be addressed through the use of ready-made *replicated data types* (aka CRDTs [29]). These encapsulate commutative implementations of policies for merging concurrent updates to the database.

6.2.2 Strengthening consistency

The guarantees provided by causal consistency are too weak to ensure certain integrity invariants. For example, in our banking application we would like the state at each replica to satisfy the invariant

$$I = \{\sigma \mid \sigma \geq 0\}. \tag{3}$$

To ensure this, an operation for withdrawing an amount $a > 0$ could check whether the account has sufficient funds and return ✓ or ✗ depending on the result:

$$\mathcal{F}_{\mathsf{withdraw}(a)}(\sigma) = \text{if } \sigma \geq a \text{ then } (\checkmark, (\lambda\sigma'.\, \sigma' - a)) \text{ else } (\text{✗}, \mathsf{skip}).$$

This is enough to maintain the invariant when all operations are processed at the same replica, but not when they are processed asynchronously at different replicas.

The problem in this example arises because two particular operations update the database concurrently, without being aware of each other. To address this, our consistency model allows the programmer to strengthen causal consistency by specifying explicitly which operations may not be executed in this way. Namely, the model is parameterised by a *token system* $\mathcal{T} = (\mathsf{Token}, \bowtie)$, consisting of a set of *tokens* Token and a symmetric *conflict relation* $\bowtie \subseteq \mathsf{Token} \times \mathsf{Token}$. Tokens are ranged over by τ and their sets, by T.

Each operation may acquire a set of tokens. To account for this, we redefine the type of $\mathcal{F}$ in (1) as

$$\mathcal{F} \in \mathsf{Op} \rightarrow (\mathsf{State} \rightarrow (\mathsf{Val} \times (\mathsf{State} \rightarrow \mathsf{State}) \times \mathcal{P}(\mathsf{Token}))) \tag{4}$$

$$
\begin{aligned}
\mathsf{Token} &= \{\tau\} \\
\bowtie &= \{(\tau,\tau)\} \\
\mathcal{F}_{\mathsf{deposit}(a)}(\sigma) &= (\bot, (\lambda\sigma'.\, \sigma' + a), \emptyset) \\
\mathcal{F}_{\mathsf{interest}}(\sigma) &= (\bot, (\lambda\sigma'.\, \sigma' + 0.05 * \sigma), \emptyset) \\
\mathcal{F}_{\mathsf{query}}(\sigma) &= (\sigma, \mathsf{skip}, \emptyset) \\
\mathcal{F}_{\mathsf{withdraw}(a)}(\sigma) &= \text{if } \sigma \geq a \text{ then } (\checkmark, (\lambda\sigma'.\, \sigma' - a), \{\tau\}) \\
& \text{ else } (\text{✗}, \mathsf{skip}, \{\tau\})
\end{aligned}
$$

Figure 3: Operation semantics for the banking application. Note that $a > 0$.

and let

$$\forall o, \sigma.\, \mathcal{F}_o(\sigma) = (\mathcal{F}_o^{\mathsf{val}}(\sigma), \mathcal{F}_o^{\mathsf{eff}}(\sigma), \mathcal{F}_o^{\mathsf{tok}}(\sigma)).$$

Thus, $\mathcal{F}_o^{\mathsf{tok}}(\sigma) \in \mathcal{P}(\mathsf{Token})$ gives the set of tokens acquired by the operation o when executed in the state σ. Informally, our consistency model guarantees that operations that acquire tokens conflicting according to $\bowtie$ have to be causally dependent one way or another: the origin replica of one operation must have incorporated the effect of the other by the time the former operation executes. Ensuring this in implementations requires replicas to synchronise [5, 23].

In our consistency model, we can guarantee the preservation of invariant (3) in the banking application by defining operation semantics as in Figure 3. Thus, withdraw acquires a token τ conflicting with itself, and all other operations do not acquire any tokens. Then the scenario where two replicas make a withdraw without being aware of each other cannot happen: one withdrawal would have to be aware of the other and would therefore fail. However, deposits and interest accruals can be causally independent with all operations, and replicas can therefore execute them without any synchronisation [5, 23]. In this example, the token τ is analogous to a mutual exclusion lock in shared-memory concurrency. Our proof method establishes that this use of the token is indeed sufficient to preserve the integrity invariant (3).

Since operations acquiring conflicting tokens have to be causally dependent, causal message propagation ensures that all replicas see such operations in the same order. This allows us to require commutativity only for operations that do not acquire conflicting tokens.

6.3 Formal semantics

Next we define formal semantics of the consistency model. Our formalism does not refer to implementation-level concepts, such as replicas or messages. We build on an approach previously used to specify forms of eventual consistency [10]. Namely, our denotations of database computations consist of a set of events, representing operation invocations by clients, and a relation on events, describing abstractly how the database processes the corresponding operations.

Assume a countably infinite set Event of *events*, ranged over by e, f, g. A relation is a *strict partial order* if it is transitive and irreflexive. For a relation R we write $(e, f) \in R$ and $e \xrightarrow{R} f$ interchangeably.

DEFINITION 1. Given a token system $\mathcal{T} = (\mathsf{Token}, \bowtie)$, an **execution** is a tuple $X = (E, \mathsf{oper}, \mathsf{rval}, \mathsf{tok}, \mathsf{hb})$, where:

- E is a finite subset of Event;
- $\mathsf{oper} : E \to \mathsf{Op}$ gives the operation whose invocation a given event denotes;
- $\mathsf{rval} : E \to \mathsf{Val}$ gives the return value of the operation;
- $\mathsf{tok} : E \to \mathcal{P}(\mathsf{Token})$ gives the set of tokens acquired by the operation;
- $\mathsf{hb} \subseteq E \times E$, called ***happens-before***, is a strict partial order such that

$$\forall e, f \in E.\ \mathsf{tok}(e) \bowtie \mathsf{tok}(f) \implies (e \xrightarrow{\mathsf{hb}} f \vee f \xrightarrow{\mathsf{hb}} e). \tag{5}$$

Operationally, each event represents an invocation of an operation at its origin replica. The applications of the operation's effect at other replicas are not recorded in an execution explicitly. Instead, the happens-before relation records causal dependencies between operations arising from such applications: $e \xrightarrow{\mathsf{hb}} f$ means that either the operations denoted by e and f were executed at the same replica in this order, or they were executed at different replicas and the message containing the effect of e had been delivered to the replica performing f before f was executed. Hence, if we have $e \xrightarrow{\mathsf{hb}} f$, then the effect of e is incorporated into the state to which f is applied and may influence its return value.

The condition (5) formalises the stronger consistency guarantee provided by tokens: operations acquiring conflicting tokens have to be causally dependent. For example, since the two withdraw operations in Figure 2 acquire a token τ with $\tau \bowtie \tau$, they have to be related by happens-before. Finally, we require executions to contain only finitely many events, because in this paper we are only concerned with safety properties of applications.

6.4 State-based proof rule

We consider the following verification problem: given a token system $\mathcal{T} = (\mathsf{Token}, \bowtie)$, prove that operations $\mathcal{F}$ maintain an integrity invariant $I \subseteq \mathsf{State}$ over database states. In [19] we established that any execution consistent with $\mathcal{T}$ and $\mathcal{F}$ evaluates to a state satisfying I. For example, we show that any execution consistent with Figure 3 evaluates to a state satisfying the invariant (3). Hence, a query operation will always return a non-negative balance.

The key challenge of the above verification problem is the need to consider infinitely many executions consistent with $\mathcal{T}$ and $\mathcal{F}$. Our main technical contribution is the proof rule for solving this problem that avoids considering all such executions explicitly. Instead, the proof rule is *modular* in that it allows us to reason about the behaviour of every operation separately. Our proof rule is also *state-based* in that it reasons in terms of states obtained by evaluating parts of executions or, from the operational perspective, in terms of replica states.

We give our proof rule in Figure 4 and explain it from the operational perspective. The rule assumes that the invariant I holds of the initial database state σ_{init} (condition S1). Consider a computation of the database implementation from Section 6.2 and a state σ of a replica r at some point in this computation. The proof rule assumes that $\sigma \in I$ and aims to establish that executing any operation o at r will preserve the invariant I. This is easy if we only consider how o's effect changes the state of r, since this effect is applied to the state σ where it was generated:

$$\forall \sigma.\ (\sigma \in I \implies \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma) \in I). \tag{6}$$

$$\dfrac{\begin{array}{l}\exists G_0 \in \mathcal{P}(\mathsf{State} \times \mathsf{State}), G \in \mathsf{Token} \to \mathcal{P}(\mathsf{State} \times \mathsf{State}) \\ \text{such that} \\ \quad \text{S1. } \sigma_{\mathsf{init}} \in I \\ \quad \text{S2. } G_0(I) \subseteq I \wedge \forall \tau.\ G(\tau)(I) \subseteq I \\ \quad \text{S3. } \forall o, \sigma, \sigma'.\ (\sigma \in I \wedge (\sigma, \sigma') \in (G_0 \cup G((\mathcal{F}^{\mathsf{tok}}_o(\sigma))^\perp))^*) \\ \qquad\qquad \implies (\sigma', \mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma')) \in G_0 \cup G(\mathcal{F}^{\mathsf{tok}}_o(\sigma))\end{array}}{\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathsf{eval}^{-1}_{\mathcal{F}}(I)}$$

Figure 4: State-based proof rule for a token system $\mathcal{T} = (\mathsf{Token}, \bowtie)$. For $T \subseteq \mathsf{Token}$ we let $G(T) = \bigcup_{\tau \in T} G(\tau)$ and $T^\perp = \{\tau \mid \tau \in \mathsf{Token} \wedge \neg\exists \tau' \in T.\ \tau \bowtie \tau'\}$. We denote by R^* the reflexive and transitive closure of a relation R. For a relation $R \in \mathcal{P}(A \times B)$ and a predicate $P \in \mathcal{P}(A)$, the expression $R(P)$ denotes the image of P under R.

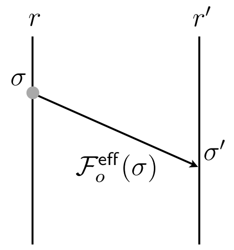


Figure 5: Graphical illustration of the state-based rule.

The difficulty comes from the need to consider how o's effect changes the state of any other replica r' that receives it; see Figure 5(a). At the time of the receipt, r' may be in a different state σ', due to operations executed at r' concurrently with o. We can show that it is sound to assume that this state σ' also satisfies the invariant. Thus, to check that the operation o preserves the invariant when applied at any replica, it is sufficient to ensure

$$\forall \sigma, \sigma'.\ (\sigma, \sigma' \in I \implies \mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma') \in I). \tag{7}$$

However, establishing this without knowing anything about the relationship between σ and σ' is a tall order. In the bank account example, both $\sigma = 100$ and $\sigma' = 0$ satisfy the integrity invariant (3). Then $\mathcal{F}^{\mathsf{eff}}_{\mathsf{withdraw}(100)}(\sigma)(\sigma') = -100$, which violates the invariant. Condition (7) fails in this case because it does not take into account the tokens acquired by withdraw.

The proof rule in Figure 4 addresses the weakness of (7) by allowing us to assume a certain relationship between the state where an operation is generated (σ) and where its effect is applied (σ'), which takes into account the tokens acquired by the operation. To express this assumption, the rule uses a form of rely-guarantee reasoning [20]. Namely, it requires us to associate each token τ with a *guarantee* relation $G(\tau)$, describing all possible state changes that an operation acquiring τ can cause. Crucially, this includes not only the changes that the operation can cause on the state of its origin replica, but also any change that its effect causes at any other replica it is propagated to. We also have a guarantee relation G_0, describing the changes that can be performed by an operation without acquiring any tokens. Condition S2 requires the guarantees to preserve the invariant.

Like (7), condition S3 considers an arbitrary state σ of o's origin replica r, assumed to satisfy the invariant I. The condition then considers any state σ' of another replica r' to which the effect of o is propagated. The conclusion of S3 requires us to prove that applying the effect $\mathcal{F}_o^{\mathsf{eff}}(\sigma)$ of the operation o to the state σ' satisfies the union of the guarantees associated with the tokens $\mathcal{F}_o^{\mathsf{tok}}(\sigma)$ that the operation o acquires. By S2, this implies that the effect of the operation preserves the invariant. Condition S3 further allows us to assume that the state σ' of r' can be obtained from the state σ of r by applying a finite number of changes allowed by G_0 or the guarantees for *those tokens that do not conflict with any of the tokens acquired by the operation* o, i.e., $G_0 \cup G((\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp})$. Informally, acquiring a token denies other replicas permissions to concurrently perform changes that require conflicting tokens.

We now use our proof rule to show that the operations in the banking application (Figure 3) preserve the integrity invariant (3). We assume that the initial state σ_{init} satisfies the invariant. The guarantees are as follows:

$$\begin{aligned} G(\tau) &= \{(\sigma, \sigma') \mid 0 \leq \sigma' < \sigma\}; \\ G_0 &= \{(\sigma, \sigma') \mid 0 \leq \sigma \leq \sigma'\}. \end{aligned} \tag{8}$$

Since withdrawals acquire the token τ, the guarantee $G(\tau)$ for this token allows decreasing the balance without turning it negative; the guarantee G_0 allows increasing a non-negative balance. Then condition S2 is satisfied. We show how to check the condition S3 in the most interesting case of $o = \mathsf{withdraw}(a)$. Consider σ and σ' satisfying the premiss of S3:

$$\sigma \in I \wedge (\sigma, \sigma') \in (G_0 \cup G((\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp}))^*.$$

Since $\mathcal{F}_o^{\mathsf{tok}}(\sigma) = \{\tau\}$, we have that $(\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp} = \emptyset$. Thus, $(\sigma, \sigma') \in G_0^*$. This and $\sigma \in I$ imply that

$$0 \leq \sigma \leq \sigma'. \tag{9}$$

If $\sigma < a$, then $\mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma') = \sigma'$. Furthermore, $\sigma' \geq 0$ by (9). Thus, $(\sigma', \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma')) = (\sigma', \sigma') \in G_0$, which implies the conclusion of S3.

If $\sigma \geq a$, then $\mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma') = \sigma' - a$. Since $\sigma \leq \sigma'$, by (9) we have $\sigma' \geq a$. Thus, $(\sigma', \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma')) = (\sigma', \sigma' - a) \in G(\{\tau\})$, which implies the conclusion of S3. Operationally, in this case our proof rule establishes that, if there was enough money in the account at the replica where the withdrawal was made, then there will be enough money at any replica the withdrawal is delivered to. This completes the proof of our example.

6.5 Courseware application

This example illustrates an integrity invariant and the use of replicated data types [29] to construct commutative operations. Figure 6 shows a fragment of a courseware application. We assume sets of courses Course and students Student. A client can add a course c using $\mathsf{addCourse}(c)$ and register a student s using $\mathsf{register}(s)$. A registered student s can be enrolled into a course c using $\mathsf{enroll}(s, c)$. In the application fragment we consider, student registrations and enrollments cannot be cancelled. However, a course c that has

$$
\begin{aligned}
\mathsf{State} &= \mathcal{P}(\mathsf{Student}) \times \mathsf{RWset}(\mathsf{Course}) \times \mathcal{P}(\mathsf{Student} \times \mathsf{Course}) \\
\sigma_{\mathsf{init}} &= (\emptyset, \emptyset_{\mathsf{RWset}}, \emptyset) \\
I &= \{(S, C, E) \mid E \subseteq \mathcal{P}(S \times \mathsf{contents}(C))\} \\
\mathsf{Token} &= \{\tau_{e(c)}, \tau_{r(c)} \mid c \in \mathsf{Course}\} \\
\bowtie &= \{(\tau_{e(c)}, \tau_{r(c)}), (\tau_{r(c)}, \tau_{e(c)}) \mid c \in \mathsf{Course}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{F}_{\mathsf{register}(s)}((S, C, E)) = \\
&\quad (\bot, (\lambda(S', C', E').\, (S' \cup \{s\}, C', E')), \emptyset) \\
&\mathcal{F}_{\mathsf{addCourse}(c)}((S, C, E)) = \\
&\quad (\bot, (\lambda(S', C', E').\, (S', \mathsf{add}(c, C'), E')), \emptyset) \\
&\mathcal{F}_{\mathsf{enroll}(s,c)}((S, C, E)) = \\
&\quad \text{if } (s \notin S \vee c \notin \mathsf{contents}(C)) \text{ then } (\text{✗}, \mathsf{skip}, \{\tau_{e(c)}\}) \\
&\quad \text{else } (\checkmark, (\lambda(S', C', E').\, (S', C', E' \cup \{(s, c)\})), \{\tau_{e(c)}\}) \\
&\mathcal{F}_{\mathsf{remCourse}(c)}((S, C, E)) = \\
&\quad \text{if } (c \notin \mathsf{contents}(C) \vee \exists s.\, (s, c) \in E)) \text{ then } (\text{✗}, \mathsf{skip}, \{\tau_{r(c)}\}) \\
&\quad \text{else } (\checkmark, (\lambda(S', C', E').\, (S', \mathsf{remove}(c, C'), E')), \{\tau_{r(c)}\}) \\
&\mathcal{F}_{\mathsf{query}}((S, C, E)) = ((S, \mathsf{contents}(C), E), \mathsf{skip}, \emptyset)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{RWset}(\mathsf{Course}) &= \mathcal{P}(\mathsf{Course}) \times \mathcal{P}(\mathsf{Course}) \\
\emptyset_{\mathsf{RWset}} &= (\emptyset, \emptyset) \\
\mathsf{add}(c, (A, T)) &= (A \cup \{c\}, T) \\
\mathsf{remove}(c, (A, T)) &= (A, T \cup \{c\}) \\
\mathsf{contents}((A, T)) &= A - T
\end{aligned}
$$

Figure 6: A fragment of a courseware application.

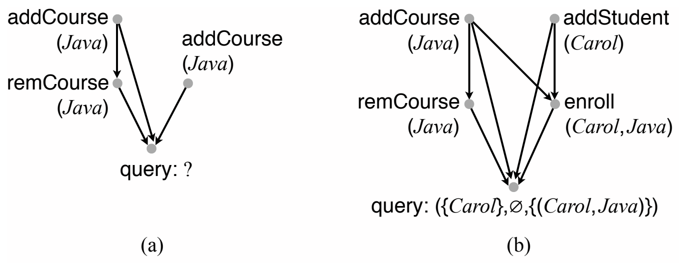


Figure 7: Executions illustrating the need for (a) replicated data types and (b) tokens in the courseware application.

not secured any student enrollment can be removed using $\mathsf{remCourse}(c)$. As usual, we also have a query operation.

A database state (S, C, E) consists of the set of students S, the set of courses C and the enrollment relation E between students and courses. The set of courses is actually not just an ordinary set, but a *replicated remove-wins set* $\mathsf{RWset}(\mathsf{Course})$, explained in the following. The effects of operations are mostly as expected, with courses accessed using special functions add, remove and $\mathsf{contents}$ on the replicated set. Note that the operation $\mathsf{enroll}(c, s)$ takes effect only if the student s is registered and the course c exists. The operation $\mathsf{remCourse}(c)$ removes the course c only when it exists and has no students enrolled into it.

Using a replicated data type for the set of courses is needed to satisfy commutativity, because additions to and removals from a usual set do not commute. To illustrate, consider the execution in Figure 7(a). There Alice adds a course on Java and then changes her mind and removes the course; concurrently, Bob adds the same Java course. If we maintained the information about courses using a usual set, then the outcome of the query in the figure would depend on the order in which we evaluate the effects of the causally independent operations $\mathsf{addCourse}(\mathit{Java})$ and $\mathsf{remCourse}(\mathit{Java})$: the query would return $\emptyset$ if the addition was evaluated before removal, and $\{\mathit{Java}\}$ otherwise. In an actual database, implementing the operations using ordinary sets would violate the replica convergence property.

Replicated data types [29] provide implementations of operations on data structures with commutative effects. They differ in the way in which they resolve conflicting updates to the data structure, such as those in Figure 7(a): when using an *add-wins* set, the query in the figure will return $\{\mathit{Java}\}$, and when using a *remove-wins* set, $\emptyset$ [28]. The decision which data type to use ultimately depends on application requirements. To keep presentation manageable, in our example we use one of the simplest set data types, which provides a rudimentary version of the remove-wins semantics.

The data type represents the replicated set of courses using a pair of sets (A, T). The function $\mathsf{add}(c, \cdot)$ puts c into the set of A, and the function $\mathsf{remove}(c, \cdot)$ puts c into the set T, called the *tombstone* set. To get the contents of the replicated set, we just take the difference of A and T. The functions $\mathsf{add}(c, \cdot)$ and $\mathsf{remove}(c, \cdot)$ commute: even if the removal is evaluated first, it will still cancel the subsequent addition.

The integrity invariant I we would like to maintain in this application is that the enrollment relation refers to existing courses and students only. This property is an instance of *referential integrity*, which requires an object referenced in one part of the database to exist in another. Without using tokens, the operations in our application can break the invariant. This is illustrated by the execution in Figure 7(b). There a Java course initially has no students enrolled. Then Alice removes the course and concurrently Bob enrolls Carol into it, thinking that the course is still available. This results in Carol being enrolled into a non-existent course.

To ensure that such situations do not happen, we use a pair of conflicting tokens for each course $c \in \mathsf{Course}$: $\tau_{e(c)}$ and $\tau_{r(c)}$. The operation $\mathsf{enroll}(s, c)$ acquires $\tau_{e(c)}$, and the operation $\mathsf{remCourse}(c)$ acquires $\tau_{r(c)}$. Then for every pair of operations $\mathsf{enroll}(s, c)$ and $\mathsf{remCourse}(c)$, either the enrollment operation is aware that the course has been removed, or the removal is aware that there are still students enrolled into the course; in either case the corresponding operation takes no effect. However, other pairs of operations can be causally independent and, hence, do not have to synchronise. This includes pairs of operations enrolling students into courses and pairs of operations manipulating courses, such as those in Figure 7(a). The above use of tokens is equivalent to associating every course with a multi-level lock [5] that can be in one of two modes, one of which allows enrolling students into a course ($\tau_{e(c)}$) and the other removing the course ($\tau_{r(c)}$). Unlike in the auction application above, neither of the tokens $\tau_{e(c)}$ or $\tau_{r(c)}$ conflicts with itself, and thus, neither of the above lock modes is exclusive.

Our proof rule can establish that the above consistency choice is sufficient to preserve the integrity invariant.

6.6 Conclusions and future work

The work in this section was submitted near the end of year 2 in the paper [19] (see Appendix I). In this paper we present the first proof rule establishing that a given consistency choice in a replicated database is sufficient to preserve a given integrity invariant. The proof rule is modular and simple to use. This was demonstrated by verifying small but nontrivial examples, and by reducing the verification conditions of the proof rule to SMT checks. Despite this simplicity, the soundness of our proof rule is nontrivial: the rule fully exploits the guarantees provided by our consistency model while correctly accounting for anomalies it allows.

These results represent only an initial step in building an infrastructure of reasoning methods for applications using modern replicated databases. They open several avenues for future work. First, our generic consistency model is not implemented by any database in its full generality; we use it only as a means to compactly represent a selection of more specific models in existing implementations. However, in the future the generic model can serve as a basis for exploring the space of possible hybrid consistency models. One could also consider a database that implements our model in its general form. Second, we will investigate how to combine CISE with the invariant repair techniques (see Section 7), to give a new and powerful program development methodology.

7 Explicit consistency using invariant repair

7.1 Overview

Typically consistency models specify the allowed ordering of operations across in different sites, forcing programmers to reason about the possible interleavings of operations to analyze the correctness of the application. In the first year we proposed Explicit Consistency [5], an alternative consistency model that allows operations to be executed by any order in any replica as long as the application invariants are maintained when replicas converge. This model has minimal constraints, however it is quite powerful when combined with a static analysis tool. It would be too difficult for the programmer to determine what concurrent executions would break the invariants of the application, but with the help of a static analysis tool, the programmer can get that information and prevent those executions with a few extra lines of code.

In the first year of the project we mainly focused on ensuring that applications remain correct without relying on coordinated update execution to preserve invariants. We developed a concurrency control mechanism, called reservations, that allows replicas to execute operations locally in many cases and yet preserve application invariants. Our previous approach is capable of providing low-latency operations in many cases, however it is limited in terms of availability. The problem is that the execution of some operations might depend on the acquisition of some lock that might be remote, therefore, when the network is partitioned, it might be impossible to acquire that lock and the system becomes unavailable to process that request.

In the second year we started thinking in improving the fault-tolerance properties of the system and the usability of the tool, to make it easier to be adopted by programmers. We present the new ideas that we are pursuing to provide better availability and present a compelling example to explain them.

7.2 Prevention is not always better than cure

In the last year, we proposed controlling the execution of conflicting operations, i.e. that their concurrent execution does not violate the application invariants, through a mechanism called Reservations. Reservations allow only certain instances of operations to proceed locally if their execution is safe in respect to the invariants, otherwise the system must acquire a reservation to enable their execution. We implemented two reservation mechanisms:

- Escrow reservations: Based on the classical escrow-model [27], we devise a new data-type that is capable of ensuring numerical invariants through resource allocation to each replica.
- Multi-level lock: A token that allows a unique or multiple clients to execute one or various operations that are not conflicting in respect to each other and precludes the execution of the conflicting.

Reservations can be exchanged outside the critical path of operation execution, hence operation have low-latency, when the replica holds the reservations before the request arrives.

We have identified two limitations with the reservation mechanisms that we propose:

- **Unavailability** When some remote replica is down, it might be impossible for a replica to acquire a reservation. This might prevent a replica from accepting some client requests, therefore the system loses availability. All reservation mechanisms suffer this limitation, as a reservation is some sort of contract of what operations each replica can execute, therefore replicas need to coordinate to establish the agreement.
- **Multi-level lock convergence** When locks exhibit good locality, i.e. client requests are able to satisfy all necessary locks locally, the system performs well with low latency. However, if the operation requires some lock to be free remotely, the latency might become very high, because locks are exchanged in a peer-to-peer fashion. Although one can provide better heuristics to exchange locks proactively, conflicting operations need to wait for each other before proceeding.

Giving the limitations of the reservations technique we have started studying alternatives that preserves availability at all time. The main idea is that the system cannot rely on any form of coordination, even outside the critical path of execution, to ensure invariant preservation. In order to accomplish that, we allow operations to execute optimistically, in the local replica and fix invariant violations when they are detected.

7.3 Invariant repair technique

The invariant repair approach is based on the idea of compensations. When the system arrives at some state that is not correct, it can automatically apply some effects to restore the correctness of the database.

The Sagas model [17] proposes the use of compensations to improve the commit rate of systems that must handle long-lived transactions. The idea is that long-lived transactions can be split into smaller transactions to avoid holding locks for too long. But if some transaction conflicts with any of of the smaller pieces of the chopped transaction, the system might have to undo that transaction, by applying a compensating transactions that revert the mutations that have already committed.

Bayou [32] is a replicated system that accepts write operations at each server and performs semantic-level conflict resolutions, similar to what we want to do. The system is composed by a number of servers that accept requests from local clients. When an operation is sent by a client to the server, it enters in a tentative state: First, it might conflict with concurrent operations, in which case it might not be able to execute because the state of the server does not respect the pre-conditions of the operation. When this occur, a programmer-defined procedure is applied to solve the conflict. Second, more conflicting updates might arrive later in which case the tentative updates might be reordered and re-executed to establish a total ordering of updates. The effect of the operation only becomes permanent after a primary server establishes the final ordering of this operation.

Both approaches have limitations in terms of scalability and availability: Sagas does not address a replication context and therefore would need to be layered on top of some sort of state-machine replication scheme; Bayou relies on a total order to make operations stable, which is not highly-available. Also neither system addresses a partial replication, which might impose extra coordination steps to execute the conflict detection and resolution protocols.

We propose an alternative solution that addresses these concerns: 1. Operations have immediate and permanent effect over the database state; 2. Operation effects can be propagated asynchronously to any replica with the guarantee that the invariants are preserved. 3. Database partitioning does not require any extra communication to detect for invariant violations.

We are capable of handling non-idempotent conflicts as long as the resolution is idempotent, however we cannot provide non-idempotent conflict resolutions, this is because it is necessary to enforce exactly-one execution, which requires coordination.

In the next sections we revisit the tournament application that we studied in Indigo and present two examples of conflicts that can be solved by repairing the invariants.

7.4 A new perspective over the tournament example

The tournament example is a toy micro-service that could be used to support many common competition online games. The operations described here are based on a previous version of the same example, first presented in [5]. We now describe the features of the application.

Players participate in tournaments and compete against each other in matches. A tournament has three phases: an enrollment phase where players can enroll in the tournament, an active phase where there can be no modifications to the participants of the tournament and a finished phase, when the tournament is concluded and a winner is elected, based on the number of points achieved in each match. A tournament cannot be removed after it starts and has a minimum and maximum number of participants. A tournament has a leader that can start or remove the tournament, the leadership role can be shared with other players. A player can deposit and spend credit anytime to buy items that are used in-game to get advantage over the adversary. Items have limited availability.

7.4.1 Example 1: A matter of ordering

While a tournament does not start, players can enroll and disenroll, but the tournament can only start after a minimum number of players have enrolled in the tournament. When a partial ordering of execution is allowed, this constitutes a problem for invariant preservation: a leader of the tournament can start the tournament because he observer, in the local replica, that there is a minimum number of players enrolled, however, concurrently, at a remote replica, a player might disenroll from the tournament, dropping the number of players below minimum. This annomaly is preventend under serial execution, because one of the operations will fail, i.e., either the player cannot disenroll from the tournament, because it starts before, or the tournament cannot start because it does not have enough players. Despite the fact that serialization ensures the applications invariants, programmers need to check that the preconditions of the operations are met before modifying the state of the database.

Under partial ordering execution, the operations must also check the pre-conditions of the operations before taking any action locally, but that does not preclude a concurrent operation from interfering with this one. It might occur that a concurrent remote operation also satisfies its local dependencies but is conflicting with the current operation, and, when both operations are delivered in the same replica, an invariant violation occurs.

Different strategies to repair the invariant violation are possible: we can apply a repair function that makes none of the operations take effect; or the player is not disenrolled from the tournament and the tournament can start, or the player is disenrolled from the tournament and the tournament is canceled. The first solution does not provide a good user experience, because both users will see their actions retracted. The other two repair functions provide a semantic equivalent to the serializable execution, i.e. operations appear to have executed one after the other. However, there is an important caveat with this conflict resolution: more operations might depend on the operation being repaired, for instance, a player might have participated a match after the tournament had started and if we chose to cancel the tournament, that game should have not occurred. In this case it is easy to stop invariant violation from contaminating other operations. We can chose to remove the player from the tournament, in which case no other operation is affected by this convergence policy because no other operation in the workload depends on the player not being enrolled in the tournament to be able to execute.

In general, it might be necessary to analyze conflict resolution strategies in order to prevent the generation of new conflicts. We intend to study static analysis to evaluate the quality of repair strategies.

7.4.2 Example 2: When ordering is not enough

In some situations, invariant violations are not easily repaired. Consider that two players concurrently bought the last unit of an item in the application. For this conflict we cannot apply a repair function that produces a state equivalent to one operation executing after the other, because one of the requests would have different effects, i.e. the operation would fail because there are no available resources left. This situation occurs when operations are not commutative, which means that we cannot arbitrate an ordering for their execution without producing different effects. This is different from the previous example because, in the first case, despite arbitrating the execution ordering of the pair of operations, the effects of both operations appear to have been preserved.

In fact, a serial execution is what makes most sense in the real life, as it would be impossible to duplicate resources. We could think of a service that allows items to be sold in parallel and therefore overselling, but we cannot take more items then physically available.

If this invariant is important for the application, we have no option then to use a strong coordination mechanism to ensure that no user buys more resources then available. However, some invariants, or lets say, application properties, are desirable properties and not essential for correctness, in which case more solutions are possible. To not be unfair with any player, the applications could allow the item to be sold twice which is equivalent to the semantics of eventual consistency, or remove the item from one of the player's inventory and give back some credit. In this case, she might have used the item already and that would create more conflicts. The developer can still make this choice, as long as she is able to repair any operation that used the resource. The last alternative is to create new items to compensate for the advantage that were given to both players.

Our conclusion is that some operations naturally require a coordinated execution, but one can make an alternative version of the same algorithm that does not require serialization and apply a compensation when things go wrong. This is how online stores deal with exhausted stocks, or ATMs handle withdrawals that cannot read the actual balance of an account [8], do in practice.

7.5 How to make the tournament more available

In the previous section we discussed two examples of invariant violations and described possible semantics to correct them. In this section, we are going to describe different algorithms to implement those repairs.

7.5.1 Invariant-aware convergence rules

Consider that we repair the invariant violation of section 7.4.1 by keeping the player in the tournament, when the player is disenrolled concurrently. We pursue a repair strategy that does not impair the availability of the system, and that can be applied in a partitioned databases.

The algorithm we propose is based on the convergence rules used in CRDTs [29]. CRDTs can ensure add-/remove-wins policies when concurrent add and remove operations execute over the same data-type. This means that we can select the outcome of a concurrent add/remove operation of the same element to a set. In the example, the pre-condigions to execute the begin operation is to check that the tournament has the minimum number of players and then the operations changes the value of some flag to true, meaning that the tournament has started. The concurrent disenroll operation removes one element from the participants set, making its size smaller then the minimum and when both operations are propagated to the same replica we end in a state with a set of participants that is smaller than the required for the value of the flag being true.

To solve the violations, consider that the set of participants uses a add-wins strategy for handling conflicting adds and removes. This allows to ensure that that the size of the tournament does not decrease with any concurrent remove, because we can cancel the effect of the remove with the add. In order to enforce this behavior, when starting the tournament, we just add again, to the set of participants, all the players that belong to the set of player in the moment the tournament starts. The merge strategy of the set enforces that any concurrent remove will take no effect. Adding all the elements to the set again can be done in an efficient way, to avoid processing overheads when the tournament is large, we will address this by developing new CRDT data types.

The benefit of this strategy is that it does not require any additional mechanism to detect conflicts, nor apply the repairs, as the execution of the operations automatically addresses any possible conflict. Also, if is compatible with partial replication, since, if the transaction has multiple effects over different objects, the conflict resolution of the data-type automatically handles any conflicting update without further coordination. To implement this strategy we require identifying the pre-conditions of the operations, i.e. what are the conditions for the operation to execute, and instrument the code with the extra updates to enforce those conditions upon operation delivery to a remote replica. Also it is necessary to check that the transformation to the operations are compatible regarding each other, thus not generating an infinite cycle of repairs. We intend to build on the static analysis that we developed in year 1 to implement the new analysis.

7.5.2 Compensating for conflicts

In section 7.4.2, we describe an example where the invariant violation cannot be repaired, thus the system must handle it as part of an exception of the workload. To handle those situations we want to apply some action that compensates for the occurrence. In the previous example, the solutions consist in doing nothing, remove the item from one player's item list, or create new resources. However, two things have to be taken into consideration when writing compensations. First, does the compensation conflict with any other invariant? Second, what happens if two different replicas compensate the same action? To answer the first question we can consider a compensating action as part of the workload and use the same tools to detect the conflicts of the application. Regarding the second question, applying compensating actions is trivial when the operations are idempotent, because they can execute at multiple machines without producing further outputs. The problem with applying the compensating actions is when the operation is non-idempotent, in which case, multiple executions of the same operation produce cumulative effects. For instance, if the compensation was to create new resources, it could create the resource multiple times.

The simplest way to implement the compensation mechanism is to use a central authority that would guarantee that the compensating action only occurred once, or use a consensus algorithm. We are still studying how can we handle these situations with better availability. However, if we can make the compensations idempotent they can be applied multiple times without violating the intention of the programmer, however it is important to understand how does the reordering of compensations affect the intention of the user.

Another property of compensating that we might leverage is that they may not have to be executed immediately, i.e., the system can delegate applying the fix to the future, which can be convenient in some cases.

7.6 Conclusions and future work

We are currently studying invariant-repair as a way to implement Explicit Consistency. The idea of invariant-repair is attractive as no-coordination is the only way to ensure true high-availability. Since causally consistent key-value stores are limited in supporting invariant preservation, it is important to study how can we leverage on the application semantics to push the envelope without requiring extra coordination requirements. The reservations approach was good to provide lower latencies with invariant preservation, but now we want to further improve the availability and fault-tolerance of the system.

The work we are proposing has two main foci: one is the development of analytical tools that can identify and fix invariant violations; the second is to efficiently implement the repairs.

The analytical tools mainly need to detect and suggest conflict resolutions, either by providing transformed conflicting operations or pin-pointing the changes that have to be done. This has to take into account the semantics of the data-types and possible new invariant violations that might arise from the transformations. We intend to extend the current analysis we have in Indigo with a functionality that, for a given set of operations transformations, detects that the database eventually converges and respects the invariants that are defined for the application. n order to accomplish that, we need to take the merge strategies defined per each data-type into account and iteratively test that the proposed modifications do not conflict with other operations, or if they do, there is another repair operation that fixes it and the process terminates.

The other research we need to do is how to efficiently implement the repair operations. As explained in the first example, we may have to reproduce multiple effects to ensure the invariant preservation, which can be very expensive in terms of processing. Also, at this point, it is not clear to to apply compensating action in an highly-available fashion.

We are also taking into consideration how to make the whole approach more automatic, to reduce the effort for the programmer and make the model more attractive. The natural step is to combine the invariant repair approach with the CISE proof system (see Section 6). This can potentially lead to a new and powerful program development methodology.

8 Specification and static verification

8.1 Overview

In last years report we demonstrated different ways of specifying applications, which use eventual consistent data stores. We also showed how these specifications can be related to classical, sequential specifications. These specifications are one requirement for verifying applications. Another requirement is a model of the underlying system on which the application is running. We have created a simplified model of Antidote, which includes the semantics of causal consistency, eventually consistent transactions, and data type semantics. The model is formalized in the interactive theorem prover Isabelle/HOL, which allows us to perform machine checked proofs of application properties. The checkable properties go beyond simple invariants on the database state. We can also proof properties about the history and reason about effects of concurrently executed calls.

We start with a motivating example in Section 8.2. Next we introduce our formalized model in Section 8.3. Finally, Section 8.4 explains how we use the framework to reason about program correctness.

8.2 Userbase example

To show the need to reason about causal consistency and the choice of data types we consider a small application to manage user accounts. This application provides the following API to clients:

- `registerUser(name: String, email: String): Id`
  Creates a new user account with the given data and return the identifier of the new user.
- `removeUser(id: Id)`
  Removes a user from the database.
- `updateMail(id: Id, newMail: String)`
  Updates the mail address of a user.
- `getUser(id: Id): {name: String, email: String}`
  Returns the data of a user or `Undef`, if the user does not exist or was removed.

One difficulty in implementing this example on a weakly consistent data store is to make `removeUser` work correctly. In particular a user that is removed should eventually be removed on all replicas, and not reappear because of other operations being called.

When using CRDTs, eventual consistency and high availability come for free. However, a developer still has to ensure that the semantics of the chosen data types match with the desired application level semantics. For example the following data type choices would lead to problems:

- Last-writer-wins semantics could lead to a situation where `updateMail` wins over `removeUser` because of a later timestamp, even without clock-skew.
- Similarly, with add-wins semantics the effects of `registerUser` or `updateMail` could win over `removeUser`.
- If the update method would not check, whether the user is deleted and just does a blind update, then even with remove-wins semantics a user could reappear after being removed.

```
def registerUser(name, mail) {
  val result = newId()
  atomic {
    users[result]['id'].write(result)
    users[result]['name'].write(name)
    users[result]['mail'].write(mail)
  }
  return result
}

def removeUser(userId) {
  users[userId].delete()
}

def updateMail(userId, newMail) {
  atomic {
    val exists = users[userId].exists()
    if exists {
      users[userId]['mail'].write(newMail)
    }
  }
}

def getUser(userId) {
  atomic {
    val exists = users[userId].exists()
    val name = users[userId]['name'].read()
    val mail = users[userId]['mail'].read()
    return (if exists then {'name': name, 'mail': mail} else Undef)
  }
}
```

Figure 8: Pseudocode implementation of example application to manage user accounts.

- If the identifier for an user was not generated in a way which guarantees uniqueness, then a user could reappear as well.

A possible implementation of this example is given in Figure 8. The variable `users` contains an instance of a map data type and maps user identifiers to another map containing the user data. The inner map contains entries for id, name, and mail which all are last-writer-wins registers. The registers provide `write` and `read` methods. The maps allow to lookup a key (squared brackets syntax) and they allow to `delete` entries and to check whether an entry `exists`.

8.3 A model for causally consistent data stores with CRDTs

Our model includes several aspects. There is a minimal language to model programs written on top of the eventually consistent database. There is the database itself with causally consistent semantics and support for transactions. Finally, there are data types which can be composed to construct application specific types.

```
datatype stmt =
    Do "localState ⇒ localState ⇒ bool"
  | Goto "localState ⇒ nat"
  | CrdtCall string (generateOp:"localState ⇒ any")
  | BeginAtomic
  | EndAtomic

datatype procedure =
  Procedure
    (proc_name:string)
    (proc_args:"string list")
    (proc_body:"stmt list")
```

Figure 9: Language for modeling applications.

8.3.1 The language

The implementation language used in our model is very minimal. Control flow is based on numbered statements and a single `Goto` statement, which calculates the next statement based on the current local state. An abstract `Do` statement can transform the local state.

For interacting with the database layer, there is the `CrdtCall` statement, which generates an operation from the local state and stores the result of the call in a variable. Transactions are supported via the `BeginAtomic` and `EndAtomic` statements.

A program consists of several procedures, and each procedure consists of a name, a list of parameters, and a list of statements as the body.

8.3.2 The database model

Our model does not keep an explicit state, but instead defines the semantics based on a history of all calls to the database. This follows the approach used in related work [10, 18] as well as our previous work on verifying CRDTs [36]. By using a similar approach, we can use the same kind of specifications for CRDTs, which we have previously verified. Therefore we do not have to care about the low level implementation details of CRDTs when verifying application level properties.

All parts of the state are shown in the record definition in Figure 10. Each call to the database is identified by a `callId` and all calls are stored in the finite set `calls`. A `callId` is just an integer, so that we also get a total order on calls that can be used for arbitration, e.g. in last-writer-win-registers. For a call we can get the operation with `callOps`, the returned result via `callres`, and the replica which executed the call via `callReplica`.

A replica always sees a subset of all calls, which is stored in `visibleCalls`. The subset of visible calls represents the operations which already have been synchronized to the replica, so this set is monotonically growing over time. Calls are related by the `happensBefore` relation, which is a partial order. Intuitively $(a, b) \in$ `happensBefore(s )`, when operation b was executed with knowledge of operation a.

For modeling transactions, we store the current transaction for each replica in `CurrentTransaction` and we maintain the equivalence relation `sameTransaction` which relates calls which were executed in the same transaction.

```
datatype config =
    Running (pc_method:string) (pc_line:nat) localState
  | Accept

record state =
  state_calls :: "callId fset"
  state_callOps :: "callId ⇒ any"
  state_callRes :: "callId ⇒ any"
  state_callReplica :: "callId ⇒ replicaId"
  state_visibleCalls :: "replicaId ⇒ callId fset"
  state_happensBefore :: "(callId × callId) set"
  state_sameTransaction :: "(callId × callId) set"
  state_local :: "replicaId ⇒ config"
  state_currentTransaction :: "replicaId ⇀ callId fset"
  state_knownIds :: "int fset"
```

Figure 10: The state

Also each replica has a local configuration which is either `Accept` when the replica is waiting for function calls from clients, or it is `Running` when a method is currently executed.

Finally, the state includes the finite set `knownIds`, which contains all identifiers known to clients. This is necessary, so that we can for example prohibit clients from calling *removeUser*(x) before the user x was even created and the identifier of x returned to the client.

8.3.3 Execution model

We now explain the steps that the system can take. As shown in Figure 11 we distinguish between different kinds of actions: calls from clients, responses to clients, replicas receiving data (pull), internal steps, and failures. The function `step1` in Figure 12 defines the effect of executing one single action.

In case of an `Intern` action, the current statement is determined based on the current procedure and the program counter. The execution of a single statement is handled by the definition `execStmt` shown in Figure 13.

The `Do` statement simply increments the program counter and allows updates of the local state according to the given relation. The use of a relation allows to use this statement for simple assignments, complex local calculations, and nondeterministic behavior.

The `Goto` statement is used to implement control flow in procedures. It simply calculates a new program counter based on the current local state and updates it accordingly.

The more interesting statements are the ones for handling interactions with the database. A `CrdtCall` is handled by the function `makeCall`. It chooses a new identifier `aId` for the call and uses the datatype semantics of the program (`dtsem`) to determine a valid result for the call. Note that the determination is again handled by a relation, so that it is possible for a datatype to be nondeterministic or block until some condition is met. We discuss the modelling of datatype semantics in Section 8.3.4. The call is then stored in the state and added to the visible calls on the current replica. The happens-before relation is updated by adding (x, aId) to the relation, for all x visible at the current replica. If the

```
datatype action =
    Intern (action_replica:replicaId)
  | ClientCall (action_replica:replicaId) (action_method:string) (
   action_args:"any list")
  | ClientResponse (action_replica:replicaId) (action_result:any)
  | Pull  (action_replica:replicaId)
  | Fail  (action_replica:replicaId)
```

Figure 11: The actions

call happens in a transaction, the action is also added to the current transaction.

A transaction is started by the `BeginAtomic` statement. It just sets the current transaction to an empty set of calls. When `EndAtomic` is executed the current transaction is set to `None` again and the equivalence relation `sameTransaction` is updated.

Transactions play an important role in the `Pull` action, which models the exchange of calls between replicas. The set `New` is a set of calls which are visible after pulling. The new set must at least include all previously visible calls and it must be causally consistent according to the happens-before relation. Furthermore, the set cannot contain any calls which are still part of a transaction. When one call of a transaction is in `New`, then also all other calls have to be in `New`. Finally a `Pull` can only happen if the current replica is not currently in a trasaction.

Another important aspect of the model is the `Fail` action, which also involves the handling of transactions. When a replica fails the current transaction is undone by removing all calls in the transaction from the state. The local state and the current transaction are reset to initial values, so that the replica can again accept new calls from clients.

This brings us to the last two actions, namely the handling of calls from clients and responses to clients. If a replica is in the `Accept` state, a client can call any method defined in the program. The number of parameters must be correct and the parameters must only contain UIDs which previously have been exposed to the client via a response. Otherwise no restrictions are made on the possible calls.

A response to the client is possible when control has reached the end of a procedure, i.e. when the program counter is after the last statement. The local variable `result` determines the result returned to the client. If the variable is not set, then `Undef` is returned.

Discussion. The formal model differs from Antidote in some minor points. While Antidote allows one replica to process several requests concurrently, the model is limited to one concurrent request at a time. However, from the view of the application this makes no difference, since the same effect could be achieved with a higher number of replicas. Therefore, if we prove an application correct for an arbitrary number of replicas, then it is also correct for any number of concurrent connections to the database. The difference is mainly in the implementation of Antidote, where concurrency on the same replica has to be handled differently from concurrency on multiple replicas.

The model does also not make the distinction between concurrency between datacenters and within a datacenter, as this is not important for correctness. The main difference in the implementation is the delay between updates and corresponding pulls on other

```
fun step1 :: "program ⇒ action ⇒ state ⇒ state ⇒ bool" where
  "step1 prog (Intern r) s s' = (case state_local s r of
        Accept ⇒ False
      | Running proc pc ls ⇒ (case getStatement prog proc pc  of
           None ⇒ False
         | Some stmt ⇒ execStmt prog r proc pc ls stmt s s'))"
| "step1 prog (Pull r) s s' = (∃New.
    New |⊆| state_calls s
    (* new visible calls at least contain old visible calls *)
    ∧ New |⊆| state_visibleCalls s r
    (* pull causally consistent subset *)
    ∧ (∀x y. y |∈| New ∧ (x,y)∈ state_happensBefore s ⟶ y |∈| New)
    (* do not pull changes which are still not committed *)
    ∧ (∀r trans. state_currentTransaction s r = Some trans
                                            ⟶ trans |∩| New = {||})
    (* pull changes of complete transactions: *)
    ∧ (∀x y. x |∈| New ∧ (x,y)∈state_sameTransaction s ⟶ y |∈| New)
    (* do not allow pull inside a transaction *)
    ∧ state_currentTransaction s r = None
    ∧ s' = s⦇
      state_visibleCalls := (state_visibleCalls s)(r := New)
    ⦈)"
| "step1 prog (Fail r) s s' = (
    let transaction = (case state_currentTransaction s r of
                          None ⇒ {||}
                        | Some tr ⇒ tr)
    in s' = (s⦇
        (* undo transaction *)
        state_calls := state_calls s |-| transaction,
        (* reset transactionstate *)
        state_currentTransaction :=
            (state_currentTransaction s)(r := None),
        (* reset local state *)
        state_local := (state_local s)(r := Accept),
        state_happensBefore := {(x,y)∈state_happensBefore s.
                  x|∉|transaction ∧ y|∉|transaction},
        state_visibleCalls :=
            (state_visibleCalls s)
                (r := state_visibleCalls s r - transaction)
      ⦈))"
| "step1 prog (ClientCall r proc args) s s' = (∃method.
      state_local s r = Accept
    ∧ Some method = lookup_method prog proc
    ∧ (∀arg∈set args. includedUids arg |⊆| state_knownIds s)
    ∧ length (proc_args method) = length args
    ∧ (let ls = map_of (zip (proc_args method) args)
       in s' = s⦇ state_local := (state_local s)(r:=Running proc 0 ls)
    ⦈))"
| "step1 prog (ClientResponse r res) s s' = (∃proc method pc ls.
      state_local s r = Running proc pc ls
    ∧ Some method = lookup_method prog proc
    ∧ pc ≥ length (proc_body method)
    ∧ res = (case ls ''result'' of Some res ⇒ res | None ⇒ Undef)
    ∧ s' = s⦇ state_local := (state_local s)(r := Accept),
        state_knownIds := state_knownIds s |∪| includedUids res⦈)"
```

Figure 12: Single step in the execution

```
fun configUpdate :: "replicaId ⇒ (localState ⇒ localState ⇒ bool) ⇒
     state ⇒ state ⇒ bool" where
"configUpdate r rel s s' = (∃ls'. case (state_local s r) of
    Running proc pc ls ⇒ rel ls ls'  ∧  s' = (s⦇state_local :=
            (state_local s)(r := Running proc (pc+1) ls')⦈)
  | _ ⇒ False)"

fun makeCall :: "dataTypeSem ⇒ replicaId ⇒ string ⇒ any ⇒ state ⇒
   state ⇒ bool" where
"makeCall dtsem r v args s s' = (∃aId res cfg'.
    aId |∉| state_calls s
∧ dtsem s r args res
∧ cfg' = (case state_local s r of
    Running proc pc ls ⇒ Running proc (pc+1) (ls(v↦res)))
∧ s' = s⦇
    state_calls := state_calls s |∪| {| aId |},
    state_callOps := (state_callOps s) (aId := args),
    state_callRes := (state_callRes s) (aId := res),
    state_visibleCalls := (state_visibleCalls s)
        (r := state_visibleCalls s r |∪| {| aId |}),
    state_happensBefore := state_happensBefore s
        ∪ ({e. e |∈| state_visibleCalls s r} ×{aId}),
    state_local := (state_local s)(r := cfg'),
    state_callReplica := (state_callReplica s) (aId := r),
    state_currentTransaction := (case state_currentTransaction s r of
        Some transaction ⇒ (state_currentTransaction s)
                        (r := Some (transaction |∪| {| aId |}))
        | None ⇒ state_currentTransaction s)
    ⦈)"

definition execStmt :: "program ⇒ replicaId ⇒ string ⇒ nat ⇒
   localState ⇒ stmt ⇒ state ⇒ state ⇒ bool" where
"execStmt prog r proc pc ls stmt s s' ≡ (case stmt of
  Do rel ⇒ configUpdate r rel s s'
| Goto line ⇒ s' = s⦇
        state_local := (state_local s)(r := Running proc (line ls) ls)
        ⦈
| CrdtCall v args ⇒ makeCall (dataTypeSem prog) r v (args ls) s s'
| BeginAtomic ⇒ s' = s⦇
        state_local := (state_local s)(r := Running proc (pc+1) ls),
        state_currentTransaction :=
            (state_currentTransaction s)(r := Some {||})
        ⦈
| EndAtomic ⇒ (∃tr. state_currentTransaction s r = Some tr
        ∧ s' = s⦇
        state_local :=
            (state_local s)(r := Running proc (pc+1) ls),
        state_currentTransaction :=
            (state_currentTransaction s)(r := None),
        state_sameTransaction :=
            state_sameTransaction s ∪ (fset tr × fset tr)
        ⦈))"
```

Figure 13: Executing a statement of the program

replicas.

