Practical Data Synchronization & *CRDTs*

Dmitry Ivanov @idajantis

QCon SAN FRANCISCO

2016

TOMTOM

A comprehensive study of Convergent and Commutative Replicated Data Types

Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski

▶ **To cite this version:**

Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marel
Convergent and Commutative Replicated Data Types
Centre Paris-Rocquencourt; INRIA. 2011, pp.50. <inri

CRDTs: Consistency without concurrency control*

Mihai Leția†, Nuno Preguiça‡, Marc Shapiro§

Thème COM — Systèmes communicants
Projet Regal

Rapport de recherche n° 6956 — Juin 2009 — 13 pages

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

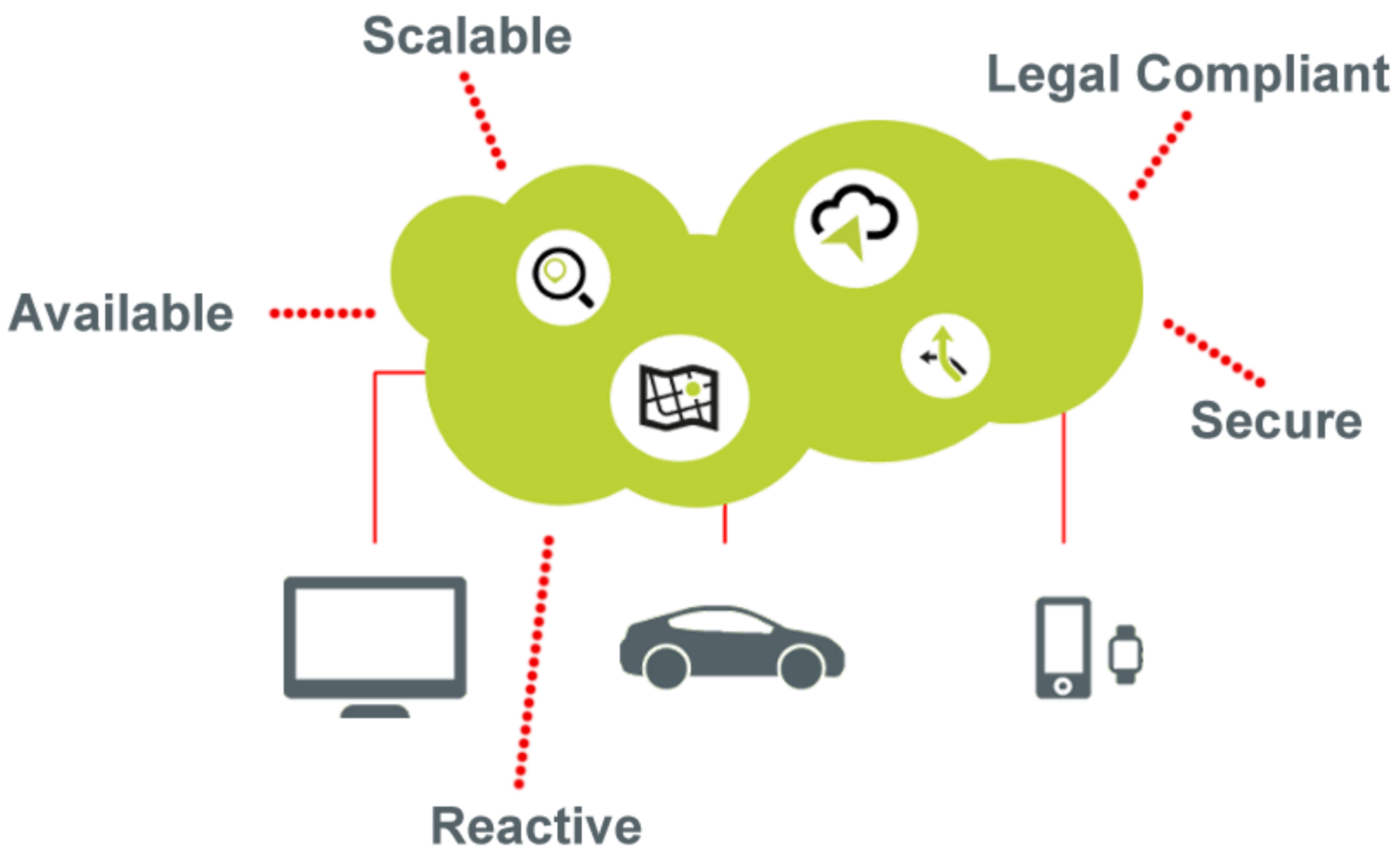
Thème COM — Systèmes communicants
Projet Regal

Rapport de recherche n° 7687 — Juillet 2011 — 18 pages

when they are concurrent.
ncurrency control. As an
uffer called Treedoc. We
Ve discuss how the CRDT

ive operations

NavCloud



Who We Are

"Fool" stack developers hacking on:

- Backend services
- Client libraries
- Infrastructure && DevOps

Backend stack

akka

RabbitMQ™

spray

riak

amazon web services™

Client Libraries

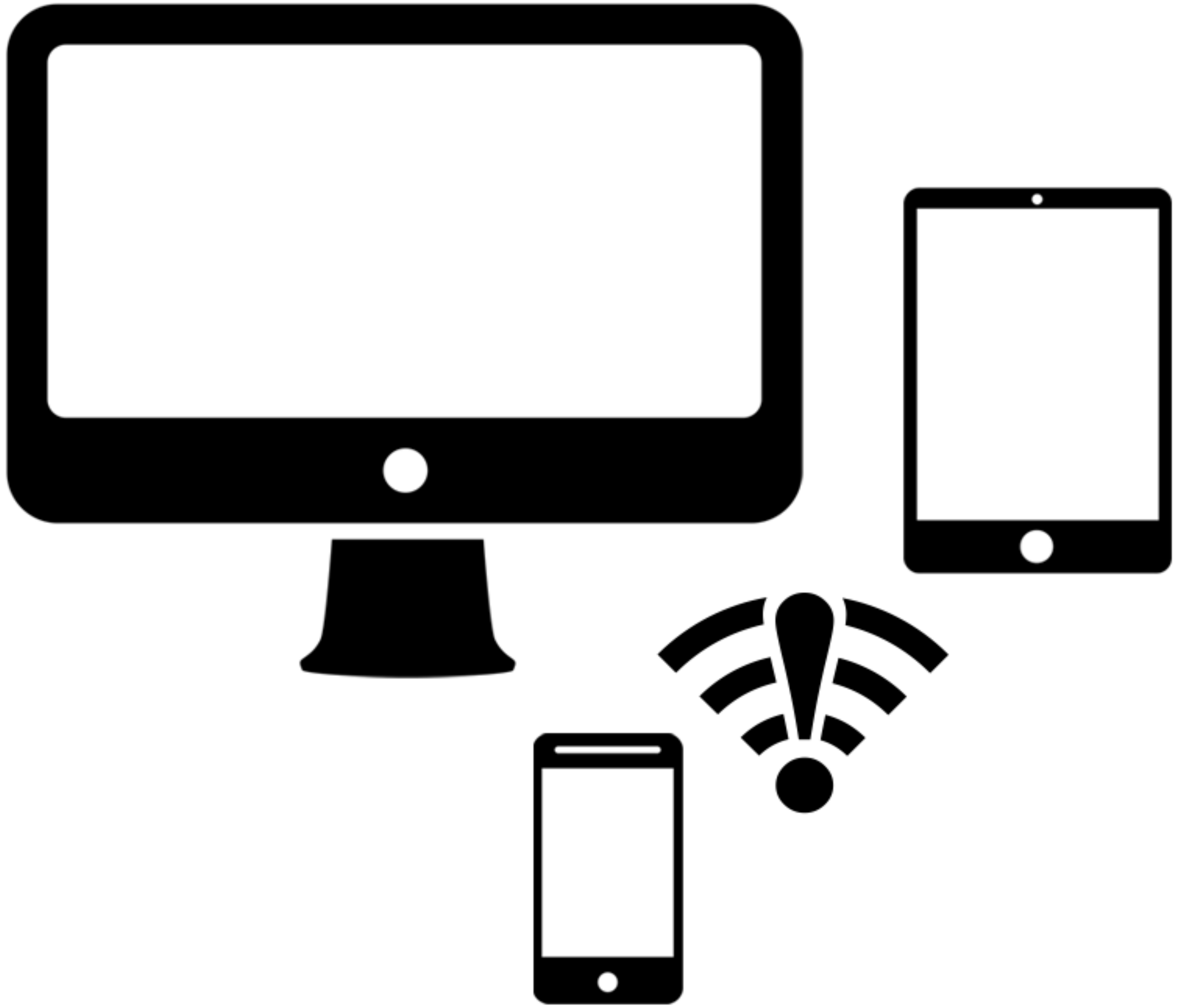
Java

C++

JavaScript

NavCloud Nature

- **Unstable connections**
- **Limited data** plans & **bandwidth**
- Seamless edit/view in **offline** mode
- Concurrent **changes** with potential **conflicts**
- No guarantee on updates **order**
- No **data loss**
- Data **convergence** to expected value



How to Deal with this Nature?

Bad programmers worry about the code. Good programmers worry about data structures

