CRDTs from the ground up

Derek Kraan - Code Beam Lite Italy 2019

Derek Kraan

Author of ***Horde*** & ***DeltaCRDT*** libraries

Independent Software Consultant based in **Amsterdam**

Problem: distributed state

Traditional approach: use a database or other external service (Redis, etc)

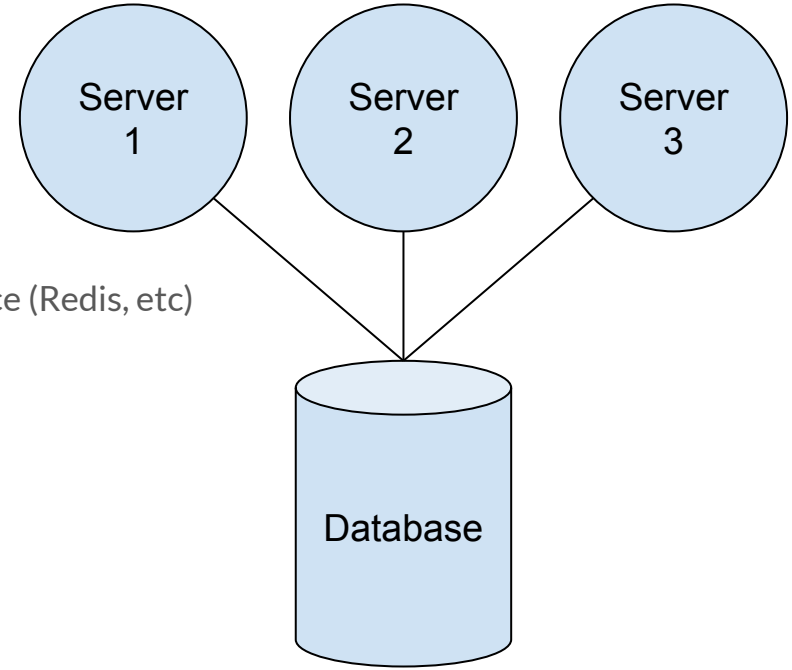
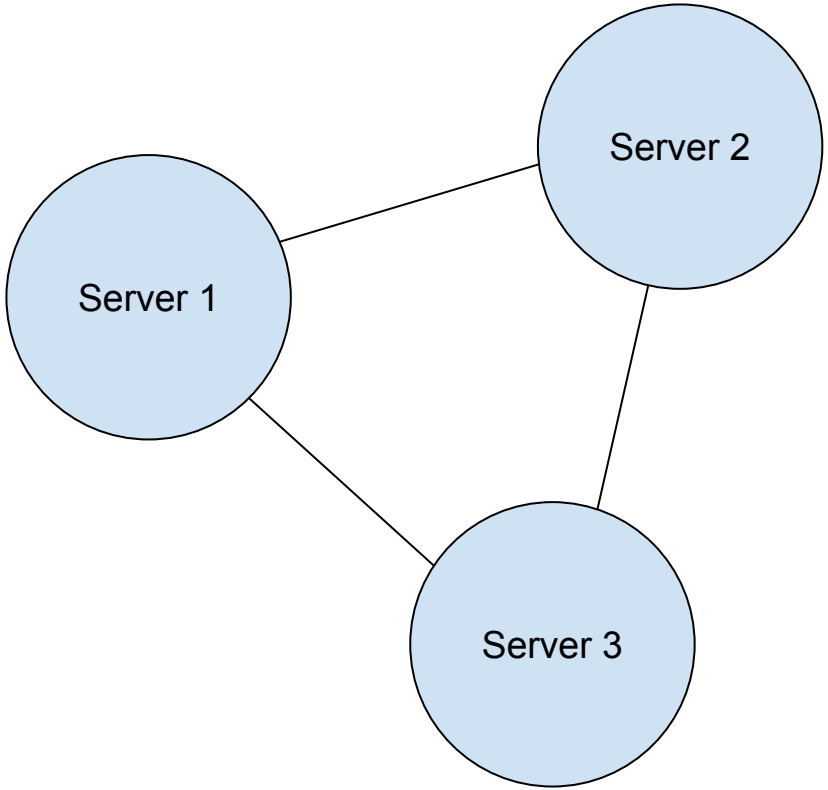


Table 1.1 Comparison of technologies used in two real-life web servers

Technical requirement	Server A	Server B
HTTP server	Nginx and Phusion Passenger	Erlang
Request processing	Ruby on Rails	Erlang
Long-running requests	Go	Erlang
Server-wide state	Redis	Erlang
Persistable data	Redis and MongoDB	Erlang
Background jobs	Cron, Bash scripts, and Ruby	Erlang
Service crash recovery	Upstart	Erlang

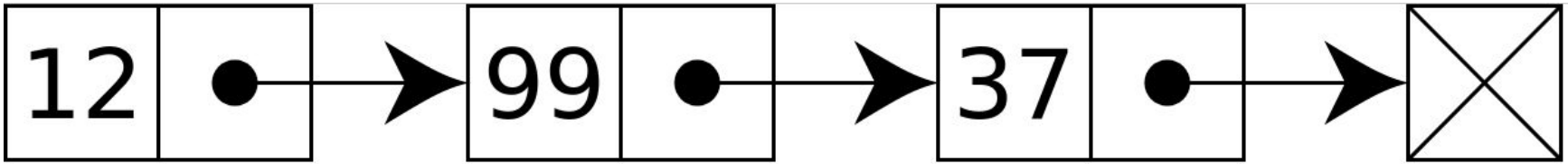
A different approach?

