Certified Mergeable Replicated Data Types

“KC” Sivaramakrishnan

joint work with

Vimala Soundarapandian, Adharsh Kamath and Kartik Nagar

IIT MADRAS

Tarides

VISA

ebay

YouTube

NETFLIX

HSBC

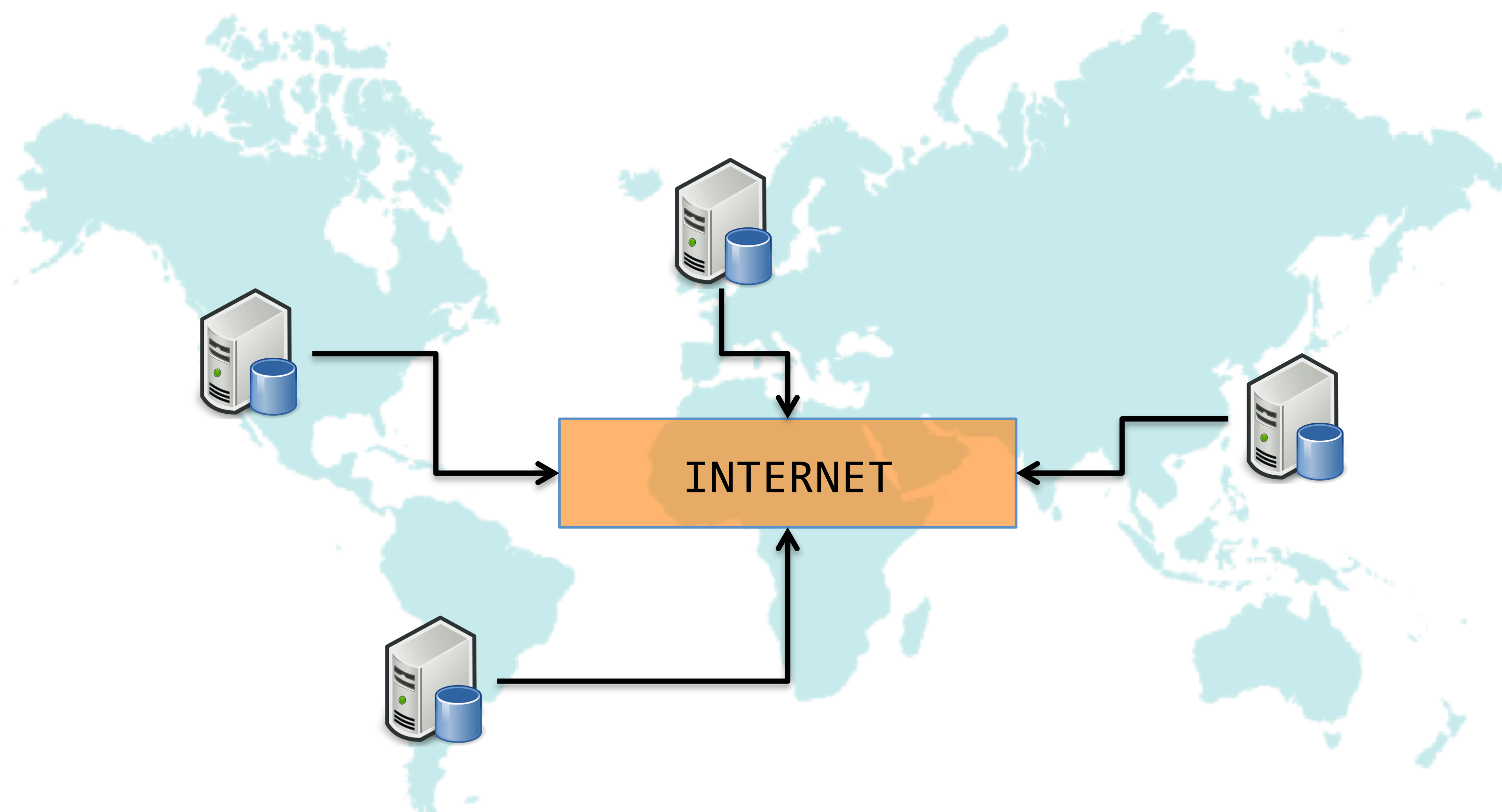
VISA

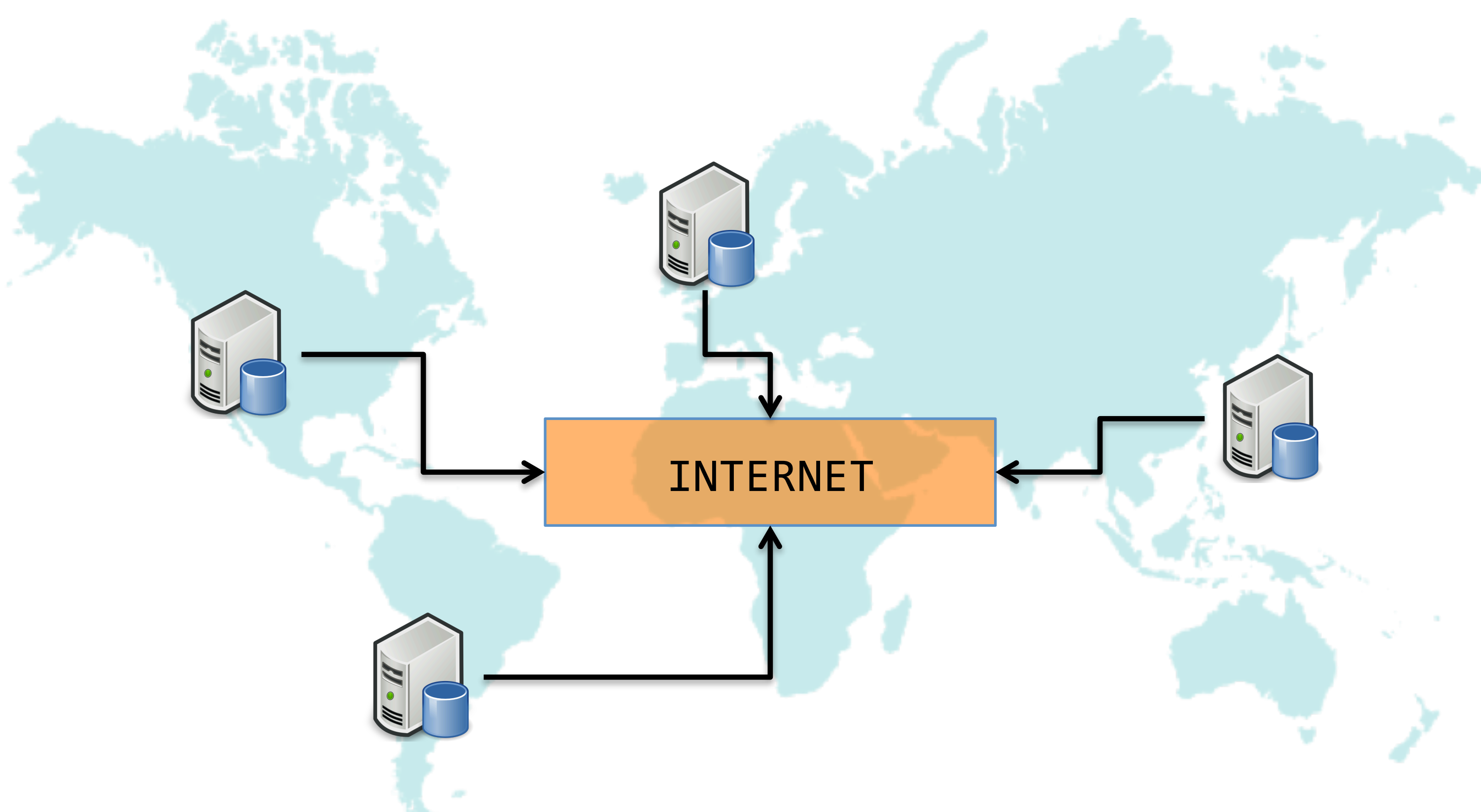
ebay

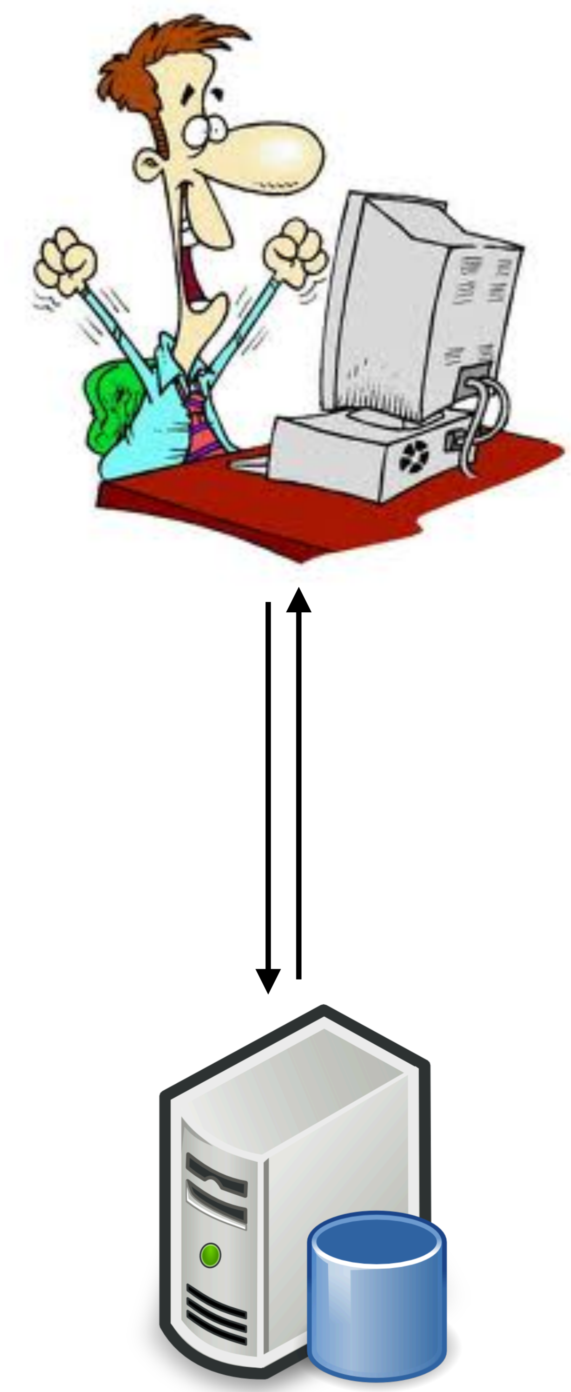
YouTube

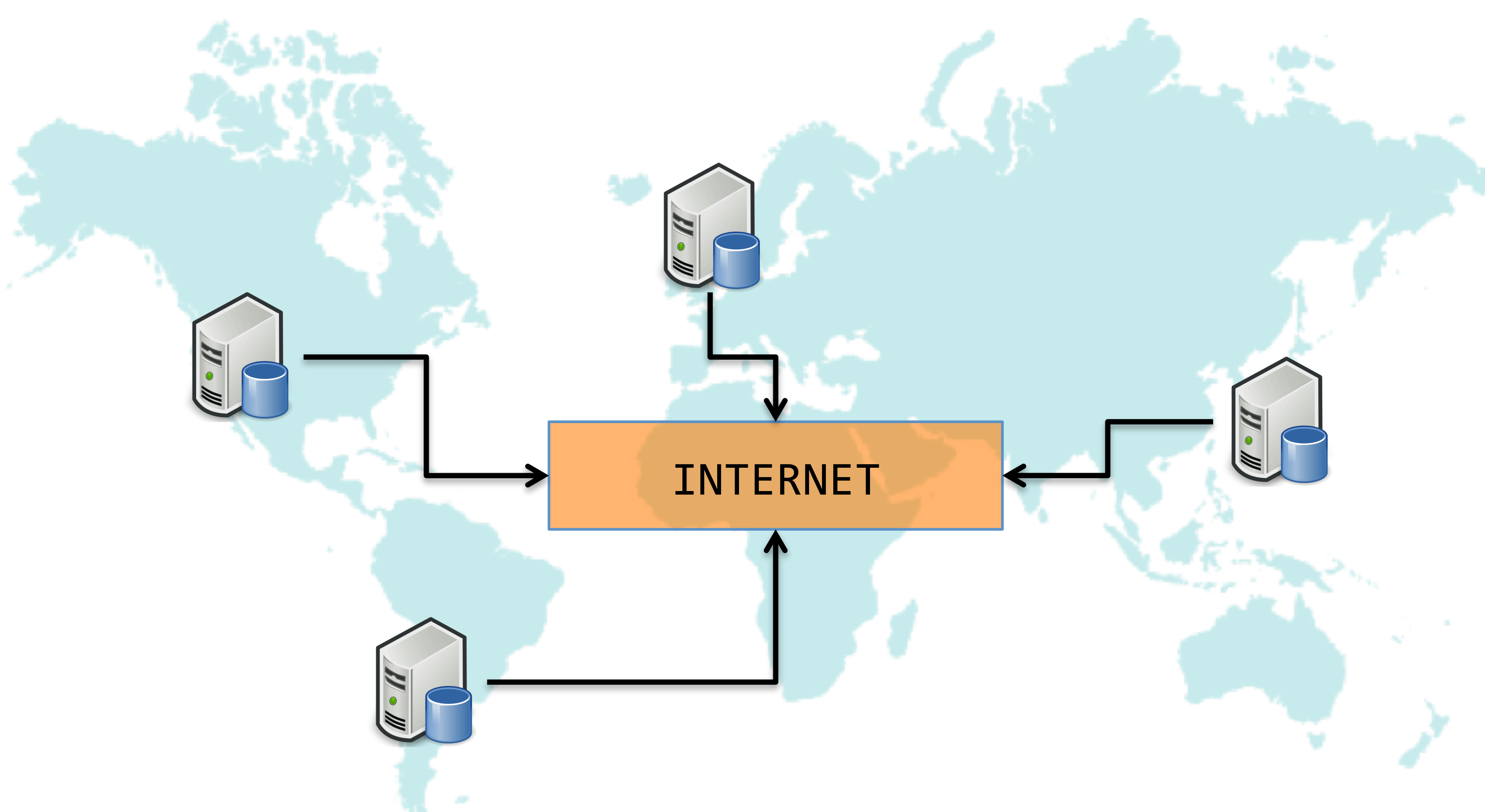
NETFLIX

HSBC

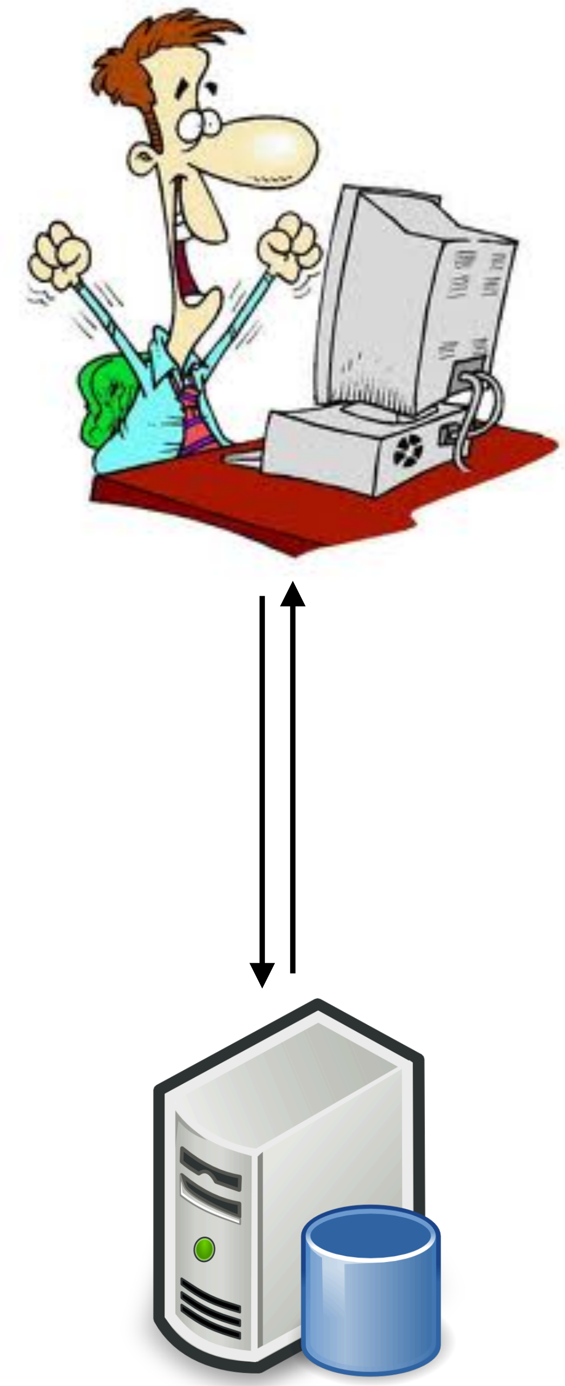


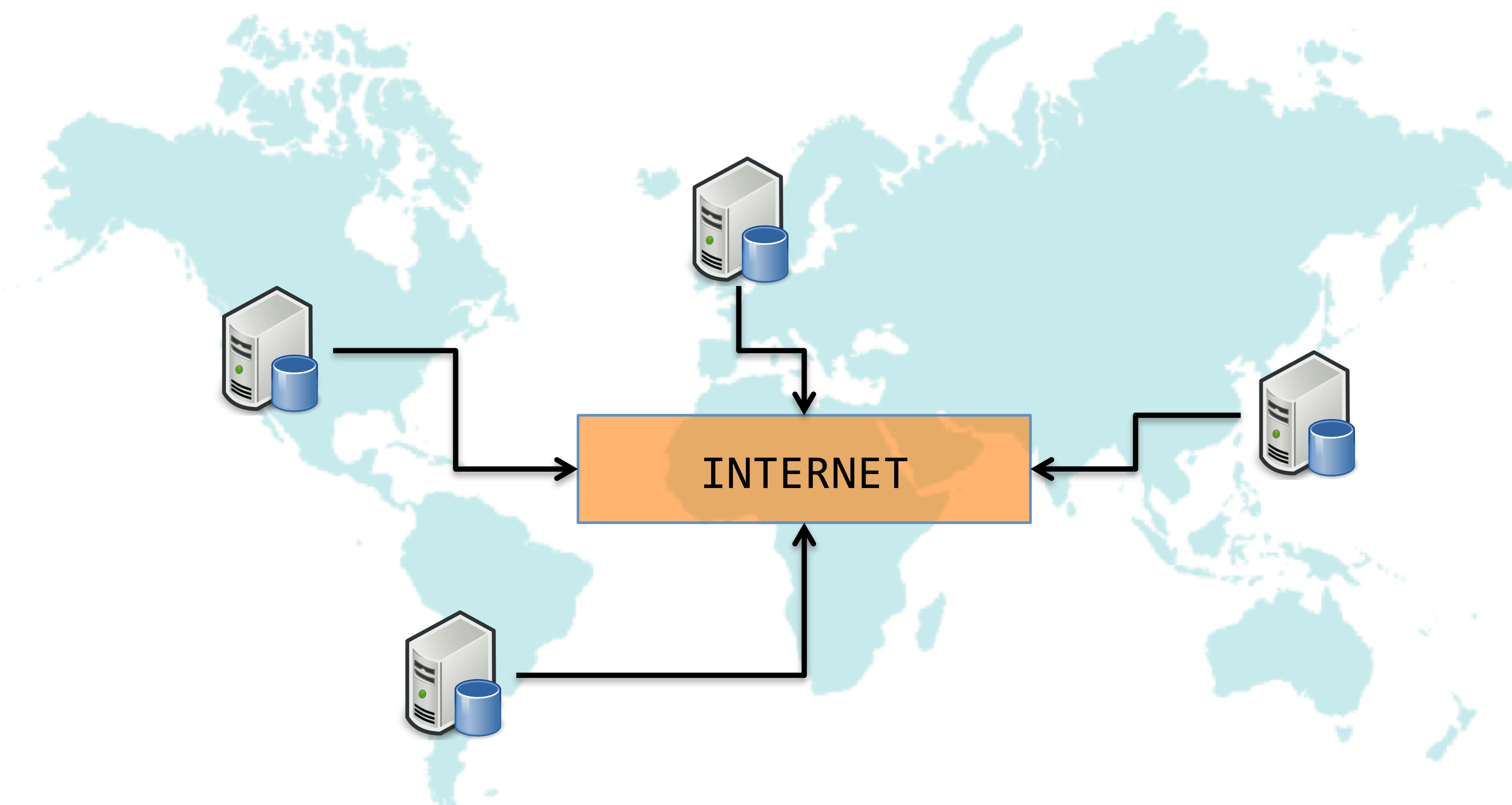

INTERNET




INTERNET

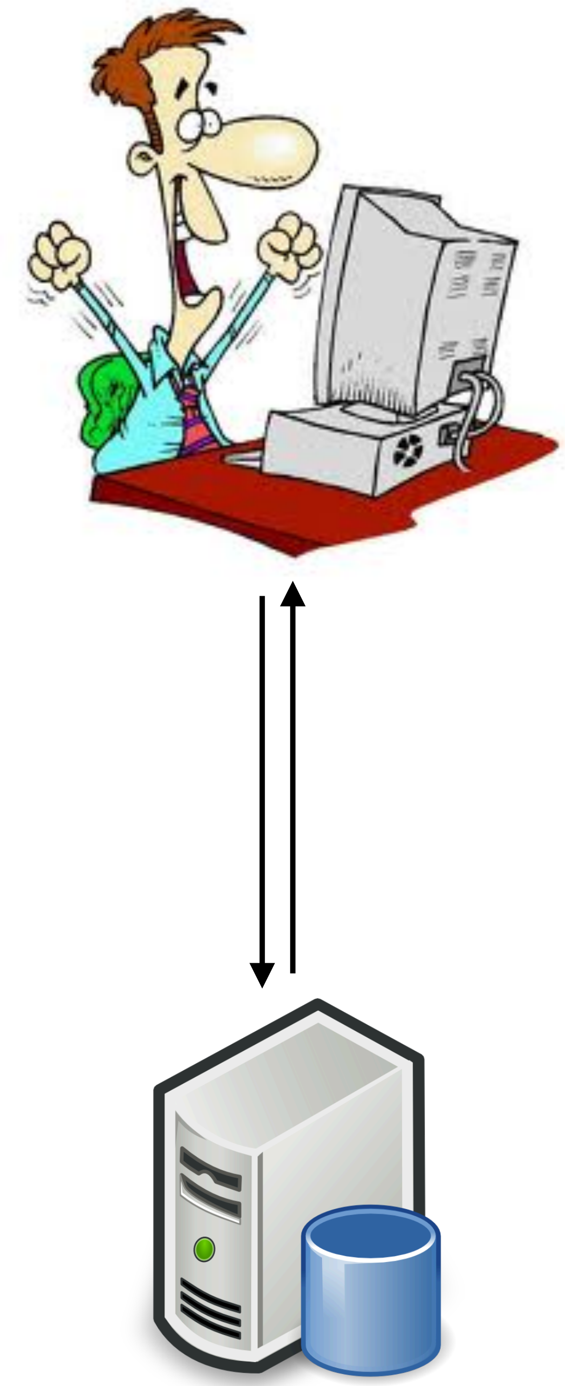