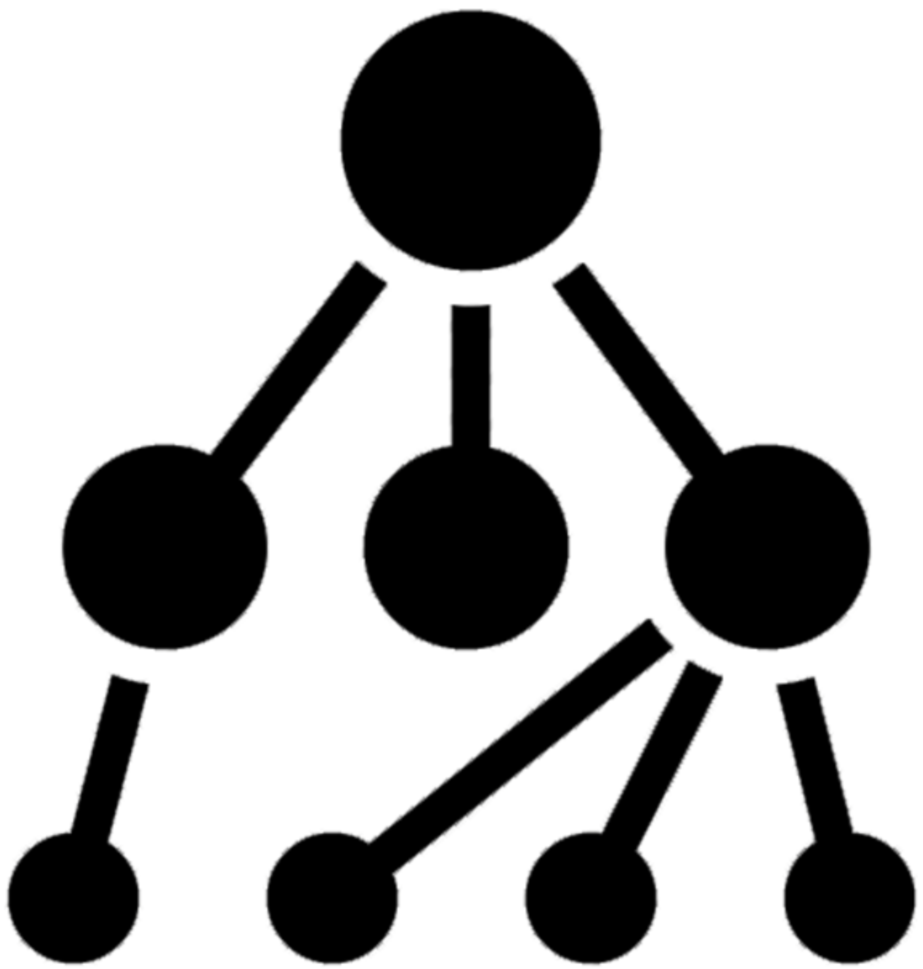
— *Linus Torvalds*

CRDT

CR**DT**

DT: Data Type

CRDT is a data type with its own algebra



CRDT

R: Replicated

CRDT is a family of data structures which has been designed to be distributed

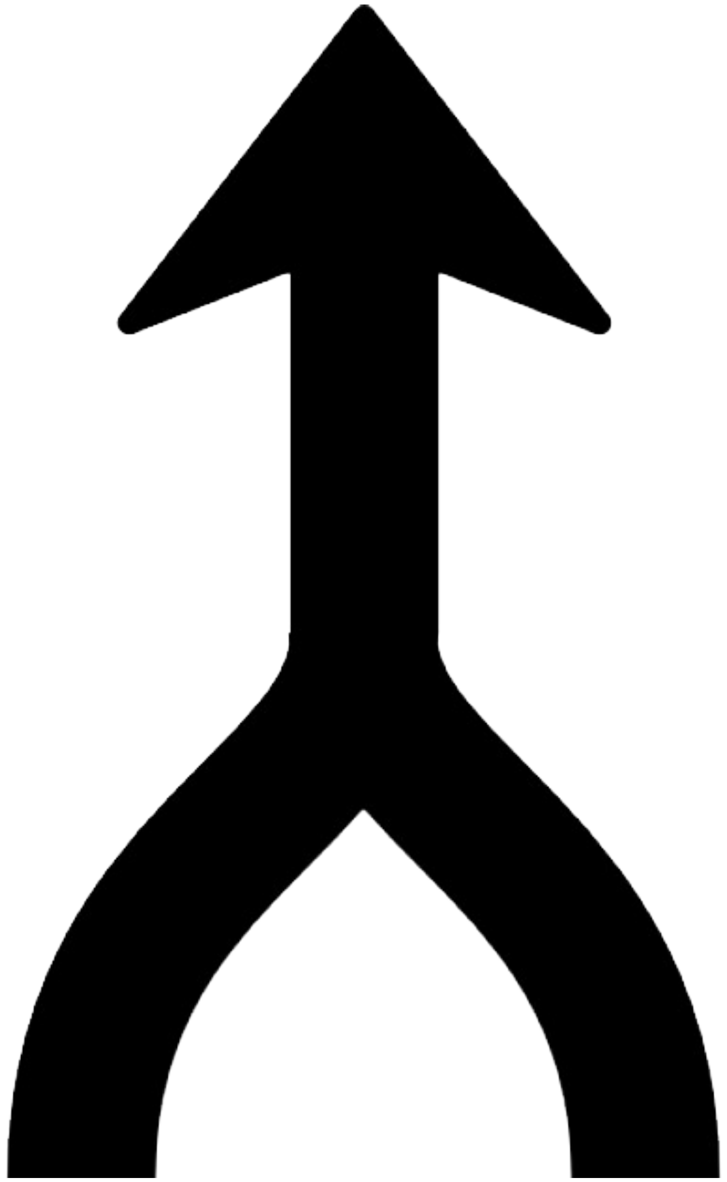
CRDT

C: Conflict Free

Resolving conflicts is done automatically

How?

Merge



What is Merge?

- A binary operation on two CRDTs
  - **Commutative**: x • y = y • x
  - **Associative**: (x • y) • z = x • (y • z)
  - **Idempotent**: x • x = x

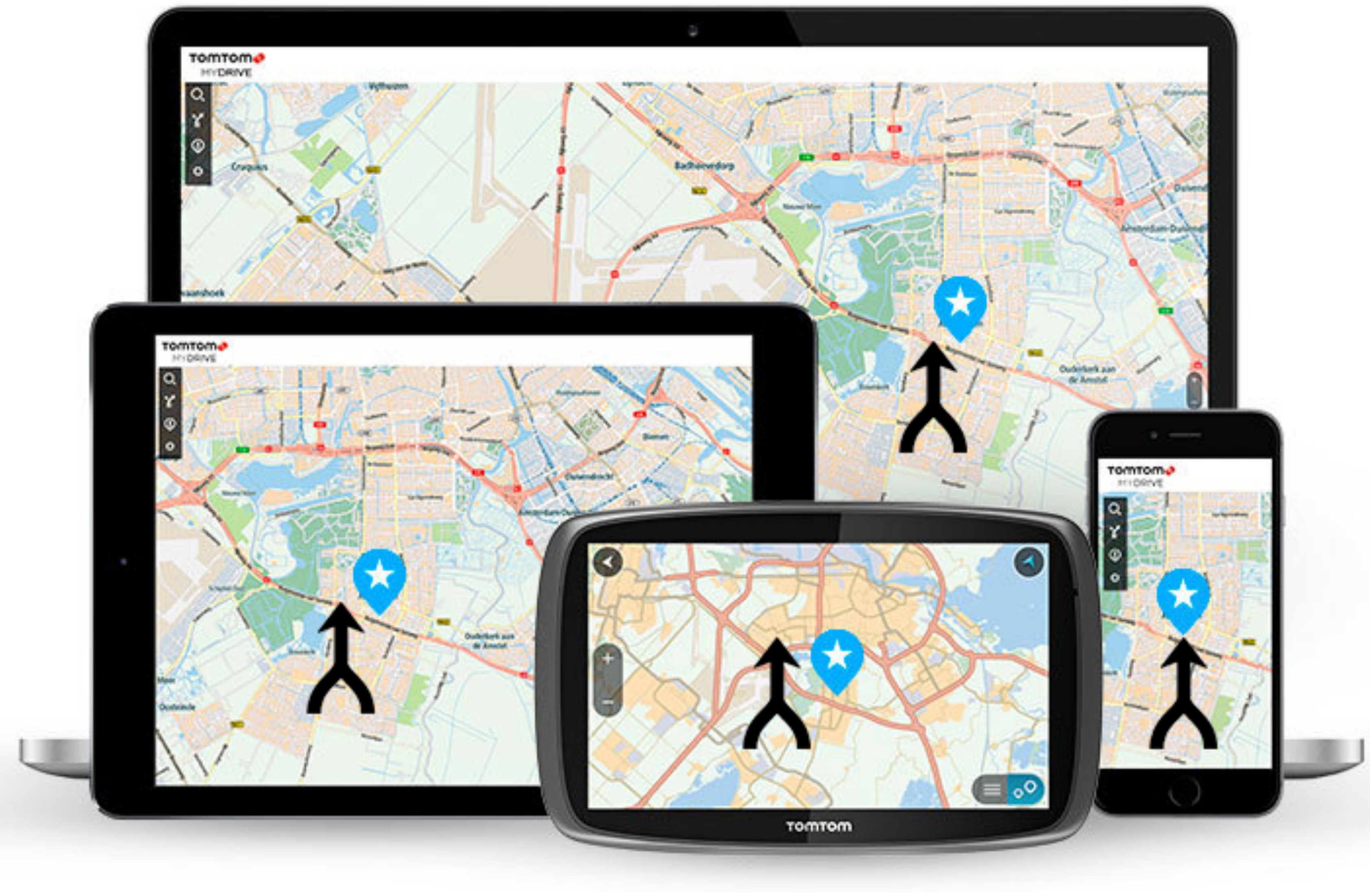
How Does it Help?

In **Distributed Systems**:

- **Order** is not guaranteed:
  - No Problem: **Merge** is **Commutative** and **Associative**
- Events can be delivered more than **once**:
  - No problem: **Merge** is **Idempotent**

What Does it Bring in Practice?

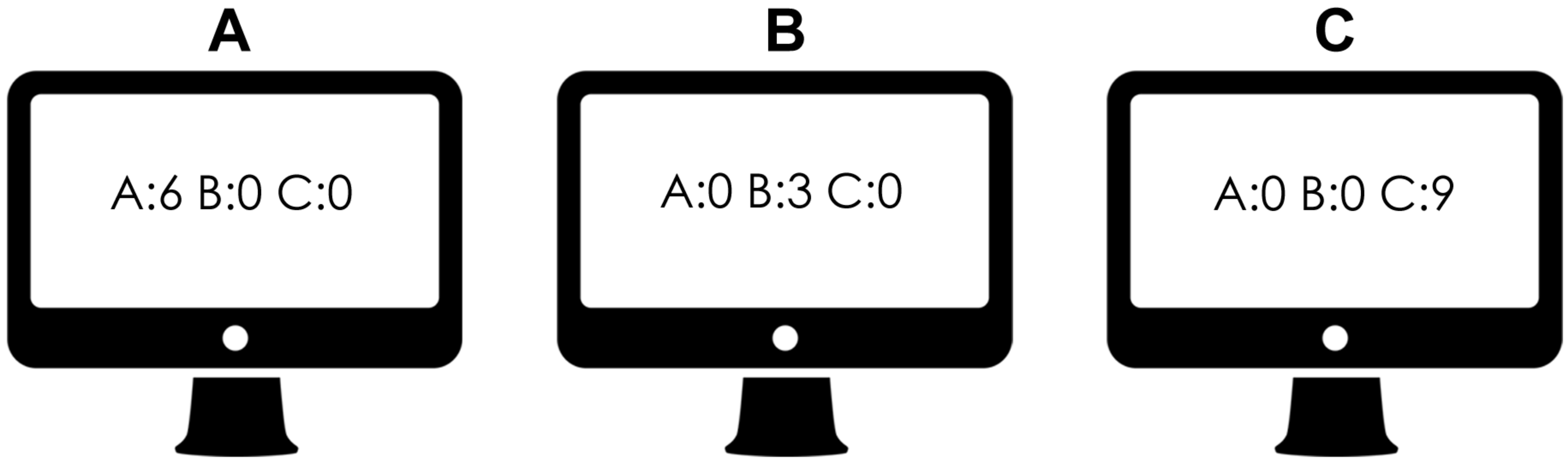
- **Local** updates
- **Local merge** of receiving data
- All local merges **converge**



Examples

G-Counter

G-Counter



Merge: **Max** of corresponding elements: A:6 B:3 C:9

TotalValue: Sum of all elements: 6 + 3 + 9 = 18

Max Function

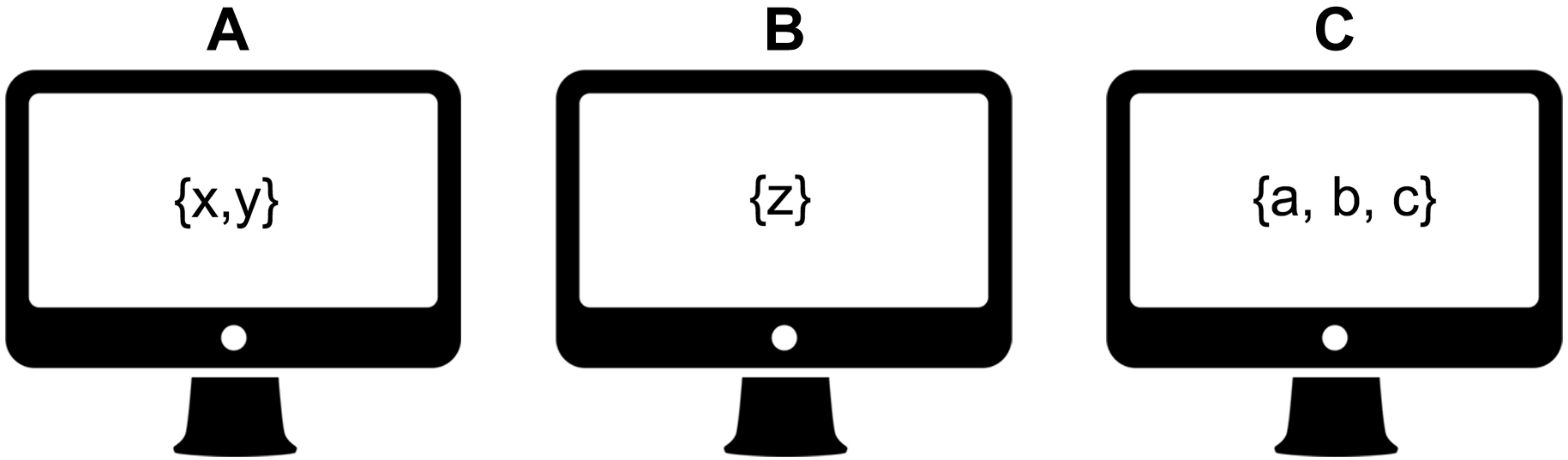
- A binary operation on two CRDTs
  - **Commutative**: x max y = y max x
  - **Associative**: (x max y) max z = x max (y max z)
  - **Idempotent**: x max x = x

G-Set

Union Function

- A binary operation on two CRDTs
  - **Commutative**: $x \cup y = y \cup x$
  - **Associative**: $(x \cup y) \cup z = x \cup (y \cup z)$
  - **Idempotent**: $x \cup x = x$