- Each node has a copy of the state
- Each node updates its local copy
- Each node continually merges its local state with the state of its neighbours.

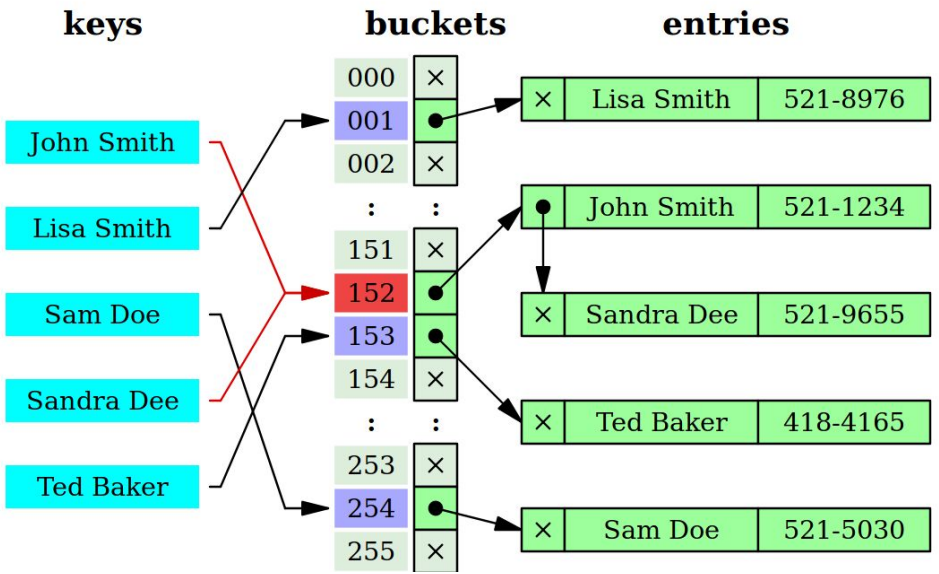


Data structures

Linked list



Map



INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Conflict-free Replicated Data Types

Marc Shapiro, INRIA & LIP6, Paris, France
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal
Carlos Baquero, Universidade do Minho, Portugal

Marek Zawirski, INRIA & UPMC, Paris, France

N° 7687

Juillet 2011

Thème COM

ISRN INRIA/RR--7687--FR+ENG

Conflict-free Replicated Data Types

A day in the life of a CRDT node

1. Set own initial state.
2. Merge state from random neighbour with own state.
3. Change own state (maybe).
4. Repeat from step 2.

The simplest CRDT we can imagine

- State: a single bit.
- Initial state is 0.
- Merge function is “OR”.

OR		
	0	1
0	0	1
1	1	1

Demonstration

 = 0

 = 1

Conflict-free

Replicated

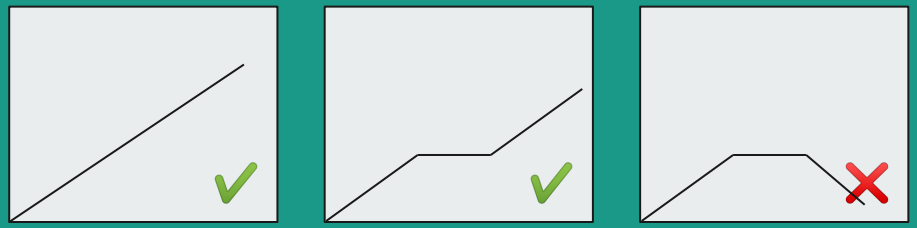
Eventually consistent

Properties of State / Merge function

Must be:

- Commutative $a \bullet b = b \bullet a$
- Associative $(a \bullet b) \bullet c = a \bullet (b \bullet c)$
- Idempotent $a \bullet b = a \bullet b \bullet b$
- Monotonic 0 -> 1 ✔️ 1 -> 0 ❌

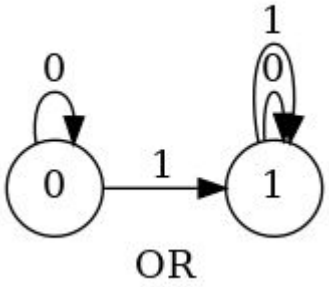
Monotonicity



Example: monotonic clock

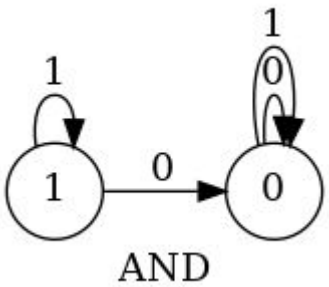
- NTP for correcting local server time -- may cause time to appear to go backwards!
- Monotonic clock will never allow time to go backwards … waits for actual time to catch up.