≠





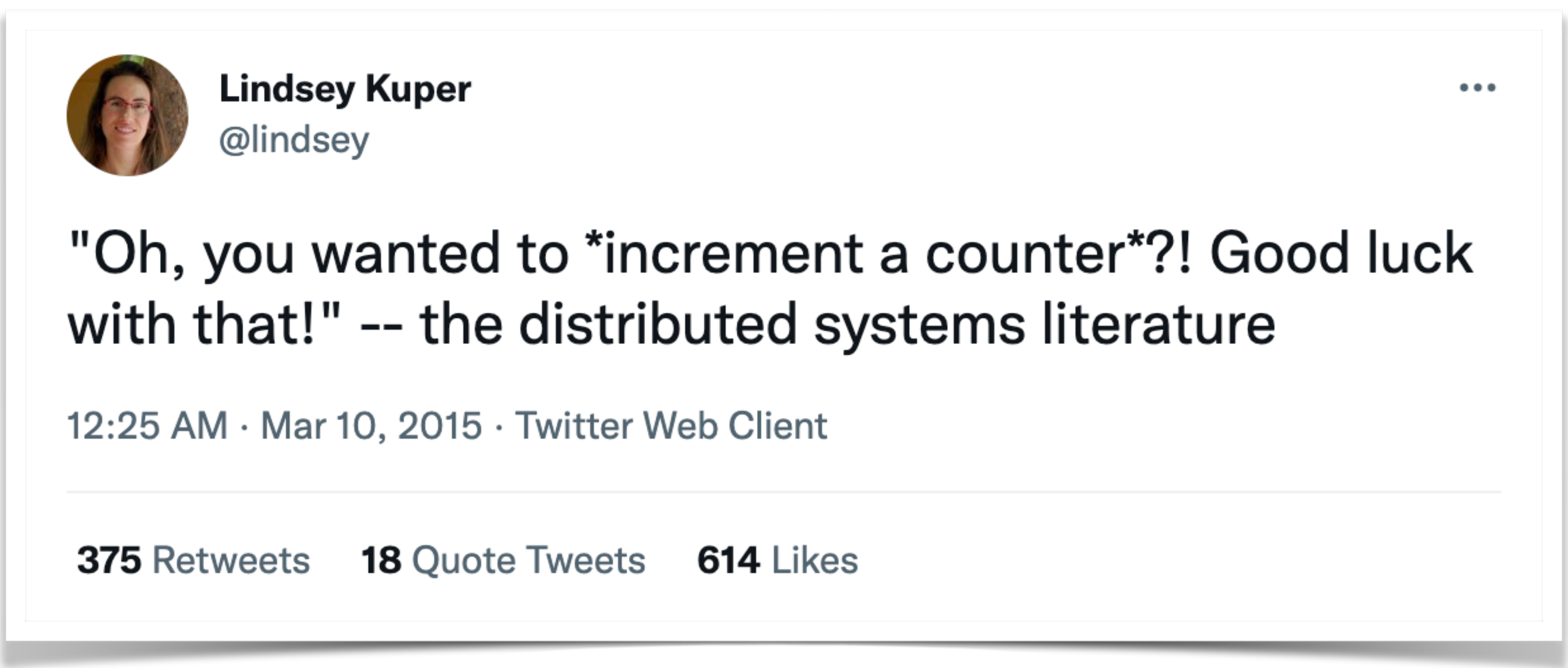
• Weak Consistency & Isolation

≠



• Serializability
• Linearizability

Even *simple* data structures attract enormous *complexity* when made *distributed*



Sequential Counter

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
end
```

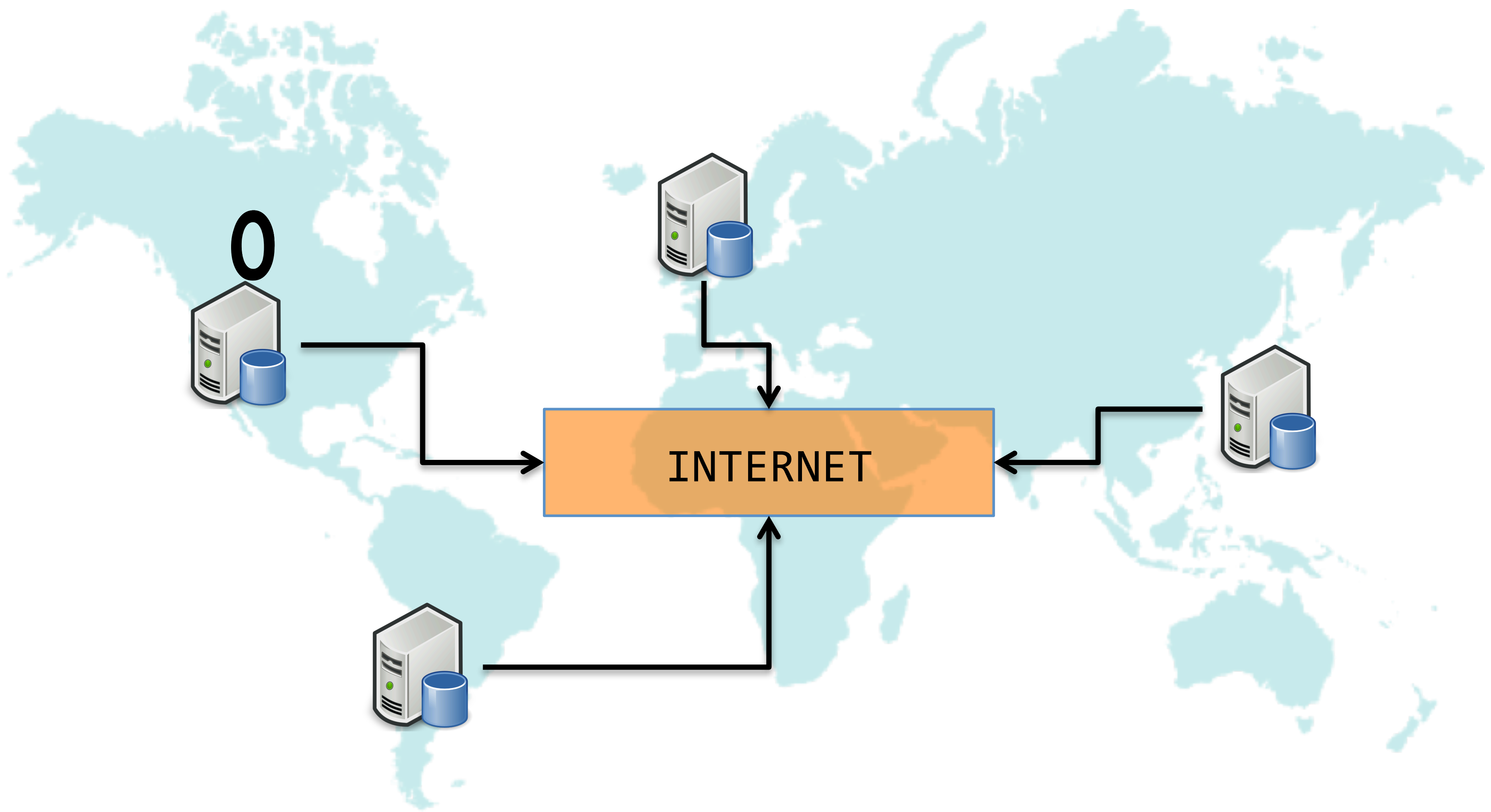
Sequential Counter

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
end
```

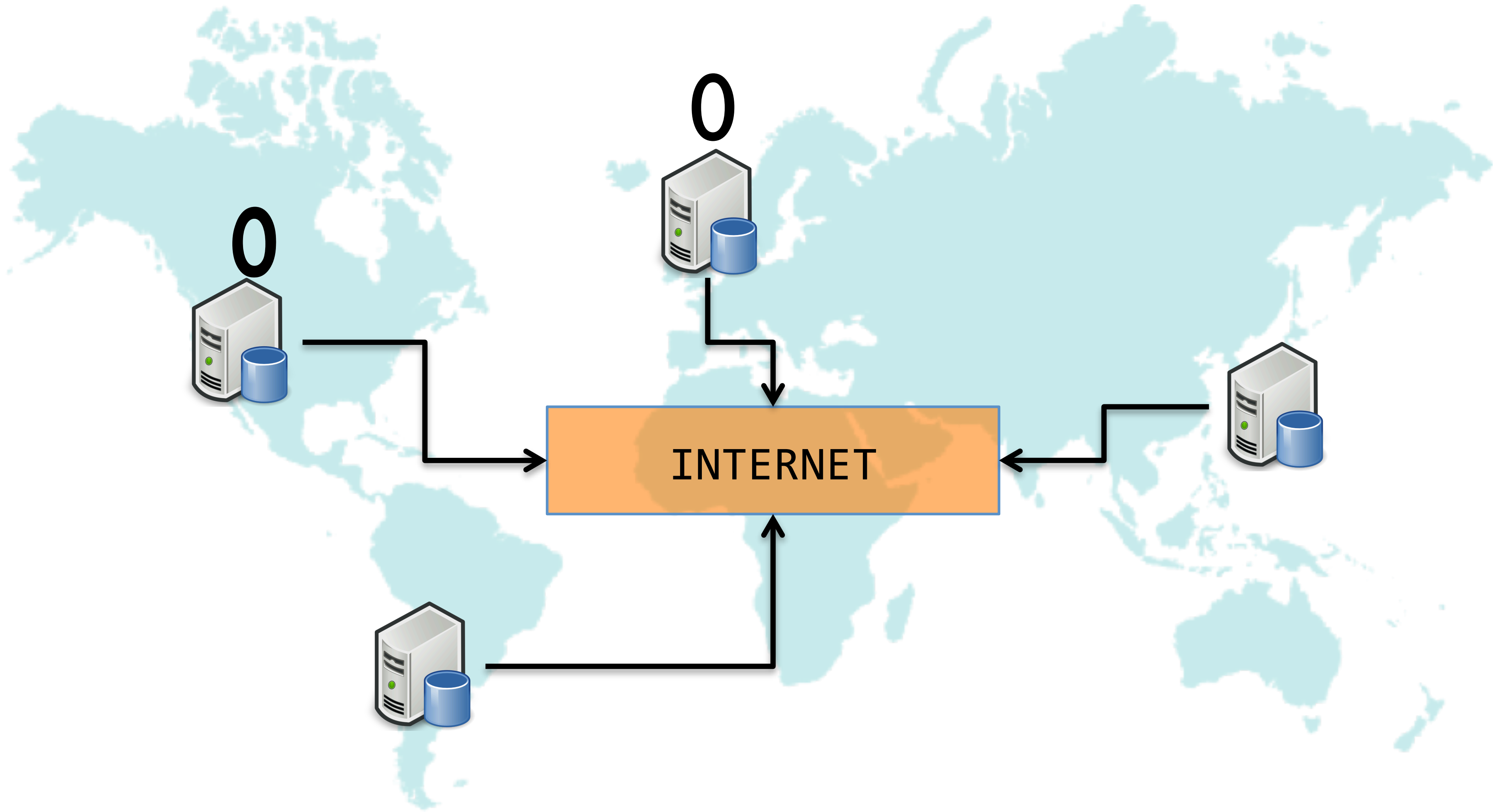
- Written in idiomatic style
- Composable

```
type counter_list = Counter.t list
```