G-Set



Merge: **Union** of sets: { x, y, z, a, b, c }

Total Value: The same as the merge result

CRDT in NavCloud

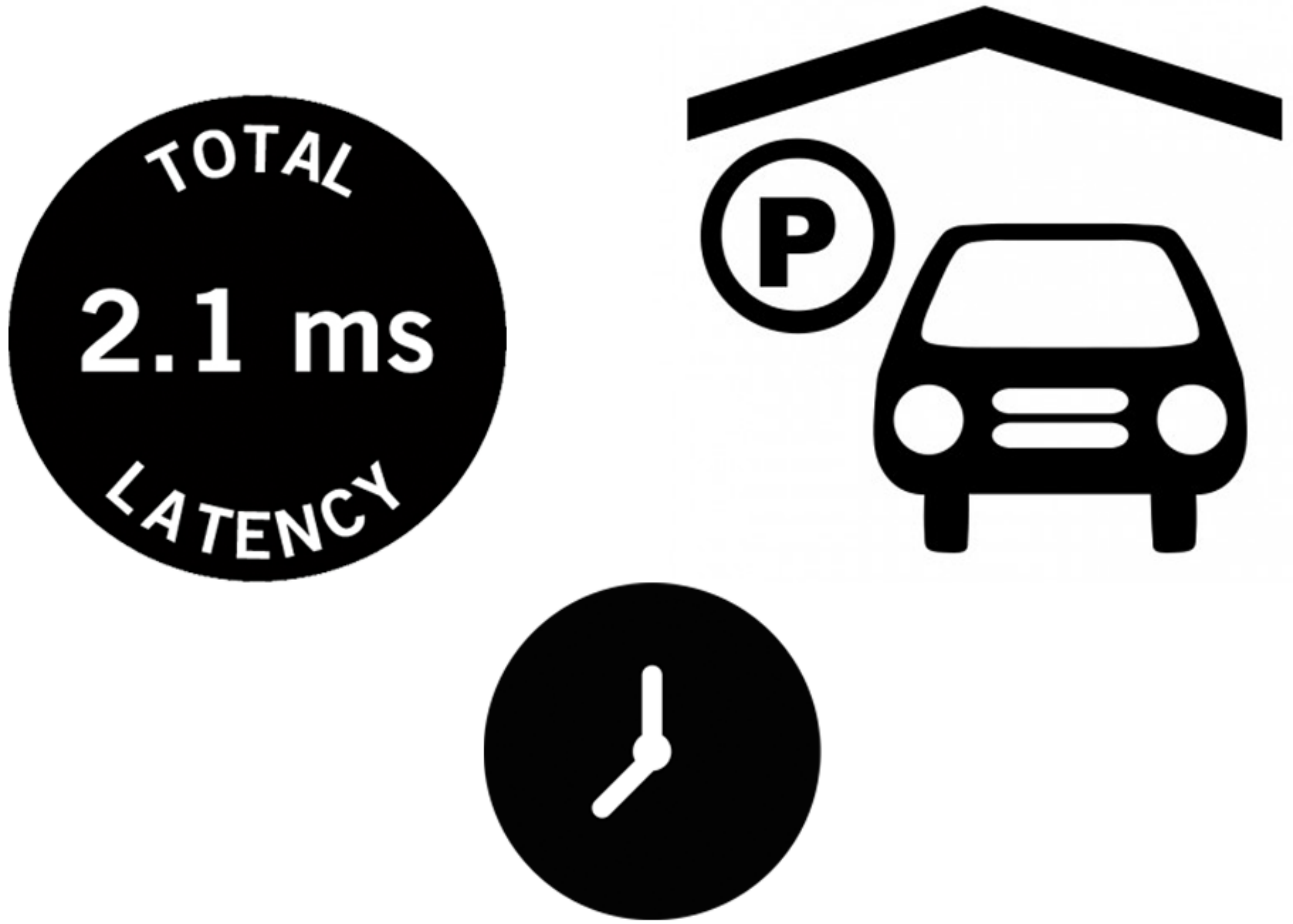
Favorite Locations Synchronization

Naive Approach?

Last Write Wins

Problems

- Unstable connections
  - Actual update time < Sent time
- Network latency
  - Sent time < Received time
- Unreliable clocks



TomTom

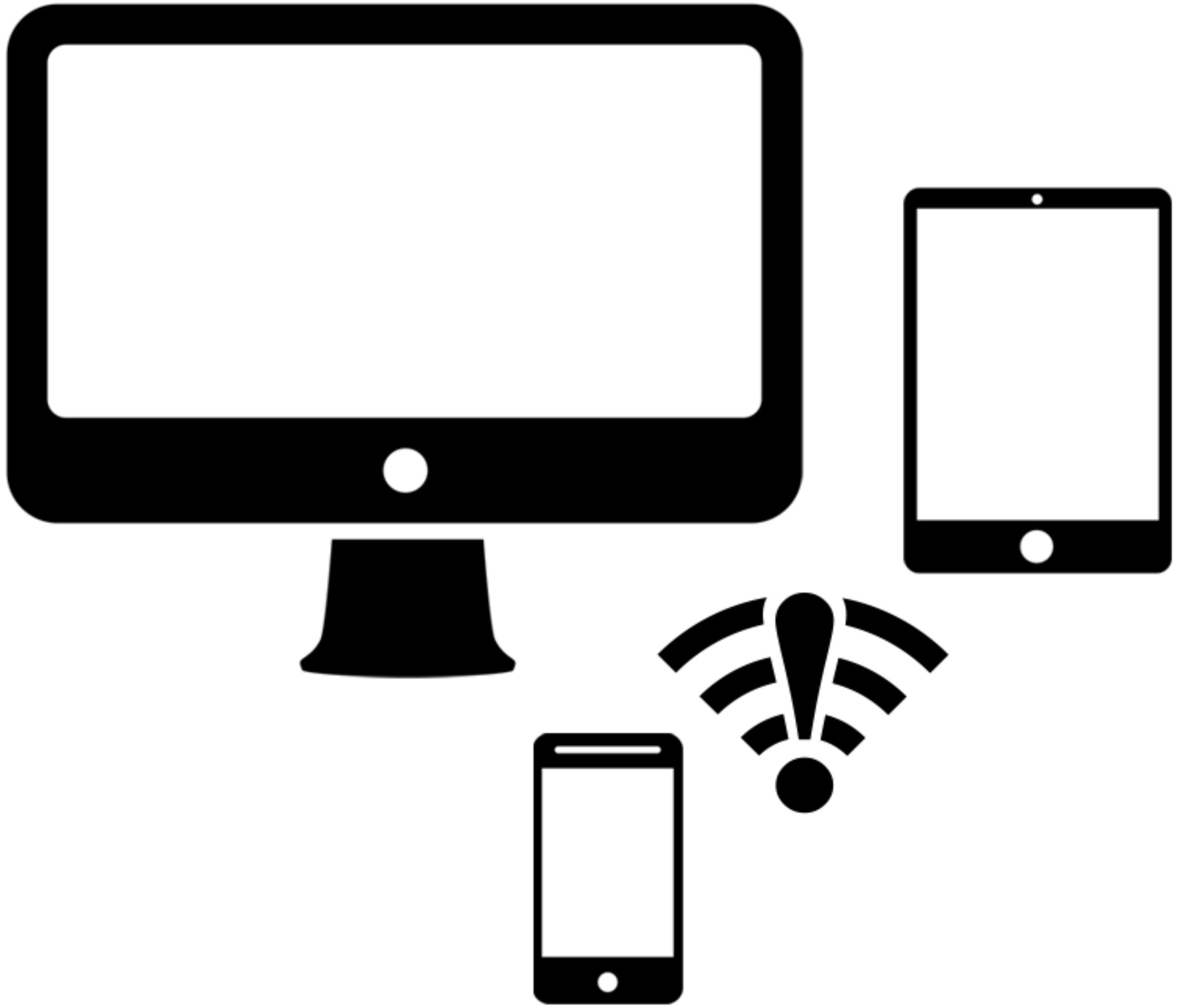
Stale update may win!

So What?

CRDT

NavCloud Nature vs CRDT

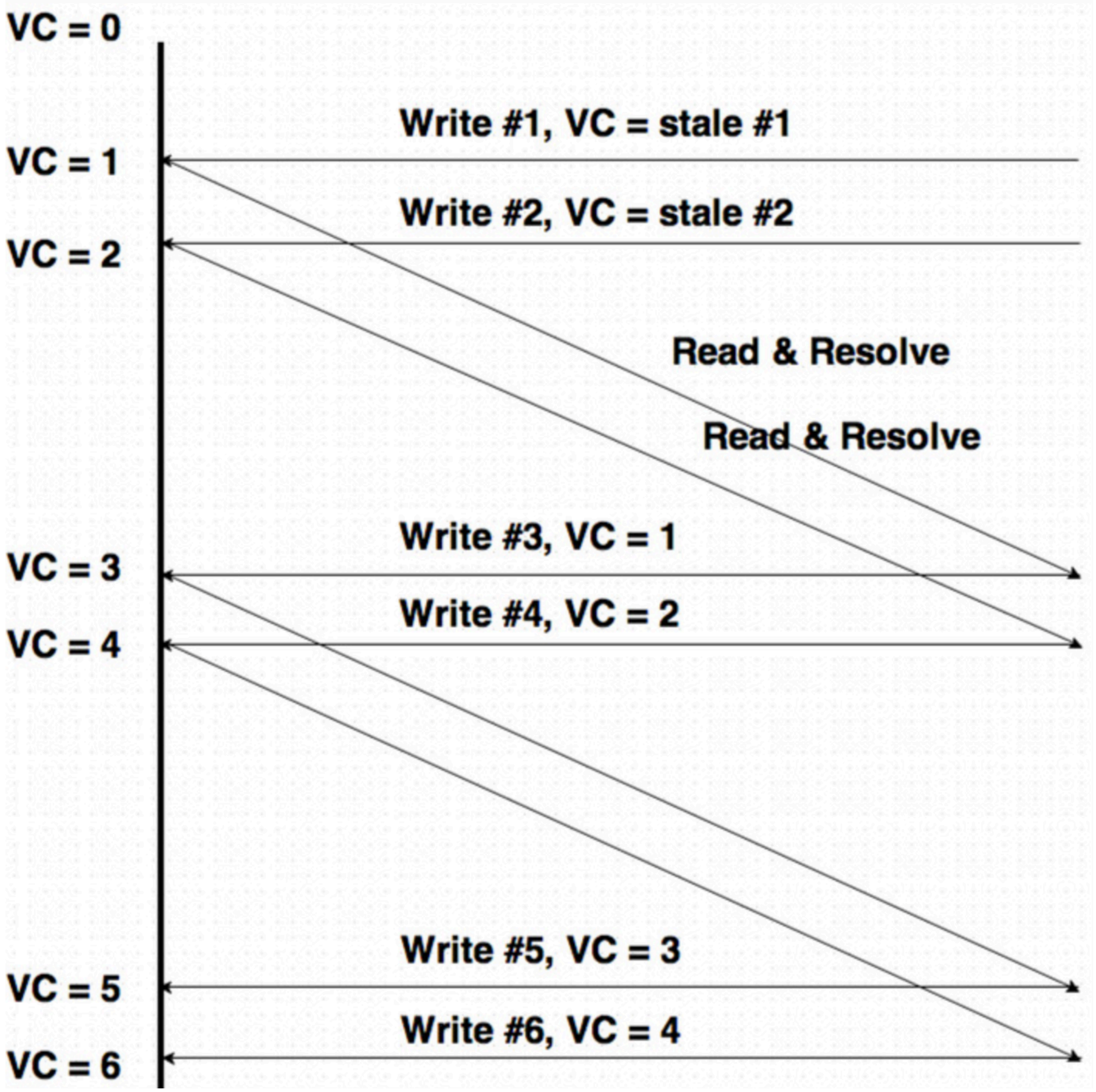
- **Unstable connections** ✔
- **Limited data** plans & **bandwidth** ✔
- Seamless edit/view in **offline** mode ✔
- Concurrent **changes** with potential **conflicts** ✔
- No guarantee on updates **order** ✔
- No **data loss** ✔
- Data **convergence** to expected value ✔



Same Data Model Everywhere

- **Server**
- **Clients**
- **Data store** riak

Merging Conflicts in Riak



The data consistency is determined by '*the weakest link*' in your pipeline

Implementing a CRDT Set

What do we want?