Example: Logical Operators



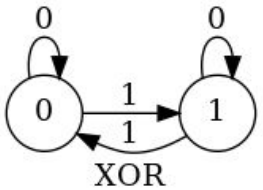
OR

✔



AND

✔



XOR

✘

Demonstration

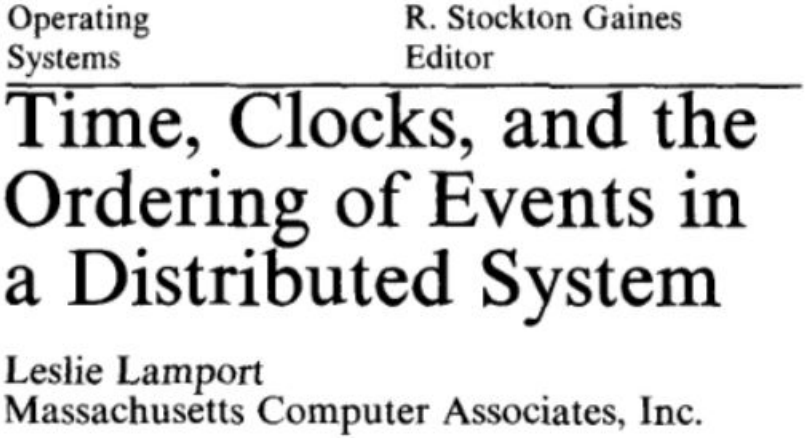
= 0

= 1

XOR		
	0	1
0	0	1
1	1	0

Tour of known CRDTs

Grow-only Counter



For n nodes: State = $\{x_1, x_2, \ldots x_n\}$ Join function = MAX

In other words, each node updates only its own counter.

❗ More reliable method for determining distributed order than relying on traditional timestamps.

AKA *G-counter, Lamport timestamp, Lamport clock, Vector clock*

Associative ✔️ Commutative ✔️ Idempotent ✔️ Monotonic ✔️

Add-only Set

State = Set<> Join function = UNION

Can only add elements, not remove them (monotonic)

AKA *Grow-only set*, *G-set*

Associative ✔️ Commutative ✔️ Idempotent ✔️ Monotonic ✔️

Non-monotonic CRDTs?

Positive-Negative Counter

For n nodes: State = $\{x_1, x_2, \ldots x_n\}, \{y_1, y_2, \ldots y_n\}$ Join function = MAX

Increment by incrementing x_n, decrement by incrementing y_n.

❗ From the outside, this appears to not be monotonic

AKA *P-N Counter*

Associative ✔️ Commutative ✔️ Idempotent ✔️ Monotonic ✔️

Add / Remove Set

Same trick as with positive-negative counter.

Seems to not be monotonic from the outside!

❗ We call the remove set “tombstones”.

Aside: state-based vs operation-based CRDTs

State-based CRDTs need at-least-once delivery (easy).

Operation-based CRDTs need exactly-once delivery (hard).

BUT! For every operation-based CRDT there is an equivalent state-based CRDT[1]

[1]*Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. Conflict-free Replicated Data Types.[Research Report] RR-7687, 2011, pp.18. <inria-00609399v1>*

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, despite outages and partitioning, whereas delayed consistency is acceptable. A typical approach in EC systems is to allow replicas of a distributed object to temporarily diverge, provided that they can eventually be reconciled into a common state. To avoid application-specific reconciliation methods, costly and error-prone, *Conflict-Free Replicated Data Types* (CRDTs) [4,5] were introduced, allowing the design of self-contained distributed data types that are always available and eventually converge when all operations are reflected at all replicas. Though CRDTs are deployed in practice and support millions of users worldwide [6,7,8], more work is still required to improve their design and performance.

CRDTs support two complementary designs: *operation-based* (or op-based) and *state-based*. In op-based designs [9,5], the execution of an operation is done in two phases: *prepare* and *effect*. The former is performed only on the local replica and looks at the operation and current state to produce a message that aims to represent the operation, which is then shipped to all replicas. Once received, the representation of the operation is applied remotely using *effect*.

Delta-CRDTs

A day in the life of a delta-CRDT

1. Set initial state, initial (empty) list of deltas.
2. On receive deltas from neighbour, merge with state and add to list of deltas; acknowledge receipt.
3. Mutate own state: generate delta, merge with state, add to list of deltas.
4. [Periodically] send list of deltas to random neighbour.
5. [Periodically] garbage collect deltas that have been acknowledged by all neighbours.
6. Repeat from step 2.

Things to keep track of?

- State [PERSIST]
- NEW! Internal counter [PERSIST]
- NEW! Deltas
- NEW! Ack map (Who has seen which deltas)

Building block: Dot

A “dot” is a two-member tuple:

```
@type dot :: {node_id :: term(), value :: non_neg_integer()}
```

Dots are generated in order!

```
next_dot(node_id, counter) = {node_id, counter + 1}
```

```
{“Derek”, 0}, {“Derek”, 1}, {“Derek”, 2}, ...
```

Building block: DotSet

DotSet is an add-remove Set, but more restrictive.

