Programming and *proving* distributed systems with *persistent data structures*

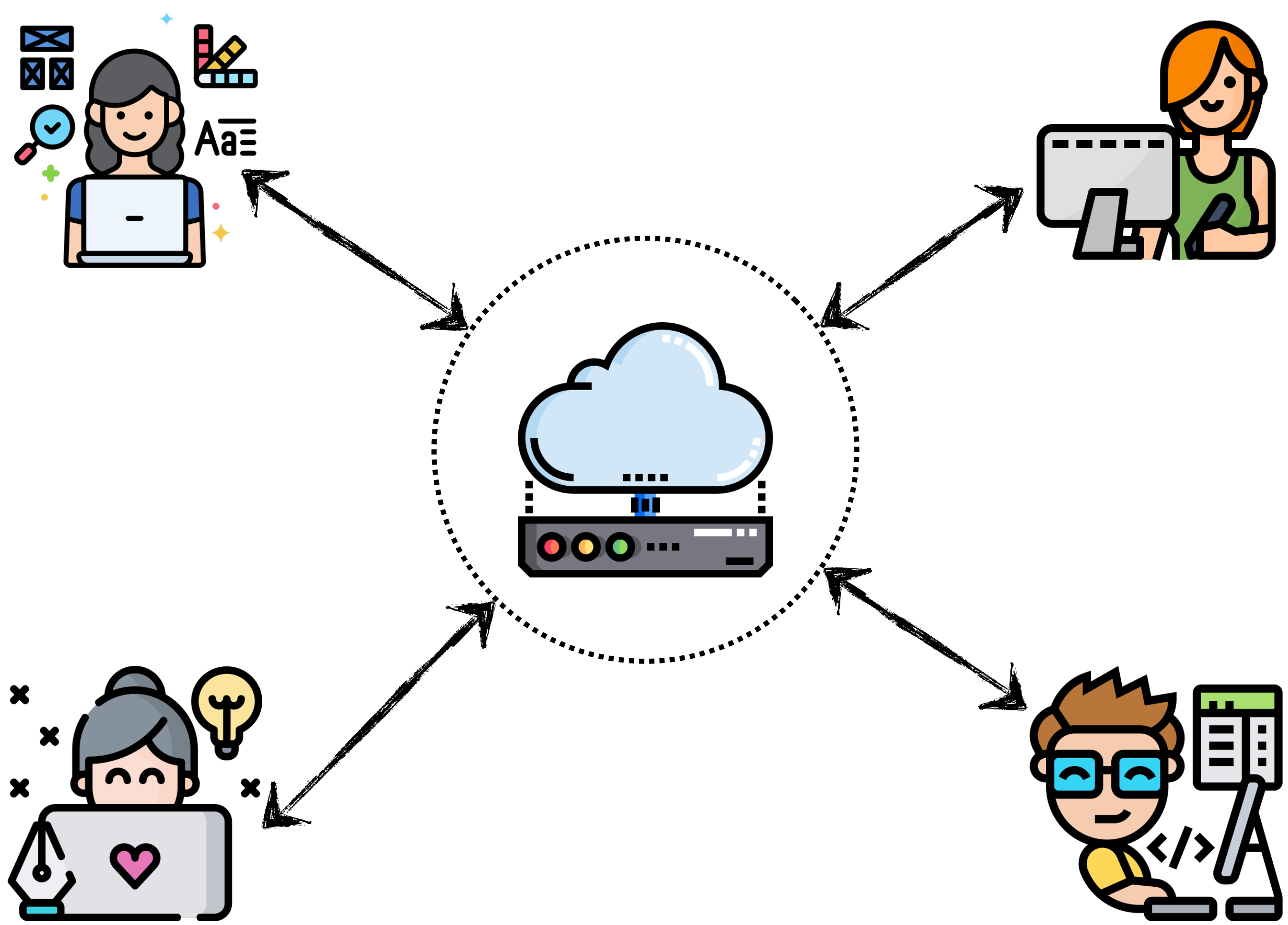
“KC” Sivaramakrishnan

IIT MADRAS

Tarides

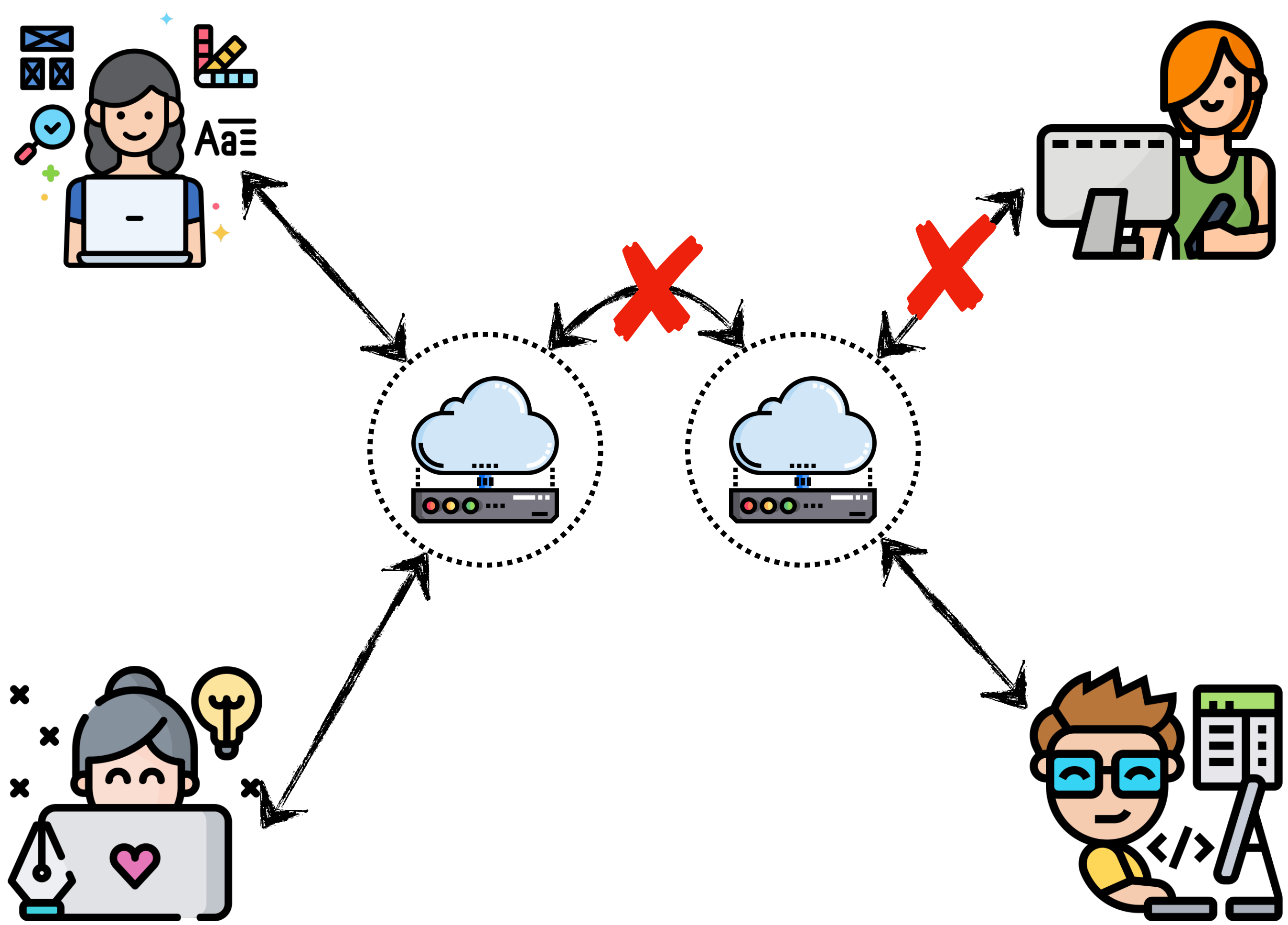
Collaborative Apps

Airtable Google Docs Figma Notion Overleaf

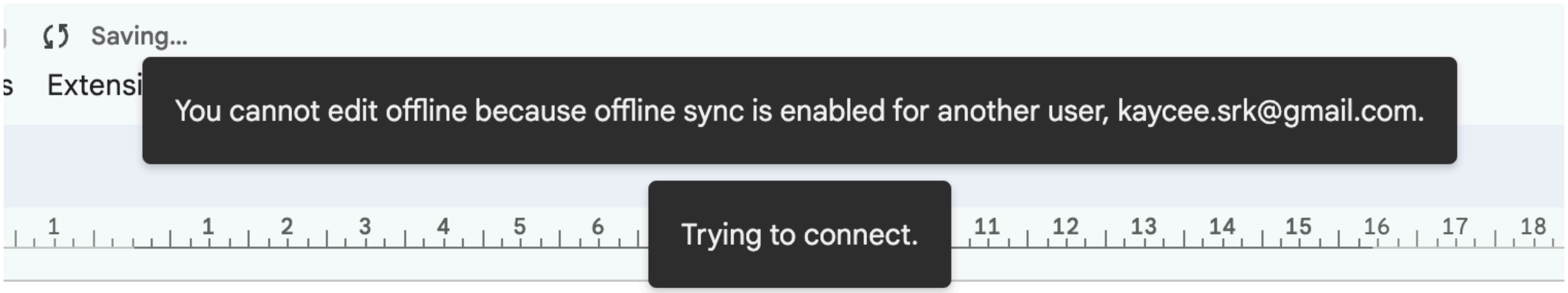


Network Partitions

Airtable Google Docs Figma Notion Overleaf



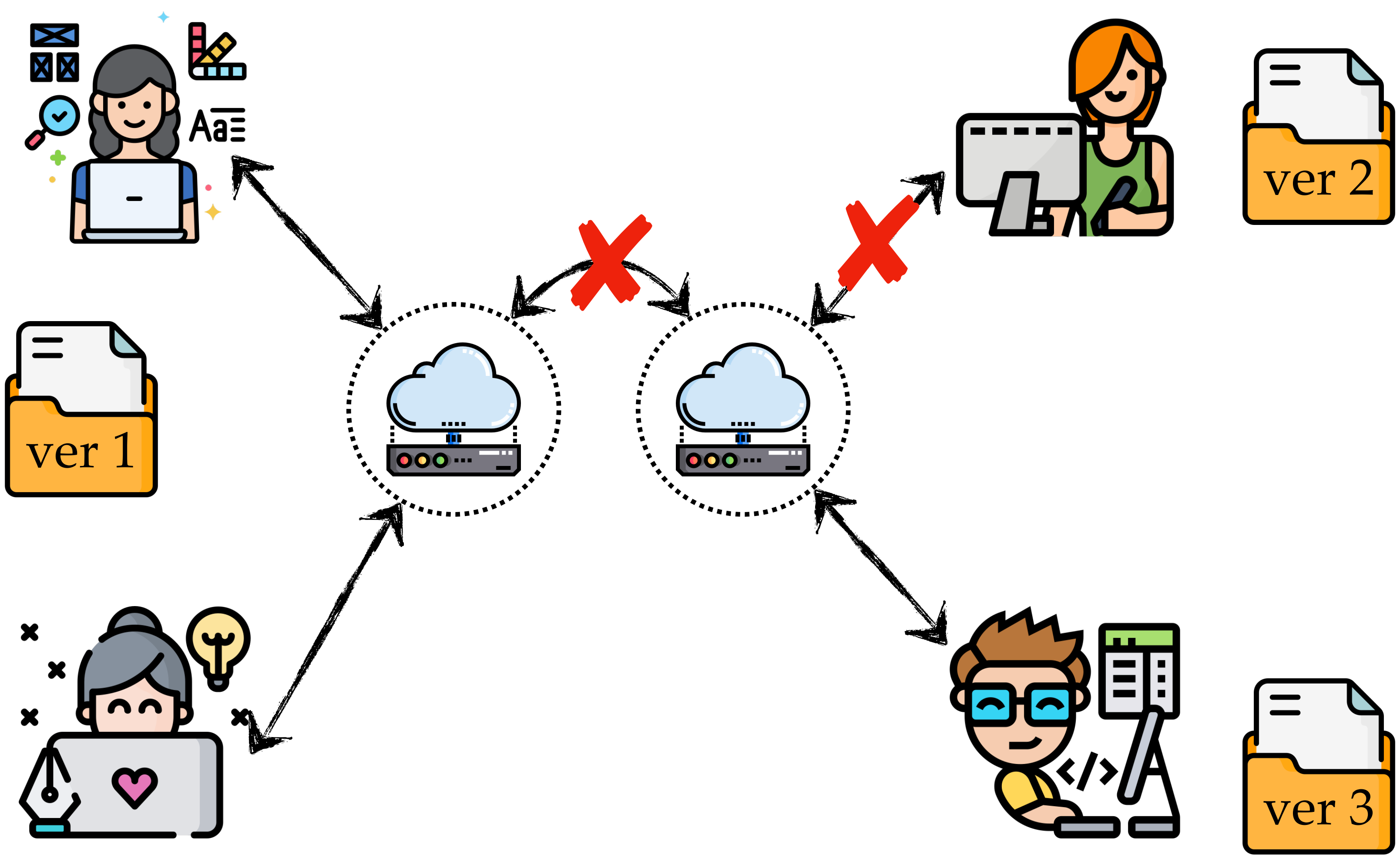
Network Partitions — Google docs



Enabling offline sync for one account prevents other accounts from working offline

