Conflict-free Replicated Data Types in Practice

Georges Younes Vitor Enes

Wednesday 11th January, 2017

HASLab/INESC TEC & University of Minho
InfoBlender

Motivation

Background

Background: CAP Theorem

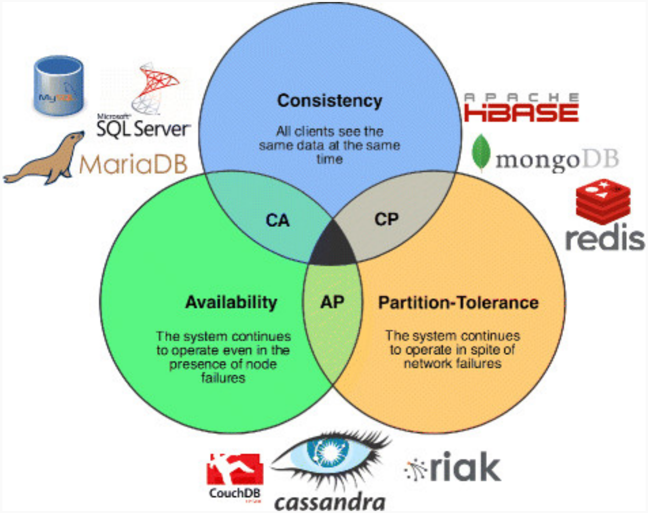
Brewer's Conjecture

2000 Eric Brewer, PoDC Conference Keynote

2002 Seth Gilbert and Nancy Lynch, ACM SIGACT News 33(2)

Of three properties of shared-data system - data **C***onsistency, system* **A***vailability and tolerance to network* **P***artitions - <u>only two</u> can be achieved <u>at any given moment</u> in time.*

Background: CAP Theorem


Consistency
All clients see the same data at the same time
Availability
The system continues to operate even in the presence of node failures
Partition-Tolerance
The system continues to operate in spite of network failures
CA
CP
AP
MySQL
Microsoft SQL Server
MariaDB
APACHE HBASE
mongoDB
redis
CouchDB
cassandra
riak

source: Lourenco et al. Journal of Big Data (2015) 2:18 DOI 10.1186/s40537-015-0025-0

“ The best thing about being me... There are so many ’me’s ”

-Agent Smith

Background: Data Replication

Data Replication: Maintaining multiple data copies on separate machines.

- Improves Availability
  - Allows access when some replicas are not available
- Improves Performance
  - reduced latency: let users access nearby replicas
  - increased throughput: let multiple machines serve the data

Pessimistic

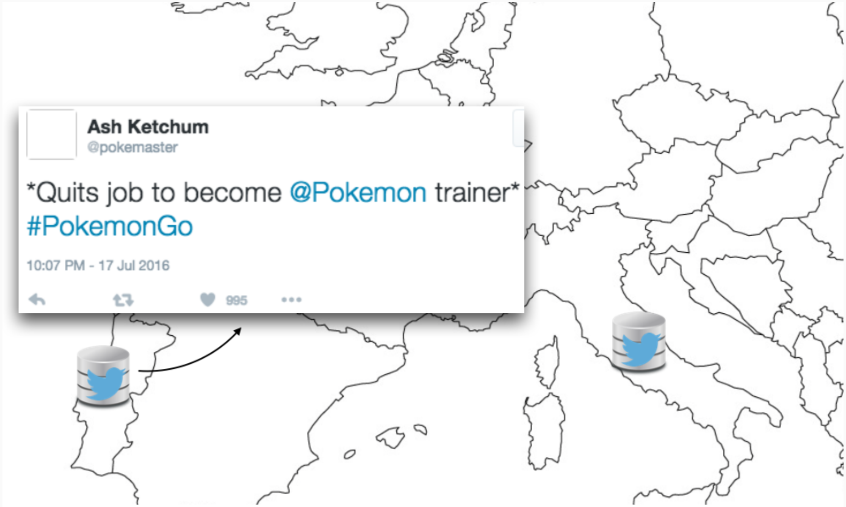
- Provide single-copy consistency
- Block access to a replica unless it is provably up to date
- Perform well in LANs (small latencies, uncommon failures)
- Not in Wide Area Networks

Optimistic

- Provide eventual consistency
- Let access to a replica without a priori synchronization
- Updates propagated in bg, occasional conflicts resolved later
- Offer many advantages over their pessimistic counterparts

Background: Eventual Consistency (EC)

Background: Eventual Consistency (EC)