Difference 1: can only add “next dot”

Difference 2: merge function

```
@type dot :: {node_id :: term(), value :: non_neg_integer()}

@type dotset :: {state :: [dot()], context :: [dot()]}
```

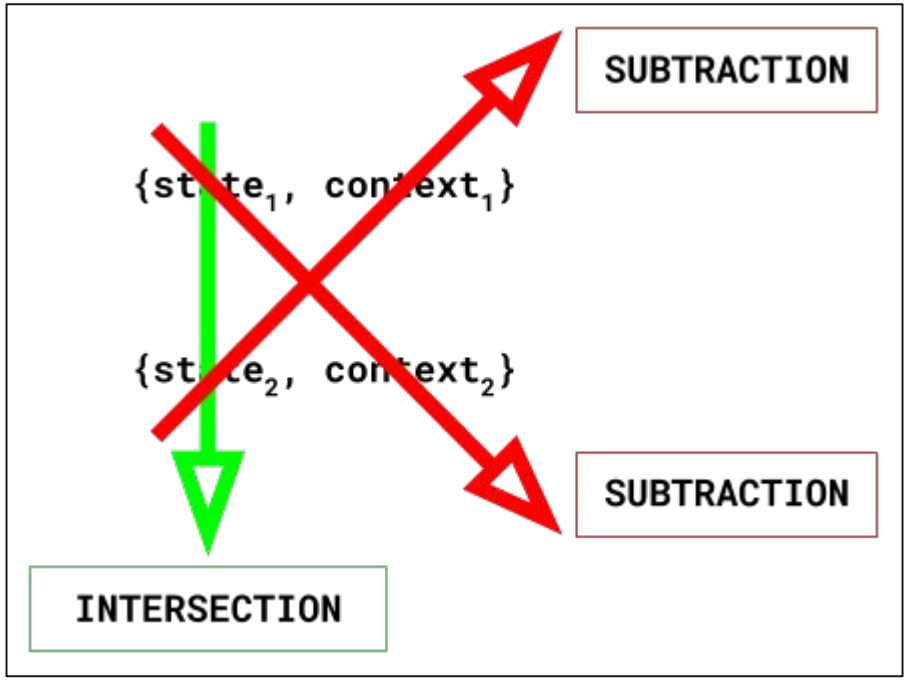
Operations: DotSet

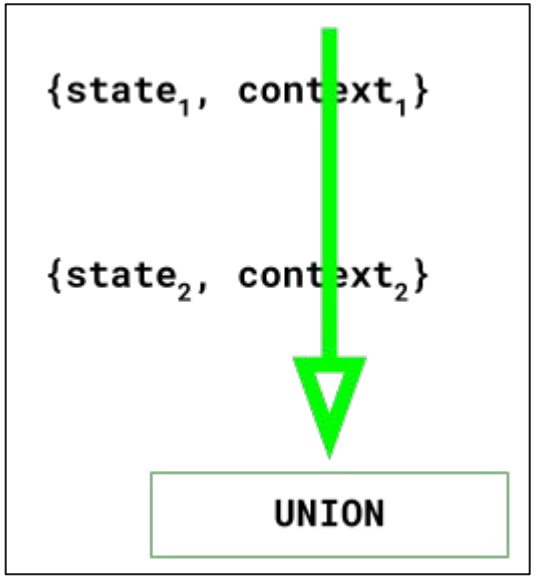
```
Add: {[next_dot()], [next_dot()]}

Remove(dot): {[], [dot]}

merge({s, c}, {s', c'}) = {(s∩s')∪(s\c')∪(s'\c), c∪c'}
```

Merge operation: DotSet





Example:

Initial state:	`{ [], []}`
Delta (*ADD*):	`{[{“Derek”, 1}], [{“Derek”, 1}]}`
Result:	`{[{“Derek”, 1}], [{“Derek”, 1}]}`
Delta (*ADD*):	`{[{“Derek”, 2}], [{“Derek”, 2}]}`
Result:	`{[{“Derek”, 1}, {“Derek”, 2}], [{“Derek”, 1}, {“Derek”, 2}]}`
Delta (*REMOVE*):	`{ [], [{“Derek”, 1}]}`
Result:	`{ [[“Derek”, 2}], [{“Derek”, 1}, {“Derek”, 2}]}`

```
Add:      {[dot], [dot]}
Remove:      {[], [dot]}
```

What can we build with it?

The simplest delta-CRDT we can imagine

- A single bit!
- Turn it off and on
- Enable-wins

$$\mathsf{EWFlag} = \mathsf{Causal}\langle\mathsf{DotSet}\rangle$$

$$\mathsf{enable}_i^\delta((s,c)) = (d, d \cup s) \quad \textbf{where } d = \{\mathsf{next}_i(c)\}$$

$$\mathsf{disable}_i^\delta((s,c)) = (\{\}, s)$$

$$\mathsf{read}((s,c)) = s \neq \{\}$$

Efficient Synchronization of State-based CRDTs

Vitor Enes
HASLab / INESC TEC and
Universidade do Minho
Portugal

Paulo Sérgio Almeida
HASLab / INESC TEC and
Universidade do Minho
Portugal

Carlos Baquero
HASLab / INESC TEC and
Universidade do Minho
Portugal

João Leitão
NOVA LINCS, FCT and
Universidade NOVA de Lisbon
Portugal

***Abstract*—To ensure high availability in large scale distributed systems, *Conflict-free Replicated Data Types* (CRDTs) relax consistency by allowing immediate query and update operations at the local replica, with no need for remote synchronization. State-based CRDTs synchronize replicas by periodically sending their full state to other replicas, which can become extremely costly as the CRDT state grows. Delta-based CRDTs address this problem by producing small incremental states (deltas) to be used in synchronization instead of the full state. However, current synchronization algorithms for delta-based CRDTs induce redundant wasteful delta propagation, performing worse than expected, and surprisingly, no better than state-based. In this paper we: 1) identify two sources of inefficiency in current synchronization algorithms for delta-based CRDTs; 2) bring the concept of join decomposition to state-based CRDTs; 3) exploit join decompositions to obtain optimal deltas and 4) improve the efficiency of synchronization algorithms; and finally, 5) experimentally evaluate the improved algorithms.**