Another difference is that the model fixes the database schema and the application code upfront, so no upgrades at runtime are possible. This aspect would go beyond the current scope of the model.

There are some restrictions on the model with regards to what kind of properties can be explored with the model. As the model uses finite traces for executions, and the model does not include any guarantees about fairness, it is also not possible to reason about liveness and fairness on the application level. Likewise statements about performance are also not possible, except for simple metrics like number of calls to the database.

It would also have been possible to make the model more flexible with respect to consistency levels. The causal consistency model is fixed in the system model. However, it is still possible to model stronger consistency by using the data type semantics. The following section describes how data types are integrated into the model.

8.3.4 Modeling data types

One of the most important parts of the model is the integration of data types. In general the semantic of a data type is simply given by a relation of type *state* ⇒ *replicaId* ⇒ *any* ⇒ *any* ⇒ *bool*. Based on the current system state, the current replica and the given operation of type *any*, the relation states whether a given result of type *any* is valid.

This approach is quite flexible and allows specifying data types which are not CRDTs. For example it is possible to specify a register which provides sequential consistency. There are still some restrictions, for example a linearizable register cannot be specified, because the model does not keep track of absolute times.

While we allow more powerful data types in general, applications we are interested in mainly use CRDTs. A CRDT specification can be seen as a function of type *updateHistory* ⇒ *any* ⇒ *any*, which takes the update history and an operation of type *any* and deterministically returns a result of type *any*. The *updateHistory* is a restricted view on the complete state. It only contains the calls visible on the current replica and does not contain irrelevant information, like local state, transaction data, or the identifiers known to clients. The restriction of the complete state to the *updateHistory* is done by the function `stateToUpdateHistory` as shown in Figure 14. The fact that CRDTs are deterministic from an internal view is captured by the fact that a *crdtSpecification* is a function of the update history and the arguments.

As our model handles the whole database as one CRDT, we need some way to compose several CRDTs into one. The most important primitive for composition is probably the map. The simplest form of map does not have a remove operation for elements. It simply forwards operations to embedded CRDTs depending on the given key. To handle heterogeneous maps, we parameterize the map CRDT with a function, which selects a *crdtSpecification* based on the key. For homogeneous maps this function is constant as there is only one type of value. The semantics of this simple map can be defined as follows:

```
fun map_semantics :: "(any ⇒ crdtSpecification) ⇒ crdtSpecification"
   where
"map_semantics keyTypes H operation = (case operation of
      :[key, operation] ⇒ (keyTypes key) (extractOperationsForKey key
     H) operation
   |  _ ⇒ Undef)"
```

```
record updateHistory =
  updateHistory_calls :: "callId fset"
  updateHistory_callOps :: "callId ⇒ any"
  updateHistory_callRes :: "callId ⇒ any"
  updateHistory_callReplica :: "callId ⇒ replicaId"
  updateHistory_happensBefore :: "(callId × callId) set"

fun stateToUpdateHistory :: "state ⇒ replicaId ⇒ updateHistory" where
"stateToUpdateHistory s r = ⦇
  updateHistory_calls = state_visibleCalls s r,
  updateHistory_callOps = restrictOn (state_visibleCalls s r) Undef (
    state_callOps s),
  updateHistory_callRes  = restrictOn (state_visibleCalls s r) Undef (
    state_callRes s),
  updateHistory_callReplica  = restrictOn (state_visibleCalls s r) 0 (
    state_callReplica s),
  updateHistory_happensBefore = {(x,y)∈state_happensBefore s. {|x,y|}
    |⊆| state_visibleCalls s r}
⦈"

(* converts a crdt specification to a general data type semantics *)
fun crdtSpecification2dataTypeSem :: "crdtSpecification ⇒ dataTypeSem
    " where
"crdtSpecification2dataTypeSem spec state replica args res = (spec (
    stateToUpdateHistory state replica) args = res)"
```

Figure 14: Crdt semantics

When the operation is a tuple with a *key* and an *operation*, then the appropiate embedded CRDT semantics is chosen based on the key. Then this CRDT semantics is used with the given operation (not including the key) and an update history including only the events relevant to the given key.

A more practical version of a map also provides an operation to delete an entry. We model this with another CRDT semantics, a deletable CRDT which wraps one arbitrary CRDT and adds a *delete* and *exists* operation. The following specification describes the delete-wins variation, where only the updates that come after all deletes are effective.

```
definition deletable_crdt :: "crdtSpecification ⇒ crdtSpecification"
    where
"deletable_crdt orig_crdt H operation = (case operation of
     :delete ⇒ :ok
  | :exists ⇒
        Bool (∃aid. aid|∈|updateHistory_calls H
                ∧ updateHistory_callOps H aid ≠ :delete
                ∧ updateHistory_callOps H aid ≠ :exists
                ∧ (∀d. d|∈|updateHistory_calls H
                     ∧ updateHistory_callOps H d = :delete
                        → (d,aid) ∈ updateHistory_happensBefore H))
  | other ⇒ let H' =  H⦇
        updateHistory_calls := ffilter
            (λaid. updateHistory_callOps H aid ≠ :delete
                 ∧ updateHistory_callOps H aid ≠ :exists
                 ∧ (∀d. d|∈|updateHistory_calls H
                      ∧ updateHistory_callOps H d = :delete
```

```
definition userCrdt :: crdtSpecification where
"userCrdt = map_semantics (λkey. case key of
    Atom ''id'' ⇒ lwwreg_semantics Undef
  | Atom ''name'' => lwwreg_semantics Undef
  | Atom ''mail'' => lwwreg_semantics Undef
  | _ ⇒ error_crdt key)"

definition exampleProgramCrdt :: dataTypeSem  where
"exampleProgramCrdt state r operation result = (
  if operation = :newUID then
    (* generate new unique id *)
    result ∉ {res. ∃aId. aId |∈| state_calls state ∧ state_callOps
   state aId = :newUID ∧ state_callRes state aId = res}
  else
     (crdtSpecification2dataTypeSem (map_semantics (λkey. case key of
       Atom ''users'' ⇒ map_semantics (λk. deletable_crdt userCrdt)
       | _ ⇒ error_crdt key
     )) state r operation result))"
```

Figure 15: Crdt semantics of the example program

```
                         → (d,aid) ∈ updateHistory_happensBefore H))
            (updateHistory_calls H)
  ⦈ in orig_crdt H' operation)"
```

The above setup allows us to easily specify custom CRDTs and compose existing specifications. However, this does not imply that implementations can be composed in the same way. The current state of the art allows only some CRDT implementations to be composed efficiently.

8.4 Reasoning about program correctness

Based on the model described in the previous section, we can now use interactive proofs in Isabelle to prove properties about programs. To show this we again consider our initial example of a user database (see Figure 8). A simple property which this application should satisfy is that deleted users do no reappear. In the following definition we formalize this property from the perspective of a client at a single replica:

```
definition "removeIsPermanent prog ≡ (∀s trace t1_call  t1_res t2_call
    t2_res r userId res1 res2.
    steps prog trace initialState s
  ∧ t1_res < t2_call
  ∧ CallResponse r (CallEvent t1_call ''removeUser'' [userId])
                   (ResponseEvent t1_res res1)
             ∈ set (callResponsePairs trace 0)
  ∧ CallResponse r (CallEvent t2_call ''getUser'' [userId])
                   (ResponseEvent t2_res res2)
             ∈ set (callResponsePairs trace 0)
  → res2 = Undef)"
```

The basis for the specification are the traces, which describe the executions from the view of a client. The property has to hold for every *trace*, such that executing the program from the *initialState* leads to a state *s*. In the trace we look at two call-response pairs. One is a call to *removeUser(userId)* and another is *getUser(userId)* with result *res2*. We assume that the call to *getUser* happens after the call to *removeUser*, formalized by comparing the start- and end-times (*t1_res* < *t2_call*). If we have a situation like this, we expect the result of the *getUser* call *res2* to always be *Undef*.

This property not only depends on the program text in Figure 8, but also on the chosen CRDTs. Figure 15 shows the CRDT semantics we use for the example. A user record is modeled by a map pointing to last-writer-wins registers with start value *Undef*. The data type for the whole program is not a CRDT, because it can generate globally unique identifiers with the operation *newUID*. Other operations are forwarded to a map with only one key named *users*. This map models global variables and could be extended with other keys. Behind the key we have another map, which maps user identifiers to user records. We wrap the *userCrdt* in a *deletable_crdt*, to get a *delete* operation with *delete*-wins semantics.

Using these semantics for the data type, we can argue (informally) why the program behaves as intended and satisfies our property:

(1) When *removeUser(id)* has been called, then an operation *users[id].delete()* must be in the database history for some point in time between the call and response.

(2) Moreover, there are no map update operations on a removed user, which happen causally after the remove (except for other removes).

  (a) The operations in the *registerUser* procedure cannot come afterwards, because *newUID* never returns an identifier known to clients. Therefore *removeUser(id)* must happen at a point in time after *registerUser* and no happens-before relation can exist which points into the past.

  (b) The operations in *updateMail* cannot happen after a remove, because the procedure checks whether the user exists before doing any updates. Because the code is packed in an atomic unit, the check and the map updates see the same set of operations. So if the update operations were executed after a remove, the existence check would have returned false.

(3) When *getUser* is called on the same replica sometime after *remove*, we get that there is a database operation for deleting the user by property (1). We also know, that this database operation happened before the operations in *getUser*, as this is guaranteed by the session order guarantees of the database. By property (2) we know that no database operation on the same user happens after the remove. There can be concurrent updates, but since we used a remove-wins semantics for the map, we always get the required result, that the user does not exist.

The first two properties are invariants, while the last property is a consequence of the invariant and the data type semantics. Properties like the first one above have to be used for most proofs, since we often want to express properties on the application level in terms of traces, but have to reason about the data type semantics and histories on the database level. To capture this, formalizations of our invariants maintain a mapping between the database operations and the corresponding *Intern* steps in the traces. Based on this it is also possible to link a call-response pair with the set of database operations invoked during the call.

In summary, we have to find an invariant *Inv(trace, state, mapping)*, which relates the trace, the current state (which includes the history of database operations), and the mapping between trace and database history. Then first we have to show that the invariant holds for the initial state with the empty trace and the empty mapping. Second we have to show that each step (see Figure 12) maintains the invariant. Currently, this means that the invariant also has to include information about local states. Therefore this process can be quite tedious.

8.5 Future work

We hope to reduce the amount of manual work required for performing proofs in the future by capturing more general properties in proof rules and by using more automation. In particular we believe that it will be possible to handle atomic blocks as one single step, which would significantly reduce the manual effort. The primitive proof rule we currently use already helps with informal reasoning about program correctness. We hope that developing more specialized proof rules will also lead to more insights for informal reasoning and thus helps developers in writing correct applications.

For programs using locks, we hope to integrate the proof rules from CISE (see Section 6) into our framework, so that these cases become easier to handle. This would combine the flexibility and expressiveness of our model with the easier and partially automatic methods of CISE. We also hope to integrate our approach of reasoning with runtime verification techniques (see Section 8) so that developers can test invariants before starting an expensive formal proof. The systematic exploration techniques would already give a high confidence in the correctness of an invariant, which could be completed by formal verification.

9 Run-time (dynamic) verification

9.1 Overview

One of the verification and validation activities within the SyncFree project is the development of a runtime verification tool that supports programs written to run on the Antidote framework. CRDT-based programming frameworks including Antidote allow programmers to write highly available applications that avoid synchronization operations. Weak consistency in general and CRDTs in particular are novel programming settings and ensuring the correctness of programs in these settings is not straightforward and may be a challenge for application programmers. By building a runtime verification tool for Antidote programs, we aim to help programmers with this challenge. By going beyond random testing, runtime verification and systematic exploration of Antidote program behaviors, especially different orderings of inter-replica message delivery with respect to replica-local operations, we provide improved early stage bug detection and reproduction ability to programmers. In this section, we present the specifics of this activity and contrast it with existing runtime verification work in the literature.

9.2 Antidote operation and specifying Antidote programs

As has been explored in work packages 1, 2 and 4 in the SyncFree project thus far, programmers naturally express desired program properties in the form of replica-local assertions expressed as first-order logic formulas in terms of replica state, and as invariants on the global state, which are also written as assertions expressed as first-order formulas in terms of the final, converged state of a (and, in fact, all) replicas. Since Antidote guarantees strong eventual consistency and causality, this rather straightforward approach to specifying application properties works well.

The checking of an assertion in terms of replica-local state variables is implemented in a straighforward manner in Erlang by evaluating the boolean predicate in the assertion. Replica-local assertions are written simply as program statements in the transactional programs running at each replica in the Antidote program. Global invariants are also Erlang statements, but their evaluation needs to wait for all messages to be delivered to the replica on which the invariant will be evaluated, and for this replica to apply the effects of all remote transactions to its state. The function computing the global invariant is thus called only after the runtime verification tool sends a message to the replica indicating that all remote transactions have been delivered to it, and after the replica has processed all the remote transactions in its queues.

9.2.1 Systematic exploration of behaviors

Operational model for Antidote. We are concerned about only four different Erlang processes running potentially concurrently on each Antidote node. These processes execute and log replica-local transactions, and receive, process, and log the remote transactions. For the simplicity, we call those processes respectively *runner*, *receiver*, *updater*, and *logger* in the rest of this document. Process *runner* is responsible for executing the transactions issued by a client attached to a replica. Runner processes also call the *logger* processes to write transactions to the log. Other processes are responsible for

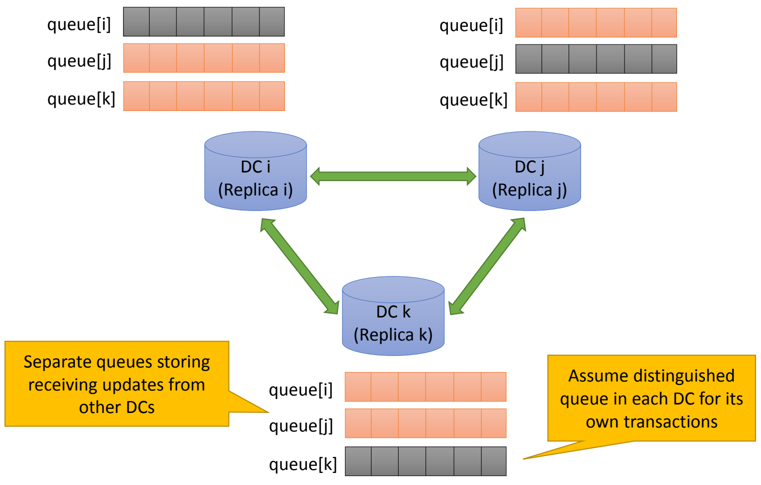


Figure 16: Operational model for Antidote.

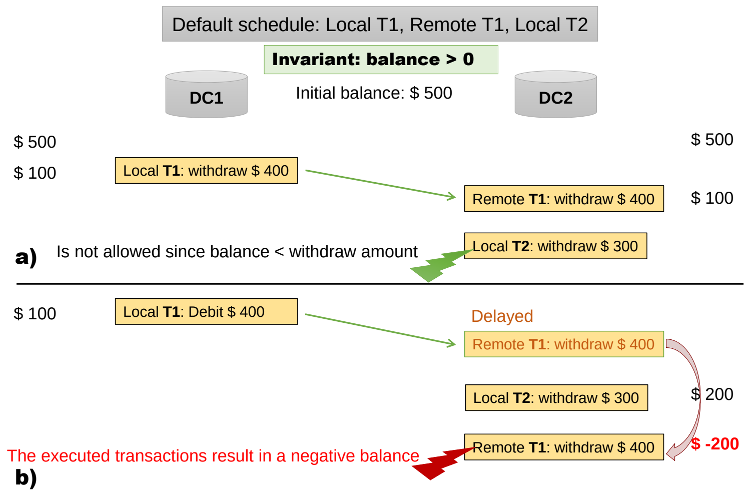

Default schedule: Local T1, Remote T1, Local T2
Invariant: balance > 0
DC1
Initial balance: $ 500
DC2
$ 500
$ 100
Local T1: withdraw $ 400
$ 500
Remote T1: withdraw $ 400
$ 100
a) Is not allowed since balance < withdraw amount
Local T2: withdraw $ 300
$ 100
Local T1: Debit $ 400
Delayed
Remote T1: withdraw $ 400
Local T2: withdraw $ 300
$ 200
Remote T1: withdraw $ 400
$ -200
The executed transactions result in a negative balance
b)

Figure 17: a) An ordering with no violation; b) Applying one delay in the ordering in (a) that violates the invariant.

processing remote transactions. The receiver process receives the incoming transactions from other replicas, and sends them to the updater process. Then, this process checks the dependencies of received transactions one by one, and calls the logger process to log the transaction, if its causal dependencies are satisfied. The timing of writing to the log is non-deterministic. Intuitively speaking, programs on weakly consistent platforms, including Antidote programs, are especially prone to property violations that arise from a replica having a possibly stale view of the state of certain CRDTs and issuing operations and transactions based on this stale view. This may lead to the execution and later communication to other replicas of transactions that may cause invariant violations. Figure 17 illustrates one such simple case. As seen in this example, exploring different timings of updates corresponding to transactions in remote replicas relative to the execution of the transactions being issued at a particular replica will uncover such bugs. In the rest of this section, we will refer to an atomic step a CRDT-based program (such as an Antidote) program as an *event* and we will describe an execution of such a program as a linearly-ordered sequence of events. Examples of events are the execution of a transaction at a replica, the delivery or receipt of a transaction to/from another replica, and updating the replica state by incorporating into it the effect of a remote transaction. Since an Antidote replica has choice over whether it processes a local transaction or applies the effects of a remote one, and since there is timing non-determinism in when messages are delivered through a network, we refer to an "interleaving" of events and speak of "schedules". The systematic exploration of all interleavings is thus the exploration of possible schedules (interleavings) and is accomplished in our runtime verification tool by an algorithm that explores a set of chosen schedules and enforces the events in the system to take place according to the chosen schedule.

However, to detect such concurrency bugs, exhaustive exploration of all possible interleaving of events is computationally too expensive. Runtime verification tools must strike a compromise between coverage of behaviors and computational cost. One promising approach for a tunable compromise in the runtime verification literature is prioritized exploration of program behaviors and has been applied successfully to shared-memory concurrent programs. These techniques have derived inspiration from the observation that many concurrency bugs can be triggered and explained by making reference to the ordering of a small number of noteworthy events. Delay-bounded scheduling 18 is one way of performing prioritized, systematic exploration of event orderings within a "neighborhood" of a given ordering, called the default ordering. Explored schedules for a delay bound of K are allowed to deviate from the original, default schedule by delaying (postponing to the end of the schedule) K tasks. Delay-bounded scheduling has been shown to be effective in bug detection in concurrent shared-memory programs and has been successfully adapted to asynchronous programs. In our runtime verification work in the SyncFree project, we explore the generalization of delay-bounded scheduling to CRDT-based, thus geo-replicated distributed programs. The next section elaborates on this investigation.

9.2.2 Delay-bounded scheduling

The delay bounding approach [14] explores the program behaviors arising in executions with a given scheduler $S(K)$ parameterized by a "delay bound" $K \in N$. $S(0)$ is the default chosen schedule. $S(K)$ is a set of schedules, where the set is characterized by the number of additional nondeterministic choices K. Ideally, for large enough K, $S(K)$ is the entire set of program interleavings for a given, fixed program. A delay-bounded scheduler will explore all $S(K)$ in increasing order of K. The approach has been shown to work well (e.g. Emmi et al. [14]) for shared-memory concurrent programs, where an efficiently-implementable "depth-first" delaying scheduler DF(K) was shown empirically to expose behaviors with few ordering dependencies using small values of K.

The main idea in delay-bounding approach is making a deterministic scheduler sufficiently nondeterministic by delaying its next scheduled task. In other words, search space in delay-bounding is parameterized by a deterministic scheduler S, and a delay bound K. Delay-bounding is independent of the deterministic scheduler, and can be applied to any scheduler such as round-robin, depth-first, and any other deterministic scheduler. So, in the execution of a deterministic schedule, at the point a task is dispatched can be delayed to execute later. An execution path is said to be K-delay-bounded if it contains at most K delay operations in the path.

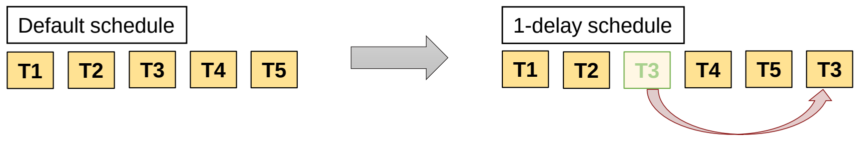


Figure 18: One-delay execution of a default schedule.

9.2.3 Generalizing delay bounding to CRDT-based programs

Delay-bounded scheduling [14] considers no dependency between the tasks. Figure 18 shows an one-delay schedule. Because of the dependency between operations in CRDT-based programs, we adjust delay-bounded scheduling in this context, by considering the causal dependency between operations. So, a transaction cannot be scheduled after the transactions which are causally dependent on it. In other words, delaying a transaction results in delaying all its causally dependent transactions, but this is counted as one delay. In addition delaying local transactions after their corresponding remote transactions is not allowed. Likewise, delaying a local transaction results in delaying its propagated version, and totally is counted as one delay.

9.3 Tool design and implementation

9.3.1 Tool architecture

Record/replay systems are used for detecting and diagnosing concurrency errors. Since concurrent and distributed systems have scheduling and timing non-determinism, it is only possible to reproduce a bug if all the non-deterministic choices that occurred in the buggy execution have been captured and can be repeated. Thus, record and replay capability is crucial in a runtime verification tool for concurrent and distributed systems, including CRDT-based ones.

We present a record/replay system, Commander, which targets the Geo-replicated applications, and reproduces their executions based on the interleaving non-determinism. The executions are reproduced and a set of executions around the recorded one are explored using delay-bounded scheduling [14, 15].

Commander is composed of three components: (1) recorder; (2) scheduler; (3) replayer. Figure 19 illustrates the architecture of the Commander tool. First, a default scheduling of the riak_test test is recorded using the recorder component. Scheduler then loads the recorded trace and schedules a set of executions around the recorded trace, that each is replayed by the replayer component. When a reproduced execution is replayed, it is also recorded by the recorder. The recently recorded trace will be loaded by the scheduler later, when the previous set of the scheduled executions is replayed. Scheduling the lately recorded trace can result in distinguished executions. So, in each pass, more executions are replayed. As an optimization, we will extend the scheduler later to check if a scheduled execution has been replayed before. The following paragraphs give more explanation of each component.

Recorder The recorder component records the default scheduling in the initial run. It fills the trace which is sent to the scheduler component later as an input. To record

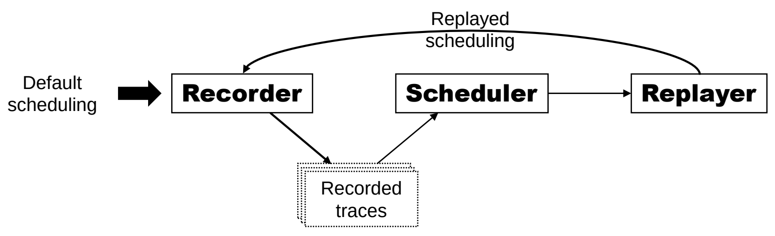


Figure 19: Architecture of the Commander tool.

the default schedule in the initial run, Commander allows the requested DC to proceed in the receiving order. Then before allowing any other DC to proceed, it waits for an acknowledge, and required data from the DC to fill, and record a new trace.

Scheduler This component is in charge of reproducing an interleaving, using delay-bounded technique, of the recorded trace to be explored. So, in each pass, the scheduler loads the recorded trace, and generates an ordering of the events, which feeds the replayer component.

Replayer The replayer is the component which plays back the scheduled interleaving. It gets the scheduled trace from the Scheduler and allows only one replica to operate at a time. Replayer also calls the recorder component to record the replayed scheduling. To replay a scheduled trace, replayer reads events of the trace one-by-one, and it allows the corresponding DC to proceed only if it is enabled. A DC is enabled, if it has already requested for the scheduled event. Finally, Replayer waits for an acknowledgment, and the required data from the DC, to update its state, and call the Recorder to fill, and record a new trace using the received data.

Therefore, to enforce a schedule in CRDT-based programs, Commander allows only one replica to operate at a time. Figure 20 depicts how the Commander orchestrates the two Antidote nodes.

9.3.2 Implementation

The Commander implements the gen_server behaviour, and its state is a record consisting of the following fields: (1) phase, denotes the phase in which the Commander is running; (2) device, the file that is used either in record or replay phase; (3) events, a sub-sequence of the scheduled trace, that has not yet been replayed; (4) waiting_requests, a set of all received request that has not yet been responded; (5) current_event, the current event which is replaying; (6) tx_mapping, a mapping from transactions' previous identifiers to their new identifiers which are generated in the replay phase. The events have been categorized as local or remote events. Local events are transactions issued on a DC by the client, whereas the remote transactions are those that has been received from other DCs, and are applied on the received DC. All requests are processed sequentially in order to have only one replica active at a time.

There is a separate module for the Recorder, Scheduler, and Replayer components, that contains the specific functions for each of them. Regarding the phase in which Commander is running, it calls the required functions of the appropriate module.

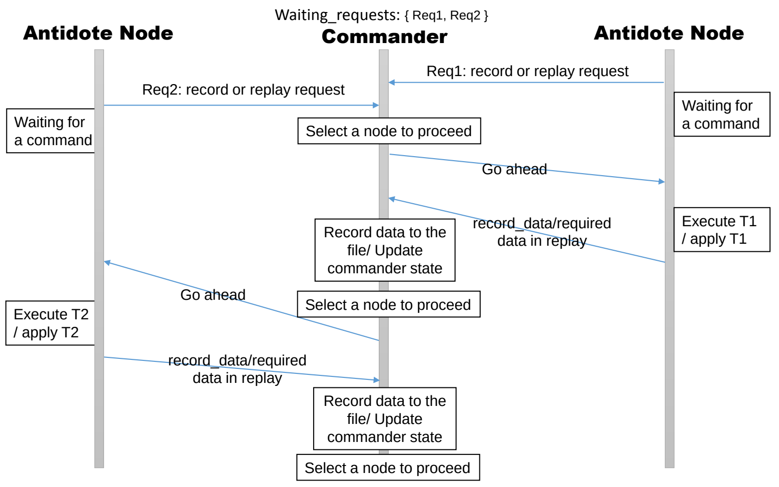
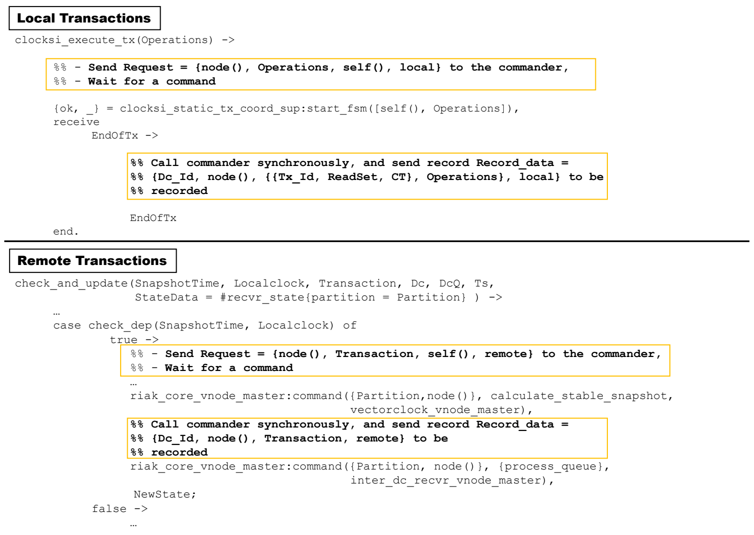

Waiting_requests: { Req1, Req2 }
Antidote Node
Commander
Antidote Node
Req1: record or replay request
Req2: record or replay request
Waiting for a command
Select a node to proceed
Waiting for a command
Go ahead
record_data/required data in replay
Record data to the file/ Update commander state
Execute T1 / apply T1
Go ahead
Select a node to proceed
Execute T2 / apply T2
record_data/required data in replay
Record data to the file/ Update commander state
Select a node to proceed

Figure 20: Commander enforces only one replica to operate a time.

We integrate the Commander in Antidote by instrumenting the Antidote source code. So, the instrumentation points are where a local transaction is about to execute, or a remote transaction is written to the log. Figure 21 depicts two example of where to instrument in the Antidote code. To delay a local transaction, the process in charge of executing the transaction need to be killed, and the transaction will be start later when it is scheduled. Delaying the remote transactions are straightforward, that the remote transaction is popped and then re-inserted in the same queue.

9.4 Related work

The related work in the literature does not address our concern which is exploring different possible behaviors of a CRDT-program. The most commonly used tool, QuickCheck [3] is a property-based testing tool with support for model-based exploration as well. It is based on data non-determinism, meaning that it generates the test cases for random inputs, whereas we address the interleaving nondeterminism. Although PULSE [12] addresses the interleaving nondeterminism, to the best of our knowledge, it only detects race conditions, and supports Erlang-level concurrency. The state-less model checker, Concuerror [11], is also based on interleaving nondeterminism, that detects deadlock, assertion violation, and abnormal termination. It explores possible executions systematically, and uses the partial order reduction techniques to reduce the state space size. But it only supports the concurrency between processes on a single Erlang node. The McErlang [16] model checker likewise the other model checking tools requires the desired property to be specified. It replaces the concurrency and message passing in Erlang runtime system. McErlang accepts an Erlang program as an input, and the properties can be specified in both Erlang language using a callback module, or linear temporal logic. Etomcrl [4] is a tool which translates the Erlang code to mcrl, a process algebraic language, which then is checked by CADP toolset. Verification of the applications using Etomcrl check a model of the program, that can be divergent from the program itself. Another approach is model checking the abstract models of the CRDT-programs written



Local Transactions

```
clocksi_execute_tx(Operations) ->

    %% - Send Request = {node(), Operations, self(), local} to the commander,
    %% - Wait for a command

    {ok, _} = clocksi_static_tx_coord_sup:start_fsm([self(), Operations]),
    receive
        EndOfTx ->

            %% Call commander synchronously, and send record Record_data =
            %% {Dc_Id, node(), {{Tx_Id, ReadSet, CT}, Operations}, local} to be
            %% recorded

            EndOfTx
    end.
```

Remote Transactions

```
check_and_update(SnapshotTime, Localclock, Transaction, Dc, DcQ, Ts,
                 StateData = #recvr_state{partition = Partition} ) ->
    …
    case check_dep(SnapshotTime, Localclock) of
           true ->
             %% - Send Request = {node(), Transaction, self(), remote} to the commander,
             %% - Wait for a command
             …
             riak_core_vnode_master:command({Partition,node()}, calculate_stable_snapshot,
                                            vectorclock_vnode_master),
             %% Call commander synchronously, and send record Record_data =
             %% {Dc_Id, node(), Transaction, remote} to be
             %% recorded
             riak_core_vnode_master:command({Partition, node()}, {process_queue},
                                            inter_dc_recvr_vnode_master),
             NewState;
        false ->
             …
```

Figure 21: Commander enforces only one replica to operate a time.

in TLA+ using TLC, that already has been done by our group, and presented at deliverable D1.2. This approach is not practical for programmers developing Antidote programs due to the divergence between the model and the program, and the need to write separate TLA+ model of the program.

Our proposed tool reproduces the set of executions systematically using delay-bounding technique. It supports features specific to CRDT-programs such as dependency between different operations, whereas other related work in the literature takes only the Erlang into consideration, thus could result in more false positives. It supports distributed Erlang, and does not require the program to be modeled. Invariants are checked using eunit assertions, and no other knowledge than Erlang is required.

9.5 Conclusions and future work

Currently, we have implemented the recorder, and the scheduler for the default execution trace. So, the initial run of a riak_test test is recorded in the default ordering. Then, scheduler returns the 0-delay-bound schedule. There was a few issues in implementing the replayer component, that we are re-factoring its implementation to solve them.

After that, we will instrument the new version of the Antidote with pub-sub inter-dc-replication layer to enable the communication between it and the Commander. Then, extend the scheduler to reproduce a set of K-delay-bounded schedules. At this point, the implementation of the replayer will not need any modification. But, later we will refactor the replayer implementation to add the capability of recording any replayed execution.

So, our future direction is extending the Commander to:

1. Instrument the new version of the Antidote, to work with the Commander. In the new version, there is a slight change in transaction execution interface, and pretty much change in the inter-dc-replication layer. Although the protocol has not been changed, because of the modifications in the code, we need to instrument some different places in the Antidote source code to have the Antidote nodes been orchestrated by the Commander.

2. We will investigate how efficient using QuickCheck, and PULSE in replay phase could be.
3. Do the record and replay iteratively. We will extend the current replayer implementation to record any replayed execution as well as the initial default execution.
4. Distribute the Commander. Currently, Commander runs on a single DC, separate from other DCs. So, if it crashes, the system will fail. Finally, to prevent single point of failure, we will distribute the Commander through all DCs, that will make it more responsive as well.

To develop a more efficient runtime verification tool for the Antidote-based applications, a knowledge of the Antidote inner working is essential, since we instrument its source code, and also take into account the causal dependency between operations captured by the Antidote, in reproducing the executions around one recorded execution. We believe that having more collaboration with Antidote team, will speedup the progress of the work. So, one member of our group will be visiting the Antidote team during the first half of the last year of this project.

10 Papers and publications

- Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Mahsa Najafzadeh, Marc Shapiro, and Nuno Preguiça. *Towards Fast Invariant Preservation in Geo-Replicated Systems*. ACM SIGOPS Operating Systems Review, 49:1(5), 2015.
- Christopher Meiklejohn and Peter Van Roy. *Lasp: A Language for Distributed Coordination-free Programming*. 17th ACM International Symposium on Principles and Practice of Declarative Programming (PPDP 2015), Siena, Italy, July 14-16, 2015.
- Christopher Meiklejohn and Peter Van Roy. *The Implementation and Use of a Generic Dataflow Behaviour in Erlang*. 14th ACM SIGPLAN Erlang Workshop, Vancouver, BC, Sep. 4, 2015.
- Christopher Meiklejohn and Peter Van Roy. *Selective Hearing: An Approach to Distributed Eventually Consistent Edge Computation*. Workshop on Planetary-Scale Distributed Systems (W-PSDS 2015, colocated with SRDS 2015), Montréal, Quebec, Sep. 28, 2015.
- Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. *Putting Consistency back into Eventual Consistency*. ACM SIGPLAN European Conference on Computer Systems (EuroSys 2015), Bordeaux, France, April 21-24, 2015.
- Christopher Meiklejohn and Peter Van Roy. *Lasp: A Language for Distributed Eventually Consistent Computations with CRDTs*. Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC 2015, colocated with EuroSys 2015), Bordeaux, France, April 21, 2015.
- David Navalho, Sérgio Duarte, and Nuno Preguiça. *A Study of CRDTs that do Computations*. Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC 2015, colocated with EuroSys 2015), Bordeaux, France, April 21, 2015.
- Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Marc Shapiro, and Mahsa Najafzadeh. *The Case for Fast and Invariant-Preserving Geo-Replication*. Workshop on Planetary-Scale Distributed Systems (W-PSDS 2014, colocated with SRDS 2014), Nara, Japan, Oct. 6, 2014.

Submission

We list this paper, even though it is still in submission, because it presents the CISE formal proof system that is relevant to the Consistency work presented in Section 6.

- Alexey Gotsman, Hongseok Yang, Carla Ferreria, Mahsa Najafzadeh, and Marc Shapiro. *'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems*. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016), submitted for publication.

References

[1] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein. Consistency without borders. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 23. ACM, 2013.

[2] Amazon. Supported operations in DynamoDB. http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/APISummary.html, 2015.

[3] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang (ERLANG '06)*, pages 2–10, New York, NY, USA, 2006. ACM.

[4] T. Arts, C. Benac Earle, and J. Derrick. Development of a verified Erlang program for resource locking. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2–3):205–220, March 2004.

[5] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. M. Preguiça, M. Najafzadeh, and M. Shapiro. Putting the consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, page 6, 2015.

[6] Basho Technologies. Using strong consistency in Riak. http://docs.basho.com/riak/latest/dev/advanced/strong-consistency/, 2015.

[7] M. Bravo, Z. Li, P. Van Roy, and C. Meiklejohn. Derflow: distributed deterministic dataflow programming for Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 51–60. ACM, 2014.

[8] E. Brewer. Cap twelve years later: How the ”rules” have changed. *Computer*, 45(2):23–29, 2012.

[9] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP*, 2012.

[10] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. San Diego, CA, USA, January 20-21, 2014, pages 271–284. ACM, 2014.

[11] Maria Christakis, Alkis Gotovos, Konstantinos Sagonas, Systematic Testing for Detecting Concurrency Errors in Erlang Programs, In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 154-163, March 18-20, 2013.

[12] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming (ICFP '09)*, pp. 149–160, New York, NY, USA, 2009. ACM.

[13] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM, 2012.

[14] M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *Proc. 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, ACM (2011) 411–422.

[15] M. Emmi, B. K. Ozkan, and S. Tasiran. Exploiting Synchronization in the Analysis of Shared-Memory Asynchronous Programs. In *SPIN 2014: International SPIN Symposium on Model Checking of Software San Jose*, California, USA, July 21-23, 2014.

[16] L.-Å. Fredlund and H. Svensson. McErlang: A model checker for a distributed functional programming language. In *ICFP*. ACM, 2007.

[17] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pages 249–259, New York, NY, USA, 1987. ACM.

[18] Alexey Gotsman and Hongseok Yang. Composite replicated data types. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015*. Volume 9032 of Lecture Notes in Computer Science, pages 585–609. Springer, 2015.

[19] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'Cause I'm Strong Enough: Reasoning about consistency choices in distributed systems. *Submitted for publication*, 2015.

[20] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*. North-Holland, 1983.

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

[22] J. Leitao, J. Pereira, and L. Rodrigues. Epidemic broadcast trees. In *26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE, 2007, pp. 301–310.

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

[24] C. Li, J. a. Leitão, A. Clement, N. Preguiça, and R. Rodrigues. Minimizing coordination in replicated systems. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, pages 8:1–8:4, New York, NY, USA, 2015. ACM.

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

[26] Microsoft. Consistency levels in DocumentDB. http://azure.microsoft.com/en-us/documentation/articles/documentdb-consistency-levels/, 2015.

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

[28] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011.

[29] 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.

[30] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 413–424, New York, NY, USA, 2015. ACM.

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

[32] 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.

[33] 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 *SOSP*, 2013.

[34] W. Vogels. Eventually consistent. *CACM*, 52(1), 2009.

[35] Marek Zawirski. *Dependable Eventual Consistency with Replicated Data Types*. PhD thesis, Université Pierre et Marie Curie (UPMC), Paris, France, Jan. 2015.

[36] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal specification and verification of CRDT. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014*, Berlin, Germany, June 3-5, 2014. Volume 8461 of Lecture Notes in Computer Science, pages 33–48. Springer, 2014.

A Towards Fast Invariant Preservation in Geo-Replicated Systems

Towards Fast Invariant Preservation in Geo-replicated Systems

Valter Balegas, Sérgio Duarte
Carla Ferreira
Rodrigo Rodrigues, Nuno Preguiça
NOVA-LINCS
DI/FCT/Univ. Nova de Lisboa

Mahsa Najafzadeh, Marc Shapiro
Inria & UPMC-LIP6

ABSTRACT

Today's global services and applications are expected to be highly available, scale to an unprecedented number of clients, and offer reliable, low-latency operations. This can be achieved through geo-replication, particularly when data consistency is relaxed. There are, however, applications whose data must obey global invariants at all times. Strong consistency protocols easily address this issue, but require global coordination among replicas and inevitably degrade application throughput and latency.

While coordination is an inherent requirement for maintaining global application invariants, there are instances where coordination on a per operation basis can be avoided. In particular, it has been shown that either moving coordination outside the critical path for executing operations, or having one coordination round for multiple operations, are both effective ways to maintain global invariants and avoid most of the penalties of coordination. However, current geo-replication protocols still have not taken advantage of these observations.

In this paper, we review the design space of current solutions for building geo-replicated applications and present our guiding vision towards a general technique for providing global application invariants under eventual consistency, as a much cheaper alternative to strong consistency.

1. INTRODUCTION

The advent of global Internet-based services and applications has exposed the challenges of building distributed applications targeting millions of users scattered across the globe. Turning users into customers or potential customers of a whole new economy of social networks and e-commerce platforms has highlighted the importance of providing a good user experience. In particular, a measure of quality of service that users perceive directly is the responsiveness of their interactions with the service. There is evidence from major industry players [22] that even a slight degradation in latency correlates with increased user dissatisfaction and, consequently, loss of revenue. In recent years, a great deal of research and technology advances have been directed to addressing this issue.

Geo-replication is a widely adopted technique to improve the responsiveness of online services. It employs multiple data centers, placed at strategic locations around the globe, and attempts to redirect user requests to a nearby replica of the service. Thus, the latency between end-users and the servers can be significantly reduced, in addition to offering improvements in system scalability and fault tolerance.

Under geo-replication, systems scale-out by partitioning data requests [10, 9, 12]. However, the need to replicate databases over high latency, intercontinental network links forces system designers to choose between system availability and data consistency, since it is not possible to have both under network partitions [7]. Eventually consistent and strongly consistent systems are at the opposite extremes of that trade-off.

Eventually consistent systems forgo tight replica coordination to favor availability, allowing replicas to diverge under network partitions [29]. Operations are executed locally and their effects are replicated asynchronously. This allows users to observe the immediate effects of their actions, but can result in concurrency anomalies, due to conflicting operations performed at other sites. In order to maintain global invariants, applications on top of eventually consistent data stores require additional programming logic, thus complicating their design and development.

Strongly consistent systems, in contrast, are well suited for applications that need to enforce global application invariants across replicas at all times [6]. In these systems, data consistency is achieved by limiting concurrency, either by funnelling all updates to a central site, or running some consensus algorithm, such as Paxos, so that all sites agree on some global order of operations. However, performing this level of coordination every time the application state is mutated is expensive, particularly in the case of replicas that are far apart, as expected in geo-replication settings. In either case, throughput and scalability are lower than in eventual consistency.

In an attempt to bridge the gap between availability and consistency, researchers sought to discern what guarantees are attainable without impairing availability [2, 27, 28]. They determined that, under some conditions, *causality* is the strongest form of always-available consistency [2]. However, this is insufficient for enforcing global application invariants, such as ensuring non-negativity of an inventory counter under concurrent decrements.

Others have pursued the approach of combining the best aspects of eventual and strong consistency into systems that choose the most appropriate consistency level for each operation in a workload [26, 15]. Whether that choice is made manually by the programmer (a potentially error prone process) or by a tool [14, 8], it remains that the strongly consistent execution path can still undermine the availability and performance of the system if those operations are frequent.

In this paper, we propose to explore a new trade-off in this tension between consistency and performance, which has the potential to demonstrate that it is possible, in many cases, to achieve the best of both worlds. Our ideas are based on the following observation. While coordination is necessary in general for enforcing global invariants under concurrency, there are also many cases where operations only have the potential to break invariants when particular limit conditions are reached. For instance, when a non-negative counter is far from zero, concurrent decrements do not produce anomalous behaviour, regardless of the order in which they are committed to the database. In other cases, the frequency of unsafe operations in a given workload may provide an opportunity to save on coordination costs. For instance, if two types of operations require coordination between each other, but one is more frequent than the other, then treating these operations in the same way may miss chances for optimization. In particular, by putting the burden of coordination in the rare operation, it becomes possible to execute the more frequent operations without any coordination, since these will be notified in case a conflicting operation wants to execute. These insights make it possible to improve geo-replication performance in a principled way, by moving coordination outside the critical execution path of operations, instead of focusing on the ordering of operations – the approach that is employed by most existing solutions.

The rest of the paper is structured as follows. In section 2, we discuss the guarantees and limitations of eventual consistency (EC). Section 3 presents prior work on using program analysis to determine which operations require coordination to ensure invariants; then, in Section 4 we review additional techniques for enforcing invariants. Section 5 presents the overall approach of our work for providing global application invariants on top of eventual consistency. Section 6 concludes the paper.

2. PROS AND CONS OF EVENTUAL CONSISTENCY

Eventual consistency guarantees that *in the future, if updates cease, all replicas will converge to the same value, becoming indistinguishable* [29]. In systems that offer eventual consistency, clients can access any replica, which allows the system to provide high availability despite failures as long as a single replica is available. Additionally, these systems tend to achieve low latency, as the client can access the closest replica. These advantages come at the price of allowing for behaviors that depart from conventional semantics, and consequently an increased complexity in application design [25].

In this section, we use a social network application to illustrate the anomalies that can occur in eventually consistent systems and how to address them by requiring additional guarantees from the system.

Many of the anomalies occur because a client is allowed execute operations at any replica, without any guarantee that subsequent operations reflect the state observed or written by the current operation. Session guarantees [27] solve most of these anomalies.

For instance, in social networks, users write posts that are added to their own wall and to the walls of all their friends. In this context, the *monotonic reads* guarantee ensures that, after observing some post, subsequent read operations return a state that includes the post (unless it was explicitly removed). To ensure that a user always reads her previous post within a session, the system must provide the *read your writes* guarantee. These properties are trivially satisfied if a user always reads from and writes to the same server [2].

A system providing the *writes follows reads* guarantee ensures that a user always sees the posts that lead to a reply she observes, which is particularly useful to keep conversations coherent. A system that enforces causality [13] provides the previous invariants. Many recent systems provide causal consistency [17, 1, 18, 32].

The anomalies we previously described refer to operations that execute one mutation at the time, but other operations may have multiple effects. For example, in social network systems, friendship is usually a bi-directional relation, i.e., if user A is a friend of user B, user B is also friend of user A. As such, when a friend request is acknowledged, both friend lists must be updated. Updating the friend lists without atomicity may result in some user observing that A is friend of B but B is not friend of A, or vice-versa. This violates the friendship relation invariant. To address this, some geo-replicated systems provide atomicity for a sequence of writes, while also enforcing causality [18, 32, 26].

All the above invariants can be provided with existing causally consistent systems. Now we consider other more complex invariants to show when eventual consistency falls short of achieving the preservation of application invariants. In a social network, users can join a group of users after receiving the invitation from the administrator. This rule is easily enforced by using causal consistency, which guarantees that the acceptance of an invitation will always follow the invitation itself. However, stricter semantics would be impossible to enforce relying only on causal consistency, particularly when concurrent operations can lead to a state where the invariant is violated. For example, it is impossible to guarantee that every member of the group is a friend of the administrator of the group, since a friendship relationship could be cancelled while a user concurrently joins the group. Generically, this is an instance of a referential integrity invariant. This type of invariant can be repaired after the violation is detected – e.g., by removing from the group the members that are no longer friends of the administrator.

Other invariants may not have a trivial repair function – consider that an award is given to a limited number of users in the group. A system relying on causal consistency could concurrently give out more awards than the limit. In this case, there is no trivial solution to select the users that should remain in the set of awardees, and the situation can be particularly problematic in case the award emails have been sent out.

The examples presented in this section show that there are several additional guarantees that eventually consistent systems should provide. However, in some cases these guarantees can be particularly difficult to enforce under eventual consistency, even with the help of a repair function. As such, to address these requirements, applications tend to adopt strong consistency models (or at least provide support for both weak and strong semantics [26, 15]).

3. MAKING THE RIGHT CHOICE

In the previous section we have seen that not all operations have the same consistency requirements. For this reason, many existing systems take the approach of supporting different levels of consistency, in order to make a more selective use of strong consistency.

Gemini [15], Bloom^L [8], Walter [26] allow developers to choose between different levels of consistency to ensure application correctness. This approach enables using eventual consistency when operations are compatible with any possible concurrent updates, and only using strong consistency when concurrent operations can make the database inconsistent. This allows for fine tuning the consistency requirements of each operation. However, it poses a heavy burden on the programmer, who must decide the correct level of consistency to use: if the programmer is too conservative, this may lead to an inefficient application; if the programmer is too relaxed, due to incorrect reasoning about the application semantics, this can lead to incorrect behavior. Recent work proposed to identify the best consistency level automatically, thus not requiring the programmer to analyze the consequences different classification choices. In particular, Sieve [14] determines the consistency level for operations that run on top of Gemini, under RedBlue consistency. It combines static and dynamic analyses to determine which operations are safe under causal consistency, and which operations need serializability to maintain invariants. The analysis considers a set of user-provided invariants and small annotations that specify the convergence techniques used for concurrent operations on the same objects.

The first step of the analysis, completed offline, generates abstract models that represent the space of possible concurrent executions during runtime and, for each model, determines the set of minimal pre-conditions for being safe to execute the operation without coordination.

At runtime, an operation executes under causal consistency if the minimal pre-conditions for weak execution determined offline are matched. Otherwise, the operation executes under strong consistency.

For example, the offline algorithm would determine that any operation that adds a negative value to a non-negative stock is unsafe to be executed under eventual consistency (as concurrent operations can lead the stock to become negative). At runtime, if an operation adds a positive value, it will execute under eventual consistency; otherwise it needs to execute under strong consistency.

Bloom^L[8] is a logic programming language for distributed applications that maintains application invariants. It is based on the observation that monotonic programs never retract information that is previously known, and therefore they converge regardless the delivery order of messages in different replicas. A total order of messages is only required for non-monotonic operations. An important part of Bloom^L is the CALM analysis that allows for identifying which parts of the program are non-monotonic.

The Bloom^L language provides a library of semi-lattice constructs that ensure convergence, similar to CRDTs[23]. The language supports non-monotonic operators: operators that may give different results depending on the arrival order of remote messages. For executing a non-monotonic operator, a coordination protocol must be executed, to ensure that the result of the non-monotonic operation is equivalent in all replicas.

Both strategies identify which operations may break invariants, and then only enforce coordination among replicas to execute those operations. This strategy is conservative, as in many executions it is safe to execute the operations without coordination. For instance, in the stock example, coordination is only necessary when the number of available units becomes low, but the system is forced to coordinate on every request because it does not take the current level of the stock into account.

When determining if an operation can execute without coordination, Bloom^L looks only at the code of operations, while Sieve takes into consideration both the code of the operation and the value of parameters. In the latter case, the final decision on whether coordination is necessary or not is determined in runtime. We argue that it is possible to extend this approach by considering also the state of the database. This has the potential to reduce, or in some executions even completely avoid the cost of global coordination by extending conflict analysis with runtime information about the database and the participants.

In the literature, some proposals use an estimate of replica divergence to avoid coordination [31, 11], either by using deterministic or stochastic models. However, these techniques cannot be applied to general invariants and only give an estimate of the divergence, allowing invariants to be broken in certain scenarios.

4. OLD TECHNIQUES REVISITED

In this section, we revisit two proposals that we build on: the escrow transactional method [19] and the demarcation protocol [5]. Then, in the next section, we discuss how to use these protocols to provide the invariants from Section 2 without using strong consistency, or global replica coordination in the general case.

The escrow model [19] was proposed to allow long-lived transactions to commit without interfering with other ongoing transactions. The key idea is to divide resources into *escrows* that can be used concurrently by different nodes. If the client has enough resources in its escrow, it can execute the operation without coordination and release the remaining resources on commit, or abort.

