

Inria



AntidoteDB: a planet scale highly-available transactional database

Sara S. Hamouda, Sorbonne-Université-LIP6 & INRIA
sara.hamouda@inria.fr

ANU CECS Seminar, Canberra, Australia, 9/12/2019

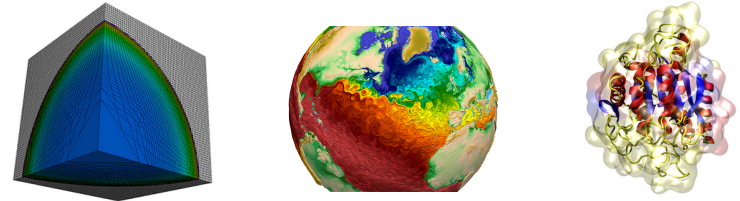
University of Sydney, Faculty of Engineering, Sydney, Australia, 20/12/2019



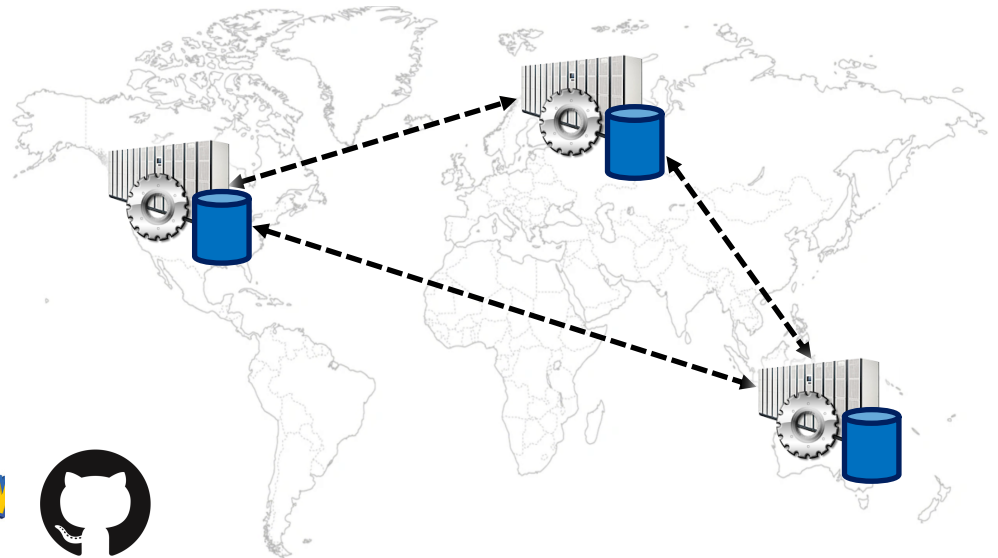


Gadi: NCI's new supercomputer

From HPC Applications

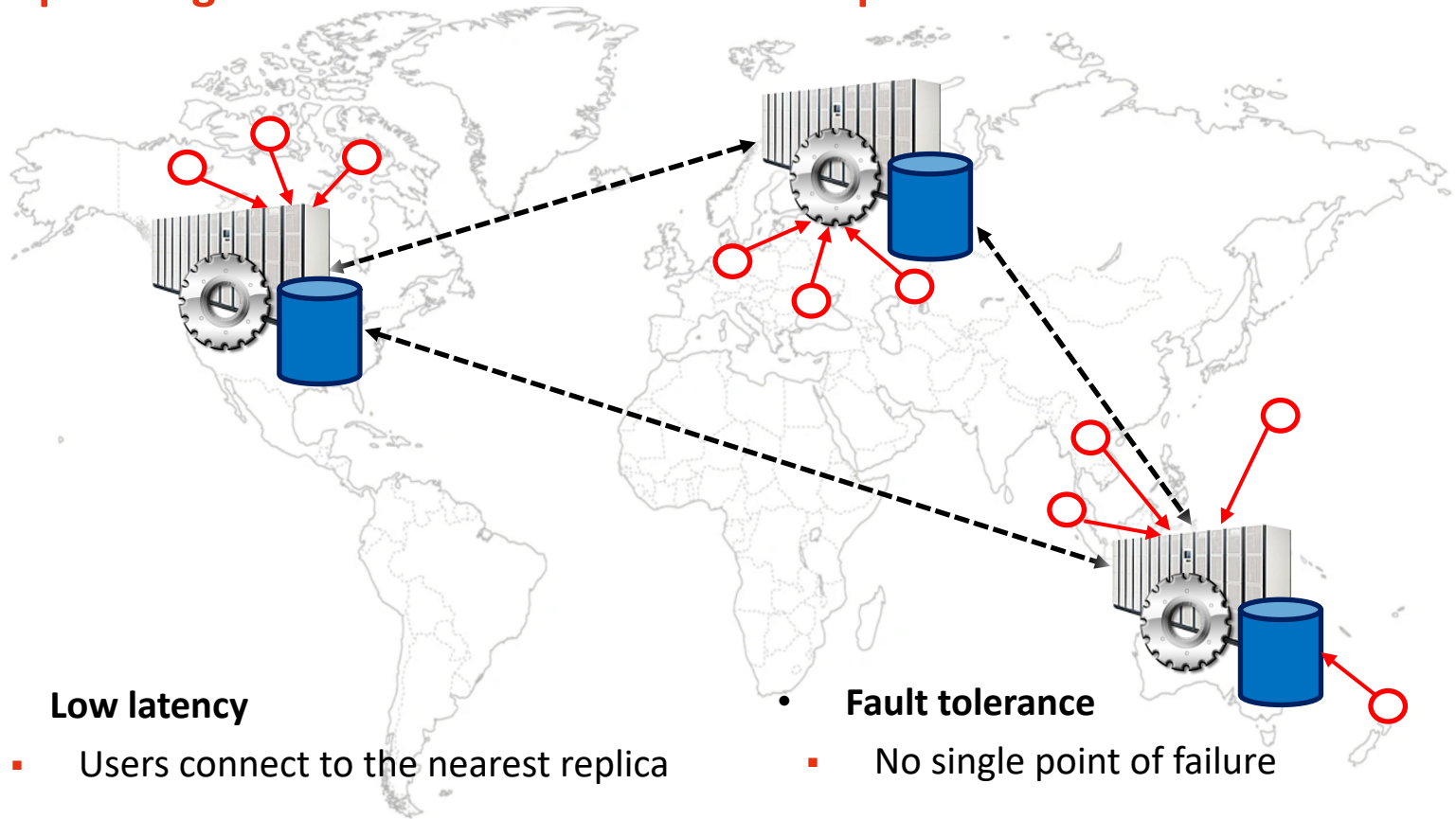


To Distributed Cloud Applications



Geo-Replication

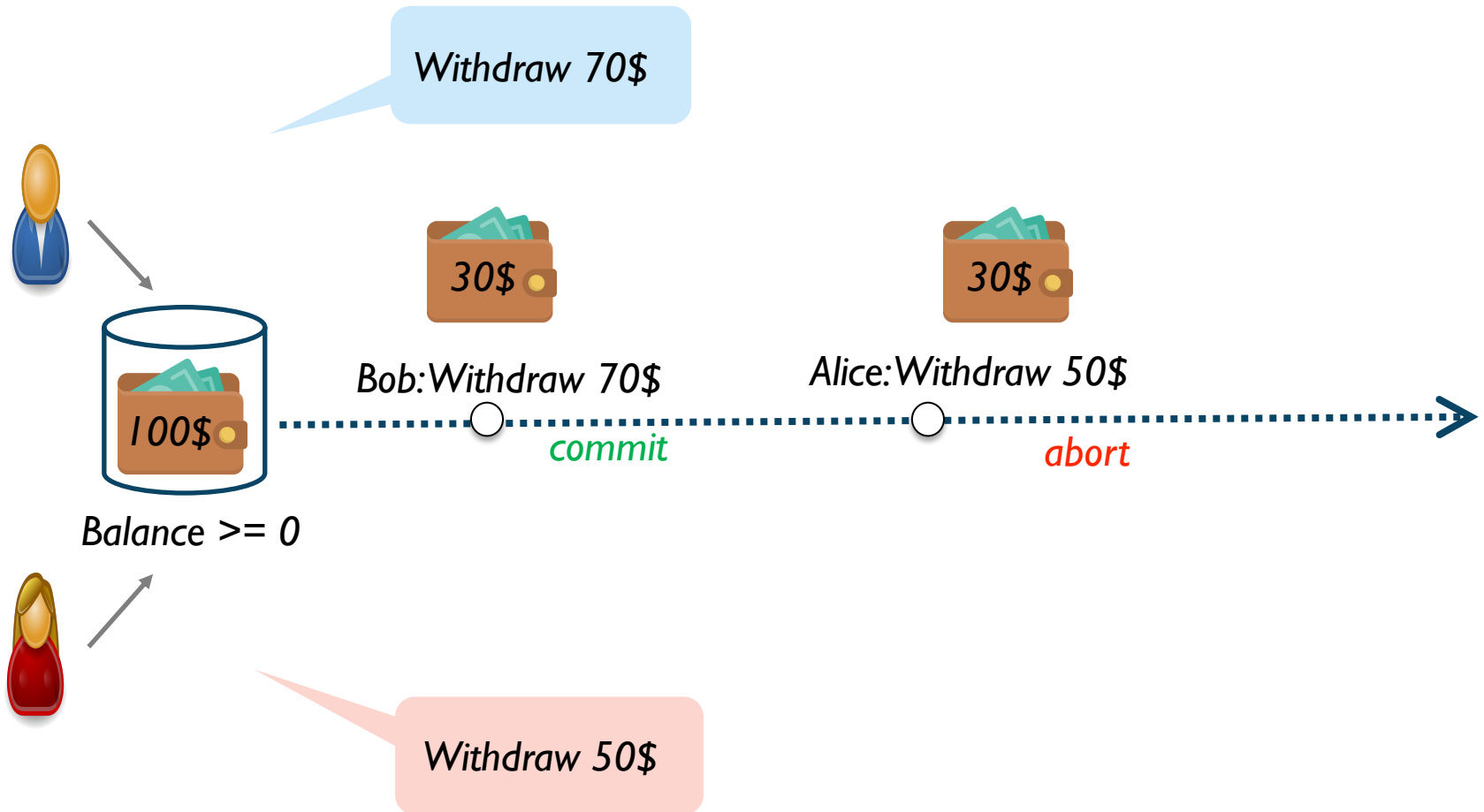
Replicating the entire service across multiple data centers



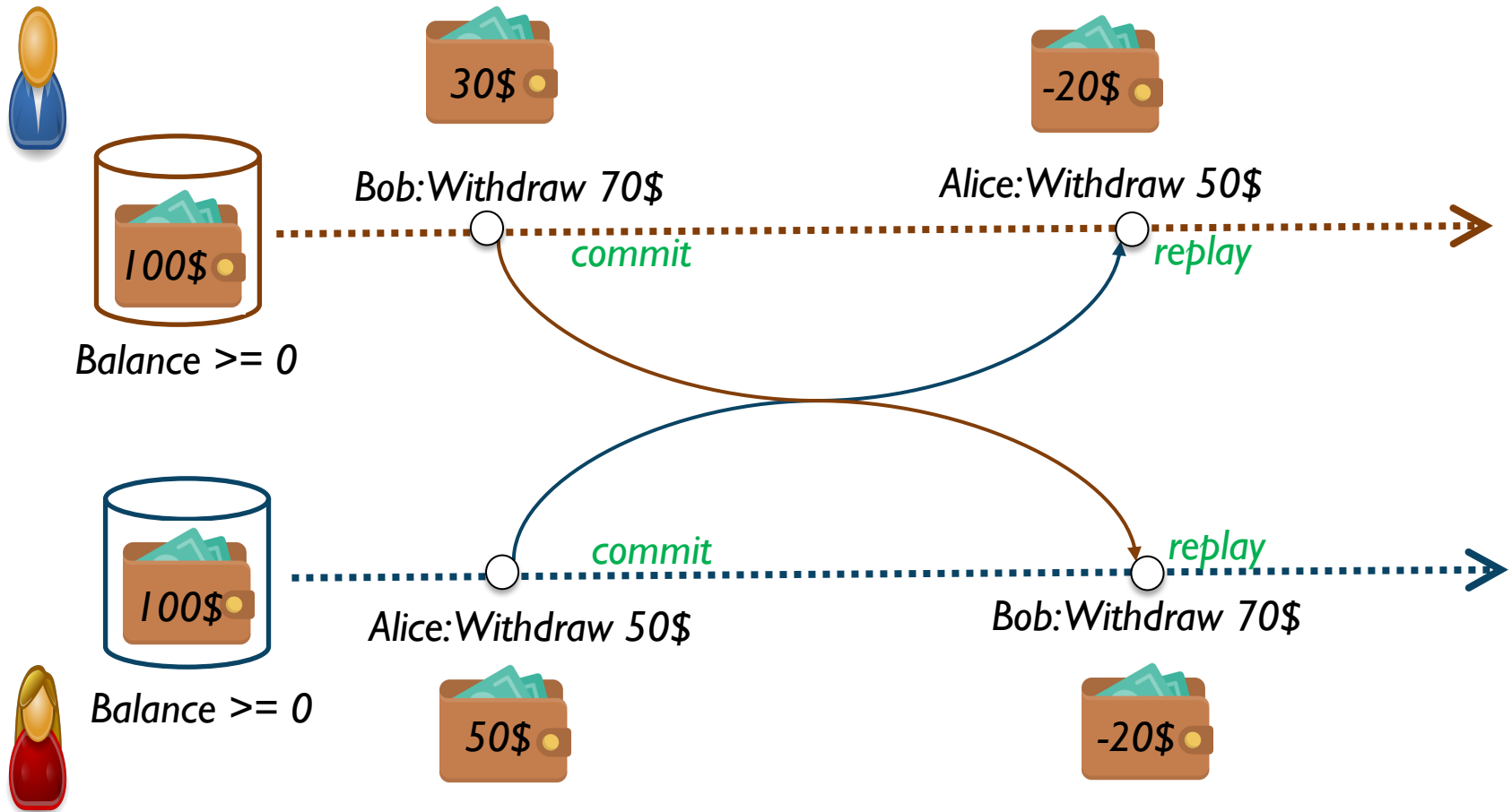
- **Low latency**
 - Users connect to the nearest replica