***Index Terms*—CRDTs; Optimal Deltas; Join Decomposition;**

I. Introduction

Large-scale distributed systems often resort to replication techniques to achieve fault-tolerance and load distribution. These systems have to make a choice between availability and low latency or strong consistency [1]–[3], many times opting for the first [5], [6]. A common approach is to allow replicas of some data type to temporarily diverge, making sure these replicas will eventually converge to the same state in a deterministic way. *Conflict-free Replicated Data Types* (CRDTs) [7], [8] can be used to achieve this. They are key components in modern geo-replicated systems, such as Riak [9], Redis [10], and Microsoft Azure Cosmos DB [11].

CRDTs come mainly in two flavors: *operation-based* and *state-based*. In both, queries and updates can be executed immediately at each replica, which ensures availability (as it never needs to coordinate beforehand with remote replicas to execute operations). In operation-based CRDTs [7], [12], operations are disseminated assuming a reliable dissemination layer that ensures exactly-once causal delivery of operations.

State-based CRDTs need fewer guarantees from the communication channel: messages can be dropped, duplicated, and reordered. When an update operation occurs, the local state is updated through a mutator, and from time to time (since we can disseminate the state at a lower rate than the rate of the updates) the full (local) state is propagated to other replicas.

Although state-based CRDTs can be disseminated over unreliable communication channels, as the state grows, sending the full state becomes unacceptably costly. Delta-based

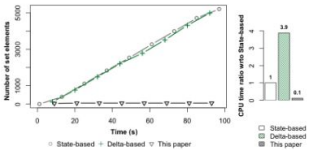


Fig. 1: Experiment setup: 15 nodes in a partial mesh topology replicating an always-growing set. The left plot depicts the number of elements being sent throughout the experiment, while the right plot shows the CPU processing time ratio with respect to state-based. Not only does delta-based synchronization not improve state-based in terms of state transmission, it even incurs a substantial processing overhead.

CRDTs [13], [14] address this issue by defining delta-mutators that return a delta (δ), typically much smaller than the full state of the replica, to be merged with the local state. The same δ is also added to an outbound δ-buffer, to be periodically propagated to remote replicas. Delta-based CRDTs have been adopted in industry as part of Akka Distributed Data framework [15] and IPFS [16], [17].

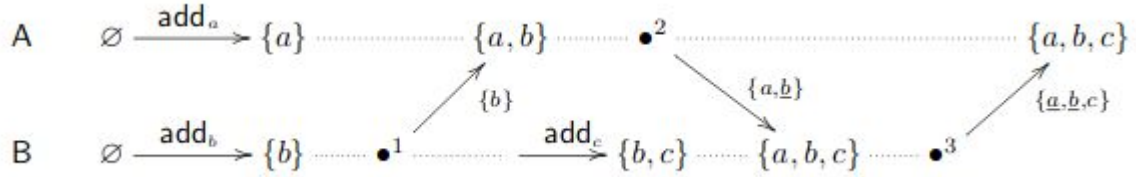
However, and somewhat unexpectedly, we have observed (Figure 1) that current delta-propagation algorithms can still disseminate much redundant state between replicas, performing worse than envisioned, and no better than the state-based approach. This anomaly becomes noticeable when concurrent update operations always occur between synchronization rounds, and it is partially justified due to inefficient redundancy detection in delta-propagation.

In this paper we identify two sources of redundancy in current algorithms, and introduce the concept of join decomposition of a state-based CRDT, showing how it can be used to derive optimal deltas ("differences") between states, as well as optimal delta-mutators. By exploiting these concepts, we also introduce an improved synchronization algorithm, and experimentally evaluate it, confirming that it outperforms current approaches by reducing the amount of state transmission, memory consumption, and processing time required for delta-based synchronization.

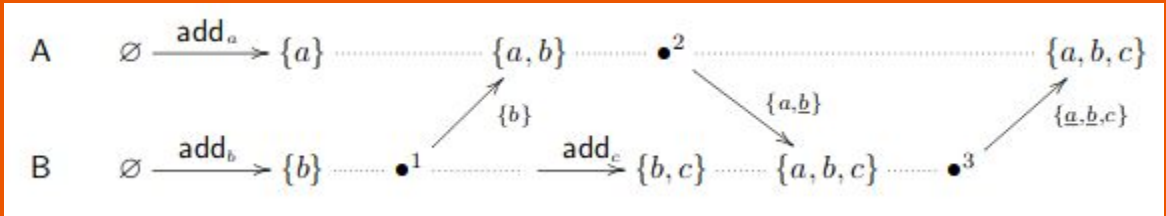
The back-propagation problem

Back-propagation

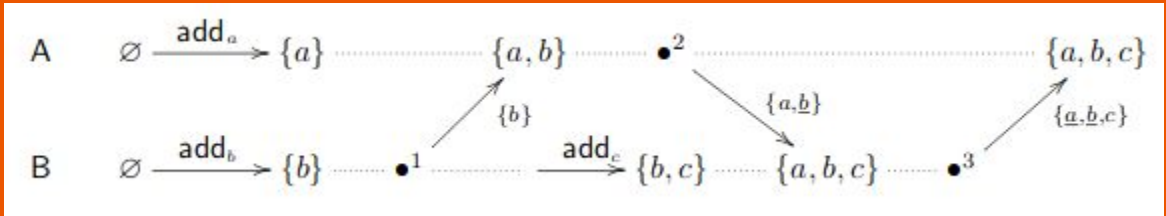
Basically: sending a node a delta it has already seen.