In the example of the limited number of awards, consider that each group has a limit of K awards. Each node i that holds a copy of group G grants awards up to a limit Y_i such that $\sum_{i=1}^{n} Y_i \leq K$, where n is the number of copies of G. While the number of given awards do not exceed the local limit Y_i, each node can execute the operations locally with low latency.

This model has been extended to support different partitionable data types [30] and operations [20, 24], but all these implementations rely on a central component to manage escrows.

The demarcation protocol [5] has a similar insight to the escrow model, but enforces invariants over multiple variables. For each variable, the protocol defines a limit for its value. The combination of the defined limits for all variables guarantees that the invariants remain valid. Thus, operations are safe if the updates do not exceed the defined limits.

If an operation requires a variable to exceed its limit, then the node executing the operation must exchange its limit with another peer to make that operation safe: the requester node sends a request with the change in the limit it requires; the node that accepts the request adjust its own limits and notifies the requester of the change; the requester then increase its safety limits with the received delta and the operation proceeds locally.

Changing the limits with point-to-point communication can be fast when nodes know enough information about the other peers. When the resources are scarce and nodes change the limits more frequently some requests might fail, leading to multiple point-to-point messages. Additionally, the point-to-point protocol needs to enforce exactly-once delivery, otherwise the limits may become more restrictive than necessary.

The authors have used this protocol to maintain a numeric invariant over resources distributed in multiple machines, to enforce uniqueness invariants, and to provide referential integrity constraints.

A referential integrity constraint is modeled by a logical implication: $predicate(A) \Rightarrow predicate(B)$, where each node stores a boolean value for each predicate, A and B. The idea is to enforce that whenever a node updates a predicate to a value that may turn the expression false (unsafe), it must enforce that the other nodes changes the value of their other predicate to maintain the expression true. In our example, we have $JoinGroup(A, G) \Rightarrow isFriend(A, B)$, where A is a user, B the administrator and G a group of users. Making $JoinGroup(A, G)$ *true* is unsafe because that value is only allowed if $isFriend(A, B)$ is true, otherwise the expression is false. Therefore the node requests the peer holding the predicate $isFriend(A, B)$ to change the minimum value for that predicate to true. The converse must also be ensured: to make $isFriend(A, B)$ *false*, the node must ask the peer holding the value for predicate $JoinGroup(A, G)$ to enforce that it becomes false.

More recently, MDCC [12] uses a variation of the demarcation protocol to extract more concurrency from commutative operations that maintain numerical constraints invariants. The homeostasis protocol [21] also extends the demarcation protocol, but requires a new set of conditions to be computed and installed in all replicas using two-phase commit. Warranties [16] provide guarantees that some properties of the state will not change optimizing read operations in centralized strong consistency architectures. We argue that it is possible to leverage these existing ideas in modern geo-replicated settings, relying only on peer-to-peer and unreliable asynchronous communication protocols. The next section shows how this can be achieved.

5. LOW-COST INVARIANTS

In the previous sections, we showed techniques that allow for the maintenance of database invariants in two different ways: the first is to identify the operations that are unsafe, and resort to strong consistency to execute them, and the second is to enforce local constraints to ensure that operations are safe, even while replicas diverge. We argue that a combination of these techniques can be used to provide a principled approach to execute operations that maintain invariants without coordination in the general case.

We envision a system that identifies operations that require strong consistency, but use an efficient protocol to guarantee that local executions are safe instead of using global coordination. The system would exchange the necessary resources outside the critical path of execution, to attempt to guarantee that operations execute locally while ensuring safety. Furthermore, the system can resort to strong consistency when those requirements are not met.

The high level idea is to perform an analysis of the code, along the lines of that done by the Sieve system, to determine the weakest preconditions to accept a set of facts, computed during runtime, to enable the execution of operations locally. For instance if the weakest precondition to execute $joingroup(A, G)$ is that $isFriend(A, B)$ is true, then we could add a fact that gives the local replica the exclusive right to modify that predicate, which would ensure that it does not become false. At runtime, if the current replica holds that guarantee, it can execute the operation without coordination because it knows that the value of that predicate can only change locally. Otherwise, it should resort to strong consistency to execute the operation.

We have previously presented a preliminary design of a data-type that maintains numerical invariants [4], which can be seen as a special case of that approach. Our data-type maintains the full state of the invariant, which allows the current value to be queried by a client. This is unlike the demarcation protocol, which may require contacting multiple nodes before knowing the actual value of the inequality. This is a first step towards providing data-types that are able to preserve the demarcation protocol invariants in a replicated system.

Our approach is able to maintain different forms of invariants, and we already have a data-type that enforces numerical invariants. However, the extent of invariants that we can express remains an open question. Bailis et al. [3] conducted a survey of the typical invariants in benchmarks and concluded that the most common invariants have the form of referential integrity, numerical constraints and uniqueness, which can all be implemented with the demarcation protocol.

To our knowledge, none of the previous approaches can be directly applied to implement our vision. None of the previous prposals addresses all the key points in building geo-replicated databases: either they only capture limited forms of invariants, do not deal with data replication, rely on a central components to manage resources, or do not provide low latency, fault tolerance and scalability to a large number of clients.

6. CONCLUSION

Current systems give up low latency and availability for consistency when invariants are essential to applications. At best, only those invariants that are compatible with eventual consistency can be enforced with low latency. For the rest, the default has been to rely on strong consistency. To help figure out which consistency level to use, recent research has produced techniques that help programmers sort out which parts of a program are unsafe under concurrency and which need global coordination.

In this paper, we propose to build on existing research to further avoid paying the full cost of coordination while enforcing global invariants on top of eventual consistency. To the best of our knowledge, no current implementations are tailored to harness these techniques on cloud infrastructures.

This paper conducts a literature review, to motivate that our proposed approach applies to the most frequent application invariants. Furthermore, we gave initial steps in this research direction, by designing a data-type that maintains numerical invariants with low latency. We are now adapting these protocols to be deployed in geo-replicated systems, and exploring the use of program analysis to determine when our optimizations can be applied.

Acknowledgments

This research is supported in part by EU FP7 SyncFree project (609551), FCT/MCT SFRH/BD/87540/2012, PTDC/EEI-SCR/ 1837/ 2012 and PEst-OE/ EEI/ UI0527/ 2014. The research of Rodrigo Rodrigues is supported by the European Research Council under an ERC Starting Grant.

7. REFERENCES

[1] S. Almeida, J. Leitão, L. Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proc. EuroSys '13*, 2013.

[2] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. Hellerstein, I. Stoica. Highly available transactions: Virtues and limitations. *PVLDB*, 7(3):181–192, 2013.

[3] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. Hellerstein, and I. Stoica. Coordination Avoidance in Database Systems. In *VLDB 2015*, 2015.

[4] V. Balegas, M. Najafzadeh, S. Duarte, C. Ferreira, R. Rodrigues, M. Shapiro, and N. Preguiça. The case for fast and invariant-preserving geo-replication. In *Proc. W-PSDS 2014*, 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, 1994.

[6] P. A. Bernstein, V. Hadzilacos, N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1986.

[7] E. A. Brewer. Towards robust distributed systems (abstract). In *Proc. PODC'00*, 2000.

[8] N. Conway, W. Marczak, P. Alvaro, J. Hellerstein, D. Maier. Logic and lattices for distributed programming. In *Proc. SoCC'12*, 2012.

[9] J. Corbett, J. Dean, M. Epstein, et. al. Spanner: Google's globally-distributed database. In *Proc. OSDI'12*, 2012.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP'07*, 2007.

[11] T. Kraska, M. Hentschel, G. Alonso, D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. In *Proc. VLDB Endow.*, 2(1):253–264, 2009.

[12] T. Kraska, G. Pang, M. Franklin, S. Madden, A. Fekete. Mdcc: Multi-data center consistency. In *Proc. EuroSys '13*, 2013.

[13] L. Lamport. Time, clocks, the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, 1978.

[14] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proc. USENIX ATC'14*, 2014.

[15] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proc. OSDI'12*, 2012.

[16] J. Liu, T. Magrino, O. Arden, M. George, A. Myers. Warranties for faster strong consistency. In *Proc. NSDI'14*, 2014.

[17] W. Lloyd, M. Freedman, M. Kaminsky, D. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proc. SOSP'11*, 2011.

[18] W. Lloyd, M. Freedman, M. Kaminsky, D. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proc. NSDI'13*, 2013.

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

[20] N. Preguiça, J. L. Martins, M. Cunha, H. Domingos. Reservations for conflict avoidance in a mobile database system. In *Proc. MobiSys'03*, 2003.

[21] S. Roy, L. Kot, N. Foster, J. Gehrke, H. Hojjat, C. Koch. Writes that fall in the forest and make no sound: Semantics-based adaptive data consistency. *CoRR*, abs/1403.2307, 2014.

[22] E. Schurman and J. Brutlag. Performance related changes and their user impact. In Velocity, 2009.

[23] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski. Conflict-free replicated data types. In *Proc. SSS'11*, 2011.

[24] L. Shrira, H. Tian, D. Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proc. Middleware '08*, 2008.

[25] J. Shute, R. Vingralek, B. Samwel, et. al. F1: A distributed SQL database that scales. *PVLDB*, 6(11):1068–1079, 2013.

[26] Y. Sovran, R. Power, M. K. Aguilera, J. Li. Transactional storage for geo-replicated systems. In *Proc. SOSP'11*, 2011.

[27] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, B. Welch. Session guarantees for weakly consistent replicated data. In *Proc. PDIS'94*, 1994.

[28] D. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proc. SOSP'13*, 2013.

[29] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. SOSP'95*, 1995.

[30] G. Walborn and P. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Proc. SRDS'95*, 1995.

[31] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proc. OSDI'00*, 2000.

[32] 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. *CoRR*, abs/1310.3107, 2013.

B Lasp: A Language for Distributed Coordination-free Programming

Lasp: A Language for Distributed, Coordination-Free Programming

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

Peter Van Roy
Université catholique de Louvain
peter.vanroy@uclouvain.be

Abstract

We propose Lasp, a new programming model designed to simplify large-scale distributed programming. Lasp combines ideas from deterministic dataflow programming together with conflict-free replicated data types (CRDTs). This provides support for computations where not all participants are online together at a given moment. The initial design presented here provides powerful primitives for composing CRDTs, which lets us write long-lived fault-tolerant distributed applications with nonmonotonic behavior in a monotonic framework. Given reasonable models of node-to-node communications and node failures, we prove formally that a Lasp program can be considered as a functional program that supports functional reasoning and programming techniques. We have implemented Lasp as an Erlang library built on top of the Riak Core distributed systems framework. We have developed one nontrivial large-scale application, the advertisement counter scenario from the SyncFree research project. We plan to extend our current prototype into a general-purpose language in which synchronization is used as little as possible.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; E.1 [*Data Structures*]: Distributed data structures

Keywords Eventual Consistency, Commutative Operations, Erlang

1. Introduction

Synchronization of data across systems is becoming increasingly expensive and impractical when running at the scale required by "Internet of Things" [29] applications and large online mobile games.[1] Not only does the time required to coordinate with an ever growing number of clients increase with each additional client, but techniques that rely on coordination of shared state, such as Paxos and state-machine replication, grow in complexity with partial replication, dynamic membership, and unreliable networks. [14]

This is further complicated by an additional requirement for both of these applications: each must tolerate periods without connectivity while allowing local copies of replicated state to change. For example, mobile games should allow players to continue to accumulate achievements or edit their profile while they are riding in the subway without connectivity; "Internet of Things" applications should be able to aggregate statistics from a power meter during a snowstorm when connectivity is not available, and later synchronize when connectivity is restored. Because of these requirements, the burden is placed on the programmer of these applications to ensure that concurrent operations performed on replicated data have both a deterministic and desirable outcome.

For example, consider the case where a user's gaming profile is replicated between two mobile devices. Concurrent operations, which can be thought of as operations performed during the period where both clients are online but without communication, can modify the same state: the burden is placed on the application developer to write application logic that resolves these conflicting updates. This is true even if the changes commute: for instance, concurrent modifications to the user profile where client A modifies the profile photo and client B modifies the profile's e-mail address.

Recently, a formalism has been proposed by Shapiro et al. for supporting deterministic resolution of individual objects that are acted upon concurrently in a distributed system. These data types, referred to as Conflict-Free Replicated Data Types (CRDTs), provide a property formalized as Strong Eventual Consistency: given all updates to an object are eventually delivered in a distributed system, all copies of that object will converge to the same state. [32, 33]

Strong Eventual Consistency (SEC) results in deterministic resolution of concurrent updates to replicated state. This property is highly desirable in a distributed system because it no longer places the resolution logic in the hands of the programmer; programmers are able to use replicated data types that function as if they were their sequential counterparts. However, it has been shown that arbitrary composition of these data types is nontrivial. [6, 13, 15, 26]

To achieve this goal, we propose a novel programming model aimed at simplifying correct, large-scale, distributed programming, called Lasp.[2] This model provides the ability to use operations from functional programming to deterministically compose CRDTs into larger computations that observe the SEC property; these applications support programming with data structures whose values appear nonmonotonic externally, while computing internally with the objects' monotonic metadata. This model builds on our previous work, Derflow and Derflow$_L$ [12, 27], which provide a distributed,

[1] Rovio, developer of the popular "Angry Birds" game franchise reported that during the month of December 2012 they had 263 million active users. This does not account for users who play the game on multiple devices, which is an even larger number of devices requiring some form of shared state in the form of statistics, metrics, or leaderboards. [3]

[2] Inspired by LISP's etymology of "LISt Processing", our fundamental data structure is a join-semilattice, hence Lasp.

PPDP '15, July 14 - 16, 2015, Siena, Italy.
Copyright © 2015 ACM 978-1-4503-3516-4/15/07...$15.00.
http://dx.doi.org/10.1145/2790449.2790525

fault-tolerant variable store powering a deterministic concurrency programming model.

This paper has the following contributions:

- **Formal semantics:** We provide the formal semantics for Lasp: the monotonic **read** operation; functional programming operations over sets, including **map**, **filter**, and **fold**; and set-theoretic operations, including **product**, **union**, and **intersection**.
- **Formal theorem:** We formally prove that a distributed execution of a Lasp program can be considered a functional program that supports functional reasoning and programming techniques.
- **Prototype implementation:** We provide a prototype implementation [1] of Lasp, implemented as an Erlang library using the Riak Core [22] distributed systems framework.
- **Initial evaluation:** We perform an initial evaluation of Lasp by prototyping the eventually consistent advertisement counter scenario from the SyncFree project [4] and improve on the design of the Bloom KVS presented by Conway et al. [15]

This paper is an extension of the previously published work-in-progress report on Lasp [28] and is structured as follows: Section 2 motivates the need for Lasp. Section 3 defines Lasp's semantics and operations. Section 4 defines and proves the fundamental theorem of Lasp execution. Section 5 explains the implementation of our prototype. Section 6 evaluates the expressiveness of our prototype by showing how to implement two nontrivial distributed applications. Section 7 explains how Lasp is related to other work on models and languages for distributed programming. Section 8 outlines the extensions that we are planning for the Lasp prototype. Section 9 gives a brief conclusion.

2. Motivation

In this section, we motivate the need for Lasp.

2.1 Conflict-Free Replicated Data Types

Conflict-Free Replicated Data Types [32, 33] (CRDTs) are distributed data types that are designed to support temporary divergence at each replica, while guaranteeing that once all updates are delivered to all replicas of a given object they will converge to the same state. There are CRDT designs for commonly used sequential data types: counters, registers, sets, flags, dictionaries, and graphs; however, each of these data structures, while guaranteeing to converge, will observe a predetermined bias on how to handle concurrent operations, given that behavior is undefined in its sequential counterpart.

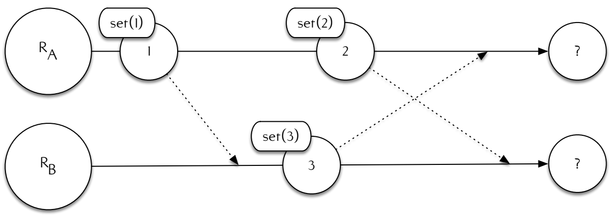


Figure 1: *Example of divergence due to concurrent operations on replicas of the same object. In this case, it is unclear which update should win when replicas eventually communicate with each other.*

We provide an example in Figure 1. In this example, concurrent operations on a replicated register result in divergence at each replica: replica A (R_A) is set to the value 2 whereas replica B (R_B) is set to the value 3. How do we reconcile this divergence?

Two major strategies have been used in practice by several production databases [18, 25]: "Last-Writer-Wins", where the last object written based on wall clock time wins, or "Semantic Resolution" where both updates are stored at each replica, and the burden of resolving the divergence is placed on the developer.

Both of these strategies have deficiencies:

- "Last-Writer-Wins" ultimately results in some valid operations being dropped during merge operations based solely on the scheduling of operations.
- "Semantic Resolution" places the burden on the application developer to provide a deterministic merge function.[3]

CRDTs solve this problem by formalizing a series of data types that fulfill two major goals: capturing causal information about updates that have contributed to their current state, and providing a deterministic merge operation for combining the state across multiple replicas.

Figure 2 shows one possible way to define a merge function that is deterministic regardless of ordering of messages: if we take advantage of the order of natural numbers using the *max* operation, we ensure that all replicas will eventually converge to a correct, equivalent state once they have observed all updates. While this is one very simple example of a distributed data structure with a deterministic merge function, Shapiro et al. outline different designs for registers, sets, counters, and graphs [33].

We will now formalize the Observed-Remove Set CRDT to explore problems of composition with CRDTs that have visible nonmonotonic behavior.

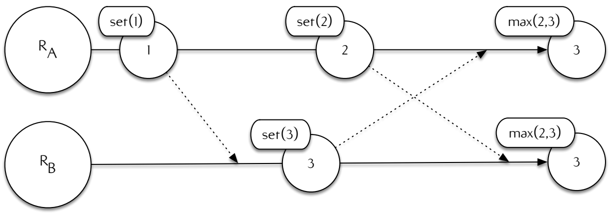


Figure 2: *Example of resolving concurrent operations with a type of state-based CRDT based on a natural number lattice where the join operation computes max.*

2.2 Observed-Remove Set CRDT

We take a moment to introduce the Observed-Remove Set CRDT (OR-Set), which will be used as the basis for the formalisms in this paper. We focus on the OR-Set because it is the least complex CRDT which serves as a general building block for applications.[4]

We start with a description of lattices, which are used as the basis of state-based CRDTs, one of the two major types of CRDTs.

Definition 2.1. A **bounded join-semilattice** is a partially ordered set that has a binary operation called the *join*. The *join* is associative, commutative, and idempotent, and induces a partial order over a nonempty finite subset such that the result given any two elements

[3] When described by DeCandia et al. in 2007 [18], this mechanism results in "concurrency anomalies", where updates seem to reappear due to concurrent operations in the network; this is the main focus of the the CRDT work as presented by Shapiro et al. in [32, 33].

[4] For instance, the Grow-Only Set (G-Set) does not allow removals, the Two-Phase Set (2P-Set) only allows one removal of a given item, and the OR-Set Without Tombstones (ORSWOT) adds additional complexity in the form of optimizations, which lie outside of the core semantics.

is the least upper bound of the input with respect to the partial order. The semilattice is bounded, as it contains a least element. [16]

Definition 2.2. A replicated triple (S, M, Q) where S is a bounded join-semilattice representing the state of each replica, M is a set of functions for mutating the state, and Q is a set of functions for querying the state, is one type of **state-based CRDT**. [5]

Functions for querying and mutating the CRDT can always be performed as they are executed on the replica's local state and the entire state is propagated to other replicas in the replica set. When a replica receives a state from another replica, the received state is merged into the local state using the $join$ operation. Given the algebraic properties of the $join$ operation, once updates stop being issued, a join across all replicas produces equivalent state; the merge function is deterministic given a finite set of updates.

Mutations at a given replica always return a monotonically greater state as defined by the partial order of the lattice, therefore any subsequent state always subsumes a previous state. We refer to these mutations as **inflations**.

Definition 2.3. A **stream** s is an infinite sequence of states of which only a finite prefix of length n is known at any given time.

$$s = [s_i \mid i \in \mathbb{N}] \tag{1}$$

The execution of one CRDT replica is represented by a stream of states, each of which is an element of the lattice S. The execution of a full CRDT instance with n replicas is represented by n streams.

Definition 2.4. Updates performed on a given state are **inflations**; for any mutation, the state generated from the mutation will always be strictly greater than the state generated by the previous mutation.

$$m \in M \wedge s_i \in S \wedge s_i \sqsubseteq m(s_i) \tag{2}$$

The **Observed-Remove Set** CRDT models arbitrary nonmonotonic operations, such as additions and removals of the same element, monotonically in order to guarantee convergence with concurrent operations at different replicas.

Definition 2.5. The **Observed-Remove Set (OR-Set)** is a state-based CRDT whose bounded join-semilattice is defined by a set of triples, where each triple has one value v, and extra information (called *metadata*) in the form of an add set a and a remove set r. At most one triple may exist for each possible value of v.

$$s_i = \{(v, a, r), (v', a', r'), \ldots\} \tag{3}$$

The OR-Set has two mutations, **add** and **remove**. The metadata is used to implement both mutations monotonically.

Definition 2.6. The **add** function on an OR-Set generates a unique constant u for each invocation. Given this constant, add u to the add set a for value v, if the value already exists in the set, or add a new triple containing v, an add set $\{u\}$ and a remove set $\{\}$.

$$\begin{aligned} add(s_i, v) = s_i - \{(v, \_, \_)\} \\ \cup \{(v, a \cup \{u\}, r) \mid (v, a, r) \in s_i \wedge u = unique()\} \\ \cup \{(v, \{u\}, \{\}) \mid \neg(v, \_, \_) \in s_i \wedge u = unique()\} \end{aligned} \tag{4}$$

Definition 2.7. The **remove** function on an OR-Set for value v unions all values in the add set for value v into the remove set for value v.

$$remove(s_i, v) = s_i - \{(v, \_, \_)\} \cup \{(v, a, a \cup r) \mid (v, a, r) \in s_i\} \tag{5}$$

The OR-Set has one query function that returns the current contents of the set: **query**.

Definition 2.8. The **query** function for an OR-Set returns values which are currently present in the set. Presence of a value v in a given state s_i is determined by comparison of the remove set r with the add set a. If the remove set r is a proper subset of the add set a, the value v is present in the set.

$$query(s_i) = \{v \mid (v, a, r) \in s_i \wedge r \subset a\} \tag{6}$$

The Observed-Remove Set is one instance of a CRDT that has a **query** function that is nonmonotonic: the data structure allows arbitrary additions and removals of elements in a set. It is important to distinguish between the external representation of the set (the output of a query, which is nonmonotonic) and the internal representation (the result of add and remove operations, which are monotonic).

2.3 Composition

The convergence properties of CRDTs are highly desirable for computation in distributed systems: these data structures are resilient to update reordering, duplication, and message delays, all of which are very relevant problems for computation on an unreliable asynchronous network. However, these convergence properties only hold for individual replicated objects and do not extend to computations that compose more than one CRDT.

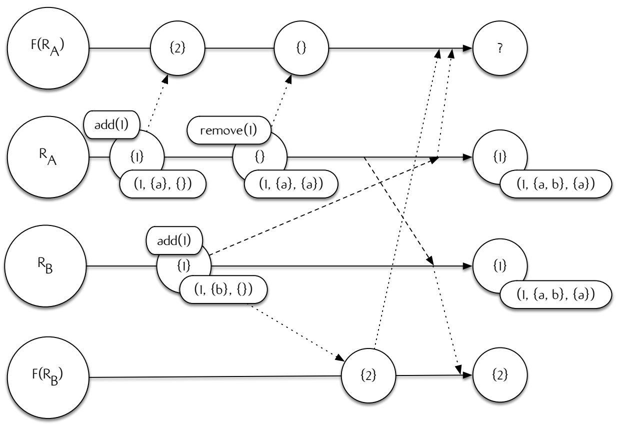


Figure 3: *Example of CRDT composition. In this example, there are two replicas of a CRDT, R_A and R_B; a function F, defined as $\lambda x.2x$, is applied to each element in the set at each replica using the **map** function. Without properly mapping the metadata, the convergence property does not hold for the result of the function application.*

In Figure 3, we see an example where the internal state of both replicas A and B (R_A and R_B) allows us to reason about state that reflects visible nonmonotonic behavior, additions and removals of the same element, by modeling the state changes monotonically. However, if we apply a function to the external representation of the value then we sacrifice the convergence property.

This example describes a case where the output of a function F, defined as $\lambda x.2x$, applied to each element at replica A ($F(R_A)$) using the **map** function receives state from the same function applied to the elements at replica B ($F(R_B)$). It is unclear how to merge the incoming state given we can not determine if the incoming state has been previously observed or not.[5]

3. Lasp

We now present the API and semantics of Lasp, a programming model designed for building convergent computations by composing CRDTs.

[5] Technically, in this naive mapping the state and value are the same.

3.1 API

Lasp's programming model is provided as a library in the Erlang programming language. This library implements the core semantics of Lasp and provides a distributed runtime for executing Lasp applications. The primary data type of Lasp is the CRDT. Given a CRDT instance of type t, the Lasp API is designed as follows:

Core API Core functions are responsible for defining variables, setting their values and reading the result of variable assignments.

- $\mathit{declare}(t)$: Declare a variable of type t.[6]
- $\mathit{bind}(x, v)$: Assign value v to variable x. If the current value of x is w, this assigns the join of v and w to x.
- $\mathit{update}(x, \mathrm{op}, a)$: Apply op to x identified by constant a.
- $\mathit{read}(x, v)$: Monotonic read operation; this operation does not return until the value of x is greater than or equal to v in the partial order relation induced over x at which time the operation returns the current value of x.
- $\mathit{strict\_read}(x, v)$: Same as $\mathit{read}(x, v)$ except that it waits until the value of x is strictly greater than v.

Functional Programming API Functional programming primitives define processes that never terminate; each process is responsible for reading subsequent values of the input and writing to the output. Figure 4 shows use of the **map** function.

- $\mathit{map}(x, f, y)$: Apply function f over x into y.
- $\mathit{filter}(x, p, y)$: Apply filter predicate p over x into y.
- $\mathit{fold}(x, \mathrm{op}, y)$: Fold values from x into y using operation op.

Set-Theoretic API Set-theoretic functions define processes that never terminate; each process is responsible for reading subsequent values of the input and writing to the output.

- $\mathit{product}(x, y, z)$: Compute product of x and y into z.
- $\mathit{union}(x, y, z)$: Compute union of x and y into z.
- $\mathit{intersection}(x, y, z)$: Compute intersection of x and y into z.

3.2 Processes

The previously introduced Lasp operations, functional and set-theoretic, create processes that connect all replicas of two or more CRDTs. Each process tracks the monotonic growth of the internal state at each replica and maintains a functional semantics between the state of the input and output instances. Each process correctly transforms the internal metadata of the input CRDTs to compute the correct mapping of value and metadata for the output CRDT.[7] For example, the Lasp **map** operation can be used to connect two instances of the Observed-Remove Set CRDT.

In the **map** example seen in Figure 4, whenever an element e is added or removed from the input set, the mapped version $f(e)$ is correctly added or removed from the output set. The other operations provided by Lasp are analogous: the user visible behavior is the normal result of the functional or set-theoretic function.

3.3 Variables

As we will prove in Section 4, each state-based CRDT in Lasp has the appearance of a single state sequence that evolves monotonically over time as update operations are issued; this is similar to the definition of inflation provided earlier (Definition 2.4). The current state of the CRDT is stored in a variable; successive values of the variable form the CRDT's state sequence.

We now formally define variables in Lasp and invariants the Lasp system preserves for each variable.

```
%% Create initial set S1.
{ok, S1} = lasp:declare(riak_dt_orset),

%% Add elements to initial set S1 and update.
{ok, _} = lasp:update(S1, {add_all, [1,2,3]}, a),

%% Create second set S2.
{ok, S2} = lasp:declare(riak_dt_orset),

%% Apply map operation between S1 and S2.
{ok, _} = lasp:map(S1, fun(X) -> X * 2 end, S2).
```

Figure 4: *Map function applied to an OR-Set using the Erlang API of our Lasp prototype. We ignore the return values of the functions, given the brevity of the example.*

3.4 Monotonic Read

The *monotonic read* operation ensures that read operations always read an equivalent or greater value when provided with the result of a previous read. This behavior is very important to our system when dealing with replicated data to ensure forward progress. Consider the following example:

- Variable a is replicated three times, on three nodes: a_1, a_2, a_3.
- Application reads variable a from replica a_1.
- Application modifies replica a_1; state is then asynchronously propagated to replicas a_2 and a_3.
- Application reads variable a from replica a_2, because replica a_1 is temporarily unreachable.

In this example, it is possible for replica a_2 to temporarily have previous state than replica a_1, given message delays, failures, and asynchronous replication.[8] The *monotonic read* operation ensures that the read will not complete until an equivalent or greater state as defined over the partial order for a's lattice is available at a given replica based on a *trigger* value.

Formally, we define the *monotonic read* operations as follows:

Definition 3.1. The **monotonic read** operation defines a process that reads the known elements of the input stream s and waits until some s_i is equal to or monotonically greater than s_e. At this point, s_i is returned.

$$\begin{aligned} \mathit{read}(s, s_e) = \exists i.\ & s_i \in s \wedge s_e \sqsubseteq s_i \\ & [t_j \mid t_j = (j \geq i \Rightarrow s_i; \bot)] \end{aligned} \tag{7}$$

We also provide a strict version of the *monotonic read* operation, which does not return until a strict inflation of a previous read has been observed. This allows us to build recursive functions, such as our functional programming operations and set-theoretic operations, in terms of tail-recursive processes which continuously observe increasing state.

We define the strict version of the *monotonic read* operation as follows:

[6] Given the Erlang programming library does not have a rich type system, it is required to declare CRDTs with an explicit type at initialization time.

[7] The internal metadata of each CRDT is responsible for ensuring correct convergence; the transformation is therefore required to be deterministic.

[8] This is a core idea behind eventual consistency and replication strategies such as optimistic replication. Eventually consistent systems ensure updates are eventually visible (i.e., in finite time), but make no guarantees about when the updates will be visible. [18, 31]

Definition 3.2. The **monotonic strict read** operation defines a process that reads the known elements of the input stream s and waits until some s_i is monotonically greater than s_e. At this point, s_i is returned.[9]

$$\begin{aligned} \mathit{strict\_read}(s, s_e) = \exists i.\ s_i \in s \wedge s_e \sqsubset s_i \\ [t_j \mid t_j = (j \geq i \Rightarrow s_i; \bot)] \end{aligned} \tag{8}$$

3.5 Functional Programming

We now look at the semantics for functional programming primitives that are lifted to operate over CRDTs: **map**, **filter**, and **fold**. We formalize them as follows:

Definition 3.3. The **map** function defines a process that never terminates, which reads elements of the input stream s and creates elements in the output stream t. For each element, the value v is separated from the metadata, the function f is applied to the value, and new metadata is attached to the resulting value $f(v)$. If two or more values map to the same $f(v)$ (for instance, if the function provided to map is surjective), the metadata is combined into one triple for all values of v.

$$\begin{aligned} F(s_i, f) &= \{f(v) \mid (v, \_, \_) \in s_i\} \\ A(s_i, f, w) &= \bigcup \{a \mid (v, a, \_) \in s_i \wedge w = f(v)\} \\ R(s_i, f, w) &= \bigcup \{r \mid (v, \_, r) \in s_i \wedge w = f(v)\} \\ \mathit{map}'(s_i, f) &= \{(w, A(s_i, f, w), R(s_i, f, w)) \mid w \in F(s_i, f)\} \\ \mathit{map}(s, f) &= t = [\mathit{map}'(s_i, f) \mid s_i \in s] \end{aligned} \tag{9}$$

Figure 4 provides an example of applying the **map** function to an OR-Set. In this example, the user does not need to know the internal data structure of each CRDT, but only the nonmonotonic external representation, as the Lasp runtime handles the metadata mapping automatically.

Definition 3.4. The **filter** function defines a process that never terminates, which reads elements of the input stream s and creates elements in the output stream t. Values for which $p(v)$ does not hold are removed by a metadata computation, to ensure that the filter is a monotonic process.

$$\begin{aligned} \mathit{filter}'(s_i, p) &= \{(v, a, r) \mid (v, a, r) \in s_i \wedge p(v)\} \\ &\cup \{(v, a, a \cup r) \mid (v, a, r) \in s_i \wedge \neg p(v)\} \\ \mathit{filter}(s, p) &= t = [\mathit{filter}'(s_i, p) \mid s_i \in s] \end{aligned} \tag{10}$$

Definition 3.5. The **fold** function defines a process that never terminates, which reads elements of the input stream s and creates elements in the output stream t. Given $\mathit{query}(s_i) = V = \{v_0, ..., v_{n-1}\}$ and an operation op of t's type with neutral element e, this should return the state $t_i = e \text{ op } v_0 \text{ op } v_1 \cdots \text{ op } v_{n-1}$. If $\mathit{remove}(v_k)$ is done on s_i, then v_k is removed from V, so v_k must be removed from this expression in order to calculate t_{i+1}. The difficulty is that this must be done through a monotonic update of t_i's metadata. We present a correct but inefficient solution below; we are actively working on more efficient solutions.

$$\begin{aligned} \mathit{fold}'(s_i, \text{op}) &= \text{Op}_{(v,a,r) \in s_i}(\text{Op}_{u \in a} v \text{ op } \text{Op}'_{u \in r} v) \\ \mathit{fold}(s, \text{op}) &= t = [\mathit{fold}'(s_i, f) \mid s_i \in s] \end{aligned} \tag{11}$$

This solution assumes that op is associative, commutative, and has an inverse denoted by op$'$. Note that the elements u of a are not used directly; they serve only to ensure that the operation op(v) is repeated $|a|$ times (and analogously for op$'(v)$ which is repeated $|r|$ times). Since a and r grow monotonically, it is clear that the computation of $\mathit{fold}'(s_i, \text{op})$ also grows monotonically.

[9] This waits for a strict inflation in the lattice, as opposed to an inflation, which triggers when the value does not change.

3.6 Set-Theoretic Functions

We now look at the semantics for the set-theoretic functions that are lifted to operate over CRDTs: **product**, **union**, and **intersection**. We formalize them as follows:

Definition 3.6. The **product** function defines a process that never terminates, which reads elements of the input streams s and u, and creates elements in the output stream t. A new element is created on t for each new element read on either s and u. Metadata composition ensures that if v_s is removed from s or v_u is removed from u, then all pairs containing v_s or v_u are removed from t.[10]

$$\begin{aligned} \mathit{product}'(s_i, u_j) &= \{((v, v'), a \times a', a \times r' \cup r \times a') \\ &\mid (v, a, r) \in s_i, (v', a', r') \in u_j\} \\ \mathit{product}(s, u) &= t = [\mathit{product}'(s_i, u_j) \mid s_i \in s, u_j \in u] \end{aligned} \tag{12}$$

Definition 3.7. The **union** function defines a process that never terminates, which reads elements of the input streams s and u, and creates elements in the output stream t. A new element is created on t for each new element read on either s or u. We combine the metadata for elements that exist in both inputs, similar to the definition of the **map** operation.

$$\begin{aligned} \mathit{un}_1(s_i, u_j) &= \{(v, a, r) \mid (v, a, r) \in s_i \oplus (v, a, r) \in u_j\} \\ \mathit{un}_2(s_i, u_j) &= \{(v, a \cup a', r \cup r') \mid (v, a, r) \in s_i, (v, a', r') \in u_j\} \\ \mathit{union}(s, u) &= t = [\mathit{un}_1(s_i, u_j) \cup \mathit{un}_2(s_i, u_j) \mid s_i \in s, u_j \in u] \end{aligned} \tag{13}$$

Definition 3.8. The **intersection** function defines a process that never terminates, which reads elements of the input streams s and u, and creates elements in the output stream t. A new element is created on t for each new element read on either s and u. We combine the metadata such that only elements that are in both s and u appear in the output.

$$\begin{aligned} \mathit{inter}'(s_i, u_j) &= \{(v, a \times a', a \times r' \cup r \times a') \\ &\mid (v, a, r) \in s_i, (v, a', r') \in u_j\} \\ \mathit{intersection}(s, u) &= t = [\mathit{inter}'(s_i, u_j) \mid s_i \in s, u_j \in u] \end{aligned} \tag{14}$$

4. Fundamental Theorem of Lasp Execution

How easy is programming in Lasp? Can it be as easy as programming in a non-distributed language? Is it possible to ignore the replica-to-replica communication and distribution of CRDTs? Because of the strong semantic properties of CRDTs, it turns out that this is indeed possible. In this section we formalize the distributed execution of a Lasp program and we prove that there is a centralized execution, i.e., a single sequence of states, that produces the same result as the distributed execution. This allows us to use the same reasoning and programming techniques as centralized programs.

The programmer can reason about instances of CRDTs as monotonic data structures linked by monotonic functions, which is a form of deterministic dataflow programming. It has the good properties of functional programming (e.g., confluence and referential transparency) in a concurrent setting.[11]

4.1 Formal Definition of a CRDT Instance

We provide a formal definition of a CRDT instance and its distributed execution. For reasons of clarity, we borrow the notations of the original report on CRDTs [32].

[10] When $r = a$ or $r' = a'$ then $a \times r' \cup r \times a' = a \times a'$, and when $r \subset a$ and $r' \subset a'$ then $a \times r' \cup r \times a' \subset a \times a'$.

[11] See chapter 4 of [34] for a detailed presentation of deterministic dataflow.

Notation for Replication and Method Executions Assume a replicated object with n replicas and one state per replica. We use the notation s_i^k for the state of replica i after k method executions. The vector $(s_0^{k_0}, \cdots, s_{n-1}^{k_{n-1}})$ of the states of all replicas is called the object's *configuration*. A state is computed from the previous state by a method execution, which can be either an update or a merge. We have $s_i^k = s_i^{k-1} \circ f_i^k(a)$ where $f_i^k(a)$ is the k-th method execution at replica i. An update is an external operation on the data structure. A merge is an operation between two replicas that transfers state from one to another. A method execution that is an update is denoted $u_i^k(a)$ (it updates replica i with argument a). A method execution that is a merge is denoted $m_i^k(s_{i'}^{k'})$ (where $i \neq i'$; it merges state $s_{i'}^{k'}$ into replica i).

Definition 4.1. Causal order of method executions Method executions $f_i^k(a)$ have a causal order $\leq_H$ (H for *happens before*) defined by the following three rules:

1. $f_i^k(a) \leq_H f_i^{k'}(a')$ for all $k \leq k'$ (causal order at each replica)
2. $f_{i'}^{k'}(a) \leq_H m_i^k(s_{i'}^{k'})$ (causal order of replica-to-replica merge)
3. $f_i^k(a) \leq_H f_{i'}^{k'}(a')$ if there exists $f_{i''}^{k''}(a'')$ such that $f_i^k(a) \leq_H f_{i''}^{k''}(a'')$ and $f_{i''}^{k''}(a'') \leq_H f_{i'}^{k'}(a')$ (transitivity)

Definition 4.2. Delivery Using causal order we define the concept of delivery: an update $u_i^k(a)$ is *delivered* to a replica i at state $s_{i'}^{k'}$ if $u_i^k(a) \leq_H f_{i'}^{k'}(a)$.

Definition 4.3. State-based CRDT A CRDT is a replicated object that satisfies the following conditions:

- **Basic structure:** It consists of n replicas where each replica has an initial state, a current state, and two methods query and update that each executes at a single replica.
- **Eventual delivery:** An update delivered at some correct replica is eventually delivered at all correct replicas.
- **Termination:** All method executions terminate.
- **Strong Eventual Consistency (SEC):** All correct replicas that have delivered the same updates have equal state.

This definition is slightly more general than the definition of report [32]. In that report, an additional condition is added: that each replica will always eventually send its state to each other replica, where it is merged using a join operation. We consider that this condition is too strong, since there are many ways to ensure that state is disseminated among the replicas so that eventual delivery and strong eventual consistency are guaranteed. In its place, we assume a weak synchronization model, Property 4.2, that is not part of the CRDT definition, and we allow each CRDT to send the merge messages it requires to satisfy the CRDT properties.

Theorem 4.1. Monotonic semilattice condition for CRDTs *A replicated object is a* ***state-based CRDT instance*** *(in short, a CRDT instance), if the following three conditions hold:*

1. *The set of possible values of a state s_i^k forms a semilattice ordered by $\sqsubseteq$.*
2. *Merging state s with state s' computes the Least Upper Bound (join) of the two states $s \circ s'$.*
3. *The state is monotonically non-decreasing across updates: $s \sqsubseteq s \circ u$ for any update u.*

We say that any CRDT instance satisfying this theorem is a ***monotonic semilattice object****.*

Proof Proof is given in [32].

Definition 4.4. SEC state From the commutativity and associativity of the join operator $\circ$, it follows that for any execution of a monotonic semilattice object, if updates $U = \{u_0, ..., u_{n-1}\}$ are all delivered in state s, then $(u_0 \circ u_1 \circ \cdots \circ u_{n-1}) \sqsubseteq s$, that is, s is an inflation of the join of all updates in U. It is not necessarily equal since other updates may have occurred during the execution. We call $(u_0 \circ u_1 \circ \cdots \circ u_{n-1})$ the *SEC state* of updates U.

4.2 Formal Definition of a Lasp Process

We provide a formal definition of a Lasp process.

Definition 4.5. Monotonic m-ary function Given an m-ary function f between states such that $s = f(s_0, s_1, \cdots, s_{m-1})$. Then f is a *monotonic function* if $\forall i : \; s_i \sqsubseteq s_i' \Rightarrow f(\cdots, s_i, \cdots) \sqsubseteq f(\cdots, s_i', \cdots)$.

Definition 4.6. Lasp process A *Lasp process* is a pair of a sequence of m CRDT instances and one monotonic m-ary function f, written as $([C_0, \cdots, C_{m-1}], f)$. The process defines its output as n states where each state is the result of applying f on the corresponding replicas of the input CRDTs.

4.3 System Properties

The following properties are needed to prove the fundamental theorem.

Property 4.1. Fault model and repair We assume the following three conditions:

- **Crash-stop failures:** replicas fail by crashing and any replica may fail at any time.
- **Anti-entropy:** after every crash, a fresh replica is eventually created with state copied from any correct replica.
- **Correctness:** at least one replica is correct at any instant.

The first condition is imposed by the environment. The second condition is the repair action done by every CRDT when one of its replicas crashes. The third condition is what must hold globally for the CRDT to continue operating correctly.

Property 4.2. Weak synchronization For any execution of a CRDT instance, it is always true that eventually every replica will successfully send a message to each other replica.[12]

Property 4.3. Determinism Given two executions of a CRDT instance with the same sequence of updates but a different merge schedule, i.e., a different sequence of replica-to-replica communication, replicas in the first execution that have delivered the same updates as replicas in the second execution have equal state.

Since we intend Lasp programming to be similar to functional programming, it is important that computations are deterministic. We remark that SEC by itself is not enough to guarantee that; we provide a simple counterexample with the OR-Set:

Assume that replica A (R_A) does an $add(1)$ followed by a $remove(1)$ and replica B (R_B) does an $add(1)$. When all replicas have delivered these three updates, the state of the OR-Set will either contain 1 or not contain 1. It will not contain 1 if the second $add(1)$ is in the causal history of the $remove(1)$. In the other case, it will contain 1. Both situations are possible depending on whether or not the merge schedule communicates the state of replica B to replica A after its $add(1)$ and before the $remove(1)$. Figure 5 illustrates this scenario.

Therefore, to correctly use an OR-Set in Lasp, it is important to impose conditions that ensure determinism. The following two conditions are sufficient to guarantee determinism for all merge schedules:

[12] The content of this message depends on the definition of the CRDT.

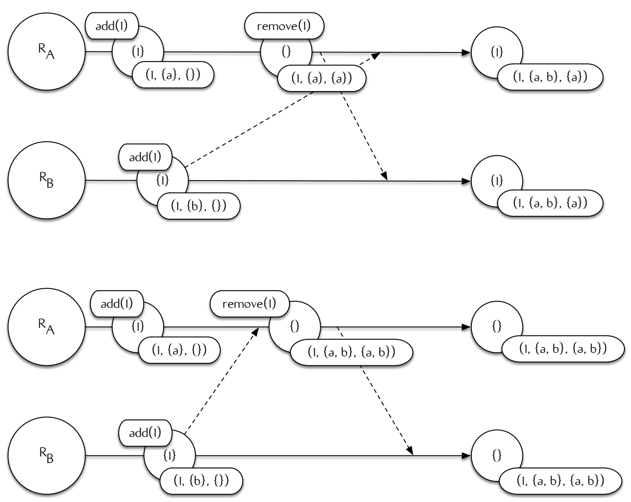


Figure 5: *Example of nondeterminism introduced by different replica-to-replica merge schedules. In the top example, merging after the remove results in the item remaining in the set, where merging before the remove results in the item being removed.*

- A $remove(v)$ is only allowed if an $add(v)$ with the same value has been done previously at the same replica.
- An $add(v)$ with the same value of v may not be done at two different replicas.

4.4 Lemmas

Lemma 4.2. Eventual delivery for faulty execution *Each update in a CRDT instance execution that satisfies Property 4.1 and Property 4.2, and for which the messages in Property 4.2 are delivered according to a continuous probability distribution is eventually delivered at all replicas or at no replicas, with probability 1.*

Proof According to Property 4.1, a replica may crash and be replaced by a new replica with state copied from any live replica. In any configuration of a CRDT execution, there will be $m \leq n$ live replicas of which $m' \leq m$ have delivered the update. Initially when the update is done, $m' = 1$. Crash of a replica that has delivered the update will decrease m'. Replica-to-replica communication will increase m' if done from a replica that has delivered the update to a replica that has not. Otherwise it will not affect m'. As the CRDT instance continues its execution, Property 4.1 implies that one of two situations will eventually happen: either all live replicas deliver the update, or no live replicas deliver the update. Once one of these situations happens, the third condition of Property 4.1 ensures it will continue indefinitely.

The continuous probability distribution ensures that all infinite non-converging executions have probability zero. For example, given three replicas R_A, R_B, R_C, and only R_C has delivered. It is possible that R_A crashes just after R_C delivers to it, followed by a new replica $R_{A'}$ created from R_B. This can repeat indefinitely while satisfying Property 4.1 and Property 4.2. With a continuous probability distribution, each repetition multiplies the probability by a number less than 1, so the infinite execution has probability zero.

Definition 4.7. Compatibility Given a CRDT instance execution and a finite set U of updates in this execution. We say that a state is *compatible* with U in the CRDT execution if it consists of the join of all updates in U inflated with any subset of the other updates occurring before the state.

Compatibility makes precise the notion that all replicas reach the same state if no other updates occur (SEC) but that other updates might occur in the meantime. All the replica states are not necessarily the same, but they are all inflations of the SEC state.

Lemma 4.3. Reduction of CRDT execution to a single state execution *For any CRDT instance execution, there exists a single state execution such that any finite set U of updates from the CRDT execution is eventually delivered to the single state execution and gives a state that is compatible with U in the CRDT execution.*

Proof Define a single state execution whose updates are a topological sort of the updates in the CRDT execution that respects the causal order $\leq_H$ of these updates. The resulting execution satisfies all four properties of Definition 4.3. In particular, for eventual delivery, it is clear that the single state execution eventually delivers all updates in U. All other updates occurring before this state are either causally before or concurrent with an update in U.

Lemma 4.4. Reduction of Lasp process to a CRDT execution *A Lasp process behaves as if it were a single CRDT instance with n replicas. Each replica state consists of an $(m+1)$-vector of the m states of the input CRDT instances and the corresponding state of the output as defined by f applied to the m input states.*

Proof The execution of the Lasp process satisfies all four properties of Definition 4.3. In particular, for strong eventual consistency is clear that each CRDT instance will eventually deliver its updates to all its replicas, resulting in a state compatible with these updates. When this happens for all CRDT instances, then all n replicas of the output state will be equal.

4.5 Fundamental Theorem

We present the fundamental theorem of Lasp.

Definition 4.8. Simple Lasp program A simple Lasp program consists of either:

- A single CRDT instance, or
- A Lasp process with m inputs that are simple Lasp programs and one output CRDT instance.

Theorem 4.5. *A simple Lasp program can be reduced to a single state execution.*

Proof This is straightforward to prove by induction. We construct the program in steps starting from single CRDTs. By Lemma 4.3, a single CRDT instance can be reduced to a single state execution. For each Lasp process, we replace it by a single CRDT instance whose updates are the updates of all its input CRDT instances. By Lemma 4.4, this is correct. We continue until we have constructed the whole program. By Lemma 4.2, if there are faults then the worst that can happen is that some updates are ignored.

5. Implementation

Our prototype of Lasp is implemented as an Erlang library. We leverage the *riak_dt* [2] library from Basho Technologies, Inc., which provides an implementation of state-based CRDTs in Erlang.

5.1 Distribution

Lasp distributes data using the Riak Core distributed systems framework [22], which is based on the Dynamo system [18].

Riak Core The Riak Core library provides a framework for building applications in the style of the original Dynamo system. Riak Core provides library functions for cluster management, dynamic membership and failure detection.

Dynamo-style Partitioning and Hashing Lasp uses Dynamo-style partitioning of CRDTs: consistent hashing and hash-space partitioning are used to distribute copies of CRDTs across nodes in a cluster to ensure high availability and fault tolerance. Replication of each CRDT is performed between adjacent nodes in a cluster. While the partitioning mechanism and implementation is nuanced, it is sufficient to realize the collection of CRDTs as a series of disjoint replica sets, of which the data is sharded across, with full replication between the nodes in any given replica set.

Anti-Entropy Protocol We provide an active anti-entropy protocol built on top of Riak Core that is responsible for ensuring all replicas are up-to-date. Periodically, a process is used to notify replicas that contain CRDT replicas with the value of a CRDT from a neighboring replica.[13]

Quorum System Operations In Section 4, we outline the three properties of our system: crash-stop failures, anti-entropy, and correctness. While these properties are sufficient to ensure confluence of computations, they do not guarantee that all updates will be observed if a given replica of a CRDT fails before communicating its state to a peer replica. Therefore, to guarantee safety and be tolerant to failures, both read and update operations are performed against a quorum of replicas. This ensures fault tolerance: by performing read and write operations against a majority, the system is tolerant to failures. The system remains safe and does not make progress when the majority is not available. Additionally, quorum operations can be used to increase liveness in the system: by writing back the merged value of the majority, we can passively repair objects during normal system operation, improving anti-entropy.[14]

Replication and Execution of Operations Given replication of the objects themselves, to ensure fault-tolerance and high-availability, our functional programming operations and set-theoretic operations must be replicated as well. To achieve this, quorum replication is used to contact a majority of replicas near the output CRDT, which are responsible for reading the input CRDT and performing the transformation.

Given the **map** example in Figure 4, we spawn processes at a majority of the output CRDT replicas, *S2*, which read from the input replicas of *S1*.

To ensure forward progress of these computations, each of our operations uses the strict version of the *monotonic read* operation to prevent from executing over stale values when talking to replicas which are out-of-date. In the **map** example, the transformation is performed for a given observation in the stream of updates to variable *S1* with the output written into the stream for variable *S2*, at which the process tail-recursively executes and wait to observe a causally greater value than the previously observed *S1* before proceeding. This prevents duplication of already computed work and ensure forward progress at each replica.

Additionally, we can apply read repair and anti-entropy techniques to repair the value of *S2* if it falls very far behind instead of relying on applying operations from *S1* in order.

6. Evaluation

In this section, we look at two applications that can be implemented with Lasp.

[13] We plan to design an optimized version, similar to the Merkle tree based approach in [18]; our current protocol is sufficient to ensure progress.

[14] In [18], this process is referred to as read repair.

6.1 Advertisement Counter

One of the use cases for our model is supporting clients that need to operate without connectivity. For example, imagine a provider of mobile games that sells advertisement space within their games.

In this example, the correctness criteria are twofold:

- Clients will go offline: consider mobile devices such as cellular phones that experience periods without connectivity. When the client is offline, advertisements should still be displayable.
- Advertisements need to be displayed a minimum number of times. Additional impressions are not problematic.