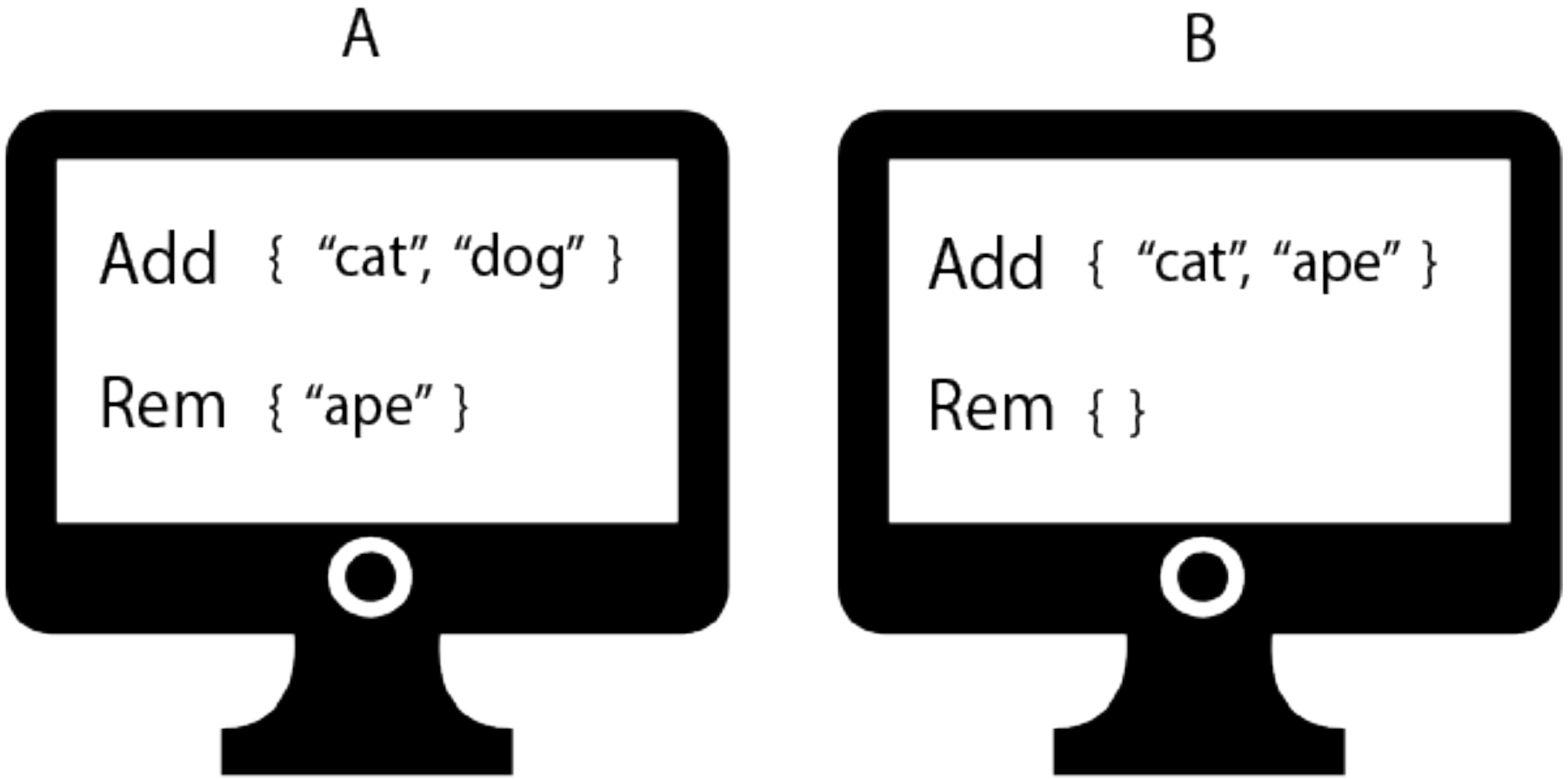
- Support for addition and removal operations.
- Optimized for element mutations.
- Footprint as compact as possible.

2-Phase-Set

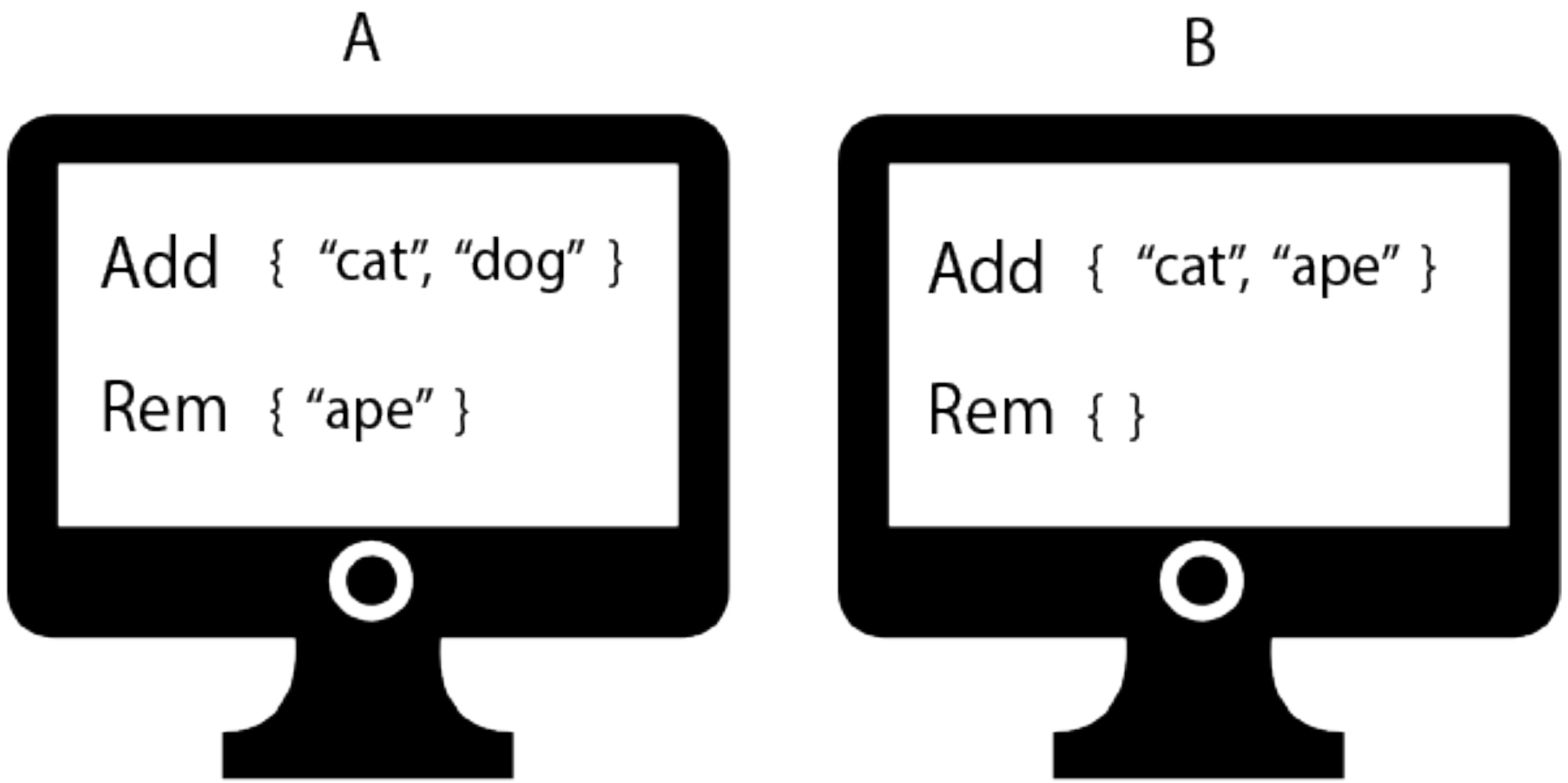
Supports additions and removals.

- **G-Set** for added elements
- **G-Set** for removed elements aka *Tombstones*

2-Phase-Set



2-Phase-Set



Merge: [Add { "cat", "dog", "ape" }; Rem { "ape" }]
Lookup: { "cat", "dog" }

2-Phase-Set

Lookup

```
def lookup: Set[E] = addSet.diff(removeSet).lookup
```

Merge

```
def merge(anotherSet: TwoPSet[E]): TwoPSet[E] =
   new TwoPSet(     addset.merge(anotherSet.addSet),
                  removeSet.merge(anotherSet.removeSet))
```

2-Phase-Set

Doesn't work for us:

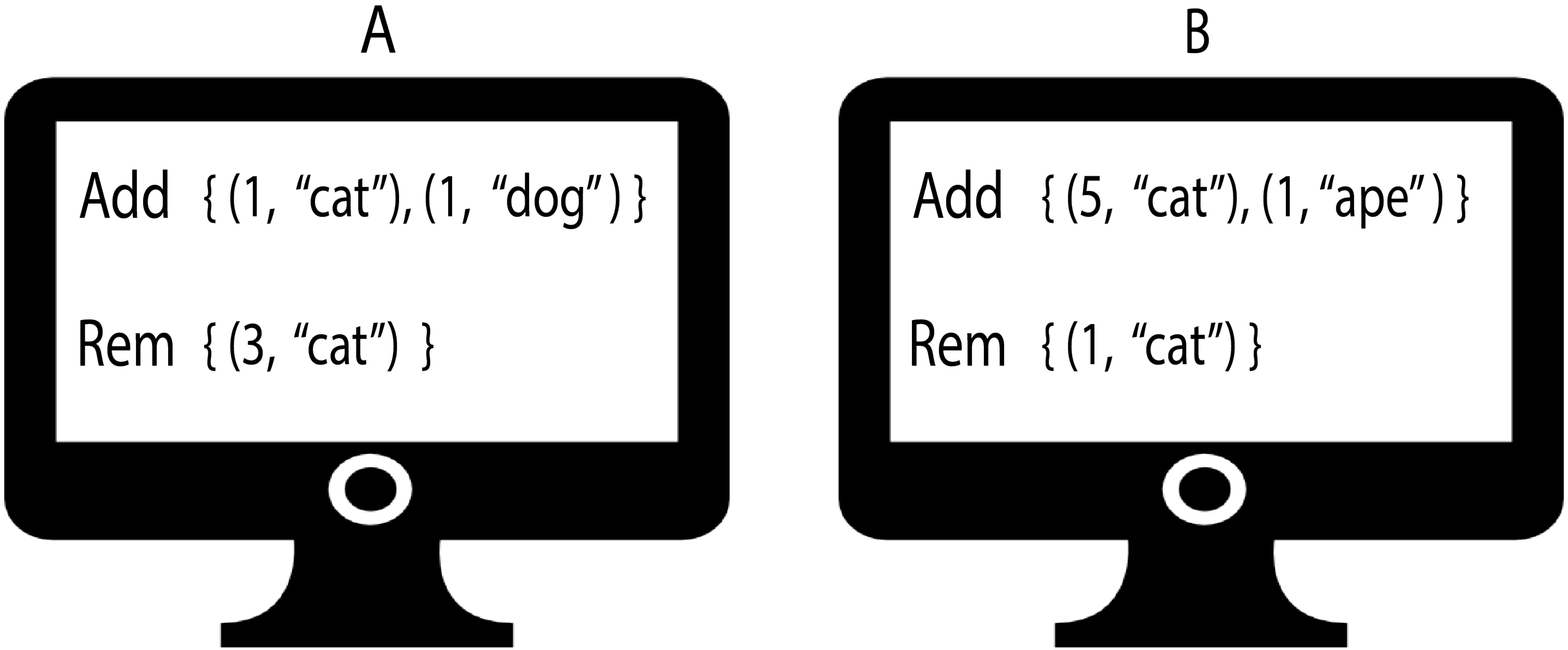
- Removed element can't be added again
- Immutable elements: no updates possible