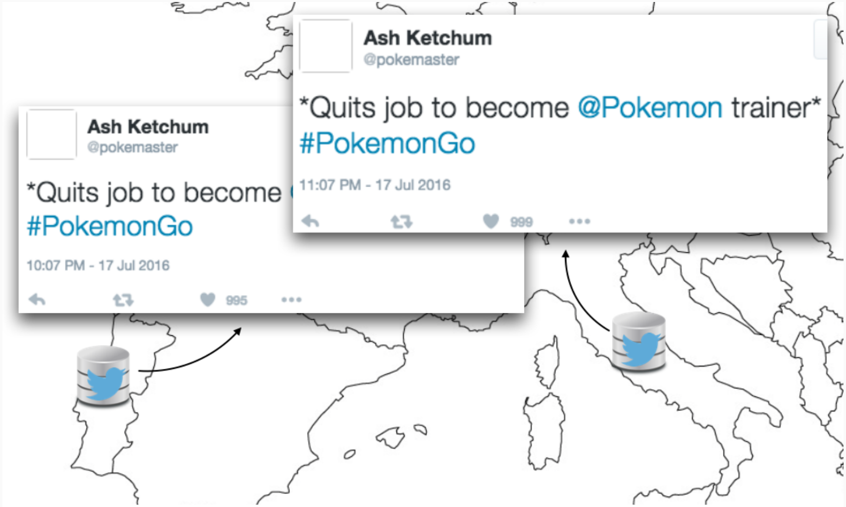
Ash Ketchum
@pokemaster

Quits job to become @Pokemon trainer #PokemonGo

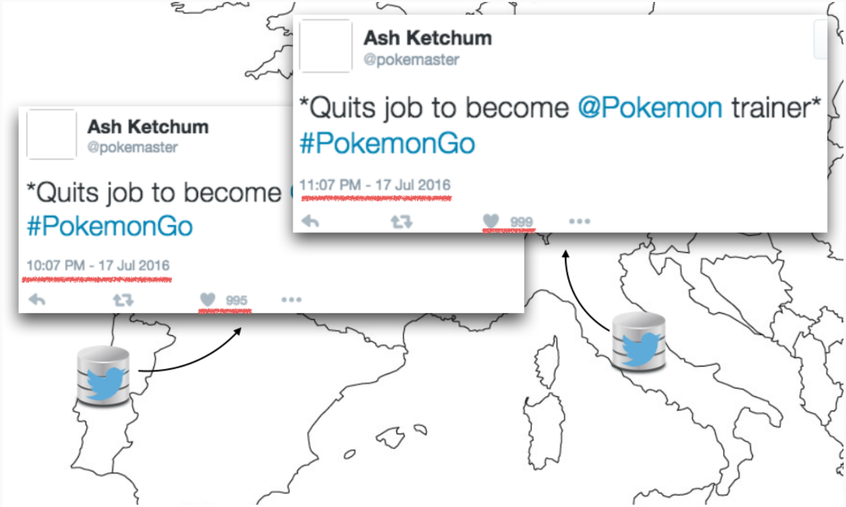
10:07 PM - 17 Jul 2016

995

Background: Eventual Consistency (EC)



Background: Eventual Consistency (EC)



At the same moment, two users can see the same tweet with different number of favorites

Conflict-free Replicated Data Types

Conflict-free Replicated Data Types

CRDTs

- Conflict-free: resolves conflicts automatically converging to the same state
- Replicated: multiple independent copies
- Data Types: Registers, Counters, Maps, Sequences, Sets, Graphs..etc

CRDT Flavors

There are two flavors of CRDTs:

- Operation-based: broadcasts update operations to other replicas
- State-based: propagates the local state to other replicas

Op-based CRDTs

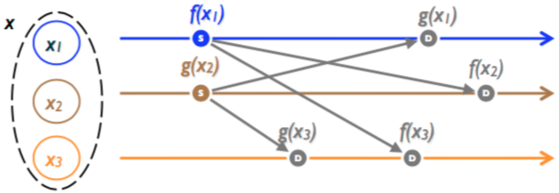


Figure 1: Operation based replication

For each update operation do:

- Prepare: Calculate the downstream (the effect observed locally)
- Apply the effect locally
- Disseminate (reliable causal broadcast) the calculated effect to be applied on all other replicas

Op-based (Counter :: $\mathbb{N}$)

replica A

$S_A = 0$

$S_A = \text{update}(\textbf{inc}, S_A) = 0 + 1 = 1$

$m_1 = \textbf{inc}$

$\textbf{send}_B(m_1)$

---

on receive$_R(m_2)$:

$\quad S_A = \text{update}(dec, S_A) = 1 - 1 = 0$

Op-based (Counter :: $\mathbb{N}$)

replica B

$S_B = 0$

$S_B = \text{update}(\textbf{dec}, S_B) = S_B - 1 = -1$

$m_2 = \textbf{inc}$

$\textbf{send}_A(m_2)$

---

on receive$_R(m_1)$:

$\quad S_B = \text{update}(inc, S_B) = -1 + 1 = 0$

What if m_1 is lost?