Avoid back-propagation



Remove redundant state from delta before adding it to deltas



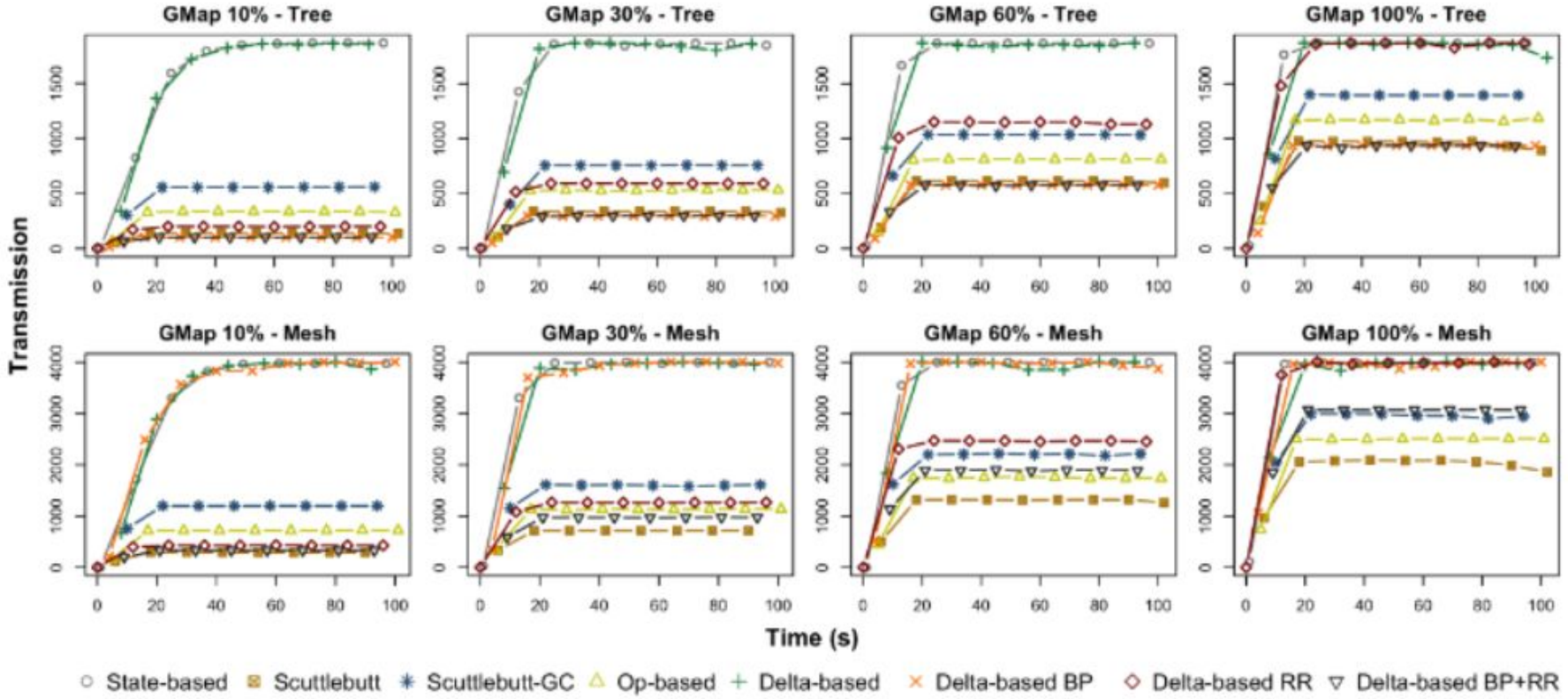
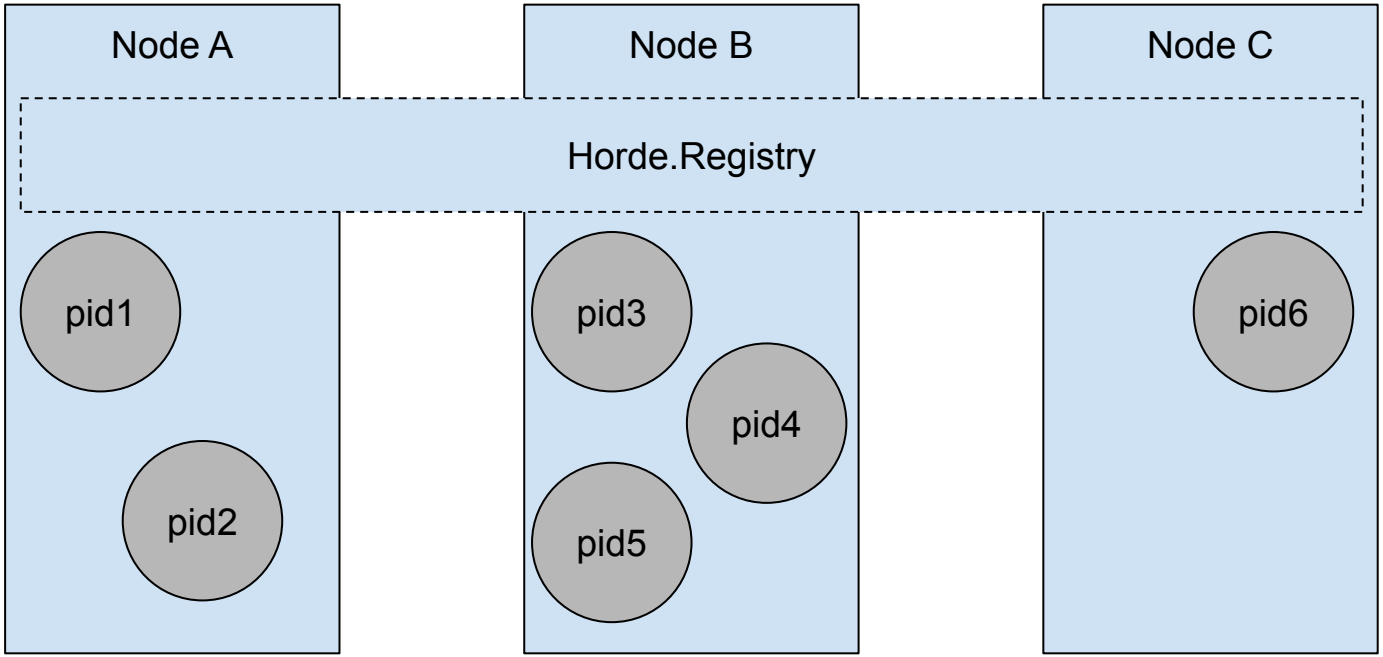