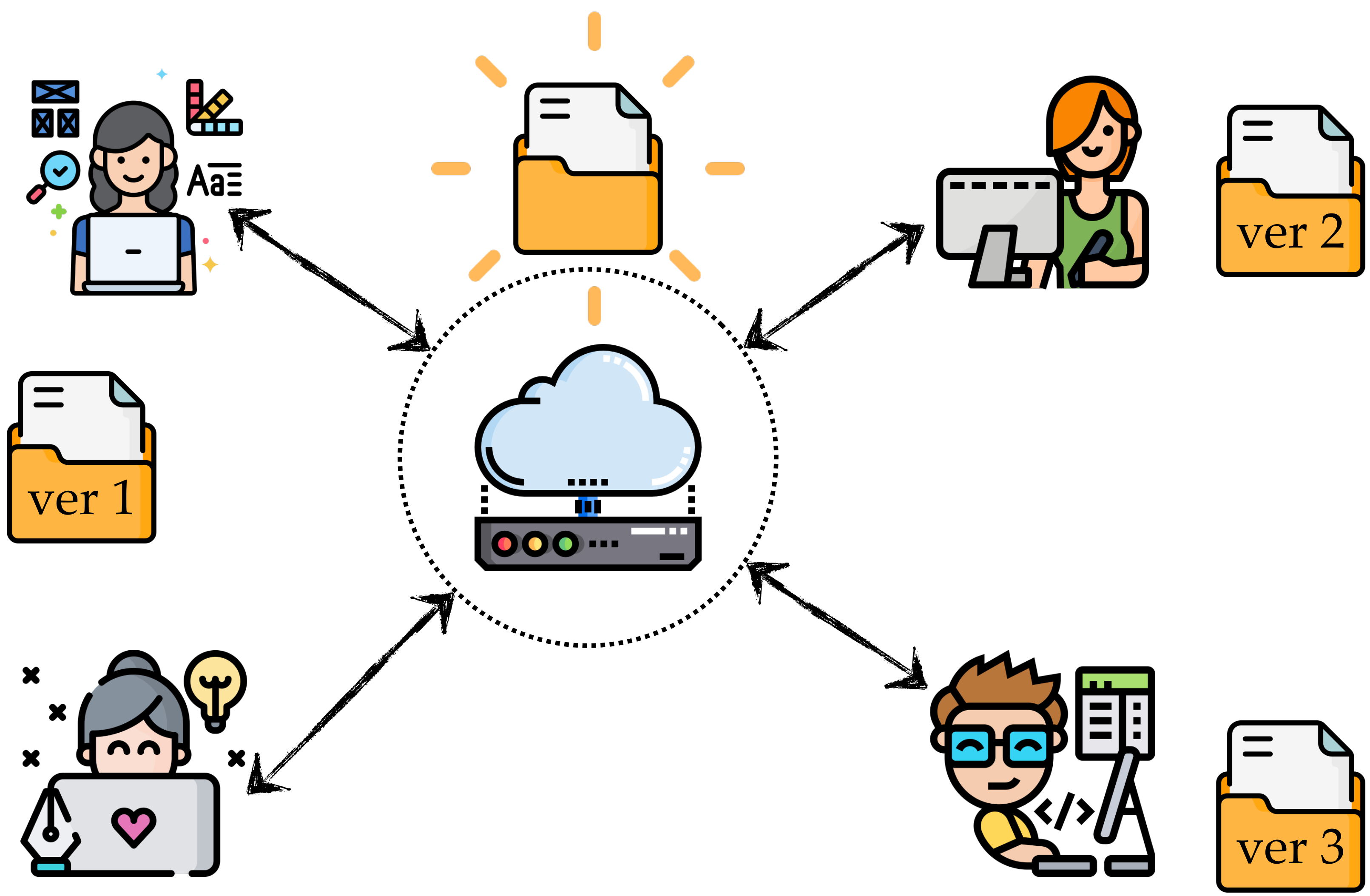
Local-first software

Airtable Google Docs Figma Notion Overleaf



Local-first software

Airtable Google Docs Figma Notion Overleaf



How do we build such applications?

Make data types aware of replication

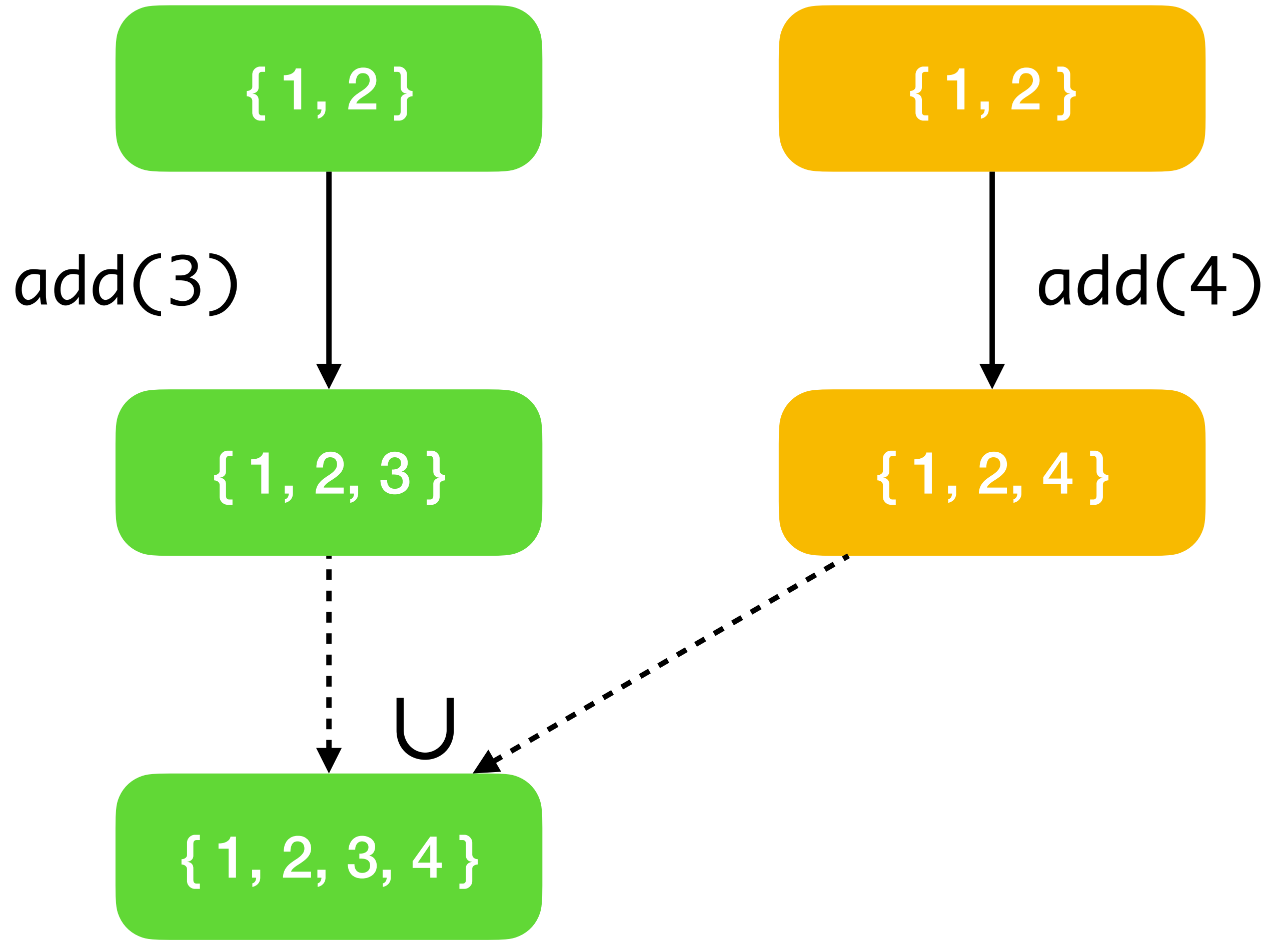
CRDTs

- Conflict-free Replicated Data Types (CRDTs)
  - Multiple *replicas* of the data types
- Supports *local* operations
- Share updates *asynchronously* and ensure *convergence*
  - *Strong* eventual consistency

[1] Nuno Preguica et al. "Confict-free Replicated Data Types", arXiv:1805.06358

Grow-only Set

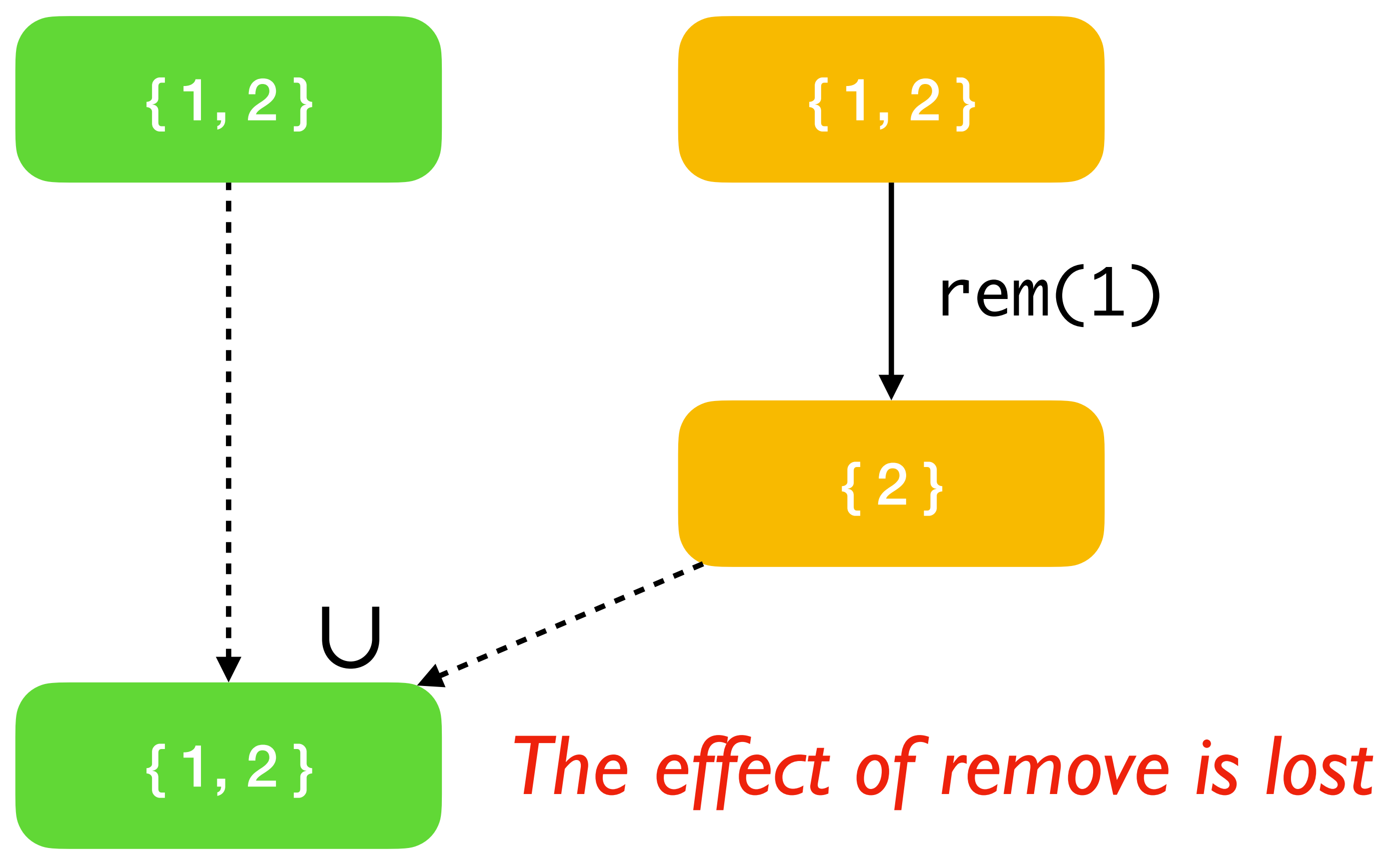
- Supports *add* and *lookup* operations