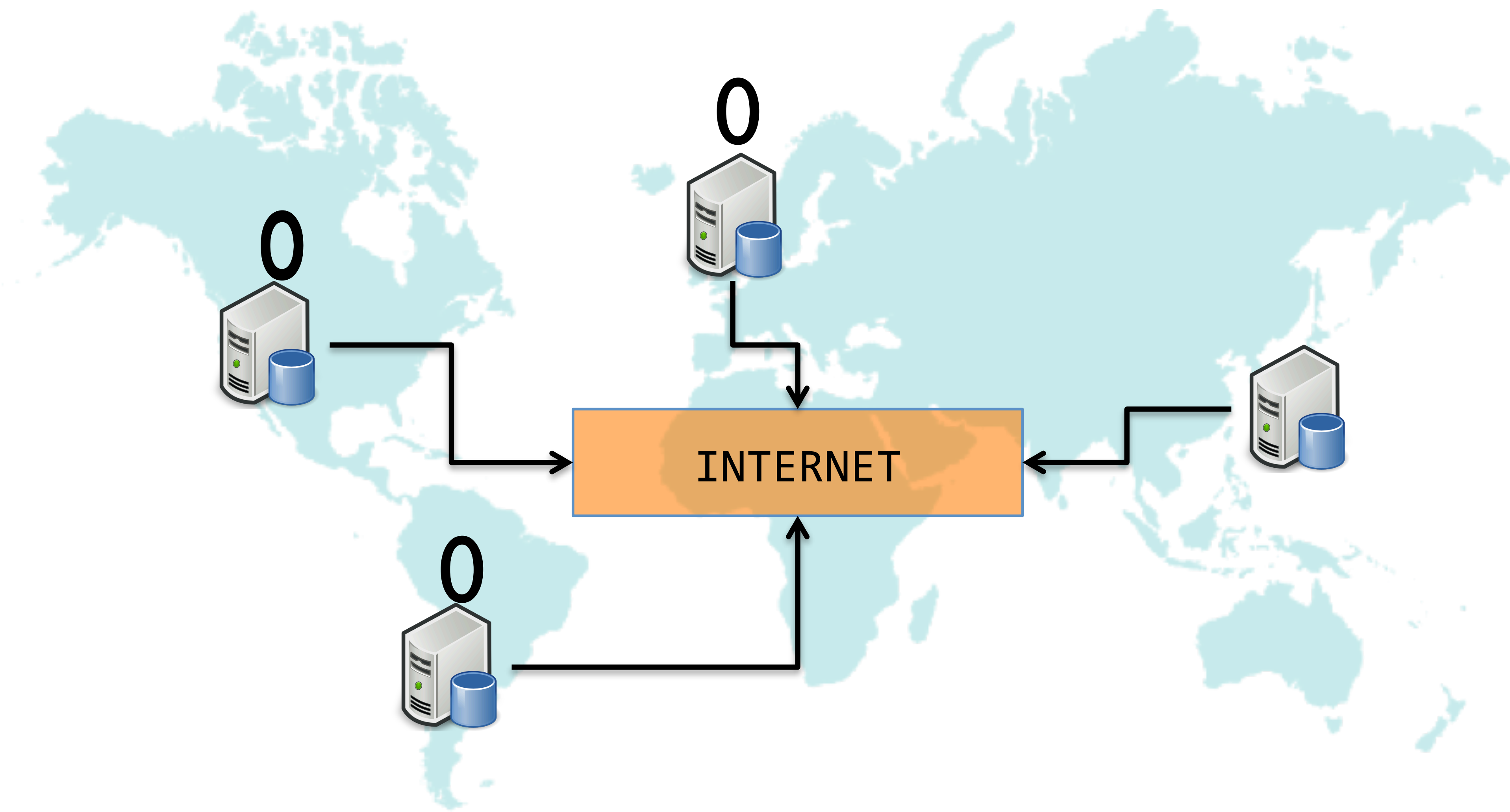
Replicated Counter



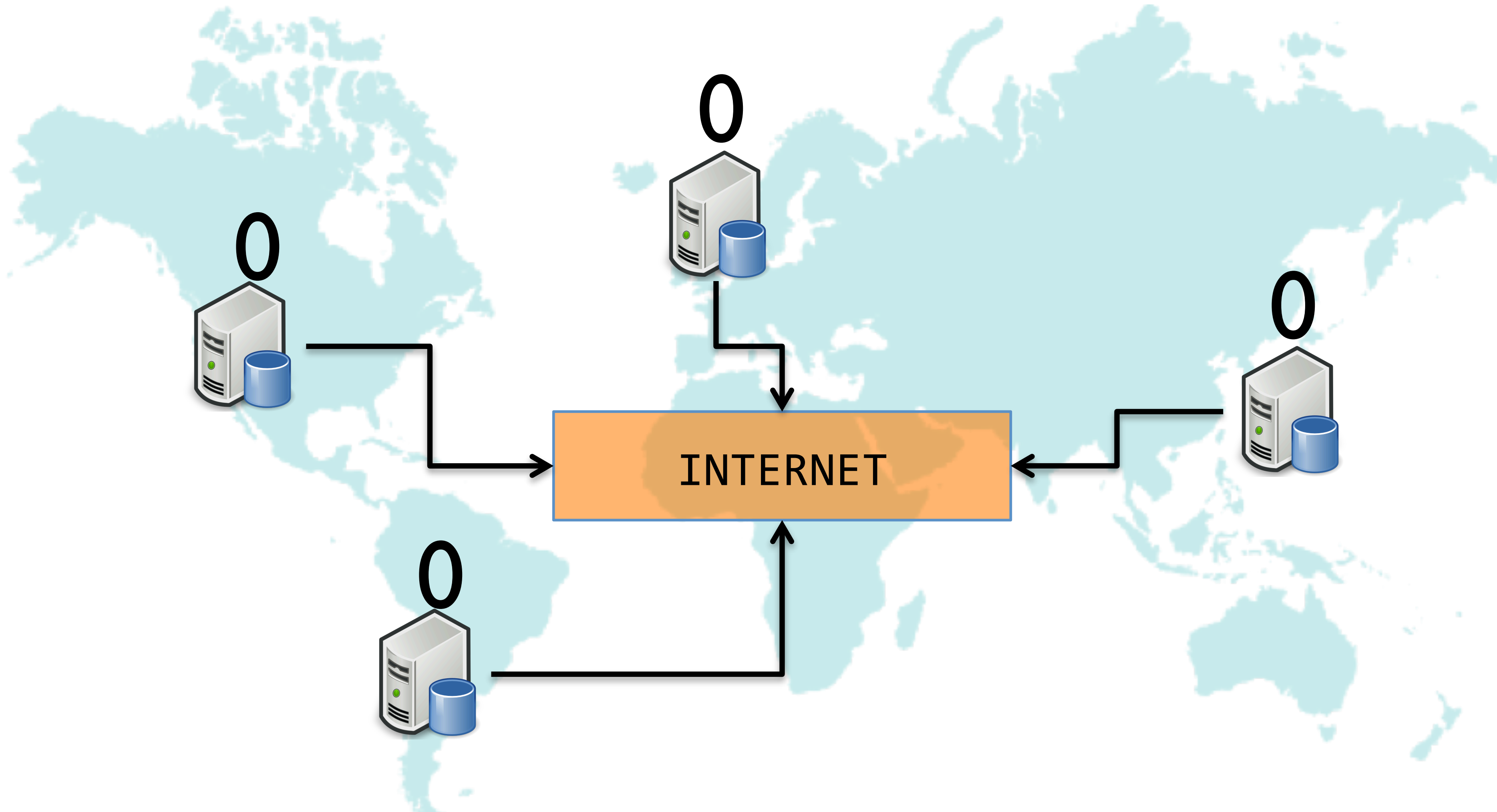
Replicated Counter



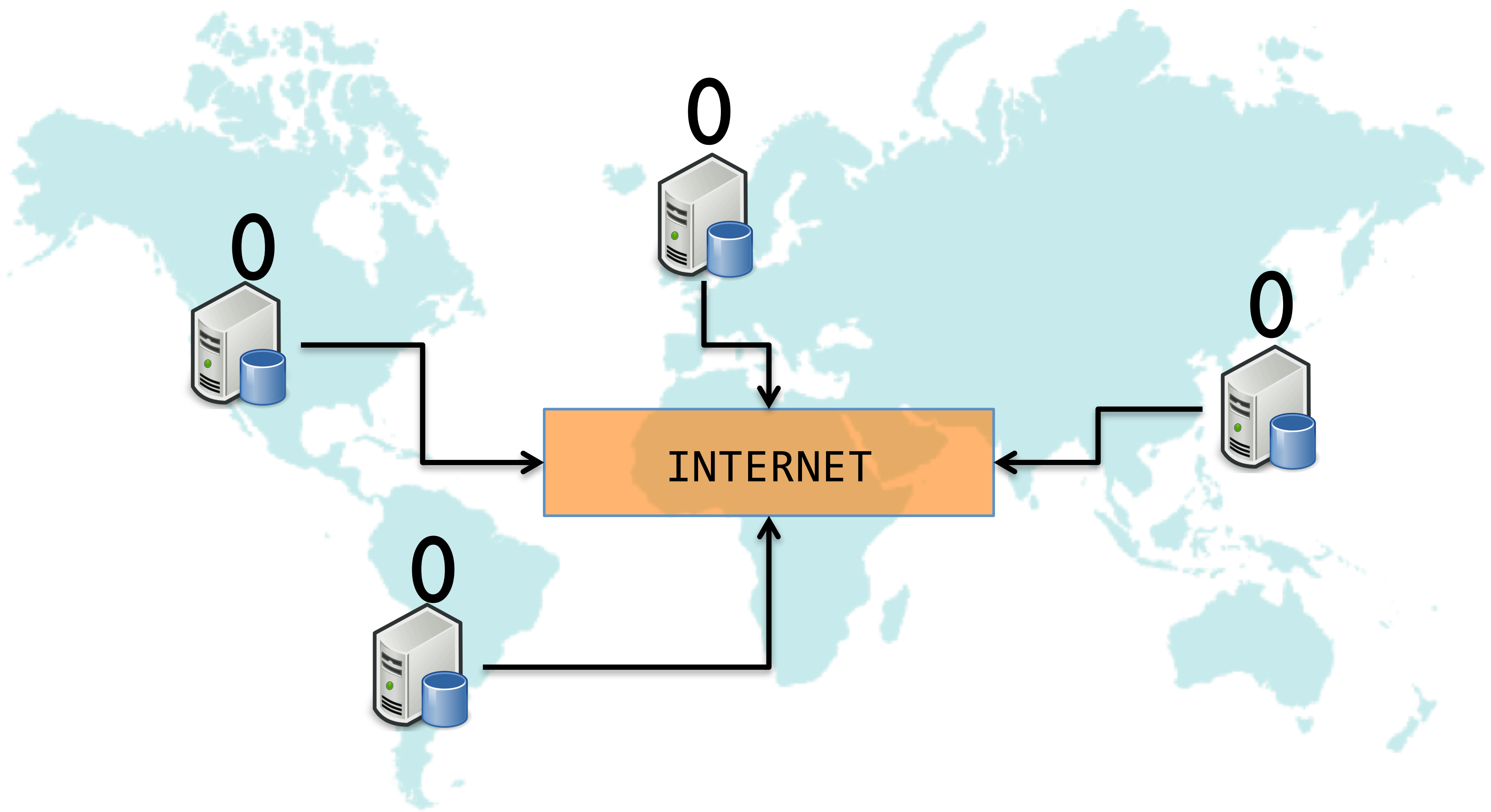
Replicated Counter



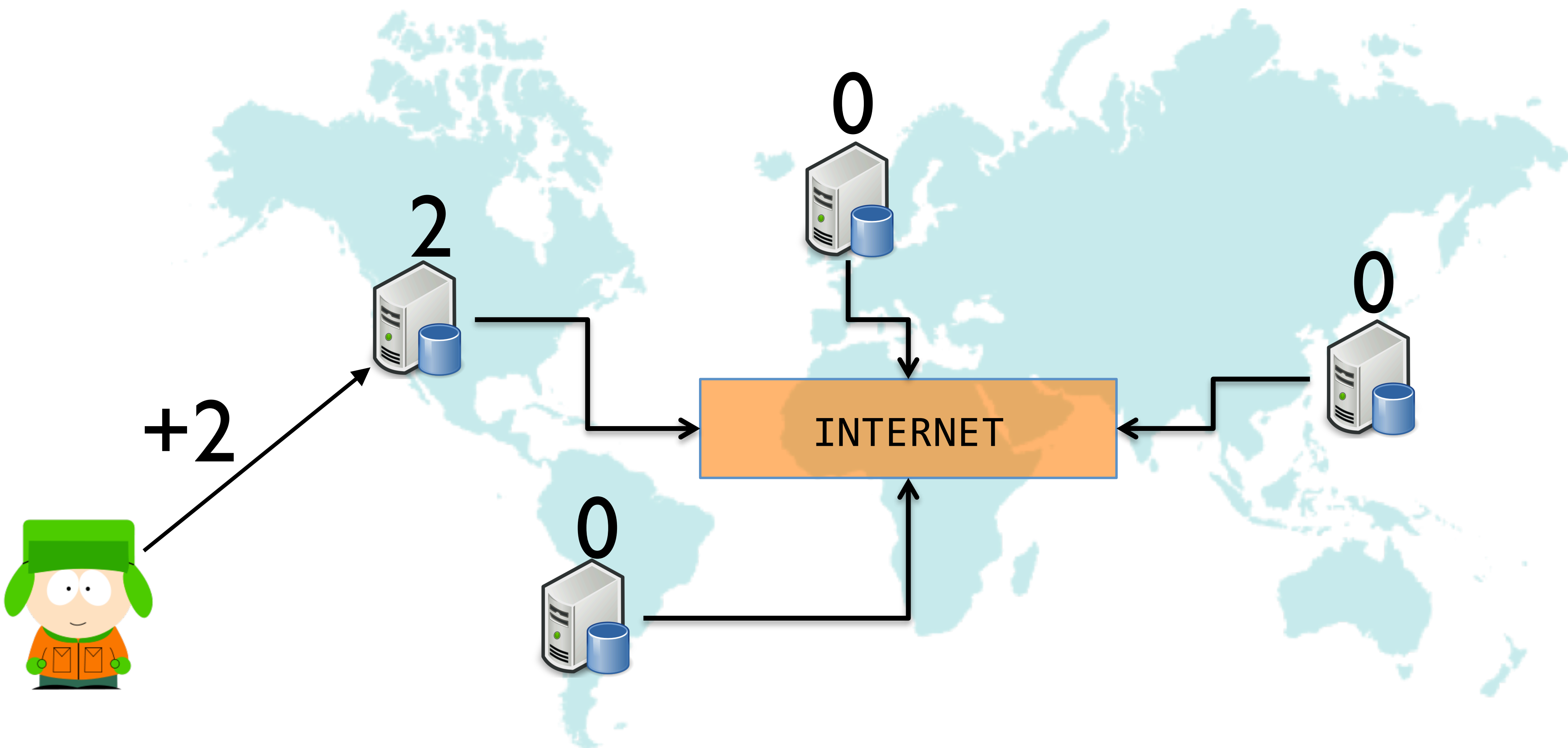
Replicated Counter



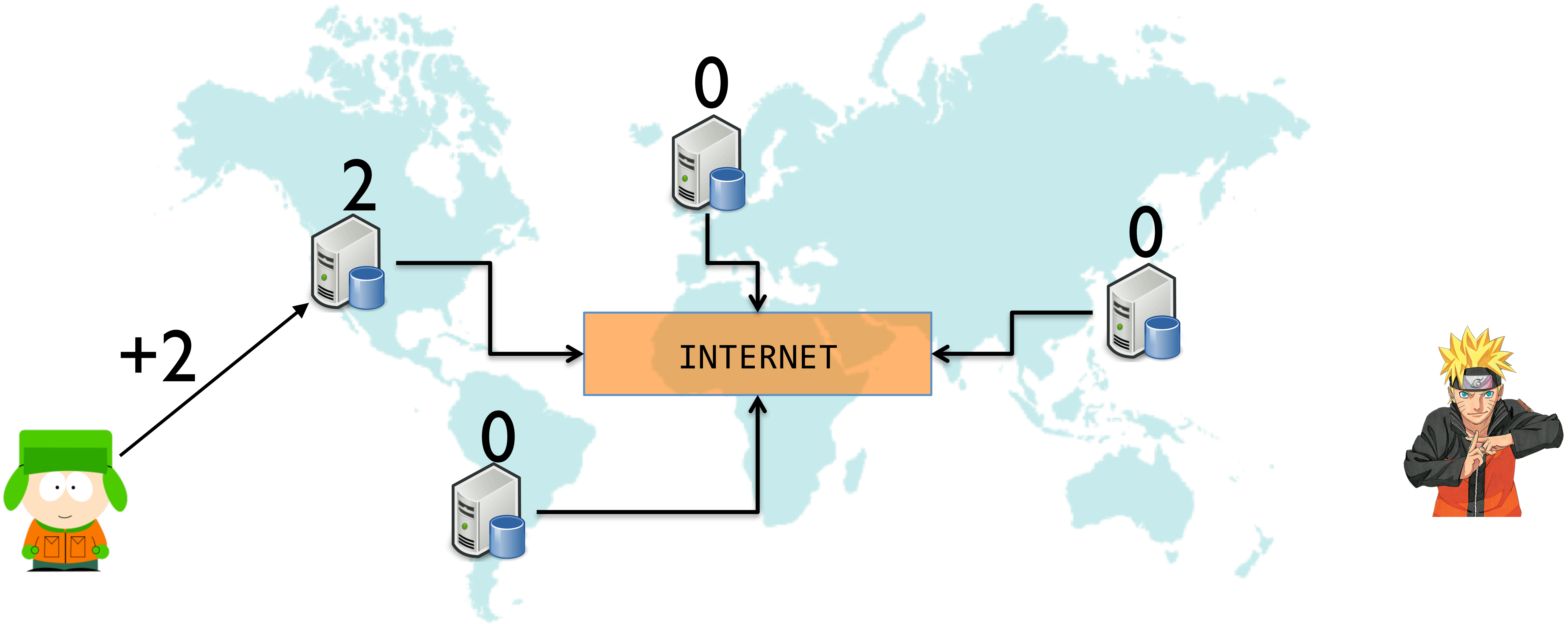
Replicated Counter



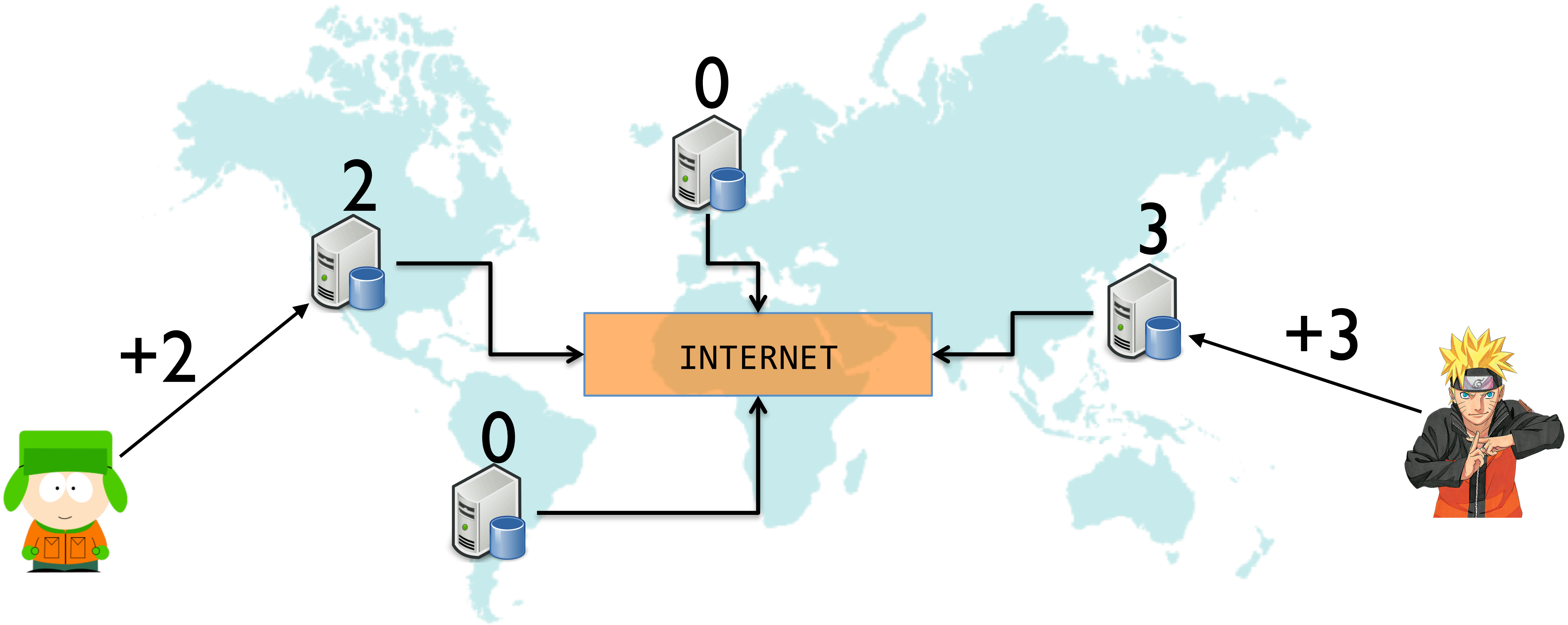
Replicated Counter



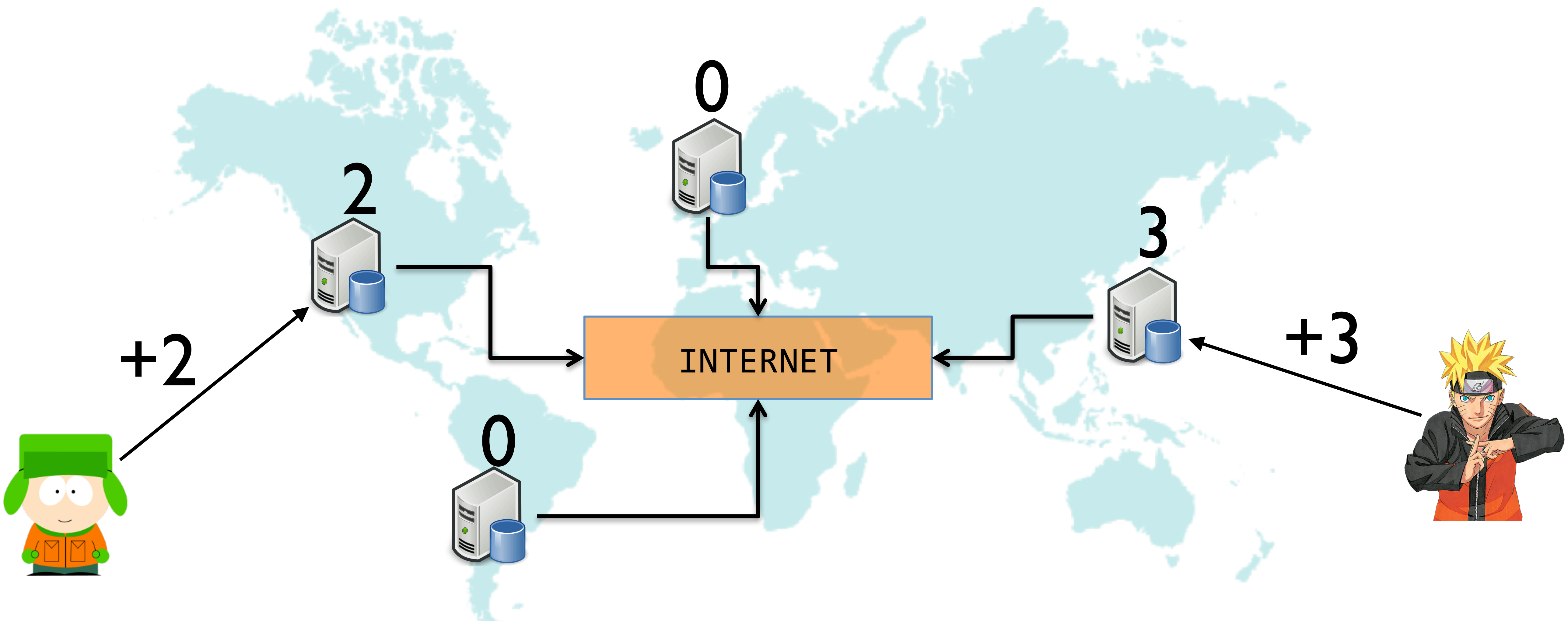
Replicated Counter



Replicated Counter

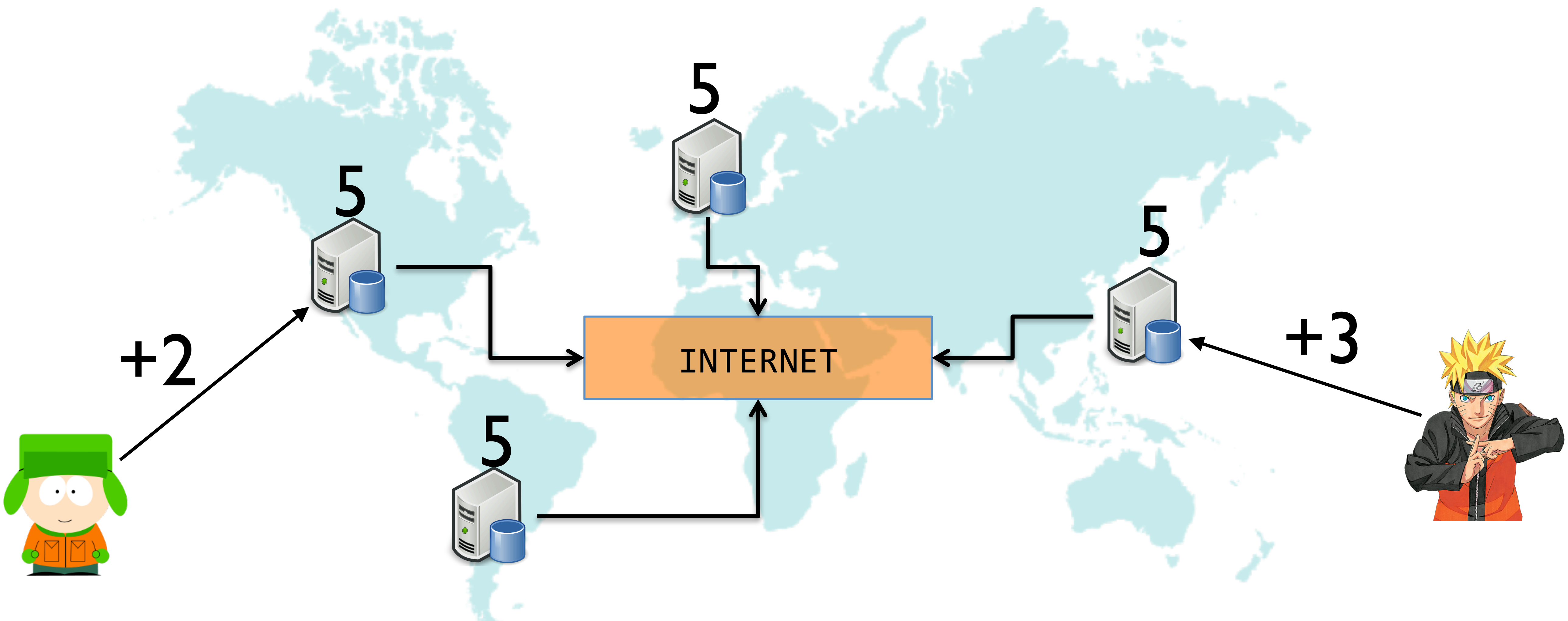


Replicated Counter



- ***Idea:*** Apply the local operations at all replicas

Replicated Counter



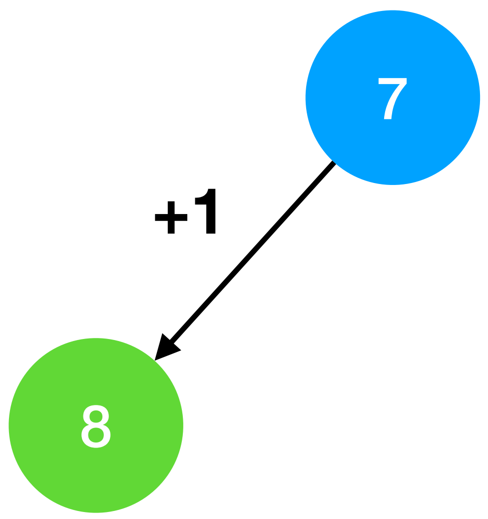
- ***Idea:*** Apply the local operations at all replicas

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```

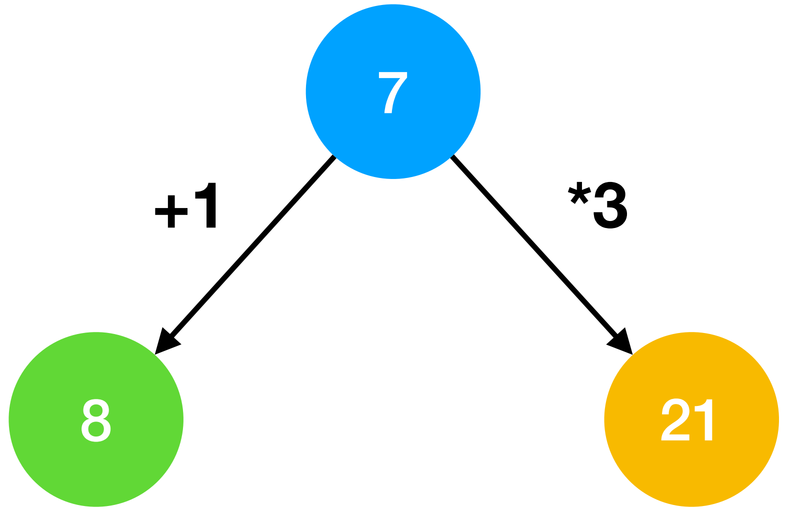
7

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```