What if m_2 is duplicated?

How do solve these problems?

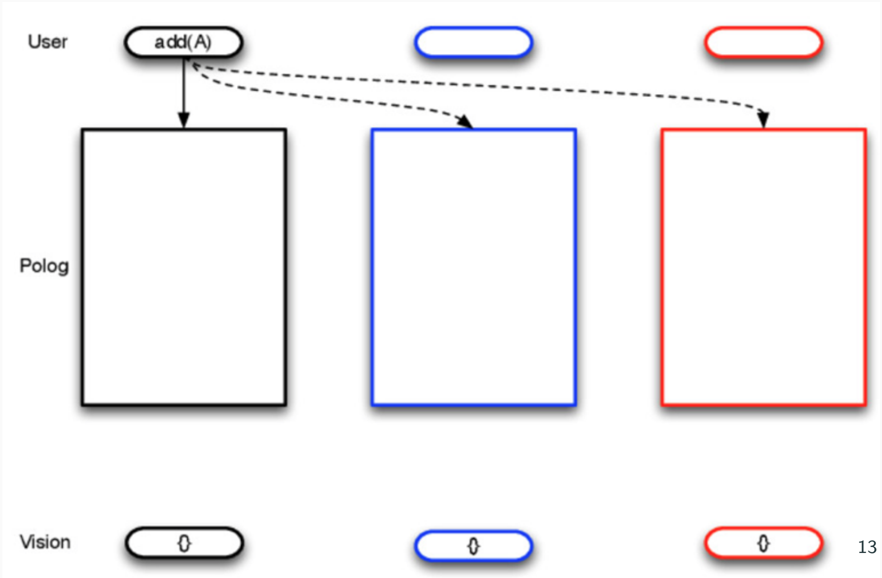
TRCB: Tagged Reliable Causal Broadcast

- Tags each operation with a unique timestamp (VC)
- Delivery of operations respecting causal order
- Reliable bcast: at-least-once to prevent message loss
- Reliable bcast: at-most-once to prevent message duplication

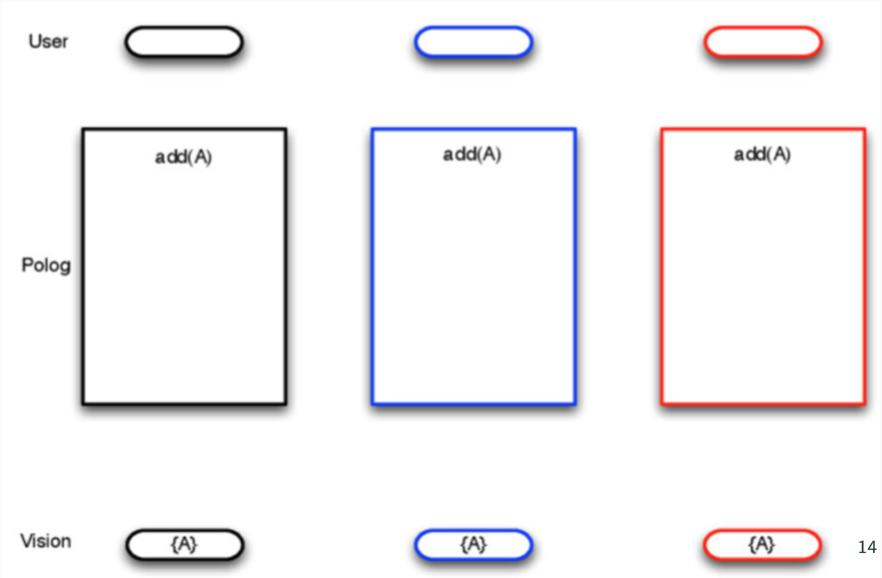
Are all data types commutative?

- Counters are commutative: $incr; decr; == decr; incr;$
- Not all data types are (In fact most are NOT)
- A Set S_1 with $add(element, Set)$ and $rmv(element, Set)$ operation is not
- $add(a, S_1)$; $rmv(a, S_1)$; $value(S_1) = \{\}$
- $rmv(a, S_1)$; $add(a, S_1)$; $value(S_1) = \{a\}$
- How do we solve concurrent operations?