A set with only *add* and *lookup* is *monotonic*

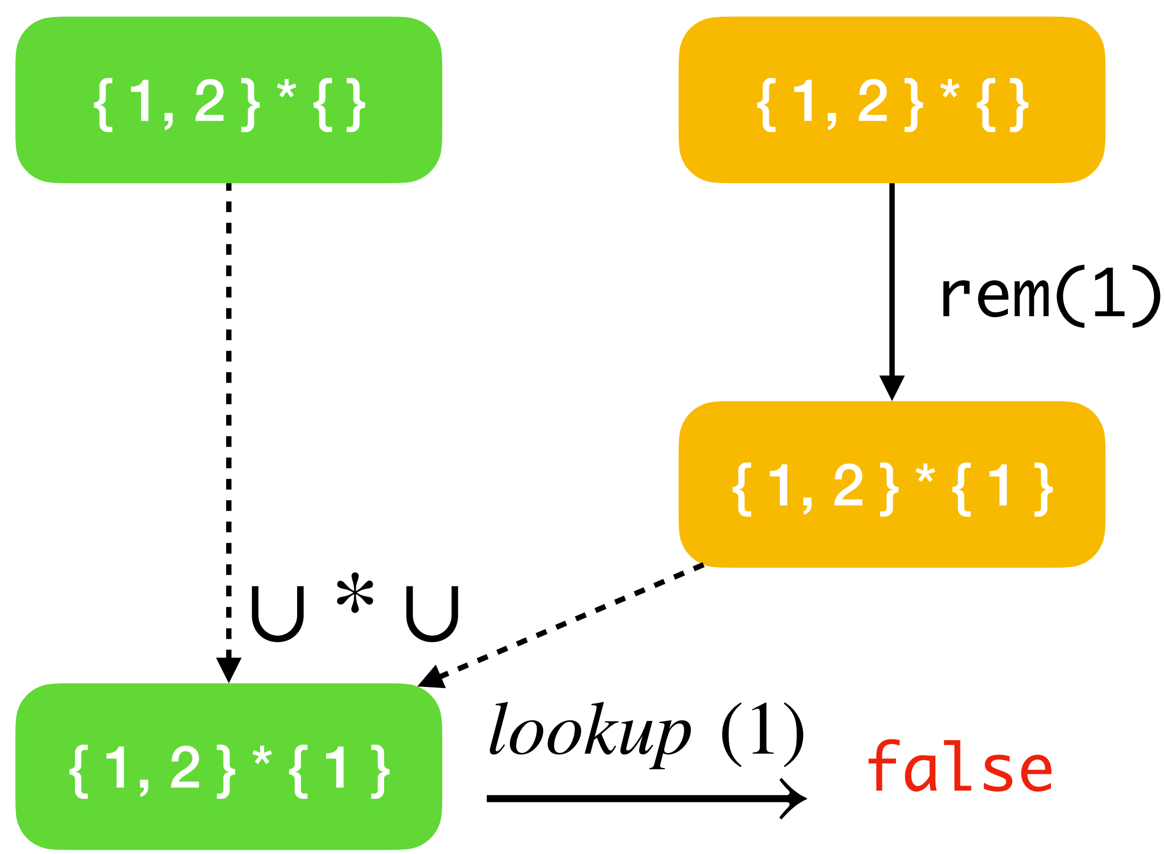
Include remove operation

- Let's include *remove* operation
  - ✦ No longer monotonic

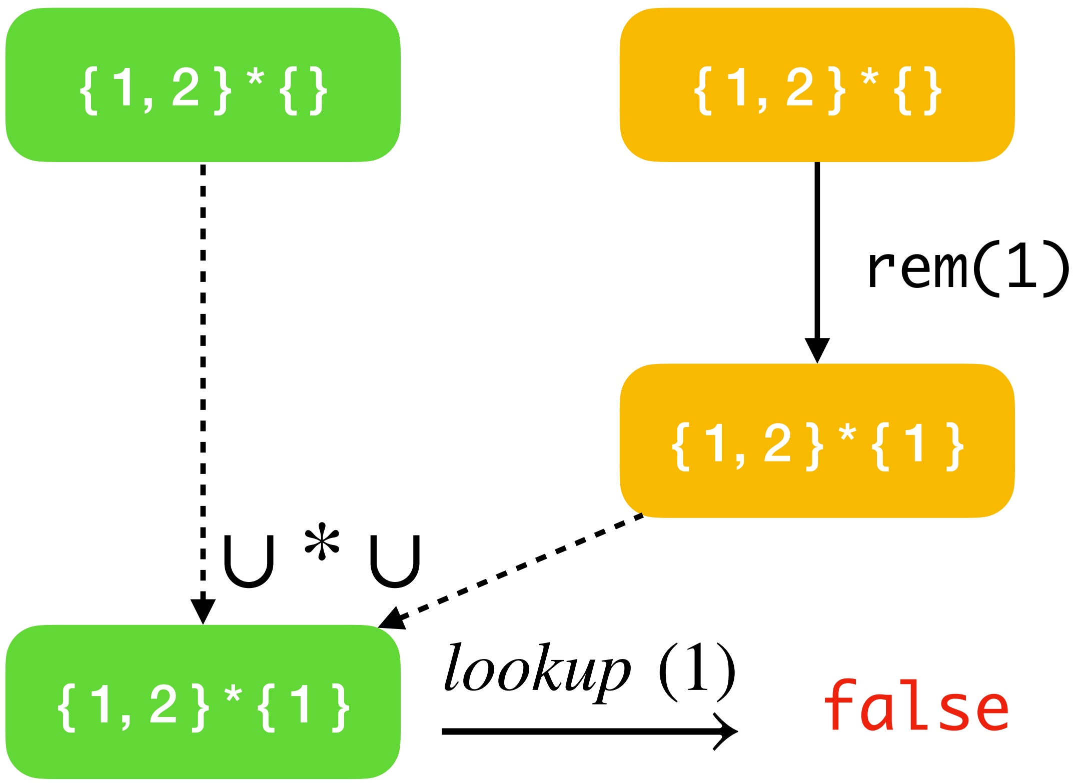


Two-phase set

- Represent the set with a pair of sets to track *additions* and *removals* — A * R
  - ✦ Lookup is performed in A / R
  - ✦ Merge is pair-wise union of A and R sets



Two-phase set — Observations



- Monotonic — Simulate remove with adds
- Remove-wins semantics
- *Reengineer the set implementation*
- Tombstones — *elements removed by adding to R set!*
- *Removed elements gone forever*

Challenges with CRDTs

- Monotonicity forces *reengineering of data structures from scratch*
  - Challenges proving correctness of *even sequential operations*
- Not space and time-efficient
  - *Tombstones* affect time- and space-efficiency
- Little attention has been paid to *composition* of RDTs
  - Parametric polymorphism for RDTs
    - Like to compose `'a set` with a `counter` to get `counter set`
  - *How to compose proofs of correctness?*

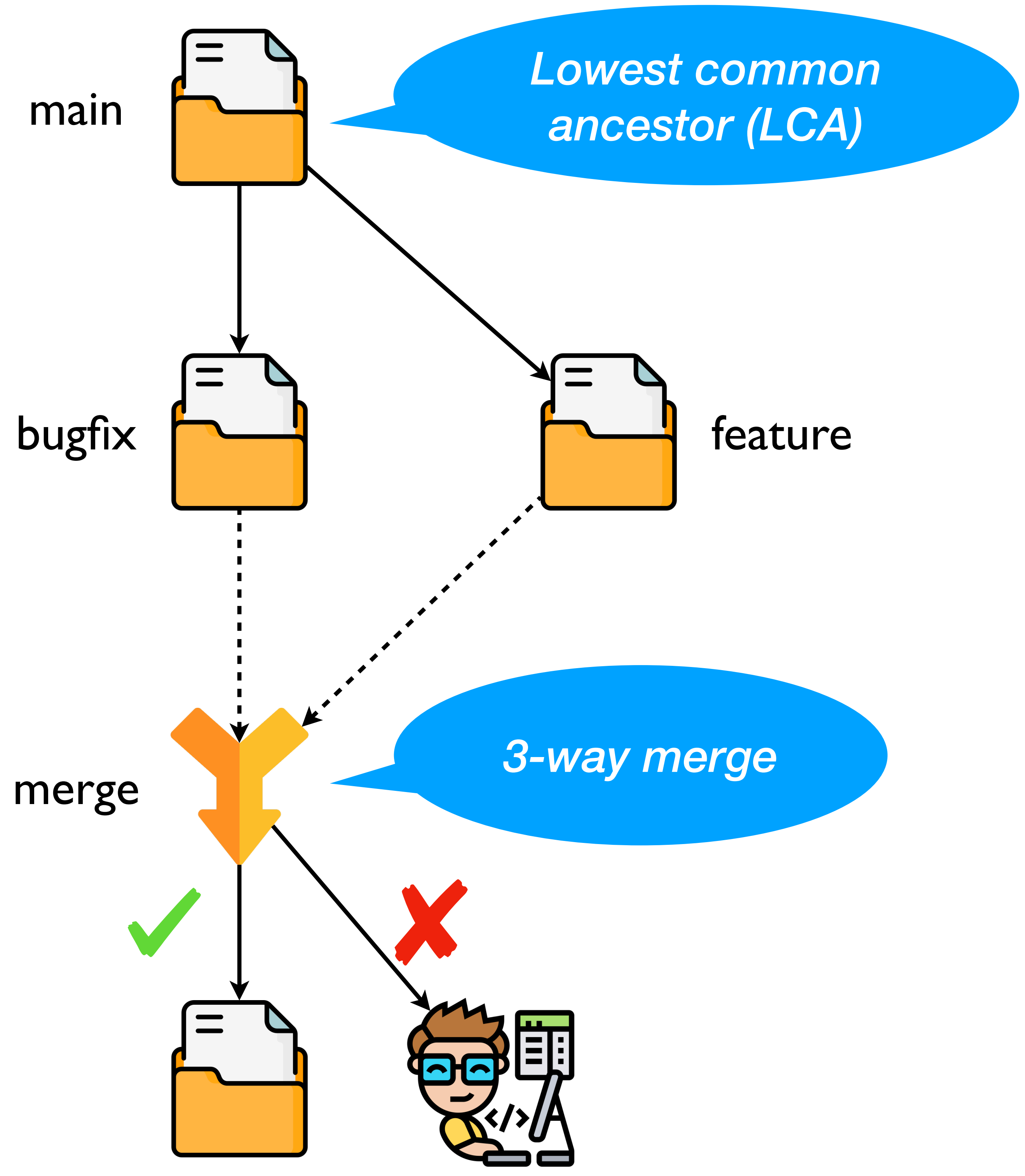
Can we do better?

Sequential data types

+

Git

Distributed Version Control Systems

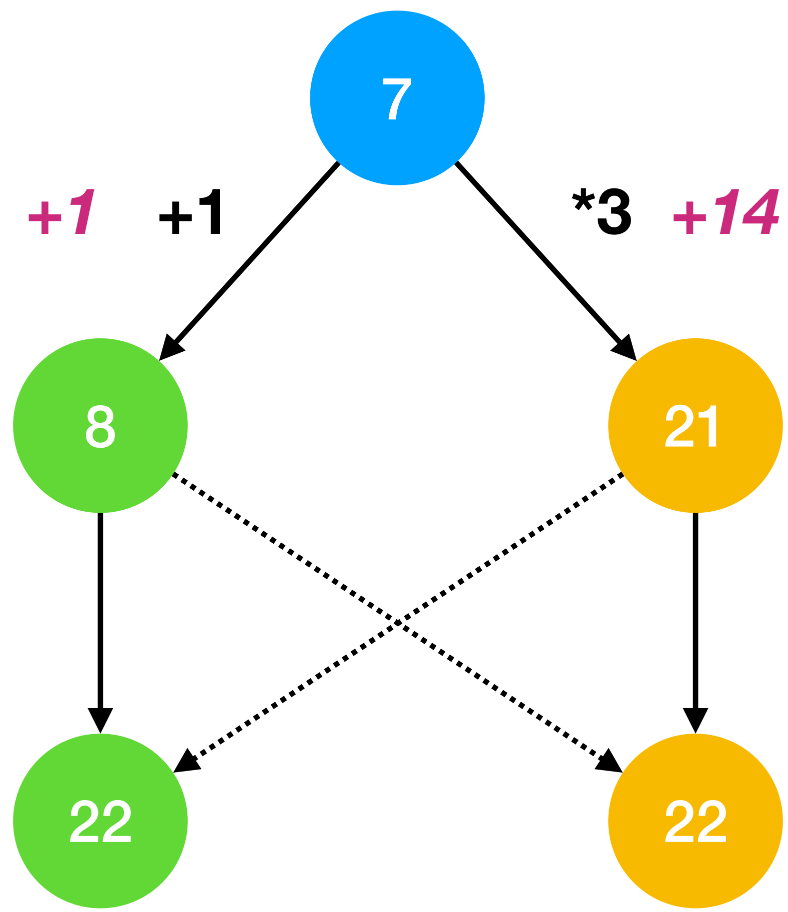


Mergeable Replicated Data Types

- MRDTs — DVCS for *data types* rather than just *text files*
    - *Branches* are *replica states*
- Sequential data type + 3-way merge = replicated data type!

Counter MRDT

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end
```



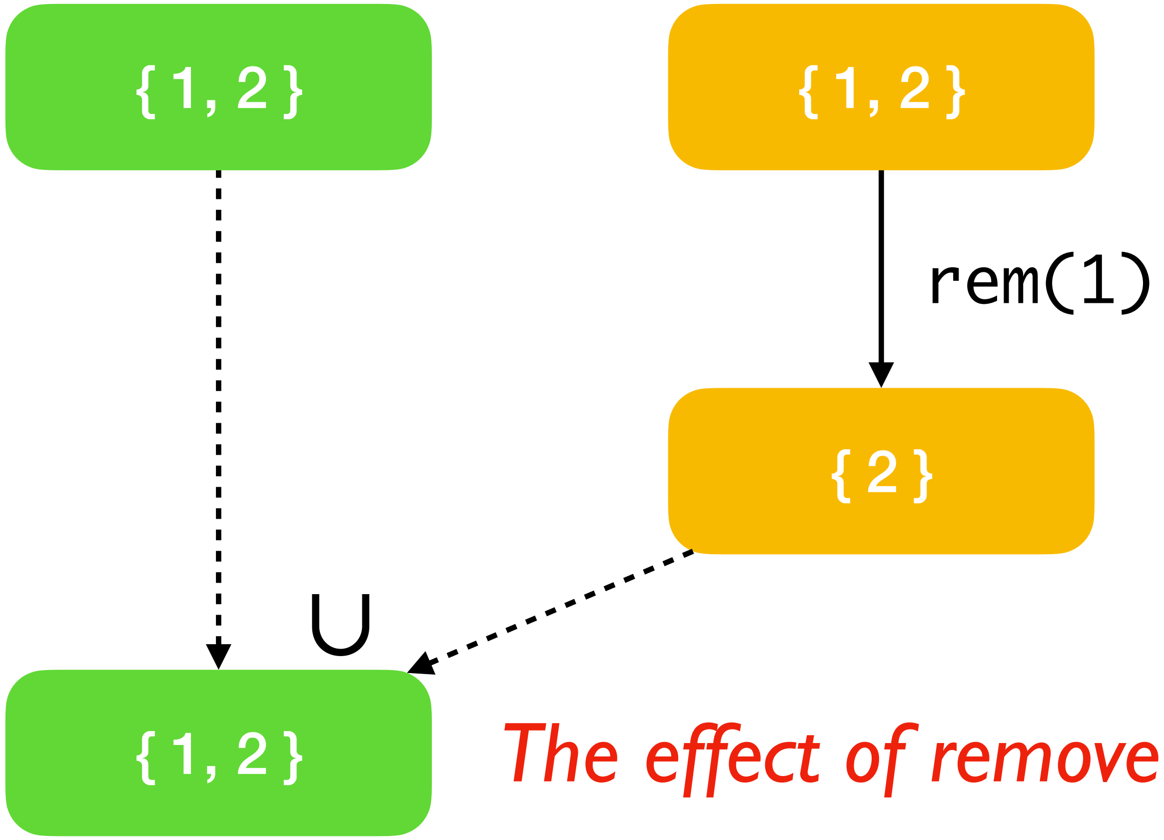
22 = 7 + (8-7) + (21 -7)

+1 *+14*

Set MRDT

```
let merge ~lca ~v1 ~v2 =
  (lca ∩ v1 ∩ v2) (* common elements *)
  ∪ (v1 - lca)    (* added in v1 *)
  ∪ (v2 - lca)    (* added in v2 *)
```

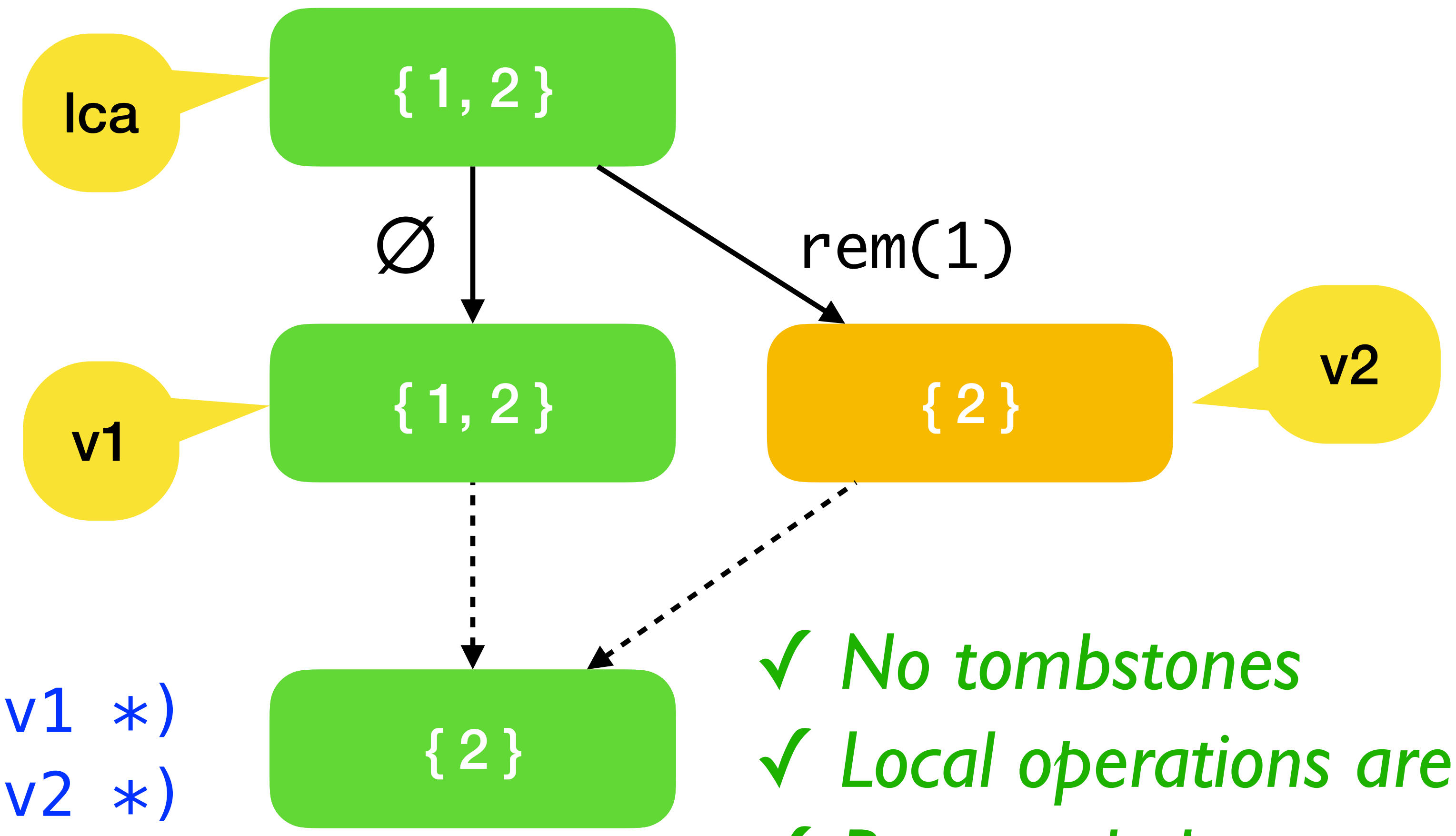
isomorphic to counter merge if you squint



The effect of remove is lost

Set MRDT