- **Fault tolerance**
 - No single point of failure

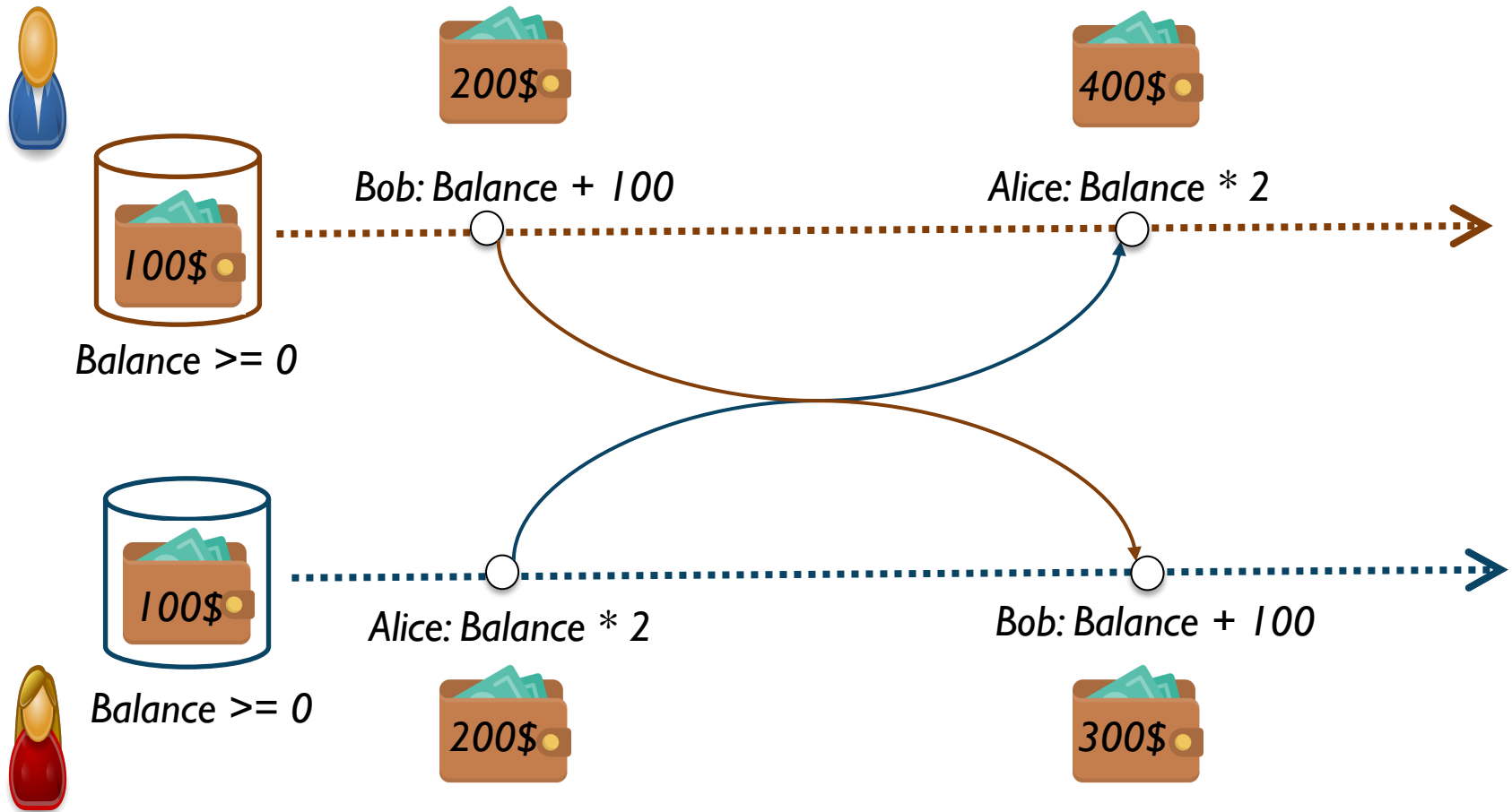
Consistency: Integrity Constraints



Consistency: Integrity Constraints



Consistency: Ordering Anomalies



The Problem with Concurrent Writes

- **Synchronous writes**
 - Slow
 - Replicas are always consistent
- **Asynchronous writes**
 - Fast
 - Replicas may diverge

AntidoteDB Research Aim



- **Goal**

- Geo-replicated objects
- Fast reads and writes
- Strong convergence guarantees
- Easy to program

- **Contributions**

- Strong Eventual Consistency
- Conflict-free Replicated Data Types
- Transactional Causal Consistency (TCC)





AntidoteDB

Strong Eventual Consistency

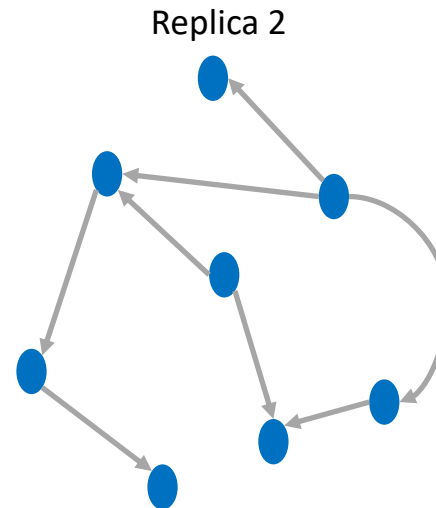
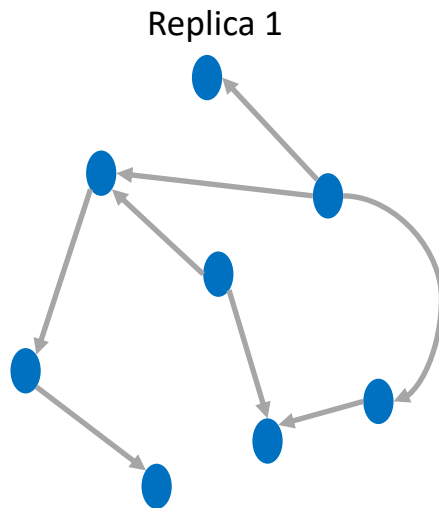
Let's consider a replicated graph

State:

Nodes, Edges

Operations:

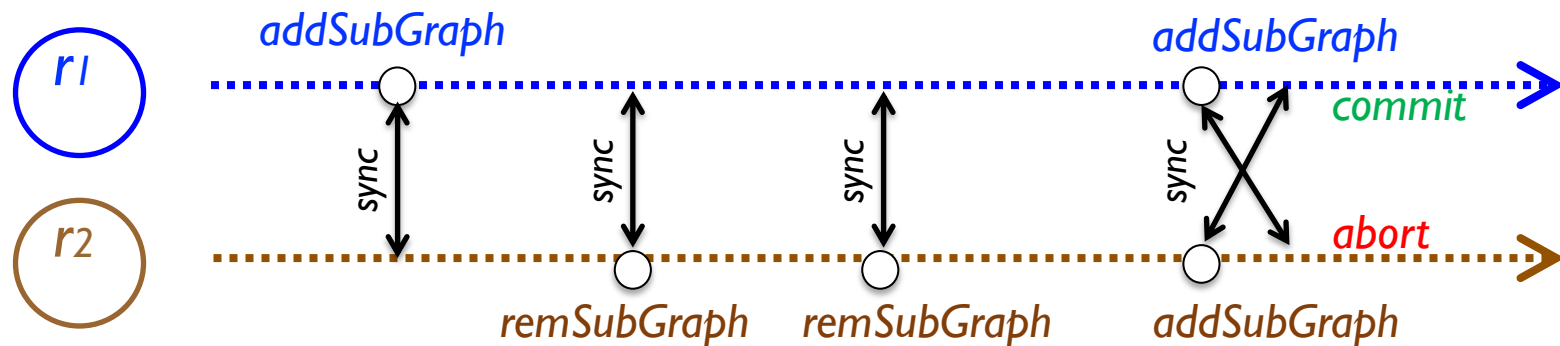
- addSubGraph
- remSubGraph



Slide courtesy of Marc Shapiro.

Strong Consistency

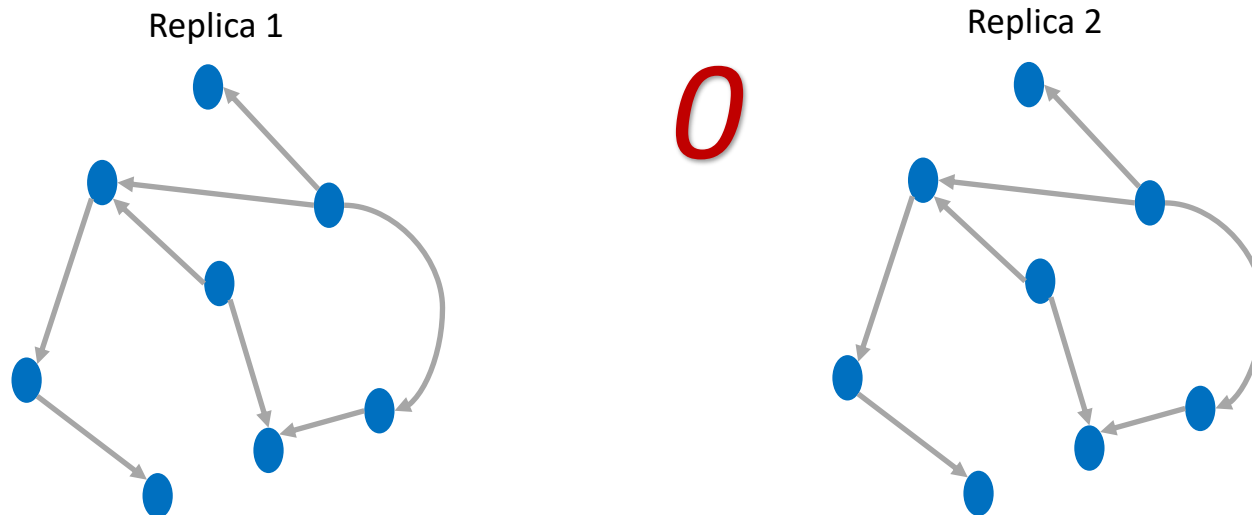
- Mimics a centralized database behaviour by synchronizing all writes (using a consensus protocol like Paxos).



Slide courtesy of Marc Shapiro.

Strong Consistency

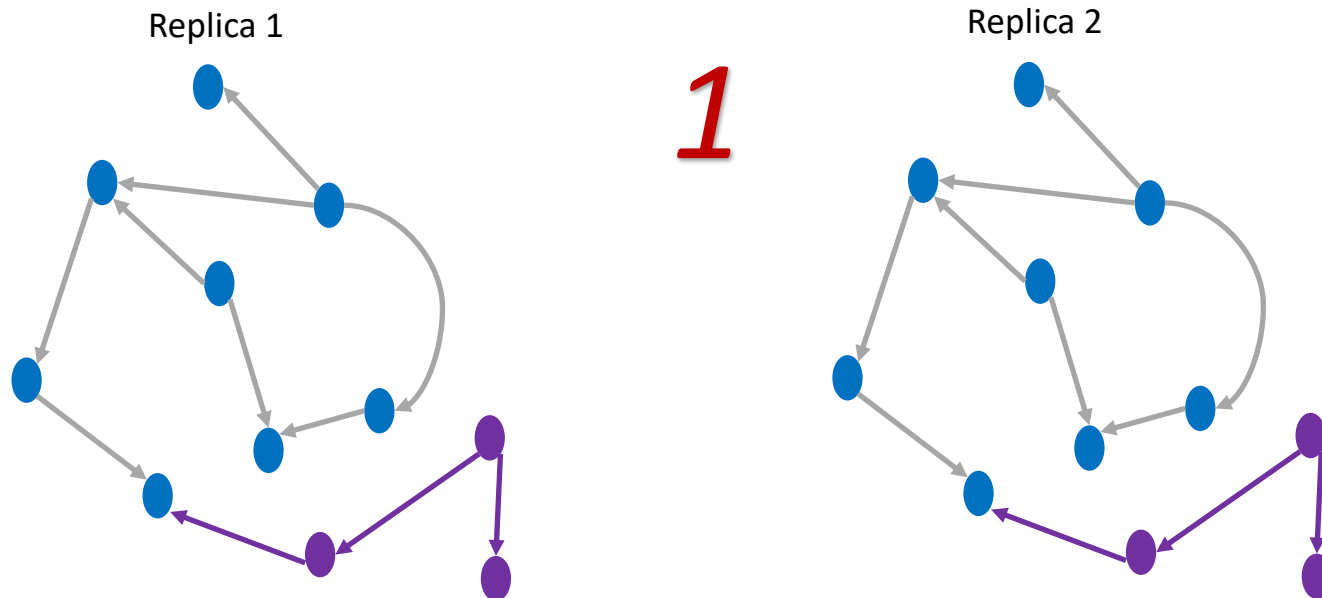
- Mimics a centralized database behaviour by synchronizing all writes (using a consensus protocol like Paxos).



Slide courtesy of Marc Shapiro.

Strong Consistency

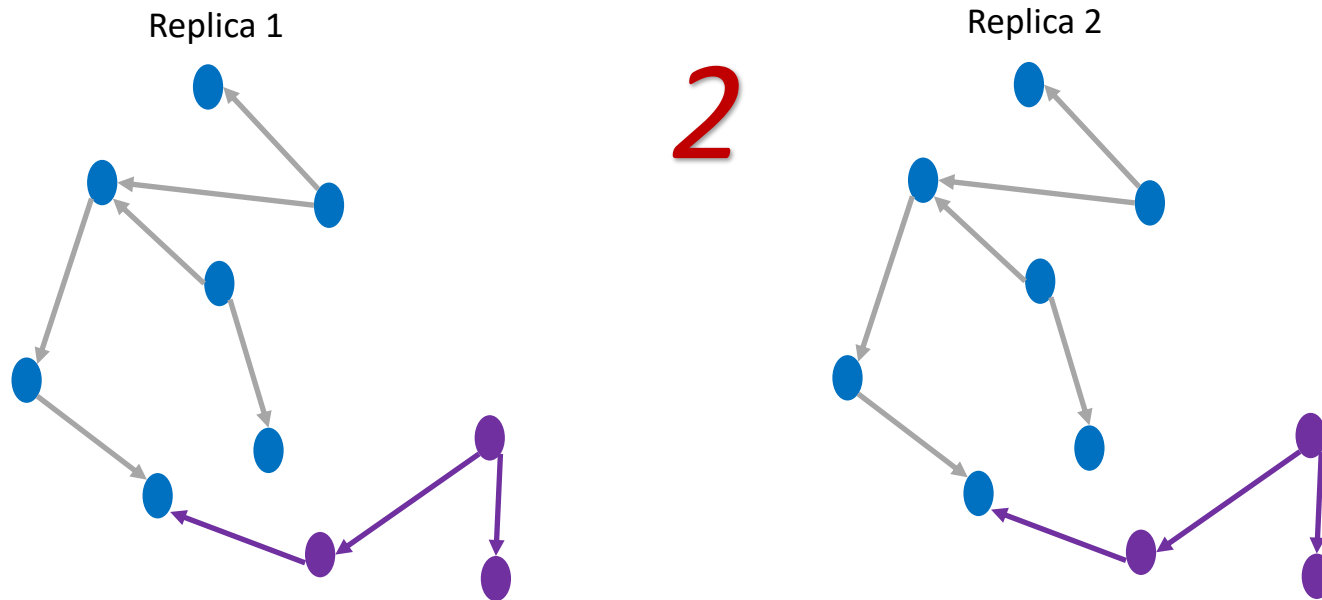
- Mimics a centralized database behaviour by synchronizing all writes (using a consensus protocol like Paxos).



Slide courtesy of Marc Shapiro.