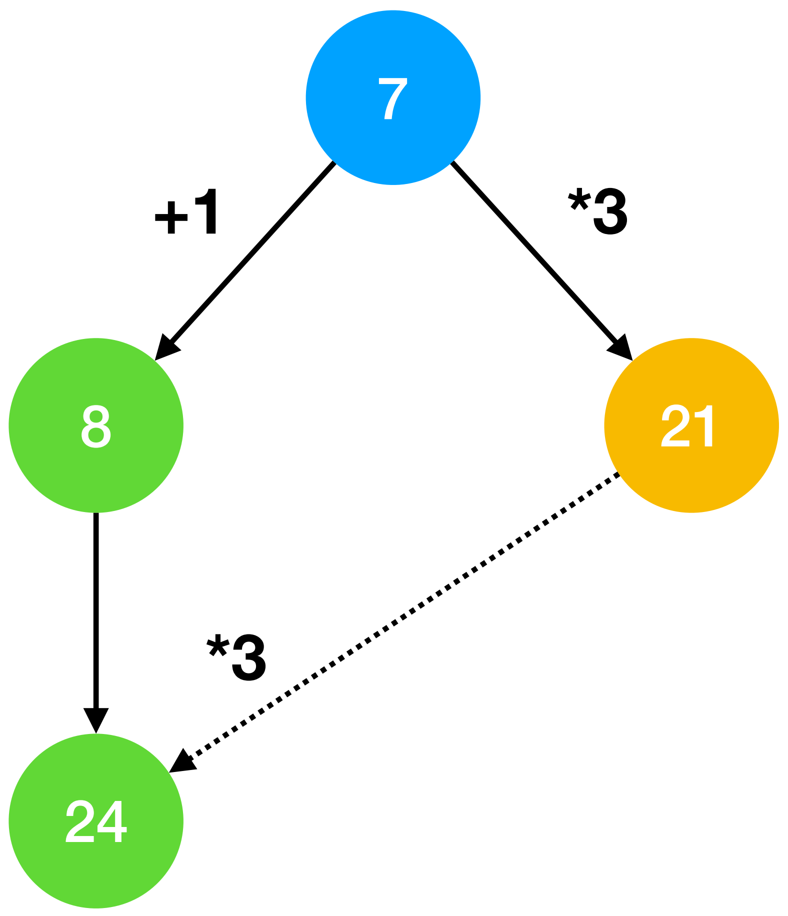
```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



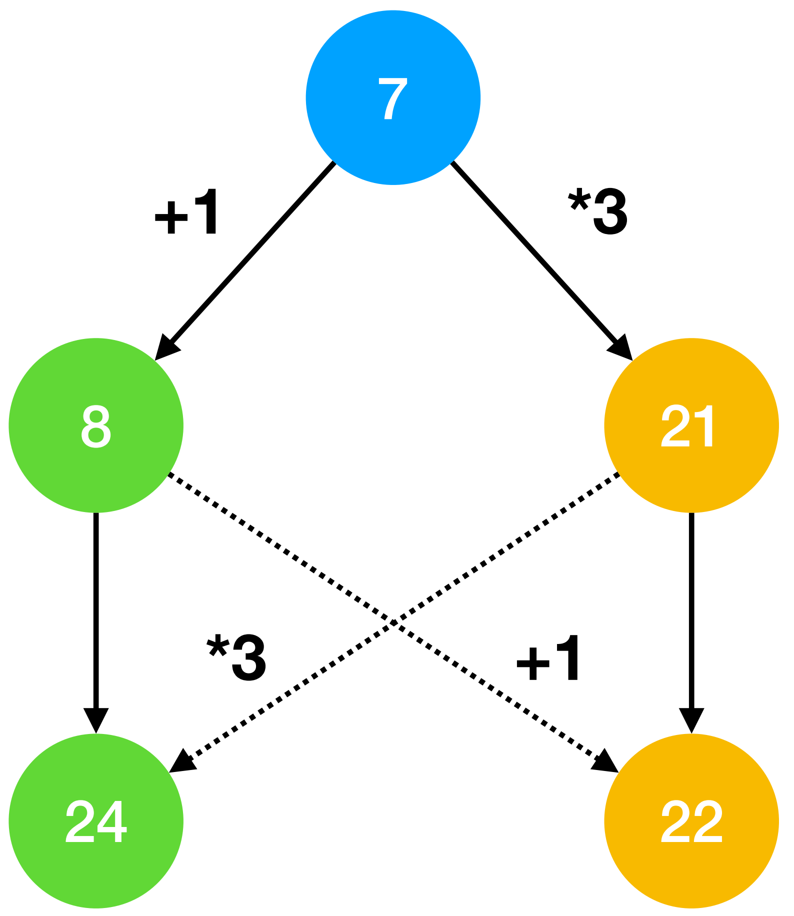
```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



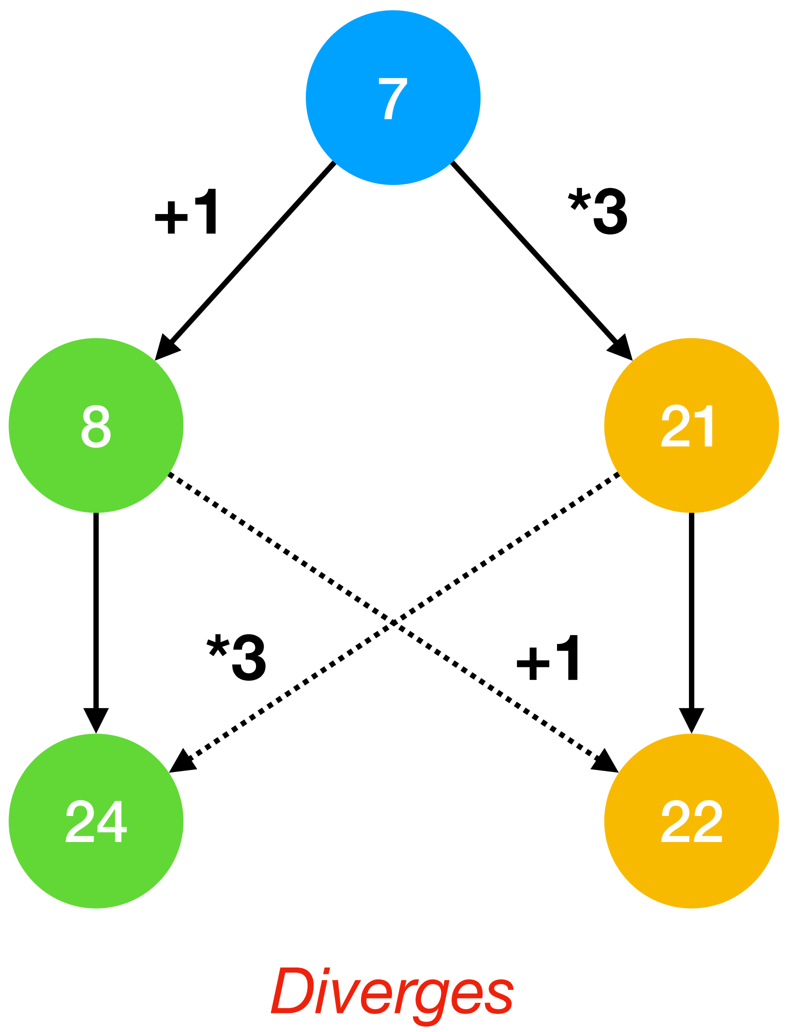
```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



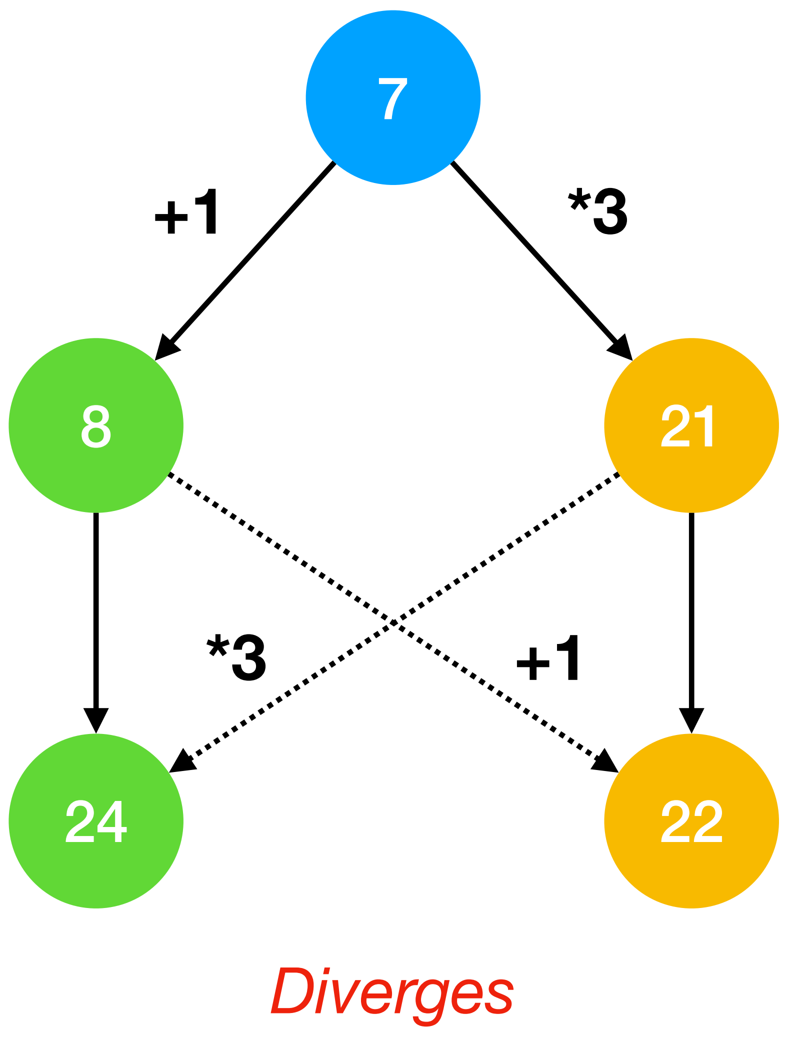
```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



Addition and multiplication do not commute

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```

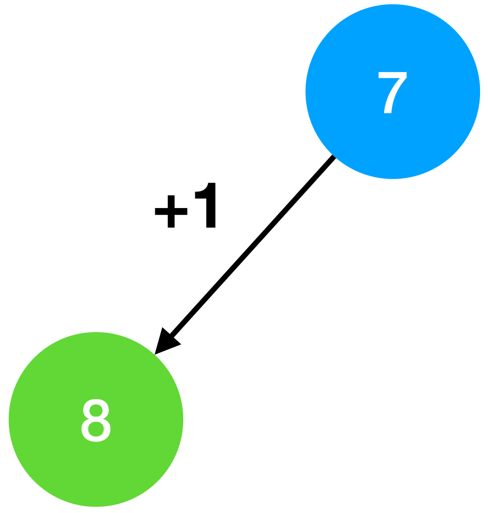
- ***Idea:*** Capture the effect of multiplication through the *commutative* addition operation

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```

7

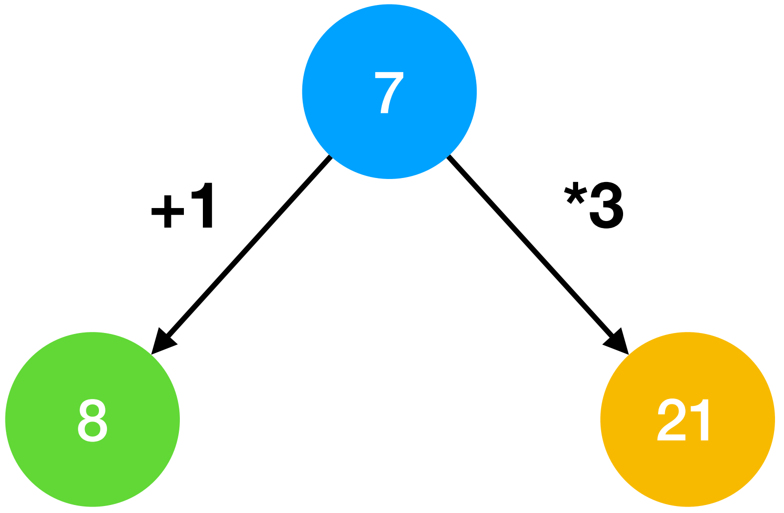
- ***Idea:*** Capture the effect of multiplication through the *commutative* addition operation