```
let merge ~lca ~v1 ~v2 =
  (lca ∩ v1 ∩ v2) (* common elements *)
  ∪ (v1 - lca)     (* added in v1 *)
  ∪ (v2 - lca)     (* added in v2 *)
```



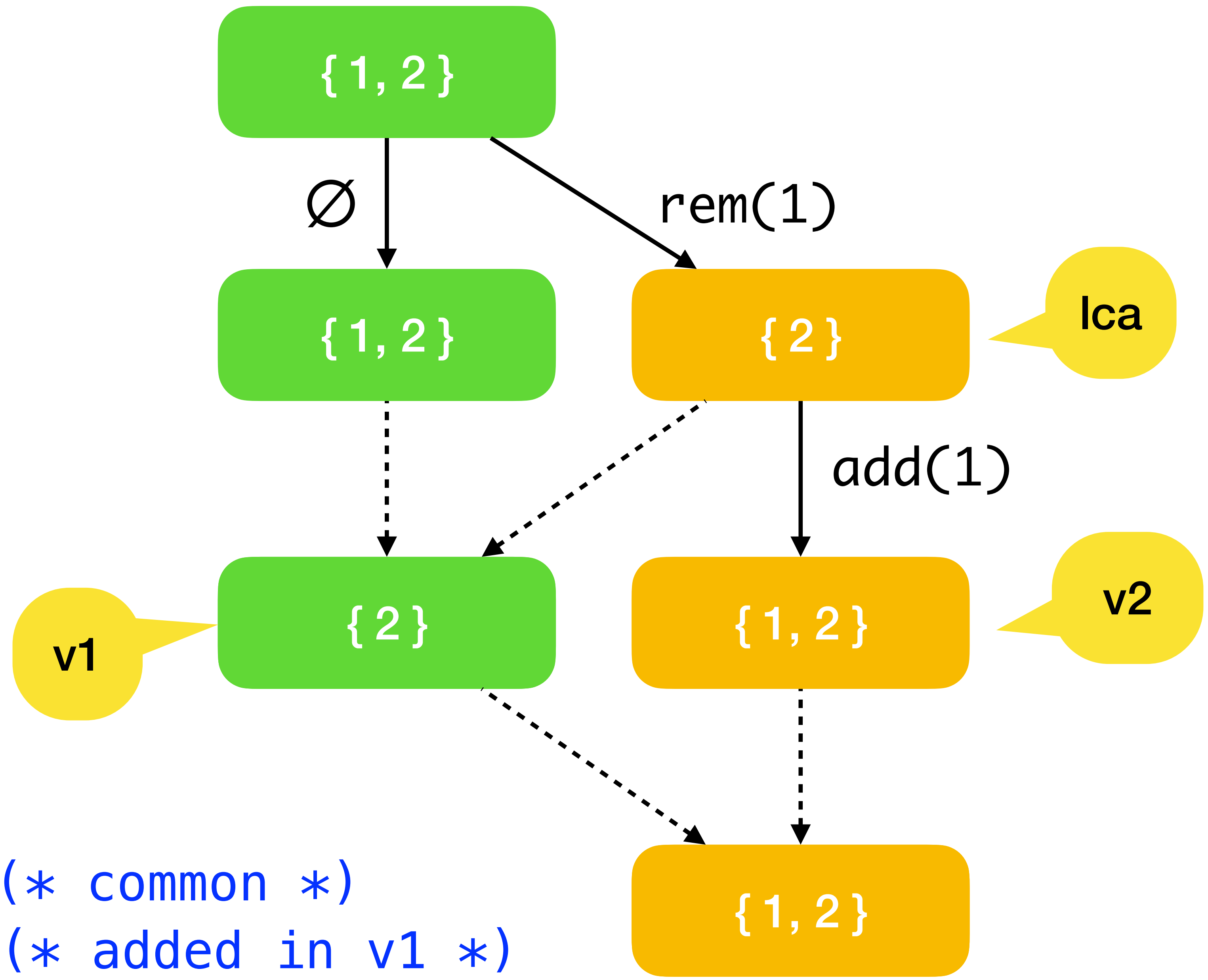

```
  {2} (* common *)
∪ {}  (* added in v1 *)
∪ {}  (* added in v2 *)
```

✓ No tombstones

✓ Local operations are efficient

✓ Removed elements can be added back

Set MRDT — Add after Remove



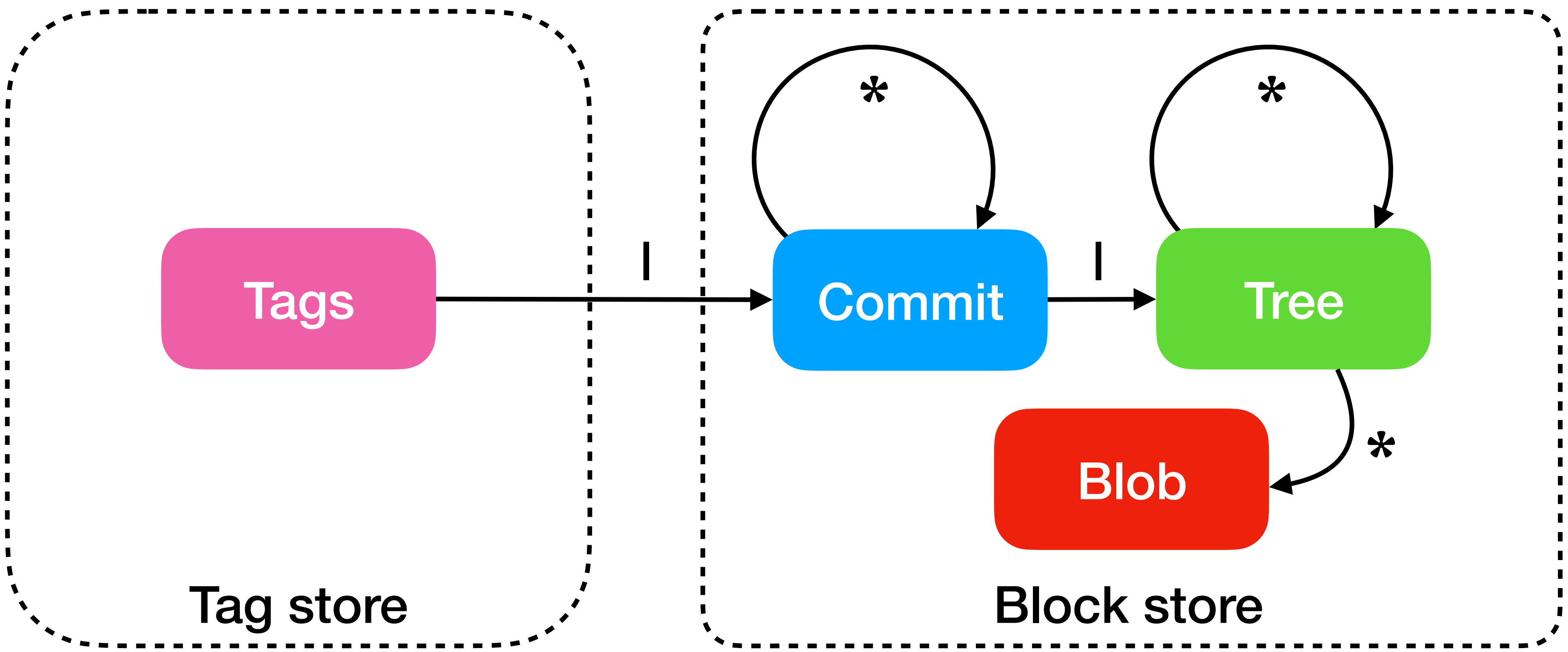