LWW-Element-Set

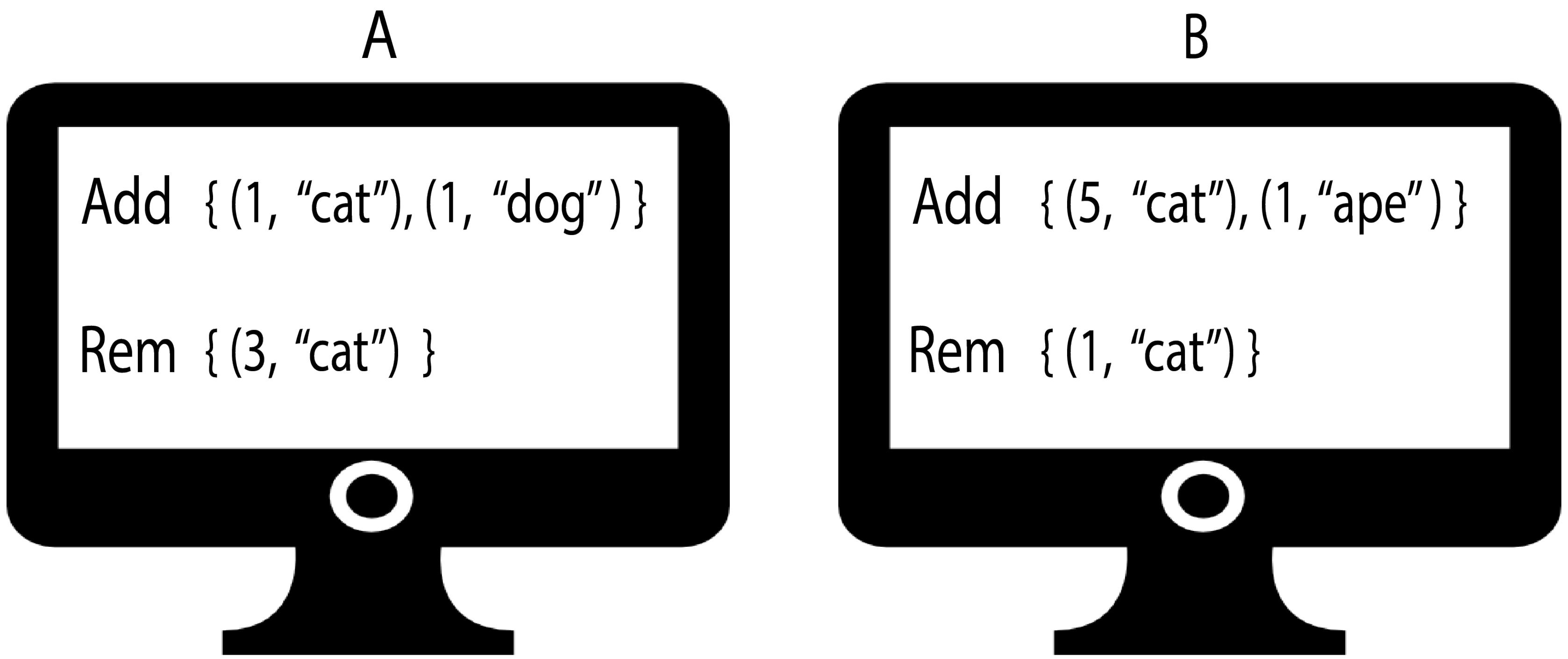
Supports additions and removals, with **timestamps**.

- **G-Set** for added elements
- **G-Set** for removed elements aka *Tombstones*
- Each element has a timestamp
- Supports re-adding removed element using a higher timestamp

LWW-Element-Set



LWW-Element-Set



Merge

Add { (1, "cat"), (5, "cat"), (1, "dog"), (1, "ape") }

Rem { (1, "cat"), (3, "cat") }

LWW-Element-Set

Merge

Add { (1, "cat"), (5, "cat"), (1, "dog"), (1, "ape") }
Rem { (1, "cat"), (3, "cat") }

Lookup

{ "cat", "dog", "ape" }

LWW-Element-Set

Lookup

```
def lookup: Set[E] = addSet.lookup.filter { addElem =>
    !removeSet.exists { removeElem =>
      removeElem.value == addElem.value && removeElem.timestamp > addElem.timestamp
    }
  }.map(_.value)
```

Merge

```
def merge(LWWSet<E> anotherSet): LWWSet<E> =
    new LWWSet(    addset.merge(anotherSet.addSet),
                removeSet.merge(anotherSet.removeSet))
```

LWW-Element-Set

Doesn't work for us:

- Immutable elements: no updates possible.

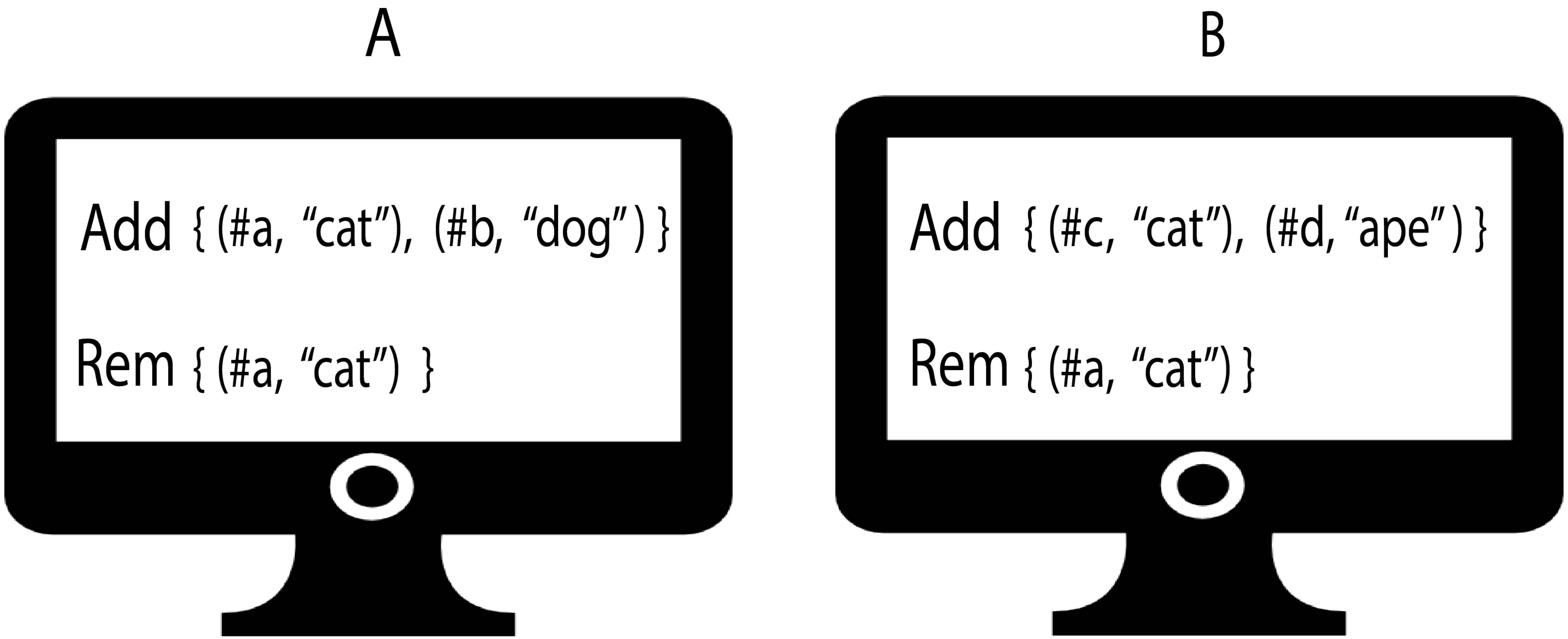
OR-Set

OR - Observed / Removed

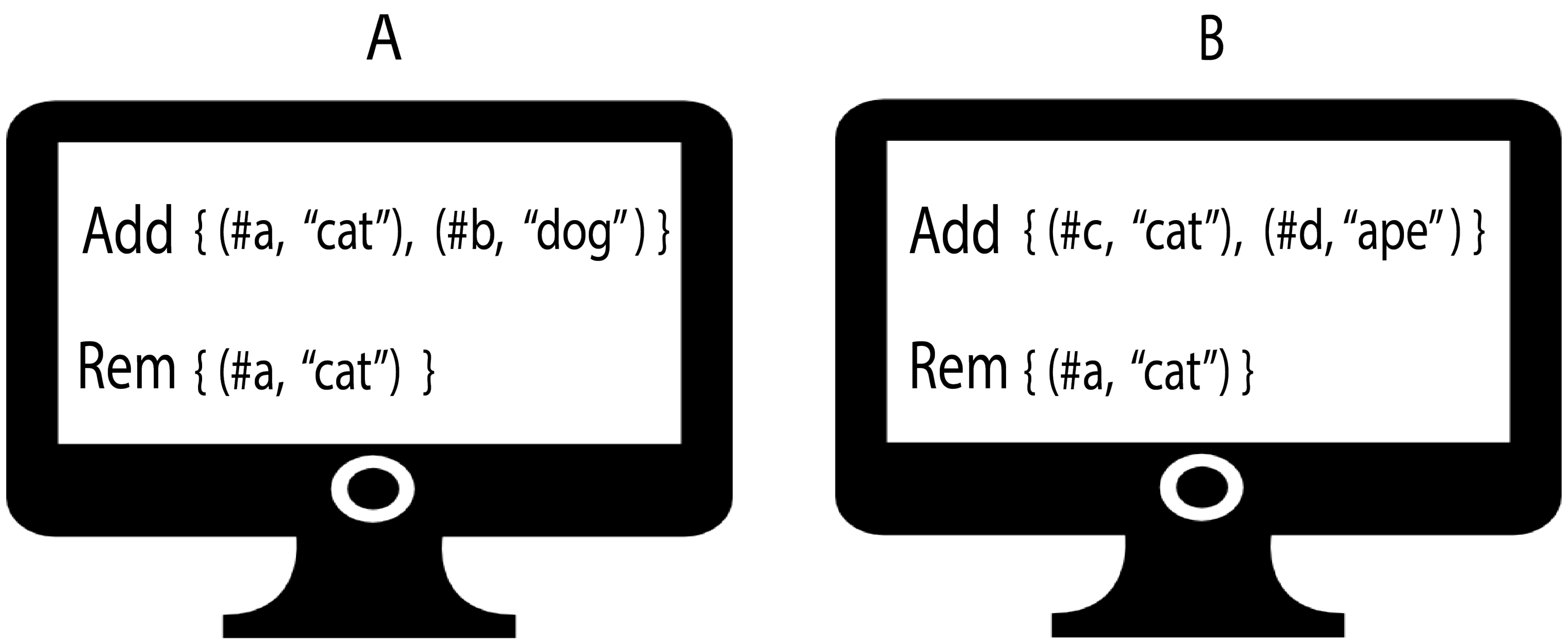
Supports additions and removals, with **tags**.

- **G-Set** for added elements
- **G-Set** for removed elements aka *Tombstones*
- Unique **tag** is associated with each element
- Supports re-adding removed elements

OR-Set



OR-Set



Merge

Add { (#a, "cat"), (#c, "cat"), (#b, "dog"), (#d, "ape") }

Rem { (#a, "cat") }

OR-Set

Merge

Add { (#a, "cat"), (#c, "cat"), (#b, "dog"), (#d, "ape") }
Rem { (#a, "cat") }

Lookup

{ "cat", "dog", "ape" }

OR-Set

Lookup

E exists iff it has in AddSet a tag that is not in the RemoveSet.

```
def lookup(): Set<E> =
  addSet.filter { addElem =>
        !removeSet.exists { remElem =>
            addElem.value == remElem.value
            && remElem.tag.equals(addElem.tag) }
        }
        .map(_.value);
```

OR-Set

Merge

```
def merge(anotherSet: ORSet[E]): ORSet[E] =
   new ORSet(   addset.merge(anotherSet.addSet),
             removeSet.merge(anotherSet.removeSet))
```

OR-Set

Doesn't work for us:

- Immutable elements: no updates possible.