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



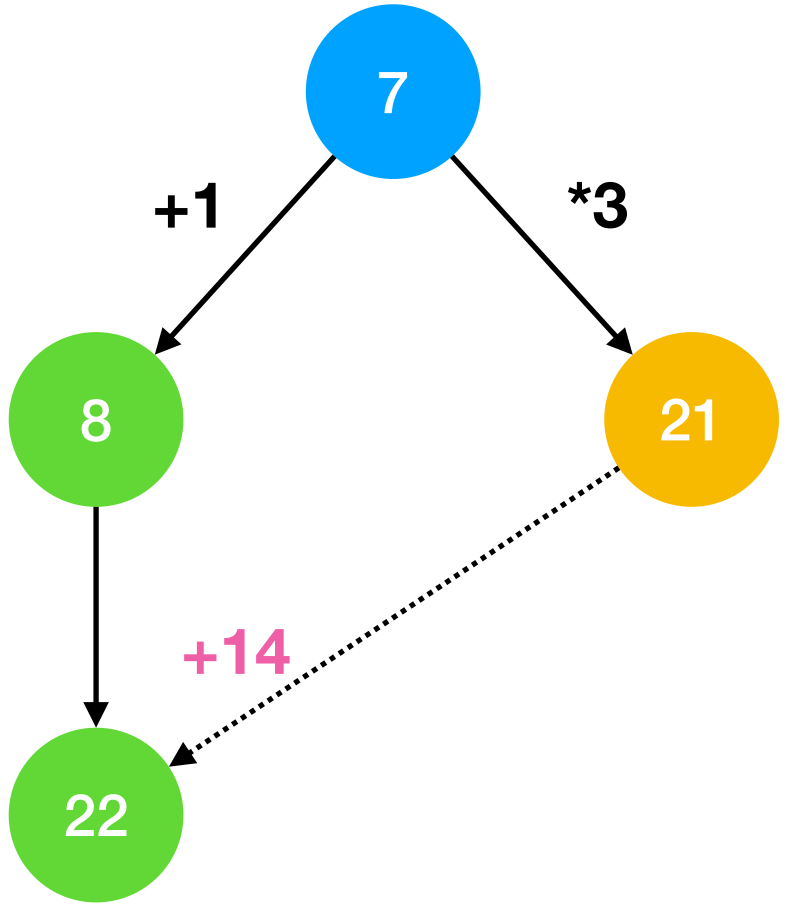
- ***Idea:*** Capture the effect of multiplication through the *commutative* addition operation

```ocaml
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



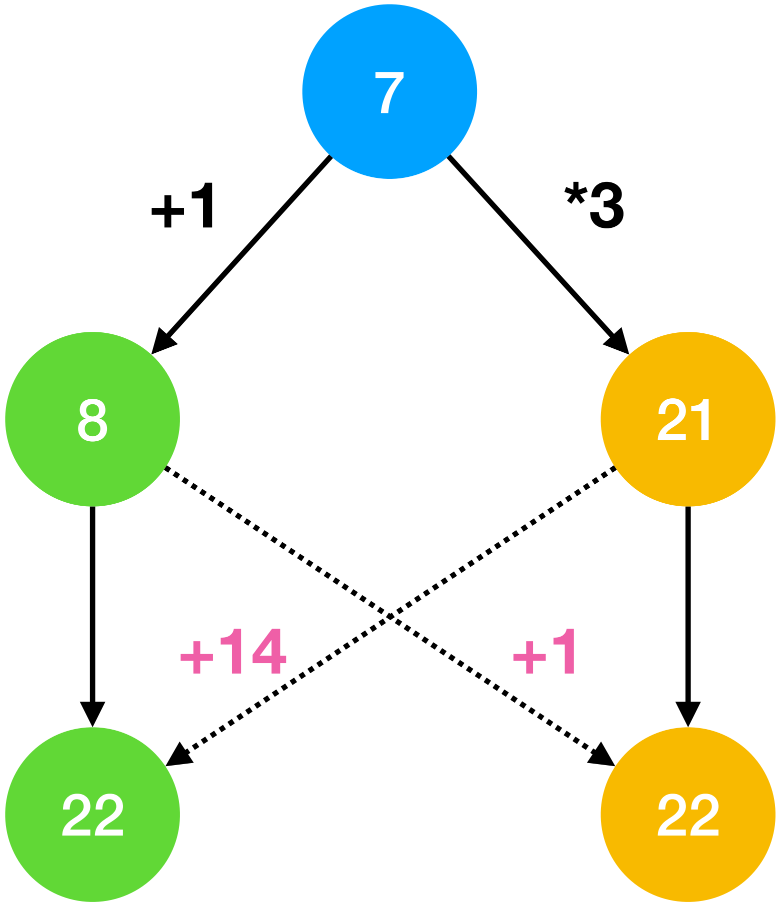
- ***Idea:*** Capture the effect of multiplication through the *commutative* addition operation

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



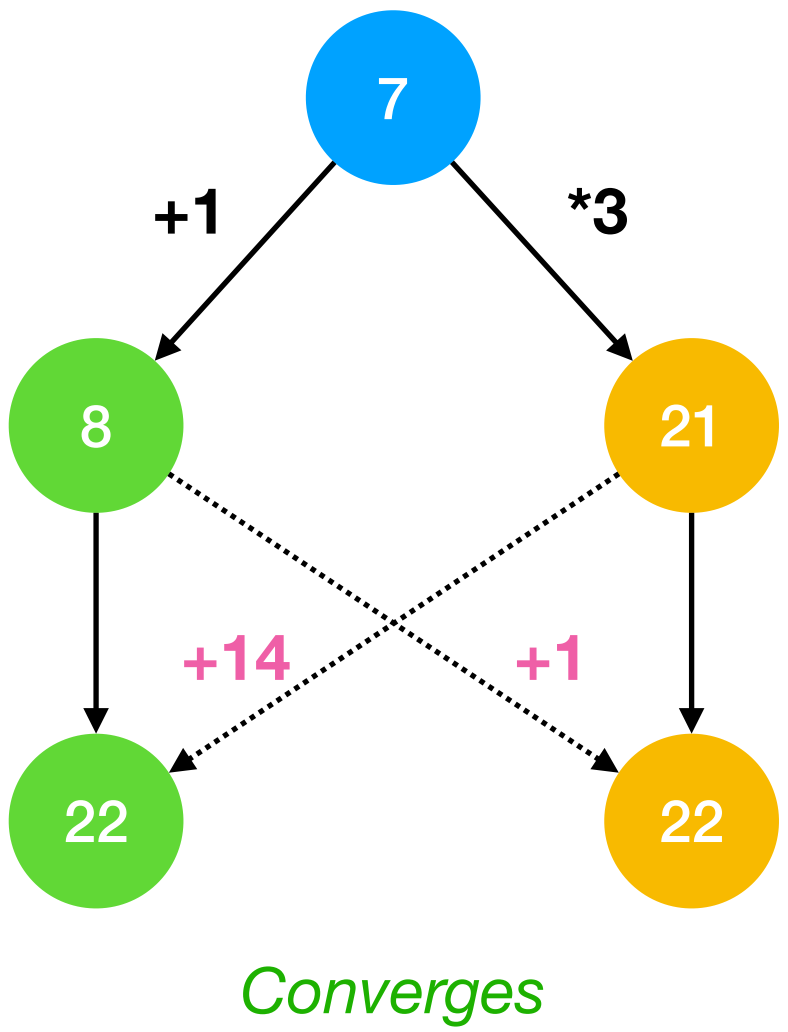
- ***Idea:*** Capture the effect of multiplication through the *commutative* addition operation

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



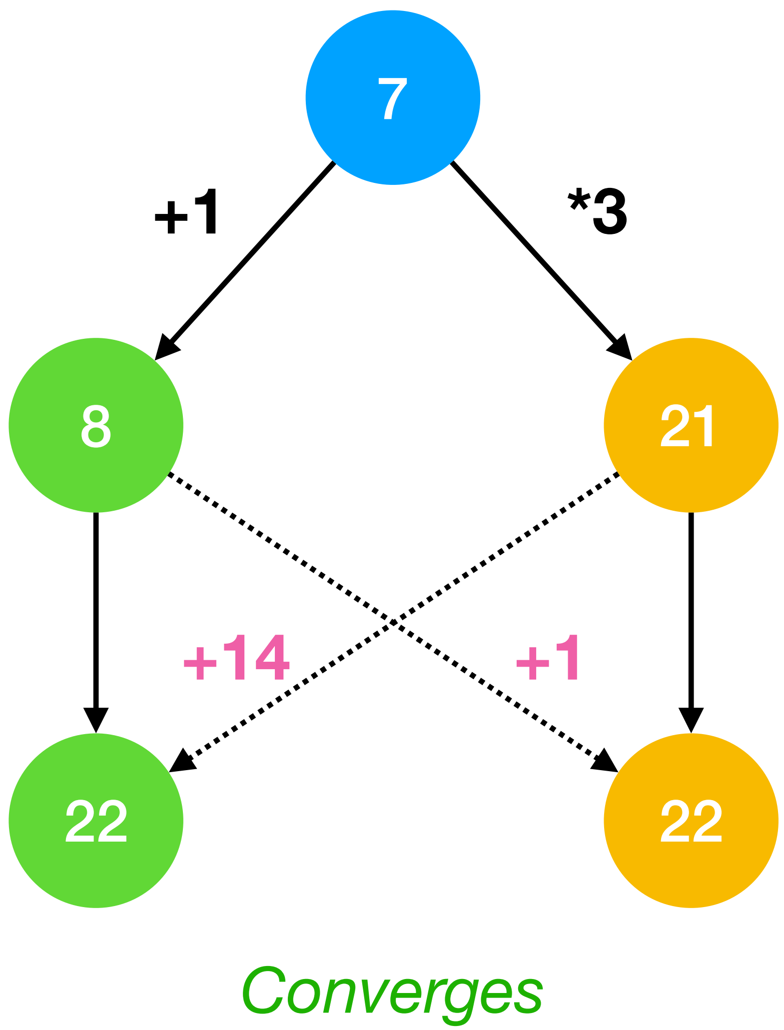
- ***Idea:*** Capture the effect of multiplication through the *commutative* addition operation

```ocaml
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



- ***Idea:*** Capture the effect of multiplication through the *commutative* addition operation

```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
```



- ***Idea:*** Capture the effect of multiplication through the *commutative* addition operation
- *CRDTs*

Convergent Replicated Data Types (CRDT)

Convergent Replicated Data Types (CRDT)

- CRDT is guaranteed to ensure *strong eventual consistency (SEC)*
  - ★ G-counters, PN-counters, OR-Sets, Graphs, Ropes, docs, sheets
  - ★ Simple interface for the clients of CRDTs

Convergent Replicated Data Types (CRDT)

- CRDT is guaranteed to ensure *strong eventual consistency (SEC)*
  - ★ G-counters, PN-counters, OR-Sets, Graphs, Ropes, docs, sheets
  - ★ Simple interface for the clients of CRDTs
- Need to reengineer every datatype to ensure SEC (commutativity)
  - ★ Do not mirror sequential counter parts => implementation & proof burden
  - ★ Do not compose!
    - ✦ `counter set` is not a composition of `counter` and `set` CRDTs

Can we *program & reason about* replicated data types as an extension of their sequential counterparts?

Can we *program & reason about* replicated data types as an extension of their sequential counterparts?

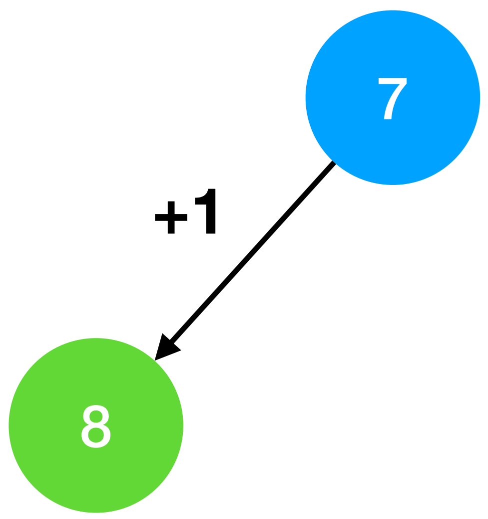
MRDT

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

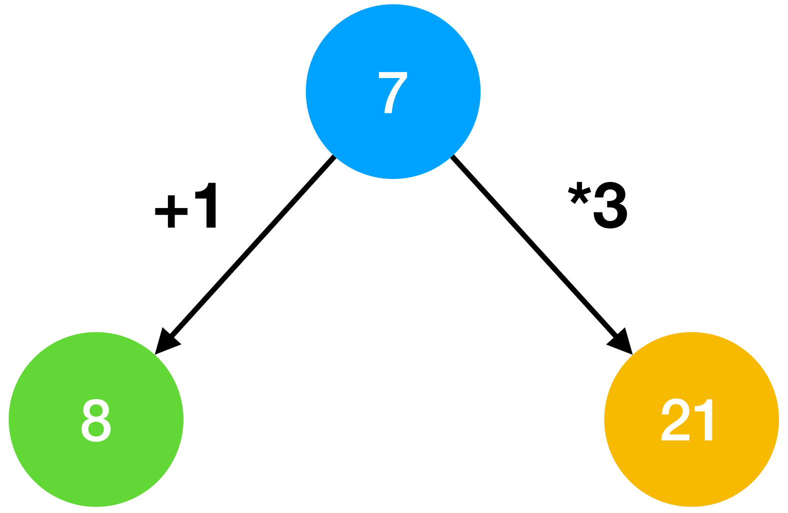
```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge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

7

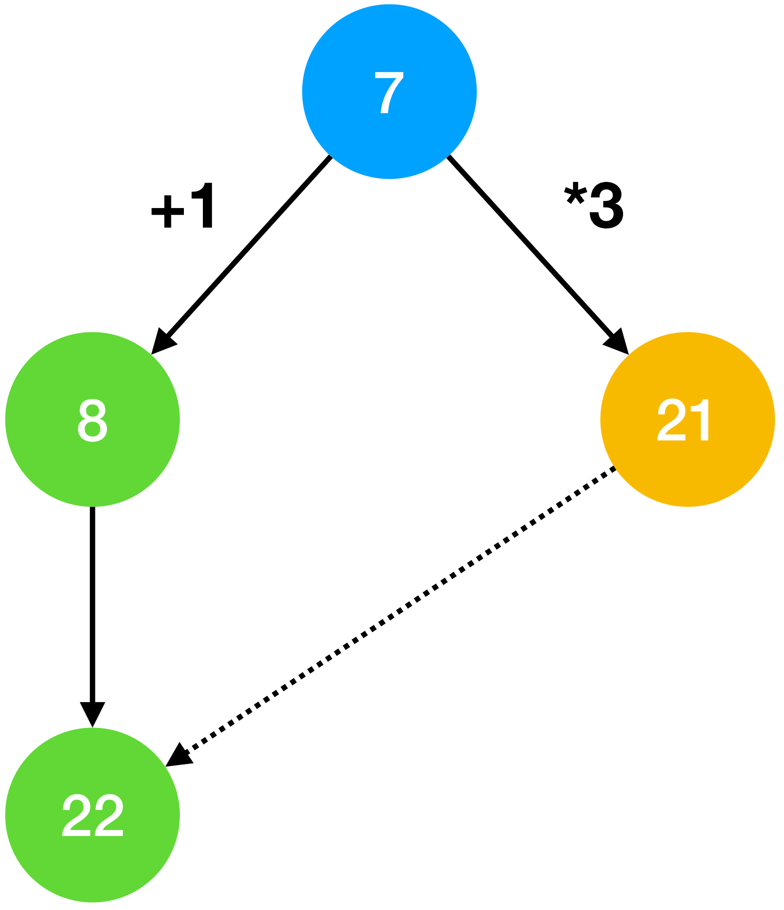
```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge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



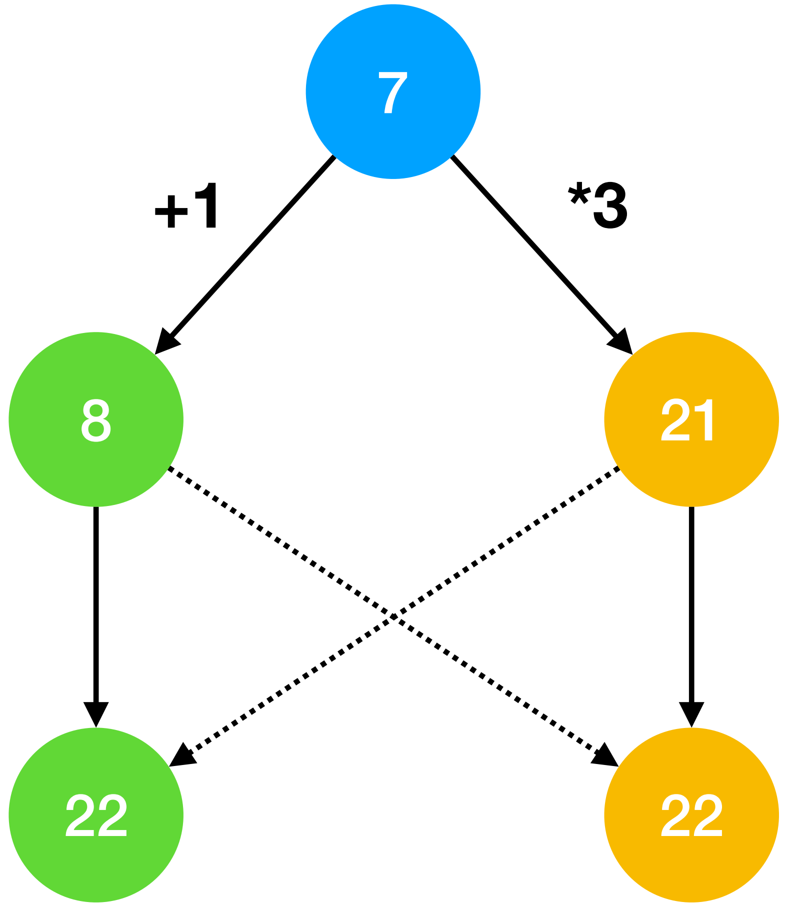
```ocaml
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let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



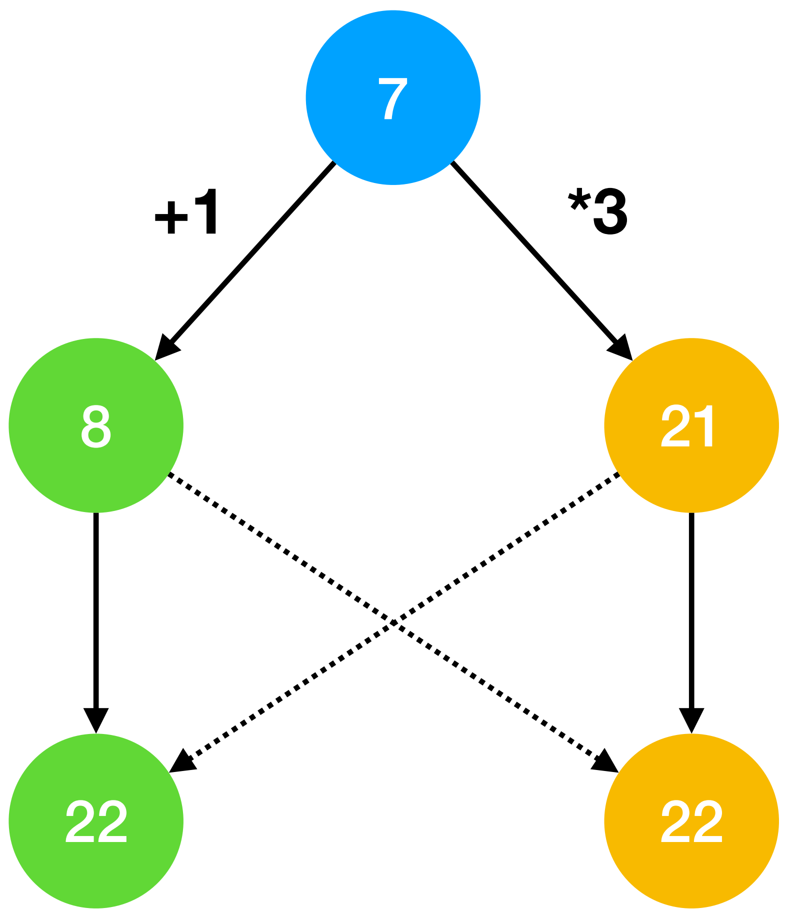
```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge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