```
  {2} (* common *)
U {}  (* added in v1 *)
U {1} (* added in v2 *)
```

MRDTs and Causal History

- How did we get away with no tombstones in MRDT set?
- Tombstones in CRDTs record *history*
- *Git records the causal history in MRDTs!*
    - ✦ Presented via LCA in 3-way merge
- How does Git keep track of causal history efficiently?

Persistent Data Structures

Git store



- Branches / tags
- Mutable

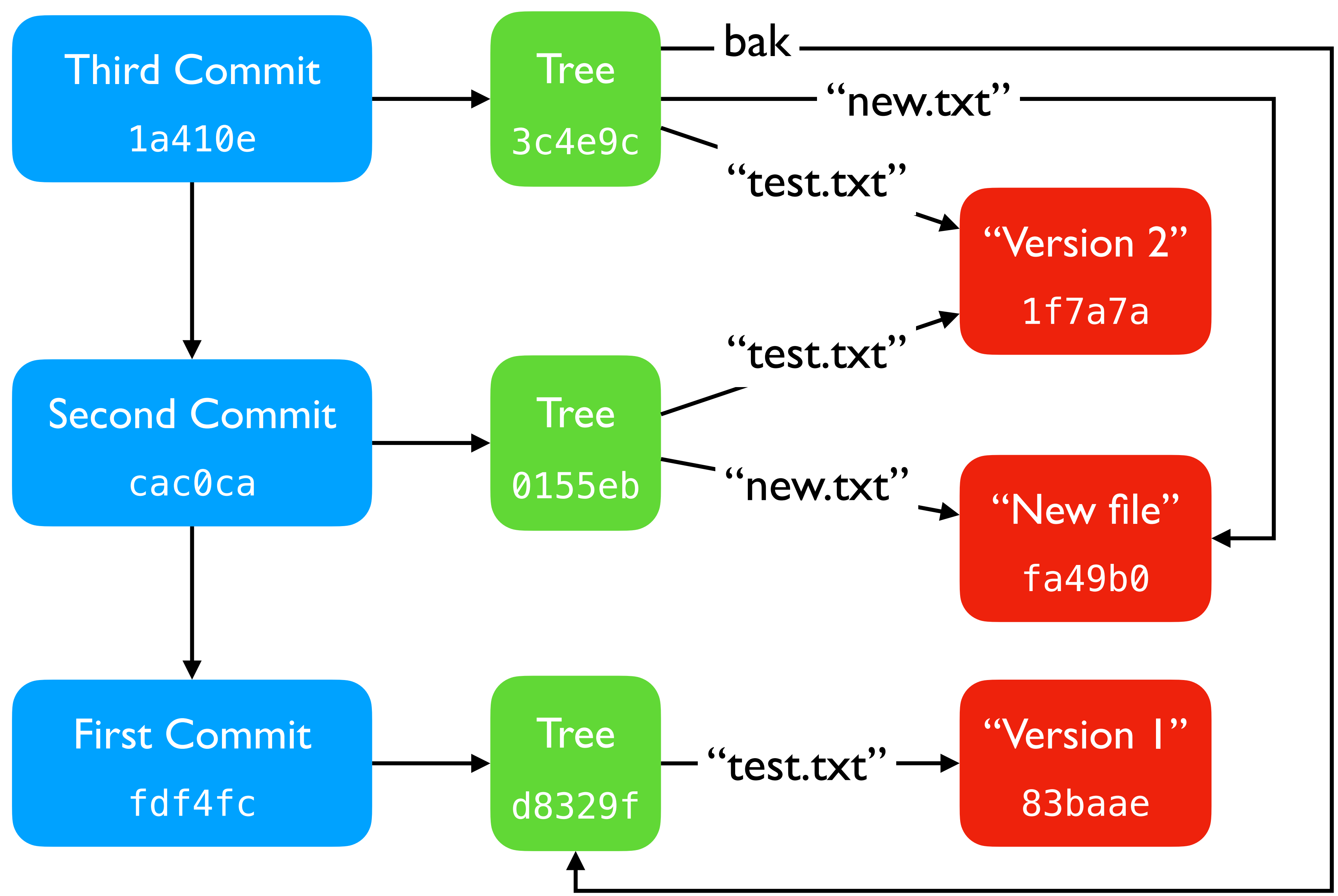
- Stores the files under version control
- Immutable, append-only & content addressed
  - hash → object

Block store and persistence

Added original `/text.txt` at `/bak/text.txt`

Added `/new.txt`
Modified `/test.txt`

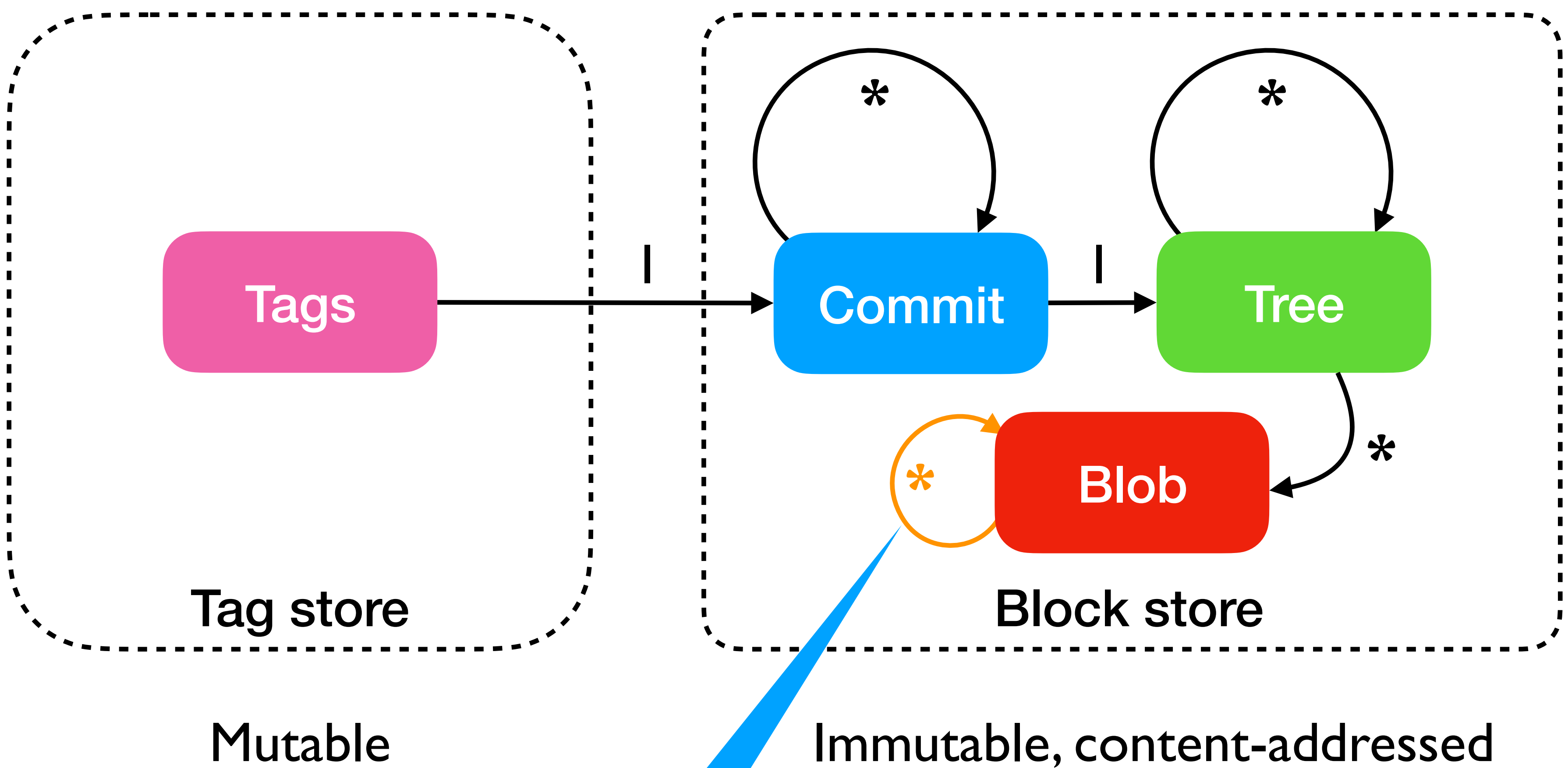
Added `/test.txt`



Example from Pro Git book: https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

Irmin store

- A Git-like distributed database
- MRDTs are executed on top of Irmin

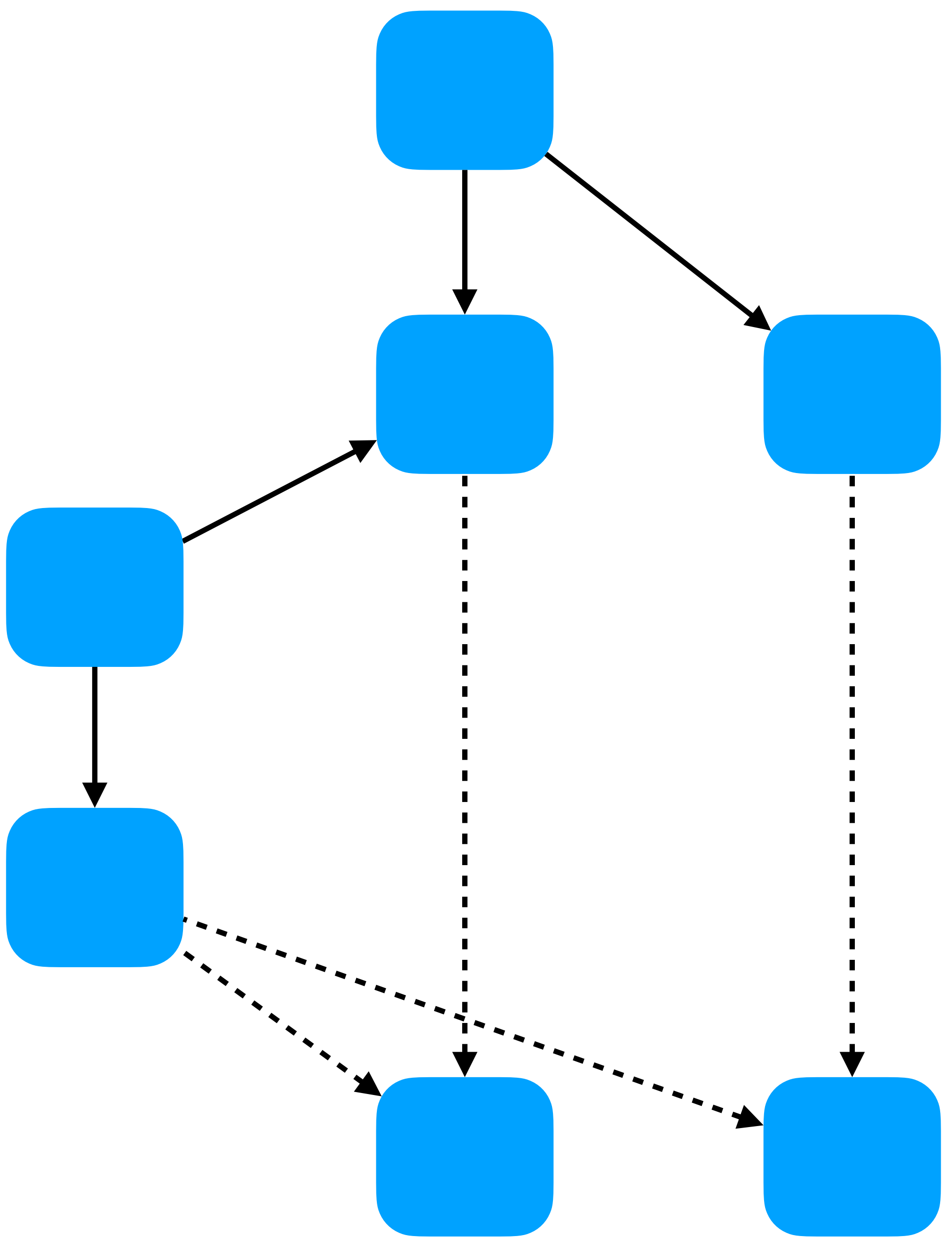


Persistent algebraic data types:
Stacks, Queues, Ropes, Balanced binary trees, etc.

Kaki et al, "*Mergeable Replicated Data Types*", OOPSLA 2019

Commit DAG

- Commit nodes form a DAG
  - ✦ Captures causal history ⇒ *Happens-before* relation
- MRDTs provide *causal* consistency
  - ✦ Strongest consistency level without coordination
- LCA discovered by traversing the commit DAG



Forgetting History

- Git remembers the entire history
  - Useful if *provenance* is necessary
- If not, keeping entire history is wasteful
  - Nodes will *run out of storage* quicker
- But, history needed for LCA in a 3-way merge
- *How much history should we keep?*
- *Any commits older than the latest commit "K" known by all replicas can be GCed*

Dubey et al, "*Banyan: Coordination-free Distributed Transactions over Mergeable Types*", APLAS 2020

Dubey, "*Banyan: Coordination-free Distributed Transactions over Mergeable Types*", MS Thesis, IIT Madras

MRDTs = Sequential data types + 3-way merge

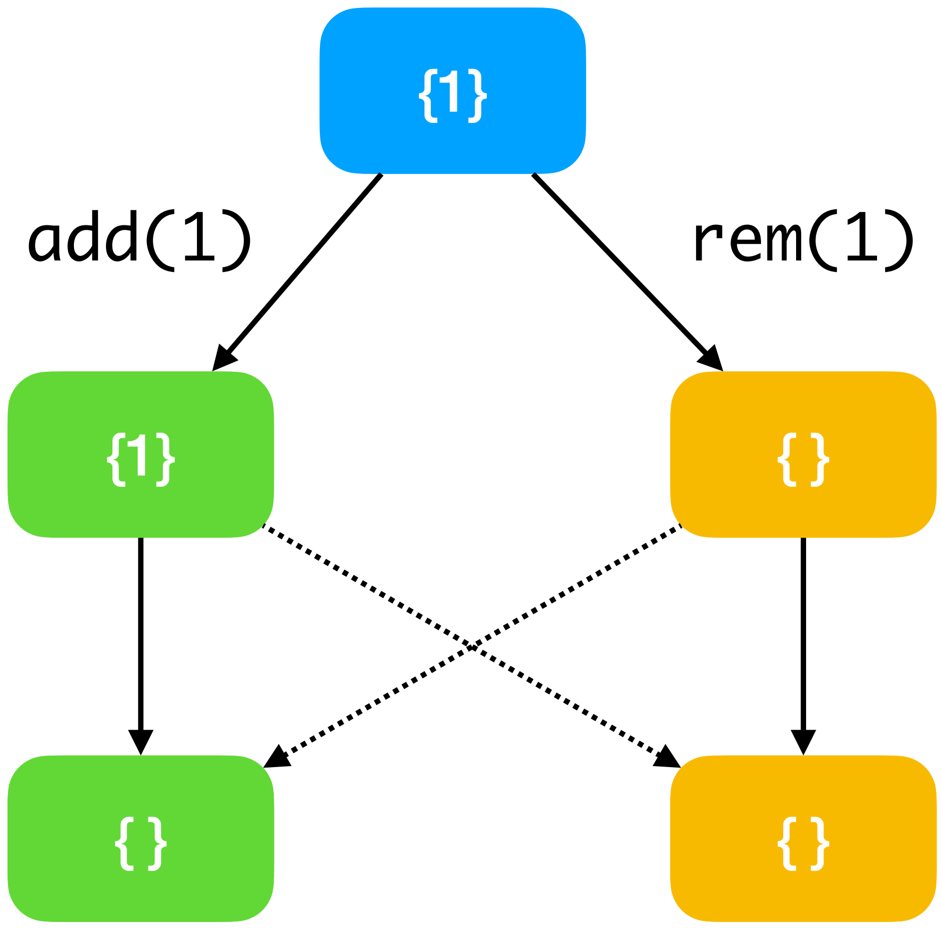
Does this make proving MRDTs correct easier?

Is our set an add-wins set?

- *add-wins* when there is a concurrent add and remove of the same element
- Also known as *Observed-Removed set (OR-set)*