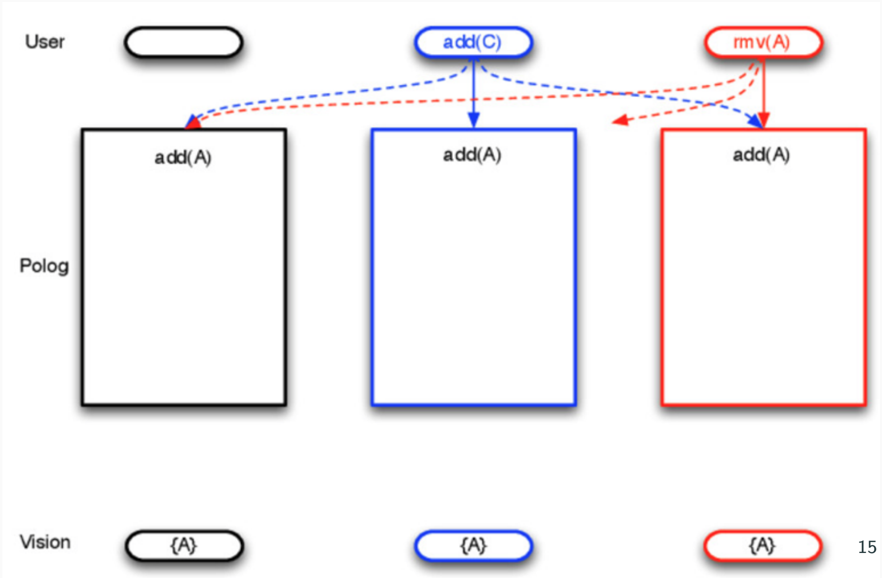
POLog: Partially Ordered Log



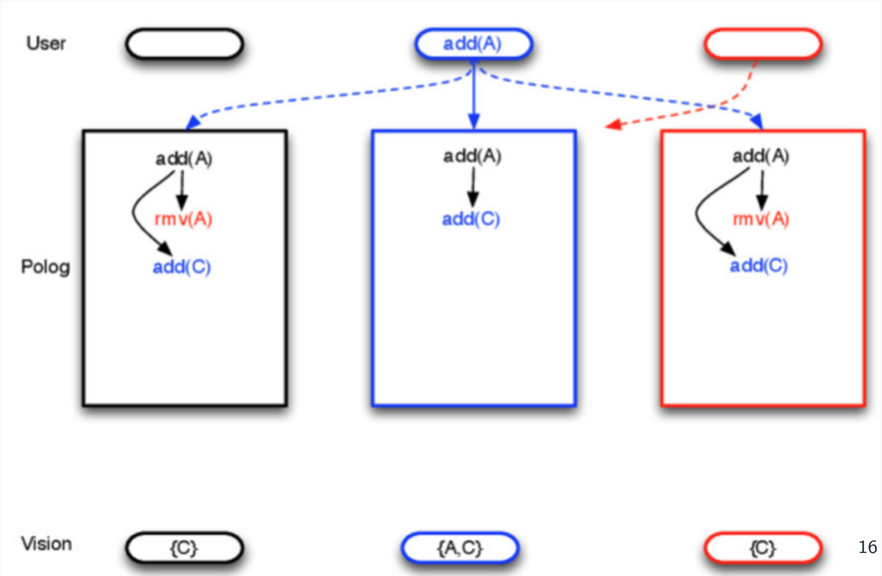
POLog: Partially Ordered Log



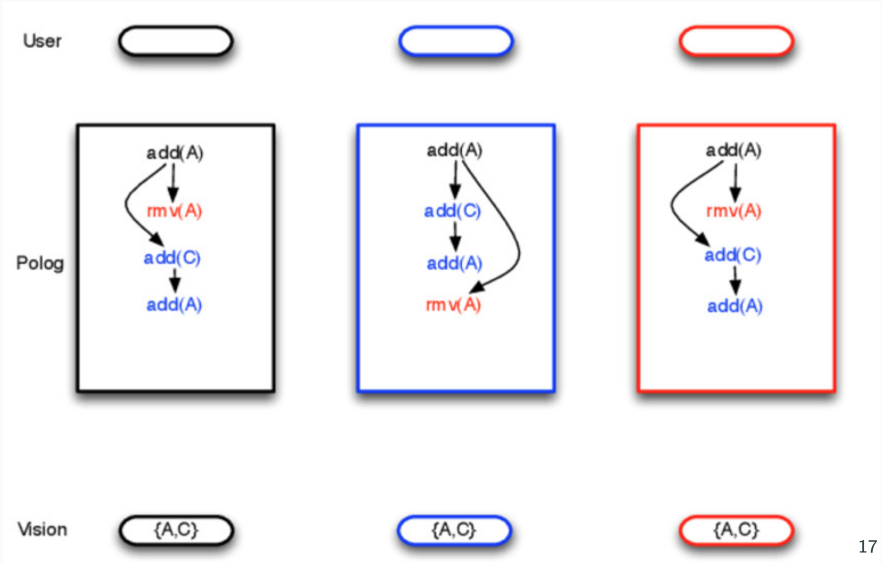
POLog: Partially Ordered Log



POLog: Partially Ordered Log



POLog: Partially Ordered Log



POLog: Partially Ordered Log

Querying the Add-Wins-Set (AWSet)