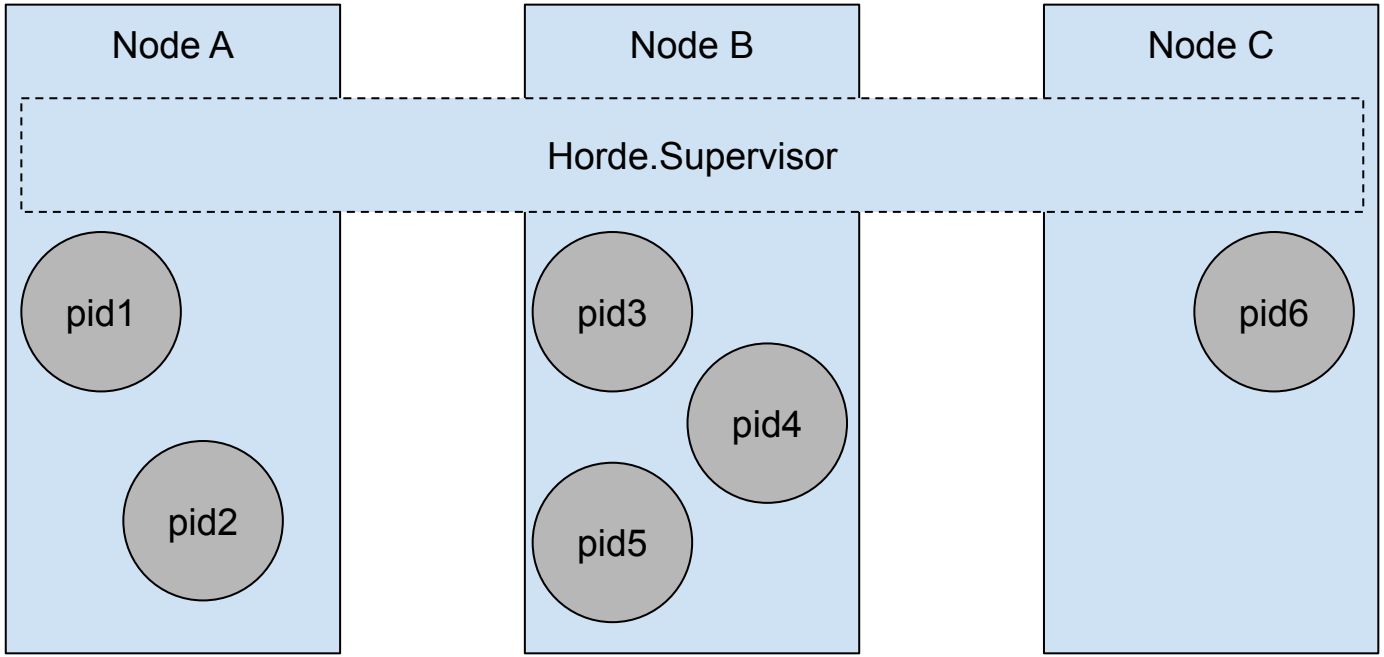
Fig. 8: Transmission of GMap 10%, 30%, 60% and 100% – tree and mesh topologies.