Figure 6 presents one design for an eventually consistent advertisement counter written in Lasp. In this example, squares represent primitive CRDTs and circles represent CRDTs that are maintained using Lasp operations. Additionally, Lasp operations are represented as diamonds and edges represent the monotonic flow of information.

Our advertisement counter operates as follows:

- Advertisement counters are grouped by vendor.
- All advertisement groups are combined into one list of advertisements using a **union** operation.
- Advertisements are joined with active "contracts" into a list of displayable advertisements using both the **product** and **filter** operations.
- Each client selects an advertisement to display from the list of active advertisements.
- For each advertisement displayed, each client updates its local copy of the advertisement counter.
- Periodically, advertisement counters are merged upstream.
- When a counter hits at least 50,000 advertisement impressions, the advertisement is "disabled" by removing it from the list of advertisements.

The implementation of this advertisement counter is completely monotonic and synchronization-free. Adding and removing ads, adding and removing contracts, and disabling ads when their contractual number of views is achieved are all modeled as the monotonic growth of state in CRDTs connected by active processes. Programmer-visible nonmonotonicity is represented by monotonic metadata in the CRDTs.

The full implementation of the advertisement counter is available in the Lasp source code repository and consists of 213 LOC. In this example, transparent distribution and failure handling is supported by the runtime environment, and not exposed to the developer. For brevity, we provide only two code samples: the advertisement counter "server" process, that is responsible for disable advertisements when their threshold is reached, and example use of the **product** and **filter** operations used for composing the advertisements with their contracts.

Figure 7 provides an example of the advertisement counters server process: this process is responsible for performing a blocking read on each counter that will disable the counter by removing it from the set once the threshold is reached. One server is launched per counter to manage its lifecycle.

Figure 8 provides an example of the advertisement counters dataflow: both of the **product** and **filter** operations spawn processes that continuously compute the composition of both the set of advertisements and the set of counters as each data structure independently evolves.

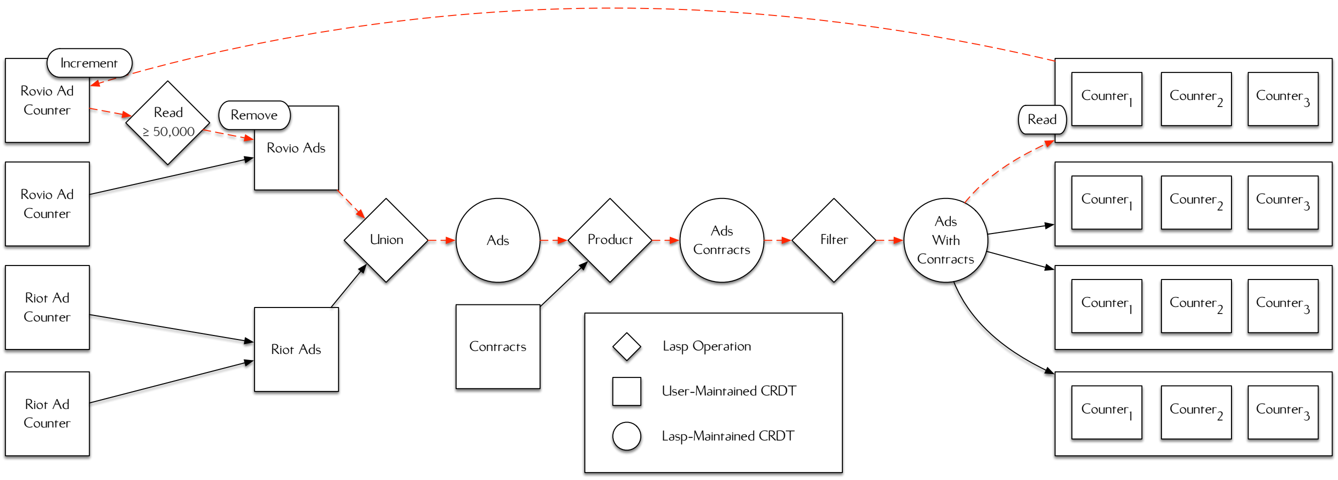


Figure 6: *Eventually consistent advertisement counter. The dotted line represents the monotonic flow of information for one counter.*

```
%% @doc Server for the advertisement counter.
server({#ad{counter=Counter}=Ad, _}, Ads) ->
    %% Blocking monotonic read for 50,000
    {ok, _} = lasp:read(Counter, 50000),

    %% Remove the advertisement.
    {ok, _} = lasp:update(Ads, {remove, Ad}, Ad),

    lager:info("Removing ad: ~p", [Ad]).
```

Figure 7: *Example use of the monotonic read operation in the advertisement counter application. A process is spawned that blocks until the advertisement counter reaches 50,000 impressions, after which it removes itself from the list of advertisements.*

```
    %% Compute the Cartesian product of both
    %% ads and contracts.
    {ok, AdsContracts} = lasp:declare(?SET),
    ok = lasp:product(Ads, Contracts, AdsContracts),

    %% Filter items by join.
    {ok, AdsWithContracts} = lasp:declare(?SET),
    FilterFun = fun({#ad{id=Id1},
                   #contract{id=Id2}}) ->
        Id1 =:= Id2
    end,
    ok = lasp:filter(AdsContracts,
                   FilterFun,
                   AdsWithContracts),
```

Figure 8: *Example use of dataflow operations in the advertisement counter application. These operations together compute a join between a set of advertisements and a set of counters to compute a list of displayable advertisements.*

6.2 Bloom^L Replicated Key-Value Store

We provide an example of a replicated key-value store (KVS) similar to the key-value store example presented by Conway et al. [15]. In this example, we show how our model supports writing this replica in a easy to reason about functional manner.

Our key-value store is a simple recursive function that receives three types of messages from clients: *get*, *put*, and *remove*.

- *get:* Retrieve a value from the KVS by name.
- *put:* Store a value in the KVS by name, computing the *join* of the new value and the current value.
- *remove:* Remove observed values in the KVS by key.

Figure 9 contains the code for a single server replica. A *riak_dt_map*, a composable, convergent map, [13] is used for modeling the store. Given this data structure supports the composition of state-based CRDTs, we assume the values for all keys will be mergeable given the lattice defined by the data type stored.

```
receiver(Map, ReplicaId) ->
    receive
        {get, Key, Client} ->
            {ok, {_, MapValue0, _}} = lasp:read(Map),
            MapValue = riak_dt_map:value(MapValue0),
            case orddict:find(Key, MapValue) of
                error ->
                    Client ! {ok, not_found};
                Value ->
                    Client ! {ok, Value}
            end,
            receiver(Map, ReplicaId);
        {put, Key, Value, Client} ->
            {ok, _} = lasp:update(Map,
                                  {update,
                                   [{update, Key,
                                    {add, Value}}]},
                                  ReplicaId),
            Client ! ok,
            receiver(Map, ReplicaId);
        {remove, Key, Client} ->
            {ok, _} = lasp:update(Map,
                                  {update,
                                   [{remove, Key}]},
                                  ReplicaId),
            Client ! ok,
            receiver(Map, ReplicaId)
    end.
```

Figure 9: *Simple replicated key-value store in Lasp. This tail-recursive process is responsible for receiving messages from client processes, and processes them in serial order.*

In our example, we use a simple recursive process for modeling the key-value store. This process is responsible for responding to both *get* and *put* messages: when a message is received the appropriate action is performed on the given key. When a *put* message arrives, the map is updated by performing two actions: first, merging the current value with the provided value in the map, second, merging the updated map back into the variable store with the new map. This operation is done atomically by Lasp using the *update* operation. When a *get* message arrives, we return the current value from the map for the provided key. Multiple instances of the replicated KVS can merge state by periodic exchange of their maps.

We improve on the Bloom^L KVS by supporting concurrent removal operations: removals observed at a replica remove all observed values for a key while concurrent additions for the same key win against concurrent removals. Lasp's programming model removes the restrictions placed on lattices having external monotonic behavior by using CRDTs as the primary programming abstraction, while additionally providing a familiar functional programming semantics to simplify distributed programming.

7. Related Work

In the following section, we identify related work.

7.1 Distributed Oz

Distributed Oz [19, 34] provides an extension of the Oz programming model allowing for asynchronous communication and mobile processing. Distributed Oz formalizes this by extending the Oz centralized execution semantics with semantics for distributed execution. Distributed Oz has a functional core that performs distributed unification over rational trees [35]. For unifications without conflicting bindings, this satisfies the definition of a CRDT.

Lasp's use of CRDTs solves the problem of conflicting bindings: for each type of CRDT, there is always a merge function that can resolve concurrent operations in a deterministic manner. In addition, Lasp provides deterministic dataflow over general CRDTs, whereas Distributed Oz provides deterministic dataflow over just one CRDT, namely rational trees. Finally, Lasp uses metadata computation to support nonmonotonic operations in a functional setting.

7.2 FlowPools

FlowPools [30] provide a lock-free deterministic concurrent dataflow abstraction for the Scala programming language. FlowPools are essentially a lock-free collection abstraction that support a concurrent append operation and a set of combinators and higher-order operations. FlowPools are designed for multi-threaded computation, not distributed computation.

While higher-order operations such as **foreach** and **aggregate** function similarly to the **map** and **fold** operations in Lasp, namely they execute once for each element that will eventually exist in the FlowPool, these operations are somewhat limited. Each FlowPool can only be appended to, and each element is single-assignment. Computations using the **aggregate** operation require that the FlowPool be sealed before the result of the aggregation is realized.

7.3 Derflow and Derflow$_L$

Derflow and Derflow$_L$ are direct precursors to Lasp. Derflow [12] defines a fault-tolerant single-assignment data store. It implements deterministic dataflow programming [34] on the Dynamo-inspired, Riak Core distributed systems framework. [18, 22]

Derflow$_L$ [27] extends Derflow to join-semilattices. Derflow$_L$ relies on user-specified composition of CRDTs. While this model is sufficient for composition of less complex CRDTs, it fails to scale to the more complex and efficient CRDTs since it requires the programmer to explicitly handle the composition of metadata.

7.4 Bloom^L

Bloom^L [15] provides Datalog-style operations over monotonically growing lattices in a distributed environment. Applications in Bloom^L can be analyzed to identify locations where nonmonotonic operations occur, where coordination can be used to enforce order. Differences in the programming abstraction notwithstanding, we highlight two differences between Bloom^L and Lasp:

Retraction of Information Retraction of information in Bloom^L is nonmonotonic, and therefore not confluent. By using composition of OR-Sets, Lasp can offer an eventually consistent (monotonic and confluent) mechanism for the retraction of information, but can not guarantee when the update might be visible.

Sealing Lattices used by the Bloom^L system lack causal information, which places the requirement on monotone functions to, once satisfied, freeze, or seal, their values. [7]

For instance, consider the case of a monotonic mapping between two booleans, a to b: once a becomes true, b becomes true. Once the condition is met in a, and b is set to true; the property is considered "satisfied" and can no longer become "unsatisfied". This prevents the situation where an earlier version of an update is delivered to the system and prevents the condition from observing nonmonotonic behavior. Lasp can detect these scenarios using metadata in the form of logical clocks which can be tracked through morphisms, preventing an earlier update from causing the regression of the value.

7.5 LVars

LVars [24] formalizes lattice variables for use in parallel computations in single machine settings that enforce determinism. While LVars shares a similar functional programming core with Lasp, each system differs in its distribution and failure modes given they were designed to solve different problems. We also believe the threshold read operation formalized in the LVars work is insufficient for use with advanced types of CRDTs.

We discuss both of these issues below:

Differences in Design: LVars vs. Lasp Focusing on single machine computations, LVars is aimed at running computations in parallel, over shared state, while preserving determinism in an otherwise functionally pure application. Focusing on distributed computations, Lasp is aimed at running fault-tolerant applications, which are designed to diverge and later converge deterministically, given periods of time where processes may not be able to communicate.

Threshold Reads vs. Monotonic Reads The *threshold read* operation, both originally formalized over lattices and later reformalized over state-based CRDTs[15] [23] by Kuper and Newton, makes two assumptions: *a priori* knowledge of the internal state of a CRDT to properly threshold on the value, and that the queryable value of a CRDT is monotone.

For example, the Grow-Only Set CRDT (G-Set) observes both of these properties: when writing a deterministic computation over a known stream, you are able to satisfy both conditions. First, the internal representation of the CRDT requires only storage of the set structure itself; concurrent operations can potentially add the same element, however, the *join* operation between two sets will remove duplicates and advance the data structure in the partial order. Second, the value of the set will always be increasing and therefore is monotone.

However, when dealing with a design like the Observed-Remove Set CRDT (OR-Set), which allows the repeated addition

[15] The citation refers to these as "CvRDTs", convergent replicated data types, a legacy name for the more recent nomenclature, "state-based."

and removal of arbitrary elements, individual operations on the data structure must be uniquely identified for correct semantics, even if they represent the concurrent addition of the same element at two different replicas. This prevents the first property from being fulfilled: *a priori* knowledge of the unique constant identifiers given multiple executions of the same program under different interleavings is not possible. The second property cannot be fulfilled either: the Observed-Remove Set has a nonmonotonic query function because elements can be removed.

Lasp is designed to specifically address the two previously discussed problems: the problem of *threshold reads* given CRDTs with nondeterministic internal state, and the problem of properly composing these data types, while preserving the internal knowledge required for correct convergence.

7.6 Discretized Streams

Discretized Streams (D-Streams) [37] is a programming model for stream processing that supports efficient parallel recovery of faults. D-Streams realize infinite streams as a series of small immutable batches over which deterministic computations can be done, as typically seen in the MapReduce model. [17] The major contribution of this work is efficient parallel recovery during faults; instead of replicating the computations of the streams or using upstream backup, [8, 20] lost computations can be recomputed in parallel.

The model exploits the immutable nature of the individual events in infinite streams; D-Streams assume that a batch is considered sealed at a given time and events are grouped into batches based on when the event arrives at the ingestion point. On the other hand, Lasp assumes that individual data structures, along with compositions of these data structures, will monotonically evolve over time while preserving determinism.

7.7 Summingbird

Summingbird [11] is an open-source domain specific language for integrating online and batch computations into a single programming abstraction. Summingbird can be used to build complex DAG workflows, where processing is performed between sources and sinks, with the additional ability to persist both partial and final results to a data store such as MySQL or HBase.

In enabling correct, efficient aggregation of computations, operations in the "reduce" phase are restricted to commutative semigroups. This prevents incorrect operation in the event of network or processing anomalies such as out-of-order message delivery.

Lasp's primary programming abstraction is the state-based CRDT: a convergent data structure formalized with a bounded join-semilattice. Given a semilattice is a commutative idempotent semigroup, and a bounded join-semilattice forms a commutative idempotent monoid which induces a partial order using the join operation, this allows Lasp to handle both the network anomalies of duplicated and reordered messages, as well as to reason about the ordering of updates to a given item.

8. Current and Future Work

In the following section, we identify current and future work.

8.1 General Concepts

Currently, Lasp is a first-order model that allows defining data structures and operations performed on them. Future extensions will add abstraction mechanisms and other concepts, as they are needed by the application scenarios that we intend to implement. The concepts will be designed according to the needs of expressiveness and efficiency explained in the following two sections.

8.2 Invariant Preservation

Some computations require the preservation of invariants between sets of replicated CRDTs. One such example is the students and teams example posed by Conway et al. [15] when discussing the "scope problem" of CRDTs. In this scenario, removing a student from the set of active students should also remove the student from any teams they were participating in.

We envision a way to specify these invariants between CRDTs as contracts: these contracts would be enforced by a mechanism at runtime given an allowed amount of divergence. We look at two examples of where contracts would be useful:

- In the advertisement counter example (Figure 6), clients can locally increment their counter, and either synchronize with the server side advertisement counter for a given advertisement after every impression or after a given number of impressions. How often a client chooses to synchronize is a measure of how much we allow this counter to diverge.
- In the students and teams example, we may want to enforce the invariant locally at each replica, but allow the system to temporarily diverge to reduce the amount of synchronization.

We believe we can leverage recent work in contract enforcement and invariant preservation in eventually consistent systems, specifically "invariant-based programming" by Balegas et al. [9] and Quelea by Kaki et al. [21].

8.3 Optimizations

We plan to explore optimizing the Lasp programming model through the use of metadata reduction, reduced state propagation, and intermediate tree elimination.

Metadata Reduction Lasp currently has limited support for the Optimized Conflict-Free Replicated Set (ORSWOT) as described by Bieniusa et al. [10]. This set contains a novel algorithm for avoiding the requirement of tracking tombstones (i.e., it does not need garbage collection of metadata). This makes it an ideal data structure for use in production systems. This data structure is also the basis for the convergent, conflict-free replicated map, as described by Brown et al. [13].

Reduced State Propagation We would like to explore methods for reducing the amount of computation needed to propagate state changes through the graph. Two such approaches for this are Operation-based CRDTs [33], which propagate commutative operations through a reliable channel instead of the full state, and $\delta-$state CRDTs [5], which propagate minimal state representing the delta derived by applying the operation locally.

Distributed Intermediate Tree Elimination Computations in Lasp are formed using a very small subset of a functional language: this results in very large tree structures, where the intermediate computations might not be necessary. This is a side effect of the functional programming style. We imagine that techniques such as Wadler's "deforestation" can be used to eliminate these structures in a distributed fashion for more efficient computation, resulting in less network communication. [36]

9. Conclusions

We introduced the Lasp programming model and motivated its use for large-scale computation over replicated data. Our future plans for Lasp include extending it to become a full-fledged language and system, identifying optimizations for more efficient state propagation, exploring stronger consistency models, and optimizing distribution and replica placement for better fault tolerance and reduced latency. We also plan to evaluate the Lasp system and to test our hypothesis that Lasp's weak synchronization model is well-suited for scalable and high-performance applications, in particular in settings with intermittent connectivity such as mobile applications and "Internet of Things". Our ultimate goal is for Lasp to become a general purpose language for building large-scale distributed applications in which synchronization is used as little as possible.

Acknowledgments

Thanks to Peter Alvaro, Sean Cribbs, Scott Lystig Fritchie, Seyed (Hossein) Haeri, Joel Knighton, John Muellerleile, Andrew Stone, and our anonymous reviewers for their feedback. The research leading to these results has been partially funded by the SyncFree Project [4] in the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n^o 609551.

References

[1] Lasp source code repository. https://github.com/lasp-lang/lasp. Accessed: 2015-06-14.

[2] Riak DT source code repository. https://github.com/basho/riak_dt. Accessed: 2015-03-30.

[3] 263 Million Monthly Active Users In December. http://www.rovio.com/en/news/blog/261/263-million-monthly-active-users-in-december/. Accessed: 2015-02-13.

[4] SyncFree: Large-scale computation without synchronisation. https://syncfree.lip6.fr. Accessed: 2015-02-13.

[5] P. S. Almeida, A. Shoker, and C. Baquero. Efficient State-based CRDTs by Delta-Mutation. *arXiv preprint arXiv:1410.2803*, 2014.

[6] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein. Consistency without borders. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 23. ACM, 2013.

[7] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed programs. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 52–63. IEEE, 2014.

[8] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)*, 33(1):3, 2008.

[9] V. Balegas, M. Najafzadeh, S. Duarte, M. Shapiro, R. Rodrigo, and N. Preguiça. Putting Consistency Back into Eventual Consistency. *Submitted to EuroSys 2015*.

[10] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. *arXiv preprint arXiv:1210.3368*, 2012.

[11] O. Boykin, S. Ritchie, I. OConnell, and J. Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *Proceedings of the VLDB Endowment*, 7(13), 2014.

[12] M. Bravo, Z. Li, P. Van Roy, and C. Meiklejohn. Derflow: distributed deterministic dataflow programming for Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 51–60. ACM, 2014.

[13] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott. Riak DT map: a composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, page 1. ACM, 2014.

[14] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[15] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM, 2012.

[16] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 2002.

[17] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[18] 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 *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[19] S. Haridi, P. Van Roy, and G. Smolka. An overview of the design of Distributed Oz. In *Proceedings of the second international symposium on parallel symbolic computation*, pages 176–187. ACM, 1997.

[20] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 779–790. IEEE, 2005.

[21] G. Kaki, K. Sivaramakrishnan, and S. Jagannathan. Declarative programming over eventually consistent data stores. *To appear at PLDI'15*.

[22] R. Klophaus. Riak core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.

[23] L. Kuper and R. R. Newton. Joining forces. *Draft*.

[24] L. Kuper and R. R. Newton. LVars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84. ACM, 2013.

[25] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[26] C. 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*, page 13. ACM, 2014.

[27] C. Meiklejohn. Eventual Consistency and Deterministic Dataflow Programming. *8th Workshop on Large-Scale Distributed Systems and Middleware*, 2014.

[28] C. Meiklejohn and P. Van Roy. Lasp: a language for distributed, eventually consistent computations with CRDTs. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, page 7. ACM, 2015.

[29] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.

[30] A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. Flowpools: A lock-free deterministic concurrent dataflow abstraction. In *Languages and Compilers for Parallel Computing*, pages 158–173. Springer, 2013.

[31] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, 2005.

[32] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. Technical Report RR-7687, INRIA, 07 2011.

[33] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report RR-7506, INRIA, 01 2011.

[34] P. Van Roy and S. Haridi. *Concepts, techniques, and models of computer programming*. MIT Press, 2004.

[35] P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3): 569–626, 1999.

[36] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP'88*, pages 344–358. Springer, 1988.

[37] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

C The Implementation and Use of a Generic Dataflow Behaviour in Erlang

The Implementation and Use of a Generic Dataflow Behaviour in Erlang

Christopher Meiklejohn
Basho Technologies, Inc.
Bellevue, WA, United States
cmeiklejohn@basho.com

Peter Van Roy
Université catholique de Louvain
Louvain-la-Neuve, Belgium
peter.vanroy@uclouvain.be

Abstract

We propose a new "generic" abstraction for Erlang/OTP that aids in the implementation of dataflow programming languages and models on the Erlang VM. This abstraction simplifies the implementation of "processing elements" in dataflow languages by providing a simple callback interface in the style of the `gen_server` and `gen_fsm` abstractions. We motivate the use of this new abstraction by examining the implementation of a distributed dataflow programming variant called Lasp.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; E.1 [*Data Structures*]: Distributed data structures

Keywords Dataflow Programming, Erlang, Concurrent Programming

1. Introduction

The dream of dataflow programming [18, 19] is to simplify the act of writing declarative applications that can easily be parallelisable but do not introduce any accidental nondeterminism. Not only does dataflow programming alleviate the need for the developer to reason about the difficulties in concurrent programming: shared memory, thread-safety, reentrancy, and mutexes; dataflow programming provides a declarative syntax focused around data and control flow. By design, dataflow programs lend themselves well to analysis and optimization, preventing the developer from having to explicitly handle parallelism in a safe and correct manner.

While the power of Erlang/OTP is in its highly concurrent, shared-nothing actor system, this only increases the potential concurrency in the system making it difficult to prevent the introduction of accidental nondeterminism in computations. Erlang provides a solution for this problem with the Open Telecom Platform (OTP) "generic" abstractions.[1] These abstractions aim to simplify concurrent programming: developers author code that adheres to a specific "behaviour"[2] and these abstractions then provide a common way to supervise, manage, and reason about processing concurrent messages to a single actor.

We propose a new Erlang/OTP "generic" abstraction for dataflow programming on the Erlang VM called `gen_flow`. This abstraction allows for the arbitrary composition of stateful actors into larger dataflow computations. This abstraction is sufficiently generic to aid in the implementation of an arbitrary dataflow language in Erlang/OTP: we motivate the use of `gen_flow` through the implementation of a deterministic distributed dataflow variant called Lasp. [14, 15]

This paper contains the following contributions:

- **`gen_flow` abstraction:** We propose `gen_flow`, a new abstraction in the style of the "generic" abstractions provided by Erlang/OTP for building dataflow "processing elements."
- **Lasp integration:** We motivate the use of this new abstraction in building the Erlang-based, Lasp programmming model for distributed computing.

2. The `gen_flow` Abstraction

We first discuss an example of how the `gen_flow` abstraction can be used to build a dataflow application and then discuss its implementation.

2.1 Overview

We propose `gen_flow`, a new abstraction for representing "processing elements" in dataflow programming. This abstraction is presented in Erlang/OTP as a behaviour module with a well defined interface and set of callback functions, similar to the existing `gen_server` and `gen_fsm` abstractions provided by Erlang/OTP.

Consider the example in Figure 1. In this example, our processing graph contains three data nodes: two input sets, and one output set, and one function that is computing the intersection of the two sets. When either of the input sets change independently, the output set should be modified to reflect the change in the input.

The goal of this abstraction is to enable the trivial composition of dataflow components to build larger dataflow applications on the Erlang VM. Figure 2 provides the Erlang code to implement the design in Figure 1. The abstraction focuses around two major components: a function that will be responsible for performing the arbitrary computation given some inputs, and a list of anonymous functions that are used to derive the value of the inputs.

[1] When we refer to the Erlang/OTP "generic" abstractions we are referring to the set of behaviours provided: `gen_server`, `gen_fsm`, `gen_event` and `supervisor`.

[2] Behaviours are essentially interfaces specifying callback functions that a module must implement.

Erlang'15, September 04 2015, Vancouver, BC, Canada.
Copyright © 2015 ACM 978-1-4503-3805-9/15/09…$15.00.
http://dx.doi.org/10.1145/2804295.2804300

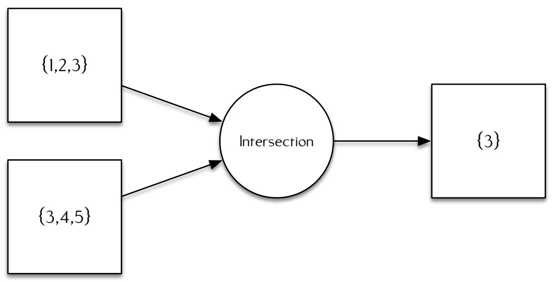


Figure 1: Dataflow example of computing the intersection of two sets. As the input sets change independently, the output is updated to reflect the change.

```erlang
-module(gen_flow_example).
-behaviour(gen_flow).

-export([start_link/1]).
-export([init/1, read/1, process/2]).

-record(state, {pid}).

start_link(Args) ->
    gen_flow:start_link(?MODULE, Args).

init([Pid]) ->
    {ok, #state{pid=Pid}}.

read(State) ->
    ReadFuns = [
        fun(_) -> sets:from_list([1,2,3]) end,
        fun(_) -> sets:from_list([3,4,5]) end
    ],
    {ok, ReadFuns, State}.

process(Args, #state{pid=Pid}=State) ->
    case Args of
        [undefined, _] ->
            ok;
        [_, undefined] ->
            ok;
        [X, Y] ->
            Set = sets:intersection(X, Y),
            Pid ! {ok, sets:to_list(Set)},
            ok
    end,
    {ok, State}.
```

Figure 2: Example use of the `gen_flow` behaviour. This module is initialized with a process identifier in the `init` function, spawns two functions to read the inputs to the `process` function, via the `read` function. The `process` function sends a message to the pid once both `ReadFuns` have returned an initial value.

2.2 Behaviour

The `gen_flow` behaviour requires three callback functions (as depicted in Figure 2):

- `Module:init/1`: Initializes and returns state.
- `Module:read/1`: Function defining how to issue requests to read inputs; takes the current state and returns a list of `ReadFuns` along with an updated state. `ReadFuns` should be arity 1 functions that take the previously read, or cached, value.
- `Module:process/2`: Function defining how to process a request; takes the current state and an argument list of the values read, and returns the new state.

In our example, `Module:init/1` is used to initialize state local to the process: this state should be used in the same fashion that the local state is used by `gen_server` and `gen_fsm`. Here, the local state is used to track the process identifier that should receive the result of the dataflow computation. `Module:init/1` is triggered once at the start of the process.

`Module:read/1` is responsible for returning a list of `ReadFuns`: these functions are responsible for retrieving the current state of the input value. In our example, we have these functions return immediately with a value of the input. However, in a pratical application, these functions would most likely talk to another process, such as a `gen_server` or `gen_fsm`, to retrieve the current state of another dataflow element.

`Module:process/2` is called every time one of the input values becomes available. This function is called with the current local state and a list of arguments that are derived from the input values returned from `Module:read/1`. In our example, once we have one value for both inputs, we compute a set intersection and send the result via Erlang message passing.

To summarize, each instance of `gen_flow` spawns a tail-recursive process that performs the following steps:

1. Launches a process for each argument in the argument list that executes that argument's `ReadFun`. This is returned by the `Module:read/1` function. This process then waits for a response from the `ReadFun` and replies back to the coordinator the result of the value. Each of these processes are **linked to the coordinator process**, so if any of them die, the entire process chain is terminated and restarted by the supervisor.
2. As soon as the coordinator receives the first response, it updates a local cache of read values for each argument in the argument list.
3. The coordinator executes the `Module:process/2` function with the latest value for each argument in the argument list from the cache. In the event that one of the argument values is not available yet, a bottom value is used; in Erlang, the atom `undefined` is used as the bottom value.
4. Inside `Module:process/2`, the user chooses how to propagate values forward. In our example, we used normal Erlang message passing.

Figure 3 diagrams one iteration of this process. Requests are initially made, from `gen_flow`, to read the current value of the inputs; once the values are known and sent back to the `gen_flow` process, the result of the function is evaluated; finally, the result is written to some output.

This model of computation is similar to how push-based functional reactive programming languages (FRP) [20] operate. In these systems, discrete changes to data items, either referred to as **events** or the more general concept of **signals** [7], notify any "processing elements" of changes to their value which triggers changes to propagate through the graph.[3]

Specifally in push-based FRP, the topology is static and events are pushed through the graph as signals change. We can imagine the `gen_flow` `ReadFuns` as establishing a just-in-time topology: they

[3] We purposely avoid the discussion of behaviors in FRP, given our system focuses on Erlang's basic data structures, none of which observe continuous changes in value.

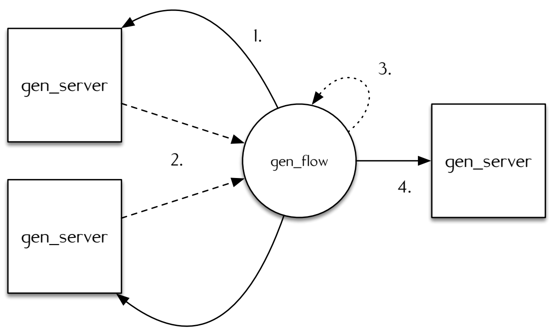


Figure 3: One iteration of the `gen_flow` abstraction. State is first read from inputs, computed locally, and sent to outputs. In this example, we use the `gen_server` abstraction for storage of state.

essentially ask the inputs to notify the **coordinator** in the event of a value change.

2.3 Reading from Inputs

When reading from input values, the `gen_flow` process spawns a set of linked processes to perform each read. These processes are linked to the `gen_flow` process, to ensure if any of them happen to fail, the entire process is crashed (and restarted, if supervised.) Additionally, each request is designated with its position in the argument list, to ensure that when responses arrive, `gen_flow` knows how to properly map the response from the read operation to the correct argument. We see the implementation of this in `gen_flow` below.

```
lists:foreach(fun(X) ->
    ReadFun = lists:nth(X, ReadFuns),
    CachedValue = orddict:fetch(X, DefaultedCache),
    spawn_link(fun() ->
                    Value = ReadFun(CachedValue),
                    Self ! {ok, X, Value}
            end)
    end,
    lists:seq(1, length(ReadFuns))).
```

For each argument to the function, a process is spawned to execute the arguments `ReadFun` given the previously read value taken from the local cache.

2.4 Cache

Additionally, given that the `Module:process/2` function might result in the composition of the arguments, if only one input, of many, happen to change, we need to be able to recompute the function without having to retrieve a value we have already observed for the other inputs. To facilitate this, a local cache is kept at the `gen_flow` process and updated as the value of inputs change. This cache is maintained using an `orddict` local to the `gen_flow` process. We see the implementation of `gen_flow` where it performs the update of this cache and executes `Module:process/2` with the most recent values below.

```
receive
    {ok, X, V} ->
        Cache = orddict:store(X, V, Cache0),
        RealizedCache = [Value || {_, Value}
                                <- orddict:to_list(Cache)],
        {ok, State} = Module:process(RealizedCache,
                                     State0)
end.
```

2.5 Usage

We envision that `gen_flow` can be combined with the existing "generic" abstractions provided by Erlang/OTP to build large, declarative, concurrent dataflow applications in Erlang/OTP.

Both the Erlang/OTP abstractions `gen_server` and `gen_fsm` have shown to be very powerful in practice for the management of state: `gen_server` representing a "generic" server process that receives and responds to messages from clients and `gen_fsm` representing a finite state machine that transitions based on the messages it receives.

Figure 3 outlines an example of how we imagine these facilities can be combined together. In this example, an instance of `gen_flow` is used to built a dataflow composition between state stored in two `gen_server` instances.

3. Lasp

We now motivate the use of `gen_flow` using Lasp.

3.1 Overview

Lasp is a distributed, fault-tolerant, dataflow programming model, with a prototypical implementation provided as a library for use in Erlang/OTP. At its core, Lasp uses distributed, convergent data structures, formalized by Shapiro et al. as Conflict-Free Replicated Data Types (CRDTs) [17], as the primary data abstraction for the developer. Lasp allows users to compose these data structures into larger applications that also observe the same properties that individual CRDTs do.

3.2 Conflict-free Replicated Data Types

Conflict-free Replicated Data Types (CRDTs) are data structures designed for use in replicated, distributed computations. These data types come in a variety of flavors: maps, sets, counters, registers, flags, and provide a programming interface that is similar to their sequential counterparts. These data types are designed to capture concurrency properly: for example, guaranteeing deterministic convergence after concurrent additions of the same element at two different replicas of a replicated set.

One variant of these data structures is formalized in terms of bounded join-semilattices. Regardless of the type of mutation performed on these data structures and whether that function results in a change that is externally non-monotonic, state is always monotonically increasing and two states are always *join*able via a binary operation that computes a *supremum*, or least-upper-bound. To provide an example, adding to a set is always monotonic, but removing an element from a set is non-monotonic. CRDT-based sets, such as the Observed-Remove Set (OR-Set)[4] used in our example, model non-monotonic operations, such as the removal of an item from a set, in a monotonic manner. To properly capture concurrent operations that occur at different replicas of the same object, individual operations, as well as the actors that generate those operations, must be uniquely identified in the state.

The combination of monotonically advancing state, in addition to ensuring that replicas can converge via a deterministic merge operation, provides a strong convergence property: with a deterministic replica-to-replica communication protocol that guarantees that all updates are eventually seen by all replicas, multiple replicas of the same object are guaranteed to deterministically converge to the same value. Shapiro et al. have formalized this property as Strong Eventual Consistency (SEC) in [17].

To demonstrate this property, we look at three examples. In each of these examples, a circle represents an operation at a given

[4] The `riak_dt_orset` used in our examples is a purely functional implementation of the Observed-Remove Set (OR-Set) in Erlang.

replica and a dotted line represents a message sharing that state with another replica, where it is merged in with its current state.

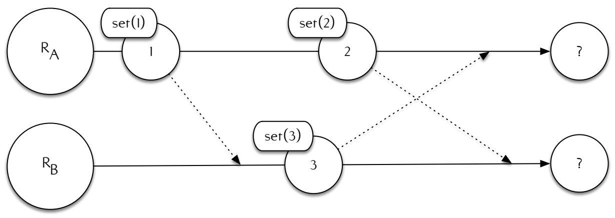


Figure 4: Example of divergence due to concurrent operations on replicas of the same object. In this case, it is unclear which update should win when replicas eventually communicate with each other.

Figure 4 diagrams an example of a distributed register. In this example, concurrent operations happen at each replica resulting in a question of how to handle the merge operation when performing replica-to-replica communication. In this example, it is up to the developer to decide how to resolve a concurrent update.

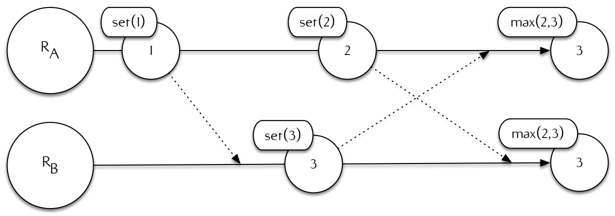


Figure 5: Example of resolving concurrent operations with a type of state-based CRDT based on a natural number lattice where the *join* operation computes *max*.

Figure 5 diagrams a simple state-based CRDT for a max value register, which extends our example in Figure 4. This data structure supports concurrent operations at each replica. In this example, concurrent operations occur where each replica sets the value to a different value (2 vs. 3). However, the CRDT ensures that the objects converge to the correct value: in this case, the **max** function, here used as the merge, is deterministic and monotonic.

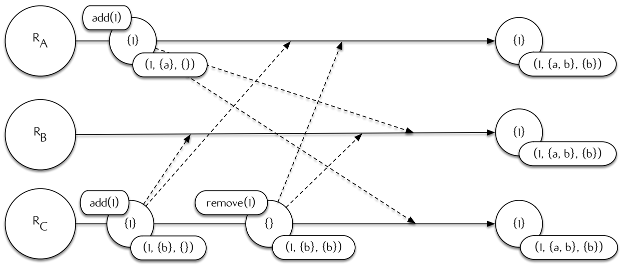


Figure 6: Example of resolving concurrent operations with an Observed-Remove Set (OR-Set). In this example, concurrent operations are represented via unique identifiers at each replica.

Finally, Figure 6 provides an example of the Observed-Remove Set (OR-Set) CRDT, a set that supports the arbitrary addition and removal of the same element repeatedly. In this set, state at each replica is represented as a set of triples, (v, a, r), where v represents the value, a is a set of unique identifiers for each addition, and r is a subset of a for each addition that has been removed. When each addition to the set is performed, the replica performing the addition generates a unique identifier for that operation; when a removal is done, the unique identifiers in the addition set are unioned into the remove set. Presence in a set for a given value is determined on whether the remove set r is a proper subset of a.

This allows the set to properly capture addition and removal operations in a monotonic fashion, supporting the removal and re-addition of the same element multiple times. One caveat does apply, however: when removing an element, removals remove all of the "observed" additions, so under concurrent additions and removals, the set biases towards additions. This is a result of attempting to provide a distributed data structure that has a sequential API. The OR-Set is just one type of CRDT that can model externally non-monotonic behaviour as monotonic growth of internal state.

Lasp is a programming model that uses CRDTs as the primary data abstraction. Lasp allows programmers to build applications using CRDTs while ensuring that the composition of the CRDTs also observed the same strong convergence properties (SEC) as the individual objects do. Lasp provides this by ensuring that the monotonic state of each object maintains a homomorphism with the program state.

3.3 API

Lasp provides five core operations over CRDTs:

- $declare(t)$: Declare a variable of type t.[5]
- $bind(x, v)$: Assign value v to variable x. If the current value of x is w, this assigns the join of v and w to x.
- $update(x, \mathrm{op}, a)$: Apply op to x identified by constant a. *op* is a data structure that performs an operation that is known to t.
- $read(x, v)$: Monotonic read operation; this operation does not return until the value of x is greater than or equal to v at which time the operation returns the current value of x.
- $strict\_read(x, v)$: Same as $read(x, v)$ except that it waits until the value of x is strictly greater than v.

Lasp provides functional programming primitives for transforming CRDT sets:

- $map(x, f, y)$: Apply function f over x into y.
- $filter(x, p, y)$: Apply filter predicate p over x into y.
- $fold(x, \mathrm{op}, y)$: Fold values from x into y using operation op.

Lasp provides set-theoretic functions for composing CRDT sets:

- $product(x, y, z)$: Compute product of x and y into z.
- $union(x, y, z)$: Compute union of x and y into z.
- $intersection(x, y, z)$: Compute intersection of x and y into z.

Figure 7 provides the code for a simple Lasp application. In this application, two sets (`S1` and `S2`) are initially created. The first set (`S1`) is updated to contain three elements, and then the values of the first set (`S1`) are composed into the second set (`S2`) via a higher-order **map** operation. This application is visually depicted in Figure 8.

Each of the functional programming primitives and set-theoretic functions are modeled as Lasp **processes**: as the values of the input CRDTs to these functions change, the value of the output CRDT resulting from applying the function also changes. Lasp processes

[5] Given the Erlang programming library does not have a rich type system, it is required to declare CRDTs with an explicit type at initialization time.

```
%% Create initial set.
{ok, S1} = lasp:declare(riak_dt_orset),

%% Add elements to initial set and update.
{ok, _} = lasp:update(S1, {add_all, [1,2,3]}, a),

%% Create second set.
{ok, S2} = lasp:declare(riak_dt_orset),

%% Apply map operation between S1 and S2.
{ok, _} = lasp:map(S1, fun(X) -> X * 2 end, S2).
```

Figure 7: Example Lasp application that defines two sets and maps the value from one into the other. We ignore the return values of the functions, given the brevity of the example.

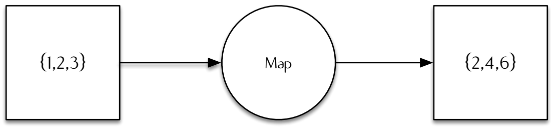


Figure 8: Dataflow graph representing the flow of information described by Figure 7.

are an instance of the gen_flow abstraction. This will be discussed in Section 3.5.

3.4 Distribution

Before discussing Lasp processes, it is important to discuss the implementation of Lasp's distributed runtime.[6]

By default, Lasp provides a centralized runtime that is used for taking single instances of CRDTs on a local machine and composing them into larger applications as discussed in Section 3.3.

In order for Lasp to support highly-available and fault-tolerant distributed computations, Lasp provides a distributed runtime for the execution of Lasp applications. The distributed runtime for Lasp ensures that variables are replicated across a cluster of nodes and operations are performed against a majority quorum of these replicas; this ensures that Lasp applications can tolerate a number of failures while still making progress. The CRDTs that Lasp uses as the primary data abstraction provide safety: even under failures and message re-orderings or duplication, computations will deterministically converge to the correct result once all messages are delivered.

If we return to Figure 8, it is important to realize that all objects in this graph are replicated: there are three copies of the input, three copies of the output, and three copies of the computation running in a distributed cluster at the same time. Additionally, each node in this graph may or may not be running on the same node in the cluster. This is visually depicted in Figure 9.

In Figure 9, it is also important to realize that some replicas may temporarily lag behind, or contain earlier values, given failures in the network. We rely on majority quorums to ensure that we can tolerate failures, in addition to an anti-entropy protocol to ensure that all replicas eventually receive all messages. Under failure conditions, Lasp operations, like **map**, may need to talk to a replica that contains earlier state.

[6] The implementation of Lasp's distributed runtime is out of scope for this paper. However, the reader is referred to our previous work [4] for a discussion on building a distributed deterministic dataflow variant on top of the Erlang-based, Riak Core [12] distributed systems framework.

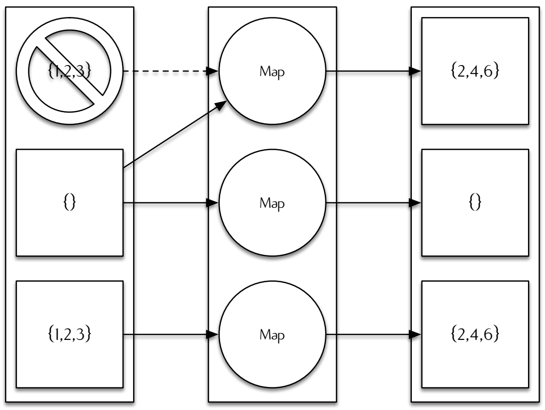


Figure 9: Replicated execution of Figure 8. In this example, since majority quorums are used to perform update operations, some replicas may lag behind until receiving state from another replica due to failures in the network. In the event of a failure, functional operations like the **map** may need to talk to a replica that contains earlier state.

3.5 Execution

As discussed in Section 3.3, all of Lasp's functional programming primitives and set-theoretic functions are implemented in terms of Lasp processes. Lasp **processes** are recursive processes that wait for changes in any of their inputs, compute a function over those inputs, and produce and output based on the functions execution. Lasp processes are implemented in terms of gen_flow.

Lasp's runtime can operate in two modes: centralized and distributed. The distributed runtime is the default for Lasp: variables are replicated across a cluster of nodes. The centralized runtime is provided for testing the semantics of Lasp independently of the distribution layer.

In the centralized runtime, Lasp's variables are stored locally in either a LevelDB instance [5] or in an ETS (Erlang Term Storage) table. The centralized runtime is used for the execution of our QuickCheck [6] model which verifies that the semantics are correct. Additionally, the centralized runtime can be used as the basis for another distribution layer.

In the distributed runtime, Lasp's variables are distributed across a cluster of nodes. Each node, because of Lasp's implementation as a Riak Core application, uses a single Erlang process for managing access to the variables at that node. In Riak Core, this process is referred to as a instance of the riak_core_vnode behaviour; this behaviour is nothing more than a wrapped gen_fsm behaviour with some additional operations focused around distribution of the state and how to route requests in the cluster.

In both of these cases, the ability to provide specific ReadFuns to gen_flow, as discussed in Section 2.2, has proven very valuable. Let us look at two examples:

Centralized Execution When testing the transformation of state with our QuickCheck model, we want to avoid routing requests through the distribution layer. The primary reasons for this are twofold: distribution adds latency to each operation, and distribution makes it harder to reason about when messages may be delivered. In this model, at compile time, we use an Erlang macro to override the ReadFun of our "processing elements" to operate locally on an ETS table and return immediately. This allows for faster execution and the ability to test language semantics separately from the distributed runtime.

Distributed Execution In the distributed execution, we have two concerns that do not appear in the centralized execution. First, replicas may become unavailable and stale replicas may be contacted, as depicted in Figure 9. In this case, we want to ensure that the functions we provide as `ReadFuns` only read forward: Lasp provides a read operation that only returns if the objects state is monotonically greater than a previously observed state; this is commonly referred to as a session guarantee in distributed systems literature. Finally, in the event that replicas may exist on remote nodes, the `ReadFuns` should contain information on how to route the request based on its location.

3.6 Example

Figure 10 shows Lasp's use of `gen_flow`. In this example, the Lasp function and its inputs are passed in through the `gen_flow` initialization function and stored in local state. The `ReadFuns` supplied use the Lasp monotonic read operation: we ensure that given the previous value observed from the cache, we always read forward in causal time. This prevents the observance of earlier values in the event of failures.

4. Evaluation

We have found that having a generic abstraction for dataflow programming has allowed us to greatly simplify the implementation of three dataflow variants: Derflow [4], Derflow$_L$ [13], and Lasp [14].

Our previous work on Derflow provides a distributed, deterministic, single-assignment dataflow programming model for Erlang that was later extended to operate over bounded join-semilattices with Derflow$_L$. Both of these models are direct precursors to Lasp and during the implementation of Lasp we were able to greatly simplify the dataflow "processing elements" by using the generic abstraction presented in this paper. This greatly reduced the complexity and code duplication of the implementation of Lasp; for example, the implementation of an operation that applies identity from one input to an output was reduced from 455 LOC to 128 LOC. Similar results exist for the other operations in Lasp. Additionally, using `gen_flow` enabled us to test the implementation of the CRDT transformation independently of the distribution layer, which was not possible with Derflow and Derflow$_L$.

5. Related Work

In the following section, we identify related work.

5.1 Kahn Process Networks

Kahn process networks (KPNs) [11] present a general model of parallel computation where processes are used to compose data that arrives on input channels into output channels. KPNs are both deterministic and monotonic and are modeled around processes that never terminate.

The `gen_flow` abstraction is both influenced by, and can be used to build applications in the style of, KPNs. Supervised instances of `gen_flow` can be used to model a computing process in a KPN; each of these instances of `gen_flow` never terminates unless instructed to by the application or terminated as a result of a fault in the system. Similar to the binding of the formal parameters at the call site where processes in a KPN are instantiated, functions for reading inputs and producing outputs with `gen_flow` can be provided at runtime or compile time, as seen in Figure 10.

5.2 Functional Reactive Programming

We acknowledge the relationship with Functional Reactive Programming [9, 20], but focus on libraries providing dataflow abstractions that can be combined with idiomatic programming in the host language instead of the traditional approach with domain specific languages.

```erlang
-module(lasp_process).
-behaviour(gen_flow).
-export([start_link/1]).
-export([init/1, read/1, process/2]).
-record(state, {read_funs, function}).

start_link(Args) ->
    gen_flow:start_link(?MODULE, Args).

%% @doc Initialize state.
init([ReadFuns, Function]) ->
    {ok, #state{read_funs=ReadFuns,
                function=Function}}.

%% @doc Return list of read functions.
read(#state{read_funs=ReadFuns0}=State) ->
    ReadFuns = [gen_read_fun(Id, ReadFun) ||
        {Id, ReadFun} <- ReadFuns0],
    {ok, ReadFuns, State}.

%% @doc Computation to execute when inputs change.
process(Args, #state{function=Function}=State) ->
    case lists:any(fun(X) -> X =:= undefined end,
         Args) of
        true ->
            ok;
        false ->
            erlang:apply(Function, Args)
    end,
    {ok, State}.

%% @doc Generate ReadFun.
gen_read_fun(Id, ReadFun) ->
    fun(Value0) ->
        Value = case Value0 of
            undefined ->
                undefined;
            {_, _, V} ->
                V
        end,
        {ok, Value1} = ReadFun(Id, {strict, Value}),
        Value1
    end.
```

Figure 10: Lasp's use of `gen_flow`. In this example, inputs and the Lasp function are passed as arguments to `gen_flow` and stored in local state. The `ReadFuns` supplied take the previously observed value from the cache and perform a monotonic read: this ensures we only ever read forward in causal time.

5.3 FlowPools

FlowPools [16] provide a lock-free, deterministic, concurrent dataflow abstraction for the Scala programming language. FlowPools are essentially a lock-free collection abstraction that support a concurrent append operation and a set of combinators and higher-order operations. FlowPools are designed for multi-threaded computation, but not distributed computation.

FlowPools have a similar abstraction to `gen_flow`, but are focused around connecting collections together using combinators. FlowPools allow lock-free append operations to be performed to add elements to the collection and on these collections support two types of functional transformations: `foreach` and `aggregate`. As elements are added, the `foreach` operation asynchronously executes and applies a transformation to a given collection: to guarantee determinism when functions provided to the foreach may contain side-effects, FlowPools ensure that the function supplied to `foreach` is only called once for each element in the collection. `aggregate` is similar to a fold in functional programming: to ensure determinism a collection must be "sealed" before completing the fold operation across the collection.

5.4 Javelin

Javelin [1] is a Clojure library for performing spreadsheet-like, cell-based dataflow programming. Javelin lets you declare "cells": a "cell" is given an arbitrary S-expression with arguments of other "cell"s. As the value of the source "cell"s change, the S-expression is re-evaluated with the current value of the inputs.

Javelin is unique in that it leverages Clojure's `IWatchable` interface: types that implement `IWatchable` execute a list of functions, stored in the object's metadata, whenever the object's value changes. Our solution uses Erlang behaviours, given that Erlang does not have a way to extend the type system in a similar fashion.

5.5 Luke and Riak Pipe

Riak Pipe [10] and its predecessor, Luke [2] are both Erlang libraries for performing "pipeline processing" developed by Basho Technologies. Both of these libraries focus around creating acyclic processing graphs and provide a similar behaviour interface.

Both of these libraries focus on fixed topologies; the system can not create a new node in the graph once the topology has been instantiated and processing has began. This is a conscious design decision; these frameworks were originally designed for MapReduce [8] style processing in Riak Core [12] based systems where a "pipeline" is established to process a finite set of data with a set group of "phases", or "processing elements." By design, if either of the "phases" happen to fail, the entire "pipeline" is collapsed and an error returned to the caller.

Riak Pipe and Luke also rely on Erlang message passing to deliver output results to the next stage of processing. Riak Pipe synchronizes on the mailbox size, through a series of acknowledgements from the receiver, to support a backpressure mechanism to prevent overloading slow "phases". This is similar to the design outlined by Welsh et al. in their work on SEDA. [21]

We believe that the `gen_flow` abstraction is generic enough to support the implementation of Riak Pipe and Luke. We plan to explore an implementation of Riak Pipe that uses `gen_flow`.