- An element v is in the set if there is an add_v operation in the set that is not succeeded by rmv_v operation
- $v | (t, add_v) \in POLog \land \nexists (t', rmv_v) \in POLog . t \rightarrow t'$

So does the POLog keep growing?

- In the op-based CRDT model, the POLog keeps growing
- Pure op-based CRDT model was introduced to:
  - Apply GC on the POLog
  - Reduces the message size
  - Provides a more generic API

Pure op-based in Redis

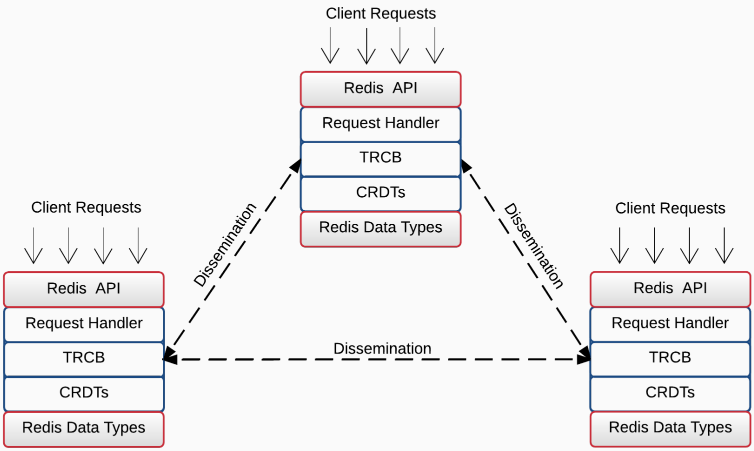


Figure 2: Pure op-based Architecture in Redis

Operation-based (GCounter :: $\mathbb{N}$)

$$\begin{tikzcd}
\mathbf{A} & 1 \arrow[r, "\textbf{inc}"] & 2 \arrow[rr, "\textbf{inc}"] \arrow[dr, "m_1{=}\textbf{inc}"] & & 3 \arrow[dr, "m_2{=}\textbf{inc}"] & \\
\mathbf{B} & 1 \arrow[rr, dotted, no head] & & 2 \arrow[rr, dotted, no head] & & 3
\end{tikzcd}$$

State-based (GCounter :: $\mathbb{I} \hookrightarrow \mathbb{N}$)

$$\begin{tikzcd}
\mathbf{A} & \{B_1\} \arrow[r, "\textbf{inc}"] & \{A_1, B_1\} \arrow[rr, "\textbf{inc}"] \arrow[dr, "m_1{=}\{A_1{,}B_1\}"] & & \{A_2, B_1\} \arrow[dr, "m_2{=}\{A_2{,}B_1\}"] & \\
\mathbf{B} & \{B_1\} \arrow[rr, dotted, no head] & & \{A_1, B_1\} \arrow[rr, dotted, no head] & & \{A_2, B_1\}
\end{tikzcd}$$

What if m_1 is lost? (monotonicity)

What if m_2 is duplicated? (idempotence)

What if m_1 and m_2 are reordered? (commutativity)

Why are state-based CRDTs so cool?

State-based CRDT

A state-based CRDT is a join-semilattice $(S, \sqsubseteq, \sqcup)$ where S is a poset, $\sqsubseteq$ its partial order, and $\sqcup$ a binary join operator that derives the least upper bound for every two elements of S.

$\forall s, t, u \in S$:

- $s \sqcup s = s$ (idempotence)
- $s \sqcup t = t \sqcup s$ (commutativity)
- $s \sqcup (t \sqcup u) = (s \sqcup t) \sqcup u$ (associativity)

$ $ full state transmission $ $

Avoiding full state transmission

How we can we avoid replica A sending the full state to replica B?