Strong Consistency

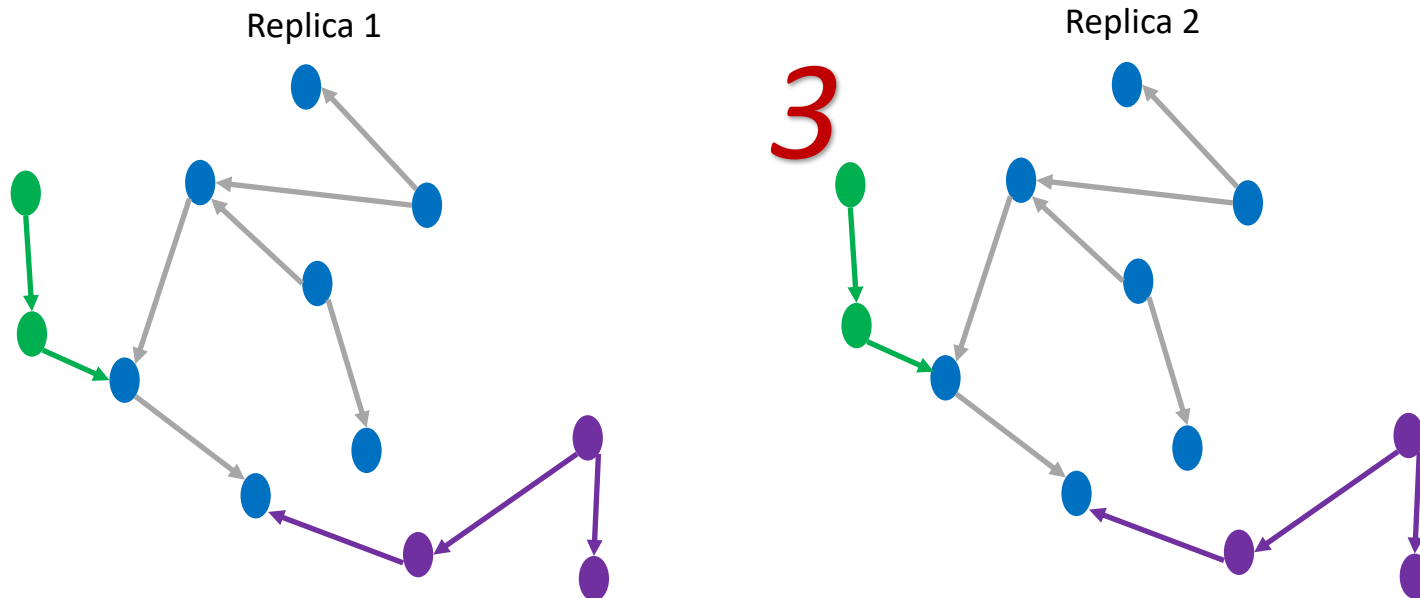
- Mimics a centralized database behaviour by synchronizing all writes (using a consensus protocol like Paxos).



Slide courtesy of Marc Shapiro.

Strong Consistency

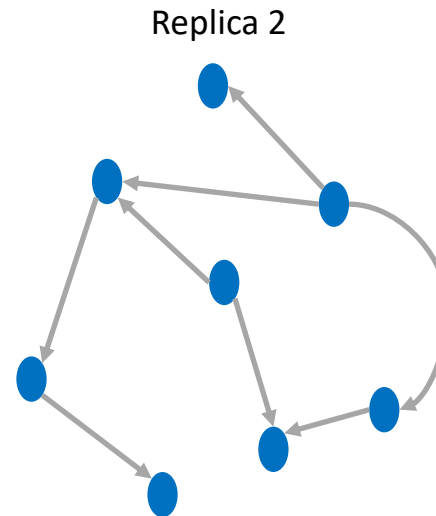
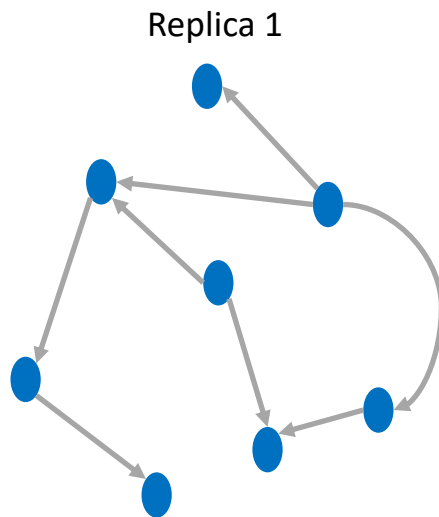
- Mimics a centralized database behaviour by synchronizing all writes (using a consensus protocol like Paxos).
 - Slow and unavailable under network partition.
 - + Easy to program - replication is almost transparent.



Slide courtesy of Marc Shapiro.

Eventual Consistency

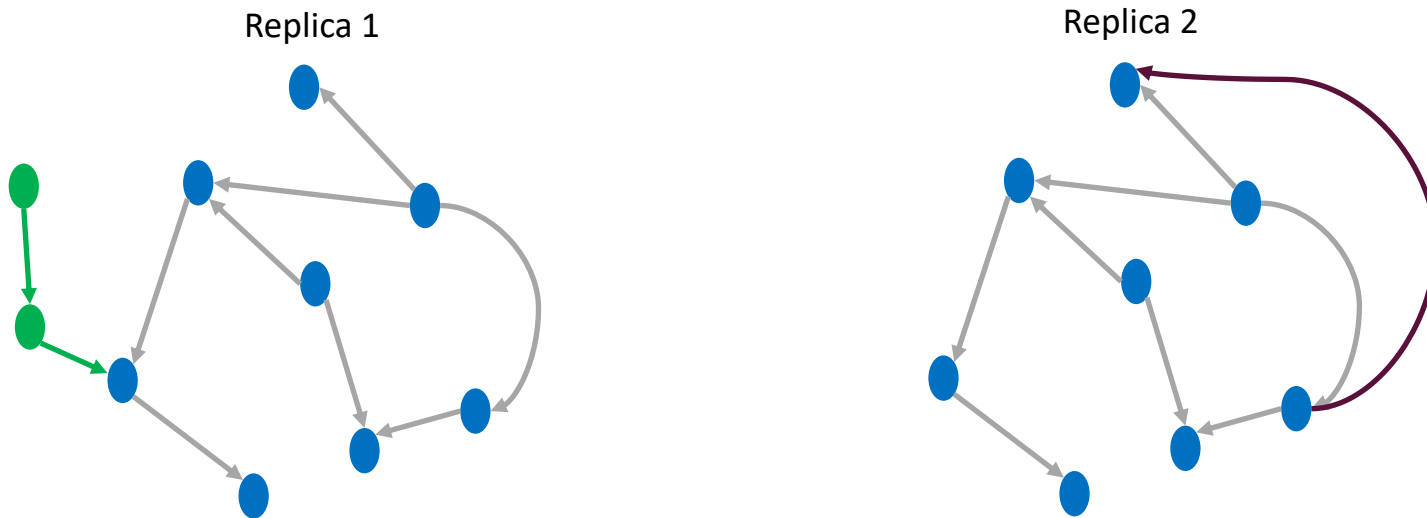
- Update locally, propagate asynchronously.
- On conflict: consensus in the background, rollback, or arbitrate.



Slide courtesy of Marc Shapiro.

Eventual Consistency

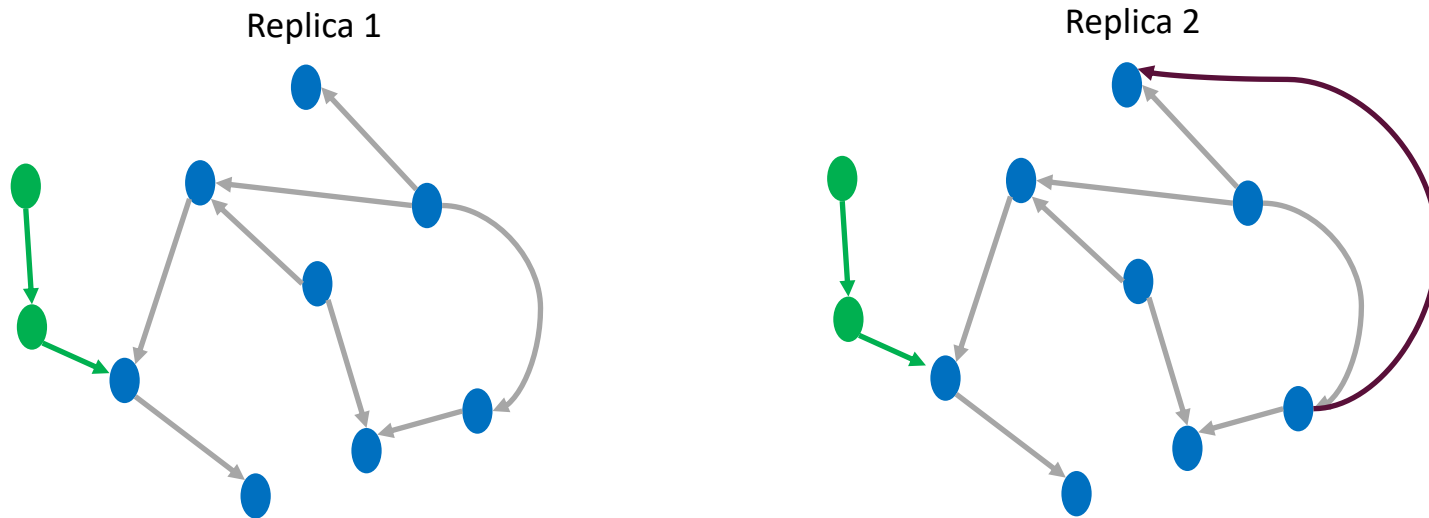
- Update locally, propagate asynchronously.
- On conflict: consensus in the background, rollback, or arbitrate.



Slide courtesy of Marc Shapiro.

Eventual Consistency

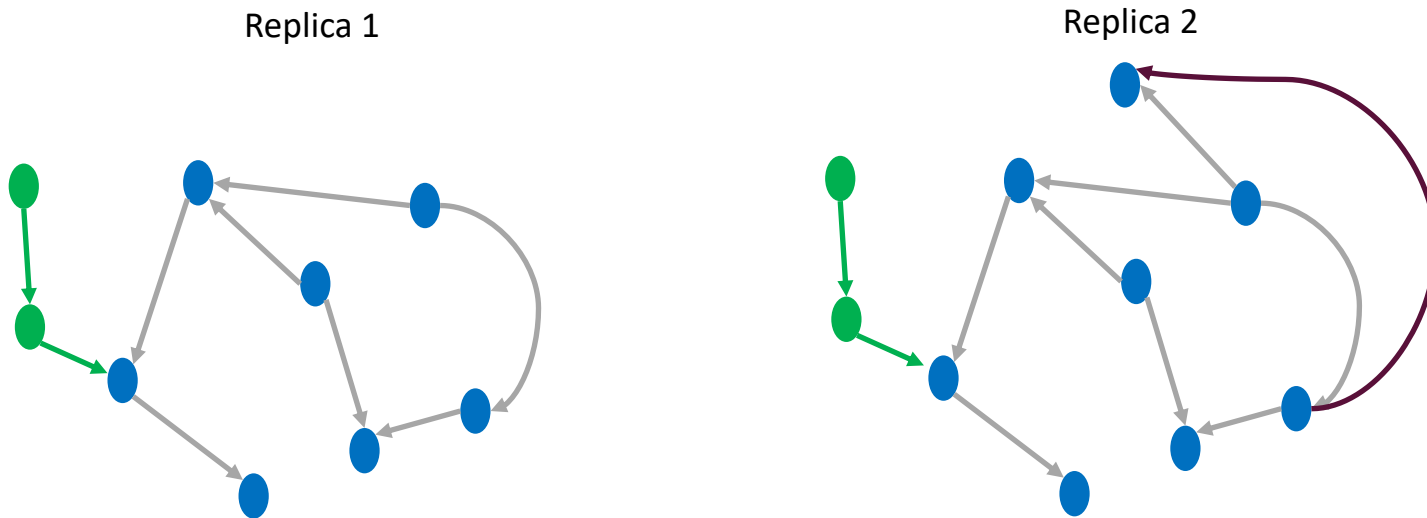
- Update locally, propagate asynchronously.
- On conflict: consensus in the background, rollback, or arbitrate.



Slide courtesy of Marc Shapiro.

Eventual Consistency

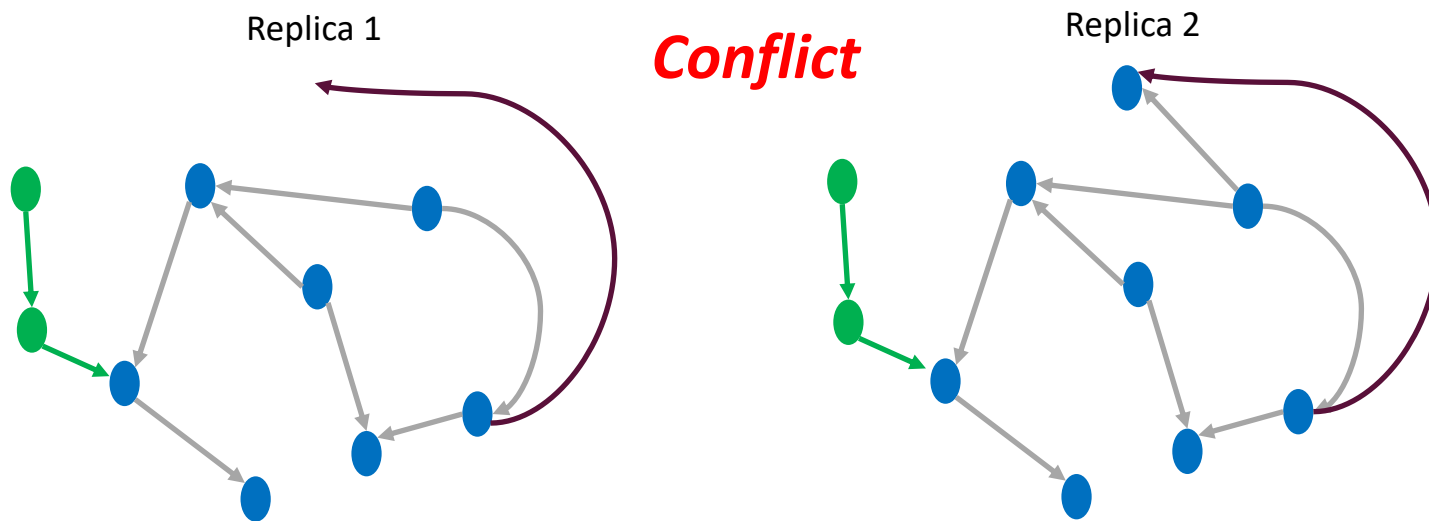
- Update locally, propagate asynchronously.
- On conflict: consensus in the background, rollback, or arbitrate.



Slide courtesy of Marc Shapiro.