6. Conclusion

We have presented a new "generic" abstraction for Erlang/OTP to support the implementation of dataflow programming languages called `gen_flow`. We have motivated the use of this new abstraction through the implementation of a distributed, deterministic dataflow programming model called Lasp. We have demonstrated that use of this new abstraction has helped reduce code duplication and made it easier to reason about computations in dataflow programming in Erlang/OTP, as well as provided a way to integrate dataflow "processing elements" with the existing Erlang/OTP "generic" abstractions.

A. Source Code Availability

Lasp and `gen_flow` are available on GitHub under the Apache 2.0 License at `http://github.com/lasp-lang/lasp`.

Acknowledgments

Thanks to Sean Cribbs for providing valuable feedback during the research and development of this work. The research leading to these results has been partially funded by the SyncFree Project [3] in the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n^{o} 609551.

References

[1] Javelin source code repository. `https://github.com/tailrecursion/javelin`. Accessed: 2015-05-23.

[2] Luke source code repository. `https://github.com/basho/luke`. Accessed: 2015-05-22.

[3] SyncFree: Large-scale computation without synchronisation. `https://syncfree.lip6.fr`. Accessed: 2015-02-13.

[4] M. Bravo, Z. Li, P. Van Roy, and C. Meiklejohn. Derflow: distributed deterministic dataflow programming for Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 51–60. ACM, 2014.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[6] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 46(4): 53–64, 2011.

[7] E. Czaplicki. Elm: Concurrent FRP for Functional GUIs. *Master's thesis, Harvard*, 2012.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] C. M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 25–36. ACM, 2009.

[10] B. Fink. Distributed computation on Dynamo-style distributed storage: Riak Pipe. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang*, pages 43–50. ACM, 2012.

[11] G. Kahn. The semantics of a simple language for parallel programming. In *In Information Processing74: Proceedings of the IFIP Congress*, volume 74, pages 471–475, 1974.

[12] R. Klophaus. Riak core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.

[13] C. Meiklejohn. Eventual Consistency and Deterministic Dataflow Programming. *8th Workshop on Large-Scale Distributed Systems and Middleware*, 2014.

[14] C. Meiklejohn and P. Van Roy. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, PPDP '15, pages 184–195. ACM, 2015.

[15] C. Meiklejohn and P. Van Roy. Lasp: a language for distributed, eventually consistent computations with CRDTs. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, page 7. ACM, 2015.

[16] A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. FlowPools: A lock-free deterministic concurrent dataflow abstraction. In *Languages and Compilers for Parallel Computing*, pages 158–173. Springer, 2013.

[17] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report RR-7506, INRIA, 01 2011.

[18] T. B. Sousa. Dataflow Programming Concept, Languages and Applications. In *Doctoral Symposium on Informatics Engineering*, 2012.

[19] Van Roy, Peter and Haridi, Seif. *Concepts, techniques, and models of computer programming*. MIT Press, 2004.

[20] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *ACM SIGPLAN Notices*, volume 35, pages 242–252. ACM, 2000.

[21] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243. ACM, 2001.

D Selective Hearing: An Approach to Distributed Eventually Consistent Edge Computation

Selective Hearing: An Approach to Distributed, Eventually Consistent Edge Computation

Christopher Meiklejohn
Basho Technologies, Inc.
Bellevue, WA
Email: cmeiklejohn@basho.com

Peter Van Roy
Université catholique de Louvain
Louvain-la-Neuve, Belgium
Email: peter.vanroy@uclouvain.be

***Abstract*—We present a new programming model for large-scale mobile and "Internet of Things" style distributed applications. The model consists of two layers: a language layer based on the Lasp language with a runtime layer based on epidemic broadcast. The Lasp layer provides deterministic coordination-free computation primitives based on conflict-free replicated data types (CRDTs). The epidemic broadcast layer is based on the Plumtree protocol. It provides a communication framework where clients may only have a partial view of membership and may not want to participate in or have knowledge of all active computations. We motivate the new model with a nontrivial mobile application, a distributed ad counter, and we give the model's formal semantics.**

I. Introduction

Traditional approaches to synchronization increasingly have problems when clients become geographically distributed and more numerous. Specifically, they do not operate within the acceptable latency requirements of most consumer-facing applications.[1] This problem is further complicated by the recent addition of two new classes of large-scale Internet applications: "Internet of Things" sensor networks and mobile applications. "Internet of Things" sensor networks rely on tiered aggregation networks that leverage devices with limited connectivity, limited power, and limited local storage capacity. Mobile applications usually operate with replicated state and allow offline modifications to this state, placing the onus on the application developer to resolve concurrent modifications to replicated data items.

In previous work, we have proposed a solution to the problem of large-scale, coordination-free programming, namely the Lasp programming model [2], [3]. Lasp uses functional programming operations to deterministically compose conflict-free replicated data types (CRDTs) [4]. CRDTs are guaranteed to converge under concurrent operations to replicated state. The composition of CRDTs into larger applications preserves these convergence properties. This means that applications can make progress while offline, propagating their state upstream as connectivity becomes available, and are resilient to both re-ordering and replay of messages. We have implemented the Lasp programming model on a consistent-hashed ring in a datacenter (see Section II-C). However, this architecture is inappropriate for edge computing because ring management is increasingly difficult for growing numbers of limited nodes with intermittent connectivity.

[1]For example, Amazon estimated that every 100ms in latency resulted in a 1% sales loss [1].

This paper proposes a new programming model for edge computing applications such as Internet of Things and mobile applications. The model combines an execution layer based on Lasp with a gossip layer based on epidemic broadcast. These two layers work well together: gossip is adapted to loosely coupled systems and Lasp is adapted to the properties of the gossip layer. Using gossip provides improved placement of application state and computations with that state across a large and variable set of nodes. Using Lasp provides an inherent ability to continue computing despite frequent node disconnections, node failures, and message reordering. The gossip layer is based on previous work on epidemic broadcast trees [5], which provides the efficient and reliable delivery of messages to clusters containing large and dynamically variable numbers of nodes. This paper presents and motivates the programming model and gives a formal semantics of its execution. We are currently implementing and evaluating the model. To our knowledge, this paper is the first to articulate a general purpose programming model using epidemic broadcast as the basis for the language's runtime.

The paper is structured as follows. Section II introduces the concepts we build on: CRDTs, Lasp and its ring-based implementation, and epidemic broadcast trees. Section III gives a motivating example, namely a distributed ad counter for mobile applications. Section IV presents Selective Hearing, which combines the Lasp execution model with a new distribution model based on gossip. Section V gives the formal semantics that defines Lasp execution on the gossip layer. Section VI relates our new model with other models of group management and execution. Section VII concludes and explains how we intend to continue this work.

II. Background

In this section we review Conflict-free Replicated Data Types, Lasp, and Epidemic Broadcast Trees.

A. *Conflict-free Replicated Data Types (CRDTs)*

CRDTs are data structures designed for use in replicated, distributed computations. They come in a variety of flavors, such as maps, sets, counters, registers, and flags, and they provide a programming interface that is similar to their sequential counterparts. They are designed to capture concurrency properly: for example, by guaranteeing deterministic convergence after concurrent additions of the same element at two different replicas of a replicated set.

One variant of these data structures is formalized in terms of bounded join-semilattices. Regardless of the type of mutation performed on these data structures and whether that function results in a change that is externally non-monotonic, state is always monotonically increasing and two states are always *join*-able via a binary operation that computes a *supremum*, or least upper bound. To provide an example, adding to a set is always monotonic, but removing an element from a set is non-monotonic. CRDT-based sets, such as the Observed-Remove Set (OR-Set) used in our example, model non-monotonic operations, such as the removal of an item from a set, in a monotonic manner. To properly capture concurrent operations that occur at different replicas of the same objet, individual operations, as well as the actors that generate those operations, must be uniquely identified in the state.

The combination of monotonically advancing state, in addition to ensuring that replicas can converge via a deterministic merge operation, provides a strong convergence property: with a deterministic replica-to-replica communication protocol that guarantees that all updates are eventually seen by all replicas, multiple replicas of the same object are guaranteed to deterministically converge to the same value. Shapiro et al. have formalized this property as Strong Eventual Consistency (SEC) in [4].

To demonstrate this property, we look at an example. In this example, a small circle represents an operation at a given replica and a dotted line represents a message sharing that state with another replica, where it is merged in with its current state.

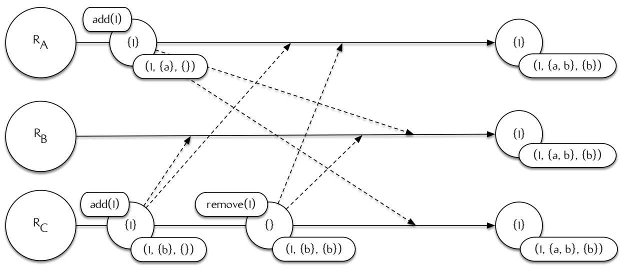


Figure 1: *Example of resolving concurrent operations with an Observed-Remove Set (OR-Set). In this example, concurrent operations are represented via unique identifiers at each replica.*

Figure 1 is an example of the Observed-Remove Set (OR-Set) CRDT. This set uses unique identifiers derived at each replica and represents state at each replica as a triple of values, a set of unique identifiers for each element addition and a set of unique identifiers for each element removal. When removing an element, removals remove all of the "observed" additions, so under concurrent additions and removals, the set biases towards additions.

B. Lasp

Lasp is a programming model that uses CRDTs as its primary data type [2], [3]. Lasp allows programmers to build applications using CRDTs while ensuring that the composition of the CRDTs also observes the same strong convergence properties (SEC) as the individual objects do. Lasp provides this by ensuring that the monotonic state of each object maintains a homomorphism with the program state.[2]

The relevant contribution of the Lasp programming model is the **process**. In Lasp, processes are used to connect two or more instances of CRDTs. One example of a Lasp process is the *filter* operation over sets: as the input set is mutated, the filter function is reevaluated, resulting in a new value for the output. Lasp processes ensure this transformation is both monotonic and deterministic.

C. Ring-Based Distribution Model for Lasp

The Lasp programming model was initially designed and implemented in Erlang [6] using the Riak Core distributed systems library. The Riak Core library provides a framework for building applications in the style of the original Dynamo system as described by DeCandia et al. in 2007 [1]. Riak Core provides library functions for cluster management, dynamic membership, failure detection and state management.

This Lasp implementation uses Dynamo-style partitioning of application state and computations: consistent hashing and hash-space partitioning are used to distribute copies of each variable and Lasp process across nodes in a cluster to ensure high availability and fault tolerance. Replication of each variable's state, and the instantiation of Lasp processes, are performed between adjacent nodes in a cluster and quorum-based operations are used to read and modify variables and the result of computations in the system. Additionally, an anti-entropy protocol is deployed alongside the quorum-based operations to ensure reliable delivery of all messages in the system.

While this model of distribution is designed for fault-tolerance and high-availability, it is inherently skewed towards clusters where both the work and latency distribution across the cluster is uniform.

D. Epidemic Broadcast Trees

Epidemic Broadcast Trees [5], or more specifically the Plumtree protocol, is an efficient, **reliable broadcast** protocol. This approach combines techniques from two previous approaches to reliable broadcast: deterministic tree-based broadcast protocols that have low complexity in message size and are therefore less fault-tolerant, and gossip protocols that have higher complexity in message size but are tolerant to faults.

To achieve efficient and fault-tolerant reliable broadcast, the protocol implements a hybrid approach for each message that is composed of two phases: given a unique identifier for the message, first push the message identifier and payload to nodes contained by the leaves of the broadcast tree, known as the **eager push** phase; then, push only the message identifier to a random sampling of other nodes known by the peer service, known as the **lazy push** phase. If any of the nodes do not receive a message they have learned about through the **lazy push** phase within a designated timeout period, they request this message by identifier from a randomly picked peer in the overlay network.

[2]For more information about how this transformation is performed and maintained, the reader is referred to [2], [3] and [6].

The Plumtree protocol starts off with a random sampling of nodes, selected from a peer service, placed in the **eager** set. As the protocol evolves, nodes are moved from the **eager** set to the **lazy** set as duplicate messages are received. This process of pruning the **eager** set is how the protocol computes the spanning tree that will be used for the eager phase of message broadcast.

III. Motivating Example

We now present an application scenario to motivate our programming model. Figure 2 visualizes an eventually consistent advertisement counter written in Lasp as originally presented in [2], [3]. In this example, shaded circles represent primitive CRDTs and white circles represent CRDTs that are maintained through composition using Lasp operations. Additionally, Lasp operations are represented as diamonds, and directed edges represent the monotonic flow of information in the Lasp application.

Our advertisement counter operates as follows:

- Advertisement counters are grouped by vendor and combined into one list of advertisements using a **union** operation.
- Advertisements are joined with active "contracts" into a list of displayable advertisements using both the **product** and **filter** operations.
- Each client periodically reads the list of active advertisements from the server and stores a copy locally. When displaying an advertisement, clients choose an advertisement from this local list and increment the counter. This allows clients to make progress while offline, but still correctly capture the number of advertisement impressions.
- Clients periodically synchronize their counters with the server. As a counter hits 50,000 advertisement impressions, the advertisement is "disabled" by removing it from the list of active advertisements.

A. *Selection of Consistency Protocol*

Our original Lasp design focused on the replication of all objects across a hash-space partitioned ring using consistent hashing. This design is problematic for the advertisement counter. In the advertisement counter, not all objects may want to adhere to the same consistency protocol. We examine two cases in Figure 2.

Advertisement Counters Each advertisement counter stored at the client is a local copy that is mutated when advertisements are viewed locally. In this case, we do not want to replicate this object at the client or across other clients; we may want to only store a single copy of this counter and periodically synchronize with the server accepting that any impressions between synchronization periods may be lost. In this example fault-tolerance is provided through periodic synchronization periods with the server.

Advertisement Transformations At the server, the transformation using **union**, **product**, **filter** is performed on a weakly consistent store with quorum-based operations. In this case, an administrator may want to use state-machine replication techniques via a system like Zookeeper, or a traditional RDMBS, such as PostgreSQL to store this information.

Given the desired flexibility, we propose that single nodes in the gossip model of this paper can themselves be implemented either as Dynamo-style rings with quorum operations, state-machine replication, or a single register.

IV. Selective Hearing

We now present our new distribution model for Lasp, called Selective Hearing. This algorithm supports large-scale computation with Lasp where clients can incrementally contribute results to computations and selectively receive results of computations as needed. The semantics for this algorithm removes the notion of "replicas" as previously presented in [2], [3] and reasons about a single copy of a data item. This "single copy" is an abstraction over the consistency protocol used to maintain it; for example, it may be maintained with state-machine replication, quorum-based operations on a weakly consistent store, or operations on a single register.

The system model consists of a set of nodes supporting Lasp operations that are implemented using epidemic broadcast. Each node is uniquely identified and tracks a monotonic counter that is incremented with each operation. Nodes can join or leave at any time. Nodes fail by crashing and all messages in the system are eventually delivered to all correct nodes by the epidemic broadcast protocol (reliable broadcast). Crashed nodes disappear from the system; whenever a node recovers it chooses a new identifier and reinitializes its monotonic counter at zero. We can therefore summarize the system model as two layers:

A. *Lasp Layer*

Each node can initiate one of the following operations:

- $declare(t)$: Given type information t, return a new unique variable identifier i that contains the type information and broadcasts this identifier to all nodes. We do not specify in more detail how t is encoded.
- $read(i, p, c)$: Test whether variable i's value v satisfies predicate $p(v)$. If so, call the continuation $c(v)$. If not, add (p, c) to the interest set for variable i. This information will be used by the $bind$ operation when the variable is bound.
- $bind(i, v)$: Broadcast value v, which is then merged with the existing value of variable i on all nodes that have an interest set for i. For these nodes, test all predicates and invoke the corresponding continuation for each predicate that succeeds.

Both predicate p and continuation c have one argument v. Any standard semantics for predicates (i.e., boolean functions) and continuations may be used; for brevity we do not give these semantics since they are not a contribution of this paper. Note that these three operations are designed to commute pairwise for any variable i (see Section V-C). This is an essential property since the gossip layer cannot guarantee delivery order.

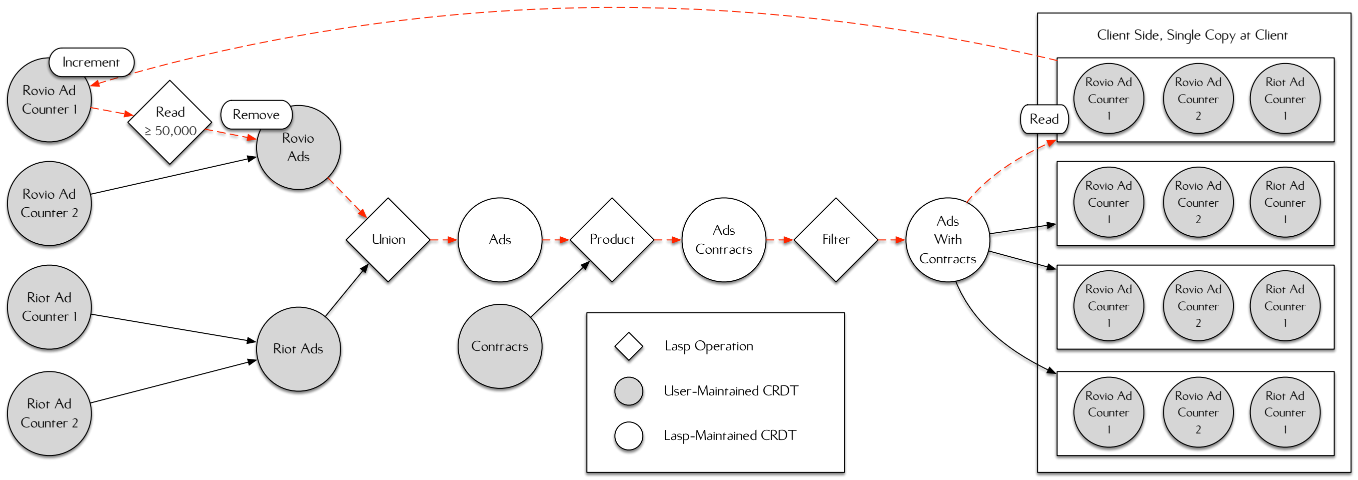


Figure 2: *Eventually consistent advertisement counter as presented in [2], [3]. The dotted line represents the monotonic flow of information for one update. In this example, clients contain only their portion of the shared state they are modifying. Only one copy of the partial counters on the client is stored, it is not replicated in the cluster.*

B. *Gossip Layer*

The gossip layer implements the Plumtree epidemic broadcast protocol. It efficiently implements the broadcast operations required by the Lasp layer. The broadcast operations are not ordered, i.e., a node may receive broadcasts in any order and different nodes may receive them in different orders. Because of the Strong Eventual Consistency property of CRDTs, this does not affect correctness. Furthermore, this allows an important optimization that reduces the computations needed to implement the bind operation.

Bind operations initiated on each node are numbered consecutively via a node-level monotonic counter. Since each variable's successive values are inflations of a lattice, a bind operation that is delivered on a node does not have to invoke local computation if another bind with a greater value has already been delivered.

The Plumtree protocol relies on three properties for fault-tolerant message delivery: (1) Each message can be uniquely identified: given the lazy push phase of the protocol broadcasts only message identifiers, a node is required to know whether that message has been received or not. (2) Nodes must store a history of all messages received. (3) When receiving an identifier for a message, a node must be able to determine if it has already been subsumed by a previous one.

To meet these requirements, we maintain a monotonic clock at each node and store a version vector for each CRDT. This version vector is used to uniquely identify the message when broadcast and allows us to identify messages that have been subsumed by other messages without comparison of payload. By leveraging a per object version vector that is incremented as mutations are performed to each object, we can store a history of all messages received with vector as wide as the number of participating actors in the system.

V. Semantics

We give the formal semantics of the operations in the Lasp layer in terms of the gossip layer's broadcast operation.

A. *Node State*

The system consists of a set of nodes, where the state of each node is a three-tuple $(\sigma, \delta_i, \delta_v)$. Here, σ is the known variables set, δ_i is the interest set, and δ_v is the known values set. The execution of each node is a sequence of states:

$$\begin{aligned}(\sigma^{(0)}, \delta_i^{(0)}, \delta_v^{(0)}) = \\ (\{\}, \{\}, \{\}) \rightarrow \cdots \rightarrow (\sigma^{(k)}, \delta_i^{(k)}, \delta_v^{(k)}) \rightarrow \cdots\end{aligned} \tag{1}$$

The sets are initially empty; the k-th state is denoted by superscript (k). We now define the content of each set.

The *known variables set* σ contains the unique variable identifiers known at the node:

$$\sigma = \{i_0, i_1, \ldots\} \tag{2}$$

The *interest set* δ_i contains information about the variables that the node is interested in, i.e., for which a read operation has been invoked but not yet resolved by the arrival of a new value that satisfies the read predicate. For each variable, the set contains the variable identifier i and a set of pairs of a one-argument predicate p and a one-argument continuation c. When the node receives a new value, then each predicate is evaluated, and for those that succeed the continuation is invoked.

$$\delta_i = \{(i_0, \{(p_0, c_0), \ldots\}),\ (i_1, \{(p_1, c_1), \ldots\}), \ldots\} \tag{3}$$

The *known values set* δ_v contains a set of pairs (i, v) of variable identifiers i and their highest values v observed on the node:

$$\delta_v = \{(i_0, v_0), (i_1, v_1), \ldots\} \tag{4}$$

B. *Basic Invariants on Node State*

We assume that node states obey the following invariants:

- The known variables set σ and the known values set δ_v both grow monotonically: variables will only be added and never removed to both sets.

$$\sigma^{(k)} \subseteq \sigma^{(k+1)} \qquad \delta_v^{(k)} \subseteq \delta_v^{(k+1)} \tag{5}$$

- For any interest set δ_i and known values set δ_v, their identifiers will be contained in some future state of σ[3].

$$\begin{aligned}\forall k.\ \exists n.\ n > k \Rightarrow \pi_i(\delta_i^{(k)}) \subseteq \sigma^{(n)} \\ \forall k.\ \exists n.\ n > k \Rightarrow \pi_i(\delta_v^{(k)}) \subseteq \sigma^{(n)}\end{aligned} \tag{6}$$

C. Operations

All Lasp operations are initiated on one node and may have effects on all nodes; we denote the initiating node by a subscript k. We specify what each operation does on a node state $(\sigma, \delta_i, \delta_v)$ to compute the subsequent state $(\sigma', \delta_i', \delta_v')$; any set that is not mentioned does not change value. In addition to local operations, some operations do a broadcast using the gossip layer; we assume the broadcast message is delivered to *all* nodes including the sending node. We specify what each receiving operation does.

declare The operation $i = \mathit{declare}(t)$ returns a new unique variable identifier i. The operation has the following local specification:

$$i = \mathit{declare}_k(t) : u = \mathit{unique}() \wedge i = (u, t) \tag{7}$$

The variable identifier is a pair of a unique constant u and type information t. The operation then broadcasts the variable identifier with the following specification (the notation $\mathit{declare}_k^j(i)$ means that node k broadcasts to node j). This adds the variable identifier to the known variables σ:

$$\mathit{declare}_k^j(i) : \sigma' = \sigma \cup \{i\} \tag{8}$$

read The operation $\mathit{read}(i, p, c)$ tests whether $p(v)$ holds for the current value v of variable i. If it does, the continuation is invoked as $c(v)$. If not, (p, c) is added to the interest set of variable i, which will delay the invocation until a binding arrives that sufficiently increases i's value.

$$\begin{aligned}\mathit{read}_k(i, p, c) : &(\exists v.(i, v) \in \delta_v \wedge p(v) \Rightarrow c(v) \\ &;\ (\exists s.(i, s) \in \delta_i \Rightarrow \\ &\quad s_n = s \cup \{(p, c)\}; s_n = \{(p, c)\}) \\ &\delta_i' = \delta_i \setminus \{(i, \_)\} \cup \{(i, s_n)\} \\ &\sigma' = \sigma \cup \{i\})\end{aligned} \tag{9}$$

bind The operation $\mathit{bind}(i, v)$ updates the current value stored in δ_v by doing a join with v. The operation has the following local specification:

$$\mathit{bind}_k(i, v) : \text{true} \tag{10}$$

The operation is then broadcast and has the following specification on all other nodes:

$$\begin{aligned}\mathit{bind}_k^j(i, v) : &(\exists v'.(i, v') \in \delta_v \Rightarrow v_n = v \sqcup v'; v_n = v) \\ &\delta_v' = \delta_v \setminus \{(i, \_)\} \cup \{(i, v_n)\} \\ &\sigma' = \sigma \cup \{i\} \\ &\exists s.(i, s) \in \delta_i \Rightarrow \\ &\quad s_{sat} = \{(p, \_) \in s \mid p(v_n)\} \\ &\quad \delta_i' = \delta_i \setminus \{(i, \_)\} \cup \{(i, s \setminus s_{sat})\} \\ &\quad \forall(\_, c) \in s_{sat} : c(v_n)\end{aligned} \tag{11}$$

In variable i's interest set, all pending read operations (all pairs (p, c) in s) are checked. Those for which p succeeds are removed from s and their continuation c is invoked.

D. Processes

A Lasp process is defined as a recursive function that uses the Lasp operations. Figure 3 shows the execution of a Lasp process running the *filter* operation. The example runs on four nodes, (1), (2), (3), and (4).

Subfigure (a) is a Lasp program that creates two instances of the Grow-Only Set (G-Set) CRDT and applies the filter operation with predicate $\lambda x.odd(x)$ from A to B. This Lasp program has four instructions, each of which executes on a different node. The executing node is written to the left of each instruction.

Subfigure (b) is a Lasp program that defines the *filter* operation: a recursive function that repeatedly reads new values of A and computes new values of B. Each iteration executes a read on A that waits for predicate P to be satisfied, at which time the continuation C is executed.

Subfigure (c) shows the operations executed and the state at each node.

This example executes as follows:

1) The $\mathit{declare_1}$ operation is executed on node 1 which locally generates the unique identifier A. This operation results in a $\mathit{declare}_1^j(A)$ message broadcast to all members of the cluster.
2) The $\mathit{declare_2}$ operation is executed on node 2 which locally generates the unique identifier B. This operation results in a $\mathit{declare}_2^j(B)$ message broadcast to all members of the cluster.
3) The *filter* operation is executed on node 3. This operation results in a $\mathit{read_3}(A, P, C)$.
4) The $\mathit{bind_4}(A, \{1, 2, 3\})$ operation is issued on node 4. This operation results in a $\mathit{bind}_4^j(A, \{1, 2, 3\})$ message broadcast to all members of the cluster; however, only node 3 is waiting for a value of A. Given the predicate is satisfied, the continuation is invoked on node 3, trigging a local $\mathit{bind_3}(B, \{1, 3\})$ operation and a broadcast of a $\mathit{bind}_3^j(B, \{1, 3\})$ message. Given no nodes are waiting for a value of B, the message is not processed.

VI. Related Work

We examine related work in programming languages and sensor networks.

A. Process Groups and Programming Languages

The earliest known use of a publish/subscribe model was by Birman and Thomas in 1987 with the ISIS Toolkit [7]. In ISIS, process groups were used to handle the replication of an object using reliable broadcast.

The Quicksilver system described by Ostrowski et al. [8] presents the design of a "live" distributed objects system that has pluggable communication substrates. This system places the onus on the communication layer for managing

[3]We define π as the standard projection.

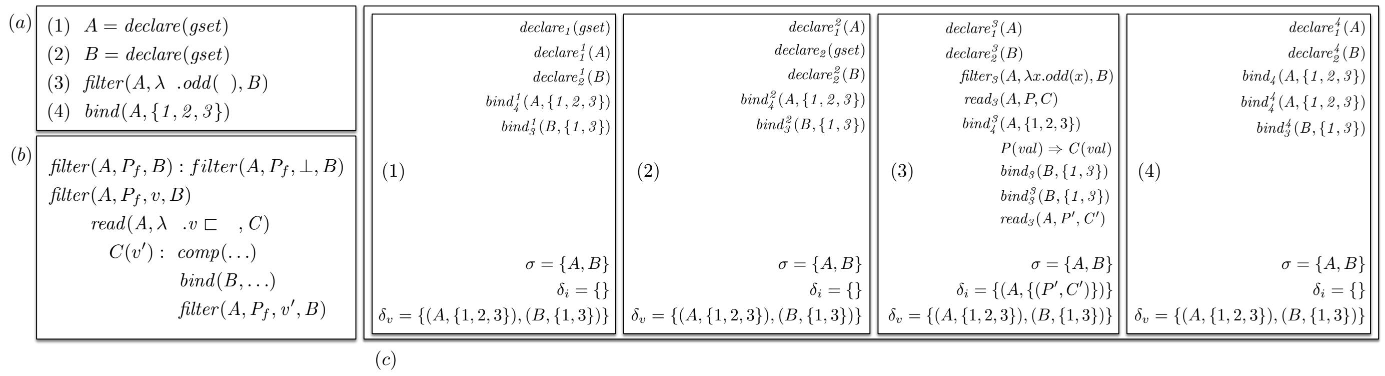


Figure 3: *Execution of a Lasp process over gossip with four nodes. Subfigure (a) shows an example program with the nodes selected for the execution of each Lasp operation; subfigure (b) shows the definition of a Lasp filter process; subfigure (c) shows where each operation executes and where each broadcast message arrives along with the final state at each node at the end of the execution.*

consistency, replication, and propagation of events. While a gossip protocol version is mentioned as future work, no further details of how it would manage object state are provided.

B. Directed Diffusion and Digest Diffusion

Directed diffusion [9] is an efficient protocol for performing realtime dissemination of "interests", metadata describing information that should be collected, and "samples", collections of information coming from the sensors in the network.

Directed diffusion shares many common traits with Selective Hearing: a publish/subscribe paradigm, use of a broadcast protocol, and a API that supports aggregation of information. However, directed diffusion's primary focus is on capturing immutable samples in a large-scale sensor network, whereas Selective Hearing focuses on general computations over shared state using a declarative, functional programming approach.

Digest diffusion [10] presents a energy-efficient model of computation, where operations that are idempotent can be decomposed and distributed in a network so that nodes will converge to a correct value. This work outlines the problems of performing operations that are not idempotent, but can be decomposed: for example, even under ideal network conditions a *count* operation can exhibit anomalies from message duplication.

Our use of the Lasp programming model yields computations that can be decomposed and distributed by design; all of the variables in the language are CRDTs: data structures designed to be resilident to replay and re-ordering of messages.

VII. Conclusion

This paper presents a new distribution model for Lasp based on combining a Lasp execution layer with an epidemic broadcast communication layer. Since the Lasp semantics obeys strong eventual consistency, it can use a communication layer that does not provide ordering guarantees. The model is designed to operate with two new classes of applications: mobile gaming with shared state, and "Internet of Things" style applications. We believe this is the first programming model to use an epidemic broadcast protocol as a core component of its runtime system. We are in the process of implementing this model and evaluating it. We plan to continue improving the design and distribution of Lasp by modeling several industrial applications, including the industrial use cases of SyncFree partner Rovio Entertainment, a mobile gaming provider [11].

Acknowledgment

Thanks to Larry Marburger and our anonymous reviewers for their feedback. This work was partially funded by the SyncFree project in the European Seventh Framework Programme (FP7/2007-2013) under Grant Agreement n° 609551.

References

[1] 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 *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[2] C. Meiklejohn and P. Van Roy, "Lasp: a language for distributed, eventually consistent computations with CRDTs," in *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, 2015, p. 7.

[3] ——, "Lasp: A language for distributed, coordination-free programming," in *Proceedings of the 17th Symposium on Principles and Practice of Declarative Programming (PPDP 2015)*. ACM, Jul. 2015.

[4] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," INRIA, Tech. Rep. RR-7506, 01 2011.

[5] J. Leitao, J. Pereira, and L. Rodrigues, "Epidemic broadcast trees," in *26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE, 2007, pp. 301–310.

[6] "Lasp source code repository," https://github.com/lasp-lang/lasp, accessed: 2015-06-14.

[7] K. Birman and T. Joseph, *Exploiting virtual synchrony in distributed systems*. ACM, 1987, vol. 21, no. 5.

[8] K. Ostrowski, K. Birman, D. Dolev, and J. H. Ahnn, "Programming with live distributed objects," in *ECOOP 2008–Object-Oriented Programming*. Springer, 2008, pp. 463–489.

[9] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: a scalable and robust communication paradigm for sensor networks," in *Proceedings of the 6th annual international conference on Mobile computing and networking*. ACM, 2000, pp. 56–67.

[10] J. Zhao, R. Govindan, and D. Estrin, "Computing aggregates for monitoring wireless sensor networks," in *Sensor Network Protocols and Applications, 2003. Proceedings of the First IEEE. 2003 IEEE International Workshop on*. IEEE, 2003, pp. 139–148.

[11] "SyncFree: Large-scale computation without synchronisation," https://syncfree.lip6.fr, accessed: 2015-02-13.

E Putting Consistency Back Into Eventual Consistency

In Year 1, this work was mentioned as being submitted. Since it was accepted, we include now the final version of the article.

Putting Consistency Back into Eventual Consistency

Valter Balegas Sérgio Duarte Carla Ferreira Rodrigo Rodrigues Nuno Preguiça
NOVA LINCS, FCT, Universidade Nova Lisboa

Mahsa Najafzadeh Marc Shapiro
Inria Paris-Rocquencourt & Sorbonne Universités, UPMC Univ Paris 06, LIP6

Abstract

Geo-replicated storage systems are at the core of current Internet services. The designers of the replication protocols used by these systems must choose between either supporting low-latency, eventually-consistent operations, or ensuring strong consistency to ease 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 identifies which operations would be unsafe under concurrent execution, and allows programmers to select either violation-avoidance or invariant-repair techniques. We show how to achieve the former, while allowing operations to complete locally in the common case, by relying on a *reservation* system that moves coordination off the critical path of operation execution. The latter, in turn, allows 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 similar latency to an eventually-consistent system in the common case.

1. Introduction

To improve 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 [9, 10, 12, 25, 27, 28, 41], 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, which can lead to non-intuitive and undesirable semantics.

These semantic anomalies do not occur in systems that enforce strict serializability, i.e., serialize all operations in real-time order. Weaker models, such as serializability or snapshot isolation, relax synchronization, but still require frequent 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, depending on the operation [25, 41, 43]. In this approach, operations requiring strong consistency still incur high latency and are unavailable when the system partitions. 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 an application specifies the invariants, or consistency rules, that the system must maintain. Unlike models defined in terms of execution orders, Explicit Consistency is defined in terms of application properties: the system is free to reorder execution of operations at different replicas, provided that the specified invariants are maintained.

In addition, we show that it is possible to implement explicit consistency while mostly avoiding cross-datacenter coordination, even for critical operations that could potentially break invariants. To this end, we propose a three-step methodology to derive a safe version of the application. First, we use static analysis to infer which operations can be safely executed without coordination. Second, for the remaining operations, we provide the programmer with a choice of invariant-repair [38] or violation-avoidance techniques. Finally, application code is instrumented with the appropriate calls to our middleware library.

Violation-avoidance extends escrow and reservation approaches [15, 17, 32, 35, 39]. The idea is that a replica coordinates in advance, to pre-allocate the permission to execute some collection of future updates, which (thanks to the reser-

EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04…$15.00.
http://dx.doi.org/10.1145/2741948.2741972

vation) will require no coordination. This amortizes the cost and moves it off the critical path.

Finally, we present the design of Indigo, a middleware for Explicit Consistency built on top of a geo-replicated key-value store. Indigo is designed in a way that is agnostic to the details of the underlying key-value store, only requiring it to ensure properties that are known to be efficient to implement, namely per-key, per-replica linearizability, causal consistency, and transactions with weak semantics [2, 27, 28].

In summary, we make the following contributions:

- Explicit Consistency, a new consistency model for application correctness, centered on the application semantics, and not on the order of operations.
- A methodology to derive an efficient reservation system for enforcing Explicit Consistency, based on the set of invariants associated with the application.
- Indigo, a middleware system implementing Explicit Consistency on top of a causally consistent geo-replicated key-value store.

The remainder of the paper is organized as follows: Section 2 introduces Explicit Consistency. Section 3 gives an overview of our approach. Section 4 presents the analysis for detecting unsafe concurrent operations. 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. Finally, Section 9 concludes the paper.

2. Explicit Consistency

In this section we define precisely the consistency guarantees that Indigo provides. We start by defining the system model, and then how Explicit Consistency restricts the set of behaviors allowed by that 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. Each 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 life cycle 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.

Applications access and modify the database by issuing high-level operations. These operations consist of a sequence of *read* and *write* operations enclosed in *transactions*. An application submits a transaction to a replica; its reads and writes execute on a private copy of the replica state. If the transaction commits, its writes are applied to the local replica (local transaction), and propagate asynchronously to remote replicas, where they are also applied (remote transaction). If the transaction aborts, it has no effect.

We denote by $t(S)$ the state after applying the write operations of committed transaction t to some state S. We define a database snapshot, S_n, as the state of the database after a sequence of committed transactions $t_1, \ldots, t_n$ from the initial database state, S_{init}, i.e., $S_n = t_n(\ldots(t_1(S_{init})))$. The state of a replica results from applying both local and remote transactions, in the order received.

The transaction set $T(S)$ of a database snapshot S is the set of transactions included in S, e.g., $T(S_n) = \{t_1, \ldots, t_n\}$. We say that a transaction t_a executing in a database snapshot S_a happened-before t_b executing in S_b, $t_a \prec t_b$, if $t_a \in T(S_b)$. Two transactions t_a and t_b are concurrent, $t_a \parallel t_b$, if $t_a \not\prec t_b \wedge t_b \not\prec t_a$ [24].

For a given set of transactions T, the happens-before relation defines a partial order among them, $O = (T, \prec)$. We say $O' = (T, <)$ is a valid serialization of $O = (T, \prec)$ if O' is a linear extension of O, i.e., $<$ is a total order compatible with $\prec$.

Transactions can execute concurrently, with each replica executing transactions according to a different valid serialization. We assume the system guarantees state convergence, i.e., 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) [38, 41].

2.2 Explicit Consistency

Explicit Consistency is a novel consistency semantics for replicated systems, where programmers define the application-specific correctness rules that should be met. These rules are expressed as invariants over the database state.

Even if each replica maintains some invariant locally, concurrent updates might still cause violation. Consider for instance a tournament with a maximum capacity, limiting the cardinality of the set of enrolled players. Two replicas could concurrently enroll players into the same tournament, each one respecting the capacity. However, if the merge function is the union of the two sets of players, the capacity might be exceeded nonetheless.

Our formal definition starts with the helper definition of an invariant I, as a logical condition over the state of the database. We say that state S is an *I-valid state* if I holds in S, i.e., if $I(S) = true$.

Definition 2.1 (I-valid serialization)**.** Given a set of transactions T and its associated happens-before partial order $\prec$, $O_i = (T, <)$ is an *I-valid serialization* of $O = (T, \prec)$ if O_i is a valid serialization of O, and I holds in every state that results from executing some prefix of O_i.

We can now formally define the conditions that a system must uphold to ensure Explicit Consistency.

Definition 2.2 (Explicit consistency)**.** A system provides Explicit Consistency if all serializations of $O = (T, \prec)$ are I-valid serializations, where T is the set of transactions executed in the system and $\prec$ their associated partial order.

This concept is related to the I-*confluence* of Bailis et al. [5]. I-confluence defines the conditions under which operations may execute concurrently, while still ensuring that the system converges to an I-valid state. The current work generalizes this to cases where coordination is needed, and furthermore proposes efficient solutions.

3. Overview

Given application invariants, our approach for Explicit Consistency has three steps: *(i)* Detect the sets of operations that may lead to invariant violation when executed concurrently, called I-*offender sets*. *(ii)* Select an efficient mechanism for handling I-offender sets. *(iii)* Instrument the application code to use the selected mechanism on top of a weakly consistent database system.

The first step consists of discovering I-offender sets. This analysis is based on a model of the effects of operations. This information is provided by the application programmer, as annotations specifying the changes performed by each operation. Using this information, combined with the application invariants, static analysis infers the sets of operation invocations that, when executed concurrently, may lead to invariant violation (I-offender sets). Conceptually, the analysis considers all reachable database states and, for each state, all sets of operation invocations that can execute in that state; it checks if executing these operations concurrently might cause an invariant violation. Obviously, it is not feasible to exhaustively consider all database states and operation sets; instead, a practical approach is to use static verification techniques, which are detailed in Section 4.

In the second, the developer decides which approach to use to handle the I-offender sets. There are two options. With the first, *invariant repair*, operations are allowed to execute concurrently, and the conflict resolution rules that merge their outputs should include code to restore the invariants. One example is a graph data structure that supports operations to add and remove vertices and edges; if one replica adds an edge while concurrently another replica removes one of the edge's vertices, the merged state might ignore the hanging edge to ensure the invariant that an edge connects two vertices [38]. A similar approach applies to trees [30].

The second option, *violation avoidance*, consists of restricting concurrency sufficiently to avoid the invariant violation. We propose a number of techniques to allow a replica to execute such operations safely, without coordinating frequently with the others. Consider for instance the enrollment invariant (if a player is enrolled in a tournament, both the player and the tournament must exist). Any replica is allowed to execute the *enrollTournament* operation without coordination, as long as all replicas are forbidden to run *removePlayer* and *removeTournament*. This *reservation* may apply to a particular subset of players and tournaments.

Our reservation mechanisms support such functionality with reservations tailored to the different types of invariants, as detailed in Section 5.

In the third step, 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 supported by Indigo.

4. Determining I-offender sets

In this section we detail the first step of our approach.

4.1 Defining invariants and post-conditions

Defining application invariants An application invariant is described by a first-order logic formula. More formally, we assume the invariant is an universally quantified formula in prenex normal form[1]

$$\forall x_1, \cdots, x_n, \varphi(x_1, \cdots, x_n).$$

First-order logic formulas can express a wide variety of consistency constraints; we give some examples in Section 4.2.

An invariant can use predicates such as $player(P)$ or $enrolled(P, T)$. A user may interpret them to mean that P is a player and that P is enrolled in tournament T; but technically the system depends only on their truth values and on the formulas that relate them. The application developer needs only to specify the effects of operations on the truth values of the terms used in the invariant.

Similarly, numeric restrictions can be expressed through the use of functions. For example, we may use $nrPlayers(T)$ (the number of players in tournament T) to limit the size of a tournament: $\forall T, nrPlayers(T) \leq 5$.

An application must satisfy the conjunction of all invariants.

Defining operation postconditions To express the side effects of operations, postconditions state the properties that are ensured after the execution of an operation that modifies the database. 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. To give some examples, operation $removePlayer(P)$, which removes player P, has a postcondition with predicate clause $\neg player(P)$, stating that predicate $player$ is false for player P. Operation $enrollTournament(P, T)$,

[1] Formula $\forall x, \varphi(x)$ is in prenex normal form if clause φ is quantifier-free. Every first-order logic formula has an equivalent prenex normal form.

which enrolls player P into tournament T, has a postcondition with two clauses, $\mathit{enrolled}(P,T)$ and $\mathit{nrPlayers}(T) = \mathit{nrPlayers}(T)+1$. If the player is already enrolled, the operation produces no side effects.

The syntax for postconditions is given by the grammar:

$$\begin{array}{lcl}
\mathit{post} & ::= & \mathit{clause}_1 \wedge \mathit{clause}_2 \wedge \cdots \wedge \mathit{clause}_k \\
\mathit{clause} & ::= & \mathit{pclause} \mid \mathit{fclause} \\
\mathit{pclause} & ::= & p(o_1, o_2, \cdots, o_n) \mid \neg p(o_1, o_2, \cdots, o_n) \\
\mathit{fclause} & ::= & f(o_1, o_2, \cdots, o_n) = \mathit{opr} \mid \mathit{opr} \oplus \mathit{opr} \\
\mathit{opr} & ::= & n \mid f(o_1, o_2, \cdots, o_n) \\
\oplus & ::= & + \mid - \mid * \mid \ldots
\end{array}$$

where p and f are predicates and functions respectively, over objects $o_1, o_2, \cdots, o_n$.

Although we impose 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 φ with postcondition $\varphi_1 \vee \varphi_2$ could be replaced by operations op_1 and op_2 with postconditions φ_1 and φ_2, respectively.

4.2 Expressiveness of Application Invariants

Despite the simplicity of our model, it can express significant classes of invariants, as discussed next.

4.2.1 Restrictions Over The State

An application can define the set of valid application states, using invariants that define conditions that must be satisfied in every database state. By combining user-defined predicates and functions, it is possible to address a wide range of application semantics.

Numeric constraints Numeric constraints refer to numeric properties of the application and set lower or upper bounds to data values. Often, they control the use or access to a limited resource. For example, to ensure that a player does not overspend her (virtual) budget: $\forall P, \mathit{player}(P) \Rightarrow \mathit{budget}(P) \geq 0$. Disallowing an experienced player from participating in a beginner tournament can be expressed as: $\forall T, P, \mathit{enrolled}(P,T) \wedge \mathit{beginners}(T) \Rightarrow \mathit{score}(P) \leq 30$. By using user-defined functions in the constraints, it is possible to express complex conditions over the database state. We have previously shown how to limit the number of enrolled players in a tournament by using a function that counts the enrolled players. The same approach can be used to limit the number of elements in the database that satisfy any generic condition.

Uniqueness, a common correctness property, may also be expressed using a counter function. For example, the formula $\forall P, \mathit{player}(P) \Rightarrow \mathit{nrPlayerId}(P) = 1$, states that P must have a unique player identifier. Whereas, the formula $\forall T, \mathit{tournament}(T) \Rightarrow \mathit{nrLeaders}(T) = 1$ states that a collection has exactly one leader.

Integrity constraints An integrity constraint specifies the relationships between different objects, such as the *foreign key constraint* in databases. A typical example is the one at the beginning of this section, stating that enrollment must refer to existing players and tournaments. If the tournament application had a score table for players, another integrity constraint might be that every table entry must belong to an existing player: $\forall P, \mathit{hasScore}(P) \Rightarrow \mathit{player}(P)$.

General constraints over the state An invariant may also capture general constraints. For example, consider an application to reserve meetings, where two meetings must not overlap in time. Using predicate $\mathit{time}(M,S,E)$ to mean that meeting M starts at time S and ends at time E, we could write this invariant as follows: $\forall M_1, M_2, S_1, S_2, E_1, E_2$, $\mathit{time}(M_1,S_1,E_1) \wedge \mathit{time}(M_2,S_2,E_2) \wedge M_1 \neq M_2 \Rightarrow E_2 \leq S_1 \vee S_2 \geq E_1$.

4.2.2 Restrictions Over State Transitions

In addition to conditions over database state, we support some forms of temporal specifications, i.e., restrictions over state transitions. Our approach is to turn this into an invariant over the state of the database, by augmenting the database with a so-called *history variable* that records its state in a given moment in the past [1, 33].

In our running example, we might want to state, for instance, that players may not enroll or drop from an active tournament, i.e., between the start and the end of the tournament. For this, when a tournament starts, the application stores the list of participants, which can later be checked against the list of enrollments. If $\mathit{participant}(P,T)$ asserts that player P participates in active tournament T, and $\mathit{active}(T)$ asserts that tournament T is active, the above rule can be specified as follows: $\forall P, T, \mathit{active}(T) \wedge \mathit{enrolled}(P,T) \Rightarrow \mathit{participant}(P,T)$.

An alternative is to use a logic with support for temporal logic expressions, which allow for writing expressions that specify rules over time [24, 34]. Such approach would require more complex specification for programmers and a more complex analysis. We decided to forgo temporal logic, since our experience showed that our simpler approach was sufficient for specifying common application invariants.

4.2.3 Existential quantifiers

Some properties require existential quantifiers, for instance to state that tournaments must have at least one player enrolled: $\forall T, \mathit{tournament}(T) \Rightarrow \exists P, \mathit{enrolled}(P,T)$. This can be easily handled, since the existential quantifier can be replaced by a function, using the technique called skolemization. For this example, we may use function $\mathit{nrPlayers}(T)$ as such: $\forall T, \mathit{tournament}(T) \Rightarrow \mathit{nrPlayers}(T) \geq 1$.

4.2.4 Uninterpreted predicates and functions

The fact that predicates and functions are uninterpreted imposes limitations to the invariants that can be expressed. It implies, for example, that it is not possible to express reachability properties or other properties over recursive data structures. To encode invariants that require such properties, the

```
@Invariant("forall(P : p, T : t) :- enrolled(p, t) =>
player(p) and tournament(t)")
@Invariant("forall(P : p) :- budget(p) >= 0")
@Invariant("forall(T : t) :- nrPlayers(t) <= Capacity")
@Invariant("forall(T : t) :- active(t)
=> nrPlayers(t) >= 1")
@Invariant("forall(T : t, P : p) :- active(t) and
enrolled(p,t) => participant(p, t)")
public interface ITournament {
 @True("player($0)")
 void addPlayer(P p);

 @False("player($0)")
 void removePlayer(P p);

 @True("tournament($0)")
 void addTournament(T t);

 @False("tournament($0)")
 void removeTournament(T t);

 @True("enrolled($0, $1)")
 @False("participant($0, $1)")
 @Increments("nrPlayers($1, 1)")
 @Decrements("budget($0, 1)")
 void enrollTournament(P p, T t);

 @False("enrolled($0, $1)")
 @Decrements("nrPlayers($1, 1)")
 void disenrollTournament(P p, T t);

 @True("active($0)")
 @True("participant(_, $0)")
 void beginTournament(T t);

 @False("active($0)")
 void endTournament(T t);

 @Increments("budget($0,$1)")
 void addFunds(P p, int amount);
}
```

Figure 1. Invariant specification for the tournament application in Java (excerpt)

programmer has to express predicates that encode coarser statements over the database, which lead to a conservative view of safe concurrency. For example, instead of specifying some property over a branch of a tree, the programmer can define the property over the whole tree.

4.2.5 Example

In Figure 1 shows how to express the invariants for the tournament application in our Java prototype. The invariants in the listing are a subset of the examples just discussed. Application invariants are entered as Java annotations to the application interface (or class), and operation side-effects as annotations to the corresponding methods. Our notation was defined to be simple to convert to the language of the Z3 theorem prover, used in our prototype.

4.3 Algorithm

To identify the sets of concurrent operations that may lead to an invariant violation, we perform static analysis of operation postconditions against invariants. This analysis focuses on the case where operations execute concurrently from the same state. Although we assume that in a sequential execution, the invariants hold[2] , nonetheless, concurrently executing operations at different replicas may cause an invariant violation, which we call a *conflict*.

First, we check whether concurrent operations may result in opposite postconditions (e.g., $p(x)$ and $\neg p(x)$), breaking the generic (implicit) invariant that a predicate cannot have two different values. For instance, consider operations $addPlayer(P)$ with effect $player(P)$, vs. $removePlayer(P)$ with effect $\neg player(P)$. These operations conflict, since executing them concurrently with the same parameter P leaves unclear whether player P exists or not in the database. The developer may address this convergence violation by using a conflict resolution policy such as *add-wins* or *remove-wins*.

The remainder of the analysis consists in checking the effect of executing pairs of operations concurrently on the invariant. Our approach is based on Hoare logic [18], where the triple $\{I \land P\}\ op\ \{I\}$ expresses that the execution of operation *op*, in a state where precondition P holds, preserves invariant I. To determine if a set of operations are safe, we substitute their effects on the invariant, obtaining I', and check that the formula I' is valid given that the preconditions to execute the operations hold.

Considering all pairs of operations is sufficient to detect all invariant violations. The intuition why this is correct is that the static analysis considers all possible initial states before executing each concurrent pair, and therefore adding a third concurrent operation is equivalent to modifying the initial state of the two other operations.