```
let merge ~lca ~v1 ~v2 =
  (lca ∩ v1 ∩ v2) (∗ common elements ∗)
  ∪ (v1 − lca)    (∗ added in v1 ∗)
  ∪ (v2 − lca)    (∗ added in v2 ∗)
```

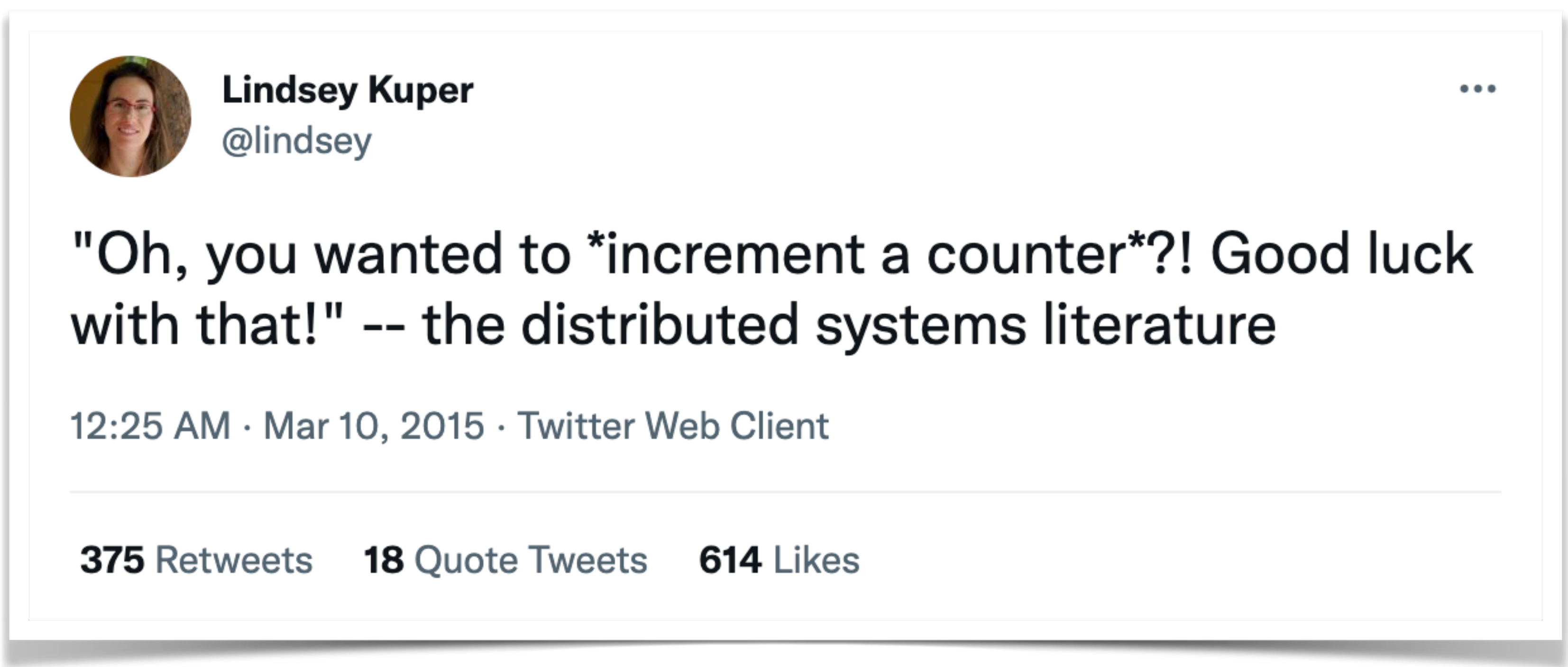
```
     { } (∗ common elements ∗)
   ∪ { } (∗ added in v1 ∗)
   ∪ { } (∗ added in v2 ∗)
   = { } ∪ { } ∪ { }
   = { } (expected {1})
```



- *Our set is not add-wins set!*
- Convergence is not sufficient; *Intent* is not preserved

Concretising Intent

- A *formal specification language* to capture the *intent* of the MRDT
    - ✦ Must be rich enough to capture distributed execution
- Even *simple* data types attract enormous *complexity* when made *distributed*



- *Mechanization* to bridge the gap between spec and impl

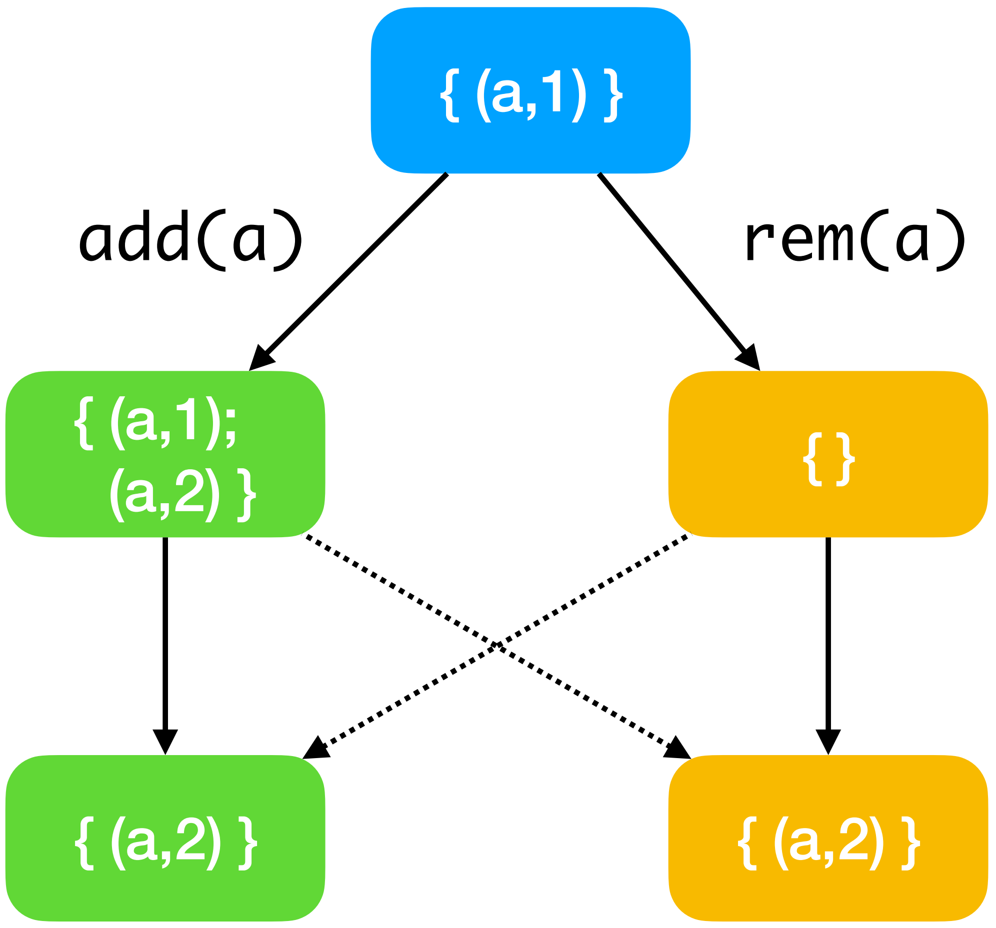
Peepul — Certified MRDTs

- F* library implementing and proving MRDTs
  - ★ F* — proof-oriented, solver-aided PL
- Specification language is event-based
  - ★ Burckhardt et al. "Replicated Data Types: Specification, Verification and Optimality", POPL 2014
- *Replication-aware simulation* to connect *specification* with *implementation*
- *Space- and time-efficient* implementations
  - ★ 1st certified implementation of a O(1) replicated queue with O(n) merge.
- *Composition* of MRDTs and their proofs!
- Extracted OCaml RDTs are compatible with *Irmin*

Fixing Add-wins Set

- Discriminate duplicate additions by associating a unique id