Eventual Consistency

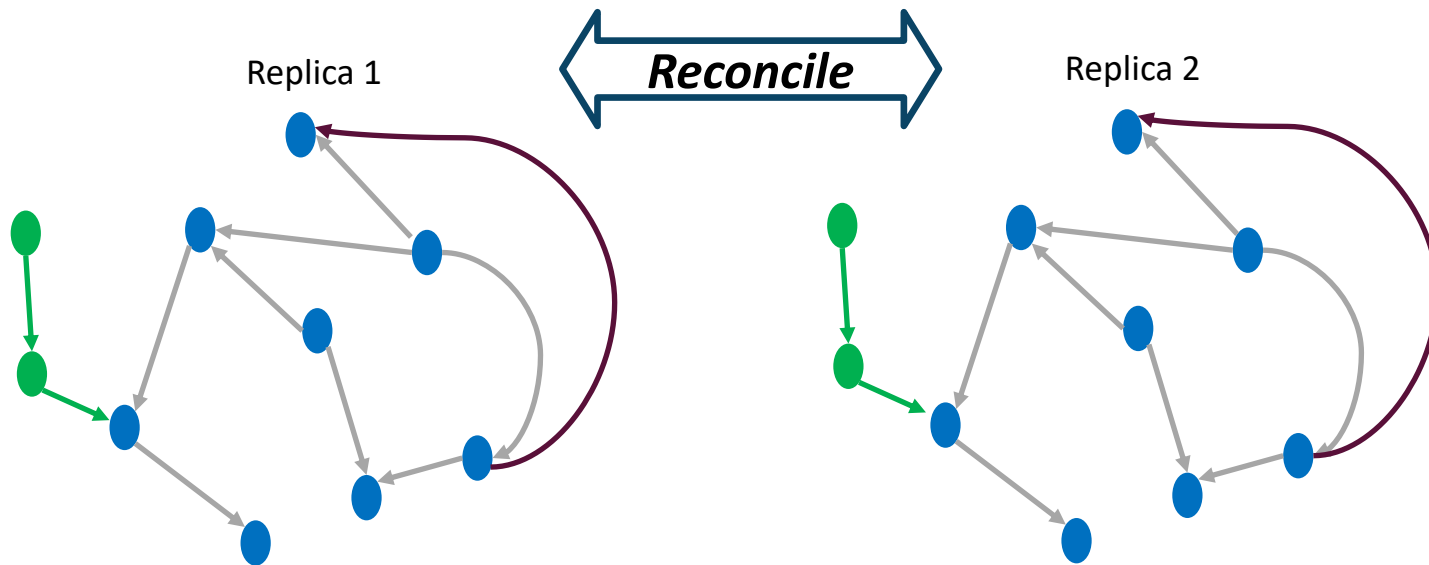
- Update locally, propagate asynchronously.
- On conflict: consensus in the background, rollback, or arbitrate.



Slide courtesy of Marc Shapiro.

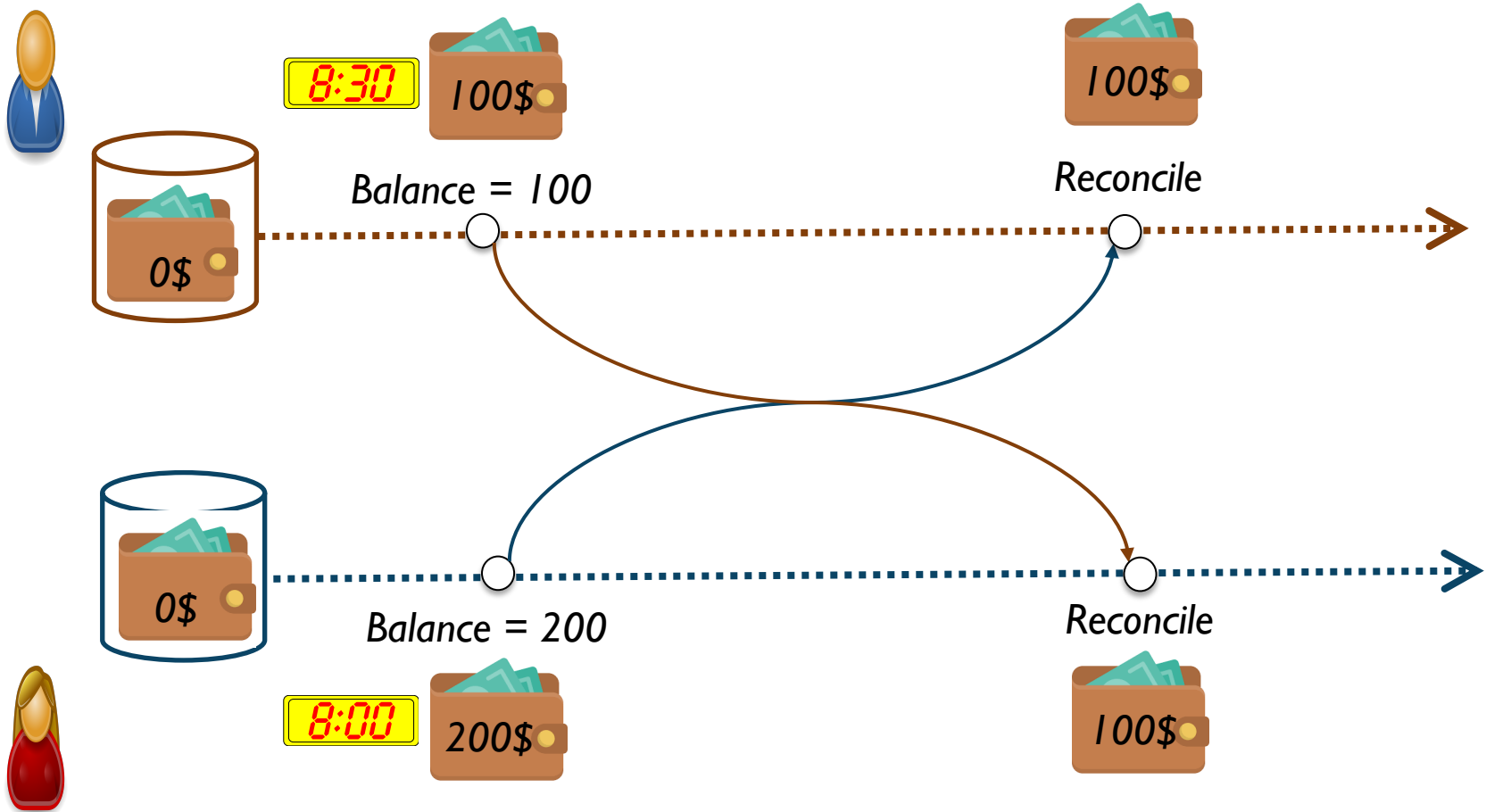
Eventual Consistency

- Update locally, propagate asynchronously.
- On conflict: consensus in the background, rollback, or arbitrate.



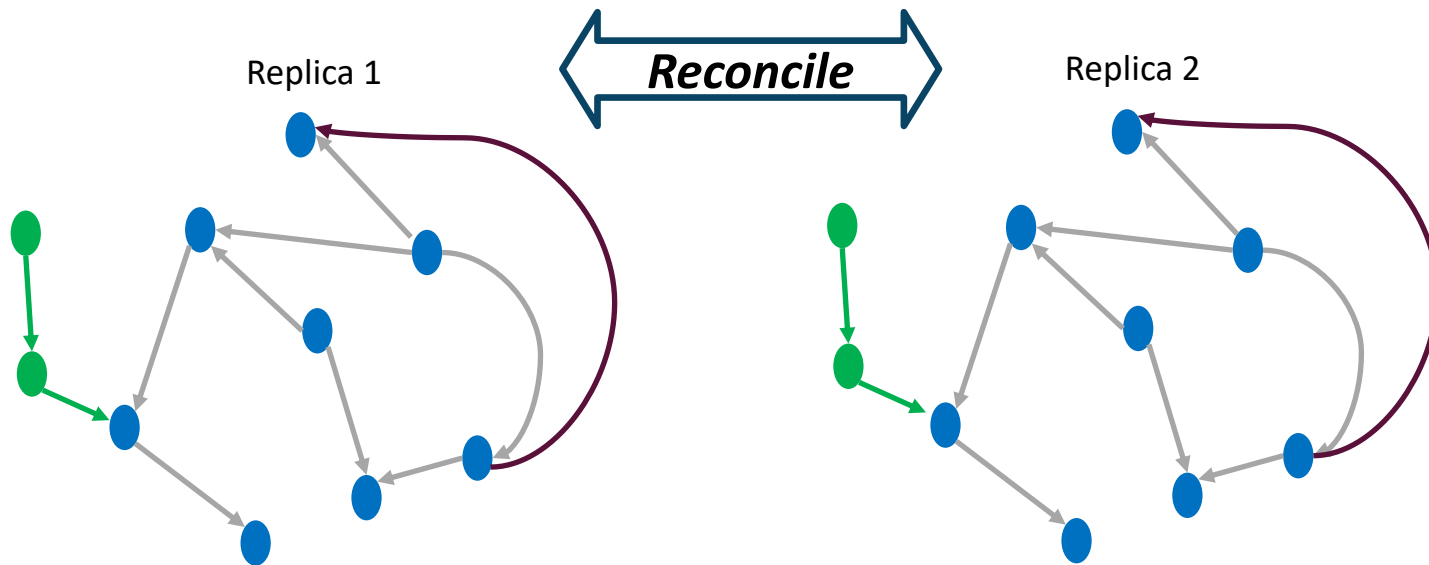
Slide courtesy of Marc Shapiro.

LWW: Last Writer Wins



Eventual Consistency

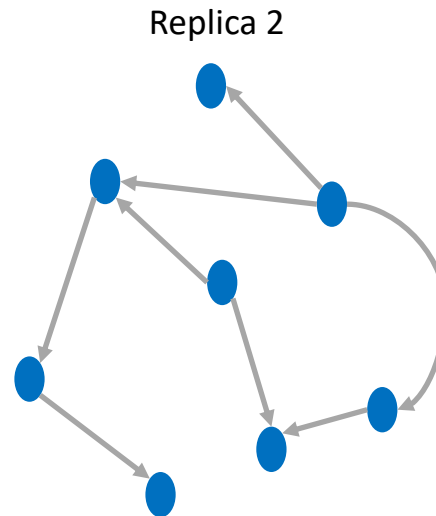
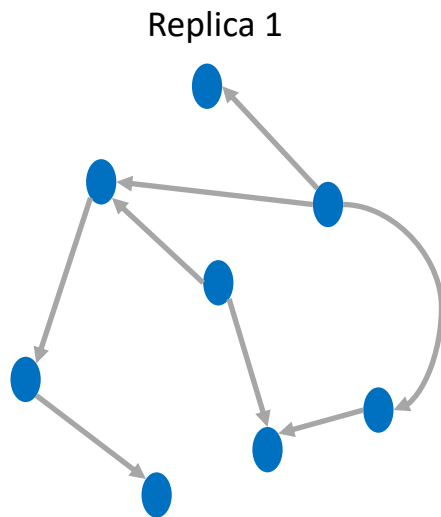
- Update locally, propagate asynchronously.
- On conflict: consensus in the background, rollback, or arbitrate.
- Conflict resolution: ad-hoc mechanisms, unclear semantics.



Slide courtesy of Marc Shapiro.

Strong Eventual Consistency

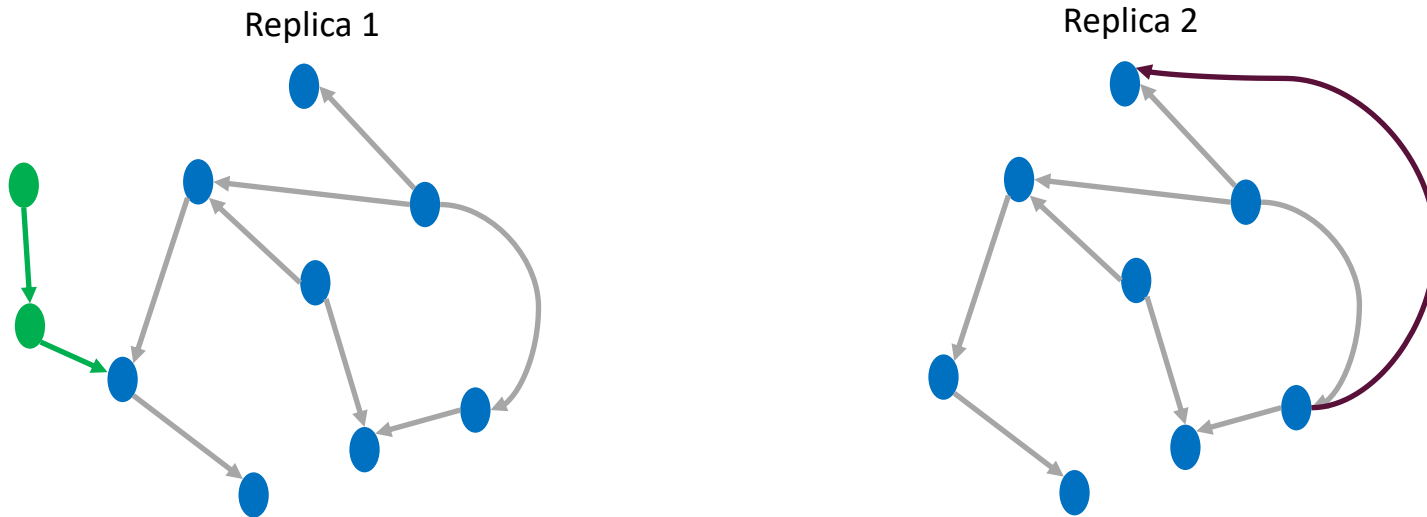
- Update locally, propagate asynchronously.
- Conflict-free objects: local deterministic conflict resolution.
- No consensus, no rollback.



Slide courtesy of Marc Shapiro.

Strong Eventual Consistency

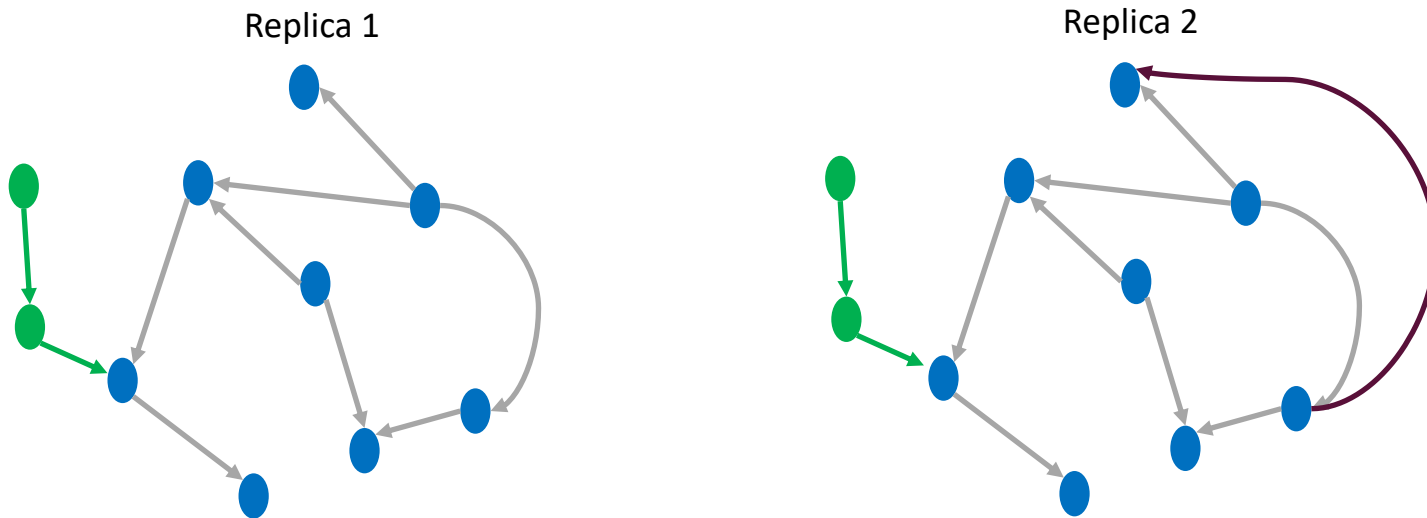
- Update locally, propagate asynchronously.
- Conflict-free objects: local deterministic conflict resolution.
- No consensus, no rollback.