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

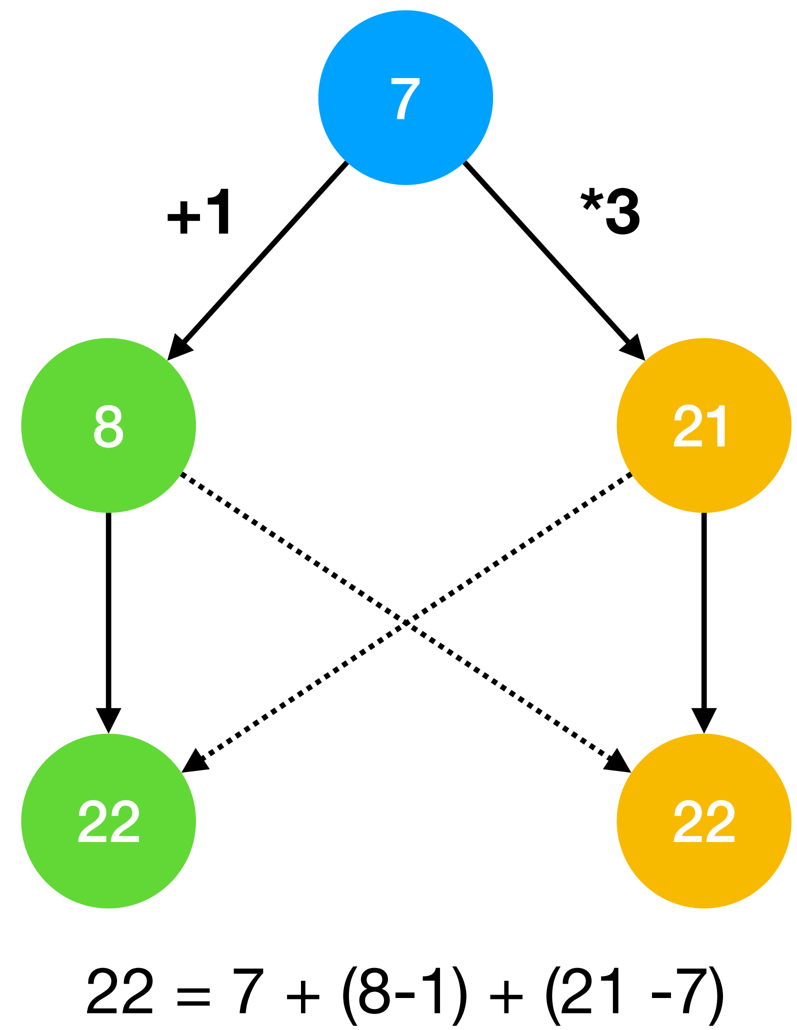


```
module Counter : sig
  type t
  val read : t -> int
  val add  : t -> int -> t
  val sub  : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge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



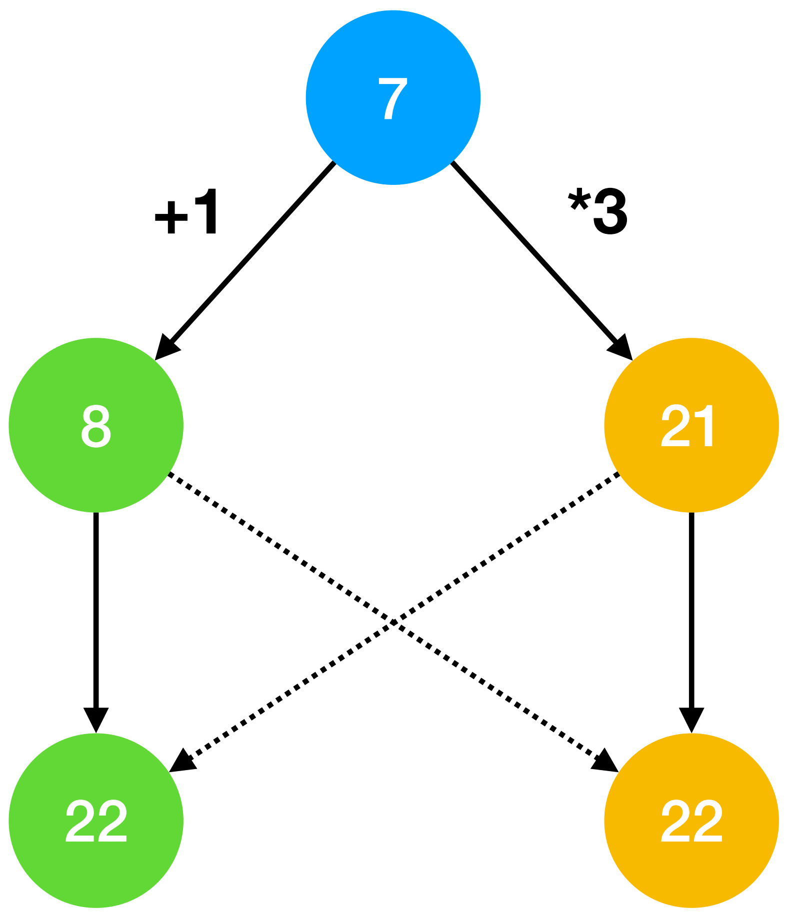
22 = 7 + (8-1) + (21 -7)

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



- 3-way merge function makes the counter suitable for distribution

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



22 = 7 + (8-1) + (21 -7)

- 3-way merge function makes the counter suitable for distribution
- Does not appeal to individual operations => independently extend data-type

Systems ➞ PL

Systems → PL

- CRDTs need to take care of systems level concerns such as *message loss, duplication and reordering*

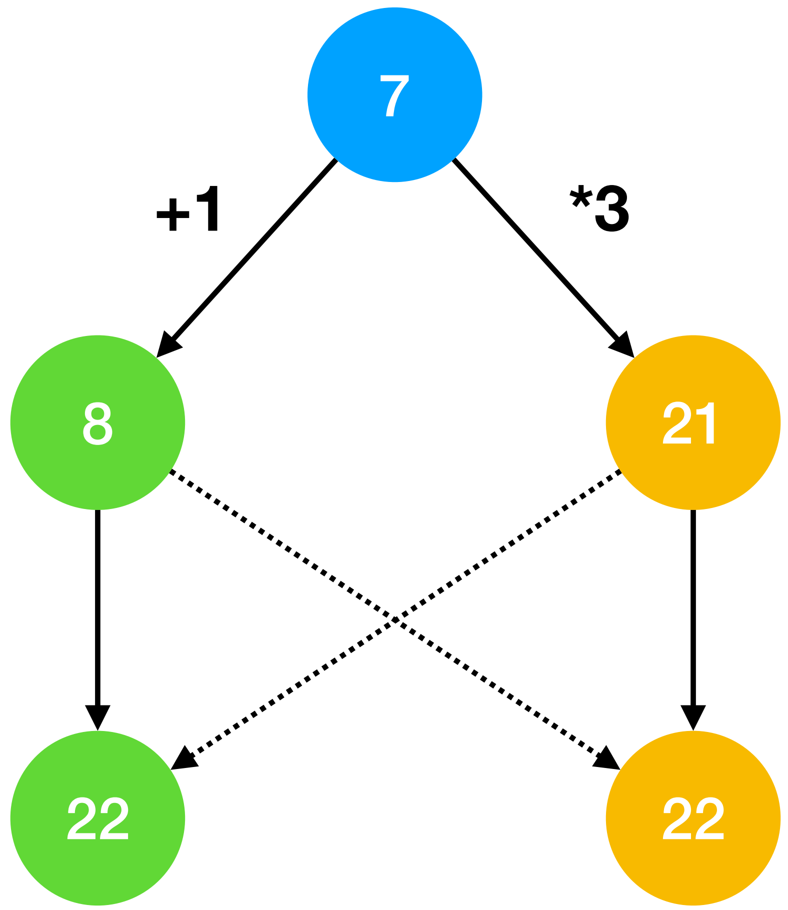
Systems → PL

- CRDTs need to take care of systems level concerns such as *message loss, duplication and reordering*
- 3-way merge is oblivious to these
  - ✦ By leaving those concerns to MRDT middleware

Systems → PL

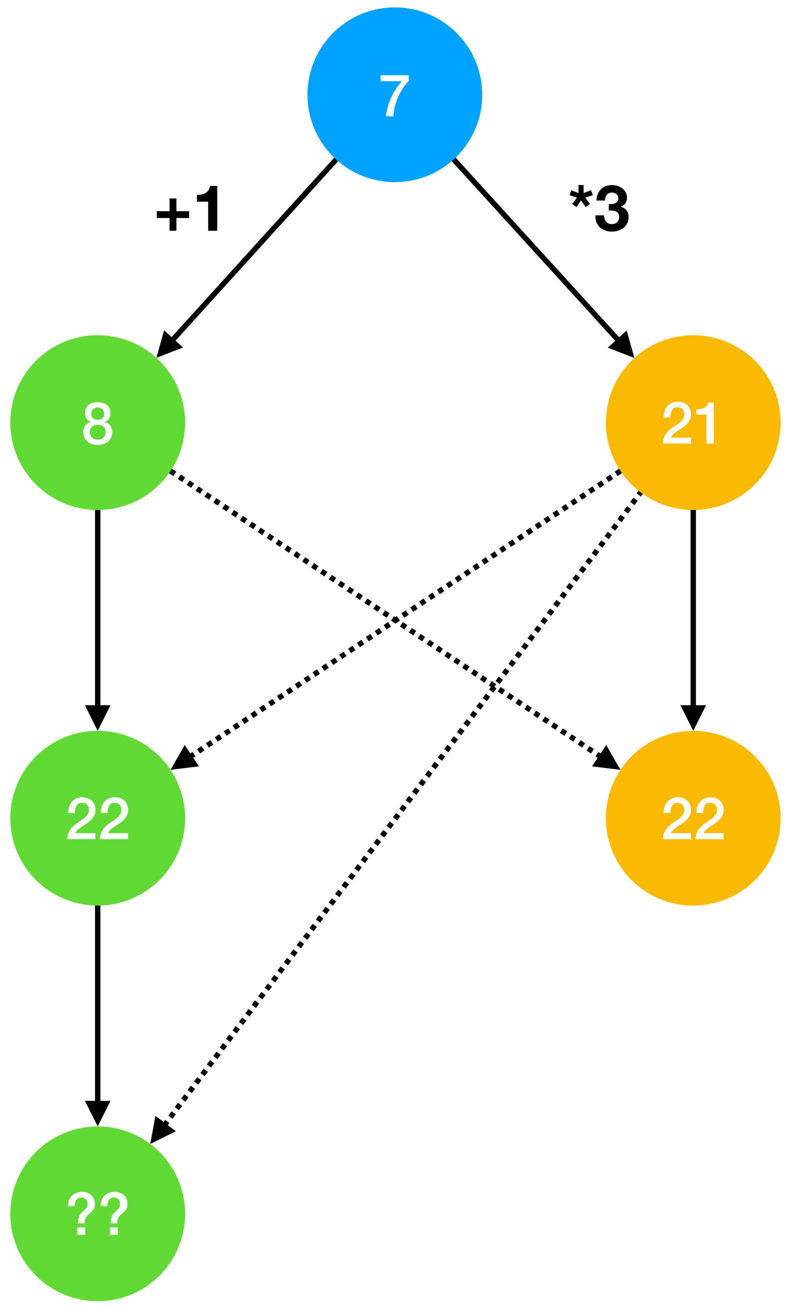
- CRDTs need to take care of systems level concerns such as *message loss, duplication and reordering*

- 3-way merge is oblivious to these
  - ✦ By leaving those concerns to MRDT middleware



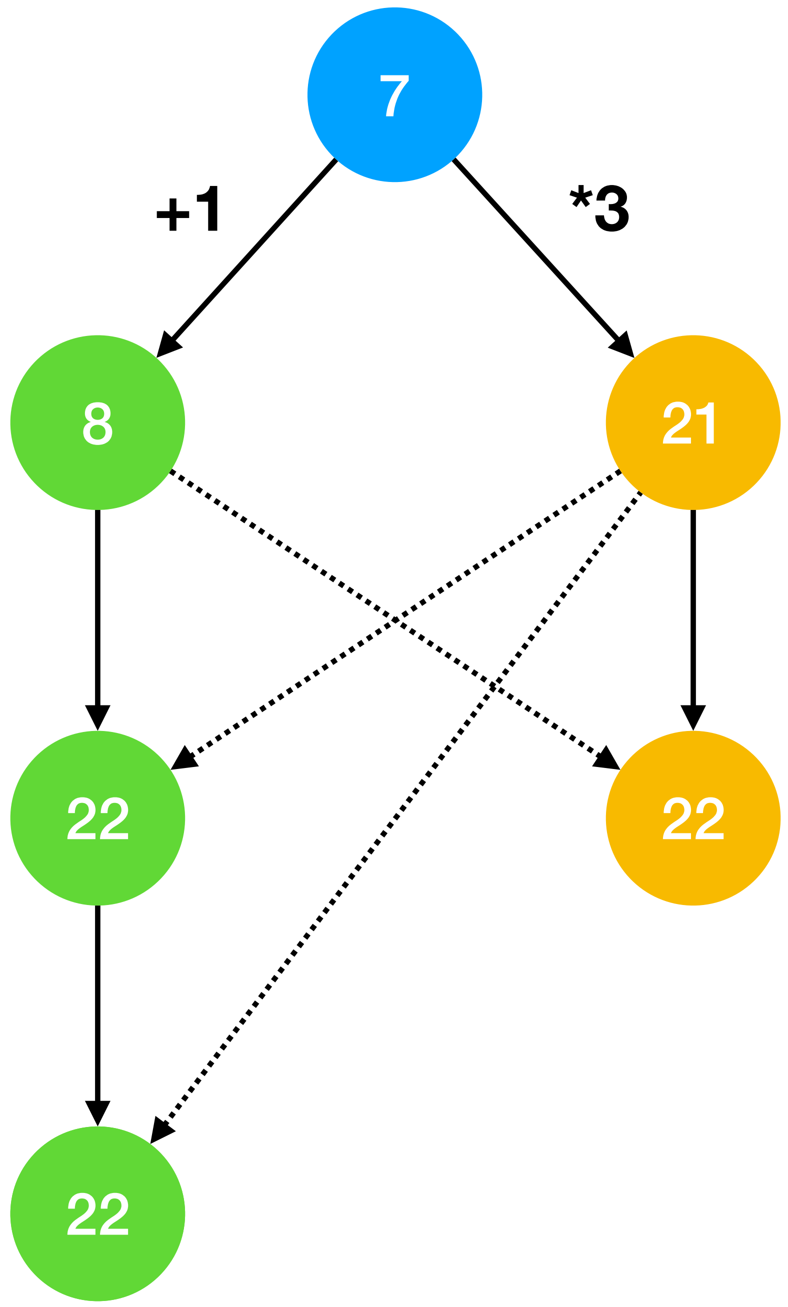
Systems → PL

- CRDTs need to take care of systems level concerns such as *message loss, duplication and reordering*
- 3-way merge is oblivious to these
  - ✦ By leaving those concerns to MRDT middleware



Systems → PL

- CRDTs need to take care of systems level concerns such as *message loss, duplication and reordering*
- 3-way merge is oblivious to these
  - ✦ By leaving those concerns to MRDT middleware



22 = 21 + (21-21) + (22 -21)

Does the 3-way merge idea generalise?

Does the 3-way merge idea generalise?

Sort of

Observed-Removed Set

Observed-Removed Set

- OR-set — *add-wins* when there is a concurrent add and remove of the same element

Observed-Removed Set

- OR-set — *add-wins* when there is a concurrent add and remove of the same element

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

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

Observed-Removed Set

- OR-set — *add-wins* when there is a concurrent add and remove of the same element

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

{1}

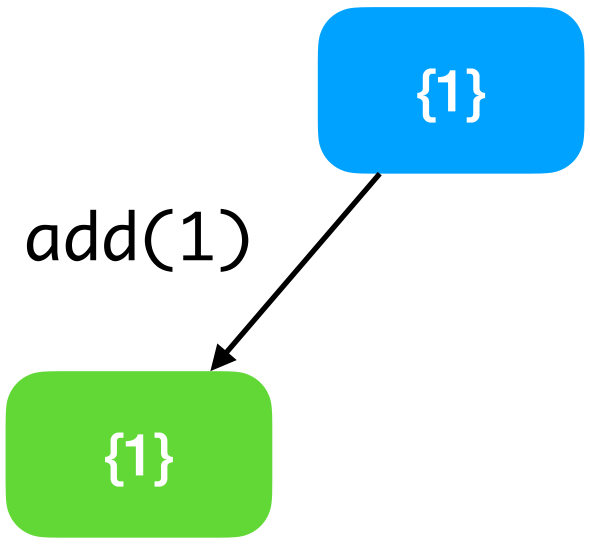
Kaki et al. “Mergeable Replicated Data Types”, OOPSLA 2019

Observed-Removed Set

- OR-set — *add-wins* when there is a concurrent add and remove of the same element

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

Kaki et al. “Mergeable Replicated Data Types”, OOPSLA 2019

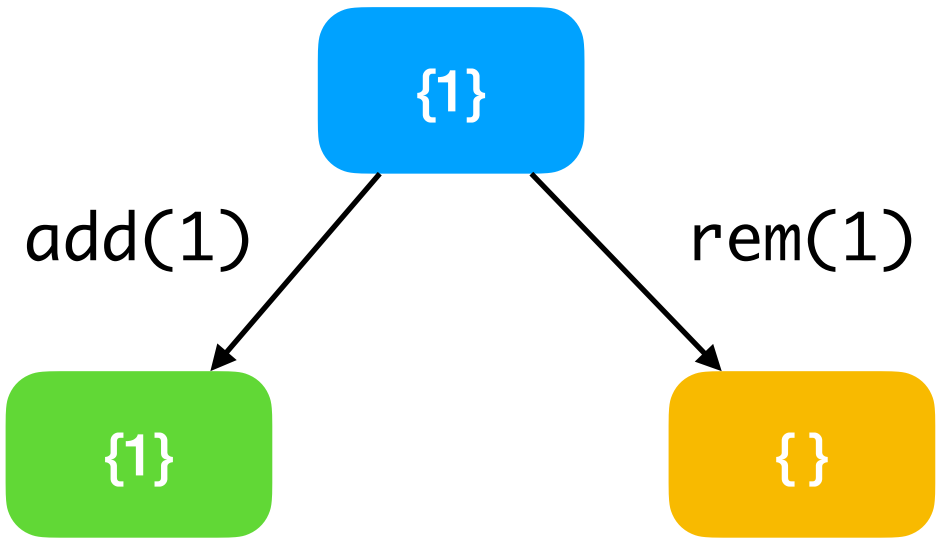


Observed-Removed Set

- OR-set — *add-wins* when there is a concurrent add and remove of the same element

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

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