Two fundamental problems

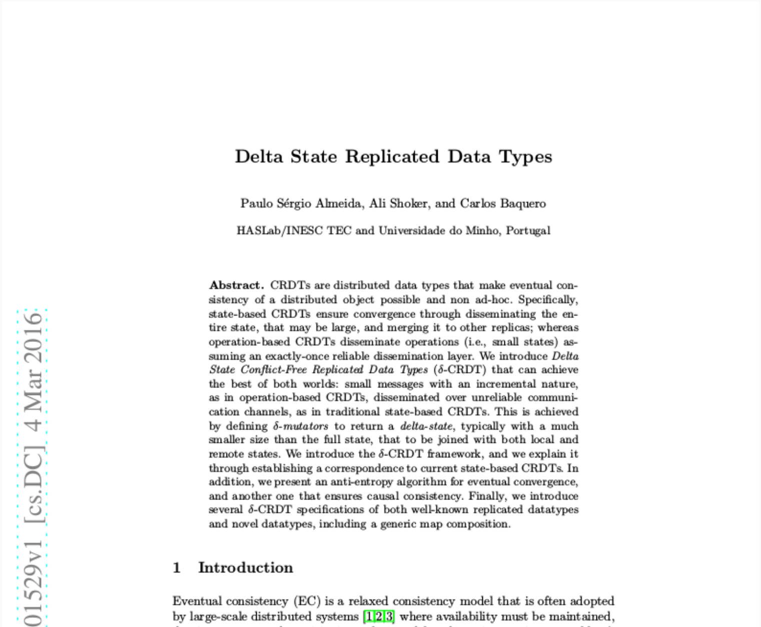
A knows something about B:

- **B** has S_{old}
- **A** has S_{new} and knows that **B** has S_{old}
- **Goal:** compute delta d of minimum size s.t. $S_{new} = S_{old} \sqcup d$

A knows nothing about B:

- **B** has S_{old}
- **A** has S_{new}
- **Goal:** protocol that minimizes communication cost

Solving the first problem



Delta State Replicated Data Types

Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero

HASLab/INESC TEC and Universidade do Minho, Portugal

Abstract. CRDTs are distributed data types that make eventual consistency of a distributed object possible and non ad-hoc. Specifically, state-based CRDTs ensure convergence through disseminating the entire state, that may be large, and merging it to other replicas; whereas operation-based CRDTs disseminate operations (i.e., small states) assuming an exactly-once reliable dissemination layer. We introduce *Delta State Conflict-Free Replicated Data Types* (δ-CRDT) that can achieve the best of both worlds: small messages with an incremental nature, as in operation-based CRDTs, disseminated over unreliable communication channels, as in traditional state-based CRDTs. This is achieved by defining *δ-mutators* to return a *delta-state*, typically with a much smaller size than the full state, that to be joined with both local and remote states. We introduce the δ-CRDT framework, and we explain it through establishing a correspondence to current state-based CRDTs. In addition, we present an anti-entropy algorithm for eventual convergence, and another one that ensures causal consistency. Finally, we introduce several δ-CRDT specifications of both well-known replicated datatypes and novel datatypes, including a generic map composition.

1 Introduction

Eventual consistency (EC) is a relaxed consistency model that is often adopted by large-scale distributed systems [1,2,3] where availability must be maintained,

Solving the first problem: Delta-CRDTs

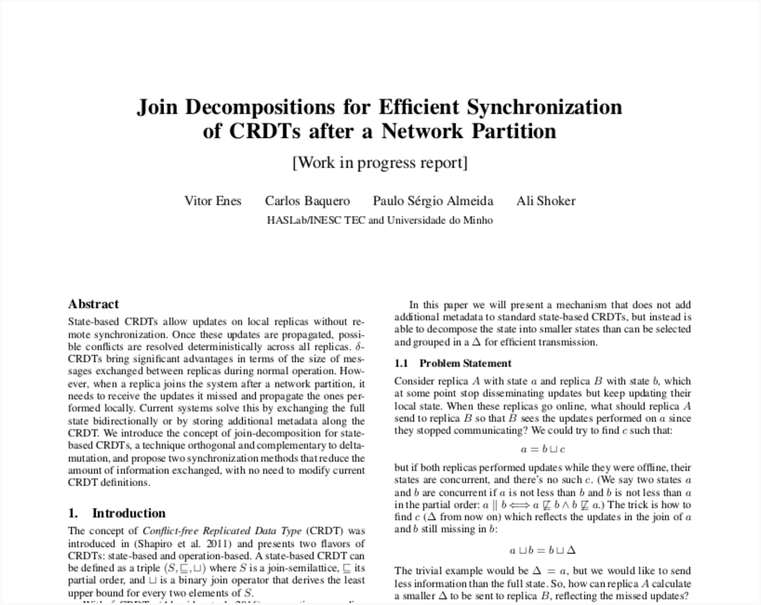
State-based (GSet :: $\mathcal{P}(E)$)

$$\begin{tikzcd}
\textbf{A} & \{\} \arrow[r, "\textbf{add } x"] & \{x\} \arrow[rr, "\textbf{add } y"] \arrow[dr, "\{x\}"] & & \{x, y\} \arrow[rrr, dotted, no head] \arrow[dr, "\{x{,}y\}"] & & & \{x, y, z\} \\
\textbf{B} & \{\} \arrow[rr, dotted, no head] & & \{x\} \arrow[rr, dotted, no head] & & \{x, y\} \arrow[r, "\textbf{add } z"] & \{x, y, z\} \arrow[ur, "\{x{,}y{,}z\}"'] &
\end{tikzcd}$$