Slide courtesy of Marc Shapiro.

Strong Eventual Consistency

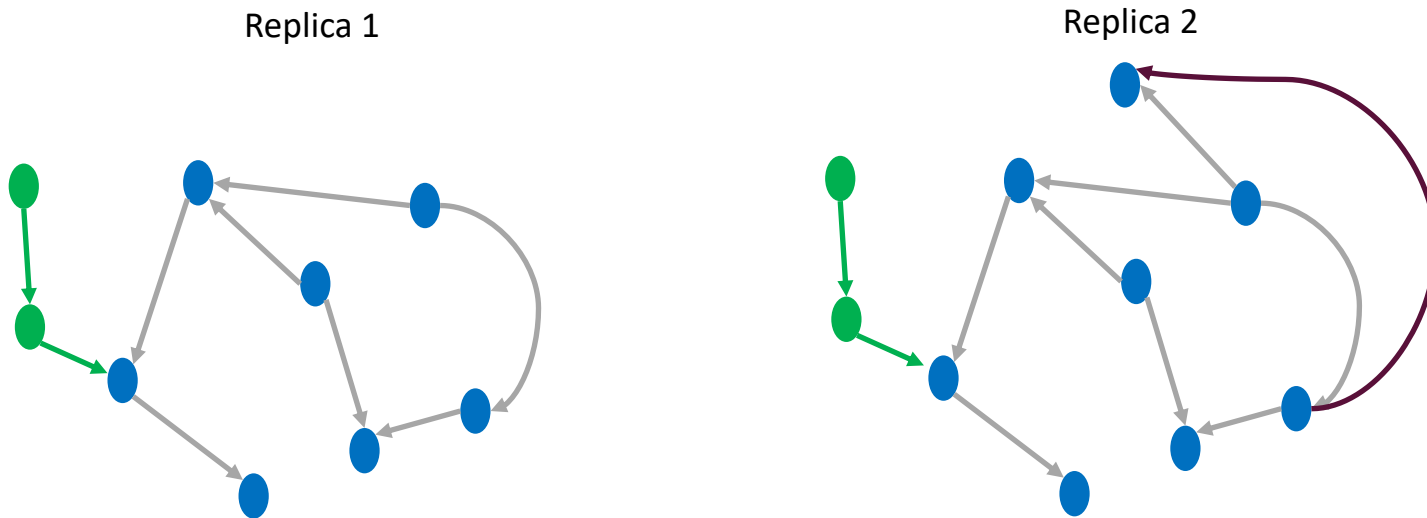
- Update locally, propagate asynchronously.
- Conflict-free objects: local deterministic conflict resolution.
- No consensus, no rollback.



Slide courtesy of Marc Shapiro.

Strong Eventual Consistency

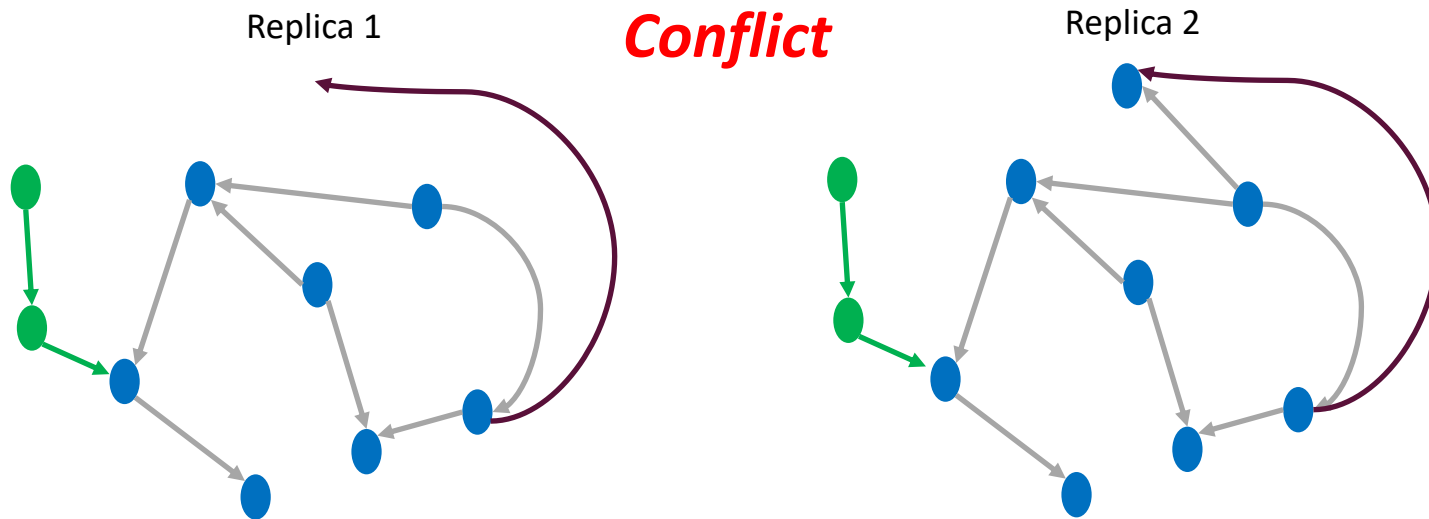
- Update locally, propagate asynchronously.
- Conflict-free objects: local deterministic conflict resolution.
- No consensus, no rollback.



Slide courtesy of Marc Shapiro.

Strong Eventual Consistency

- Update locally, propagate asynchronously.
- Conflict-free objects: local deterministic conflict resolution.
- No consensus, no rollback.



Slide courtesy of Marc Shapiro.

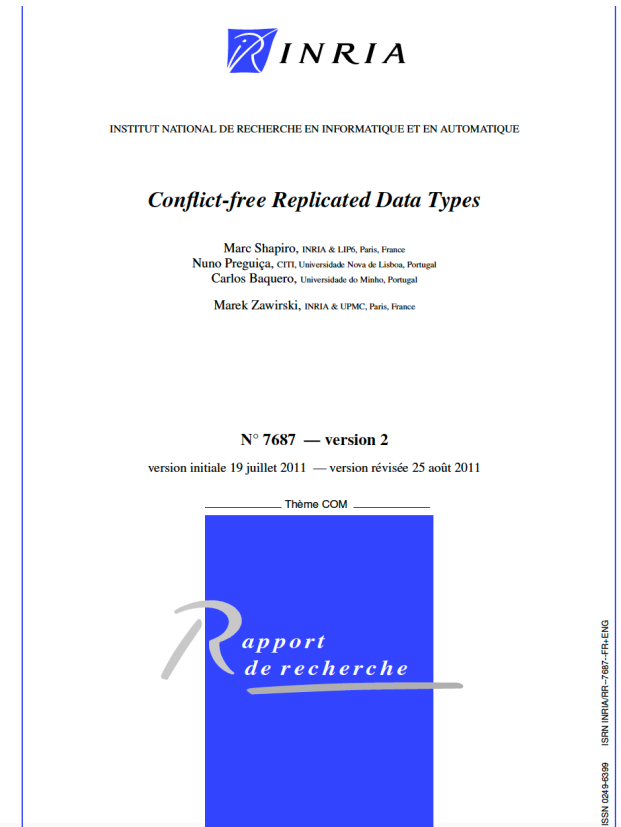
Conflict-free replicated data types

[M Shapiro](#), [N Preguiça](#), [C Baquero](#)... - Symposium on Self ..., 2011 - Springer

... **Replicating data** under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation ... refer the interested reader to a separate technical report [18] for further detail and for a **comprehensive** portfolio of ... **Conflict-Free Replicated Data Types** 395 ...

☆  Cited by 538 [Related articles](#) [All 33 versions](#)

« We propose a simple, theoretically-sound approach to eventual consistency. Our system model, Strong Eventual Consistency or SEC, avoids the complexity of conflict resolution and of roll-back. Conflict-freedom ensures safety and liveness despite any number of failures. »





AntidoteDB

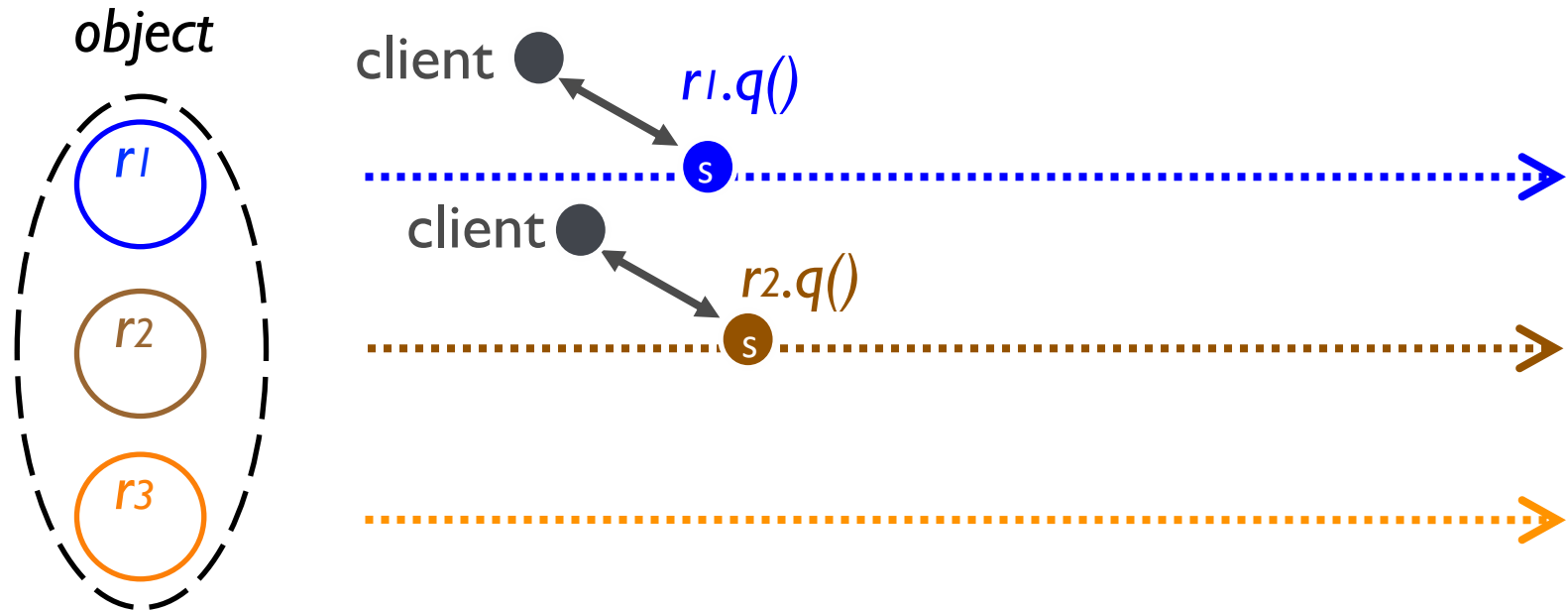
Conflict-free Replicated Data Types (CRDTs)

Basic Concepts

- Read local replica
- Update local replica, transmit later
- Deterministic conflict resolution

Slide courtesy of Marc Shapiro.

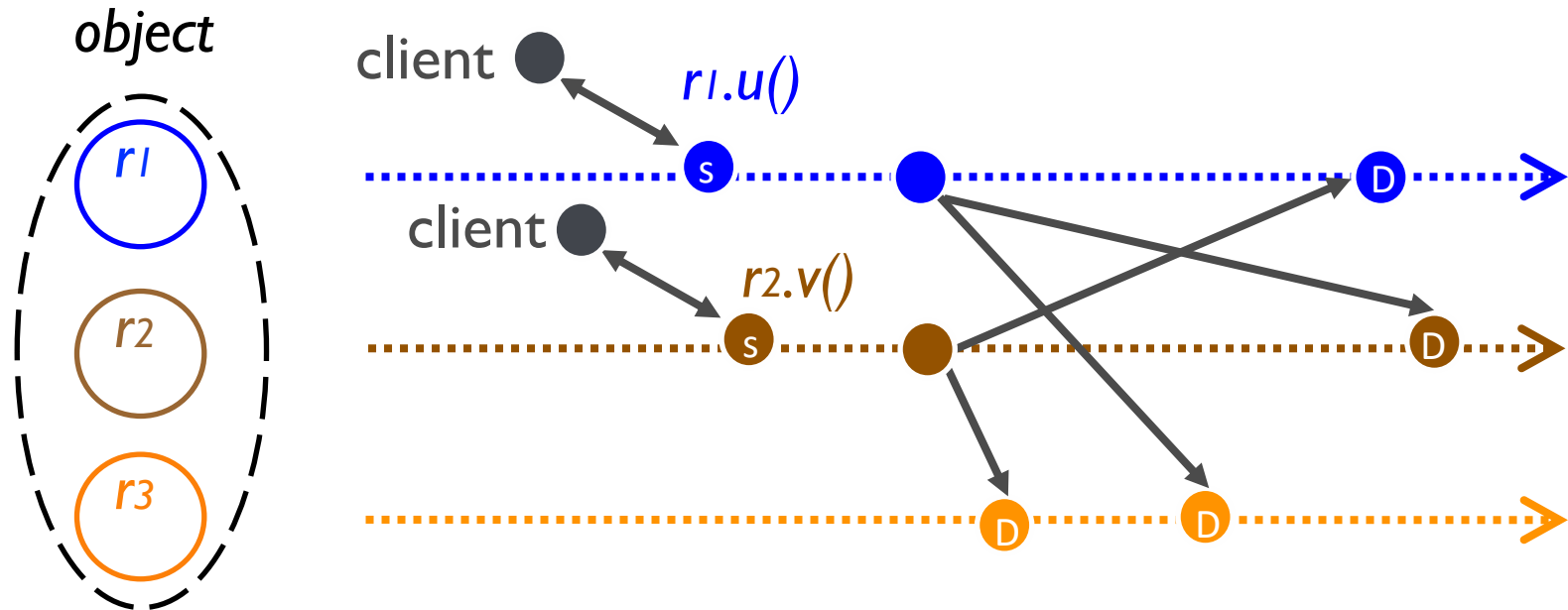
Query



- Query local replica
- Clients connect to any replica

Slide courtesy of Marc Shapiro.