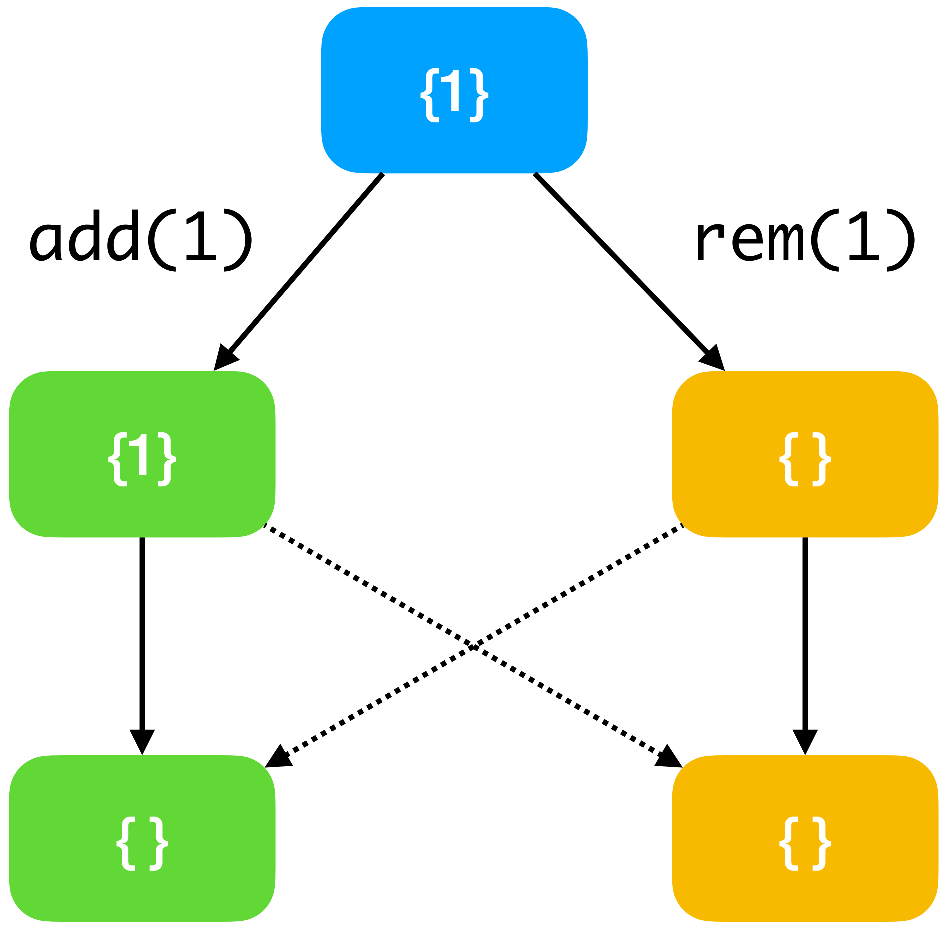


Observed-Removed Set

- OR-set — *add-wins* when there is a concurrent add and remove of the same element

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

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



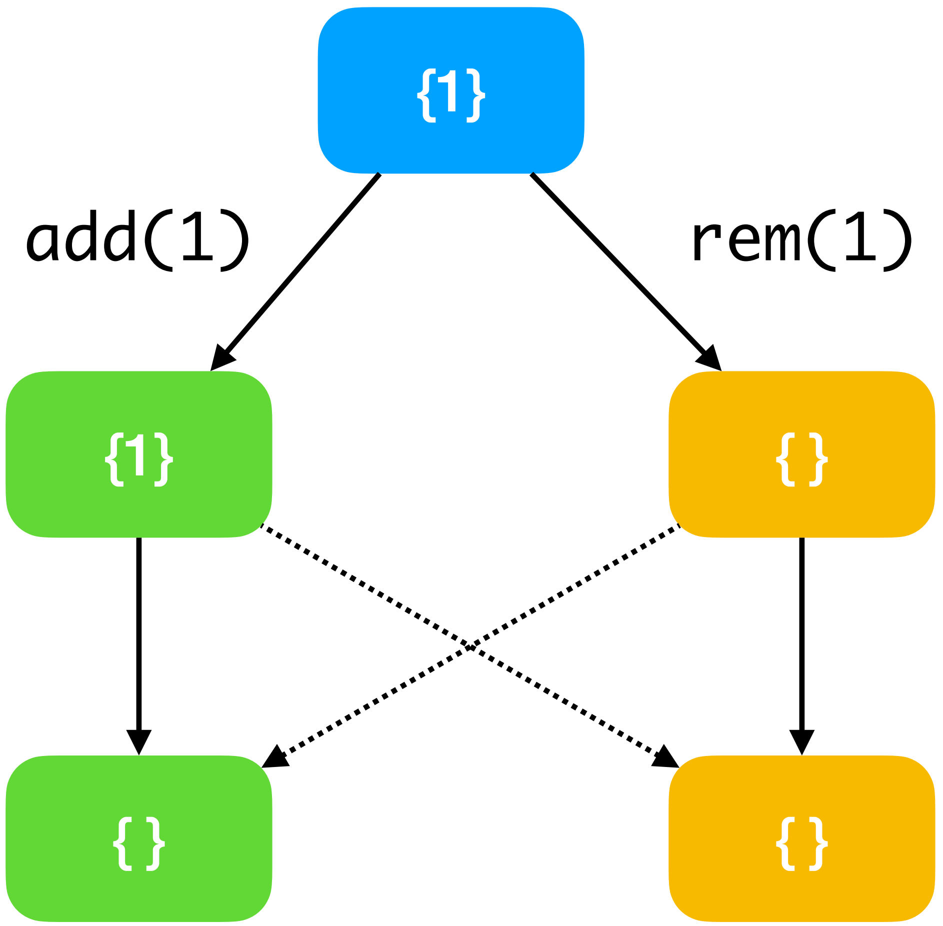
$$
\begin{aligned}
& \{\,\} \cup (\{1\} - \{1\}) \cup (\{\,\} - \{1\}) \\
= & \{\,\} \cup \{\,\} \cup \{\,\} \\
= & \{\,\} \ \textit{(expected \{1\})}
\end{aligned}
$$

Observed-Removed Set

- OR-set — *add-wins* when there is a concurrent add and remove of the same element

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

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



$\{\} \cup (\{1\} - \{1\}) \cup (\{\} - \{1\})$

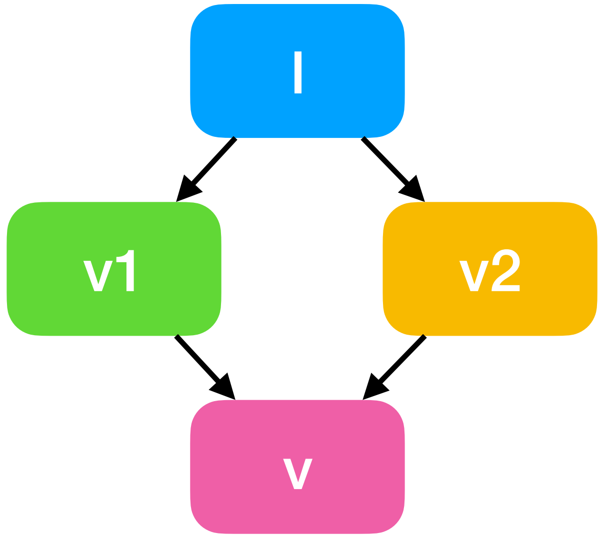
$= \{\} \cup \{\} \cup \{\}$

$= \{\}$ *(expected {1})*

- Convergence is not sufficient; *Intent* is not preserved

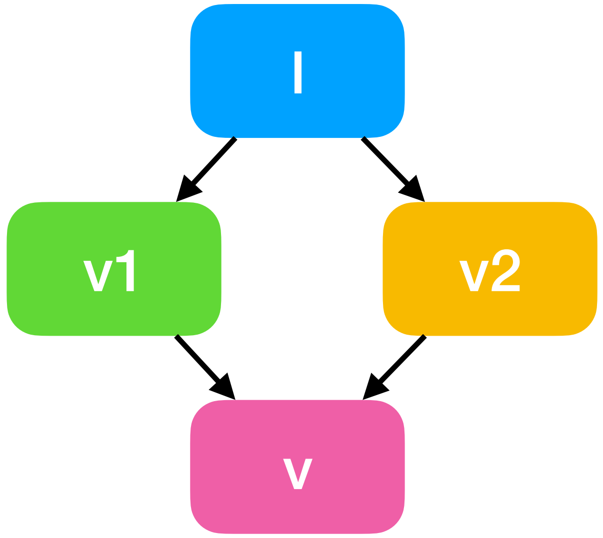
Concretising Intent

- Intent is a woolly term
  - ★ *How can we formalise the intent of operations on a data structure?*



Concretising Intent

- Intent is a woolly term
  - ★ *How can we formalise the intent of operations on a data structure?*
- We need
  - ★ A *formal language* to specify the *intent* of an RDT
  - ★ *Mechanization* to bridge the air gap between specification and implementation due to distributed system complexity



Peepul — Certified MRDTs

Peepul — Certified MRDTs

- An F* library implementing and proving MRDTs
  - ★ https://github.com/prismlab/peepul

Peepul — Certified MRDTs

- An F* library implementing and proving MRDTs
  - ★ https://github.com/prismlab/peepul
- Specification language is event-based
  - ★ Burckhardt et al. "Replicated Data Types: Specification, Verification and Optimality", POPL 2014

Peepul — Certified MRDTs

- An F* library implementing and proving MRDTs
  - ★ https://github.com/prismlab/peepul
- 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*

Peepul — Certified MRDTs

- An F* library implementing and proving MRDTs
  - ★ https://github.com/prismlab/peepul
- 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*
- Composition of MRDTs and their proofs!

Peepul — Certified MRDTs

- An F* library implementing and proving MRDTs
  - ★ https://github.com/prismlab/peepul
- 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*
- Composition of MRDTs and their proofs!
- Extracted RDTs are compatible with Irmin — a Git-like distributed database

Fixing OR-Set

- Discriminate duplicate additions by associating a unique id

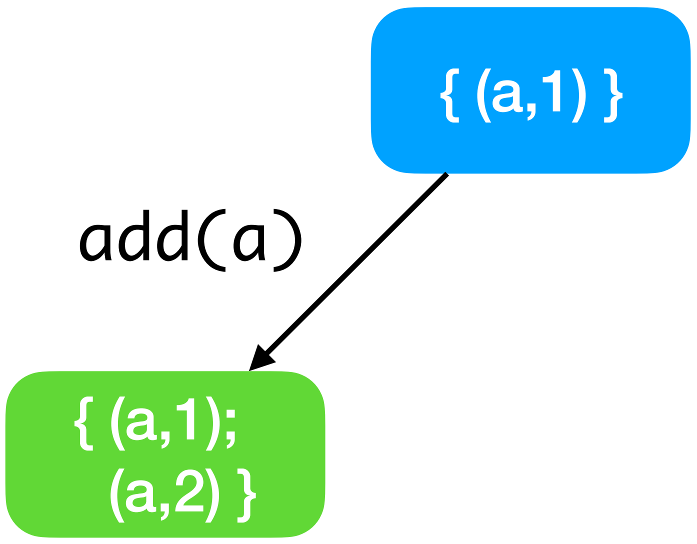
Fixing OR-Set

- Discriminate duplicate additions by associating a unique id

{ (a,1) }

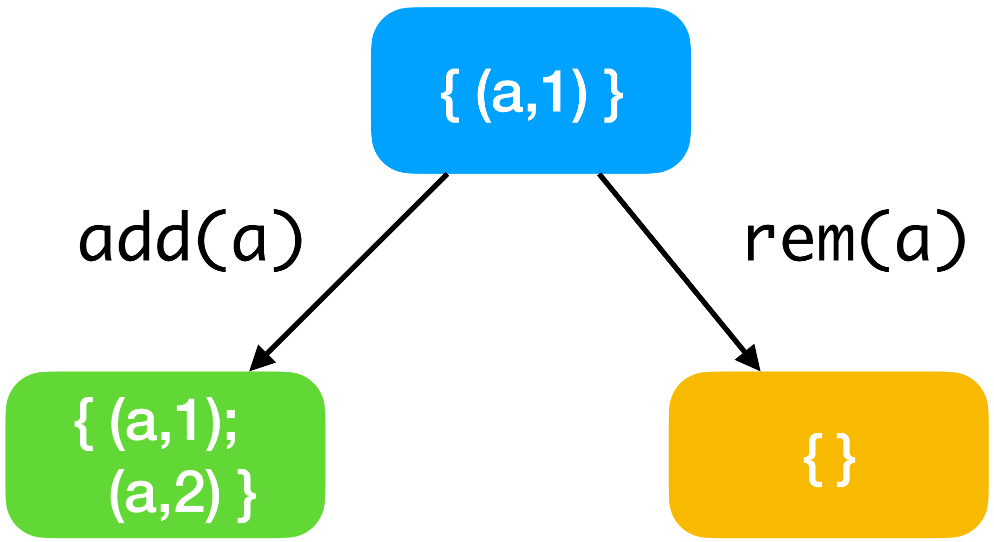
Fixing OR-Set

- Discriminate duplicate additions by associating a unique id



Fixing OR-Set

- Discriminate duplicate additions by associating a unique id



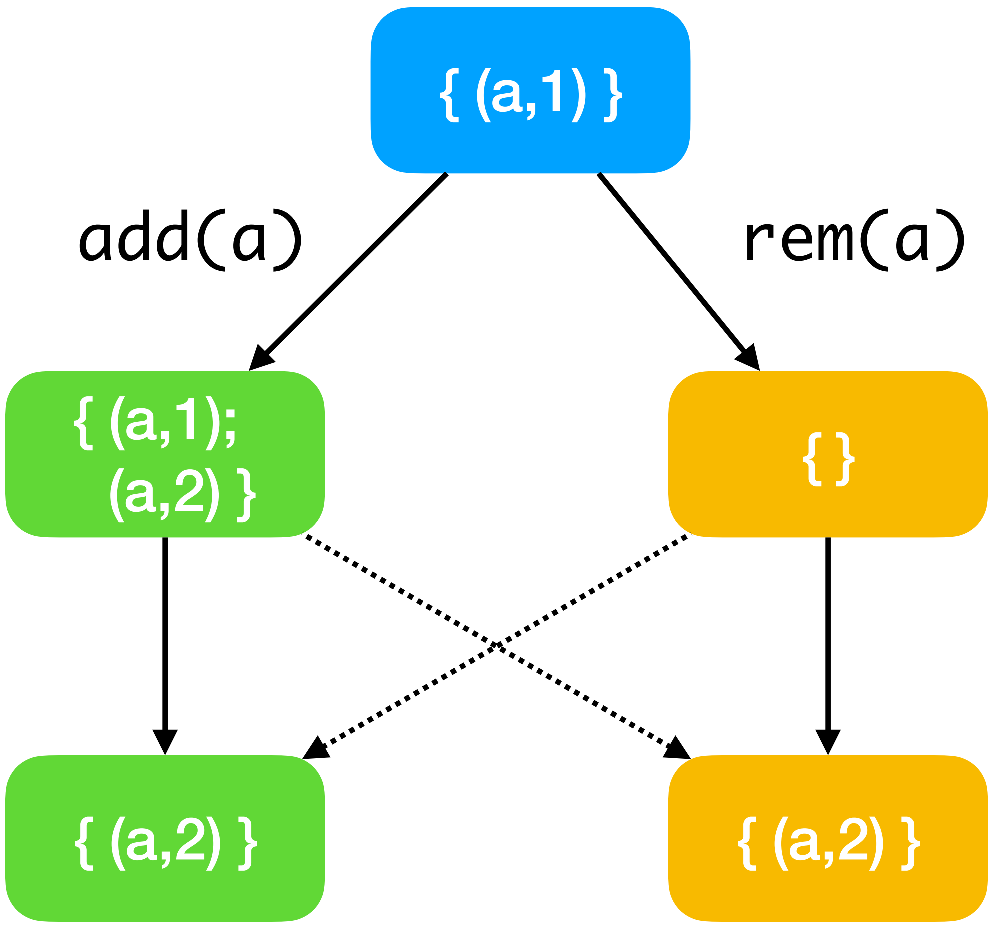
Fixing OR-Set

- Discriminate duplicate additions by associating a unique id