Delta-state-based

$$\begin{tikzcd}
\textbf{A} & \{\} \arrow[r, "\textbf{add } x"] & \{x\} \arrow[rr, "\textbf{add } y"] \arrow[dr, "\{x\}"] & & \{x, y\} \arrow[rrr, dotted, no head] \arrow[dr, "\{y\}"] & & & \{x, y, z\} \\
\textbf{B} & \{\} \arrow[rr, dotted, no head] & & \{x\} \arrow[rr, dotted, no head] & & \{x, y\} \arrow[r, "\textbf{add } z"] & \{x, y, z\} \arrow[ur, "\{z\}"'] &
\end{tikzcd}$$

Solving the second problem



Join Decompositions for Efficient Synchronization of CRDTs after a Network Partition

[Work in progress report]

Vitor Enes Carlos Baquero Paulo Sérgio Almeida Ali Shoker
HASLab/INESC TEC and Universidade do Minho

Abstract

State-based CRDTs allow updates on local replicas without remote synchronization. Once these updates are propagated, possible conflicts are resolved deterministically across all replicas. δ-CRDTs bring significant advantages in terms of the size of messages exchanged between replicas during normal operation. However, when a replica joins the system after a network partition, it needs to receive the updates it missed and propagate the ones performed locally. Current systems solve this by exchanging the full state bidirectionally or by storing additional metadata along the CRDT. We introduce the concept of join-decomposition for state-based CRDTs, a technique orthogonal and complementary to delta-mutation, and propose two synchronization methods that reduce the amount of information exchanged, with no need to modify current CRDT definitions.

1. Introduction

The concept of *Conflict-free Replicated Data Type* (CRDT) was introduced in (Shapiro et al. 2011) and presents two flavors of CRDTs: state-based and operation-based. A state-based CRDT can be defined as a triple $(S, \sqsubseteq, \sqcup)$ where S is a join-semilattice, $\sqsubseteq$ its partial order, and $\sqcup$ is a binary join operator that derives the least upper bound for every two elements of S.

In this paper we will present a mechanism that does not add additional metadata to standard state-based CRDTs, but instead is able to decompose the state into smaller states than can be selected and grouped in a Δ for efficient transmission.

1.1 Problem Statement

Consider replica A with state a and replica B with state b, which at some point stop disseminating updates but keep updating their local state. When these replicas go online, what should replica A send to replica B so that B sees the updates performed on a since they stopped communicating? We could try to find c such that:

$$a = b \sqcup c$$

but if both replicas performed updates while they were offline, their states are concurrent, and there's no such c. (We say two states a and b are concurrent if a is not less than b and b is not less than a in the partial order: $a \parallel b \iff a \not\sqsubseteq b \wedge b \not\sqsubseteq a$.) The trick is how to find c (Δ from now on) which reflects the updates in the join of a and b still missing in b:

$$a \sqcup b = b \sqcup \Delta$$

The trivial example would be $\Delta = a$, but we would like to send less information than the full state. So, how can replica A calculate a smaller Δ to be sent to replica B, reflecting the missed updates?

Solving the second problem: Join Decompositions

State-driven

$$\begin{tikzcd}
\mathbf{A} & \{a\} \arrow[r, "\textbf{add } x{,}y"] & \{a, x, y\} \arrow[r, dotted, dash] & \{a, x, y, z\} \arrow[dr, "\{x{,}y\}"] & \\
\mathbf{B} & \{a\} \arrow[r, "\textbf{add } z"] & \{a, z\} \arrow[ur, "\{a{,}z\}"'] \arrow[rr, dotted, dash] & & \{a, x, y, z\}
\end{tikzcd}$$

Digest-driven

$$\begin{tikzcd}
\mathbf{A} & \{a\} \arrow[r, "\textbf{add } x{,}y"] & \{a, x, y\} \arrow[r, dotted, dash] & \{a, x, y\} \arrow[dr, "(\{x{,}y\}{,}d_B)"] \arrow[rr, dotted, dash] & & \{a, x, y, z\} \\
\mathbf{B} & \{a\} \arrow[r, "\textbf{add } z"] & \{a, z\} \arrow[ur, "d_B"'] \arrow[rr, dotted, dash] & & \{a, x, y, z\} \arrow[ur, "\{z\}"'] &
\end{tikzcd}$$

Who uses that?