Update and Transmit



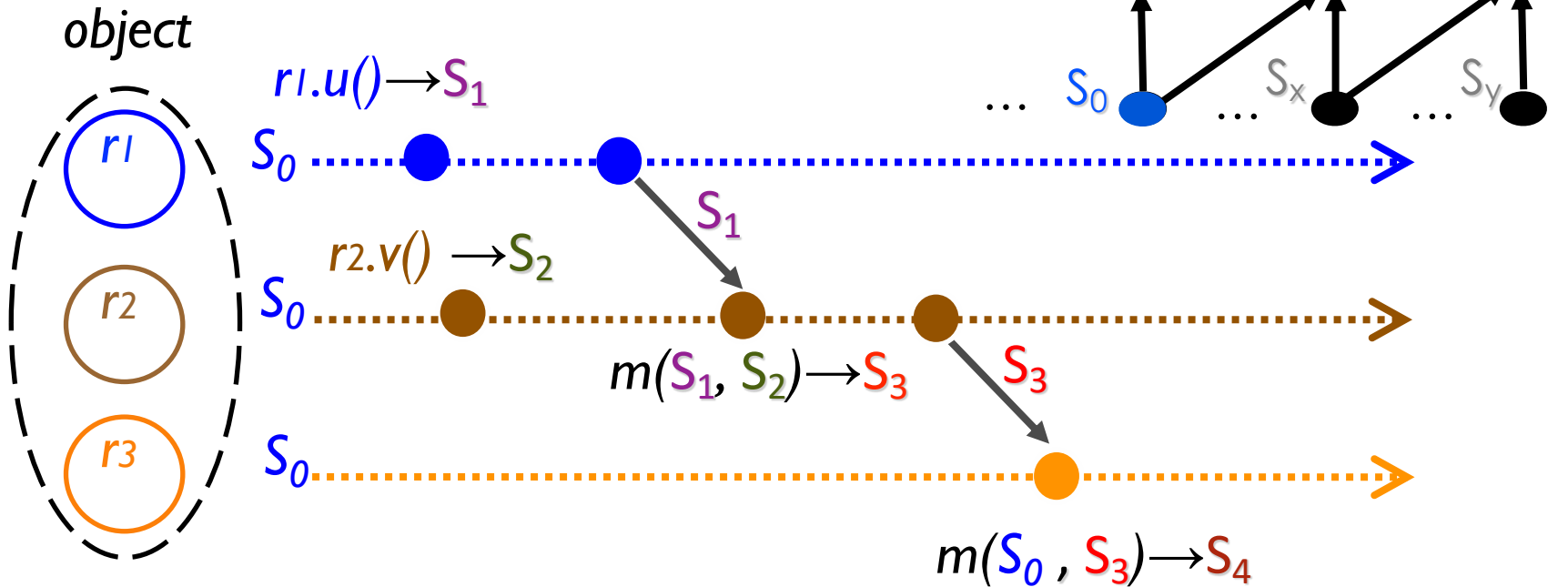
- Update source replica
- Transmit to downstream replicas later
- Receiver applies update

Slide courtesy of Marc Shapiro.

Replication Models

- **State-based Replication**
 - Source replica propagates **full state**
 - Downstream replicas **merge states**
- **Operation-based Replication**
 - Source replica propagates **functions**
 - Downstream replicas **replay received functions**

State-based Replication



- **Convergence: sufficient condition**
 - States form a monotonic semi-lattice
 - Merge computes Least Upper Bound

Slide courtesy of Marc Shapiro.

Example: State-based Repl.

Grow-Only Counter

- **Increment**

- Payload: $P = [0, 0, \dots]$

- $\text{value}() = \sum_i P[i]$

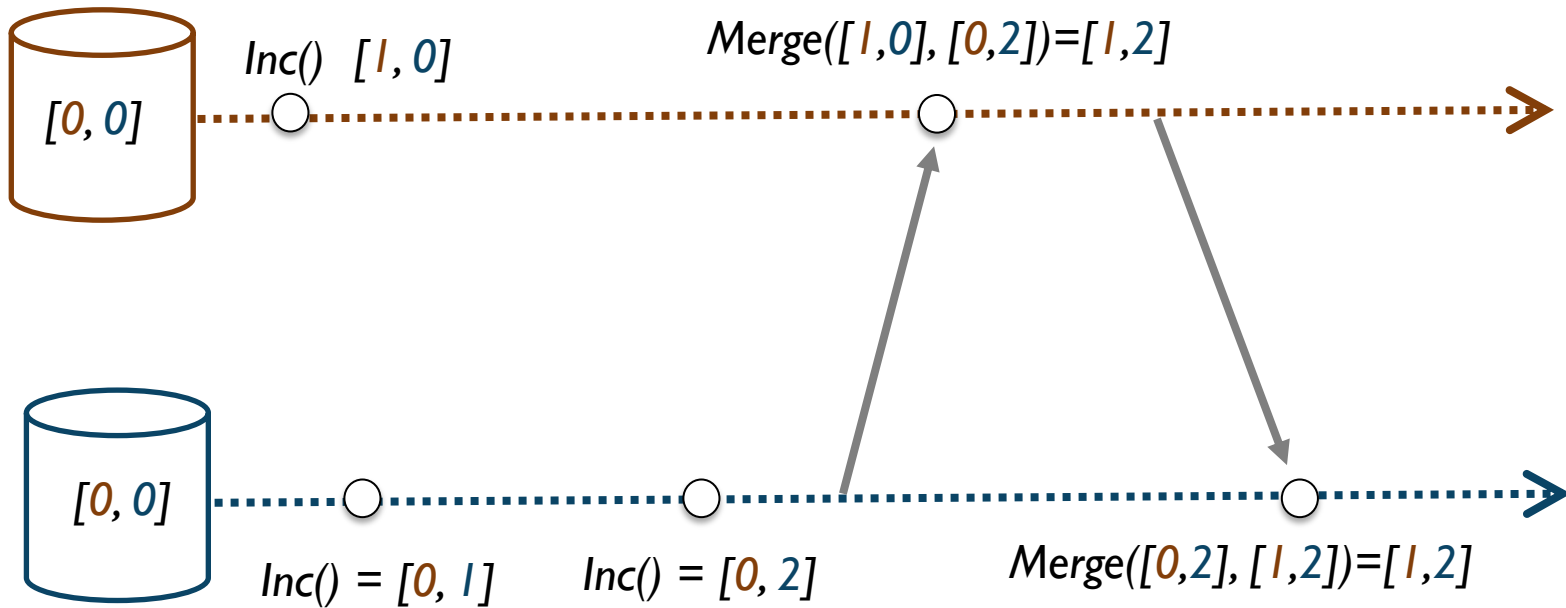
- $\text{increment}() = P[\text{MyRepID}]++$

- $\text{merge}(S_1, S_2) =$

$$P = [\dots, \max(S_1.P[i], S_2.P[i]), \dots]$$

Example: State-based Repl.

Grow-Only Counter



Example: State-based Repl.

Positive-Negative Counter

- **Increment / decrement**

- Payload: $P = [0, 0, \dots],$

- $N = [0, 0, \dots]$

- $\text{value}() = \sum_i P[i] - \sum_i N[i]$

- $\text{increment}() = P[\text{MyRepID}]++$

- $\text{decrement}() = N[\text{MyRepID}]++$

- $\text{merge}(S_1, S_2) =$

- $P = [\dots, \max(S_1.P[i], S_2.P[i]), \dots],$

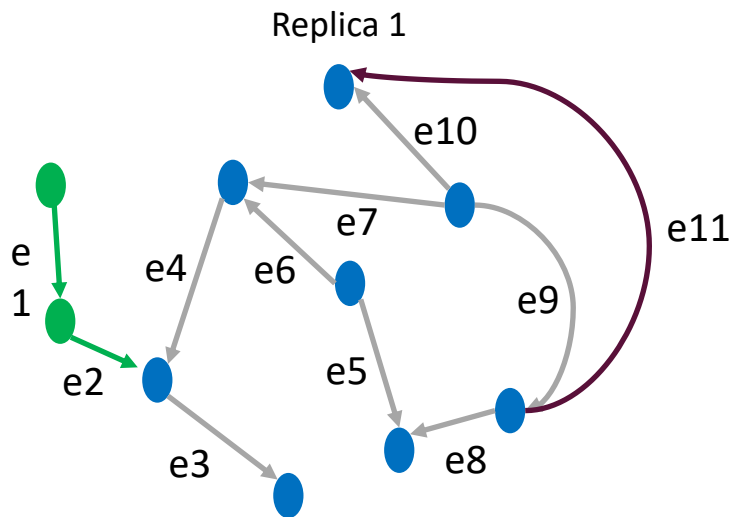
- $N = [\dots, \max(S_1.N[i], S_2.N[i]), \dots]$

Example: State-based Repl.

Graph

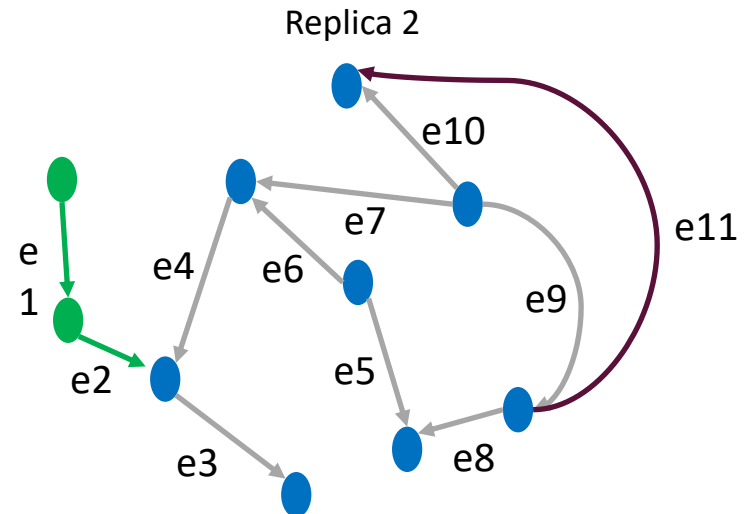
State (replica1):

- Edges = { e1, e2, e3, e4, e5, e6, e7, e8, e9, e10 }
- Deleted = { e11 }



State (replica2):

- Edges = { e1, e2, e3, e4, e5, e6, e7, e8, e9, e10 }
- Deleted = { e11 }



Sreeja S Nair
Sorbonne Université—LIP6 & Inria,
Paris, France
sreeja.nair@lip6.fr

Gustavo Petri
ARM Research, Cambridge, UK
gustavo.petri@arm.com

Marc Shapiro
Sorbonne Université—LIP6 & Inria,
Paris, France
marc.shapiro@acm.org

Static Analysis for State- based CRDTs

ABSTRACT

We study a proof methodology for verifying the safety of data invariants of highly-available distributed applications that replicate state. The proof is (1) modular: one can reason about each individual operation separately, and (2) sequential: one can reason about a distributed application as if it were sequential. We automate the methodology and illustrate the use of the tool with a representative example.

KEYWORDS

Replicated data, Consistency, Automatic verification, Distributed application design, Tool support

ACM Reference Format:

Sreeja S Nair, Gustavo Petri, and Marc Shapiro. 2019. Invariant Safety for Distributed Applications. In *6th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '19)*, March 25, 2019, Dresden, Germany. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3301419.3323970>

1 INTRODUCTION

A distributed application often replicates its data to several locations, and accesses the closest available replica. Examples include social networks, multi-user games, co-operative engineering tools, collaborative editors, source control repositories, or distributed file systems. To ensure availability, an update must not synchronise across replicas; otherwise, when a network partition occurs, the system will block. Asynchronous updates may cause replicas to diverge or to violate the data invariants of the application.

To address the first problem, Conflict-free Replicated Data Types (CRDTs)[13] have mathematical properties to ensure that all replicas that have received the same set of updates converge to the same state [13]. To ensure availability, a CRDT replica executes both queries and updates locally and

immediately, without remote synchronisation. It propagates its updates to the other replicas asynchronously.

There are two basic approaches to update propagation: to propagate operations, or to propagate states. In the former approach, an update is first applied to some origin replica, then sent as an operation to remote replicas, which in turn apply it to update their local state. Operation-based CRDTs require the message delivery layer to deliver messages in causal order, exactly once; the set of replicas must be known.

In the latter approach, an update is applied to some origin replica. Occasionally, one replica sends its full state to some other replica, which merges the received state into its own. In turn, this replica will later send its own state to yet another replica. As long as every update eventually reaches every replica transitively, messages may be dropped, re-ordered or duplicated, and the set of replicas may be unknown. Replicas are guaranteed to converge if the set of states, as a result of updates and merge, forms a monotonic semi-lattice [13]. Due to these relaxed requirements, state-based CRDTs have better adoption [1]. They are the focus of this work.

As a running example, consider a simple auction system. The state of an auction consists of status, a set of bids, and a winner. This state is replicated at multiple servers; CRDTs ensures that all replicas eventually converge. Users at different locations can start an auction, place bids, close the auction, declare a winner, inspect the local replica, and observe if a winner is declared and who it is. All replicas will eventually agree on the same auction status, same set of bids and the same winner.

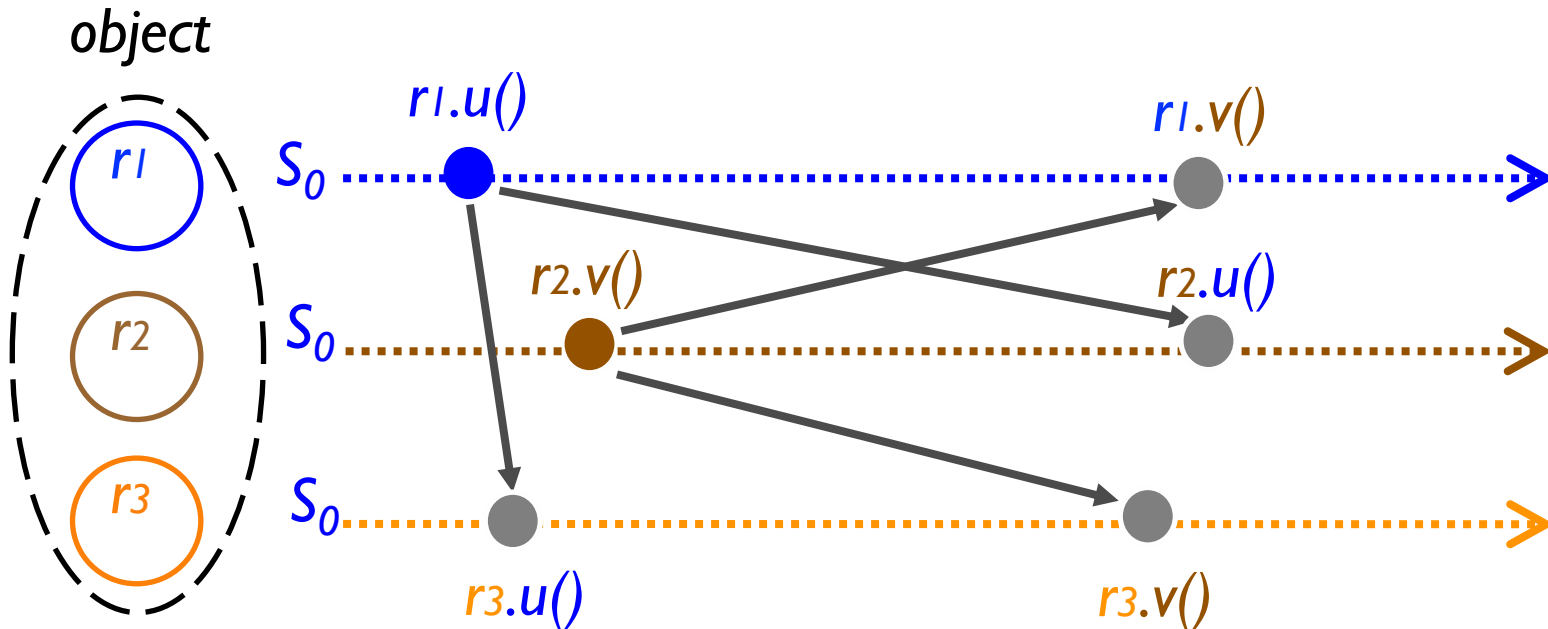
However, the application may also require to maintain a correctness property or *invariant* over the data. An invariant is an assertion on application data that must evaluate to true in every state of every replica. For instance, the auction's invariant is that: when the auction is closed, there is a winner; there is a single winner; and the winner's bid is the highest.