Applications on the BEAM

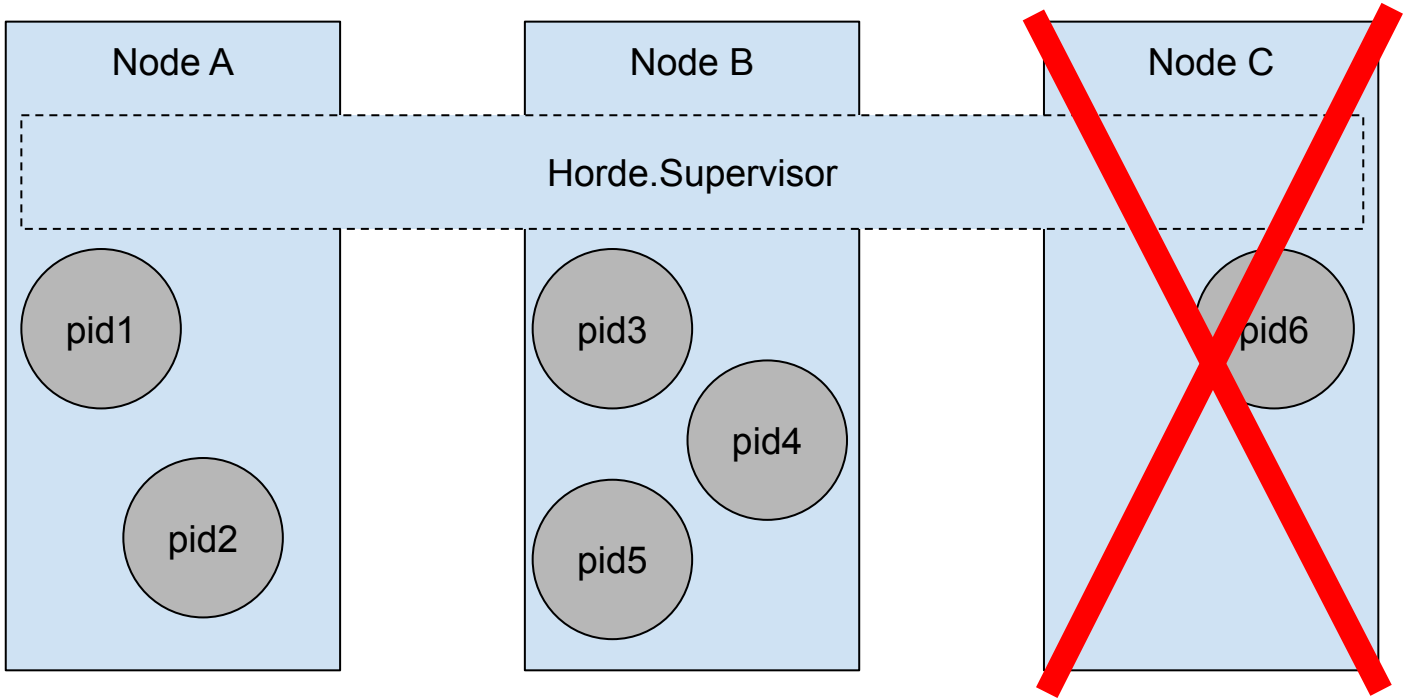
Horde.Registry



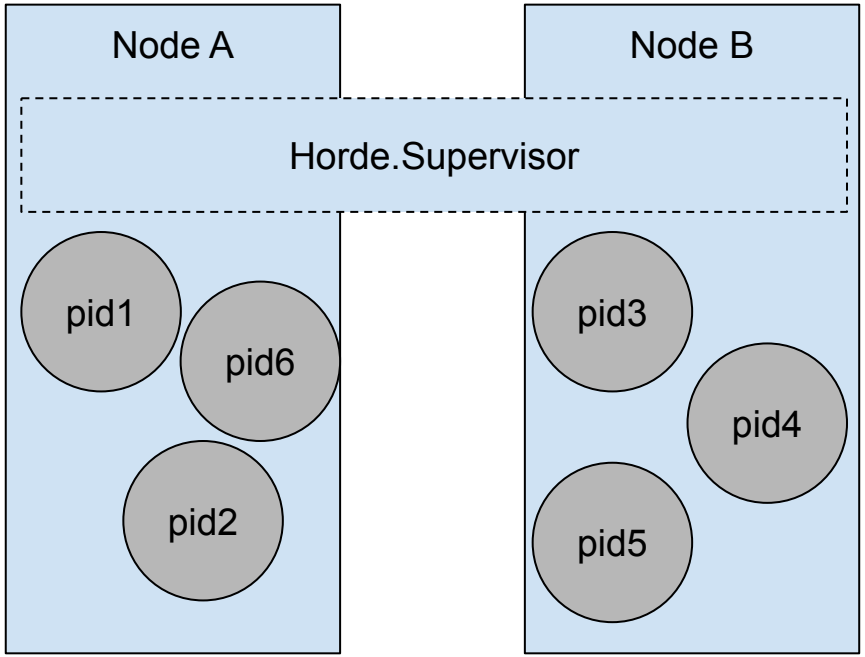
Horde.Supervisor



Horde.Supervisor



Horde.Supervisor



Links!

Horde - https://github.com/derekkraan/horde/

DeltaCRDT - https://github.com/derekkraan/delta_crdt_ex

Conflict-free Replicated Data Types - https://hal.inria.fr/inria-00609399v1/document

Delta State Replicated Data Types - https://arxiv.org/pdf/1603.01529.pdf

Efficient Synchronization of State-Based CRDTs - https://arxiv.org/pdf/1803.02750.pdf

Time, Clocks, and the Ordering of Events in a Distributed System - https://lamport.azurewebsites.net/pubs/time-clocks.pdf

Questions?