OUR-Set

Our take on **O**bserved-**U**pdated-**R**emoved Set

- Each element has a unique **identifier**
- Element can be changed if identifier remains the same
- Each element has a **timestamp**
- Timestamp is updated on each element mutation

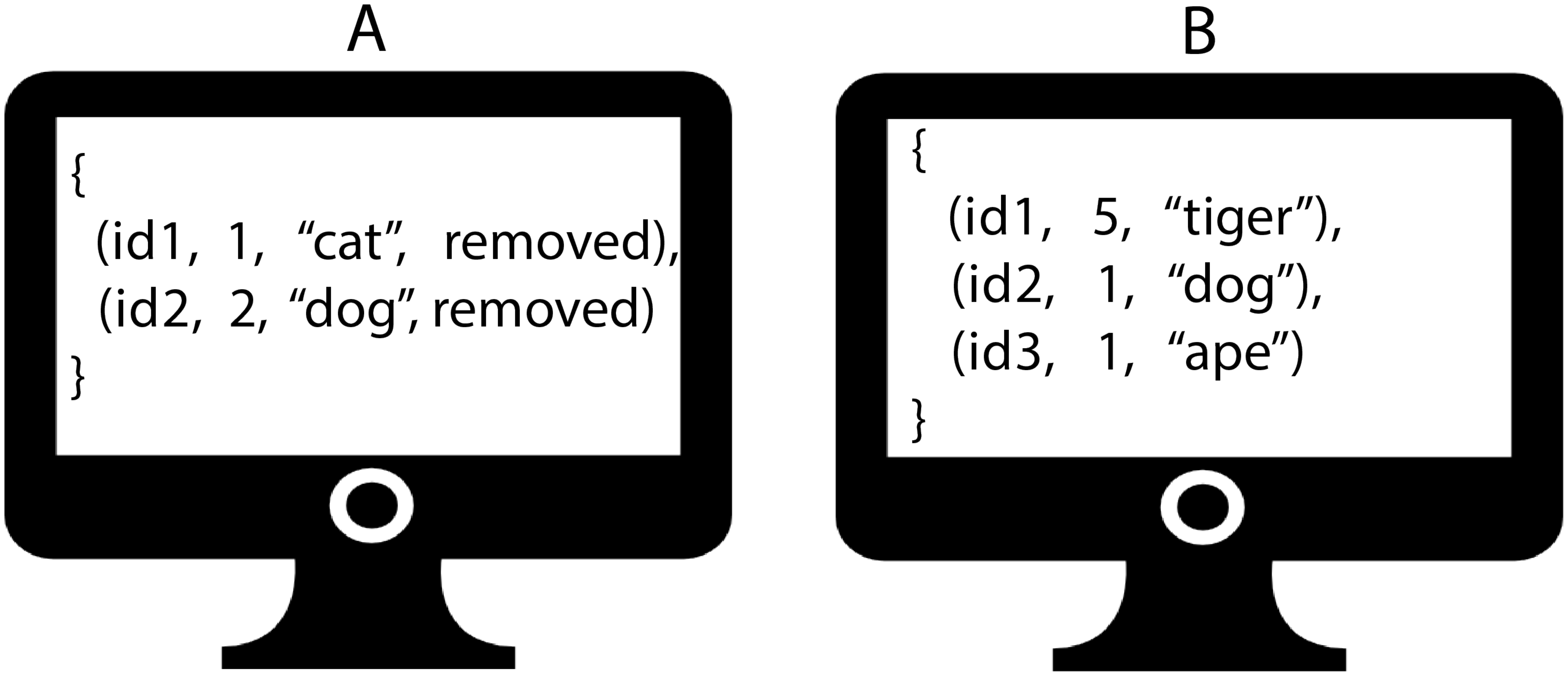
Identity (immutable unique id) vs **Value** (mutable)

OUR-Set

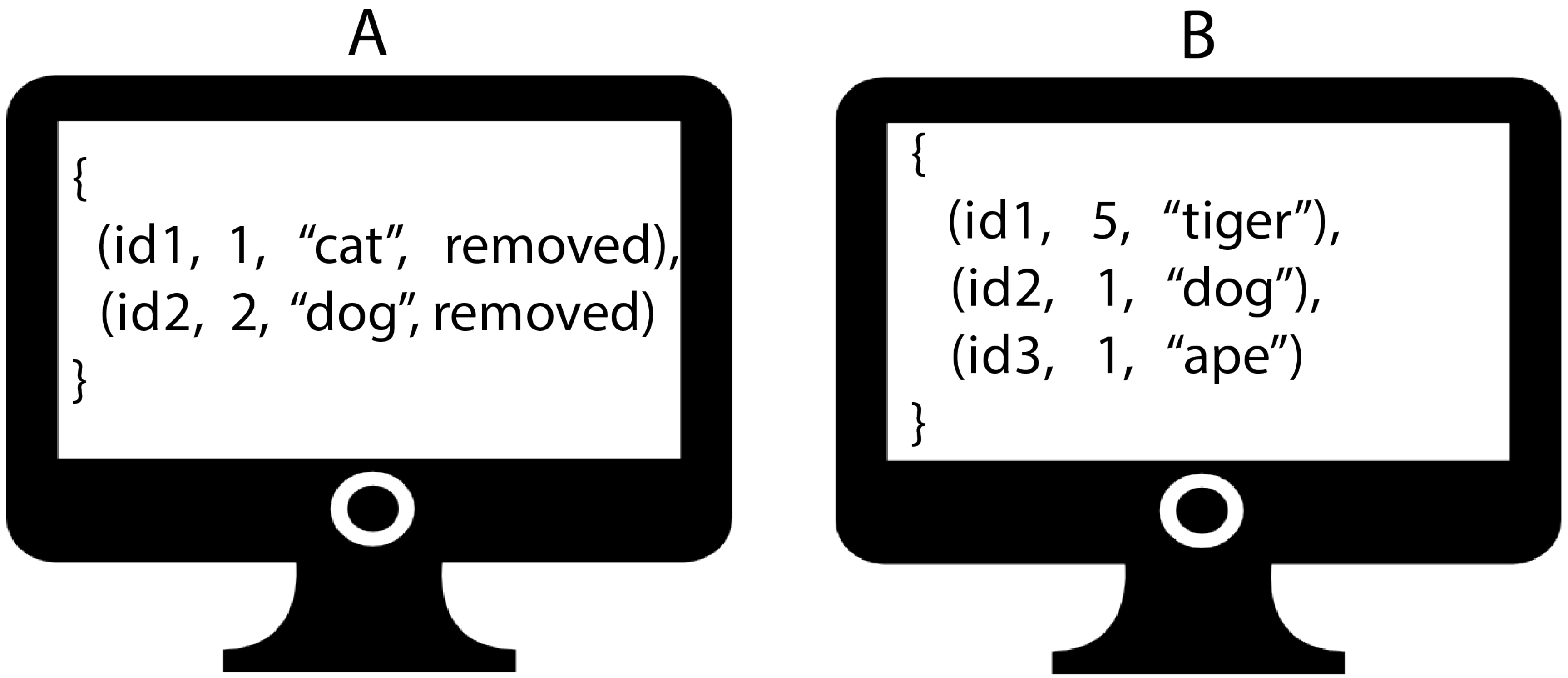
Contains a single underlying set of **elements with metadata**:

- Each element has a unique **id** field (e.g. a *UUID*)
- Each element has a **"removed"** boolean flag
- Each element has a **timestamp**
- Set can only contain one element with a *particular id*

OUR-Set


A
B
{
(id1, 1, “cat”, removed),
(id2, 2, “dog”, removed)
}
{
(id1, 5, “tiger”),
(id2, 1, “dog”),
(id3, 1, “ape”)
}

OUR-Set



Merge

{ (id1, 5, "tiger"), (id2, 2, "dog", removed), (id3, 1, "ape") }

OUR-Set

Merge:

{ (id1, 5, "tiger"), (id2, 2, "dog", removed), (id3, 1, "ape") }

Lookup

{ "tiger", "ape" }

OUR-Set

Merge

```
def merge(anotherSet: OURSet[E]]): OURSet[E] =
  OURSet[E]( elements ++ anotherSet.elements)
              .groupBy (_.id)
              .map     (group => group._2.maxBy(_.timestamp))
              .toSet)
```

Lookup

```
def lookup(ourSet: OURSet[E]): Set[E] =
   ourSet.filter (!_.removed)
         .map    (_.value)
```

Implementation

NavCloud CRDT Model: **Favorites**

CRDT Model: Favorites

FavoriteState element:

- **ID** (to uniquely identify a favorite)
- **Timestamp** (to indicate the last change time)
- **Removed** flag (to indicate if favorite has been removed)
- Favorite data: (**Name**, **Location**, ...)

Convergence in case of equal **timestamps**

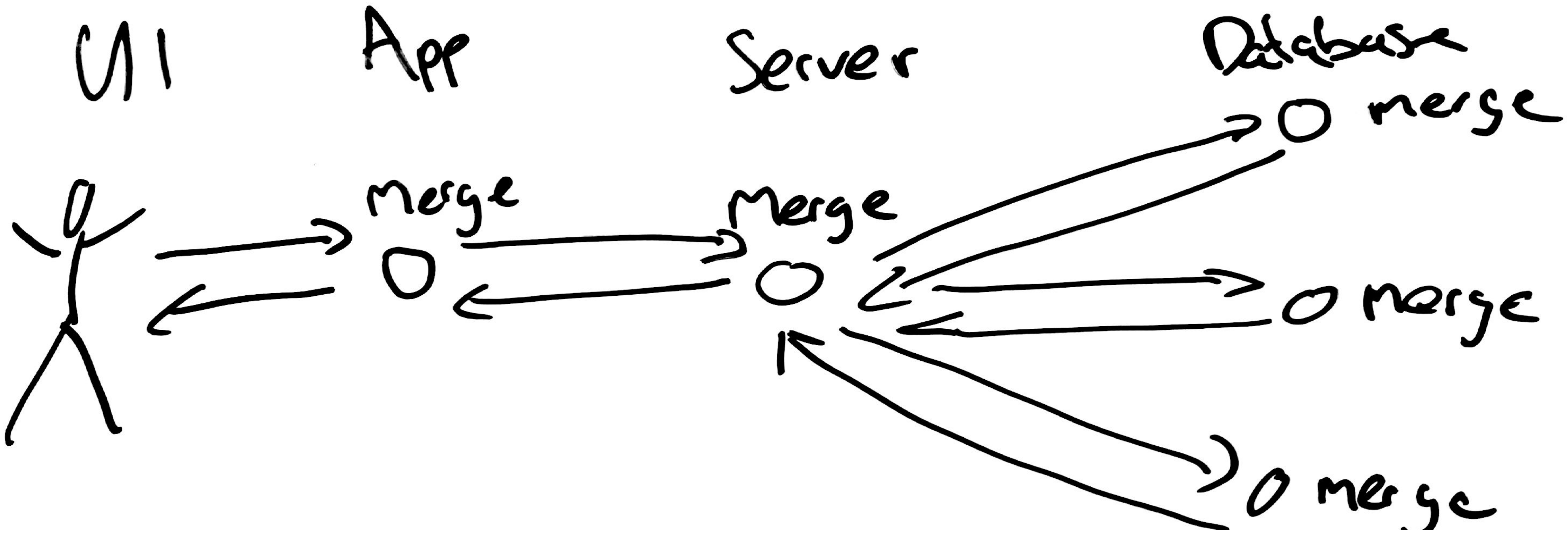
Compare function checks all the fields in order of priority:

- Timestamp
- Removed flag (*Add* or *Delete* bias)
- .. rest attributes ..