Such an invariant is easy to ensure in a sequential system, but concurrent updates might violate it. In this case, the application would need to synchronise some updates between replicas, in order to maintain the invariant. For instance, in the absence of sufficient synchronisation, a replica might close the auction and declare a winner, while concurrently a user at a different replica is placing a higher bid.

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

PaPoC '19, March 25, 2019, Dresden, Germany
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6276-4/19/03...\$15.00
<https://doi.org/10.1145/3301419.3323970>

Operation-based Replication



- **Convergence: sufficient condition**
 - ~~Reliable exact-once delivery~~
 - Operations must commute and be idempotent

Slide courtesy of Marc Shapiro.

Example: Operation-based Repl.

Grow-Only Set

Sequential specification of Set:

$\{true\}$ $add(e)$ $\{e \in S\}$

Commutative operations ($e \neq f$):

$\{true\}$ $add(e)$ || $add(e)$ $\{e \in S\}$
 $\{true\}$ $add(e)$ || $add(f)$ $\{e, f \in S\}$

Example: Operation-based Repl. Set

Sequential specification of Set:

$\{true\}$	add(e)	$\{e \in S\}$
$\{true\}$	rmv(e)	$\{e \notin S\}$

Example: Operation-based Repl.

Set

$\{true\}$	add(e) rmv(e)	$\{????\}$
	add wins	$\{e \in S\}$
	remove wins	$\{e \notin S\}$
	error state	$\{\perp e \in S\}$
	last writer wins	$\{ \text{add}(e) < \text{rmv}(e) \Rightarrow e \notin S \wedge$ $\text{rmv}(e) < \text{add}(e) \Rightarrow e \in S \}$

Resort to coordination...

Alexey Gotsman
IMDEA Software Institute, SpainHongseok Yang
University of Oxford, UKCarla Ferreira
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa, PortugalMahsa Najafzadeh
Sorbonne Universités, Inria,
UPMC Univ Paris 06, FranceMarc Shapiro
Sorbonne Universités, Inria,
UPMC Univ Paris 06, France

Static Analysis for Operation- based CRDTs

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.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

Keywords Replication; causal consistency; integrity invariants

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 node 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 [2, 23].

For this reason, modern replicated databases often eschew synchronisation completely; such databases are commonly dubbed *eventually consistent* [5, 7]. 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 a 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* [5, 3], 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 compu-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
ACM, 978-1-4503-3549-2/16/01...\$15.00
<http://dx.doi.org/10.1145/2837614.2837625>

Library of CRDTs

- **Register**

- Last-Writer Wins
- Multi-Value

- **Set**

- Grow-Only
- 2P (Two Phase)
- OR (Observed Remove)

- **Map**

- **Tree**

- **Counter**

- Unlimited
- Non-negative

- **Graph**

- Directed
- Monotonic DAG
- Edit graph

- **Sequence**

CRDTs in Industry



facebook

bet365



Not Everything is a CRDT

- **Some application invariants cannot be maintained without synchronization**
 - Example: bounded resources invariants
 - Balance ≥ 0
 - Tickets ≤ 1000
 - Students enrolled ≤ 200

Bank Account

- **Invariant**

- $\text{balance} \geq 0$

- **Deposit (amt)**

- Precondition: TRUE
- Effect: $\text{balance} = \text{balance} + \text{amt}$

- **Withdraw (amt)**

- Precondition: $\text{amt} \leq \text{balance}$
- Effect: $\text{balance} = \text{balance} - \text{amt}$

Precondition Stability Analysis

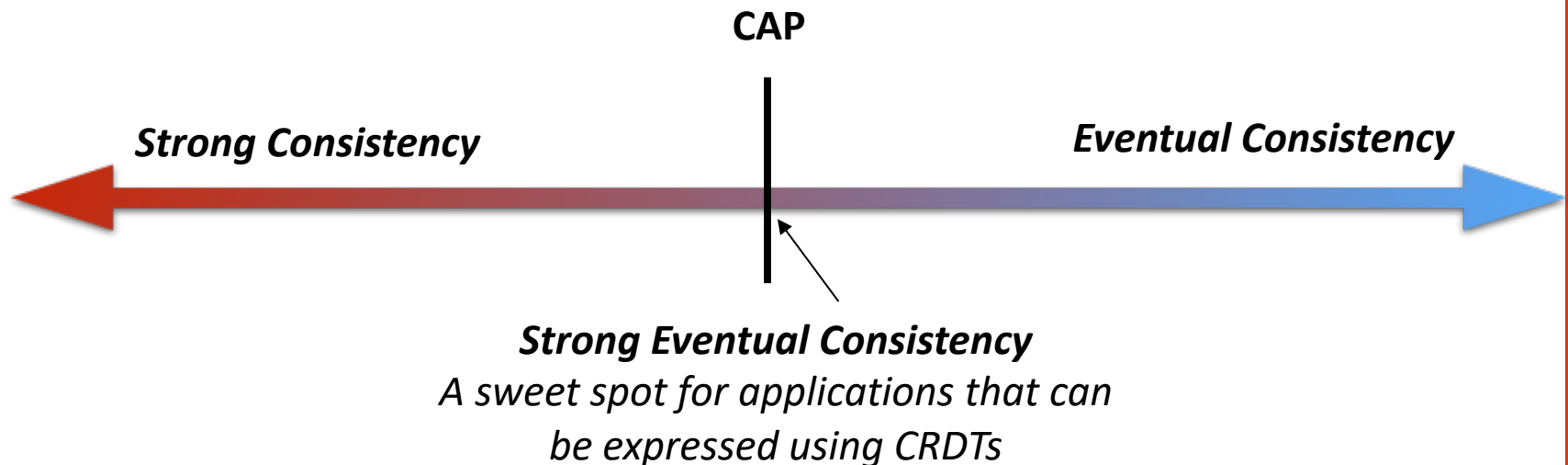
	deposit	withdraw
deposit	✓	✓
withdraw	✓	✗

Sync-free

Sync with other withdrawals

CAP Theorem

A service can either guarantee **C**onsistency or **A**vailability under network **P**artition





AntidoteDB

Transactional Causal Consistency

Transactional Causal Consistency

Cure: Strong semantics meets high availability and low latency

Deepthi Devaki Akkoorath*, Alejandro Z. Tomsic¹, Manuel Bravo², Zhongmiao Li²,
Tyler Crain¹, Annette Bieniusa*, Nuno Preguiça³, Marc Shapiro⁴

*University of Kaiserslautern, ¹Inria & LIP6-UPMC-Sorbonne Universités
²Université Catholique de Louvain, ³NOVA LINES

- **Support for atomicity**
 - Highly-available
 - No aborts
 - Strongest possible consistency while maintaining availability
 - Interactive read-write transactions
- **TCC in AntidoteDB**
 - Supports replication and sharding

Abstract—Developers of cloud-scale applications face a difficult decision of which kind of storage to use, summarised by the CAP theorem. Currently the choice is between classical CP databases, which provide strong guarantees but are slow, expensive, and unavailable under partition; and NoSQL-style AP databases, which are fast and available, but too hard to program against. We present an alternative: Cure provides the highest level of guarantees that remains compatible with availability. These guarantees include: causal consistency (no ordering anomalies), atomicity (consistent multi-key updates), and support for high-level data types (developer friendly API) with safe resolution of concurrent updates (guaranteeing convergence). These guarantees minimise the anomalies caused by parallelism and distribution, thus facilitating the development of applications. This paper presents the protocols for highly available transactions, and an experimental evaluation showing that Cure is able to achieve scalability similar to eventually-consistent NoSQL databases, while providing stronger guarantees.

I. INTRODUCTION

Internet-scale applications are typically layered above a high-performance distributed database engine running in a data centre (DC). A recent trend is to use geo-replication across several DCs to avoid wide-area network latency and to tolerate downtime. This scenario poses big challenges to the distributed database. Since network failures (called partitions) are unavoidable, according to the CAP theorem [20] the database design must sacrifice either strong consistency or availability. Traditional databases are “CP”; they provide consistency and a high-level SQL interface, but lose availability. NoSQL-style databases are “AP”, highly available, which brings significant performance benefits. However, AP-databases expose application developers to inconsistency anomalies, and most provide only low-level key-value interface.

To alleviate this problem, recent work has focused on enhancing AP designs with stronger semantics [23, 24, 28]. This paper presents Cure, our contribution in this direction. While providing availability and performance, Cure supports: (i) causal+ consistency, ensuring that if one update happens before another, they will be observed in the same order, and that replicas converge to the same state under concurrent conflicting updates, (ii) support for high-level replicated data types (CRDTs) such as counters, sets, tables

and sequences, with intuitive semantics and guaranteed convergence even in the presence of concurrent conflicting updates and partial failures, and (iii) transactions, ensuring that multiple keys (objects) are both read and written consistently, in an interactive manner.

Causal+ consistency (CC+) [6, 23] represents a sweet spot in the availability-consistency tradeoff. It is the strongest model compatible with availability [8] for individual operations. Since it ensures that the causal ordering of operations is respected, it is easier to reason about for programmers and users. Consider, for instance, a user who posts a new photo to her social network profile, then comments on the photo on her wall. Without causal consistency, a user might observe the comment but not be able to see the photo, which requires extra programming effort to avoid the anomaly at the application level.

CC+ requires that replicas converge to the same state under concurrent conflicting updates. For guaranteeing convergence, many existing causal+ consistent systems adopt the last-writer-wins rule [7, 17, 19, 23, 24], where the update that occurs “last” overwrites the previous ones. We rely on CRDTs, developer-friendly high-level data types that guarantee convergence and have rich semantics [27]. Operations on CRDTs are not only register-like assignments, but methods corresponding to a CRDT object’s type. For example, a set supports *add(item)* and *remove(item)* operations. The implementation of a CRDT set guarantees that no matter the order in which a replica receives two conflicting add and remove operations, the state of the set will converge at different replicas without the need for synchronization or application conflict handling. For instance, the Bet365 developers report that using Set CRDTs changed their life, freeing them from low-level detail and from having to compensate for concurrency anomalies [25].

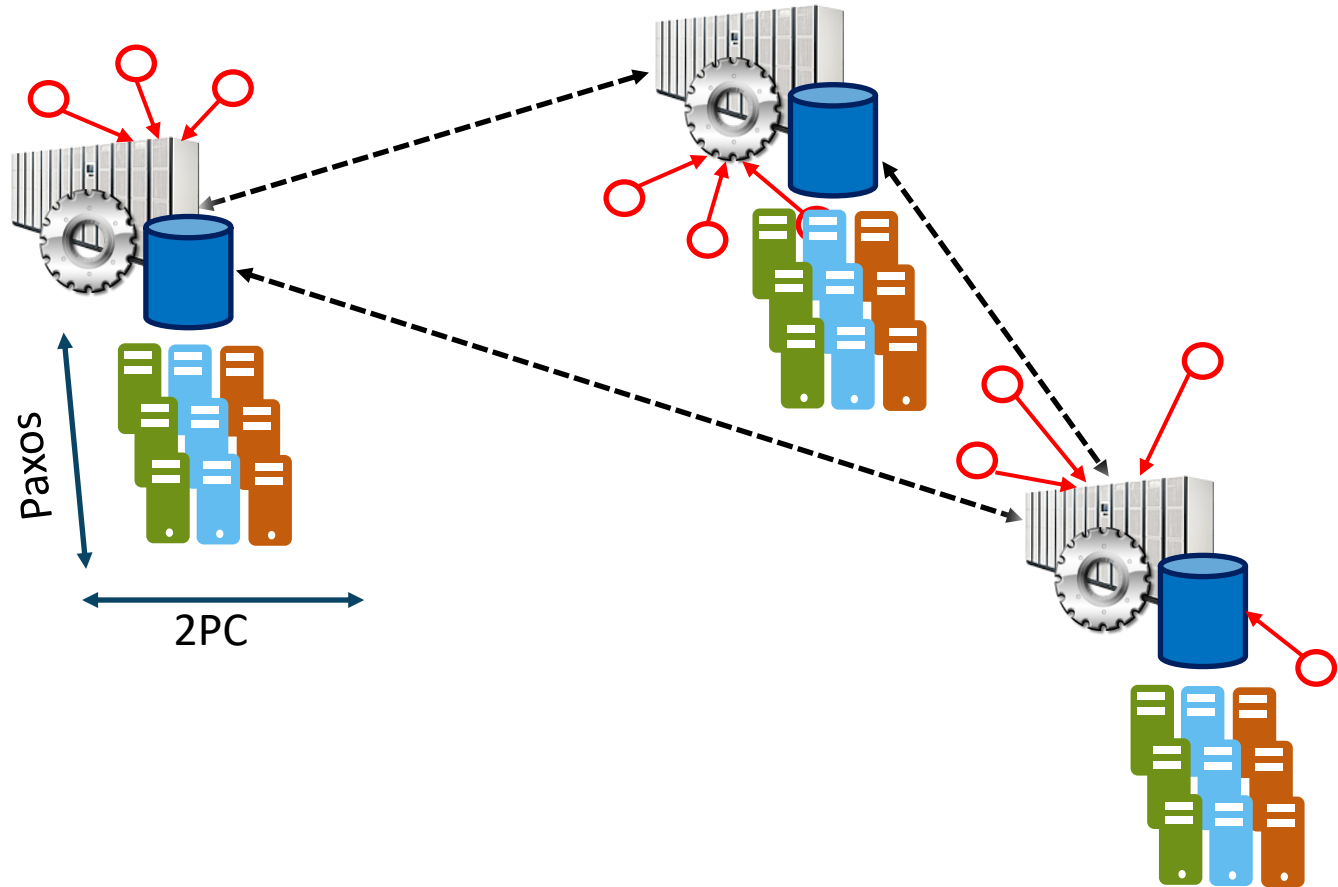
Performing multiple operations in a transaction enables the application to maintain relations between multiple objects or keys. *Highly Available Transactions* (HATs) eschew traditional strong isolation properties, which require synchronisation, in favour of availability and low latency [9, 14]. Existing CC+ HAT implementations provide either reading from a snapshot [7, 17, 19, 23, 24] or atomicity of updates [11, 24]; we introduce Transactional Causal Consistency (TCC), where all transactions provide both.