Lasp AntidoteDB TOMTOM bet365

akka basho riak TRIFORK. ...think software

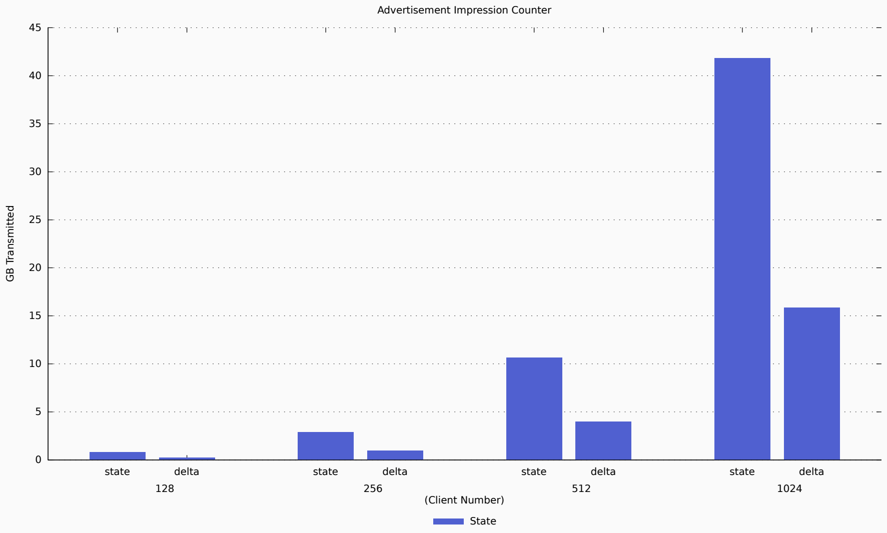
SOUNDCLOUD PROJECTFIFO HELIUM Tapjoy

ROVIO ANGRY BIRDS RIOT GAMES LEAGUE of LEGENDS

Lasp

- Lasp is a language for distributed, eventually consistent computations $\Rightarrow$ CRDTs
- from **basho/riak_dt** to **lasp/types**
- **lasp/types**:
    - State-based CRDTs
    - Delta-based CRDTs
    - Pure-op-based CRDTs

GCounter state transmission



LDB & LSim

LDB

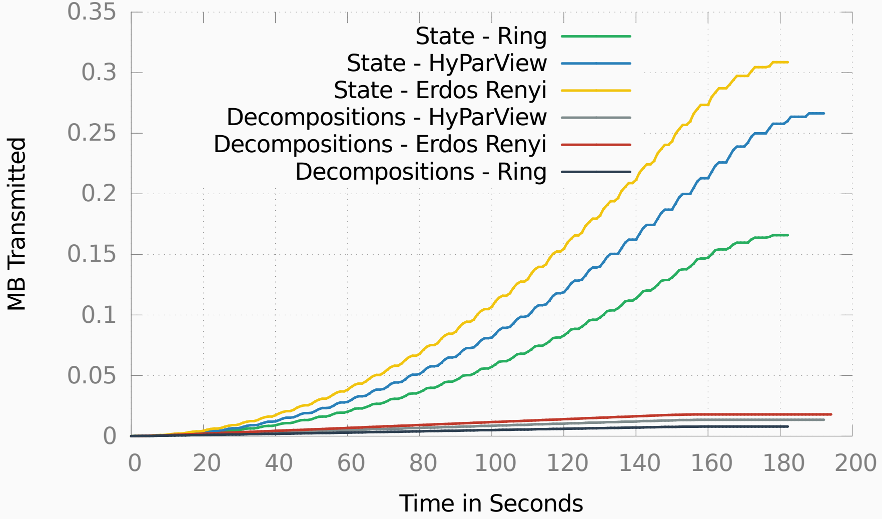
- benchmarking platform for CRDTs
- **lasp/types** + **replication**
    - State-based CRDTs
    - Delta-based CRDTs
    - Delta-based CRDTs + Join Decompositions
    - Pure-op-based CRDTs

LSim

- LDB + Peer Services + Workloads
- experiments in DC/OS (Apache Mesos + Marathon)

docker MARATHON MESOS

GSet state transmission



Challenges

1. Log aggregation
2. Time
   - Time-based graphs (e.g. x axis is time)
   - Total-effort graphs demands equal total-time across all runs
3. **Bugs**
   - LDFI
   - IronFleet
   - Jepsen
   - ...
4. $$ (On-Demand vs Spot Instances)

Questions?

bit.ly/crdts-infoblender