```
{ } ∪          (* common *)
{ (a,2) } ∪ (* added left *)
{ }            (* added right *)
= { (a,2) }
```



MRDT Implementation

$$D_\tau = (\Sigma, \sigma_0, do, merge)$$

$\Sigma = \mathcal{P}(\mathbb{N} \times \mathbb{N})$

$\sigma_0 = \{\}$

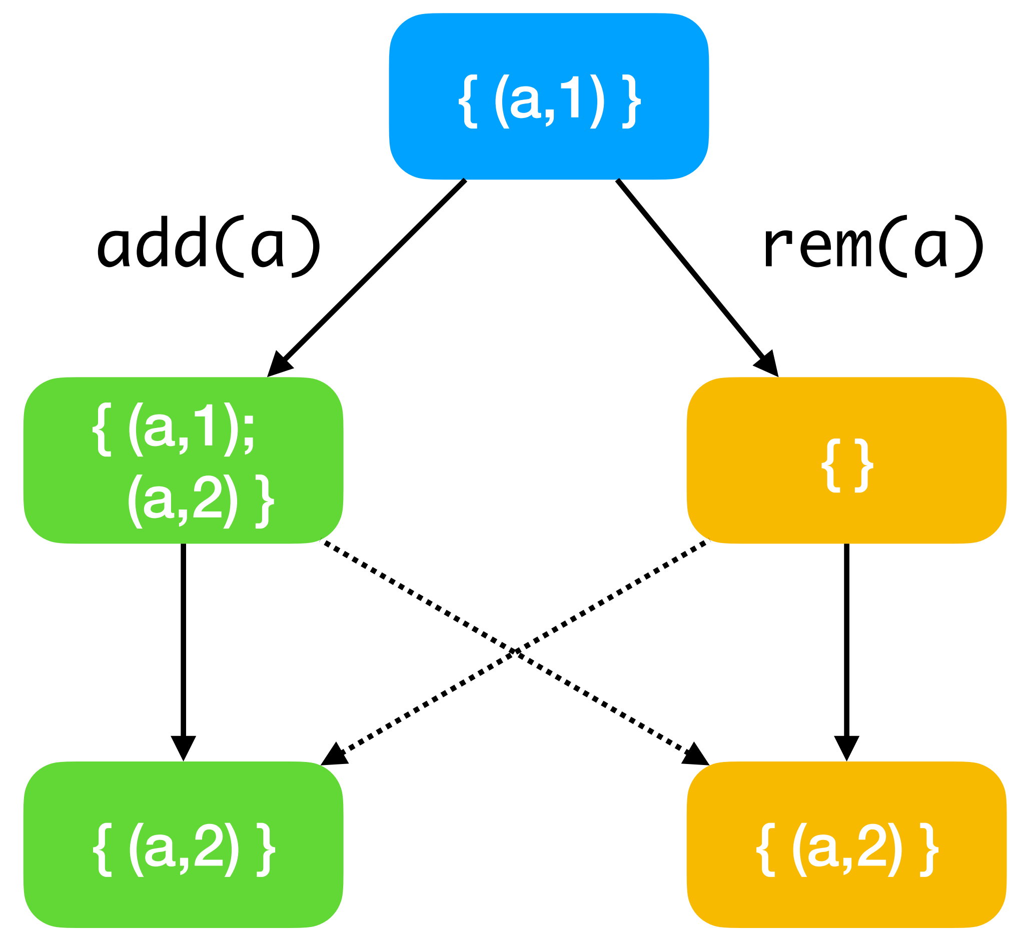
$do(rd, \sigma, t) = (\sigma, \{a \mid (a, t') \in \sigma\})$

$do(add(a), \sigma, t) = (\sigma \cup \{(a, t)\}, \bot)$

$do(remove(a), \sigma, t) = (\{e \in \sigma \mid fst(e) \neq a\}, \bot)$

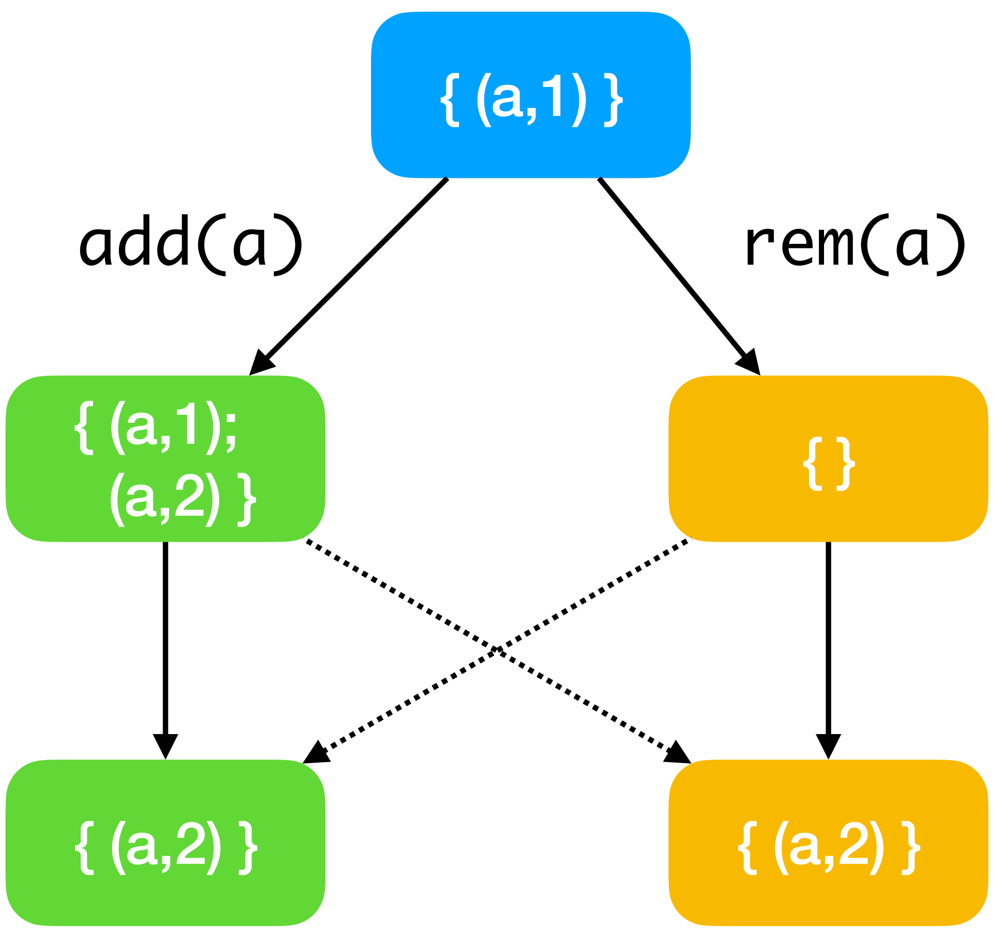
$merge(\sigma_{lca}, \sigma_a, \sigma_b) =$
$(\sigma_{lca} \cap \sigma_a \cap \sigma_b) \cup (\sigma_a - \sigma_{lca}) \cup (\sigma_b - \sigma_{lca})$

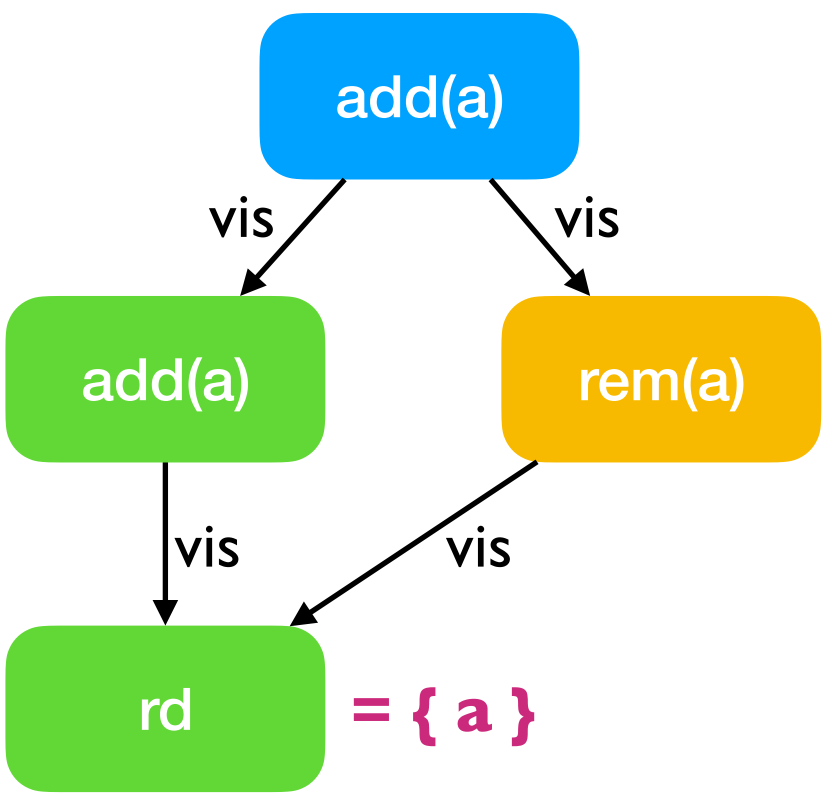
Unique Lamport Timestamp



Specifying Add-wins Set

Abstract state $I = \langle E, oper, rval, time, vis \rangle$





$$\mathcal{F}_{orset}(\mathrm{rd}, \langle E, oper, rval, time, vis \rangle) = \{a \mid \exists e \in E.\ oper(e)$$

$$= \mathrm{add}(a) \wedge \neg(\exists f \in E.\ oper(f) = \mathrm{remove}(a) \wedge e \xrightarrow{vis} f)\}$$

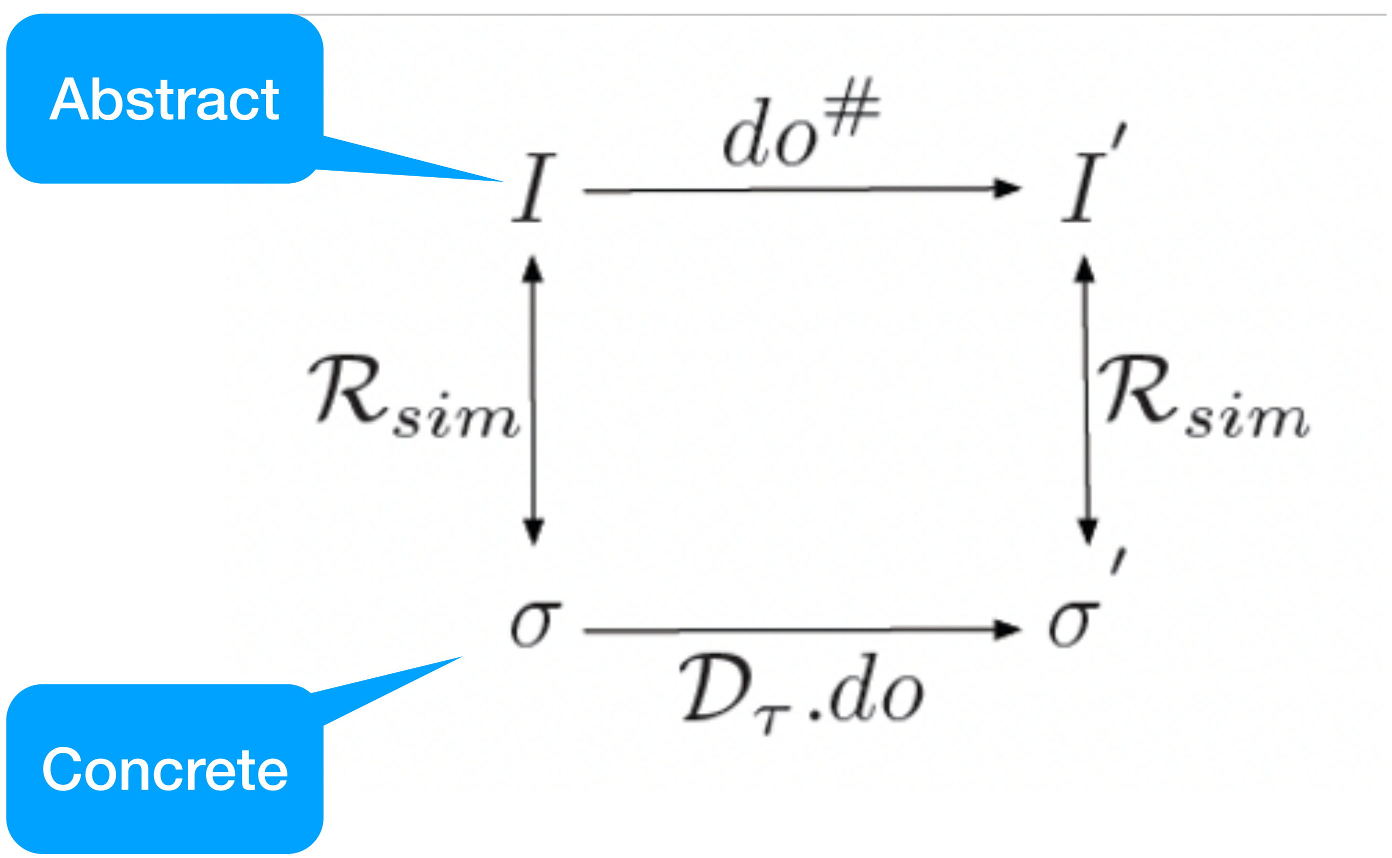
Simulation Relation

- Connects the *abstract state* with the *concrete state*
- For the add-wins set,

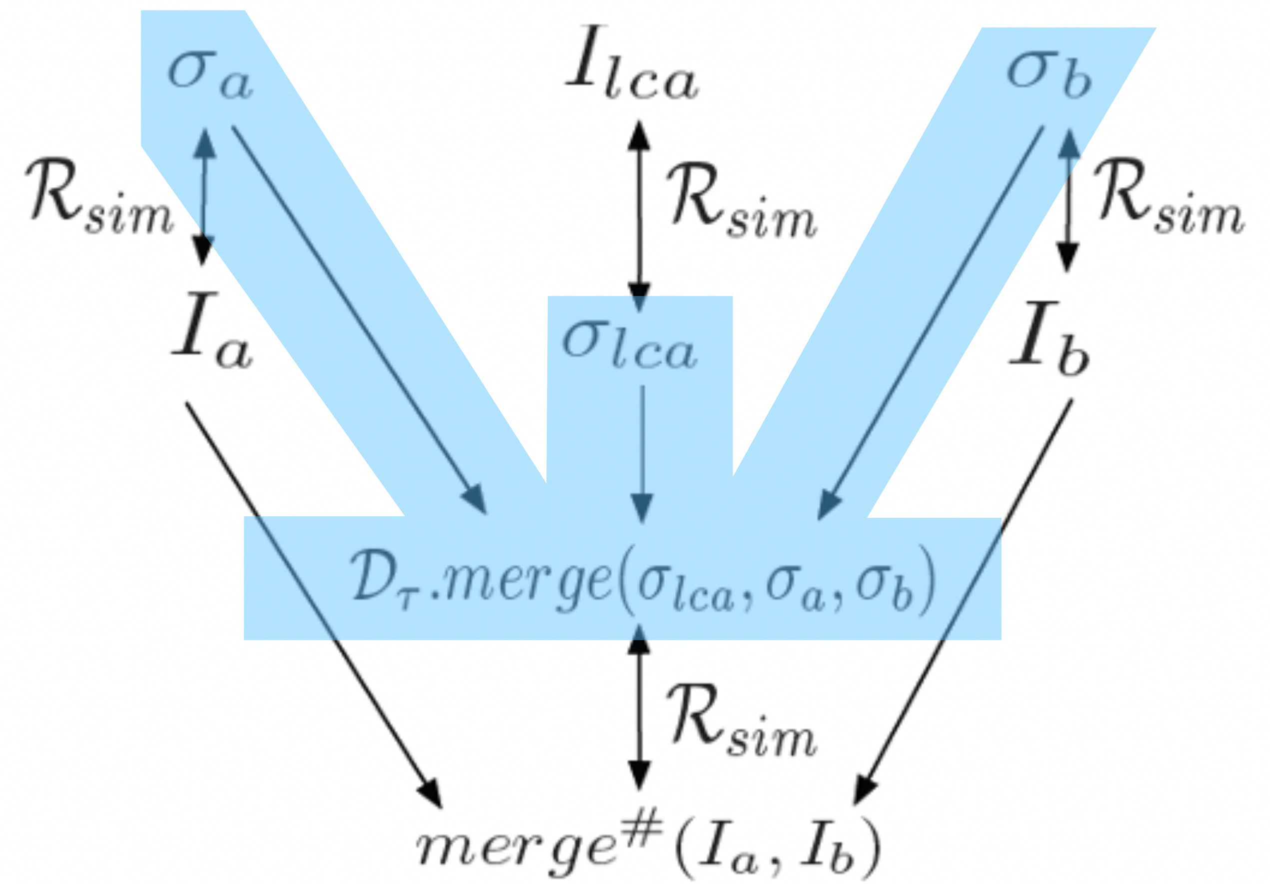
$$\mathcal{R}_{sim}(I,\sigma) \iff (\forall (a,t) \in \sigma \iff$$
$$(\exists e \in I.E \wedge I.\,oper(e) = add(a) \wedge I.time(e) = t \wedge$$
$$\neg(\exists f \in I.E \wedge I.\,oper(f) = remove(a) \wedge e \xrightarrow{vis} f)))$$

- The main verification effort is to show that the relation above is indeed a *simulation relation*
  - ★ Shown separately for *operations* and *merge function*
  - ★ Proof by induction on the execution trace

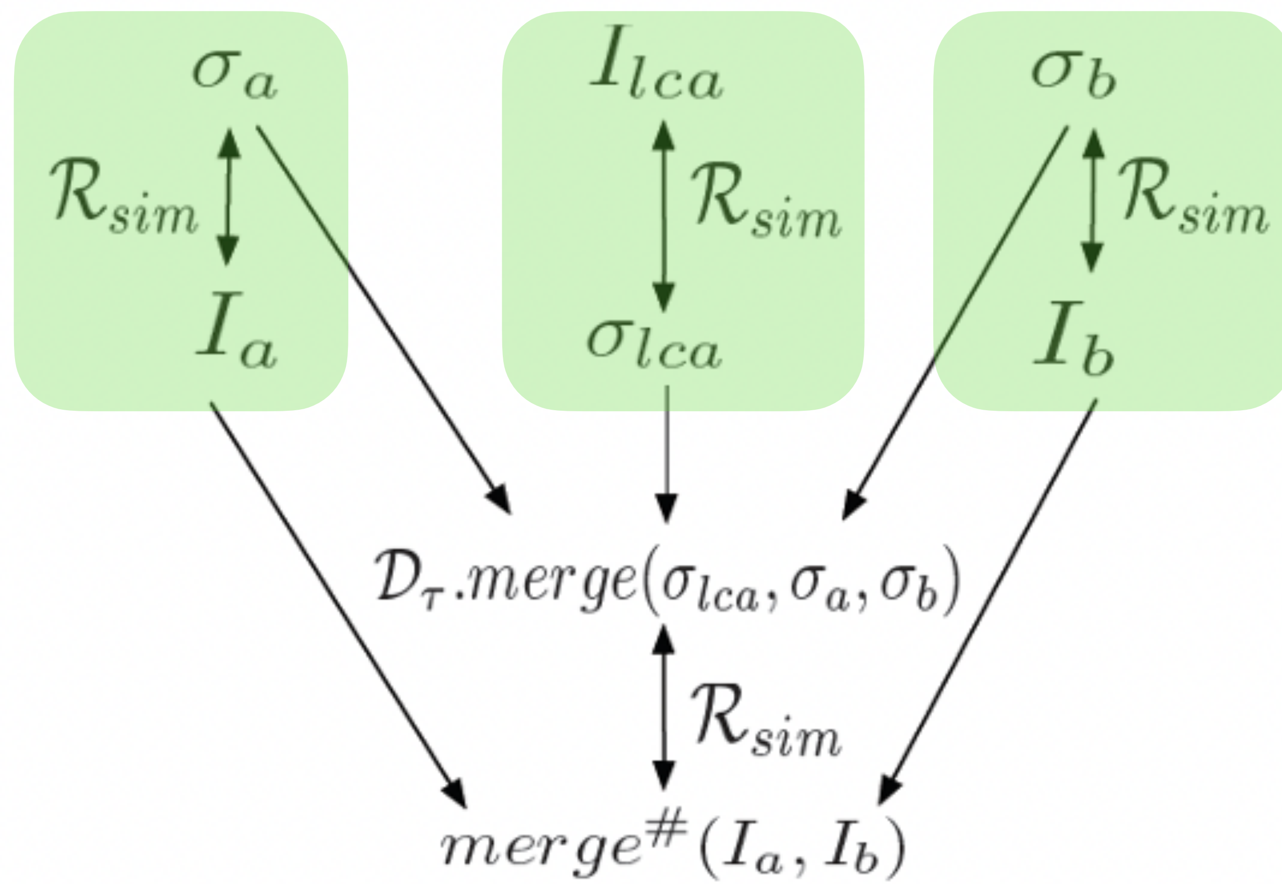
Verifying operations



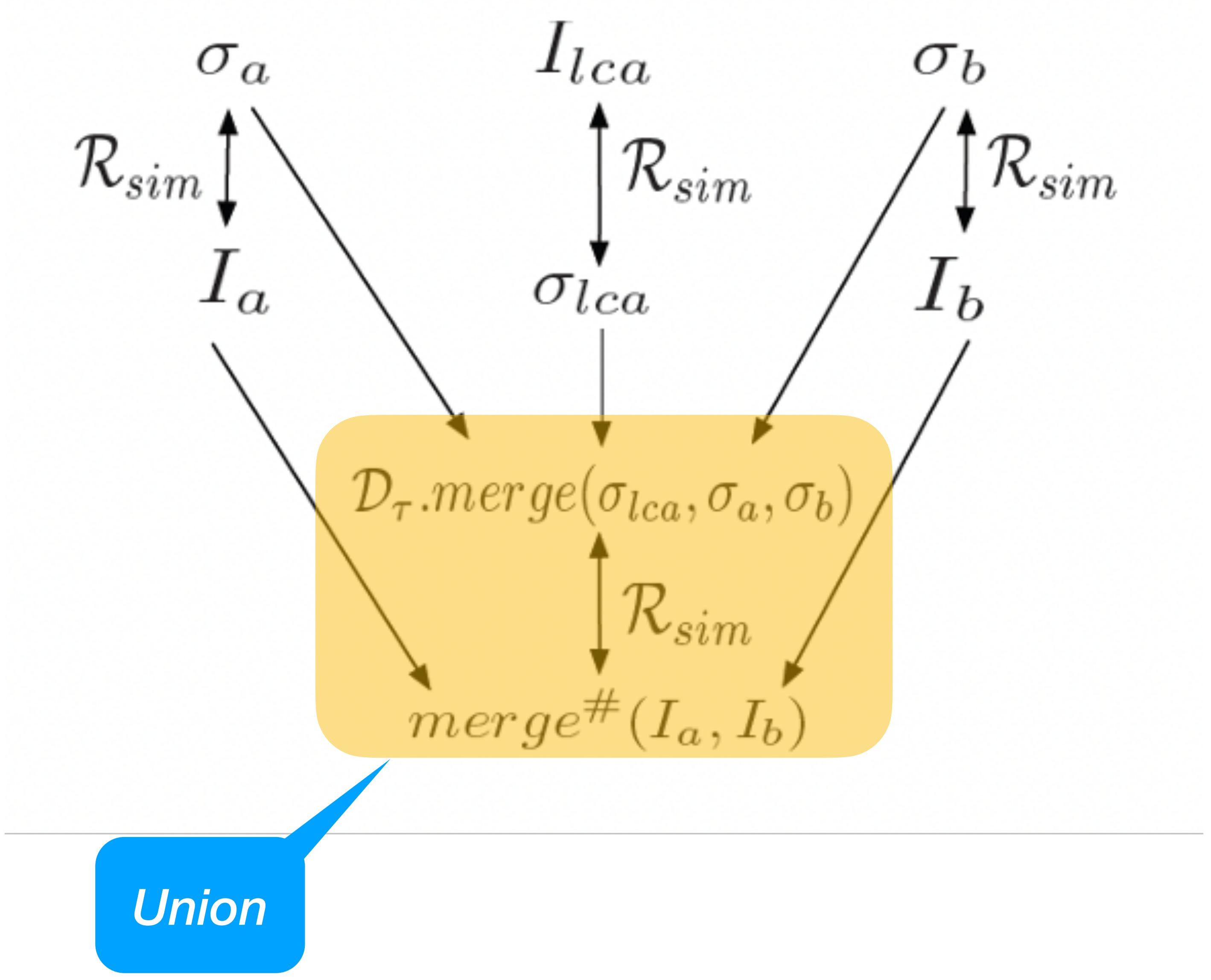
Verifying Merge function



Verifying Merge function



Verifying Merge function



Verification effort

MRDTs verified	#Lines code	#Lines proof	#Lemmas	Verif. time (s)
Increment-only counter	6	43	2	3.494
PN counter	8	43	2	23.211
Enable-wins flag	20	58	3	1074
		81	6	171
		89	7	104
LWW register	5	44	1	4.21
G-set	10	23	0	4.71
		28	1	2.462
		33	2	1.993
G-map	48	26	0	26.089
Mergeable log	39	95	2	36.562
OR-set (§2.1.1)	30	36	0	43.85
		41	1	21.656
		46	2	8.829
OR-set-space (§2.1.2)	59	108	7	1716
OR-set-spacetime	97	266	7	1854
Queue	32	1123	75	4753

Composing RDTs is HARD!

Martin Kleppmann
@martinkl

Today in "distributed systems are hard": I wrote down a simple CRDT algorithm that I thought was "obviously correct" for a course I'm teaching. Only 10 lines or so long. Found a fatal bug only after spending hours trying to prove the algorithm correct. 😭

4:18 AM · Nov 13, 2020 · Tweetbot for iOS

41 Retweets **4** Quote Tweets **541** Likes

Martin Kleppmann @martinkl · Nov 13, 2020

The interesting thing about this bug is that it comes about only from the interaction of two features. A LWW map by itself is fine. A set in which you can insert and delete elements (but not update them) is fine. The problem arises only when delete and update interact.

1 16

Composing IRC-style chat

- Build IRC-style group chat
  - ★ Send and read messages in channels
- Represent application state as a map MRDT
  - ★ String (*channel name*) keys → mergeable log MRDT values
  - ★ Mergeable log — message + timestamp; merge ordered by timestamp
- **Goal:**
  - ★ *map* and *log* proved correct separately
  - ★ Use the proof of underlying RDTs to prove chat application correctness

Generic Map MRDT

Implementation

$$\mathcal{D}_{\alpha-map} = (\Sigma, \sigma_0, do, merge_{\alpha-map}) \text{ where}$$

$$\Sigma_{\alpha-map} = \mathcal{P}(string \times \Sigma_\alpha)$$

The values in the MRDT map are MRDTs