To illustrate this process, we consider our tournament application, with the following invariant I:

$$
\begin{aligned}
I = \forall P, T,\ & enrolled(P,T) \Rightarrow player(P) \land tournament(T) \\
& \land \\
& nrPlayers(T) \leq 5
\end{aligned}
$$

For simplicity of presentation, let us examine each of the conjuncts defined in invariant I separately. First, we consider the numeric restriction: $\forall T, nrPlayers(T) \leq 5$, to illustrate how to check if multiple instances of the same operation are self-conflicting. In this case, one of the operations we need to take into account is $enrollTournament(P,T)$ whose outcome affects $nrPlayers(T)$. This operation has precondition $nrPlayers(T) \leq 4$, the weakest precondition that ensures the sequential execution does not break the invariant (see Footnote 2). To determine if this may break the invariant, we substitute the effects of running the $enrollTournament$ operation twice into invariant I. We then check whether this results in a valid formula, when considering also the weakest precondition. In this example, this corresponds to the following derivation (where notation $I\langle f\rangle$ describes the application of effect f in invariant I):

$$
\begin{aligned}
I\ \langle nrPlayers(T) \leftarrow nrPlayers(T)+1\rangle \\
\langle nrPlayers(T) \leftarrow nrPlayers(T)+1\rangle \\
nrPlayers(T) \leq 5\ \langle nrPlayers(T) \leftarrow nrPlayers(T)+1\rangle \\
\langle nrPlayers(T) \leftarrow nrPlayers(T)+1\rangle \\
nrPlayers(T)+1 \leq 5\ \langle nrPlayers(T) \leftarrow nrPlayers(T)+1\rangle \\
nrPlayers(T)+1+1 \leq 5
\end{aligned}
$$

[2] This can be achieved by having a precondition such that an operation produces no side effects, if its sequential execution against a state that does not meet that precondition would violate invariants.

Algorithm 1 Algorithm for detecting unsafe operations.

```
Require: $I$ : invariant; $O$ : operations.
 1: $C \leftarrow \emptyset$ {subsets of unsafe operations}
 2: for $op \in O$ do
 3:     if $\textit{self-conflicting}(I, \{op\})$ then
 4:         $C \leftarrow C \cup \{\{op\}\}$
 5: for $op, op' \in O$ do
 6:     if $\textit{opposing}(I, \{op, op'\})$ then
 7:         $C \leftarrow C \cup \{\{op, op'\}\}$
 8: for $op, op' \in O : \{op, op'\} \notin C$ do
 9:     if $\textit{conflict}(I, \{op, op'\})$ then
10:         $C \leftarrow C \cup \{op, op'\}\}$
11: return $C$
```

The resulting assertion $I' = \textit{nrPlayers}(T) + 1 + 1 \leq 5$ is not ensured when both the initial invariant and the weakest precondition $\textit{nrPlayers}(T) \leq 4$ hold. This shows that concurrent executions of $\textit{enrollTournament}(P, T)$ conflict and *enrollTournament* is a self-conflicting operation.

The second clause of I is $\forall P, T, \textit{enrolled}(P, T) \Rightarrow \textit{player}(P) \wedge \textit{tournament}(T)$. This case illustrates a conflict between different operations. In this case, we check whether concurrent $\textit{enrollTournament}(P, T)$ and $\textit{removePlayer}(P)$ may violate the invariant. Again, we substitute the effects of these operations into the invariant and check whether the resulting formula is valid, assuming that initially the invariant and the preconditions of the two operations hold.

$$\begin{array}{r}
I\ \langle \textit{enrolled}(P,T) \leftarrow \textit{true}\rangle\ \langle \textit{player}(P) \leftarrow \textit{false}\rangle \\
\textit{enrolled}(P,T) \Rightarrow \textit{player}(P) \wedge \textit{tournament}(T)\ \ \langle \textit{enrolled}(P,T) \leftarrow \textit{true}\rangle \\
\langle \textit{player}(P) \leftarrow \textit{false}\rangle \\
\textit{true} \Rightarrow \textit{player}(P) \wedge \textit{tournament}(T)\ \ \langle \textit{player}(P) \leftarrow \textit{false}\rangle \\
\textit{true} \Rightarrow \textit{false} \\
\textit{false}
\end{array}$$

As the resulting formula is not valid, another pair of I-offenders is identified: $\{\textit{enrollTournament}, \textit{removePlayer}\}$.

We now present the complete logic to detect I-offender sets in Algorithm 1. This algorithm statically determines the pairs of operation that are conflicting, which are defined as follows.

Definition 4.1 (Conflicting operations)**.** Operations op_1, op_2, $\cdots$, op_n *conflict* with respect to invariant I if, assuming that I is initially true and the preconditions for op_1 and op_2 to produce side effects are initially true, the result of substituting the postconditions of both operations into the invariant is not a valid formula.

The core of the algorithm is made of auxiliary functions, which use the satisfiability modulo theory (SMT) solver Z3 [11] to verify the validity of the logical formulas used in Definition 4.1. Function $\textit{self-conflicting}(I, \{op\})$ determines whether op is self-conflicting, i.e., if concurrent executions of op with the same or different arguments may break the invariant. Function $\textit{opposing}(I, \{op, op'\})$ determines whether op and op' have opposing postconditions. Function $\textit{conflict}(I, \{op, op'\})$ determines whether the pair of operations break invariant I, by making it false under concurrent execution. They use the solver to check the validity of a set of formulas, namely the invariant, the preconditions for producing effects, and the updated invariant after substituting the effects of both operations.

Algorithm 1 uses these functions for computing I-offender sets in three steps. The initial step (line 2) determines self-conflicting operations. The second step (line 5) determines opposing operations by detecting contradictory predicate assignments for any pair of operations. The last step (line 8) determines other I-offender sets by checking if combining the effects of any two distinct operations raises an invariant violation. If it leads to a conflict, it adds the pair to the set of I-offender sets.

The number of test cases generated is polynomial in the number of operations, $\mathcal{O}(|O|^2)$. However, the satisfiability problem to be solved in each auxiliary function is, in the general case, NP-complete [19]. Z3 relies on heuristics to analyze formulas efficiently, in most cases. The results presented in Section 7.1.1 suggest that it is fast enough to be practical.

5. Handling I-offender sets

The previous step identifies I-offender sets. These sets are reported to the programmer, who decides how each situation should be addressed. We now discuss the techniques that are available to the programmer in Indigo.

5.1 Invariant Repair

One approach is to allow the conflicting operations to execute concurrently, and to repair invariant violations after the fact. Indigo has only limited support for this approach, since it can only address invariants defined in the context of a single database object (even though the object can be complex, such as a tree or a graph). To this end, Indigo provides a library of objects that repair invariants automatically using techniques proposed in the literature, e.g., sets, maps, graphs, trees with different conflict resolution policies [30, 38].

Application developers may extend this library, in order to support additional invariants. For instance, the programmer might want to extend the unbounded set provided by the library, to implement a set with bounded capacity n. She could modify queries such that they ignore excess elements from the underlying unbounded set; however, she must take care to use a deterministic and monotonic algorithm to select the elements to ignore [31].

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, which extend previous ideas from the literature [17, 32, 35, 39, 44]. In comparison to the previous work, we not only combine these ideas in the same system, but we also propose a new implementation, which is optimized for a geo-replicated setting by requiring only peer-to-peer communication, and relying on CRDTs to manage information [38].

5.2.1 Reservations

We now discuss the high-level semantics of the techniques used to restrict the concurrent execution of updates. The next section discusses their implementation in weakly consistent stores.

UID generator: A very common invariant is uniqueness of identifiers [5, 25]. This problem can be easily solved, without coordination, by statically splitting the space of identifiers per replica. Indigo provides this service by appending a replica-specific suffix to a locally-unique identifier.

Multi-level lock reservation: The multi-level lock reservation (or simply multi-level lock) is our base mechanism 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 one of the above rights, no other replica holds rights of a different type. For instance, if a replica holds a *shared forbid*, no other 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, $\{enrollTournament(P, T), removePlayer(P)\}$ is an I-offender set. To avoid the violation of invariants, we can associate an appropriate multi-level lock to each of the operations, for specific values of the parameters. For example, we can have a multi-level lock associated with $removePlayer(P)$, for each value of P. For executing $removePlayer(P)$, it is necessary to obtain the right *shared allow* on the reservation for $removePlayer(P)$. For executing $enrollTournament(P, T)$, it is necessary to obtain the *shared forbid* right on the reservation for $removePlayer(P)$. This guarantees that enrolling some player will not execute concurrently with deleting the same player. However, concurrent enrolls or concurrent removes are allowed. In particular, 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, in turn, 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 locks are a form of lock [17] that can be used to restrict the concurrent execution of operations in any I-offender sets. It would be possible to enforce any application invariants using only multi-level locks. However, in some cases it is possible to provide additional concurrency while enforcing invariants, by using the following reservations.

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 every pair of operations is too restrictive, as getting a *shared allow* on one operation would prevent the execution of all operations that could make any of the other predicates false. The reason why this is overly pessimistic is that, in this case, for executing an operation that makes some predicate false it suffices to guarantee that some other predicate remains true, which can be done by only forbidding the operations that make it false.

To allow for this, Indigo includes a multi-level mask reservation that can be seen as a vector of multi-level locks. For the invariant $P_1 \vee P_2 \vee \ldots \vee P_n$, a multi-level mask with n entries is created, with entry i used to control operations that may make P_i false.

When a replica obtains a *shared allow* right in one entry, it must obtain a *shared forbid* right in some other entry. For example, an operation that may make P_i false needs to obtain the *shared allow* right on the i^{th} entry and a *shared forbid* right on an entry j for which the predicate is true. At runtime, to find an entry to forbid, it is only necessary to evaluate the current value of the predicate associated with each entry that can be locked.

Escrow reservation: For numeric invariants of the form $x \geq k$, we include an escrow reservation for allowing some decrements to execute without coordination [32]. Given an initial value for $x = x_0$, there are initially $x_0 - k$ rights to execute decrements. These rights can be split dynamically among 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.

The variant called *escrow reservation for conditions* checks a count of elements against some condition; for instance, the number of participants in a tournament in the invariant $nrPlayers(T) < k$. In this case, if the same user is enrolled twice concurrently, two rights are consumed, although the number of participants increases by only one. This is conservative, but "leaks" rights. However, if the same user is disenrolled twice concurrently, then the number of users increases by only one; creating two rights might later let the invariant be violated.

Our escrow reservation for conditions addresses this problem using the following approach (considering invariant $c \geq k$). A decrement operation requires rights, just as a normal escrow reservation. However, an increment operation does not create rights immediately, but instead tags the reservation to be reevaluated. One of the replicas, marked as the primary for the reservation, is entrusted with recreating rights. To do so, it evaluates the distance between the current state and the threshold, taking into account the aggregate number of outstanding rights. More precisely, given the current value for $c = c_1$ and the number k_1 of outstanding rights (i.e., rights assigned to a replica and still not used, as known by the primary replica), $c_1 - k - k_1$ rights are created and assigned initially to the primary replica. This can be done either when the reservation is marked for reevaluation, or when new rights are needed.

Partition lock reservation: For some invariants, it is desirable to have the ability to reserve part of a partitionable resource. For example, consider the invariant that forbids two tournaments to overlap in time. Two operations that schedule different tournaments will break the invariant if the time periods overlap. Using a multi-level lock, it would be necessary to obtain an *exclusive allow* for executing any operation to schedule a new tournament.

However, no invariant violation arises if the time periods of concurrent operations do not overlap. To address this case, we provide a partition lock that allows a replica to obtain an *exclusive lock* on an interval of real values.[3] Replicas can obtain locks on multiple intervals, given that no two intervals reserved by different replicas overlap.

In our example, time would be mapped to a real number. To execute the operation that schedules a tournament, a replica would have to obtain a lock on an interval that includes the time from the start to the end of the tournament.

5.2.2 Using Reservations

The analysis from Section 4 outputs I-offender sets and the corresponding invariant violated. A programmer, electing to use the conflict avoidance approach, must select the type of reservation to be used to avoid invariant violations. Figure 1 presents a default mapping between types of invariants and the corresponding reservations. Conservatively, it is always possible to resort to multi-level locks to enforce any invariant, at the expense of admissible concurrency, as discussed earlier.

[3] Partition locks are a simplified version of partitionable objects [44] and slot reservations [35].

Invariant type	Formula (example)	Reservation
Numeric	$x < K$	Escrow(x)
Referential	$p(x) \Rightarrow q(x)$	Multi-level lock
Disjunction	$p_1 \vee \ldots \vee p_n$	Multi-level mask
Overlapping	$t(s_1, e_1) \wedge t(s_2, e_2) \Rightarrow s_1 \geq e_2 \vee e_1 \leq s_2$	Partition lock
Default	—	Multi-level lock

Table 1. Default mapping from invariants to reservations.

When using multi-level locks to prevent the concurrent execution of I-offender sets, it is possible to use different sets of reservations. We call this a reservation system. For example, consider our tournament application with the following two I-offender sets, which follow from the integrity constraint associated with enrollment: $\{enrollTournament(P, T), removePlayer(P)\}$ and $\{enrollTournament(P, T), removeTournament(P)\}$.

Given these I-offender sets, two alternative reservation systems can be used. The first system includes a single multi-level lock associated with $enroll(P, T)$, where this operation would have to obtain a *shared allow* right to execute, while both $removePlayer(P)$ and $removeTournament(T)$ would have to obtain the *shared forbid* right to execute. The second system includes two multi-level locks associated with $removePlayer(P)$ and $removeTournament(T)$, where enroll would have to obtain the *shared forbid* right in both locks to execute.

A simple optimization process is used to decide which reservations to use. As generating all possible combinations of reservation types 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 the most operations controlled by existing reservations, and repeat the previous step.

For each generated combination of reservations, 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 the appropriate rights before executing its code, and to release appropriate rights afterwards. For escrow locks, an operation that consumes rights will acquire rights before its execution (and these rights will not be released when the operation ends). Conversely, an operation that creates rights will create these rights after its execution. For multi-level masks, the programmer must provide the code that verifies the values of the predicate associated with each element of the disjunction.

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 designed to use 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, and the following operations modify its state: *escrow_consume* depletes rights assigned to the local replica; *escrow_generate* generates new rights assigned to the local replica; and *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. For the escrow lock for conditions variant, a replica is tagged as the primary. The *escrow_generate* only creates rights in the primary.

When *escrow_consume* and *escrow_transfer* operations execute in a replica, if that replica has insufficient rights, the operation fails and it 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 operations that transferred new rights from some other replica may still have to be received. Given that the execution of operations is serialized by the 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. For instance, in the tournament example, for removing player P the replica needs a *shared allow* right for player P. Thus, a multi-level lock object manages the rights for the different parameters independently. Each replica can then hold 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 the 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 is implemented using a vector of multi-level lock objects, with operations specifying which multi-level lock must be modified.

The partition lock object maintains which replica owns each interval. When it is created, a single replica holds the complete interval of values. A single operation modifies the state of the object: *pol_giveRight*, which transfers part of the interval owned by the local replica to some other replica. Using the same reasoning as in the previous cases, it is clear that the local replica always has a conservative view of the intervals it owns.

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 [38]; *(iii)* linearizable execution of operations for each object in each datacenter. There are at least two systems that support all these functionalities: SwiftCloud [46] and Walter [41]. Given that SwiftCloud has a more extensive support for CRDTs, which are fundamental for invariant-repair, we decided to build the Indigo prototype on top of SwiftCloud.

Storing reservations 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 at a given datacenter 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 (except for escrow reservations, where the rights that are consumed are not released); new rights are created in this step. After the local execution, the side effects of the operation in the data and reservation objects are propagated and executed in other replicas asynchronously and atomically.

Note that reservations guarantee that operations that can lead to invariant violation do not execute concurrently, but they do not guarantee that the preconditions for the operation to generate side effects hold. For example, in the tournament, before removing a tournament it is necessary to disenroll all players, thus guaranteeing that no player in enrolled.

Reservations manager The reservations manager is a service that runs in each datacenter and is responsible for exchanging reservations between datacenters, tracking reservations in use by local clients, and providing clients the database snapshot information to access the underlying storage. For correctness, it is necessary to enforce that updates of an operation are atomic and that reads are causally consistent with the current rights at each replica. In Indigo, these properties are guaranteed directly by the underlying storage system.

An example shows why these properties are necessary. In our tournament application, to enroll a player it is necessary to obtain the right that allows the enroll (by forbidding the removal of both the player and the tournament). After the enroll completes, the right is released and can be obtained by an operation that wants to remove the tournament. The problem is that if the state observed by the remove tournament operation did not include the previous enrollment, the application could end up deleting the tournament without disenrolling the students, leading to an invariant violation.

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 starting to obtain rights locally. Unlike the process for obtaining 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; therefore, 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, the latency of operation execution can be severely affected. Therefore, reservation rights are obtained proactively 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). The primary of an escrow lock for conditions creates new rights by computing the number of missing rights whenever either it runs out of rights or the object is marked for reevaluation. 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 more frequent enroll operation to execute locally. Partition lock rights are initially assigned to a single replica, and transferred when needed.

The reservations manager 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. The information about ongoing operations is maintained in soft-state. If the machine where the reservations manager runs fails, the ongoing operation will fail when trying to release the obtained rights.

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 a state-machine replication algorithm. Thus, the failure of a machine inside a datacenter does not lead to any data loss. This also applies to the machine running the reservations manager: as explained before, ongoing transactions will fail in this case; committed changes to the reservation objects are stored in the underlying storage system.

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 following two 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 the 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 $nrImpressions(A_i) \leq K_i$, where K_i is the maximum number of times ad A_i should be impressed and the function $nrImpressions(A_i)$ returns the number of times it has been impressed.

Advertisers will typically require ads to be impressed a minimum number of times in some countries. For instance, ad A should be impressed exactly 10,000 times, with at least 4,000 impressions in the US and another 4,000 impressions in the EU. This example is modeled through the following invariants for specifying the limits on the number of impressions (where $nrImpressionsOther$ counts the sum of the number of impressions in datacenters other than those two with the impressions in excess of $4,000$ in the EU or the US):

$$nrImpressionsEU(A) \leq 4,000$$
$$nrImpressionsUS(A) \leq 4,000$$
$$nrImpressionsOther(A) \leq 2,000$$

We modeled this application by having one counter for each ad and region pair. Invariants were defined with the target limits stored in the database: $nrImpressions(R, A) \leq targetImpressions(R, A)$ A single update operation that increments the ad tally was defined, which increments the function $nrImpressions$. Our analysis shows that two increment operations for the same counter can lead to an invariant violation, 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 is 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 numeric invariants for specifying the capacity of each tournament. This leads to a reservation system that uses both escrow lock for conditions and multi-level lock reservation objects. There are three operations that do not require any right to execute: add player, add tournament and disenroll tournament, although the latter accesses 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 rights for executing, and 14% of update operations requiring rights (8% of the operations are enrollment and disenrolments).

7.1.1 Performance of the Analysis

We implemented in Java the algorithm described in Section 4 for detecting I-offender sets, relying on the satisfiability modulo theory (SMT) solver Z3 [11] for verifying invariants. As discussed in Section 4, our algorithm relies on the efficiency of Z3 to be able to analyze programs in reasonable time.

Our prototype was was able to find the existing I-offender sets in the applications we have implemented. The average running time of this process in a recent MacBook Pro laptop was 19 ms for the ad counter applications and 730 ms for the more complex tournament application.

For the evaluation of the analysis, we additionally modeled TPC-W, so that we get results for a standard benchmark application. This application has less invariants to check than our custom applications, but has more operations. The running time for detecting I-offender sets was in this case 320 ms. These results show that although the running time increases with the number of invariants and operations, our algorithm can process realistic applications in reasonable times.

7.2 Experimental Setup

We compare Indigo against three alternative approaches:

Causal Consistency (Causal) As our system was built on top of the causally consistent SwiftCloud system [46], 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.

RedBlue consistency (RedBlue) We have emulated a system with RedBlue consistency [25] 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, while respecting causal dependencies.

Our experiments comprised 3 Amazon EC2 datacenters, US-East, US-West and EU, with inter-datacenter latency presented in Table 2. 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 the US-East datacenter to minimize the overall communication latency and this way optimize the performance of that configuration.

RTT (ms)	US-E	US-W
US-West	81	–
EU	93	161

Table 2. 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 ran the ad counter application with 1000 ads and a single invariant for each ad. The maximum 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 allowed us to measure the peak throughput when operations were able to obtain reservations in advance. The results are presented in Figure 2, and show that Indigo achieves throughput and latency similar to a causally consistent system. Strong and RedBlue results are similar to each other, as all update operations are red and execute in the master DC in both configurations.

Figure 3 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 either read-only or otherwise can be classified as blue and thus execute in the local datacenter, the throughput of RedBlue is only slightly worse than that of Indigo.

Figure 4 details these results, presenting the 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 operations 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 very 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.

Add player has a surprisingly high latency in all configurations. Analyzing the situation, we found out that the reason for this lies in the fact that this operation manipulates very large objects used to maintain indexes, causing all configurations to have a fixed overhead.

7.4 Micro-benchmarks

Next, we examine the impact of key parameters.

Increasing contention Figure 5(a) 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. Furthermore, the results reflect the fact that our middleware introduces an additional level of contention, because operations have to contact the reservation manager.

Increasing number of invariants Figure 5(b) presents the results of the ad counter application with an increasing number of invariants involved in each operation: the operation reads 5 counters (R5) and updates one to three counters (W1 to W3). In this case, the results show that the peak throughput for 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 an impact on the operations that each datacenter needs to execute. To verify our explanation, we ran a workload with operations that access the same number of counters in the weak consistency configuration. The presented results show the same pattern of decreased throughput.

Impact when transferring reservations Figure 5(c) shows the latency of individual operations executed in the US-W datacenter in the ad counter application, for a workload where increments reach the invariant limit for multiple counters and where the rights were initially assigned to a single

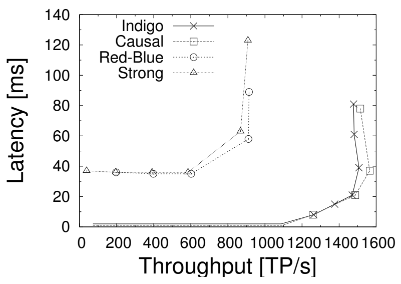


Figure 2. Peak throughput (ad counter application).

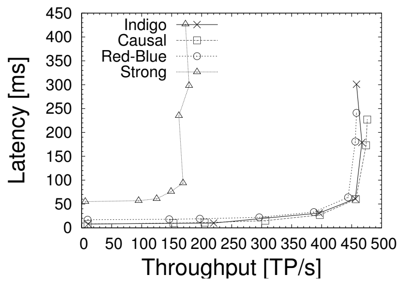


Figure 3. Peak throughput (tournament application).

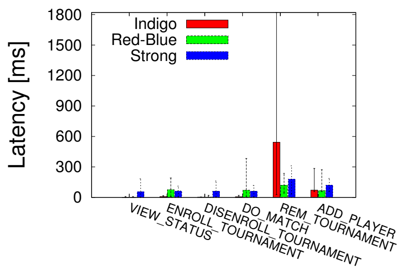
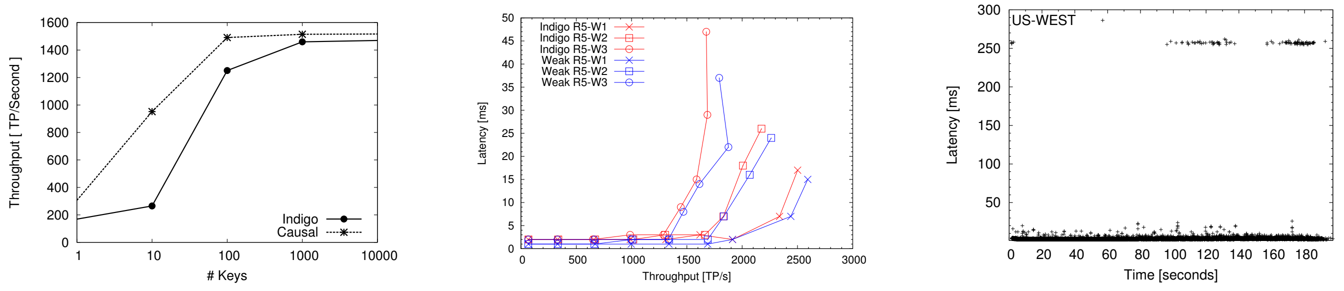


Figure 4. Average latency per op. type - Indigo (tournament app.).



(a) Peak throughput with increasing contention (ad counter application).

(b) Peak throughput with an increasing number of invariants (ad counter application).

(c) Latency of individual operations of US-W datacenter (ad counter application).

Figure 5. Micro-benchmarks.

datacenter. 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. This explains the high latency operations close to the start of the experiment. As a bulk of rights is obtained, the following operations execute with low latency until it is necessary to obtain additional rights. When a replica believes that no other replica has available rights in an escrow lock object, it does not contact replicas. Instead, the operation fail locally, leading to low latency.

In Figure 4, we showed 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 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 can 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 offer variants of eventual consistency, where operations return right after being executed in a single datacenter, usually the closest one, so that end-user response times are improved [2, 12, 23, 27, 28]. These variants target different requirements, such as: reading a causally consistent view of the database (causal consistency) [2, 3, 14, 27]; supporting limited transactions where a set of updates are made visible atomically [4, 28]; supporting application-specific or type-specific reconciliation with no lost updates [7, 12, 27, 41], 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 provides strong consistency for the whole database, at the cost of incurring coordination overhead for all updates [10]. Transaction chains support transaction serializability with latency proportional to the latency to the first replica that is accessed [47]. MDCC [22] and Replicated Commit [29] propose optimized approaches for executing transactions but still incur in inter-datacenter latency for committing transactions.

Some systems combine the benefits of weak and strong consistency models by supporting both. In Walter [41] and Gemini [25], transactions that can execute under weak consistency run fast, without needing to coordinate with other datacenters. Bayou [42] and Pileus [43] allow operations to read data with different consistency levels, from strong to eventual consistency. PNUTS [9] and DynamoDB [40] 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 [16] have been used for building non serializable schedules that preserve consistency in distributed databases. Conflict-free replicated data types [38] explore commutativity for enabling the automatic merge of concurrent updates, which Walter [41], Gemini [25] and SwiftCloud [46] use as the basis for providing eventual consistency. Indigo goes further by exploring application semantics to enforce application invariants.

Escrow transactions [32] 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 [35, 39, 44]. The demarcation protocol [6] 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 [15] 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. [5] 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 in anticipation.

Others have tried to reduce the need for coordination by bounding the degree of divergence among replicas. Epsilon-serializability [36] and TACT [45] use deterministic algorithms for bounding the amount of divergence observed by an application using different metrics: numerical error, order error and staleness. Consistency rationing [21] uses a statistical model to predict the evolution of replica state and allows applications to switch from weak to strong consistency upon 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, including in a context similar to ours [8, 13, 20]. Sieve [26] combines static and dynamic analysis to infer which operations should use strong consistency and which operations should use weak consistency in a RedBlue system [25]. Roy et al. [37] present an analysis algorithm that describes the semantics of transactions. These works are complementary to ours, since 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 in a peer-to-peer manner.

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.

Acknowledgments

We would like to thank our shepherd Gustavo Alonso, the anonymous reviewers and Alexey Gotsman for their helpful comments on a previous version of this work. This research is supported in part by EU FP7 SyncFree project (609551), FCT/MCT SFRH/BD/87540/2012, PTDC/ EEI-SCR/ 1837/ 2012 and PEst-OE/ EEI/ UI0527/ 2014. The research of Rodrigo Rodrigues is supported by the European Research Council under an ERC Starting Grant.

References

[1] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theor. Comput. Sci.*, 82(2):253–284, May 1991.

[2] 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 Com-*

puter Systems, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[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] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable Atomic Visibility with RAMP Transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 27–38, New York, NY, USA, 2014. ACM.

[5] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination Avoidance in Database Systems. *Proc. VLDB Endow. (to appear)*, 2015.

[6] 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.

[7] Basho. Riak. `http://basho.com/riak/`, 2014. Accessed Oct/2014.

[8] 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.

[9] 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.

[10] 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.

[11] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Trans. Database Syst.*, 8(2):186–213, June 1983.

[17] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Readings in Database Systems. chapter Granularity of Locks and Degrees of Consistency in a Shared Data Base, pages 94–121. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[19] H. B. Hunt and D. J. Rosenkrantz. The Complexity of Testing Predicate Locks. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 127–133, New York, NY, USA, 1979. ACM.

[20] 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 *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 41–55. Springer-Verlag, Berlin, Heidelberg, 2011.

[21] 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.

[22] 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.

[23] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2): 35–40, Apr. 2010.

[24] L. Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[25] 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.

[26] 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.

[27] 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 Princi-*

ples, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[28] 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.

[29] 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.

[30] S. Martin, M. Ahmed-Nacer, and P. Urso. Abstract unordered and ordered trees CRDT. Research Report RR-7825, INRIA, Dec. 2011.

[31] S. Martin, M. Ahmed-Nacer, and P. Urso. Controlled conflict resolution for replicated document. In *Proceedings of the 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 471–480. IEEE, Oct 2012.

[32] P. E. O'Neil. The Escrow Transactional Method. *ACM Trans. Database Syst.*, 11(4):405–430, Dec. 1986.

[33] S. S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell University, Ithaca, NY, USA, 1975.

[34] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[35] 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.

[36] K. Ramamritham and C. Pu. A Formal Characterization of Epsilon Serializability. *IEEE Trans. on Knowl. and Data Eng.*, 7(6):997–1007, Dec. 1995.

[37] S. Roy, L. Kot, N. Foster, J. Gehrke, H. Hojjat, and C. Koch. The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (to appear)*, SIGMOD '15. ACM, May-June 2015.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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 31–40, Washington, DC, USA, 1995. IEEE Computer Society.

[45] 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, Berkeley, CA, USA, 2000. USENIX Association.

[46] 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. Research Report RR-8347, INRIA, Oct. 2013.

[47] 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.

F Lasp: A Language for Distributed Eventually Consistency Computations with CRDTs

Lasp: A Language for Distributed, Eventually Consistent Computations with CRDTs

(Work in progress report)

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

Peter Van Roy
Université catholique de Louvain
peter.vanroy@uclouvain.be

Abstract

We propose Lasp, a novel programming model aimed to simplify correct, large-scale, distributed programming. Lasp leverages ideas from distributed dataflow programming extended with convergent data types. This provides support for computations where not all participants are online together at a given moment through Lasp's "convergent by design" applications. Lasp provides a familiar functional programming semantics, built on top of distributed systems infrastructure, targeted at the Erlang runtime system.

The initial Lasp design presented in this report supports synchronization free programming using convergent data types. It combines the expressiveness of these data types together with powerful primitives for composing them. This design lets us write long-lived fault-tolerant distributed applications with non-monotonic behavior. We show how to implement one nontrivial large-scale application, the ad counter scenario from the SyncFree project.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; E.1 [*Data Structures*]: Distributed data structures

Keywords Eventual Consistency, Commutative Operations, Erlang

1. Introduction

Synchronization of data across systems is becoming increasingly expensive and impractical when running at the scale required by "Internet of Things" [12] applications and large online mobile games.[1] Not only does the time required to coordinate with an ever growing number of clients increase with each additional client, but techniques that rely on coordination of shared state, such as Paxos and state-machine replication, grow in complexity with partial replication, dynamic membership, and unreliable networks. [7]

This is further complicated by an additional requirement for both of these applications: each must tolerate periods without connectivity while allowing local copies of replicated state to change. For example, mobile games should allow players to continue to accumulate achievements or edit their profile while they are riding in the subway without connectivity; "Internet of Things" applications should be able to aggregate statistics from a power meter during a snowstorm when connectivity is not available, and later synchronize when connectivity is restored. Because of these requirements, burden is placed on the programmer of these applications to ensure that concurrent operations performed on replicated data have both a deterministic and desirable outcome.

Recently, a formalism has been proposed by Shapiro et al. for supporting deterministic resolution of individual objects that are acted upon concurrently in a distributed system. These data types, referred to as conflict-free replicated data types (CRDTs), provide a property formalized as Strong Eventual Consistency: given all updates to a given object are eventually delivered in a distributed system, all copies of that object will converge to the same state. [13]

While strong eventual consistency is a highly desirable property for a distributed system because operations may arrive at a given replica reordered or duplicated without negatively affecting convergence, it has been demonstrated that the composition of arbitrary CRDTs is non-trivial. [4, 6, 8, 10]

To achieve this goal, we propose a novel programming model aimed at simplifying correct, large-scale, distributed programming, called Lasp[2]. [1] This model provides the ability to use operations from functional programming to deterministically compose CRDTs into larger computations that observe the strong eventual consistency property. This model builds on our previous work, Derflow and Derflow$_L$ [5, 11], a system that provides a distributed, fault-tolerant lattice variable store powering a deterministic concurrency programming model.

We propose the following contributions:

- **Monotonic read:** We provide a `read` operation for CRDTs, which ensures that once a given value is read, all future read operations observe a value causally equivalent or later than the previous read.
- **Functional programming operations:** We provide standard functional programming operations lifted to operate over CRDTs: `map`, `filter`, and `fold`.

[1] Rovio, developer of the popular "Angry Birds" game franchise reported that during the month of December 2012 they had 263 million active users. This does not account for users who play the game on multiple devices, which is an even larger number of devices requiring some form of shared state in the form of statistics, metrics, or leaderboards. [2]

[2] Inspired by LISP's etymology of "LISt Processing", our fundamental data structure is a join-semilattice, hence Lasp.

PaPoC'15, April 21, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3537-9/15/04…$15.00.
http://dx.doi.org/10.1145/2745947.2745954

- **Set-theoretic operations:** We provide set-theoretic operations lifted to operate over CRDTs: `product`, `union`, and `intersection`.
- **Prototype implementation:** We also provide a prototype implementation of Lasp, implemented as an Erlang library using the Riak Core [9] distributed systems framework.

2. Lasp

Lasp operations create processes that connect all replicas of two or more CRDTs.

Each of these processes track the monotonic growth of the internal CRDT state at each replica, and maintain a functional semantics between the state of the input and output instances. The process correctly transforms the internal metadata of the input CRDT to compute the metadata of the output CRDT.[3]

For example, the Lasp `map` operation can be used to connect two instances of the Observed-Remove Set CRDT. [13] The Observed-Remove Set CRDT models arbitrary non-monotonic operations, such as additions and removals of the same element, monotonically, in order to guarantee convergence with concurrent operations at different replicas.[4]

In this example, whenever an element e is added or removed from the input set, the mapped version $f(e)$ is correctly added or removed from the output set. The other operations provided by Lasp are analogous: the user visible behavior is the normal result of the function or set-theoretic operation.

2.1 Semantics

In this report, we provide an example of the semantics of Lasp: the semantics of the `map` operation over the Observed-Remove Set. We focus on the Observed-Remove Set because it is the least-complex CRDT which serves as a general building block for applications.[5]

Each CRDT in Lasp has the appearance of a single CRDT which evolves monotonically over time as update operations are issued. This single CRDT forms a stream of indefinite length, of which a prefix of length n is known, with s' representing future values of the stream. Each value of s is the state of a state-based CRDT.

Definition 2.1 (Streams)**.** A stream is a sequence of infinite length of which only a finite prefix n is known at any given time, while s' represents future values of the stream.

$$s = s_0|s_1|s_2|\ldots|s_{n-1}|s'$$

Definition 2.2 (Observed-Remove Set)**.** The Observed-Remove Set state is a set of triples, where each triple has one value v, with metadata consisting of add set a and remove set r.

$$s_i = \{(v, a, r), (v', a', r'), \ldots\}$$

Additionally, we formalize the `query` function over the Observed-Remove Set, which returns the user-visible value of the data structure, removing all metadata.

Definition 2.3 (Query Operation of an Observed-Remove Set)**.** Presence of a value v in a given Observed-Remove Set, s_i, is determined by comparison of the remove set with the add set. If the remove set is a subset of the add set, the value is in the set.

$$\{v \mid \forall(v, a, r) \in s_i, r \subset a\}$$

Definition 2.4 (Map)**.** The map procedure defines a process that never terminates, which reads elements of the input stream s and creates elements in the output stream t. For each element, the value is separated from the metadata, the function f is applied to the value, and the metadata is attached to the resulting value, $f(v)$.

Algorithm 1 Map algorithm

```
procedure MAP($s, f, t$)
    for all $s_i \in s$ do
        $e \leftarrow \{\}$
        for all $(v, a, r) \in s_i$ do
            $fv \leftarrow f(v)$
            if $\exists a', r'.(fv, a', r') \in e$ then
                $e \leftarrow e \setminus \{(fv, a', r')\} \cup \{(fv, a \cup a', r \cup r'\}$
            else
                $e \leftarrow e \cup \{(fv, a, r)\}$
            end if
        end for
        $t_i \leftarrow e$
    end for
end procedure
```

Figure 1 provides an example of applying the `map` function to an Observed-Remove Set. In this example, the user does not need to program against the internal data structure of each CRDT, only the non-monotonic external representation, as the Lasp runtime handles the metadata mapping automatically.

```erlang
%% Create initial set.
{ok, S1} = lasp:declare(riak_dt_orset),

%% Add elements to initial set and update.
{ok, _} = lasp:update(S1, {add_all, [1,2,3]}, a),

%% Create second set.
{ok, S2} = lasp:declare(riak_dt_orset),

%% Apply map operation between S1 and S2.
{ok, _} = lasp:map(S1, fun(X) -> X * 2 end, S2).
```

Figure 1: Map operation applied to an Observed-Remove Set. In this example, the application developer does not have to program using the internal structure of the CRDT, given the Lasp map operation is lifted to operate over the internal state. We ignore the return values of these functions, given the brevity of the example.

3. Advertisement Counter Example

One of the use cases for our language is supporting clients that need to operate without connectivity. For example, imagine a provider of mobile games that sells advertisement space within their games.

In this example, the correctness criteria is such:

- Clients will go offline: consider mobile devices such as cellular phones that experience periods without connectivity. In the event the client is offline, advertisements should still be able to be displayed to the user.
- Advertisements need to be displayed a minimum number of times, additional impressions, within a certain bound, is not problematic.

[3] The internal metadata of each CRDT is responsible for ensuring correct convergence; this requires that this transformation be deterministic at each replica.

[4] It is paramount that the metadata transformation is performed correctly, or replicas of the `map` operation will not converge correctly.

[5] For instance, the Grow-Only Set does not allow removals, the Two-Phase Set only allows one removal of a given item, and the Observed-Remove Set without tombstones adds additional complexity in the form of optimizations, which lie outside of the core language semantics.

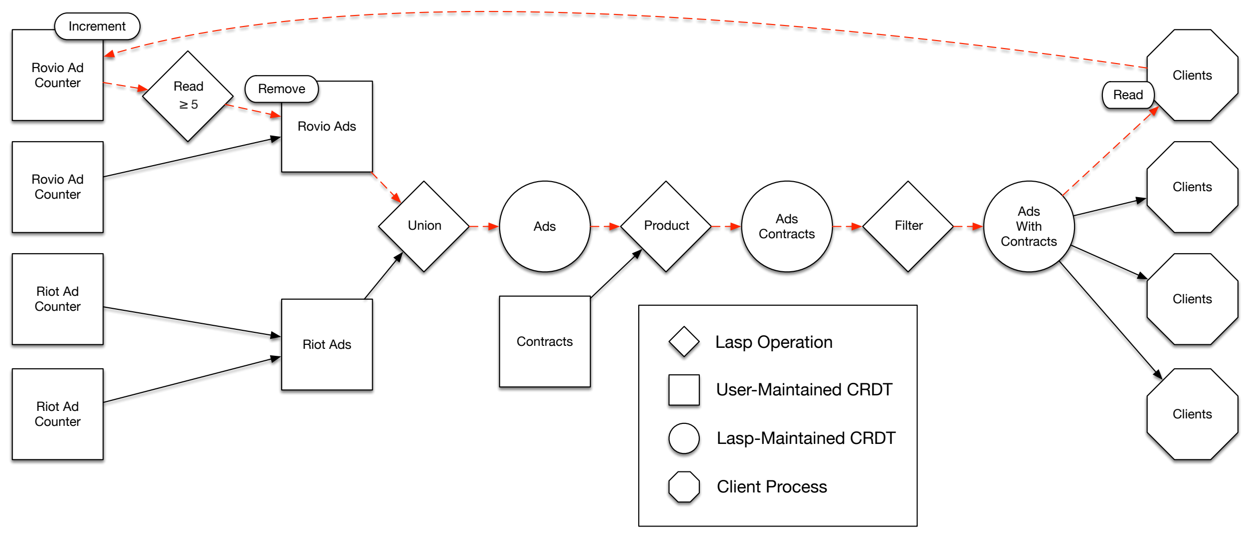


Figure 2: Eventually consistent advertisement counter. In this example, the dotted line represents the monotonic flow of information for one counter.

Figure 2 visualizes an eventually consistent advertisement counter written in Lasp. In this example, squares represent primitive CRDTs, where circles represent CRDTs that are maintained through composition using Lasp operations. Additionally, Lasp operations are represented as diamonds, and edges represent the monotonic flow of information in the Lasp application.

Our advertisement counter operates as follows:

- Advertisement counters are grouped by vendor.
- All advertisement groups area combined into one list of advertisements using a `union` operation.
- Advertisements are joined with active “contracts” into a list of displayable advertisements using both the `product` and `filter` operations.
- Each client reads the list of active advertisements when displaying an advertisement.
- For each advertisement displayed, each client updates the associated advertisement counter.
- As a counter hits five advertisement impressions, the advertisement is “disabled” by removing it from the list of advertisements.

The implementation of this advertisement counter is completely monotonic and synchronization-free. Adding and removing ads, adding and removing contracts, and disabling ads when their contractual number of views is achieved are all modeled as the monotonic growth of state in CRDTs connected by active processes. Programmer-visible non-monotonicity is represented by monotonic metadata in the CRDTs. The initial Lasp design, which supports only programming with zero synchronization and optimistic replication, has sufficient functionality to model this application.

4. Conclusion and Future Work

We introduced the Lasp programming model and motivated its use for large-scale computation over replicated data. Our future plans for Lasp include identifying optimizations for more efficient state propagation, exploring stronger consistency models, and optimizing distribution, and replica placement for better fault-tolerance and reduced latency in computations. Our ultimate goal is for Lasp to become a general purpose language for building large-scale distributed applications in which synchronization is used as little as possible.

A. Code Availability

All of the code discussed will be available on GitHub under the Apache 2.0 License at `http://github.com/cmeiklejohn/lasp`.

Acknowledgments

Thanks to Peter Alvaro, Sean Cribbs, Scott Lystig Fritchie, and Andrew Stone for their valuable feedback. The research leading to these results has been partially funded by the SyncFree Project [3] in the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n^o 609551.

References

[1] Lasp source code repository. `https://github.com/cmeiklejohn/lasp`. Accessed: 2015-02-14.

[2] 263 Million Monthly Active Users In December. `http://www.rovio.com/en/news/blog/261/263-million-monthly-active-users-in-december/`. Accessed: 2015-02-13.

[3] SyncFree: Large-scale computation without synchronisation. `https://syncfree.lip6.fr`. Accessed: 2015-02-13.

[4] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein. Consistency without borders. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 23. ACM, 2013.

[5] M. Bravo, Z. Li, P. Van Roy, and C. Meiklejohn. Derflow: distributed deterministic dataflow programming for Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 51–60. ACM, 2014.

[6] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott. Riak DT map: a composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, page 1. ACM, 2014.

[7] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[8] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM, 2012.

[9] R. Klophaus. Riak core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.

[10] C. 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*, page 13. ACM, 2014.

[11] C. Meiklejohn. Eventual Consistency and Deterministic Dataflow Programming. *8th Workshop on Large-Scale Distributed Systems and Middleware*, 2014.

[12] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.

[13] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, et al. A comprehensive study of convergent and commutative replicated data types. 2011.

G A Study of CRDTs that do Computations

A Study of CRDTs that do Computations

David Navalho Sérgio Duarte Nuno Preguiça
NOVA LINCS, FCT, Universidade NOVA de Lisboa

Abstract

A CRDT is a data type specially designed to allow instances to be replicated and modified without coordination, while providing an automatic mechanism to merge concurrent updates that guarantees eventual consistency. In this paper we present a brief study of computational CRDTs, a class of CRDTs whose state is the result of a computation over the executed updates. We propose three generic designs that reduce the amount of information that each replica maintains and propagates for synchronizations. For each of the designs, we discuss the properties that the function being computed needs to satisfy.

1. Introduction

Cloud infrastructures, composed of multiple data centers spread across the globe, have become central for the deployment of novel Internet services, from social networks to business applications. A large number of cloud databases have been developed in recent years, providing different level of consistency, from strong [5] to eventual consistency [2, 6, 7].

In this paper we focus on cloud databases that provide eventual consistency only. When using an eventually consistent database, applications can be made highly available by replicating the application code and data in multiple data centers and allowing a user to access any of these data centers. Low latency is achieved by routing the client requests to the closest data center and executing the request in the data center without coordinating with other data centers.

In such settings, concurrent updates may be executed in different replicas. Systems must provide a mechanism to handle concurrent updates and enforce eventual convergence of all replicas. CRDTs [10] have been proposed as a technique for helping application programmers to deal with concurrent updates. They provide eventual consistency with well defined semantics and thus make these systems more amenable to programmers. CRDTs have been adopted as a key feature in a leading cloud database, Riak, and are used in multiple large-scale systems, such as SoundCloud and Twitter's Summingbird[3].

Most CRDTs proposed in literature are replicated forms of collections. In such data types, a replica needs to maintain all data elements in all replicas. Thus, a model where every data replica maintains the same state and where all updates are propagated to all replicas is natural.

In some cases, applications are not interested in actual elements or updates, but instead on the result of a computation over them. We call computational CRDTs to the class of CRDTs whose state is the result of a computation over the executed updates. For example, a counter CRDT [10] counts the number of times an increment operation has been executed. In such cases, each replica does not need to maintain every individual update, but can instead maintain for each replica an integer that counts the number of increments executed at that replica. For synchronizing replicas, it also suffices to propagate an integer that summarizes a set of updates.

In the remaining of this paper we present a brief study of the properties of computational CRDTs. In particular, we propose three generic designs that minimize the data that needs to be maintained in each replica and that needs to be propagated for synchronizing replicas. We study the properties of functions suitable to each of the designs. Notably, our last design departs from the strict model of state-based CRDTs by the fact that the state of each replica does not need to converge, although the result of all queries executed in every replica is the same.

1.1 Related work

Aggregation techniques have been studied extensively in different settings, such as as sensor networks [11]. Our work can build on the proposed algorithms for creating replicated data types that perform computations in a cloud database.

The techniques used to model CRDTs have been used to express a distributed deterministic dataflow model for concurrent communication between processes [8]. They have also been used to provide algebraic structures for integration between batch and stream processing of aggregations [3] and to support incremental computations [9]. Unlike these works, this paper studies CRDTs that can be integrated in a cloud database as an elementary abstraction to perform computations without requiring additional support from the system.

PaPoC'15, April 21–24, 2015, Bordeaux, France.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3537-9/15/04. . . $15.00.
http://dx.doi.org/10.1145/2745947.2745948

The problem of optimizing information propagated for synchronizing replicas has been studied by Almeida et. al [1], who have proposed a principled approach to merge the changes produced by multiple operations and use this information to update a remote replica. In our work, the information propagated to synchronize replicas also summarizes multiple updates. However, all information is handled in the context of the CRDT. Additionally, our last design departs from the strict state-based CRDT model by allowing replicas to maintain different state.

2. System model

We adopt the CRDT state-based model [10], where replicas synchronize in a peer-to-peer way, by sending their state to other replicas, where the received state is merged with the current state. A CRDT has an interface that includes update operations that modify the state of the object. In our presentation, we define an event as an invocation of an update operation. For simplicity, we consider a single read-only operation that returns the state of the object. A CRDT includes an additional operation, *merge*, to merge a copy of a remote replica with the current replica state. In one design, we extend this model to allow a replica to send only a subset of its state to other replicas.

For fault-tolerance, we assume a crash-recovery model, where a replica that fails recovers with its state intact. In a typical cloud deployment, each data center can be seen as a single replica, although internally an object is replicated in a quorum of replicas.

3. Design 1: Incremental Computations

Our first design considers computations that can be done incrementally. In this case, computing the function over two disjoint sets of events and combining the results is equal to computing the function over the union of the two sets. Formally, a computation is incremental if there is a function `fun`, such that:

$$\mathcal{F}^{\texttt{fun}}(E_1 \cup E_2, \mathsf{hb}_{E_1 \cup E_2}) = \texttt{fun}(\mathcal{F}^{\texttt{fun}}(E_1, \mathsf{hb}_{E_1}), \mathcal{F}^{\texttt{fun}}(E_2, \mathsf{hb}_{E_2}))$$

where E_1 and E_2 are disjoint sets of events (operation invocations), hb_E is a partial causality order on E[1], and $\mathcal{F}^{\texttt{fun}}$ is the function that defines the state of a CRDT that computes fun over the observed events (following loosely the formalization proposed by Burckhardt et. al.[4]).

For example, a counter with a single update operation for increment, `inc`, can be defined as follows:

$$\begin{aligned} \mathcal{F}^{+}_{\texttt{ctr}}(E, \mathsf{hb}) &= |\{e \in E : e = \texttt{inc}\}| \\ \mathcal{F}^{+}_{\texttt{ctr}}(E_1 \cup E_2, \mathsf{hb}) &= \mathcal{F}^{+}_{\texttt{ctr}}(E_1, \mathsf{hb}) + \mathcal{F}^{+}_{\texttt{ctr}}(E_2, \mathsf{hb}) \end{aligned}$$

For these computations, Figure 1 presents a generic CRDT design that is parameterized by the following elements: (i) V_0, the initial state associated with a replica; (ii) $\texttt{fun}(o)$, the value of the computation for a single operation o; (iii) $\texttt{fun}(s_1, s_2)$, the function to compose two partial results; and (iv) $\texttt{fun}^{\texttt{max}}(v_1, v_2)$, that returns the latest of two values.

[1] For simplicity of presentation, we drop the subscripts of hb in the rest of the paper.

In this design, each replica computes its contribution to the final value of the CRDT independently. Each replica maintains a map for the contributions of each replica. When executing an update operation, a replica updates its contribution by using function `fun` to combine the previous computed contribution and the contribution of the new operation (with $s[i \mapsto \texttt{fun}(s[i], \texttt{fun}(\texttt{op}))]$ representing the replacement in s of the value of entry i by the new computed value). When merging two replicas, for the partial result of each replica, the most recently computed result must be kept, which is returned by $\texttt{fun}^{\texttt{max}}$. If the values are monotonic, it is immediate to know what is the most recent version. Otherwise, it might be necessary to maintain this information explicitly. The value of a replica is computed by applying the function `fun` to the contributions of all replicas.

As an example, a positive-negative counter, with an increment and a decrement operations can be defined by making:

$$\begin{aligned} V_0 &= (0,0) \\ \texttt{fun}(\texttt{inc}) &= (1,0) \\ \texttt{fun}(\texttt{dec}) &= (0,1) \\ \texttt{fun}((p,m),(p',m')) &= (p+p', m+m') \\ \texttt{fun}^{\texttt{max}}((p,m),(p',m')) &= (max(p,p'), max(m,m')) \end{aligned}$$

A CRDT that computes the average of values added to an object, which could be used for example to present the average rating in a web application, can be defined by making:

$$\begin{aligned} V_0 &= (0,0) \\ \texttt{fun}(\mathsf{add}(x)) &= (x,1) \\ \texttt{fun}((s,c),(s',c')) &= (s+s', c+c') \\ \texttt{fun}^{\texttt{max}}((s,c),(s',c')) &= (s,c), \text{ iff } c > c' \\ & (s',c'), \text{ iff } c \le c' \end{aligned}$$

The average is computed as s/c, with (s,c) the result of the read defined in the generic CRDT design.

Other CRDTs can be defined using a similar approach, including a CRDT that computes a histogram.