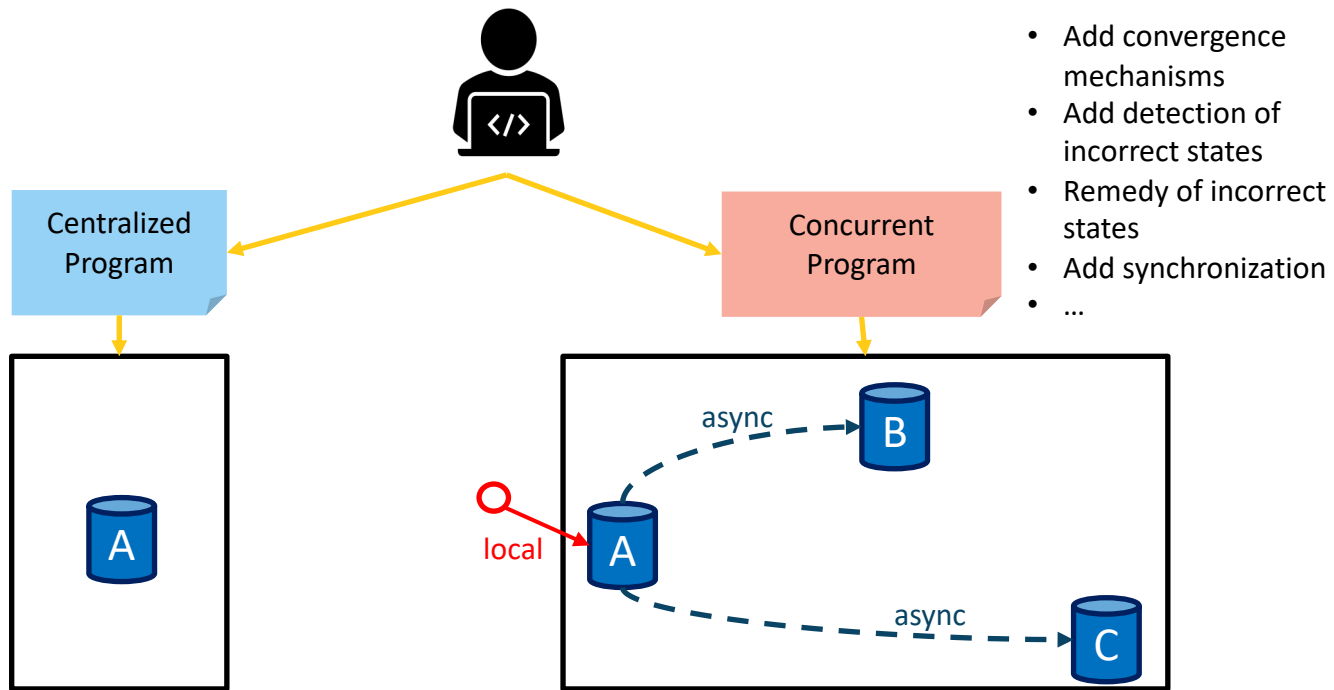
AntidoteDB

What I try to do ...

Towards Shard Replication in AntidoteDB

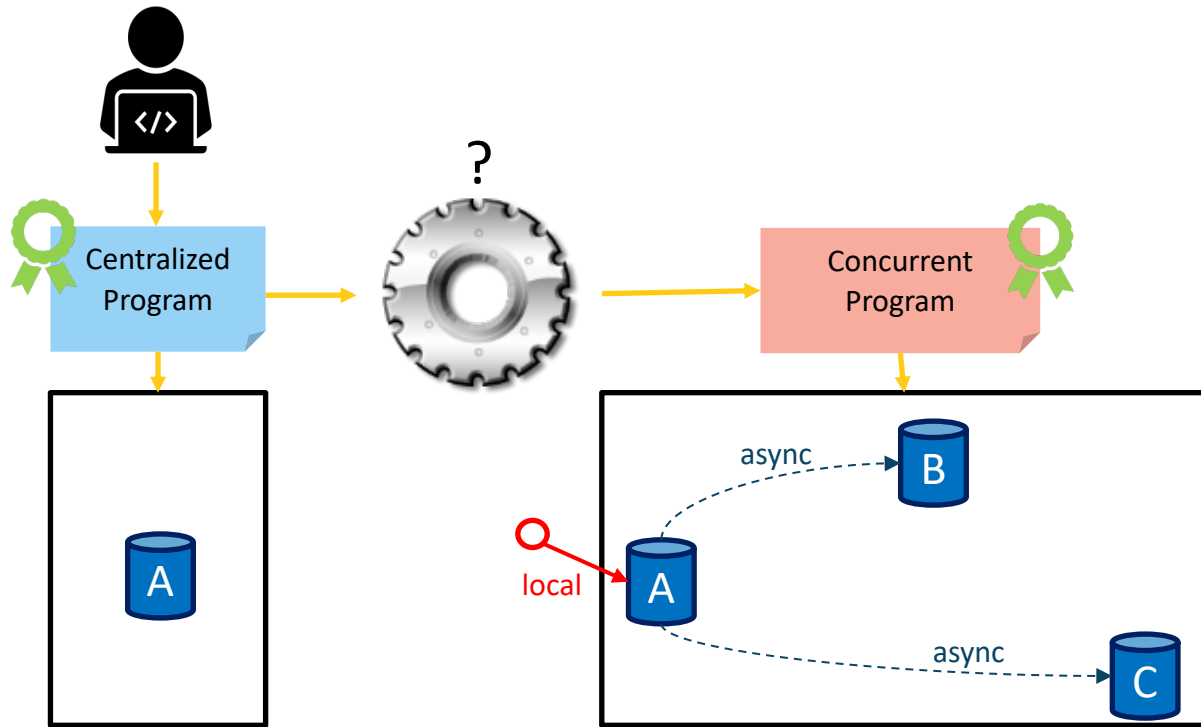


Just-Right Consistency



Just-Right Consistency

- Given a correct centralized database program, can we synthesis a correct and performant program for an AP database?



Just-Right Consistency

- Static analysis tools for state-based and operation-based CRDTs

PaPoC 2019 Invariant Safety for Distributed Applications

Sreeja S Nair
Sorbonne Université-LIP6 & Inria,
Paris, France
sreeja.nair@lip6.fr

Gustavo Petri
ARM Research, Cambridge, UK
gustavo.petri@arm.com

Marc Shapiro
Sorbonne Université-LIP6 & Inria,
Paris, France
marc.shapiro@acm.org

ABSTRACT

We study a proof methodology for verifying the safety of data invariants of highly-available distributed applications that replicate state. The proof is (1) modular: one can reason about each individual operation separately, and (2) sequential: one can reason about a distributed application as if it were sequential. We automate the methodology and illustrate the use of the tool with a representative example.

KEYWORDS

Replicated data, Consistency, Automatic verification, Distributed application design, Tool support

ACM Reference Format:
Sreeja S Nair, Gustavo Petri, and Marc Shapiro. 2019. Invariant Safety for Distributed Applications. In *6th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '19)*, March 25, 2019, Dresden, Germany. ACM, New York, NY, USA, 7 pages.
<https://doi.org/10.1145/3301419.3323970>

INTRODUCTION

Application often replicates its data to several of the closest available replica. Examining multi-user games, co-operative source control repositories, availability, otherwise,

immediately, without remote synchronisation. It propagates its updates to the other replicas asynchronously.

There are two basic approaches to update propagation: propagate operations, or to propagate states. In the former approach, an update is first applied to some origin replica, then sent as an operation to remote replicas, which in turn apply it to update their local state. Operation-based CRDTs require the message delivery layer to deliver messages in causal order, exactly once; the set of replicas must be known. In the latter approach, an update is applied to some origin replica. Occasionally, one replica sends its full state to some other replica, which merges the received state into its own. In turn, this replica will later send its own state to yet another replica. As long as every update eventually reaches every replica transitively, messages may be dropped, re-ordered or duplicated, and the set of replicas may be unknown. Replicas are guaranteed to converge if the set of states, as a result of updates and merge, forms a monotonic semi-lattice [13].

Due to these relaxed requirements, state-based CRDTs have better adoption [1]. They are the focus of this work. As a running example, consider a simple auction system. The state of an auction consists of status, a set of bids, and locations can start an auction, place bids, close the auction, declare a winner, inspect the local replica, and observe if a winner is declared and who it is. All replicas will eventually agree on the same auction status, same set of bids and the same winner.

However, the application may also require to maintain a property over the data. An invariant over the data that must evaluate to true for instance, the auction's closed, there is a winner, the highest bid is the highest, the system, but

Reasoning about Consistency Choices in Distributed Systems

Avery Gotsman
Institute, Spain

Mazdeh Afzadeh
Inria, France

'Cause I'm Strong Enough: POPL 2016

Hongseok Yang
University of Oxford, UK

Carla Ferreira
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa, Portugal

Marc Shapiro
Sorbonne Universités, Inria,
UPMC Univ Paris 06, France

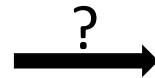
use. Ideally, we would like replicated databases to provide strong consistency, i.e., to behave as if a single centralised node handles operations. However, achieving this ideal usually requires synchronisation among replicas, which slows down the database and is unavailable if network connections between replicas

modern replicated databases often eschew synchronicity; such databases are commonly dubbed "eventually consistent". In these databases, a replica performs operations locally without any synchronisation with other replicas. The system, but

Just-Right Consistency

- I am looking at a transaction chopping criteria for TCC.

```
transfer1(acc1, acc2, amt)
{
  txn {
    acc1.balance -= amt;
    acc2.balance += amt;
  }
}
```



```
transfer2(acc1, acc2, amt) {
  chain {
    txn {
      acc1.balance -= amt;
    }
    txn {
      acc2.balance += amt;
    }
  }
}
```

Summary

- **Strong Eventual Consistency**
 - High availability
 - Strong convergence guarantees
- **CRDTs**
 - Sequential-like data structures with local deterministic conflict resolution
- **Transactional Causal Consistency**
 - Highly-available transactions with no aborts

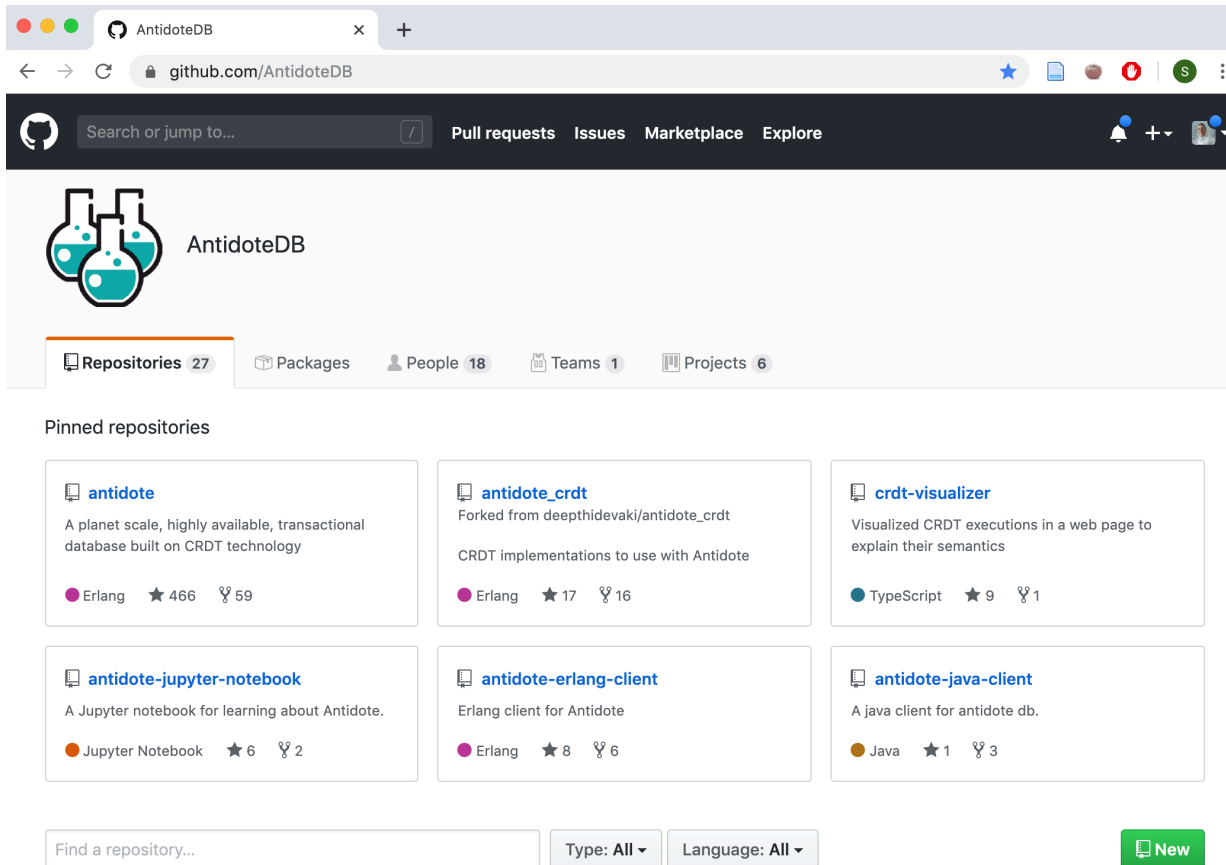
Acknowledgements

- The work presented is the result of the work of a very large number of persons, mostly in the context of SyncFree (2013-2016) and LightKone (2016-2019) projects.
- Most of the slides are from presentations prepared by Marc Shapiro (of Sorbonne-Université-LIP6 & INRIA) and Annette Bieniusa (Technical University of Kaiserslautern).
- I thank them for authorizing my use of their slides.

AntidoteDB

<https://www.antidotedb.eu/>

<https://github.com/AntidoteDB>



The screenshot shows the GitHub profile page for AntidoteDB. At the top, there's a search bar and navigation links for Pull requests, Issues, Marketplace, and Explore. The profile header includes the AntidoteDB logo (three beakers) and the name 'AntidoteDB'. Below this, there are statistics for Repositories (27), Packages, People (18), Teams (1), and Projects (6). The 'Pinned repositories' section displays six repositories:

- antidote**: A planet scale, highly available, transactional database built on CRDT technology. Language: Erlang. 466 stars, 59 forks.
- antidote_crdt**: Forked from deepthidevaki/antidote_crdt. CRDT implementations to use with Antidote. Language: Erlang. 17 stars, 16 forks.
- crdt-visualizer**: Visualized CRDT executions in a web page to explain their semantics. Language: TypeScript. 9 stars, 1 fork.
- antidote-jupyter-notebook**: A Jupyter notebook for learning about Antidote. Language: Jupyter Notebook. 6 stars, 2 forks.
- antidote-erlang-client**: Erlang client for Antidote. Language: Erlang. 8 stars, 6 forks.
- antidote-java-client**: A java client for antidote db. Language: Java. 1 star, 3 forks.

At the bottom, there is a search bar 'Find a repository...', filters for 'Type: All' and 'Language: All', and a green 'New' button.

Thank you!