Using CRDT everywhere

- Use the same algorithm everywhere

As simple as calling the **merge** function

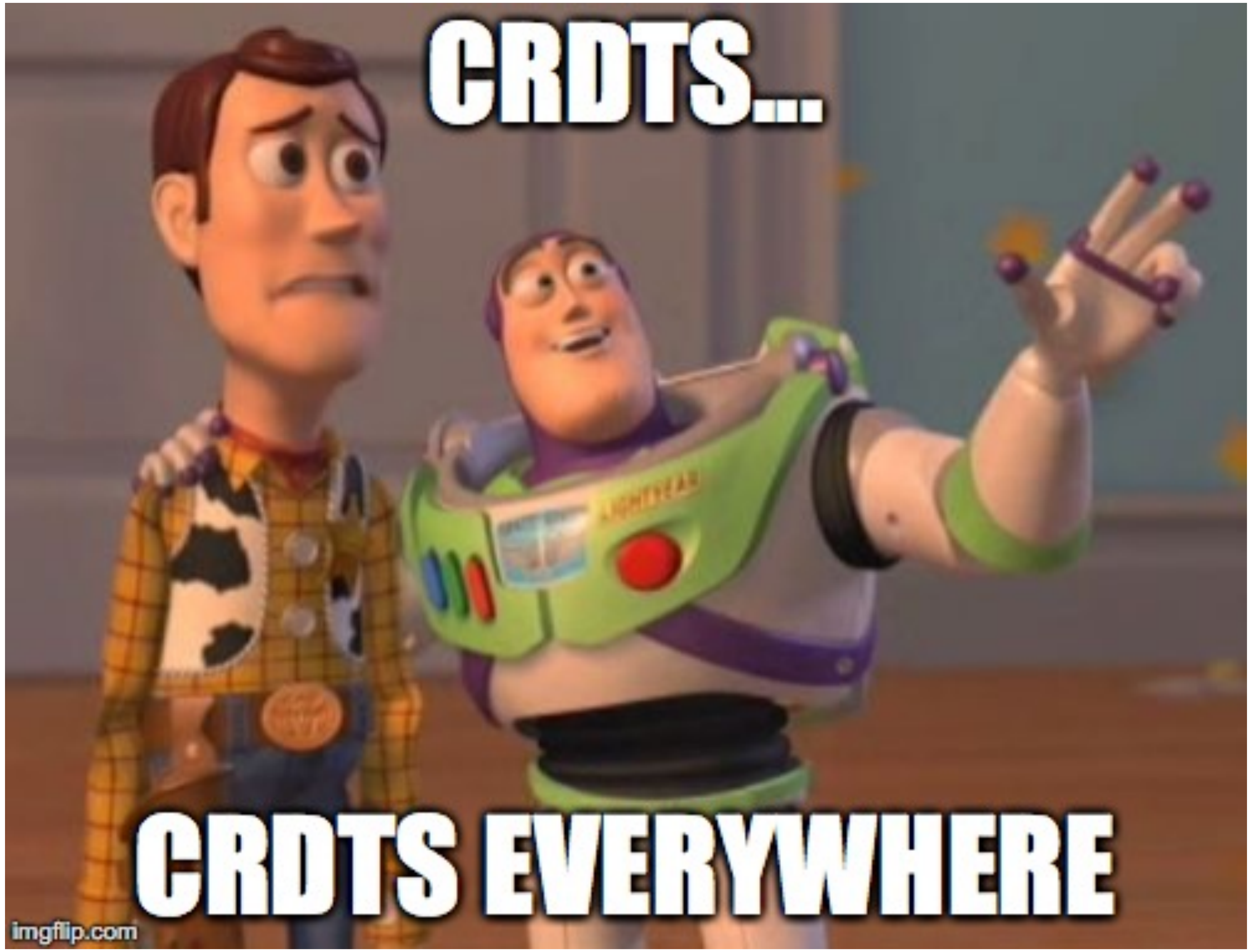


Using CRDT everywhere

Client <-> **Server** <-> Database

```
def update(fromClient: OURSet[E]): OURSet[E] = {
  val fromDatabase = database.fetch(...)
  val newSet = fromDatabase.merge(fromClient)
  database.store(..., newSet)

  newSet
}
```


CRDTS...
CRDTS EVERYWHERE
imgflip.com

Considerations & Limitations

"What about garbage?"

- CRDTs tend to grow because of **tombstones**.
- Deleted Element in the Set == *Tombstone*.
- A potentially **unbounded growth**.

Prune deleted elements

But **when**?

Requirement:
All **nodes** holding a CRDT Set replica should have seen a deleted element before it can be pruned.

Otherwise deleted elements can be **resurrected**.

Time-To-Live for *tombstones*

Prune tombstones once TTL exceeded.

```
if ((DateTime.now() - tombstone.timestamp) > TimeToLive) {
    crdtSet.remove(tombstone)
}
```

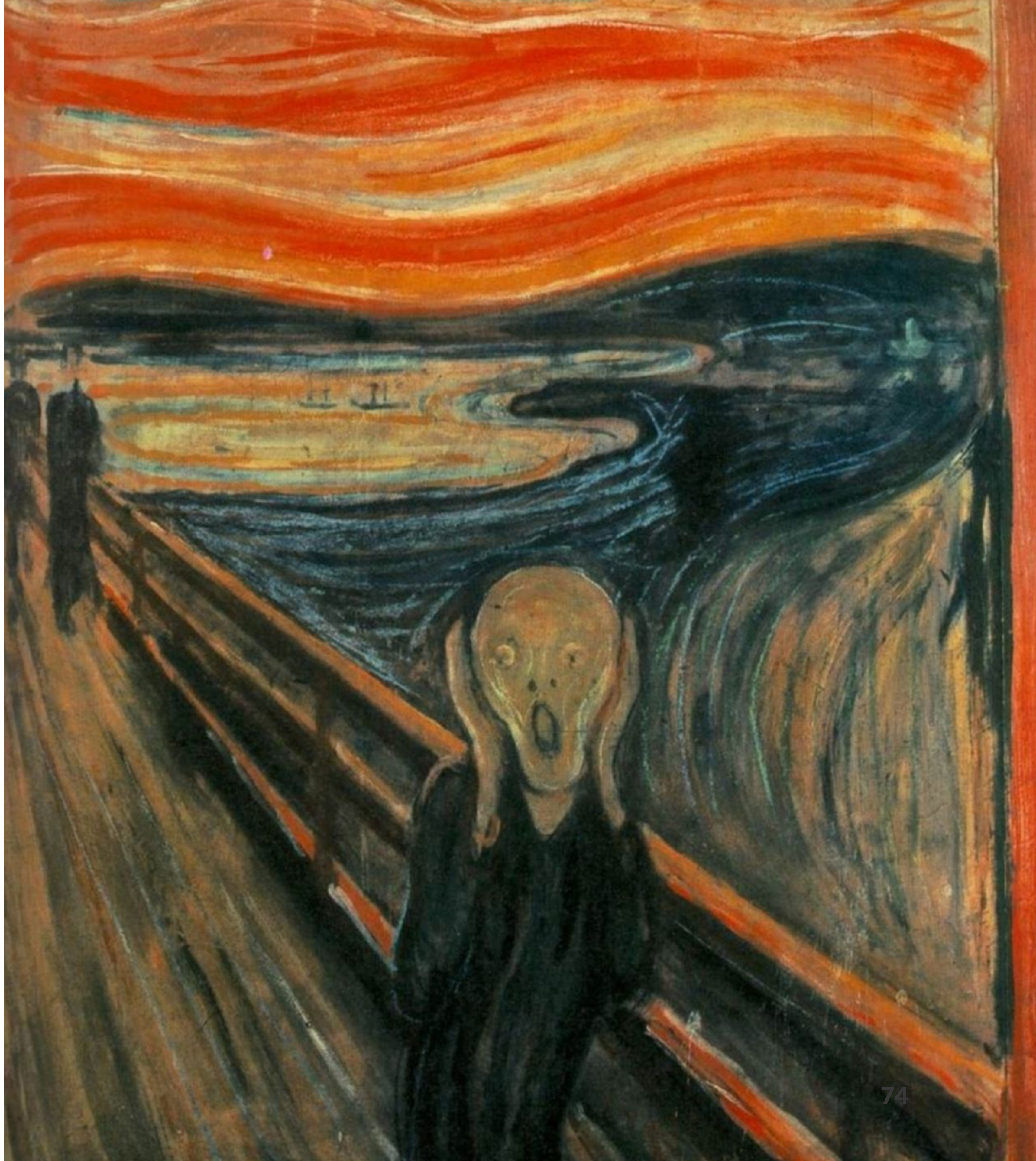
Requirement: all **nodes** holding a CRDT set should apply the same TTL rule **independently**.

Prune deleted elements

Problem

Synchronization between all replicas is needed *for correctness*.

Distributed transactions



- Academia, help!

INRIA
ROCQUENCOURT

An Optimized Conflict-free Replicated Set *

Annette Bieniusa, INRIA & UPMC, Paris, France
Marek Zawirski, INRIA & UPMC, Paris, France
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal
Marc Shapiro, INRIA & LIP6, Paris, France
Carlos Baquero, HASLab, INESC TEC & Universidade do Minho, Portugal
Valter Balegas, CITI, Universidade Nova de Lisboa, Portugal
Sérgio Duarte CITI, Universidade Nova de Lisboa, Portugal

Thème COM — Systèmes communicants
Projet Regal

Rapport de recherche n° 8083 — Octobre 2012 — 9 pages

Abstract: Eventual consistency of replicated data supports concurrent updates, reduces latency and improves fault tolerance, but forgoes strong consistency. Accordingly, several cloud computing platforms implement eventually-consistent data types.

The set is a widespread and useful abstraction, and many replicated set designs have been proposed. We present a reasoning abstraction, *permutation equivalence*, that systematizes the characterization of the expected concurrency semantics of concurrent types. Under this framework we present one of the existing conflict-free replicated data types, Observed-Remove Set.

Furthermore, in order to decrease the size of meta-data, we propose a new optimization to avoid tombstones. This approach that can be transposed to other data types, such as maps, graphs or sequences.

Key-words: Data replication, optimistic replication, commutative operations

Optimized OR-Set

Introduces *replica awareness*

Optimized OR-Set

Additional metadata is added to every transferred state.

```
{ (replica_id -> seq_nr) }
```

where:
- *replica_id* - is a unique & stable replica identifier.
- *seq_nr* - monotonically growing (after each op) local counter.

Optimized OR-Set

Each local state maintains a map:

```
{ replica_A: 1, replica_B: 1, replica_C: 3 }
```

If a received state has a *seq_nr* lower than the corresponding local value -> ignore.