4. Design 2: Incremental Idempotent Computations

In some cases, the computation to be performed besides being incremental is also idempotent. In this case, computing the function over two (potentially overlapping) sets of events and combining the results is equal to computing the function over the union of the two sets. Formally, a computation is incremental and idempotent if there is a function `fun`, such that for any sets of events E_1 and E_2 we have:

$$\mathcal{F}^{\texttt{fun}}(E_1 \cup E_2, \mathsf{hb}) = \texttt{fun}(\mathcal{F}^{\texttt{fun}}(E_1, \mathsf{hb}), \mathcal{F}^{\texttt{fun}}(E_2, \mathsf{hb}))$$

Replica state	Σ	$=$	$\mathbb{I} \rightarrow V$
Initial state	σ_i^0	$=$	V_0
Update op at replica i	$\mathtt{op}_i(s)$	$=$	$s[i \mapsto \mathtt{fun}(s[i], \mathtt{fun}(\mathtt{op}))]$
Read at replica i	$\mathtt{op}_i(s)$	$=$	$\mathtt{fun}(s[i], \forall i)$
Merge replica states	$\mathsf{deliver}\,(s, s')$	$=$	$s[i \mapsto \mathtt{fun}^{\mathtt{max}}(s[i], s'[i])], \forall i$

Figure 1: Generic CRDT for incremental computation.

Replica state	Σ	$=$	V
Initial state	σ_i^0	$=$	V_0
Update op at replica i	$\mathtt{op}_i(s)$	$=$	$\mathtt{fun}(s, \mathtt{fun}(\mathtt{op}))$
Read at replica i	$\mathtt{op}_i(s)$	$=$	s
Merge replica states	$\mathsf{deliver}\,(s, s')$	$=$	$fun(s, s')$

Figure 2: Generic CRDT for incremental idempotent computation.

For these computations, Figure 2 presents a generic CRDT design. In this case, it is possible to keep in each replica only the computed result that is modified when executing update and merge operations.

A computation that obeys these conditions is computing the maximum of the values added to an object, which could be used in a game application for keeping the highest score. This data type could be implemented, keeping a name associated with the highest score, with names totally ordered, by making:

$$\begin{aligned} V_0 &= (-,\text{minimum value}) \\ \mathtt{fun}(\mathtt{add}(n,v)) &= (n,v) \\ \mathtt{fun}((n,v),(n',v')) &= (n,v), \text{ iff } v > v' \vee (v = v' \wedge n > n') \\ &\quad\; (n',v'), \text{ otherwise} \end{aligned}$$

A generalization of the maximum CRDT is a top-K CRDT that keeps the K players with highest scores, which can be used to maintain a leaderboard in a game application. This CRDT can be implemented by making:

$$\begin{aligned} V_0 &= \{\} \\ \mathtt{fun}(\mathtt{add}(n,v)) &= \{(n,v)\} \\ \mathtt{fun}(s,s') &= \mathtt{max}_\mathtt{k}\big(\{(n,v) \in (s \cup s') : \\ &\quad \nexists (n,v_1) \in (s \cup s') : v_1 > v\}\big) \end{aligned}$$

with $\mathtt{max}_\mathtt{k}(s)$ a function that returns the k largest elements $(n,v) \in s$, with the elements ordered using the total order defined previously.

In general, this approach can be used to create CRDTs that compute a filter over the values added to the object, for which an element that does not match the filter at some moment will not match the filter at a later moment.

5. Design 3: Partially Incremental Computations

We now consider computations that are only partially incremental, in the sense that some updates observe the incremental property previously defined, while others do not. An example of such an object is a top-K object where an element can be deleted. In such cases, a value that does not belong to the top-K elements may later become part of the top, after a top element is deleted.

To address this case, a possible approach is to use a Set CRDT to maintain the set of elements that have not been deleted. In this case, all replicas maintain the complete set, and all updates need to be propagated to all replicas. The top-K can be computed locally on the value of each replica.

In Figure 3 we present an alternative approach, in which each replica maintains all operations locally executed, and each replica only propagates to other replicas the operations that might affect the computed result. Each replica maintains a set of operations and the results of the computation performed at other sites — for simplicity of notation, we assume that the result of the computation is a subset of operations. An update operation updates the local set of operations. A read operation makes the computation considering the local operations and the results of the computation at the other replicas. For synchronizing replicas, a replica sends the results of the computations to all replicas and the subset of operations known locally that can affect the computed result at other replicas (in the top-k example, a delete of an element that belongs to the top elements). When receiving the state from a remote replica, the local replica is updated by merging the local set of operations with the remote operations that may affect the result of the computation, and by registering the most recent version of the computation for each site.

A top-k replicated data type that supports an $\mathtt{add}(n,v)$ and $\mathtt{del}(n)$ operations can be defined as follows:

$$\begin{aligned} V_0 &= \{\} \\ \mathtt{fun}(s) &= \mathtt{max}_\mathtt{k}\big(\{o \in s : o = \mathtt{add}(n,v) \wedge \\ &\quad (\nexists o' \in s : o \prec o' \wedge o' = \mathtt{del}(n))\}\big) \end{aligned}$$

with $\mathtt{max}_\mathtt{k}(s)$ a function that returns the k $\mathtt{add}(n,v)$ operations with largest values (n,v) for different values of n and elements ordered using the total order defined previously. $\mathtt{fun}^{\mathtt{max}}$ can be defined by assigning a monotonic integer to the result computed in each replica, and using this integer to decide which value is the most recent.

$$
\begin{array}{llcl}
\text{Replica state} & \Sigma & = & \big(\mathcal{P}(\texttt{op}), \mathbb{I} \rightarrow V\big) \\
\text{Initial state} & \sigma_i^0 & = & \big(\{\}, i \rightarrow V_0\big) \\
\text{Update op at replica } i & \texttt{op}_i\big((s,m)\big) & = & \big(s \cup \{\texttt{op}\}, m\big) \\
\text{Read at replica } i & \texttt{op}_i\big((s,m)\big) & = & \texttt{fun}\big(s \bigcup\limits_{\forall j} m[j]\big) \\
\text{State to send from replica } i & \texttt{diff}\big(s,m\big) & = & (\{o \in s : \texttt{fun}\big(o \bigcup\limits_{\forall j} m[j]\big) \neq \texttt{fun}\big(\bigcup\limits_{\forall j} m[j]\big)\}, m[i \rightarrow \texttt{fun}(s \bigcup\limits_{\forall j} m[j])]\big) \\
\text{Merge replica states} & \mathsf{deliver}\,\big((s,m),(s',m')\big) & = & \big(s \cup s', m[j \mapsto \texttt{fun}^{\texttt{max}}(m[j], m'[j])]\forall j\big)
\end{array}
$$

Figure 3: Generic replicated data type for partially incremental computation.

This design enforces eventual consistency, assuming that replicas continue synchronizing until they reach an equivalent state, i.e., a state where read operations return the same result in every replica. However, this may not happen after the first synchronization step. For example, consider a top-1 object replicated in two sites: Site 1 executed operations $\{\texttt{add}(b,15), \texttt{add}(a,10)\}$ and site 2 executed operations $\{\texttt{add}(b,16), \texttt{add}(c,12)\}$. The two sites synchronize, with the top-1 element, $(b,16)$, being known at both replicas. After this, $\texttt{del}(b)$ executes at site 1, promoting $(a,10)$ to the top at site 1. After the propagation of $\texttt{del}(b)$ to site 2, $(c,12)$ is promoted to the top at site 2. After the next synchronization step, the top at site 1 $(a,10)$ is replaced by the same value as in site 2 $(c,12)$.

6. Final remarks

In this paper we have proposed three generic designs for replicated data types that perform a computation on the operations executed by users. These designs can be used in a system that maintains CRDT replicas at multiple sites and synchronizes them using a state-based model. We present the properties that computations must obey in order to use each of the designs. These designs try to minimize the information that each replica has to maintain and propagate to other replicas for synchronization.

The last proposed design departs from the strict CRDT state-based model, while still enforcing eventual consistency. We are currently formalizing the new model and studying the relations between replicated data types implemented using this design and state-based CRDTs that implement the same functionality. In the future, we intend to study how to integrate these designs in an eventually consistent cloud database, such as Riak.

Acknowledgments

This research is supported in part by FCT scholarship SFRH / BD / 65070 / 2009, FCT projects PTDC/ EEI-SCR/ 1837/ 2012 and PEst-OE/ EEI/ UI0527/ 2014 and EU FP7 SyncFree project (609551).

References

[1] P. S. Almeida, A. Shoker, and C. Baquero. Efficient state-based crdts by delta-mutation. In *Proc. of the Third International Conference on Networked Systems (NETYS) (to appear)*, May 2015.

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

[3] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *Proc. VLDB Endow.*, 7(13):1441–1451, Aug. 2014.

[4] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, 2014. ACM.

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, et. al. Spanner: Google's globally-distributed database. In *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, 2012. USENIX Association.

[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 *Proc. of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, 2007. ACM.

[7] 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 *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, 2011. ACM.

[8] C. Meiklejohn and P. Van Roy. Lasp: A Language for Distributed, Eventually Consistent Computations with CRDTs. In *Proc. of the Workshop on Principles and Practice of Consistency for Distributed Data*, Apr. 2015.

[9] D. Navalho, S. Duarte, N. Preguiça, and M. Shapiro. Incremental stream processing using computational conflict-free replicated data types. In *Proc. of the 3rd International Workshop on Cloud Data and Platforms*, CloudDP '13, pages 31–36, 2013. ACM.

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

[11] J. Yick, B. Mukherjee, and D. Ghosal. Wireless sensor network survey. *Comput. Netw.*, 52(12):2292–2330, Aug. 2008.

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

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 weak consistency models, applications in different data centers execute operations concurrently over the same set of data leading to temporary divergence between replicas and potentially counter-intuitive 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 centers. Data centers 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 is

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

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 center 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. 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

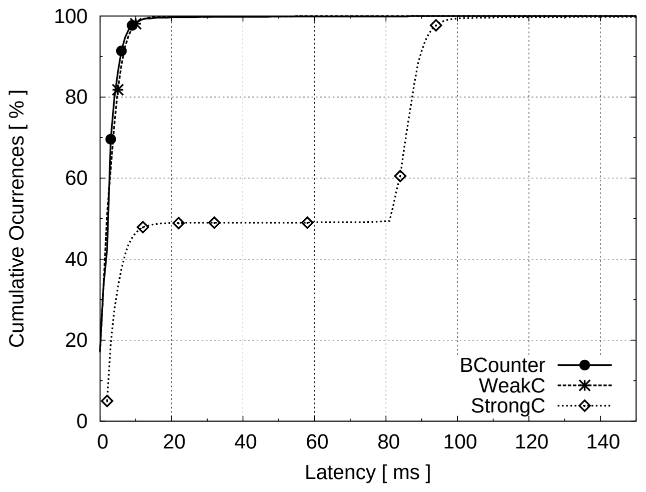


Fig. 1. CDF of latency with a single counter.

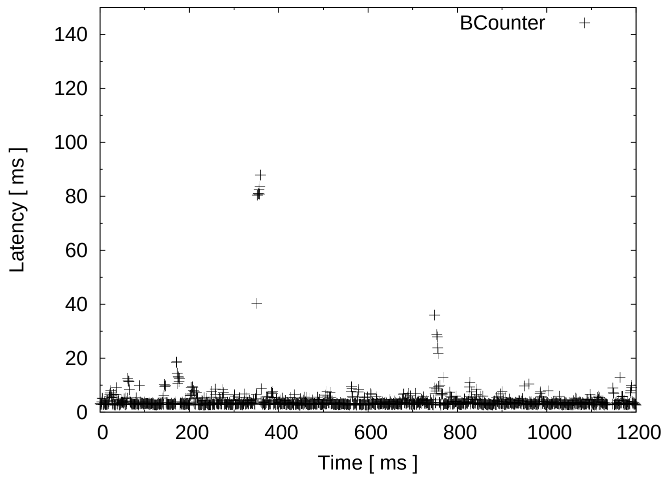


Fig. 2. Latency measured over time.

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.

I 'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems (Submitted)

’Cause I’m Strong Enough: Reasoning about Consistency Choices in Distributed Systems

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro

Abstract

Large-scale distributed systems often rely on replicated databases that allow a programmer to request different data consistency guarantees for different operations, and thereby control their performance. Using such databases is far from trivial: requesting stronger consistency in too many places may hurt performance, and requesting it in too few places may violate correctness. To help programmers in this task, we propose the first proof rule for establishing that a particular choice of consistency guarantees for various operations on a replicated database is enough to ensure the preservation of a given data integrity invariant. Our rule is modular: it allows reasoning about the behaviour of every operation separately under some assumption on the behaviour of other operations. This leads to simple reasoning, which we have automated in an SMT-based tool. We present a nontrivial proof of soundness of our rule and illustrate its use on several examples.

1. Introduction

To achieve availability and scalability, many modern distributed systems rely on *replicated databases*, which maintain multiple *replicas* of shared data. Clients can access the data at any of the replicas, and these replicas communicate changes to each other using message passing. For example, large-scale Internet services use data replicas in geographically distinct locations, and applications for mobile devices keep replicas locally to support offline use. Ideally, we would like replicated databases to provide *strong consistency*, i.e., to behave as if a single centralised replica handles all operations. However, achieving this ideal usually requires synchronisation among replicas, which slows down the database and even makes it unavailable if network connections between replicas fail [3, 24].

For this reason, modern replicated databases often eschew synchronisation completely; such databases are commonly dubbed *eventually consistent* [44]. In these databases, a replica performs an operation requested by a client locally without any synchronisation with other replicas and immediately returns to the client; the *effect* of the operation is propagated to the other replicas only *eventually*. This may lead to *anomalies*—behaviours deviating from strong consistency. One of them is illustrated in Figure 1(a). Here Alice makes a post while connected to a replica r_1, and Bob, also connected to r_1, sees the post and comments on it. After each of the two operations, r_1 sends a message to the other replicas in the system with the update performed by the user. If the messages with the updates by Alice and Bob arrive to another replica r_2 out of order, then Carol, connected to r_2, may end up seeing Bob’s comment, but not Alice’s post it pertains to. The *consistency model* of a replicated database restricts the anomalies that it exhibits. For example, the model of *causal consistency* [31], which we consider in this paper, disallows the anomaly in Figure 1(a), yet can be implemented without any synchronisation. The model ensures that all replicas in the system see *causally dependent* events, such as the posts by Alice and Bob, in the order in which they happened. However, causal consistency allows different replicas to see causally *independent* events as occurring in different orders. This is illustrated in Figure 1(b), where Alice and Bob concurrently make posts at r_1 and r_2. Carol, connected to r_3 initially sees Alice’s post, but not Bob’s, and Dave, connected to r_4, sees Bob’s post, but not Alice’s. This outcome cannot be obtained by executing the operations in any total order and, hence, deviates from strong consistency.

Such anomalies related to the ordering of actions are often acceptable for applications. What is not acceptable is to violate crucial well-formedness properties of application data, called *integrity invariants*. Consistency models that do not require any synchronisation are often too weak to ensure these. For example, consider a toy banking application where the database stores the balance of a single account that clients can make deposits to and withdrawals from. In this case, an integrity invariant may require the account balance to be always non-negative. Consider the database computation in Figure 1(c), allowed by causal consistency. Initially all replicas store the same balance of 100. Alice and Bob, connected to r_1 and r_2, both withdraw 100, thinking that there are sufficient funds available. Once the two replicas exchange the updates, the balance becomes -100, violating the integrity invariant. To ensure the integrity invariant in this example, we have to introduce synchronisation between replicas, and, since synchronisation is expensive, we would like to introduce it sparingly. To allow this, some research [10, 30, 40, 41] and commercial [7, 11, 33] databases now provide *hybrid* consistency models that allow the programmer to request stronger consistency for certain operations and thereby introduce synchronisation. For example, a consistency model may execute some operations under causal consistency, and some under strong consistency [30]. To preserve the integrity invariant in our banking application when using this model, only withdrawal operations need to use strong consistency, and hence, synchronise to ensure that the account is not overdrawn; deposit operations may use causal consistency and hence proceed without synchronisation. Requesting stronger consistency in hybrid models is similar to the use of fences in weak memory models of shared-memory multiprocessors and programming languages [12] (see §7 for a comparison).

Even though hybrid consistency models allow the programmer to fine-tune consistency level, using these models effectively is far from trivial. Requesting stronger consistency in too many places may hurt performance and availability, and requesting it in too few places may violate correctness. Striking the right balance requires the programmer to reason about the application behaviour on the subtle semantics of the consistency model, understanding which anomalies are disallowed by a particular consistency strengthening and whether disallowing these anomalies is enough to ensure correctness. This difficulty is compounded by the perennial challenge of reasoning about concurrency, present even with strong consistency—having to consider the huge number of possible interactions between concurrently executing operations.

To help programmers exploit hybrid consistency models, we propose the first proof rule and tool for proving integrity invariants of applications using replicated databases with a range of hybrid models. In more detail, our first contribution is a generic hybrid consistency model (§2) that is flexible enough to encode a variety

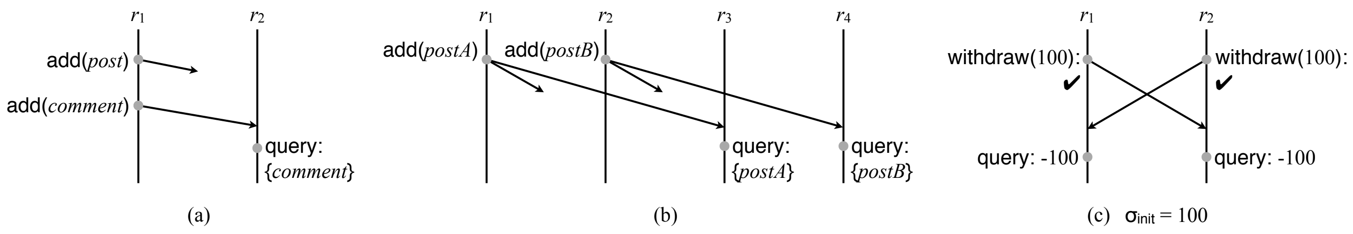


Figure 1. Illustrations of replicated database computations.

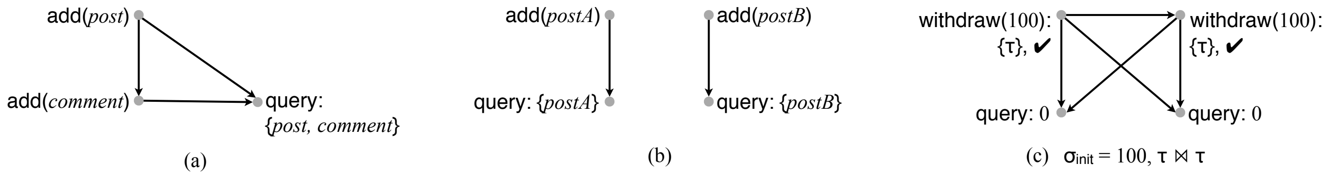


Figure 2. Examples illustrating Definition 1. We omit return values when they are $\perp$ and token sets when they are empty.

of consistency models for replicated databases proposed in the literature [10, 30, 31, 40]. It guarantees causal consistency by default and allows the programmer to additionally specify which pairs of operations may not execute without synchronisation by means of a special *conflict relation*. For example, to ensure the non-negativity of balances in the banking application, the conflict relation may require any pair of withdrawals to synchronise, so that one of them was aware of the effect of the other. This is equivalent to executing withdrawals under strong consistency. In general, different instances of the conflict relation correspond to different interfaces for strengthening consistency proposed in the literature. Our proof rule is developed for the generic consistency model and, hence, applies to existing models that can be represented as its instantiations. We specify our consistency model formally (§3) using the approach previously proposed for specifying variants of eventual consistency [16]. In this approach, a database computation is denoted by a partial order on client operations, representing causality, and the conflict relation imposes additional constraints on this order.

Our next, and key, technical contribution is a proof rule for showing that a set of operations preserves a given integrity invariant when executed on our consistency model with a given choice of conflict relation (§4). For example, we can prove that withdrawals and deposits preserve the non-negativity of balances when executed with the conflict relation described above. To avoid explicit reasoning about all possible interactions between operations, our proof rule is *modular*: it allows us to reason about the behaviour of every operation separately under some assumption on the behaviour of other operations, which takes into account the conflict relation. In this way, our proof rule allows the programmer to reason precisely about how strengthening or weakening consistency of certain operations affects correctness.

The modular nature of our proof rule allows it to reason in terms of states of a single database copy, just like in proof rules for strongly consistent shared-memory concurrency. We have proved that this simple reasoning is sound, despite the weakness of the consistency model (§5). As part of this proof we have identified a more general *event-based* rule that reasons directly in terms of partial orders on events representing database computations, instead of database states that these events lead to. The soundness of the original *state-based* rule is proved by compiling it into the event-based one. In this way, the event-based rule explicates the reasons for the soundness of the state-based rule.

We have also developed a tool that automates our proof rule by reducing checking its obligations to SMT queries (§6). Using the tool, we have verified several example applications that require strengthening consistency in nontrivial ways. These include an extension of the above banking application, an online auction service and a course registration system. In particular, we were able to handle applications using *replicated data types* (aka CRDTs [38]), which encapsulate policies for automatically merging the effects of operations performed without synchronisation at different replicas. The fact that we can reduce checking the correctness properties of complex computations in our examples to querying off-the-shelf SMT tools demonstrates the simplicity of reasoning required by our approach.

2. Consistency Model, Informally

We start by presenting our generic consistency model. Even though this model is not implemented in its full generality by an existing database, it can encode a variety of models that have in fact been implemented. In this section we present the programming interface of our consistency model and describe its semantics informally, from an operational perspective. We give a formal semantics in §3.

2.1 Causal Consistency and Its Implementation

Our hybrid model guarantees at least *causal consistency* [31], already mentioned in §1. We therefore start by presenting informally how a typical implementation of a causally consistent database operates. Let State be the set of possible states of the data managed by the database system. We denote states by σ and let σ_{init} be a distinguished initial state. Applications define a set of operations $\mathsf{Op} = \{o, \dots\}$ on the data and interact with the database by issuing these operations. For simplicity, we assume that an operation always terminates and returns a single value from a set Val. We use a value $\perp \in \mathsf{Val}$ to model operations that return no value. We do not consider operation parameters, since these can be part of the operation name.

The database implementation consists of a set of replicas, each maintaining a complete copy of the database state; we identify the replicas by $r_1, r_2, \dots$ For the purposes of the informal explanation, we assume that replicas never fail. A client operation is initially executed at a single replica, which we refer to as its *origin* replica. At this replica, the execution of the operation is not interleaved with that of others. This execution updates the replica state determinis-

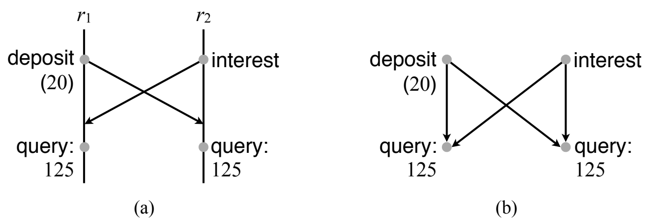


Figure 3. (a) An illustration of a database computation; (b) the corresponding execution of Definition 1. We assume $\sigma_{\mathsf{init}} = 100$.

tically, and immediately returns a value to the client. After this, the replica sends a message to all other replicas containing the *effect* of the operation, which describes the updates done by the operation to the database state. The replicas are guaranteed to receive the message at most once. Upon receipt, the replicas apply the effect to their state.

In this paper, we abstract from a particular language in which operations may be written and assume that their semantics is given by a function

$$\mathcal{F} \in \mathsf{Op} \to (\mathsf{State} \to (\mathsf{Val} \times (\mathsf{State} \to \mathsf{State}))). \tag{1}$$

To aid readability, for $o \in \mathsf{Op}$ we write $\mathcal{F}_o$ instead of $\mathcal{F}(o)$ and let

$$\forall o, \sigma.\ \mathcal{F}_o(\sigma) = (\mathcal{F}^{\mathsf{val}}_o(\sigma), \mathcal{F}^{\mathsf{eff}}_o(\sigma)).$$

Given a state σ of o's origin replica, $\mathcal{F}^{\mathsf{val}}_o(\sigma) \in \mathsf{Val}$ determines the return value of the operation and $\mathcal{F}^{\mathsf{eff}}_o(\sigma) \in \mathsf{State} \to \mathsf{State}$ its effect. The latter is a function, to be applied by every replica to its state to incorporate the operation's effect: immediately at the origin replica, and after receiving the corresponding message at all other replicas.

For example, states in the toy banking application of §1 are integers, representing the account balance: $\mathsf{State} = \mathbb{Z}$. We define the semantics of operations for depositing an amount $a > 0$, accruing a 5% interest and querying the balance:

$$\begin{aligned} \mathcal{F}_{\mathsf{deposit}(a)}(\sigma) &= (\bot, (\lambda\sigma'.\, \sigma' + a)); \\ \mathcal{F}_{\mathsf{interest}}(\sigma) &= (\bot, (\lambda\sigma'.\, \sigma' + 0.05 * \sigma)); \\ \mathcal{F}_{\mathsf{query}}(\sigma) &= (\sigma, \mathsf{skip}), \end{aligned} \tag{2}$$

where $\mathsf{skip} = (\lambda\sigma'.\, \sigma')$. Figure 3(a) illustrates a database computation involving these operations. Note that interest first computes the interest $0.05 * \sigma$ based on the balance σ at the origin replica; its effect then adds the resulting amount to the balance at each replica. In particular, in Figure 3(a) interest at r_2 does not take into account the deposit made at r_1. This behaviour is the price to pay for avoiding synchronisation between replicas. The good news is that, once the replicas r_1 and r_2 exchange the effects of deposit and interest, they converge to the same balance, which is returned by the query operations.

Such convergence is not guaranteed for arbitrary operations. For example, we could implement interest so that its effect multiplied the balance by 1.05 at each replica where it is applied:

$$\mathcal{F}^{\mathsf{eff}}_{\mathsf{interest}}(\sigma) = (\lambda\sigma'.\, (1.05 * \sigma')). \tag{3}$$

In the scenario in Figure 3(a), this would lead the query operations to return different values, 126 at r_1 and 125 at r_2. In this case, even after all messages are delivered, replicas end up in different states. This is undesirable for database users: we would like the implementation to be *convergent*, i.e., such that two replicas that see the same set of operations are in the same state. In particular, if users stop performing updates to the database, then once all outstanding messages are delivered, all replicas should reach the same state [44]. To ensure convergence, for now we require that the effects of all operations commute (we relax this condition slightly in §2.2):

$$\forall o_1, o_2, \sigma_1, \sigma_2.\ \mathcal{F}^{\mathsf{eff}}_{o_1}(\sigma_1) \circ \mathcal{F}^{\mathsf{eff}}_{o_2}(\sigma_2) = \mathcal{F}^{\mathsf{eff}}_{o_2}(\sigma_2) \circ \mathcal{F}^{\mathsf{eff}}_{o_1}(\sigma_1). \tag{4}$$

For example, this condition holds of the effects defined by (2). The requirement of commutativity is not very taxing: as we elaborate in §6, to satisfy (4), programmers can exploit ready-made *replicated data types* (aka CRDTs [38]). These encapsulate commutative implementations of policies for merging concurrent updates to the database.

As we explained in §1, asynchronous operation processing may lead to anomalies, and causal consistency disallows some of them. It ensures that message propagation between replicas is *causal*: if a replica sends a message containing the effect of an operation o_2 after it sends or receives a message containing the effect of an operation o_1, then no replica will receive the message about o_2 before the it receives the one about o_1. In this case we say that the invocation of o_2 *causally depends* on that of o_1. Causal propagation disallows the computation in Figure 1(a), but allows the one in Figure 1(b).

2.2 Strengthening Consistency

The guarantees provided by causal consistency are too weak to ensure certain integrity invariants. For example, in our banking application we would like the state at each replica to satisfy the invariant

$$I = \{\sigma \mid \sigma \geq 0\}. \tag{5}$$

To ensure this, an operation for withdrawing an amount $a > 0$ could check whether the account has sufficient funds and return ✓ or ✗ depending on the result:

$$\mathcal{F}_{\mathsf{withdraw}(a)}(\sigma) = \text{if } \sigma \geq a \text{ then } (\checkmark, (\lambda\sigma'.\, \sigma' - a)) \text{ else } (\text{✗}, \mathsf{skip}).$$

This is enough to maintain the invariant when all operations are processed at the same replica, but not when they are processed asynchronously at different replicas. This is illustrated by the computation in Figure 1(c), already explained in §1.

The problem in this example arises because two particular operations update the database concurrently, without being aware of each other. To address this, our consistency model allows the programmer to strengthen causal consistency by specifying explicitly which operations may not be executed in this way. Namely, the model is parameterised by a *token system* $\mathcal{T} = (\mathsf{Token}, \bowtie)$, consisting of a set of *tokens* Token and a symmetric *conflict relation* $\bowtie \subseteq \mathsf{Token} \times \mathsf{Token}$. Tokens are ranged over by τ and their sets, by T. For sets T_1 and T_2 of tokens we let $T_1 \bowtie T_2$ if there exists a pair of conflicting tokens coming from these sets: $\exists \tau_1 \in T_1.\, \exists \tau_2 \in T_2.\, \tau_1 \bowtie \tau_2$.

Each operation may acquire a set of tokens. To account for this, we redefine the type of $\mathcal{F}$ in (1) as

$$\mathcal{F} \in \mathsf{Op} \to (\mathsf{State} \to (\mathsf{Val} \times (\mathsf{State} \to \mathsf{State}) \times \mathcal{P}(\mathsf{Token}))) \tag{6}$$

and let

$$\forall o, \sigma.\, \mathcal{F}_o(\sigma) = (\mathcal{F}^{\mathsf{val}}_o(\sigma), \mathcal{F}^{\mathsf{eff}}_o(\sigma), \mathcal{F}^{\mathsf{tok}}_o(\sigma)).$$

Thus, $\mathcal{F}^{\mathsf{tok}}_o(\sigma) \in \mathcal{P}(\mathsf{Token})$ gives the set of tokens acquired by the operation o when executed in the state σ. Informally, our consistency model guarantees that operations that acquire tokens conflicting according to $\bowtie$ have to be causally dependent one way or another: the origin replica of one operation must have incorporated the effect of the other by the time the former operation executes. Ensuring this in implementations requires replicas to synchronise [10, 30].

In our consistency model, we can guarantee the preservation of invariant (5) in the banking application by defining operation se-

$$
\begin{aligned}
\mathsf{Token} &= \{\tau\}\\
\bowtie &= \{(\tau,\tau)\}\\
\mathcal{F}_{\mathsf{deposit}(a)}(\sigma) &= (\bot, (\lambda\sigma'.\, \sigma' + a), \emptyset)\\
\mathcal{F}_{\mathsf{interest}}(\sigma) &= (\bot, (\lambda\sigma'.\, \sigma' + 0.05 * \sigma), \emptyset)\\
\mathcal{F}_{\mathsf{query}}(\sigma) &= (\sigma, \mathsf{skip}, \emptyset)\\
\mathcal{F}_{\mathsf{withdraw}(a)}(\sigma) &= \text{if } \sigma \geq a \text{ then } (\checkmark, (\lambda\sigma'.\, \sigma' - a), \{\tau\})\\
&\quad\ \text{else } (\text{✗}, \mathsf{skip}, \{\tau\})
\end{aligned}
$$

Figure 4. Operation semantics for the banking application. Note that $a > 0$.

mantics as in Figure 4. Thus, withdraw acquires a token τ conflicting with itself, and all other operations do not acquire any tokens. Then the scenario in Figure 1(c) cannot happen: one withdrawal would have to be aware of the other and would therefore fail. However, deposits and interest accruals can be causally independent with all operations, and replicas can therefore execute them without any synchronisation [10, 30]. In this example, the token τ is analogous to a mutual exclusion lock in shared-memory concurrency. Our proof method (§4) establishes that this use of the token is indeed sufficient to preserve the integrity invariant (5).

Since operations acquiring conflicting tokens have to be causally dependent, causal message propagation (§2.1) ensures that all replicas see such operations in the same order. This allows us to weaken (4) to require commutativity only for operations that do not acquire conflicting tokens:

$$
\begin{aligned}
&\forall o_1, o_2, \sigma_1, \sigma_2.\, (\mathcal{F}^{\mathsf{tok}}_{o_1}(\sigma_1) \bowtie \mathcal{F}^{\mathsf{tok}}_{o_2}(\sigma_2)) \vee\\
&\qquad (\mathcal{F}^{\mathsf{eff}}_{o_1}(\sigma_1) \circ \mathcal{F}^{\mathsf{eff}}_{o_2}(\sigma_2) = \mathcal{F}^{\mathsf{eff}}_{o_2}(\sigma_2) \circ \mathcal{F}^{\mathsf{eff}}_{o_1}(\sigma_1)). \qquad (7)
\end{aligned}
$$

As we show in §3, this is sufficient to ensure the property of convergence that we introduced in §2.1. For example, the operations in Figure 4 satisfy (7). Furthermore, if all operations except query acquired the token τ, then we would be able to implement interest by the effect given by (3) without compromising convergence.

3. Formal Semantics

We now formally define the semantics of our consistency model, i.e., the set of all client-database interactions it allows. To keep the presentation as simple as possible, we define the semantics declaratively: our formalism does not refer to implementation-level concepts, such as replicas or messages, even though we do use these concepts in informal explanations. We build on an approach previously used to specify forms of eventual consistency [16]. Namely, our denotations of database computations consist of a set of events, representing operation invocations by clients, and a relation on events, describing abstractly how the database processes the corresponding operations.

Assume a countably infinite set Event of *events*, ranged over by e, f, g. A relation is a *strict partial order* if it is transitive and irreflexive. For a relation R we write $(e, f) \in R$ and $e \xrightarrow{R} f$ interchangeably.

DEFINITION 1. *Given a token system $\mathcal{T} = (\mathsf{Token}, \bowtie)$, an* ***execution*** *is a tuple $X = (E, \mathsf{oper}, \mathsf{rval}, \mathsf{tok}, \mathsf{hb})$, where:*

- *E is a finite subset of* Event*;*
- oper : $E \to$ Op *gives the operation whose invocation a given event denotes;*
- rval : $E \to$ Val *gives the return value of the operation;*
- tok : $E \to \mathcal{P}(\mathsf{Token})$ *gives the set of tokens acquired by the operation;*
- hb $\subseteq E \times E$*, called* ***happens-before****, is a strict partial order such that*

$$\forall e, f \in E.\, \mathsf{tok}(e) \bowtie \mathsf{tok}(f) \implies (e \xrightarrow{\mathsf{hb}} f \vee f \xrightarrow{\mathsf{hb}} e). \quad (8)$$

Operationally, each event represents an invocation of an operation at its origin replica. The applications of the operation's effect at other replicas are not recorded in an execution explicitly. Instead, the happens-before relation records causal dependencies between operations arising from such applications: $e \xrightarrow{\mathsf{hb}} f$ means that either the operations denoted by e and f were executed at the same replica in this order, or they were executed at different replicas and the message containing the effect of e had been delivered to the replica performing f before f was executed. Hence, if we have $e \xrightarrow{\mathsf{hb}} f$, then the effect of e is incorporated into the state to which f is applied and may influence its return value. We give examples of executions in Figures 2 and 3(b). The ones in Figures 2(b) and 3(b) model the computations of the database informally illustrated in Figures 1(b) and 3(a), respectively.

The transitivity of hb in Definition 1 reflects the guarantee of causal message propagation in implementations explained in §2.1 [16]. For example, in the execution of Figure 2(a), the transitivity of hb mandates the edge between the addition of a post and the query (cf. Figure 1(a)). The condition (8) formalises the stronger consistency guarantee provided by tokens: operations acquiring conflicting tokens have to be causally dependent. For example, since the two withdraw operations in Figure 2(c) acquire a token τ with $\tau \bowtie \tau$, they have to be related by happens-before. Finally, we require executions to contain only finitely many events, because in this paper we are only concerned with safety properties of applications.

We write $\mathsf{Exec}(\mathcal{T})$ for the set of all executions over the token system $\mathcal{T}$. In the following, we denote components of X and similar structures as in $X.E$. We let X_{init} be the unique execution with $X_{\mathsf{init}}.E = \emptyset$.

We now define the semantics of our consistency model as the set of all executions $X \in \mathsf{Exec}(\mathcal{T})$ over a token system $\mathcal{T}$ whose return values $X.\mathsf{rval}$ and token sets $X.\mathsf{tok}$ are computed using $\mathcal{F}$ as informally described in §2. To define this set, we first let the *context* of an event e in an execution X be

$$\mathsf{ctxt}(e, X) = (E, (X.\mathsf{oper})|_E, (X.\mathsf{rval})|_E, (X.\mathsf{tok})|_E, (X.\mathsf{hb})|_E),$$

where $E = (X.\mathsf{hb})^{-1}(e)$ and $\cdot|_E$ is the restriction to events in E. Operationally-speaking, the context consists of those events whose effects have been incorporated into the state of the replica where the operation $X.\mathsf{oper}(e)$ executes; it is these events that influence the outcomes of e—the return value $X.\mathsf{rval}(e)$ and the token set $X.\mathsf{tok}(e)$. For example, the context of each of the query events in Figure 3(b) consists of the deposit and interest events. This reflects the events that the corresponding replica has seen before executing query in Figure 3(a).

It is technically convenient for us to initially formulate definitions without assuming effect commutativity (7). In this case, $X.\mathsf{rval}(e)$ and $X.\mathsf{tok}(e)$ are not determined by $\mathsf{ctxt}(e, X)$ uniquely. In operational terms, this is because the state that a replica will be in after seeing the events in $\mathsf{ctxt}(e, X)$ depends on the order in which the replica finds out about these events: although causal message propagation ensures that messages about causally dependent events in $\mathsf{ctxt}(e, X)$ will be delivered to the replica in the order consistent with $X.\mathsf{hb}$, messages about causally independent events may be delivered in arbitrary order. We therefore first define a function

$$\mathsf{eval}^{\dagger}_{\mathcal{F}} : \mathsf{Exec}(\mathcal{T}) \to \mathcal{P}(\mathsf{State})$$

that yields the *set* of all possible states that a replica may end up in after seeing the events in a given execution, such as $\mathsf{ctxt}(e, X)$.

For an execution Y, we define $\mathsf{eval}^{\dagger}_{\mathcal{F}}(Y)$ inductively on the size of $Y.E$. If $Y.E = \emptyset$, then $\mathsf{eval}^{\dagger}_{\mathcal{F}}(Y) = \{\sigma_{\mathsf{init}}\}$. Otherwise,

$$\mathsf{eval}^{\dagger}_{\mathcal{F}}(Y) = \{\mathcal{F}^{\mathsf{eff}}_{Y.\mathsf{oper}(e)}(\sigma')(\sigma) \mid e \in \max(Y) \wedge \sigma \in \mathsf{eval}^{\dagger}_{\mathcal{F}}(Y|_{Y.E-\{e\}}) \wedge \sigma' \in \mathsf{eval}^{\dagger}_{\mathcal{F}}(\mathsf{ctxt}(e, Y))\},$$

where

$$\max(Y) = \{e \in Y.E \mid \neg\exists f \in Y.E.\, (e, f) \in Y.\mathsf{hb}\}. \qquad (9)$$

Thus, to compute $\mathsf{eval}^{\dagger}_{\mathcal{F}}(Y)$ for a non-empty Y, we choose an hb-maximal event e in Y. Operationally, this is the event whose effect is incorporated last by the replica r whose state we are determining. We then pick a state σ that r could be in right before incorporating the effect of e. The set of such states is obtained by invoking $\mathsf{eval}^{\dagger}_{\mathcal{F}}$ on the execution $Y|_{Y.E-\{e\}}$, describing the events r knew about when it incorporated e. To determine the effect of e's operation, we pick a state σ' that the replica r' that generated e could be in at the time of this generation. The set of such states is computed by invoking $\mathsf{eval}^{\dagger}_{\mathcal{F}}$ on the execution $\mathsf{ctxt}(e, Y)$, describing the events that replica r' knew about when it generated e. Then the effect of e's operation is $\mathcal{F}^{\mathsf{eff}}_{Y.\mathsf{oper}(e)}(\sigma')$, and we determine the state of the replica r after e by applying this effect to the state σ: $\mathcal{F}^{\mathsf{eff}}_{Y.\mathsf{oper}(e)}(\sigma')(\sigma)$.

To illustrate $\mathsf{eval}^{\dagger}_{\mathcal{F}}$, consider the execution Y consisting of the deposit and interest events in Figure 3(b) and the operation semantics $\mathcal{F}$ in Figure 4. Recall that in this case $\sigma_{\mathsf{init}} = 100$. We can evaluate Y in two ways, corresponding to the orders in which replicas r_1, respectively r_2, apply the effects of the events in the computation in Figure 3(a):

$$\begin{aligned}\mathsf{eval}^{\dagger}_{\mathcal{F}}(Y) &= \{\mathcal{F}^{\mathsf{eff}}_{\mathsf{interest}}(\sigma_{\mathsf{init}})(\mathcal{F}^{\mathsf{eff}}_{\mathsf{deposit}(20)}(\sigma_{\mathsf{init}})(\sigma_{\mathsf{init}})),\\ &\qquad \mathcal{F}^{\mathsf{eff}}_{\mathsf{deposit}(20)}(\sigma_{\mathsf{init}})(\mathcal{F}^{\mathsf{eff}}_{\mathsf{interest}}(\sigma_{\mathsf{init}})(\sigma_{\mathsf{init}}))\}\\ &= \{100 + 20 + 5, 100 + 5 + 20\} = \{125\}.\end{aligned}$$

Both ways of evaluation lead to the same outcome. This would not be the case if we used a function $\mathcal{F}'$ identical to $\mathcal{F}$, but with the effect of interest defined by (3), which violates (7). In this case,

$$\mathsf{eval}^{\dagger}_{\mathcal{F}'}(Y) = \{100 + 20 + 6, 100 + 5 + 20\} = \{126, 125\},$$

which corresponds to the diverging database computation we explained in §2.1.

We note that, for notational convenience, $\mathsf{eval}^{\dagger}_{\mathcal{F}}$ takes as a parameter a whole execution including return values (rval) and token sets (tok) associated with its events. However, the function as we defined it does not depend on these: the state is determined solely based on the operations performed (oper) and happens-before relationships among them (hb).

DEFINITION 2. *An execution $X \in \mathsf{Exec}(\mathcal{T})$ is **consistent** with $\mathcal{T}$ and $\mathcal{F}$, denoted $X \models \mathcal{T}, \mathcal{F}$, if*

$$\forall e \in X.E.\, \exists \sigma \in \mathsf{eval}^{\dagger}_{\mathcal{F}}(\mathsf{ctxt}(e, X)).\ (X.\mathsf{rval}(e) = \mathcal{F}^{\mathsf{val}}_{X.\mathsf{oper}(e)}(\sigma)) \wedge (X.\mathsf{tok}(e) = \mathcal{F}^{\mathsf{tok}}_{X.\mathsf{oper}(e)}(\sigma)).$$

We let $\mathsf{Exec}(\mathcal{T}, \mathcal{F}) = \{X \mid X \models \mathcal{T}, \mathcal{F}\}$ be the set of executions allowed by our consistency model.

PROPOSITION 3.

$$\forall X \in \mathsf{Exec}(\mathcal{T}, \mathcal{F}).\, \forall e \in X.E.\, (\mathsf{ctxt}(e, X) \in \mathsf{Exec}(\mathcal{T}, \mathcal{F})).$$

Operationally, $X \models \mathcal{T}, \mathcal{F}$ means that the outcomes in X can be produced by the database implementation sketched in §2 with some order of message delivery. The executions in Figures 2 and 3(b) are consistent with the parameters in Figure 4 or the expected semantics of operations on posts and comments. In particular, the execution in Figure 2(c) is consistent because the context of the right-hand-side withdraw includes the left-hand-side withdraw. Evaluating this context yields a zero balance, which causes the right-hand-side withdraw to generate skip as its effect.

LEMMA 4. *If $X \models \mathcal{T}, \mathcal{F}$, then $\mathsf{eval}^{\dagger}_{\mathcal{F}}(X)$ is a singleton set. Furthermore, so is $\mathsf{eval}^{\dagger}_{\mathcal{F}}(\mathsf{ctxt}(e, X))$ for any $e \in X.E$.*

The lemma shows that in Definition 2 it does not matter how we choose the order of evaluation in $\mathsf{eval}^{\dagger}_{\mathcal{F}}$. When viewed operationally, this independence implies the convergence property from §2.1: two replicas that see the same events will end up in the same state. The proof of Lemma 4, given in [2, §A], exploits properties (7) and (8). This proof is subtle because (7) does not require commutativity for the effects of pairs of operations that acquire conflicting tokens.

Motivated by Lemma 4, we define the evaluation of consistent executions

$$\mathsf{eval}_{\mathcal{F}} : \mathsf{Exec}(\mathcal{T}, \mathcal{F}) \to \mathsf{State}$$

as follows: $\mathsf{eval}_{\mathcal{F}}(X)$ is the unique σ such that $\mathsf{eval}^{\dagger}_{\mathcal{F}}(X) = \{\sigma\}$.

To illustrate the flexibility of our consistency model, we show how it can represent some of the existing models; we provide more instantiations in §6 and [2, §B].

Causal consistency [17, 31] is the baseline model we obtain without using any tokens: $\mathsf{Token} = \emptyset$ and $\forall o, \sigma.\, \mathcal{F}^{\mathsf{tok}}_{o}(\sigma) = \emptyset$. Then (8) is a tautology and (7) is equivalent to (4), so that all effects have to commute.

Sequential consistency [27] is a form of strong consistency and the strongest consistency model we can obtain from ours. It requires every operation to acquire a mutual exclusion token:

$$\mathsf{Token} = \{\tau\}; \quad \bowtie = \{(\tau, \tau)\}; \quad \forall o, \sigma.\, \mathcal{F}^{\mathsf{tok}}_{o}(\sigma) = \{\tau\}.$$

Then in any execution $X \in \mathsf{Exec}((\mathsf{Token}, \bowtie), \mathcal{F})$, the happens-before $X.\mathsf{hb}$ is total, and each event in X is aware of the effects of all events preceding it in $X.\mathsf{hb}$.

RedBlue consistency [30] is a hybrid consistency model that classifies operations as either *red* or *blue*: $\mathsf{Op} = \mathsf{Op}_r \uplus \mathsf{Op}_b$. Red operations are guaranteed sequential consistency, and blue operations, only causal consistency. To express this in our model, we again use a mutual exclusion token: $\mathsf{Token} = \{\tau\}$ and $\bowtie = \{(\tau, \tau)\}$. Red operations acquire τ, and blue operations acquire no tokens:

$$(\forall o \in \mathsf{Op}_r.\, \forall \sigma.\, \mathcal{F}^{\mathsf{tok}}_{o}(\sigma) = \{\tau\}) \wedge (\forall o \in \mathsf{Op}_b.\, \forall \sigma.\, \mathcal{F}^{\mathsf{tok}}_{o}(\sigma) = \emptyset).$$

Then red operations are totally ordered by happens-before, and blue ones are ordered only partially. The token assignment in our banking application (Figure 4) is an instance of the RedBlue consistency, where withdraw operations are red, and all others are blue.

4. State-based Proof Rule

We consider the following verification problem: given a token system $\mathcal{T} = (\mathsf{Token}, \bowtie)$, prove that operations $\mathcal{F}$ maintain an integrity invariant $I \subseteq \mathsf{State}$ over database states. Formally, we establish that any execution consistent with $\mathcal{T}$ and $\mathcal{F}$ evaluates to a state satisfying I:

$$\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathsf{eval}^{-1}_{\mathcal{F}}(I).$$

By Proposition 3 this implies that the return value of every event in an execution $X \in \mathsf{Exec}(\mathcal{T}, \mathcal{F})$ can be obtained by applying its operation to a state satisfying I:

$$\forall e \in X.E.\, \exists \sigma \in I.\, (X.\mathsf{rval}(e) = \mathcal{F}^{\mathsf{val}}_{X.\mathsf{oper}(e)}(\sigma)).$$

For example, we show that any execution consistent with Figure 4 evaluates to a state satisfying the invariant (5). Hence, a query operation will always return a non-negative balance.

$\exists G_0 \in \mathcal{P}(\mathsf{State} \times \mathsf{State}), G \in \mathsf{Token} \to \mathcal{P}(\mathsf{State} \times \mathsf{State})$
such that

S1. $\sigma_{\mathsf{init}} \in I$

S2. $G_0(I) \subseteq I \wedge \forall \tau.\ G(\tau)(I) \subseteq I$

S3. $\forall o, \sigma, \sigma'.\ (\sigma \in I \wedge (\sigma, \sigma') \in (G_0 \cup G((\mathcal{F}^{\mathsf{tok}}_o(\sigma))^\perp))^*)$
$\implies (\sigma', \mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma')) \in G_0 \cup G(\mathcal{F}^{\mathsf{tok}}_o(\sigma))$

$$\overline{\qquad\qquad \mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathsf{eval}^{-1}_{\mathcal{F}}(I) \qquad\qquad}$$

Figure 5. State-based proof rule for a token system $\mathcal{T} = (\mathsf{Token}, \bowtie)$. For $T \subseteq \mathsf{Token}$ we let $G(T) = \bigcup_{\tau \in T} G(\tau)$ and $T^\perp = \{\tau \mid \tau \in \mathsf{Token} \wedge \neg \exists \tau' \in T.\ \tau \bowtie \tau'\}$. We denote by R^* the reflexive and transitive closure of a relation R. For a relation $R \in \mathcal{P}(A \times B)$ and a predicate $P \in \mathcal{P}(A)$, the expression $R(P)$ denotes the image of P under R.

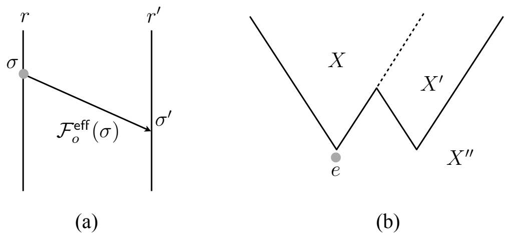


Figure 6. Graphical illustrations of (a) the state-based rule; and (b) the event-based rule.

The key challenge of the above verification problem is the need to consider infinitely many executions consistent with $\mathcal{T}$ and $\mathcal{F}$. Our main technical contribution is the proof rule for solving this problem that avoids considering all such executions explicitly. Instead, the proof rule is *modular* in that it allows us to reason about the behaviour of every operation separately. Our proof rule is also *state-based* in that it reasons in terms of states obtained by evaluating parts of executions or, from the operational perspective, in terms of replica states.

We give our proof rule in Figure 5 and explain it from the operational perspective. The rule assumes that the invariant I holds of the initial database state σ_{init} (condition S1). Consider a computation of the database implementation from §2 and a state σ of a replica r at some point in this computation. The proof rule assumes that $\sigma \in I$ and aims to establish that executing any operation o at r will preserve the invariant I. This is easy if we only consider how o's effect changes the state of r, since this effect is applied to the state σ where it was generated:

$$\forall \sigma.\ (\sigma \in I \implies \mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma) \in I). \tag{10}$$

The difficulty comes from the need to consider how o's effect changes the state of any other replica r' that receives it; see Figure 6(a). At the time of the receipt, r' may be in a different state σ', due to operations executed at r' concurrently with o. We can show that it is sound to assume that this state σ' also satisfies the invariant. Thus, to check that the operation o preserves the invariant when applied at any replica, it is sufficient to ensure

$$\forall \sigma, \sigma'.\ (\sigma, \sigma' \in I \implies \mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma') \in I). \tag{11}$$

However, establishing this without knowing anything about the relationship between σ and σ' is a tall order. In the bank account example, both $\sigma = 100$ and $\sigma' = 0$ satisfy the integrity invariant (5). Then $\mathcal{F}^{\mathsf{eff}}_{\mathsf{withdraw}(100)}(\sigma)(\sigma') = -100$, which violates the invariant. Condition (11) fails in this case because it does not take into account the tokens acquired by withdraw.