$$\sigma_0 = \{\}$$

$$\delta(\sigma, k) = \begin{cases} \sigma(k), & \text{if } k \in dom(\sigma) \\ \sigma_{0_\alpha}, & \text{otherwise} \end{cases}$$

$$do(set(k, o_\alpha), \sigma, t) = \text{let } (v, r) = do_\alpha(o_\alpha, \delta(\sigma, k), t) \text{ in } (\sigma[k \mapsto v], r)$$

$$do(get(k, o_\alpha), \sigma, t) = \text{let } (\_, r) = do_\alpha(o_\alpha, \delta(\sigma, k), t) \text{ in } (\sigma, r)$$

$$merge_{\alpha-map}(\sigma_{lca}, \sigma_a, \sigma_b) = \{(k, v) \mid (k \in dom(\sigma_{lca}) \cup dom(\sigma_a) \cup dom(\sigma_b)) \wedge v = merge_\alpha(\delta(\sigma_{lca}, k), \delta(\sigma_a, k), \delta(\sigma_b, k))$$

Merge uses the merge of the underlying value type!

Simulation Relation

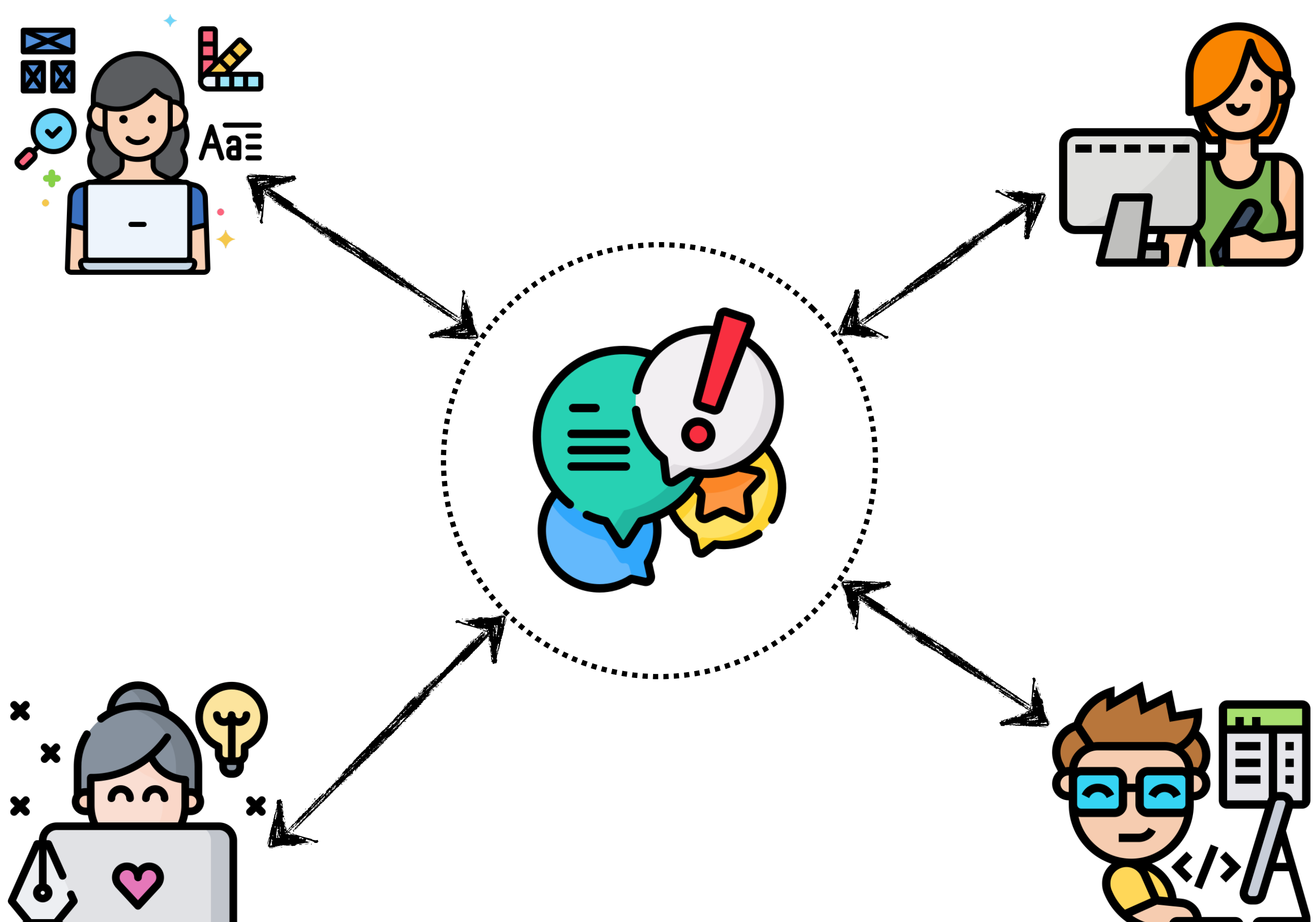
$$\mathcal{R}_{sim-\alpha-map}(I, \sigma) \iff \forall k.$$
$$(k \in dom(\sigma) \iff \exists e \in I.E.\ oper(e) = set(k, \_)) \wedge$$
$$\mathcal{R}_{sim-\alpha}\ (project(k, I), \delta(\sigma, k))$$

Simulation relation appeals to the value type's simulation relation!

Composing IRC-style chat

- IRC app state is constructed by instantiating *generic map* with *mergeable log*
- The proof of correctness of the chat application directly follows from the composition.
  - ★ See paper for details!

Soundarapandian et al, "*Certified Mergeable Replicated Data Types*", PLDI 2022

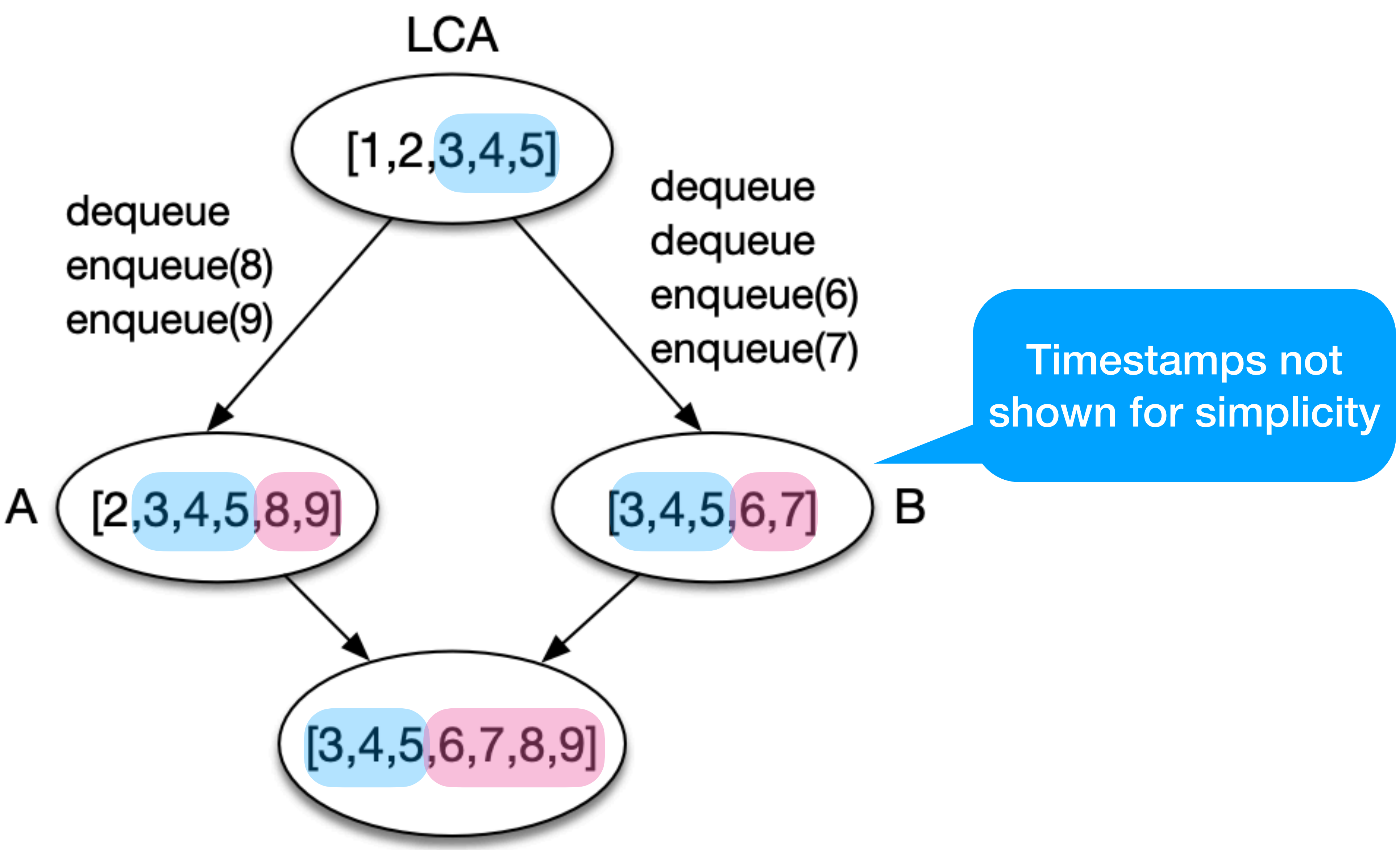


Verification is still too hard!

MRDTs verified	#Lines code	#Lines proof	#Lemmas	Verif. time (s)
Increment-only counter	6	43	2	3.494
PN counter	8	43	2	23.211
Enable-wins flag	20	58	3	1074
		81	6	171
		89	7	104
LWW register	5	44	1	4.21
G-set	10	23	0	4.71
		28	1	2.462
		33	2	1.993
G-map	48	26	0	26.089
Mergeable log	39	95	2	36.562
OR-set (§2.1.1)	30	36	0	43.85
		41	1	21.656
		46	2	8.829
OR-set-space (§2.1.2)	59	108	7	1716
OR-set-spacetime	97	266	7	1854
Queue	32	1123	75	4753

Queue MRDT — Implementation

- Two-list queue + merge function
- "At least once" semantics for dequeue
- Each element *e* enqueued is *(e, t)* where *t* is the unique Lamport timestamp



Queue MRDT — Specification

$$\text{match}_I(e_1, e_2) \Leftrightarrow I.oper(e_1) = enqueue(a)$$
$$\wedge\, I.oper(e_2) = dequeue \wedge a = I.rval(e_2)$$

- $AddRem(I) : \forall e \in I.E.\ I.oper(e) = dequeue \wedge I.rval(e) \neq \text{EMPTY} \implies \exists e' \in I.E.\ \text{match}_I(e', e)$
- $Empty(I) : \forall e_1, e_2, e_3 \in I.E.\ I.oper(e_1) = dequeue \wedge I.rval(e_1) = \text{EMPTY} \wedge I.oper(e_2) = enqueue(a) \wedge e_2 \xrightarrow{I.vis} e_1 \implies \exists e_3 \in I.E.\ \text{match}_I(e_2, e_3) \wedge e_3 \xrightarrow{I.vis} e_1$
- $FIFO_1(I) : \forall e_1, e_2, e_3 \in I.E.\ I.oper(e_1) = enqueue(a) \wedge \text{match}_I(e_2, e_3) \wedge e_1 \xrightarrow{I.vis} e_2 \implies \exists e_4 \in I.E.\ \text{match}_I(e_1, e_4)$
- $FIFO_2(I)$: $\forall e_1, e_2, e_3, e_4 \in I.E.\ \neg(\text{match}_I(e_1, e_4) \wedge \text{match}_I(e_2, e_3) \wedge e_1 \xrightarrow{I.vis} e_2 \wedge e_3 \xrightarrow{I.vis} e_4)$

- Extremely *hard* to write specs over event-based structures
- Simulation relations are *harder*

Better Specification

- *Sequential data type + constraints* as the specification for MRDT
  - ✦ Constraints — ordering, commutativity, duplication, …
  - ✦ MRDT behaviour = constrained linearisation + Sequential DT

Add-wins set ordering constraint

Op1	Op2	Order
add(a)	rem(a)	Op2, Op1
rem(a)	add(a)	Op1, Op2
add(_)	add(_)	Any
rem(_)	rem(_)	Any
add(a)	rem(b)	Any
rem(a)	add(b)	Any

Summary

- MRDT simplify the construction of RDTs
  - ✦ Sequential data types + 3-way merge functions
- *Persistent data structures* to efficiently record causal history
- 3-way merge function is a pathway to verifying MRDTs

Irmin