TOMTOM

Optimized OR-Set

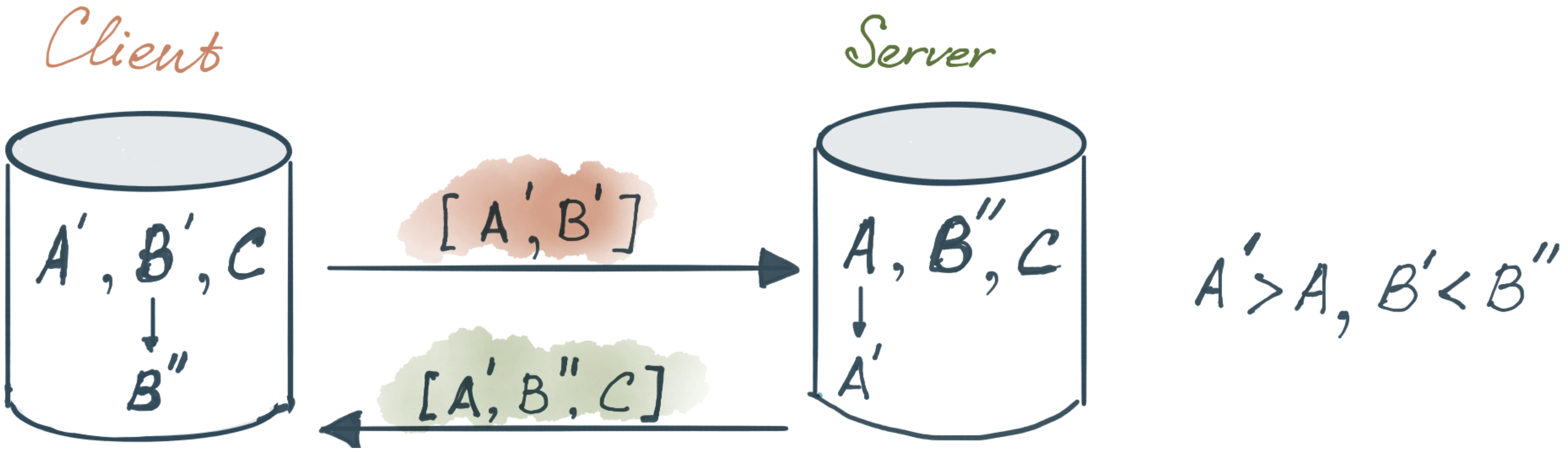
No *Tombstones*, yay! ☺️

(Slightly) more complicated API: stable *replica_id* needed. ☹️

Update & Reply with a Diff

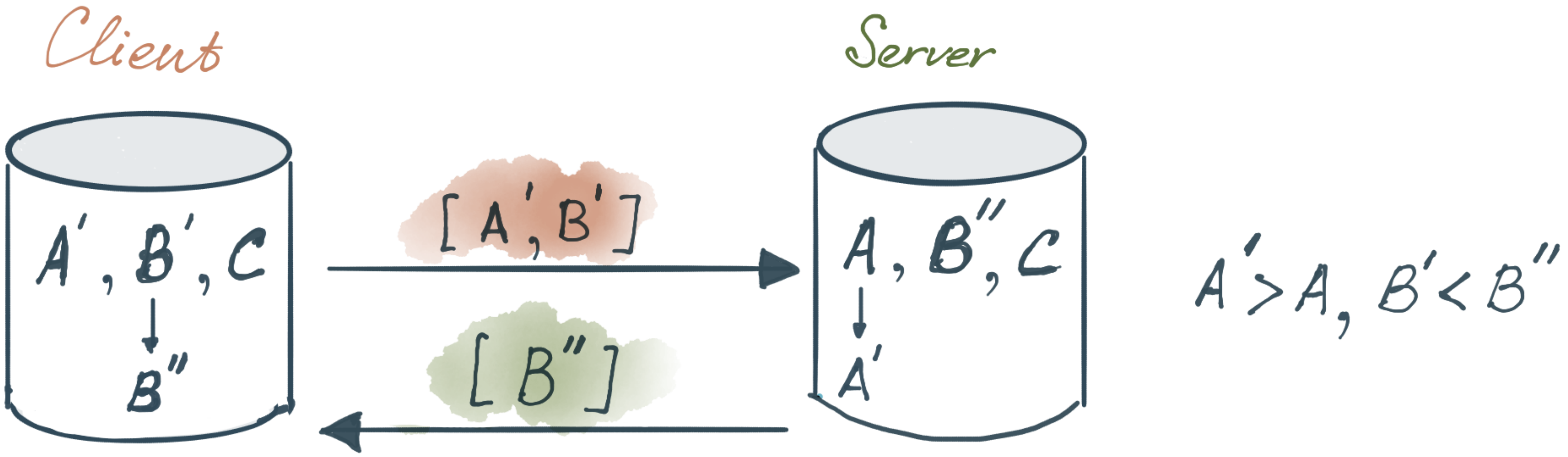
Client modifies and sends only updated elements (**Diff**).

Before: Server responds with a full merge result.

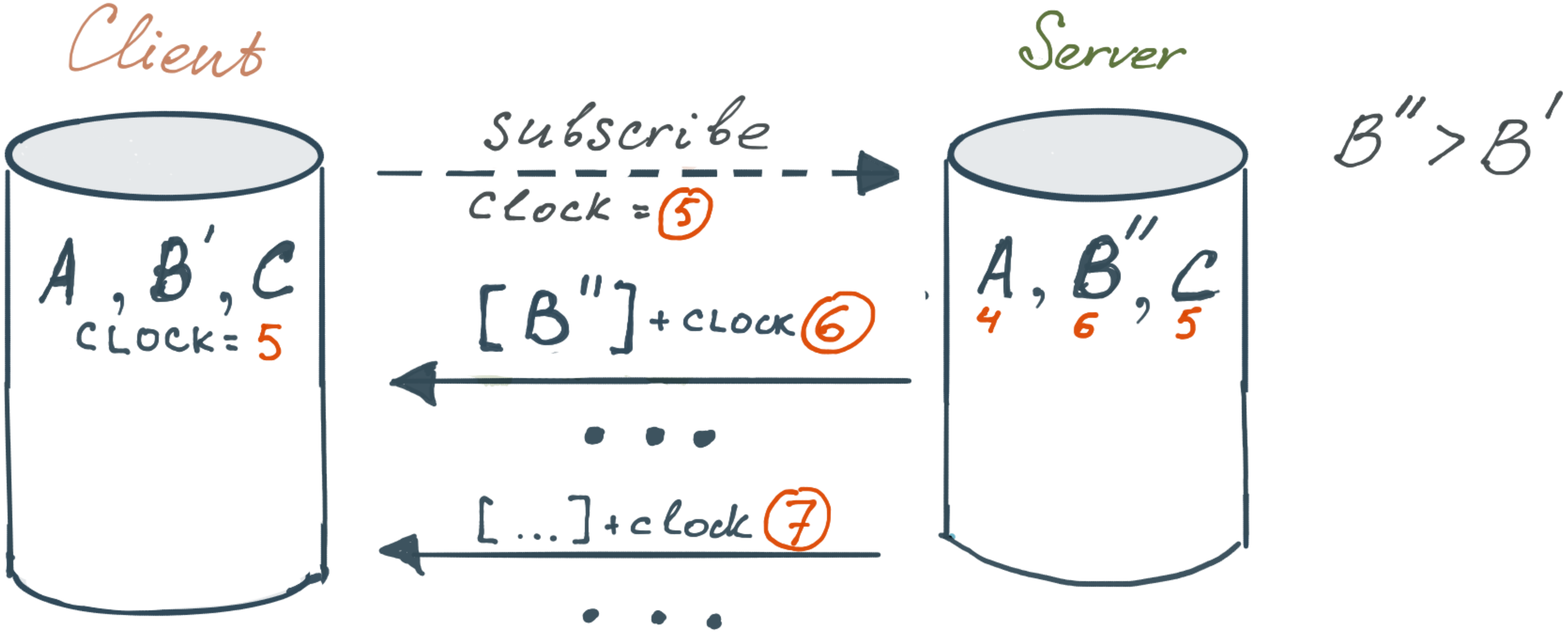


Update & Reply with a Diff

We introduced a '**Scoped Diff**':
Server responds only with the elements which have won against those sent by the client.



Server -> Client Diff



- Academia, help?..

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.

δ-CRDT

Builds on *replica awareness*

Introduces a **Causal Context**:
map of `(replica_id -> seq_nr)`.

Introduces a **Dot Store**: CRDT state (No tombstones).

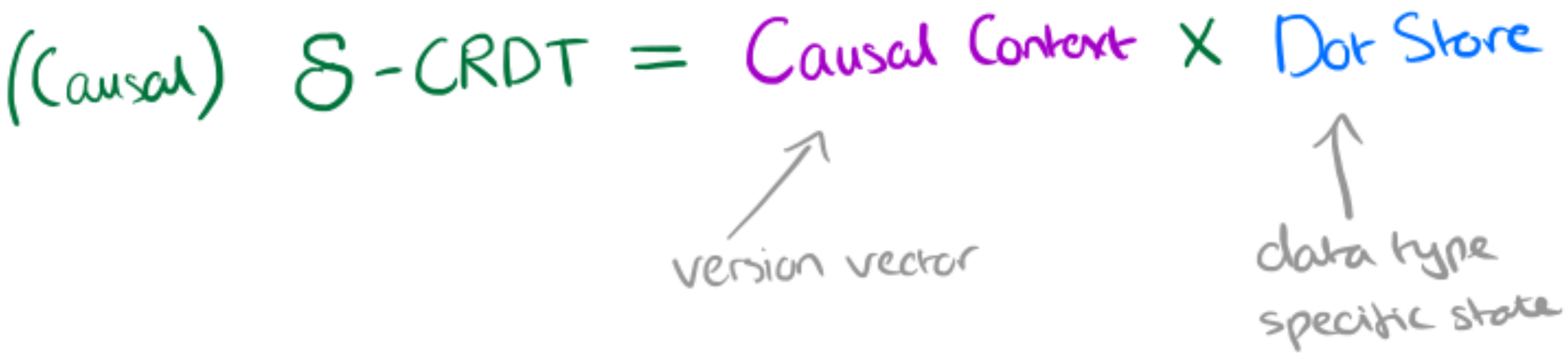
δ-CRDT

A formalized way to compute a minimal **δ-CRDT** instances against a target replica.

δ-CRDT

Adrian Colyer (The Morning Paper) wrote a great paper review:

blog.acolyer.org/2016/04/25/delta-state-replicated-data-types



Trouble With Time

There is no such thing as **reliable time***.

Tracking time is actually tracking causality.

— Jonas Bonér, "Life Beyond the Illusion of Present"

Causality & **Ordering** of events.

Time can be just **good enough**.

Ordering updates within a **single node**

Timestamp field as a **logical clock**.

Absolute value is not important,
but it should always **grow monotonically**.

Ordering updates within a **single node**

"+1 Strategy" (aka ensure monotonicity):

```
Long resolveNewTimestamp(ElementState<E> state) {
    return Math.max( retrieveTimestamp(),
                     state.lastModified() + 1 );
}
```

Ordering updates from **different** nodes

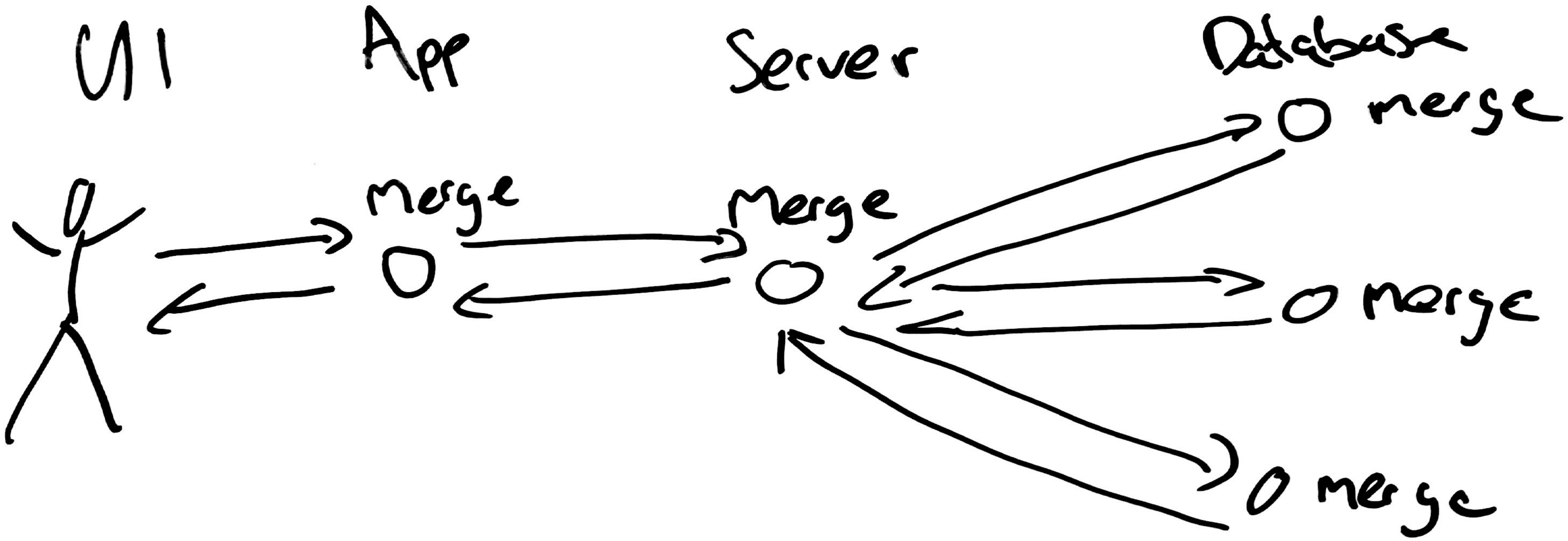
If GPS clock is available -> use it (mainly **Navigation Devices** case).

Prefer the **server time** to a client's local time.

Edge case

Multiple Clients modify the **same element** (concurrently || without a reliable clock).

One "merge" to rule them all



Clients & Server *MUST* have **same 'merge' behaviour**.

==

Given the **same input**, their '**merge**' functions emit the **same results**.

Divergence may lead to endless synchronization loops!

Lazy (data) loading

OURSet Element

- **Metadata**: *UUID, timestamp, "removed" flag*
- **Data**: <Value>

Lazy (data) loading

New OURSet Element

- **Metadata**: *UUID*, *timestamp*, *"removed" flag*, **+** *tag / hash*
- (Optional) **Data**: <Value>

Flexible synchronization strategy

Eager || **Lazy** Fetch

What have we learned?

- Academia is *not* as *scary* as it sometimes seems to *pragmatic devs*.
- We need better and simpler abstractions to develop

Offline-friendly apps.

- CRDTs give a great value, but there are some *caveats*.
- Things like *Lasp* (lasp-lang.org) also could be the answer (?).

Show me the code

github.com/ajantis/{scala | java}-crdt

Thanks!

Nami Nasserazad @namiazad

Didier Liauw

Slides: http://bit.ly/2fBlroS

Dmitry Ivanov @idajantis