The proof rule in Figure 5 addresses the weakness of (11) by allowing us to assume a certain relationship between the state where an operation is generated (σ) and where its effect is applied (σ'), which takes into account the tokens acquired by the operation. To express this assumption, the rule uses a form of rely-guarantee reasoning [26]. Namely, it requires us to associate each token τ with a *guarantee* relation $G(\tau)$, describing all possible state changes that an operation acquiring τ can cause. Crucially, this includes not only the changes that the operation can cause on the state of its origin replica, but also any change that its effect causes at any other replica it is propagated to. We also have a guarantee relation G_0, describing the changes that can be performed by an operation without acquiring any tokens. Condition S2 requires the guarantees to preserve the invariant.

Like (11), condition S3 considers an arbitrary state σ of o's origin replica r, assumed to satisfy the invariant I. The condition then considers any state σ' of another replica r' to which the effect of o is propagated. The conclusion of S3 requires us to prove that applying the effect $\mathcal{F}^{\mathsf{eff}}_o(\sigma)$ of the operation o to the state σ' satisfies the union of the guarantees associated with the tokens $\mathcal{F}^{\mathsf{tok}}_o(\sigma)$ that the operation o acquires. By S2, this implies that the effect of the operation preserves the invariant. Condition S3 further allows us to assume that the state σ' of r' can be obtained from the state σ of r by applying a finite number of changes allowed by G_0 or the guarantees for *those tokens that do not conflict with any of the tokens acquired by the operation* o, i.e., $G_0 \cup G((\mathcal{F}^{\mathsf{tok}}_o(\sigma))^\perp)$. Informally, acquiring a token denies other replicas permissions to concurrently perform changes that require conflicting tokens.

We now use our proof rule to show that the operations in the banking application (Figure 4) preserve the integrity invariant (5). We assume that the initial state σ_{init} satisfies the invariant. The guarantees are as follows:

$$G(\tau) = \{(\sigma, \sigma') \mid 0 \le \sigma' < \sigma\}; \qquad G_0 = \{(\sigma, \sigma') \mid 0 \le \sigma \le \sigma'\}. \tag{12}$$

Since withdrawals acquire the token τ, the guarantee $G(\tau)$ for this token allows decreasing the balance without turning it negative; the guarantee G_0 allows increasing a non-negative balance. Then condition S2 is satisfied. We show how to check the condition S3 in the most interesting case of $o = \mathsf{withdraw}(a)$. Consider σ and σ' satisfying the premiss of S3:

$$\sigma \in I \wedge (\sigma, \sigma') \in (G_0 \cup G((\mathcal{F}^{\mathsf{tok}}_o(\sigma))^\perp))^*.$$

Since $\mathcal{F}^{\mathsf{tok}}_o(\sigma) = \{\tau\}$, we have that $(\mathcal{F}^{\mathsf{tok}}_o(\sigma))^\perp = \emptyset$. Thus, $(\sigma, \sigma') \in G_0^*$. This and $\sigma \in I$ imply that

$$0 \le \sigma \le \sigma'. \tag{13}$$

If $\sigma < a$, then $\mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma') = \sigma'$. Furthermore, $\sigma' \ge 0$ by (13). Thus, $(\sigma', \mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma')) = (\sigma', \sigma') \in G_0$, which implies the conclusion of S3.

If $\sigma \ge a$, then $\mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma') = \sigma' - a$. Since $\sigma \le \sigma'$, by (13) we have $\sigma' \ge a$. Thus, $(\sigma', \mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma')) = (\sigma', \sigma' - a) \in G(\{\tau\})$, which implies the conclusion of S3. Operationally, in this case our proof rule establishes that, if there was enough money in the account at the replica where the withdrawal was made, then there will be enough money at any replica the withdrawal is delivered to. This completes the proof of our example.

In a banking application with multiple accounts, we could ensure non-negativity of balances by associating every account c with a token τ_c such that $\tau_c \bowtie \tau_c$, but $\tau_c \not\bowtie \tau_{c'}$ for another account c'. Thus, withdrawals from the same account would have to synchronise, while withdrawals from different accounts could proceed without synchronisation. Our proof rule easily deals with this gen-

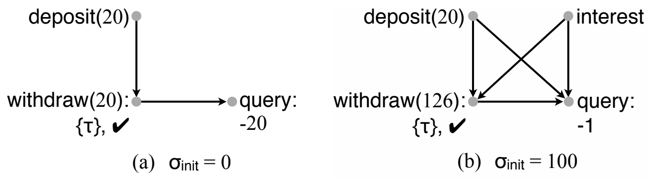


Figure 7. Executions illustrating the unsoundness of the state-based proof rule on weaker consistency models.

eralisation by associating every token τ_c with a guarantee describing the changes to the corresponding account. As we elaborate in §6, the banking application we verify with the aid of our tool allows multiple accounts. There we also provide more complex examples of using our proof rule. For now, it is instructive to see how the proof rule is specialised for some of the simpler instantiations of our consistency model from §3.

Sequential consistency. Recall that for sequential consistency, $\bowtie \; = \{(\tau, \tau)\}$ and we always have $\mathcal{F}^{\mathsf{tok}}_o(\sigma) = \{\tau\}$, so that $(\mathcal{F}^{\mathsf{tok}}_o(\sigma))^\perp = \emptyset$. Let $G_0 = \emptyset$, so that we always have $\sigma = \sigma'$ in S3. Then S2 and S3 require us to find $G(\tau)$ such that

$$G(\tau)(I) \subseteq I \wedge \forall o, \sigma.\ (\sigma \in I \implies (\sigma, \mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma)) \in G(\tau)).$$

It is easy to show that we can find such a $G(\tau)$ if and only if (10) holds for all o. Thus, in this case it is sufficient to check that the effect of an operation preserves the invariant when applied to the same state where it was generated.

Causal consistency. We have $\mathsf{Token} = \emptyset$ and the conditions S2 and S3 become equivalent to

$$G_0(I) \subseteq I \wedge (\forall o, \sigma, \sigma'.\ (\sigma \in I \wedge (\sigma, \sigma') \in G_0^*) \implies (\sigma', \mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma')) \in G_0).$$

In this case the effects of all operations are described by a single guarantee relation G_0. We need to show that every operation satisfies this guarantee while assuming that concurrently executing operations at other replicas do. Note that (11), for all o, is a special case of the above obligation for $G_0 = I \times I$. Thus, (11) is an invariant-based version of the above rely-guarantee proof rule.

As we elaborate in §7, our proof rule bears a lot of similarity to proof rules for strongly consistent shared-memory concurrency [21, 26, 34]. The reasons for the soundness of our proof in the setting of weak consistency are subtle. Its soundness relies crucially on the fact that our consistency model guarantees at least causal consistency and on the commutativity of operation effects (7). For example, some consistency models do not guarantee the transitivity of happens-before [9, 44] and thus allow the execution in Figure 7(a), which uses the operations in Figure 4. Here a withdrawal hb-follows a deposit; a query sees only the withdrawal, thus violating the integrity invariant (5). Since we have proved these operations to preserve the invariant using our proof rule, this rule is unsound over a consistency model allowing the execution in Figure 7(a). We note that the obligation (11), for all o, establishes the invariant I even for a consistency model where hb is only acyclic, but not necessarily transitive.

To illustrate that our rule becomes unsound if we drop the requirement of effect commutativity (7), consider the operations in Figure 4, but with the effect of interest defined by (3). It is easy to show that the premiss of the rule holds for the invariant (5) even with this change. At the same time, the execution in Figure 7(b) violates the invariant, yet is consistent with the operations in Figure 4 according to Definition 2. This is because the evaluation determining the effect of withdraw(126) can order deposit(20) before interest, whereas the evaluation determining the outcome of query can order these operations the other way round, resulting in a smaller balance. Again, the obligation (11) establishes the invariant even without (7): it ensures

$$\forall X \in \mathsf{Exec}(\mathcal{T}, \mathcal{F}).\ \mathsf{eval}^\dagger_{\mathcal{F}}(X) \subseteq I.$$

$$\begin{array}{l}
\exists \mathbb{G} \in \mathcal{P}(\mathsf{Exec}(\mathcal{T}) \times \mathsf{Exec}(\mathcal{T})) \text{ such that} \\
\text{E1. } X_{\mathsf{init}} \in \mathbb{I} \\
\text{E2. } \mathbb{G}(\mathbb{I}) \subseteq \mathbb{I} \\
\text{E3. } \forall X, X', X''.\ \forall e \in X''.E. \\
\quad (X \in \mathbb{I} \wedge X' = X''|_{X''.E - \{e\}} \wedge X'' \in \mathsf{Exec}(\mathcal{T}, \mathcal{F}) \wedge \\
\quad\; e \in \mathsf{max}(X'') \wedge X = \mathsf{ctxt}(e, X'') \wedge (X, X') \in \mathbb{G}^*) \\
\quad \implies (X', X'') \in \mathbb{G} \\
\hline
\qquad\qquad \mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathbb{I}
\end{array}$$

Figure 8. Event-based proof rule.

5. Event-based Proof Rule and Soundness

We now prove the soundness of the state-based proof rule. To this end, we present an *event-based* proof rule (Figure 8), from which the state-based one is derived. This event-based rule highlights the reasons for the soundness of the state-based one. Instead of reasoning about replica states, the event-based rule reasons about executions describing the events that replicas know about; the evaluation of the corresponding effects yields the replica states in the state-based rule. In particular, we specify the desired integrity invariant as a predicate on executions: $\mathbb{I} \subseteq \mathsf{Exec}(\mathcal{T})$. The event-based rule establishes that any execution consistent with given $\mathcal{T} = (\mathsf{Token}, \bowtie)$ and $\mathcal{F}$ belongs to $\mathbb{I}$: $\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathbb{I}$.

As before, we explain the event-based rule from the operational perspective. The rule again uses rely-guarantee reasoning, but with the guarantee $\mathbb{G}$ represented by a relation on executions. The guarantee describes the change to a replica's knowledge brought on by the replica executing a new operation or receiving the effect of an operation originally executed elsewhere.

Conditions E1 and E2 are similar to S1 and S2: E1 requires the invariant $\mathbb{I}$ to allow an empty execution X_{init} (§3), which evaluates to the initial database state σ_{init}; E2 requires the guarantee to preserve the invariant. Condition E3 is graphically illustrated in Figure 6(b). Similarly to S3, the condition E3 considers any operation, denoted by an event e, and checks that the change to the database state made by the operation satisfies the guarantee. This check is done not only at the origin replica r of e, but also at any other replica r' that receives its effect. The execution X can be thought of as describing the events known to the replica r when it executed the operation denoted by e. We assume that the execution X satisfies the invariant $\mathbb{I}$. The execution X' describes the events known to the replica r' just before it receives the effect of e; X'' describes the events known to r' after this, so that $X' = X''|_{X''.E-\{e\}}$. The execution X'' is consistent with $\mathcal{T}$ and $\mathcal{F}$; the conditions in the proof rule imply that so are X and X'. The condition $e \in \mathsf{max}(X'')$ (see (9)) reflects the fact that e is the latest event received by r'. The condition $X = \mathsf{ctxt}(e, X'')$ ensures that X is a part of $X' = X''|_{X''.E-\{e\}}$. This reflects the guarantee of causal message propagation: when r' receives the effect of e, this replica is guaranteed to know about all the events that the replica r knew about when it executed e.

Even though the rule allows us to assume that X is part of X', the latter may contain additional events that the replica r' found out about by the time it received the effect of e. The rule allows us to assume that the changes in the knowledge of r' brought on by adding these events satisfy the guarantee: $(X, X') \in \mathbb{G}^*$. In exchange, the rule requires us to ensure that adding the event e to the knowledge of replica r' will also satisfy the guarantee: $(X', X'') \in \mathbb{G}$.

In the following, we use the fact that the premiss of the implication in E3 entails that all events in $X'.E - X.E$ are causally independent with e.

PROPOSITION 5. *For all X, X', X'' and $e \in X''.E$,*

$$(X' = X''|_{X''.E-\{e\}} \wedge e \in \mathsf{max}(X'') \wedge X = \mathsf{ctxt}(e, X''))$$
$$\implies \neg\exists f \in (X'.E - X.E).\ (e \xrightarrow{X''.\mathsf{hb}} f \vee f \xrightarrow{X''.\mathsf{hb}} e).$$

PROOF. Consider $f \in (X'.E - X.E)$. Since $e \in \mathsf{max}(X'')$, we cannot have $e \xrightarrow{X''.\mathsf{hb}} f$. If $f \xrightarrow{X''.\mathsf{hb}} e$, then $f \in X.E$ due to $X = \mathsf{ctxt}(e, X'')$. But this contradicts $f \in (X'.E - X.E)$. □

We now give the proof of soundness of the event-based rule and sketch the derivation of the state-based one (we give a full proof of the latter in [2, §A]).

Let $\sqsubseteq$ be the following partial order on executions:

$$X \sqsubseteq X' \iff (X = X'|_{X.E} \wedge ((X'.\mathsf{hb})^{-1})(X.E) \subseteq X.E). \quad (14)$$

When $X \sqsubseteq X'$, we say that X is a *causal cut* of X'; any event is included into X together with its causal dependencies in X'. Operationally, $X \sqsubseteq X'$ means that X and X' can describe the knowledge of a replica at different points in the same database computation.

PROPOSITION 6.

$$\forall X \in \mathsf{Exec}(\mathcal{T}, \mathcal{F}).\ \forall Y.\ (Y \sqsubseteq X \implies Y \in \mathsf{Exec}(\mathcal{T}, \mathcal{F})).$$

THEOREM 7. *The event-based proof rule in Figure 8 is sound.*

PROOF. Assume E1-E3 hold. We prove that

$$\forall X'' \in \mathsf{Exec}(\mathcal{T}, \mathcal{F}).\ \forall Y.\ (Y \sqsubseteq X'' \implies (Y, X'') \in \mathbb{G}^*), \quad (15)$$

i.e., that the guarantee $\mathbb{G}$ allows us to transition into a consistent execution X'' from any of its causal cuts Y. The desired conclusion $\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathbb{I}$ follows from (15): it implies $(X_{\mathsf{init}}, X'') \in \mathbb{G}^*$, but $X_{\mathsf{init}} \in \mathbb{I}$ (E1) and $\mathbb{G}$ preserves $\mathbb{I}$ (E2).

The proof of (15) is done by induction on the size of X''. In the base case, we must have $Y = X'' = X_{\mathsf{init}}$, which implies $(Y, X'') \in \mathbb{G}^*$. In the induction step, we consider $X'' \in \mathsf{Exec}(\mathcal{T}, \mathcal{F})$ and $Y \sqsubseteq X''$ such that $Y \neq X''$. We pick an event $e \in (X''.E - Y.E)$ such that $e \in \mathsf{max}(X'')$ and define X and X' as in E3:

$$X = \mathsf{ctxt}(e, X'') \wedge X' = X''|_{X''.E-\{e\}}.$$

Then

$$Y \sqsubseteq X' \wedge X \sqsubseteq X'. \quad (16)$$

By Proposition 6 we have $X, X' \in \mathsf{Exec}(\mathcal{T}, \mathcal{F})$. Thus, we can apply the induction hypothesis to X' and its causal cuts X and Y, as well as to X and its causal cut X_{init}, getting:

$$(Y, X') \in \mathbb{G}^* \wedge (X, X') \in \mathbb{G}^* \wedge (X_{\mathsf{init}}, X) \in \mathbb{G}^*.$$

By E1 and E2, $(X_{\mathsf{init}}, X) \in \mathbb{G}^*$ implies $X \in \mathbb{I}$. Together with $(X, X') \in \mathbb{G}^*$, this allows us to apply E3 and obtain $(X', X'') \in \mathbb{G}$. This and $(Y, X') \in \mathbb{G}^*$ imply $(Y, X'') \in \mathbb{G}^*$, as required. □

In operational terms, the statement (15) established in the proof ensures that any sequence of changes in the knowledge of a replica during a database computation is described by $\mathbb{G}^*$. The above proof relies crucially on the fact that our consistency model guarantees at least causal consistency. For example, in (16) we can deduce $X \sqsubseteq X'$ from $X = \mathsf{ctxt}(e, X'')$ because happens-before is transitive.

COROLLARY 8. *The state-based proof rule in Figure 5 is sound.*

PROOF SKETCH. Assume a state-based invariant $I \subseteq \mathsf{State}$. We construct the corresponding event-based invariant $\mathbb{I}$ as the set of all executions that evaluate to a state in I: $\mathbb{I} = \mathsf{eval}_{\mathcal{F}}^{-1}(I)$. Then the conclusion $\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathbb{I}$ of the event-based rule implies the conclusion $\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathsf{eval}_{\mathcal{F}}^{-1}(I)$ of the state-based rule.

We now show that the premiss of the state-based rule implies that of the event-based rule. Assume state-based guarantees G_0 and G that satisfy S1-S3. We construct the corresponding event-based guarantee $\mathbb{G}$ by describing the change to the knowledge of a replica brought on by incorporating the effect of an operation satisfying the state-based guarantees G_0 and G:

$$\mathbb{G} = \{(X, Y) \mid \exists e.\ (Y.E - X.E) = \{e\} \wedge X \sqsubseteq Y \wedge$$
$$(\mathsf{eval}_{\mathcal{F}}(X), \mathsf{eval}_{\mathcal{F}}(Y)) \in G_0 \cup G(Y.\mathsf{tok}(e))\}. \quad (17)$$

Thus, the guarantee $\mathbb{G}$ consists of pairs (X, Y), where Y extends X by a single event e representing the operation, and the two executions evaluate to a pair of states in G_0 or $G(\tau)$ for some token τ acquired by e.

It remains to prove that the event-based guarantee $\mathbb{G}$ satisfies conditions E1-E3. Conditions E1 and E2 trivially follow from conditions S1 and S2; we thus only need to show that S3 implies E3. Assume that for some X, X', X'' and $e \in X''.E$, the premiss of E3 holds:

$$X \in \mathbb{I} \wedge X' = X''|_{X''.E-\{e\}} \wedge X'' \in \mathsf{Exec}(\mathcal{T}, \mathcal{F}) \wedge$$
$$e \in \mathsf{max}(X'') \wedge X = \mathsf{ctxt}(e, X'') \wedge (X, X') \in \mathbb{G}^*. \quad (18)$$

Let $\sigma = \mathsf{eval}_{\mathcal{F}}(X)$, $\sigma' = \mathsf{eval}_{\mathcal{F}}(X')$ and $o = X''.\mathsf{oper}(e)$. We now show that the premiss of S3 holds:

$$\sigma \in I \wedge (\sigma, \sigma') \in (G_0 \cup G((\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp}))^*. \quad (19)$$

First of all, $\sigma \in I$ follows from $X \in \mathbb{I}$ by the definition of $\mathbb{I}$. Furthermore, by Proposition 5, all events in $(X'.E - X.E)$ are unrelated to e in $(X''.\mathsf{hb} \cup (X''.\mathsf{hb})^{-1})$. But then by (8), they cannot acquire tokens that conflict with the ones acquired by e:

$$\forall f \in (X'.E - X.E).\ \neg(X''.\mathsf{tok}(e) \bowtie X''.\mathsf{tok}(f)).$$

Using this fact, $(X, X') \in \mathbb{G}^*$ given by (18) and the definition of $\mathbb{G}$ given by (17), we can show that

$$(\sigma, \sigma') = (\mathsf{eval}_{\mathcal{F}}(X), \mathsf{eval}_{\mathcal{F}}(X')) \in (G_0 \cup G((X''.\mathsf{tok}(e))^{\perp}))^*$$
$$= (G_0 \cup G((\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp}))^*,$$

thus establishing (19). Then the conclusion of S3 yields $(\sigma', \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma')) \in G_0 \cup G(\mathcal{F}_o^{\mathsf{tok}}(\sigma))$, so that

$$\begin{aligned}(\mathsf{eval}_{\mathcal{F}}(X'), \mathsf{eval}_{\mathcal{F}}(X'')) &= (\sigma', \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma')) \\ &\in G_0 \cup G(\mathcal{F}_o^{\mathsf{tok}}(\sigma)) \\ &= G_0 \cup G(X''.\mathsf{tok}(e)).\end{aligned} \quad (20)$$

This implies the conclusion of E3: $(X', X'') \in \mathbb{G}$. □

The above proof relies crucially on Lemma 4, which allows us to define $\mathsf{eval}_{\mathcal{F}}$. The lemma guarantees that, when evaluating executions, choosing different orders for causally independent events does not affect the resulting state. In (20) this allows us to choose a particular convenient order of evaluating X'' that applies the operation o last. Lemma 4 holds due to the commutativity condition (7), and this illustrates the importance of this condition for the soundness of the state-based rule.

6. Examples and Automation

We have developed a tool that automates the state-based proof rule by reducing its obligations to SMT queries. Using the tool, we

Application	# ops	# tokens	# invariants	time (ms)
Banking	5	1	1	385
Auction	14	9	12	5297
Courseware	5	5	2	512

Figure 9. Characteristics of the applications verified and the time taken by the tool. The numbers of operations are given ignoring operation parameters. The numbers of tokens are similarly given without taking into account tokens associated with different instances of the same object, such as different bank accounts. The tool was run on a Mac Mini, 3 GHz Intel Core i7.

$$
\begin{aligned}
\mathsf{State} &= \mathcal{P}(\mathbb{N}) \times (\mathbb{N} \cup \{\bot\})\\
\sigma_{\mathsf{init}} &= (\emptyset, \bot)\\
I &= \{(B, w) \mid w \neq \bot \implies B \neq \emptyset \wedge w = \max(B)\}\\
\mathsf{Token} &= \{\tau_c, \tau_p\}\\
\bowtie &= \{(\tau_c, \tau_c), (\tau_c, \tau_p), (\tau_p, \tau_c)\}\\
\mathcal{F}_{\mathsf{place}(b)}((B, w)) &= \text{if } w \neq \bot \text{ then } (\text{✗}, \mathsf{skip}, \{\tau_p\})\\
&\quad\ \text{else } (\checkmark, (\lambda(B', w').\,(B' \cup \{b\}, w')), \{\tau_p\})\\
\mathcal{F}_{\mathsf{close}}((B, w)) &= \text{if } (w \neq \bot \vee B = \emptyset) \text{ then } (\text{✗}, \mathsf{skip}, \{\tau_c\})\\
&\quad\ \text{else } (\checkmark, (\lambda(B', w').\,(B', \max(B))), \{\tau_c\})\\
\mathcal{F}_{\mathsf{query}}((B, w)) &= ((B, w), \mathsf{skip}, \emptyset)
\end{aligned}
$$

Figure 10. A fragment of an auction application.

have verified three applications: an extended version of the banking application in Figure 4, an auction service and a course registration system. Our results are summarised in Figure 9. In the following, we first show more sophisticated uses of our proof rule using fragments of the auction and courseware applications. We then present our automation approach and the complete applications that we verified.

Auction service. Figure 10 shows a fragment of an auction application. An auction can be either open or closed. While the auction is open, a client can place a bid with the amount b using the $\mathsf{place}(b)$ operation. A client can also close the auction at any time using the close operation, which declares the winner. Finally, clients can query the database state using query.

The database state is of the form (B, w). Here B consists of the amounts of the bids placed; for simplicity, we do not distinguish two bids with the same amount. The component w is either $\bot$, signifying that the auction is still open, or the winning bid. A successful $\mathsf{place}(b)$ operation has the effect of adding b to B. The close operation writes the winning bid into w. Note that the effects of two close operations do not commute. To satisfy (7), and to ensure that clients can only close the auction once, we let close operations acquire a token τ_c such that $\tau_c \bowtie \tau_c$.

The integrity invariant I we would like to maintain in the courseware application is that, if the auction is closed, then the winning bid is the maximal of all the bids placed. Without using any other tokens than τ_c, this invariant can be violated: Alice can close the auction and declare the winner, e.g., 100, without being aware of a higher bid 105 placed concurrently by Bob. A query aware of both operations will return the bid set containing 105 and 100 but mark 100 as the winning bid in the set.

To preserve the invariant in the RedBlue consistency model (§3), we would have to use strong consistency for both place and close operations, i.e., let them acquire the mutually exclusive token τ_c. To address this inefficiency, Balegas et al. [10] proposed a hybrid model where consistency can be strengthened using *multi-level locks*, analogous to readers-writer locks from shared memory. In our example, we represent such a lock by a pair of tokens: τ_c, introduced before, and τ_p. Each close operation acquires τ_c, and each place operation, τ_p. We have $\tau_c \bowtie \tau_p$. Hence, for every pair of close and $\mathsf{place}(b)$ operations, either close is aware of the bid b and takes it into account when computing the winner, or the $\mathsf{place}(b)$ operation is aware that the auction has been closed and, hence, does not place the bid. However, we do not have $\tau_p \bowtie \tau_p$ and, hence, bid placements can be causally independent. In our analogy with a readers-writer lock, bid placements play the role of readers and closing the auction, the role of a writer.

Balegas et al. [10] show how to implement multi-level locks so that a replica can place a bid without any synchronisation; only an operation closing the auction has to synchronise with other replicas to make sure that no bids are placed concurrently. Thus, the most frequent operation of bid placement is the least expensive.

We now use our proof rule to show that the above consistency choice is indeed sufficient to preserve the invariant I. Let

$$
\begin{aligned}
G_0 &= \{((B, w), (B, w)) \mid (B, w) \in I\};\\
G(\tau_p) &= \{((B, \bot), (B', \bot)) \mid B \subset B'\};\\
G(\tau_c) &= \{((B, \bot), (B, \max(B))) \mid B \neq \emptyset\}.
\end{aligned}
$$

Then the condition S2 in Figure 5 is satisfied. We show how to check the condition S3 in the most interesting case of $o = \mathsf{place}(b)$.

Consider $\sigma = (B, w)$ and $\sigma' = (B', w')$ satisfying the premiss of S3. Then $\sigma \in I$. Also, since

$$(\sigma, \sigma') \in (G_0 \cup G((\mathcal{F}^{\mathsf{tok}}_o(\sigma))^{\perp}))^*$$

and $(\mathcal{F}^{\mathsf{tok}}_o(\sigma))^{\perp} = \{\tau_p\}$, we get

$$w' = w \wedge B \subseteq B' \wedge (w \neq \bot \implies B' = B). \tag{21}$$

If $w \neq \bot$, then $\mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma') = \sigma'$ and, by (21), $\sigma = \sigma'$. Since $\sigma \in I$, we have

$$(\sigma', \mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma')) = (\sigma', \sigma') = (\sigma, \sigma) \in G_0.$$

This implies the conclusion of S3.

If $w = \bot$, then $\mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma') = (B' \cup \{b\}, w')$. In this case (21) implies $w' = w = \bot$. Thus,

$$(\sigma', \mathcal{F}^{\mathsf{eff}}_o(\sigma)(\sigma')) = ((B', w'), (B' \cup \{b\}, w')) \in (G_0 \cup G(\tau_p)),$$

the desired conclusion of S3. Operationally, our proof rule establishes that, if the auction was open at the replica where the bid was placed, then it will be open at any replica the bid is delivered to.

Similarly to our banking application (§4), we can deal with multiple auctions by using a pair of tokens (τ_c, τ_p) for every auction. The above proof generalises straightforwardly to this case.

Courseware. Our next example illustrates a different kind of an integrity invariant and the use of replicated data types [38] to construct commutative operations. Figure 11 shows a fragment of a courseware application. We assume sets of courses Course and students $\mathsf{Student}$. A client can add a course c using $\mathsf{addCourse}(c)$ and register a student s using $\mathsf{register}(s)$. A registered student s can be enrolled into a course c using $\mathsf{enroll}(s, c)$. In the application fragment we consider, student registrations and enrollments cannot be cancelled. However, a course c that has not secured any student enrollment can be removed using $\mathsf{remCourse}(c)$. As usual, we also have a query operation.

A database state (S, C, E) consists of the set of students S, the set of courses C and the enrollment relation E between students and courses. The set of courses is actually not just an ordinary set, but a *replicated remove-wins set* $\mathsf{RWset}(\mathsf{Course})$, explained in the following. The effects of operations are mostly as expected, with courses accessed using special functions add, remove and $\mathsf{contents}$ on the replicated set. Note that the operation $\mathsf{enroll}(c, s)$ takes effect only if the student s is registered and the course c exists.

$$\begin{aligned}
\mathsf{State} &= \mathcal{P}(\mathsf{Student}) \times \mathsf{RWset}(\mathsf{Course}) \times \mathcal{P}(\mathsf{Student} \times \mathsf{Course})\\
\sigma_{\mathsf{init}} &= (\emptyset, \emptyset_{\mathsf{RWset}}, \emptyset)\\
I &= \{(S, C, E) \mid E \subseteq \mathcal{P}(S \times \mathsf{contents}(C))\}\\
\mathsf{Token} &= \{\tau_{e(c)}, \tau_{r(c)} \mid c \in \mathsf{Course}\}\\
\bowtie &= \{(\tau_{e(c)}, \tau_{r(c)}), (\tau_{r(c)}, \tau_{e(c)}) \mid c \in \mathsf{Course}\}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{F}_{\mathsf{register}(s)}((S, C, E)) =\\
&\quad (\bot, (\lambda(S', C', E').\, (S' \cup \{s\}, C', E')), \emptyset)\\
&\mathcal{F}_{\mathsf{addCourse}(c)}((S, C, E)) =\\
&\quad (\bot, (\lambda(S', C', E').\, (S', \mathsf{add}(c, C'), E')), \emptyset)\\
&\mathcal{F}_{\mathsf{enroll}(s,c)}((S, C, E)) =\\
&\quad \text{if } (s \notin S \vee c \notin \mathsf{contents}(C)) \text{ then } (\text{✗}, \mathsf{skip}, \{\tau_{e(c)}\})\\
&\quad \text{else } (\checkmark, (\lambda(S', C', E').\, (S', C', E' \cup \{(s, c)\})), \{\tau_{e(c)}\})\\
&\mathcal{F}_{\mathsf{remCourse}(c)}((S, C, E)) =\\
&\quad \text{if } (c \notin \mathsf{contents}(C) \vee \exists s.\, (s, c) \in E)) \text{ then } (\text{✗}, \mathsf{skip}, \{\tau_{r(c)}\})\\
&\quad \text{else } (\checkmark, (\lambda(S', C', E').\, (S', \mathsf{remove}(c, C'), E')), \{\tau_{r(c)}\})\\
&\mathcal{F}_{\mathsf{query}}((S, C, E)) = ((S, \mathsf{contents}(C), E), \mathsf{skip}, \emptyset)
\end{aligned}$$

$$\begin{aligned}
\mathsf{RWset}(\mathsf{Course}) &= \mathcal{P}(\mathsf{Course}) \times \mathcal{P}(\mathsf{Course})\\
\emptyset_{\mathsf{RWset}} &= (\emptyset, \emptyset)\\
\mathsf{add}(c, (A, T)) &= (A \cup \{c\}, T)\\
\mathsf{remove}(c, (A, T)) &= (A, T \cup \{c\})\\
\mathsf{contents}((A, T)) &= A - T
\end{aligned}$$

Figure 11. A fragment of a courseware application.

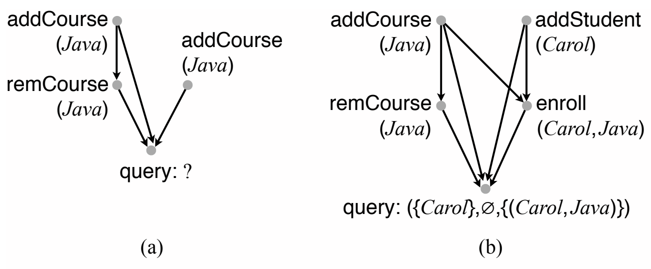


Figure 12. Executions illustrating the need for (a) replicated data types and (b) tokens in the courseware application.

The operation $\mathsf{remCourse}(c)$ removes the course c only when it exists and has no students enrolled into it.

Using a replicated data type for the set of courses is needed to satisfy (7), because additions to and removals from a usual set do not commute. To illustrate, consider the execution in Figure 12(a). There Alice adds a course on Java and then changes her mind and removes the course; concurrently, Bob adds the same Java course. If we maintained the information about courses using a usual set, then the outcome of the query in the figure would depend on the order in which we evaluate the effects of the causally independent operations $\mathsf{addCourse}(\textit{Java})$ and $\mathsf{remCourse}(\textit{Java})$: the query would return $\emptyset$ if the addition was evaluated before removal, and $\{\textit{Java}\}$ otherwise (see Definition 2). In an actual database, implementing the operations using ordinary sets would violate the replica convergence property (§2.1).

Replicated data types [38] provide implementations of operations on data structures with commutative effects. They differ in the way in which they resolve conflicting updates to the data structure, such as those in Figure 12(a): when using an *add-wins* set, the query in the figure will return $\{\textit{Java}\}$, and when using a *remove-wins* set, $\emptyset$ [37]. The decision which data type to use ultimately depends on application requirements. To keep presentation manageable, in our example we use one of the simplest set data types, which provides a rudimentary version of the remove-wins semantics.

The data type represents the replicated set of courses using a pair of sets (A, T). The function $\mathsf{add}(c, \cdot)$ puts c into the set of A, and the function $\mathsf{remove}(c, \cdot)$ puts c into the set T, called the *tombstone* set. To get the contents of the replicated set, we just take the difference of A and T. The functions $\mathsf{add}(c, \cdot)$ and $\mathsf{remove}(c, \cdot)$ commute: even if the removal is evaluated first, it will still cancel the subsequent addition[1]. This ensures that the effects of all operations in Figure 11 commute and thus satisfy (7).

The integrity invariant I we would like to maintain in this application is that the enrollment relation refers to existing courses and students only. This property is an instance of *referential integrity*, which requires an object referenced in one part of the database to exist in another. Without using tokens, the operations in our application can break the invariant. This is illustrated by the execution in Figure 12(b). There a Java course initially has no students enrolled. Then Alice removes the course and concurrently Bob enrolls Carol into it, thinking that the course is still available. This results in Carol being enrolled into a non-existent course.

To ensure that such situations do not happen, we use a pair of conflicting tokens for each course $c \in \mathsf{Course}$: $\tau_{e(c)}$ and $\tau_{r(c)}$. The operation $\mathsf{enroll}(s, c)$ acquires $\tau_{e(c)}$, and the operation $\mathsf{remCourse}(c)$ acquires $\tau_{r(c)}$. Then for every pair of operations $\mathsf{enroll}(s, c)$ and $\mathsf{remCourse}(c)$, either the enrollment operation is aware that the course has been removed, or the removal is aware that there are still students enrolled into the course; in either case the corresponding operation takes no effect. However, other pairs of operations can be causally independent and, hence, do not have to synchronise. This includes pairs of operations enrolling students into courses and pairs of operations manipulating courses, such as those in Figure 12(a). The above use of tokens is equivalent to associating every course with a multi-level lock [10] that can be in one of two modes, one of which allows enrolling students into a course ($\tau_{e(c)}$) and the other removing the course ($\tau_{r(c)}$). Unlike in the auction application above, neither of the tokens $\tau_{e(c)}$ or $\tau_{r(c)}$ conflicts with itself, and thus, neither of the above lock modes is exclusive.

Our proof rule can establish that the above consistency choice is sufficient to preserve the integrity invariant. To this end, we use the following guarantees, associating changes with tokens as expected:

$$\begin{aligned}
G_0 = (I \times I) \cap \{((S, C, E), (S', C', E)) \mid\\
S \subseteq S' \wedge \mathsf{contents}(C) \subseteq \mathsf{contents}(C')\};\\
G(\tau_{e(c)}) = (I \times I) \cap \{((S, C, E), (S, C, E')) \mid\\
\exists s.\, E' = E \uplus \{(s, c)\}\};\\
G(\tau_{r(c)}) = (I \times I) \cap \{((S, C, E), (S, C', E)) \mid\\
\mathsf{contents}(C) = \mathsf{contents}(C') \uplus \{c\}\}.
\end{aligned}$$

The actual proof is similar to that of the auction application above and is omitted.

Parallel snapshot isolation. In [2, §B] we provide an additional example illustrating the versatility of our consistency model and proof rule. We show that the consistency model can encode the recently-proposed model of parallel snapshot isolation [36, 40], which takes a different approach to strengthening consistency from the models we have considered so far. This encoding exploits the fact that an operation in our consistency model may acquire a different set of tokens depending on the state it is executed in

[1] In fact, once an element was removed, it can never be successfully added again, which may not be a desirable behaviour. There are replicated sets that provide a more sophisticated semantics [37].

$$
\begin{array}{ll}
\multicolumn{2}{l}{\exists G_0 \in \mathcal{P}(\mathsf{State} \times \mathsf{State}), G \in \mathsf{Token} \to \mathcal{P}(\mathsf{State} \times \mathsf{State})} \\
\multicolumn{2}{l}{\text{such that}} \\
\text{T1.} & \sigma_{\mathsf{init}} \in I \\
\text{T2.} & G_0(I) \subseteq I \wedge \forall \tau.\, G(\tau)(I) \subseteq I \\
\text{T3.} & \forall o.\, \exists T.\, \exists P_1, \dots, P_n, Q_1, \dots, Q_n \in \mathcal{P}(\mathsf{State}). \\
\text{T3a.} & \quad T = \bigcap \{\mathcal{F}_o^{\mathsf{tok}}(\sigma) \mid \sigma \in I\} \wedge \\
\text{T3b.} & \quad I \subseteq \bigcup_{i=1}^{n} P_i \wedge \\
\text{T3c.} & \quad \forall i = 1..n.\, P_i \subseteq Q_i \wedge \\
\text{T3d.} & \qquad\qquad (G_0 \cup G(T^{\perp}))(Q_i) \subseteq Q_i \wedge \\
\text{T3e.} & \qquad\qquad (Q_i \times (\mathcal{F}_o^{\mathsf{eff}}(P_i)(Q_i))) \subseteq (G_0 \cup G(T)) \\
\hline
\multicolumn{2}{c}{\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathsf{eval}_{\mathcal{F}}^{-1}(I)}
\end{array}
$$

Figure 13. Proof rule used by our tool. We assume a token system $\mathcal{T} = (\mathsf{Token}, \bowtie)$ and use the same notation as in Figure 5. We let $\mathcal{F}_o^{\mathsf{eff}}(P_i)(Q_i) = \{\mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma') \mid \sigma \in P_i \wedge \sigma' \in Q_i\}$.

(cf. (6)). In [2, §B] we also provide a specialisation of the state-based proof rule for parallel snapshot isolation.

Automation. Our tool uses the proof rule in Figure 13, which is derived from the one in Figure 5 and is more amenable to automation. The premisses T1 and T2 are identical to S1 and S2; T3 changes S3 in two ways. A minor change is motivated by the fact that our tool currently handles only operations that acquire the same set of tokens regardless of the state they are executed in. Hence, T3 precomputes the set of tokens T acquired by an operation o (T3a). The key way in which T3 changes S3 is that it eliminates the transitive closure of the guarantees, which is hard to automate. Whereas S3 quantifies over states σ where the effect of an operation o is generated and σ' where it is applied, T3 considers properties of these states, respectively denoted by predicates P_i and Q_i, $i = 1..n$. T3b requires the predicates P_i to cover all possible states in which o can be executed. T3c requires Q_i to cover P_i, reflecting the fact that the effect of o can be applied in a state different from the one where it was generated. T3d requires Q_i to be *stable* under the changes allowed by the guarantees [26]. Finally, T3e checks that if an effect of o is generated in a state satisfying P_i, then applying this effect to a state satisfying Q_i is consistent with the guarantees. Note that the constraints T3c and T3d have the same effect as relating the states σ and σ' in S3 by a transitive closure of guarantees.

For example, consider the operation $o = \mathsf{withdraw}(a)$ from the banking application in Figure 4. We let $T = \{\tau\}$ and use the guarantees (12). We use two predicates:

$$P_1 = \{\sigma \mid \sigma \geq a\}; \quad P_2 = \{\sigma \mid 0 \leq \sigma < a\}.$$

These are motivated by the condition of the if-then-else in $\mathcal{F}_o^{\mathsf{eff}}$, as well as the invariant I. We then let $Q_1 = P_1$ and $Q_2 = I$. It is easy to check that the obligations in T3 are fulfilled.

Our tool accepts as input a token system $\mathcal{T}$, the semantics of operations $\mathcal{F}$ and an integrity invariant I, the latter two in the SMT-LIB format (we leave a programming language for writing operations as future work). The tool generates predicates P_i from preconditions of branches in $\mathcal{F}$. As Q_i, the tool takes either P_i or the invariant I. Finally, the tool generates guarantees G_0 and G by intersecting the semantics of operations $\mathcal{F}$ with the invariant I. The required obligations are then discharged using the Z3 SMT solver [1].

Applications verified. The applications verified using our tool (Figure 9) are more realistic versions of the examples we discussed before (Figures 4, 10 and 11).

The banking application extends the one in Figure 4 by considering multiple accounts and allowing clients to transfer money between accounts. We preserve the non-negativity of all balances by associating a mutual exclusion token with each account, as described in §4.

The auction application extends the one in Figure 10 by additionally maintaining information about buyers, sellers and products, and by allowing clients to sell multiple product items in a single auction. Buyers and sellers can register and unregister. Registered buyers can bid in open auctions, and registered sellers can add products, create auctions consisting of these and close auctions. The complex data model of this application requires multiple integrity invariants, including referential integrity constraints spanning multiple parts of the database. This makes it nontrivial to see if enough synchronisation has been added to the application to preserve these invariants, and our tool copes with this task.

The courseware application extends the one in Figure 11 by allowing clients to cancel student registrations and enrollments. It also imposes an additional integrity invariant limiting the number of students that can register for a course; maintaining this invariant requires extra synchronisation.

The above case studies demonstrate the feasibility of applying our proof rule to realistic applications.

7. Related Work

Reasoning in strongly consistent shared memory. Our state-based proof rule interprets tokens as permissions to perform certain state changes. Such interpretations have been used in various logics for strongly consistent shared memory [20, 21, 34]. For example, such a logic could allow threads to modify the memory in a particular way only when holding a mutual exclusion lock, similar to our use of a token in the banking application (§4).

This similarity suggests that existing work in shared memory may be helpful in exploring the novel area of replicated databases. However, the distributed and weakly consistent setting in which our proof rule is applied makes the reasons for its soundness subtle. In this setting, we do not have an illusion of a single copy of the database state and a global notion of time this copy would evolve with: as Figure 1(b) illustrates, different processes can see events as occurring in different orders. The usual justification for the soundness of the proof rules for strong consistency relies on the concepts of global time and state: when considering a thread holding a mutex lock, such proof rules reason that no other thread can hold the lock *at the same time* and, hence, modify *the* memory state in the way associated with the lock. In this setting, locks constrain the global order on events. In contrast, tokens in our consistency model provide a more subtle guarantee (8), only constraining the partial happens-before relation.

Reasoning about consistency in distributed systems and databases. Several papers have considered reasoning about correctness properties on weak consistency models of replicated and centralised databases.

Bailis et al. [8] have proposed a criterion for checking when an integrity invariant is preserved by running operations without using any synchronisation at all. But they do not provide guidelines on how to introduce synchronisation if the invariant is violated.

Li et al. [29, 30] have proposed a static analysis that uses the proof rule (11) to check if executing operations on causal consistency preserves a given integrity invariant. In case when (11) fails for some operation o, the analysis suggests to execute o under strong consistency in the RedBlue consistency model (§3). However, the analysis does not check that the result will indeed validate the invariant, and our proof rule fills this gap.

Sivaramakrishnan et al. [39] have proposed a static analysis that automatically chooses consistency levels in a replicated database given programmer-supplied contracts. However, these contracts are more low-level than our invariants, since they typically constrain the happens-before relation. For example, in the banking application (Figure 4) their contract requires happens-before to totally order all withdrawal operations. The static analysis then ensures that the contract is followed, but not that it ensures the integrity invariant (5).

Lu et al. [32] proposed proof rules for establishing correctness properties of transactions running on non-hybrid weak consistency models of classical relational databases, such as snapshot isolation [13]. In contrast, we concentrate on hybrid consistency models of modern replicated databases, which are more sophisticated.

Fekete [22] considered a hybrid consistency model for relational databases where some transactions execute under snapshot isolation [13] and some under serialisability, a form of strong consistency. He proposed conditions determining which transactions in an application need to execute under serialisability for the whole application to be *robust*, i.e., produce only behaviours that would be obtained by executing *all* transactions under serialisability. In contrast, our proof rule only checks integrity invariants while allowing the application to produce weakly consistent behaviours and, hence, benefit from the resulting performance gains.

Weak memory models. Strong consistency is forgone not only by modern databases, but also by shared-memory multiprocessors and programming languages, which provide *weak memory models*. All such models used in practice are hybrid, in that they allow the programmer to strengthen consistency on demand, e.g., using memory fences. However, weak memory models usually provide only a limited number of operations on data, such as reads, writes and compare-and-swaps on single memory cells. Concurrent writes to the same memory cell result in one value being overwritten by the other. In contrast, we deal with arbitrary operations (6) that merge concurrent updates in a user-defined way.

That said, in the future there may be a fruitful exchange of ideas between program logics for applications using weakly consistent databases and those running on weak memory models. In particular, there have been recent proposals of program logics for the "release-acquire" fragment of the C/C++ memory model [42, 43]. This fragment is analogous to causal consistency, with the above caveats about the allowed operations. However, the published logics do not meaningfully handle operations requesting the stronger "sequentially consistent" level of C/C++. Reasoning about on-demand requests of stronger-than-causal consistency is precisely the goal of the present paper.

Several papers [4–6, 14, 18, 19] have verified application correctness on weak memory models using model checkers and abstract interpreters. These papers thus explore verification approaches different from the one considered in this paper. Additionally, most of the papers have focussed on models similar to TSO [4, 14, 18, 19], which is stronger than the causal consistency model we consider as a baseline. As the target correctness property, papers on weak memory models have often considered robustness (see above), which is too strong a requirement for our setting. On the other hand, some of the papers [4, 5, 14, 19] automatically inferred fences required to satisfy a correctness property. We do not address the inference of consistency choices, although in the future our state-based proof rule can serve as a basis for this.

Consistency models. Our conflict relations are similar to those used by Pedone and Schiper [35] to specify constraints on message delivery in a broadcast algorithm. We use the conflict relations to define a high-level consistency model, which abstracts from a message-based database implementation.

In a position paper, Li et al. [28] independently proposed an idea of a hybrid consistency model similar to ours. Their model does not have a formal semantics and is less flexible than ours, since their analogue of the conflict relation is defined directly on operations, instead of indirectly using tokens. This does not allow the synchronisation mandated for an operation to depend on the state it is executed in and, hence, does not allow expressing parallel snapshot isolation (§6 and [2, §B]).

Specifying consistency models. The formal specification of our consistency model (§3) builds on a framework previously proposed to specify forms of eventual consistency [16]. Despite this similarity, we take a somewhat different approach to specifying the semantics of operations. Previous work [16] specified the return value of an event by an arbitrary function of its context in the execution (§3). In contrast, our Definition 2 uses a particular function $\mathsf{eval}^{\dagger}_{\mathcal{F}}$, itself constructed from more primitive functions $\mathcal{F}^{\mathsf{eff}}_{o}$, operating on states. This choice allows us to define the semantics of operations in terms of states, as opposed to events, which can then be used in our state-based proof rule. The use of states also allows to use off-the-shelf SMT solvers to discharge the required verification conditions. However, it is likely that our event-based rule may be adapted to the operation specifications used in [16].

8. Conclusion and Future Work

We presented the first proof rule establishing that a given consistency choice in a replicated database is sufficient to preserve a given integrity invariant. Our proof rule is modular and simple to use. We demonstrated this by small but nontrivial examples, and by reducing the verification conditions of the proof rule to SMT checks. Despite this simplicity, the soundness of our proof rule is nontrivial: the rule fully exploits the guarantees provided by our consistency model while correctly accounting for anomalies it allows.

Our results represent only an initial step in building an infrastructure of reasoning methods for applications using modern replicated databases. They open several avenues for future work. First, our generic consistency model is not implemented by any database in its full generality; we use it only as a means to compactly represent a selection of more specific models in existing implementations. However, in the future the generic model can serve as a basis for exploring the space of possible hybrid consistency models. One could also consider a database that implements our model in its general form.

Second, the soundness of our proof rule relies on the fact that our consistency model guarantees at least causal consistency (§4). Even though causal consistency can be implemented without any synchronisation between replicas, this model has its cost [15]. In the future, we plan to propose proof rules for weaker models where causality preservation is not guaranteed for all operations. We also hope to generalise our methods to more expressive correctness properties than integrity invariants.

Third, in this paper we used the event-based proof rule just to structure the proof of soundness of the state-based one. However, the event-based rule is also interesting in its own right. In the future it can be used in cases when to prove a correctness property, we need to maintain information about the computation history. For example, this is often necessary when reasoning about shared-memory concurrency [23, 25].

Finally, we have concentrated on checking that a particular choice of a conflict relation and tokens acquired by operations is sufficient to preserve a given integrity invariant. We hope that in the future our state-based proof rule can serve as a basis for tools that infer these parameters automatically.

References

[1] https://github.com/Z3Prover/z3.

[2] Extended version of this paper. Available from the submission system.

[3] D. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2), 2012.

[4] P. A. Abdulla, M. F. Atig, and N. T. Phong. The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In *ESOP*, 2015.

[5] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *CAV*, 2014.

[6] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, 2013.

[7] Amazon. Supported operations in DynamoDB. http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/APISummary.html, 2015.

[8] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *PVLDB*, 2015.

[9] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *SIGMOD*, 2014.

[10] V. Balegas, N. Preguiça, R. Rodrigues, S. Duarte, C. Ferreira, M. Najafzadeh, and M. Shapiro. Putting the consistency back into eventual consistency. In *EuroSys*, 2015.

[11] Basho Inc. Using strong consistency in Riak. http://docs.basho.com/riak/latest/dev/advanced/strong-consistency/, 2015.

[12] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[13] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[14] A. Bouajjani, E. Derevenetc, and R. Meyer. Checking and enforcing robustness against TSO. In *ESOP*, 2013.

[15] M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. E. T. Rodrigues. On the use of clocks to enforce consistency in the cloud. *IEEE Data Eng. Bull.*, 38(1), 2015.

[16] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In *POPL*, 2014.

[17] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP*, 2012.

[18] A. M. Dan, Y. Meshman, M. T. Vechev, and E. Yahav. Predicate abstraction for relaxed memory models. In *SAS*, 2013.

[19] A. M. Dan, Y. Meshman, M. T. Vechev, and E. Yahav. Effective abstractions for verification under relaxed memory models. In *VMCAI*, 2015.

[20] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[21] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.

[22] A. Fekete. Allocating isolation levels to transactions. In *PODS*, 2005.

[23] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, 2010.

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

[25] A. Gotsman, N. Rinetzky, and H. Yang. Verifying concurrent memory reclamation algorithms with grace. In *ESOP*, 2013.

[26] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*. North-Holland, 1983.

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

[28] C. Li, J. Leitão, A. Clement, N. Preguiça, and R. Rodrigues. Minimizing coordination in replicated systems. In *Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC)*, 2015.

[29] C. Li, J. Leitão, A. Clement, N. M. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *USENIX ATC*, 2014.

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

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

[32] S. Lu, A. J. Bernstein, and P. M. Lewis. Correct execution of transactions at different isolation levels. *IEEE Trans. Knowl. Data Eng.*, 16(9), 2004.

[33] Microsoft. Consistency levels in DocumentDB. http://azure.microsoft.com/en-us/documentation/articles/documentdb-consistency-levels/, 2015.

[34] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theor. Comput. Sci.*, 375(1-3), 2007.

[35] F. Pedone and A. Schiper. Generic broadcast. In *DISC*, 1999.

[36] M. Saeida Ardekani, P. Sutra, and M. Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *SRDS*, 2013.

[37] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011.

[38] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, 2011.

[39] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan. Declarative programming over eventually consistent data stores. In *PLDI*, 2015.

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

[41] 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 *SOSP*, 2013.

[42] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, 2014.

[43] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*, 2013.

[44] W. Vogels. Eventually consistent. *CACM*, 52(1), 2009.