$\{\ \} \cup (\{ (a,1); (a,2) \} - \{ (a,1) \}) \cup (\{\ \} - \{ (a,1) \})$

$= \{\ \} \cup \{ (a,2) \} \cup \{\ \}$

$= \{ (a,2) \}$



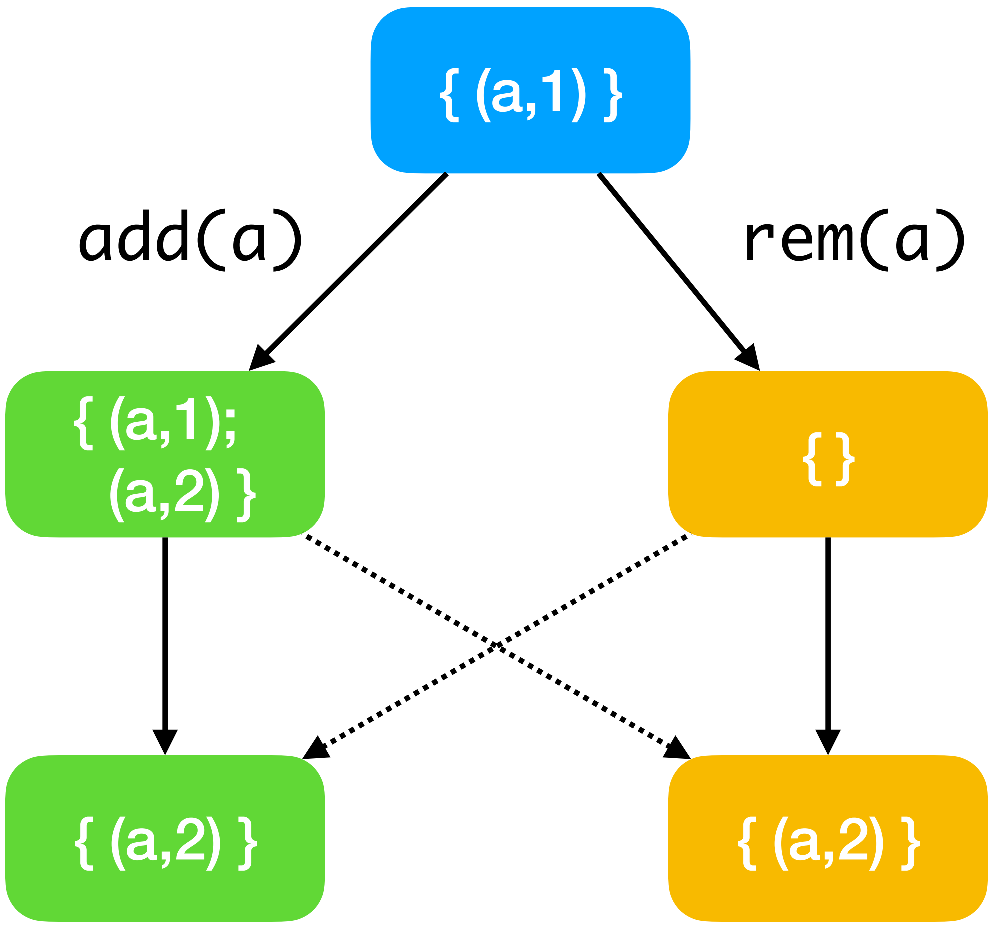
Fixing OR-Set

- Discriminate duplicate additions by associating a unique id

$$\{\ \} \cup (\{ (a,1); (a,2) \} - \{ (a,1) \}) \cup (\{\ \} - \{ (a,1) \})$$
$$= \{\ \} \cup \{ (a,2) \} \cup \{\ \}$$
$$= \{ (a,2) \}$$

- MRDT implementation

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



Fixing OR-Set

- Discriminate duplicate additions by associating a unique id

$$\{\,\} \cup (\{(a,1); (a,2)\} - \{(a,1)\}) \cup (\{\,\} - \{(a,1)\})$$
$$= \{\,\} \cup \{(a,2)\} \cup \{\,\}$$
$$= \{(a,2)\}$$

- MRDT implementation

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

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

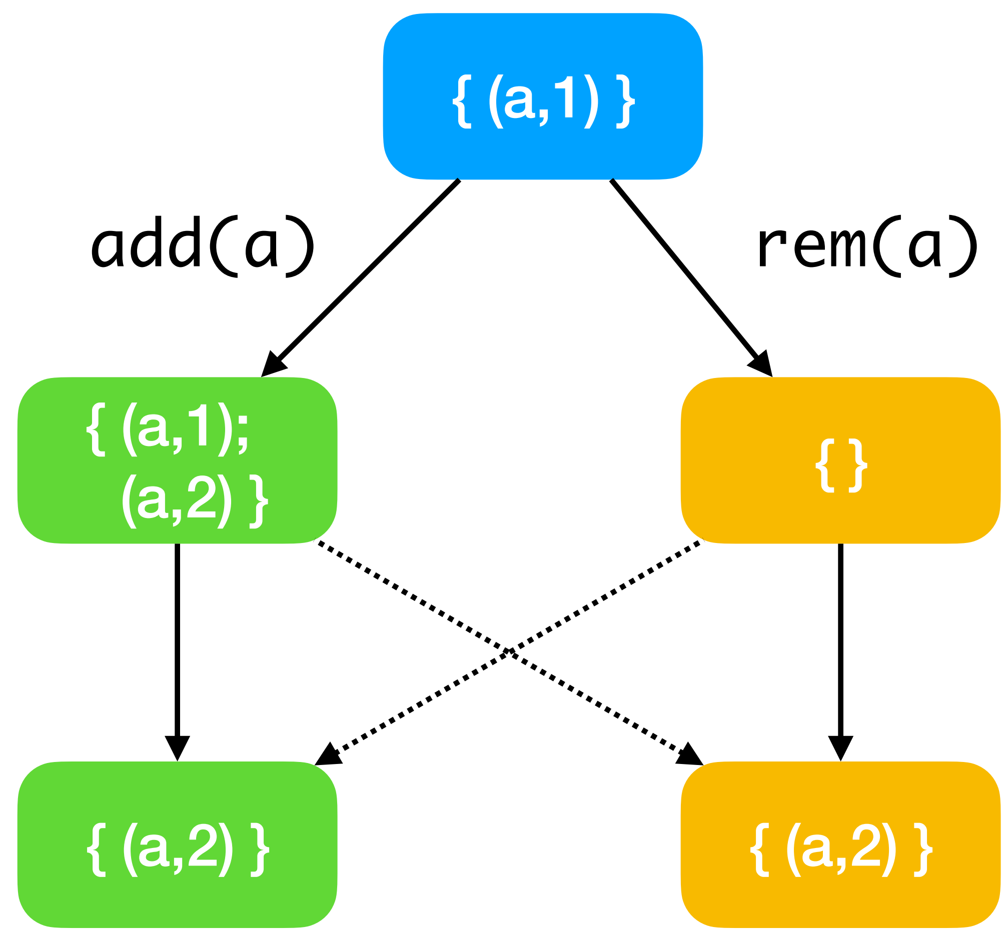
2: $\sigma_0 = \{\}$

3: $do(rd, \sigma, t) = (\sigma, \{a \mid (a,t) \in \sigma\})$

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

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

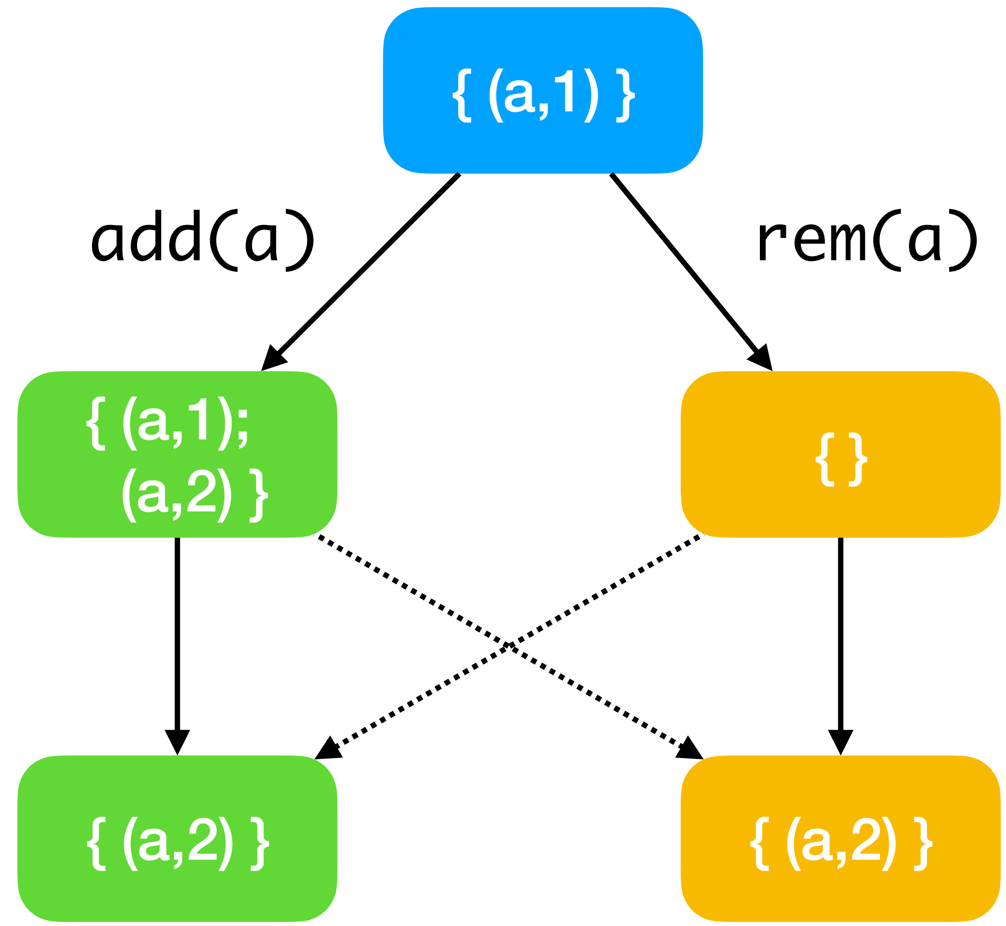
6: $merge(\sigma_{lca}, \sigma_a, \sigma_b) =$
$(\sigma_{lca} \cap \sigma_a \cap \sigma_b) \cup (\sigma_a - \sigma_{lca}) \cup (\sigma_b - \sigma_{lca})$



Fixing OR-Set

- Discriminate duplicate additions by associating a unique id

$$\{\} \cup (\{(a,1); (a,2)\} - \{(a,1)\}) \cup (\{\} - \{(a,1)\})$$
$$= \{\} \cup \{(a,2)\} \cup \{\}$$
$$= \{(a,2)\}$$



- MRDT implementation

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

Unique Lamport Timestamps

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

$\sigma_0 = \{\}$

$do(rd, \sigma, t) = (\sigma, \{a \mid (a, t) \in \sigma\})$

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

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

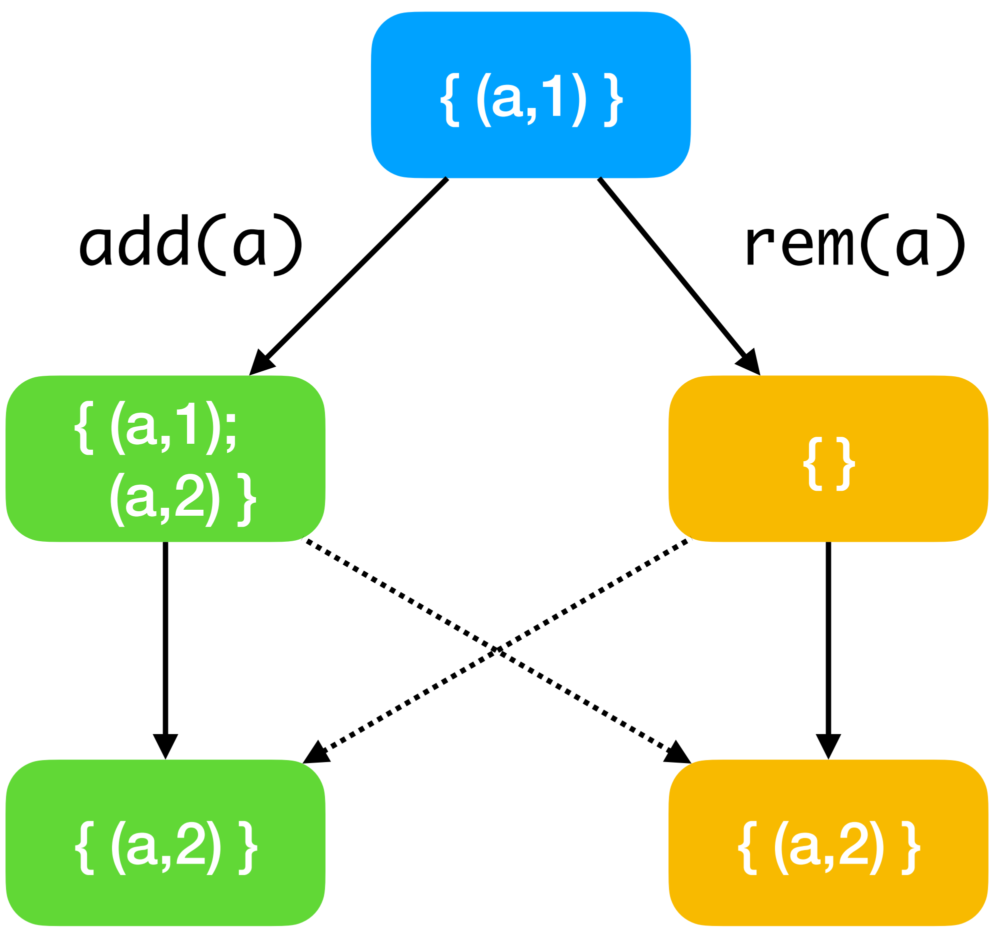
$merge(\sigma_{lca}, \sigma_a, \sigma_b) =$
$(\sigma_{lca} \cap \sigma_a \cap \sigma_b) \cup (\sigma_a - \sigma_{lca}) \cup (\sigma_b - \sigma_{lca})$

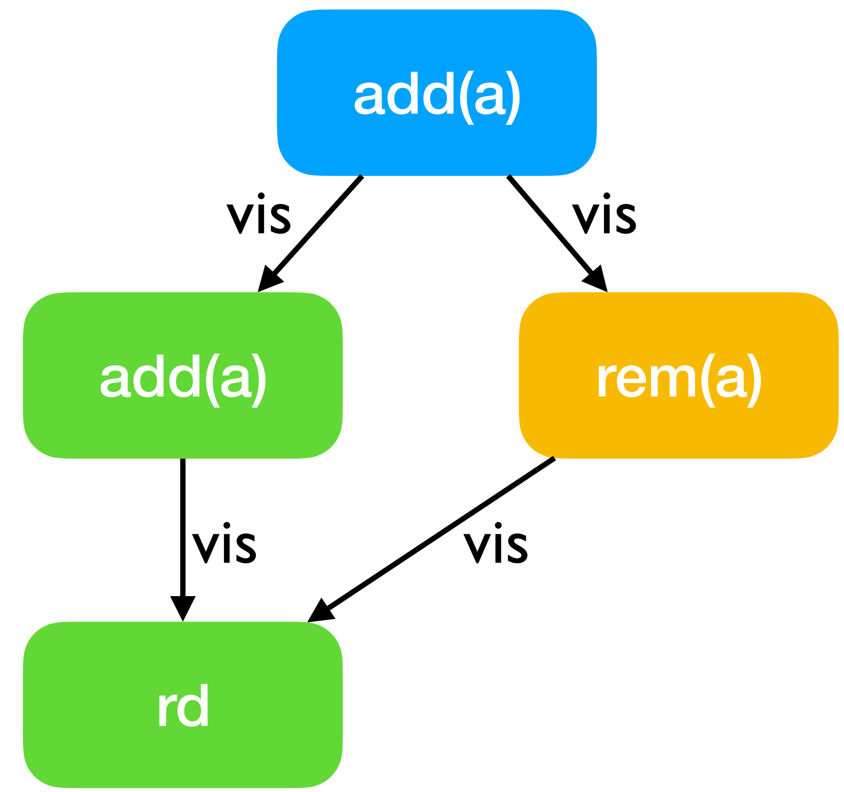
Specifying OR-Set

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

Specifying OR-Set

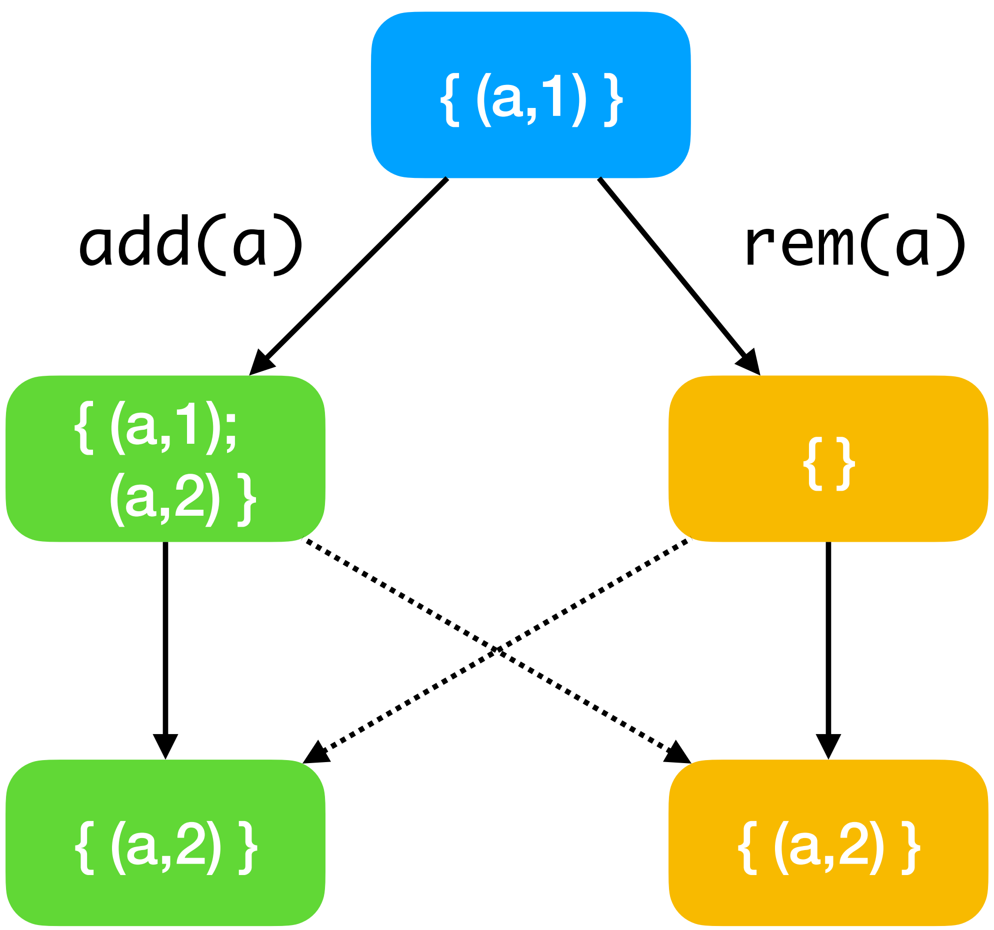
Abstract state $I = \langle E, oper, rval, time, vis \rangle$

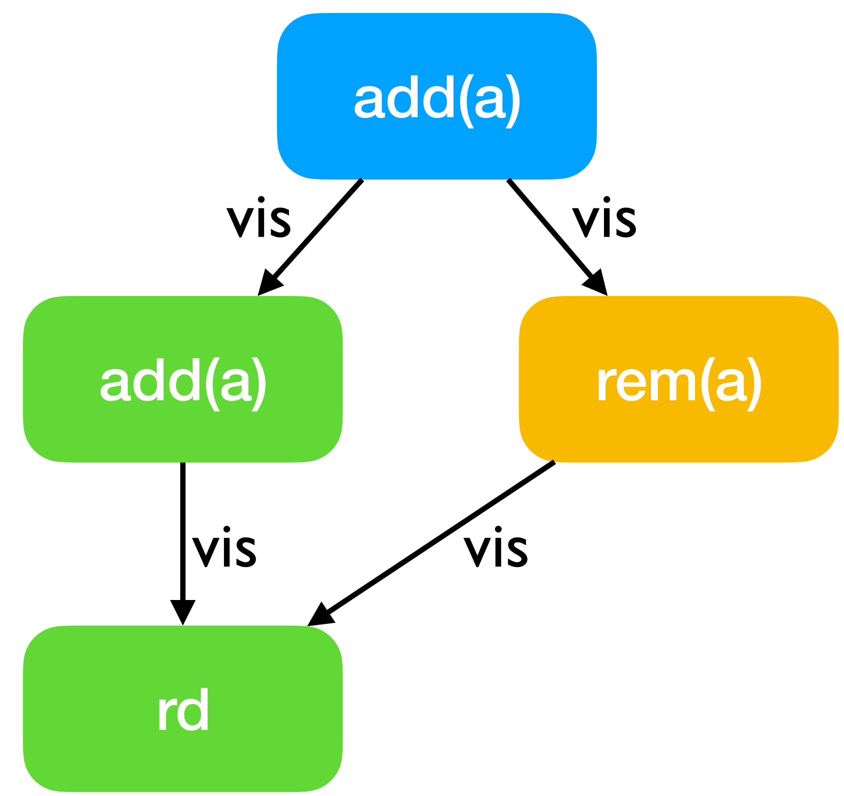




Specifying OR-Set

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

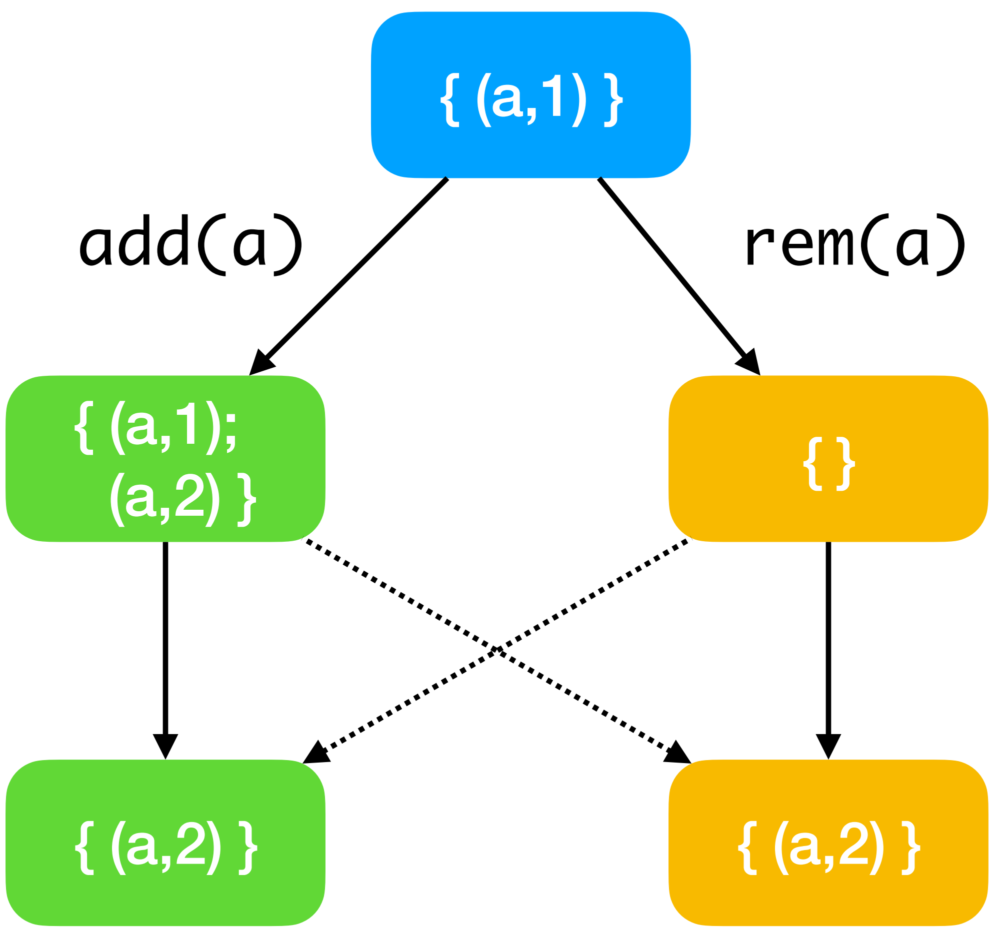


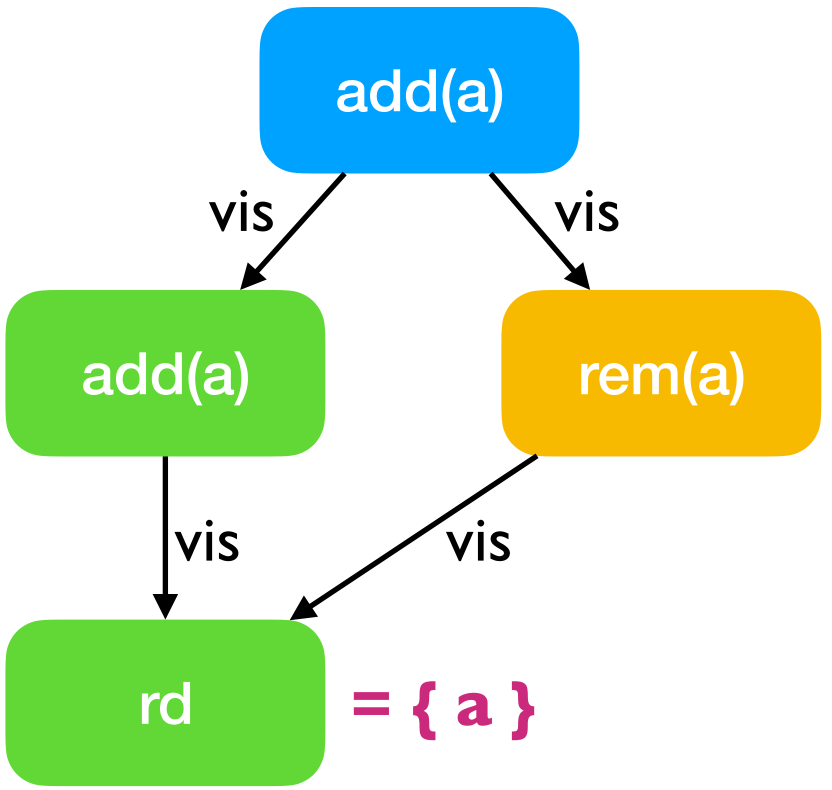


$$\mathcal{F}_{orset}(\text{rd}, \langle E, oper, rval, time, vis \rangle) = \{a \mid \exists e \in E.\ oper(e)$$
$$= \text{add}(a) \wedge \neg(\exists f \in E.\ oper(f) = \text{remove}(a) \wedge e \xrightarrow{vis} f)\}$$

Specifying OR-Set

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





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

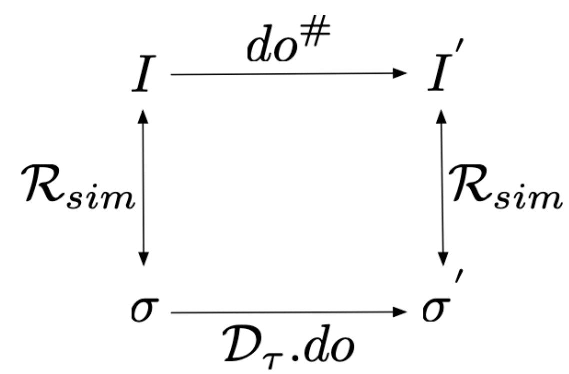
Simulation Relation

- Connects the abstract execution with the concrete state
- For the OR-set,

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

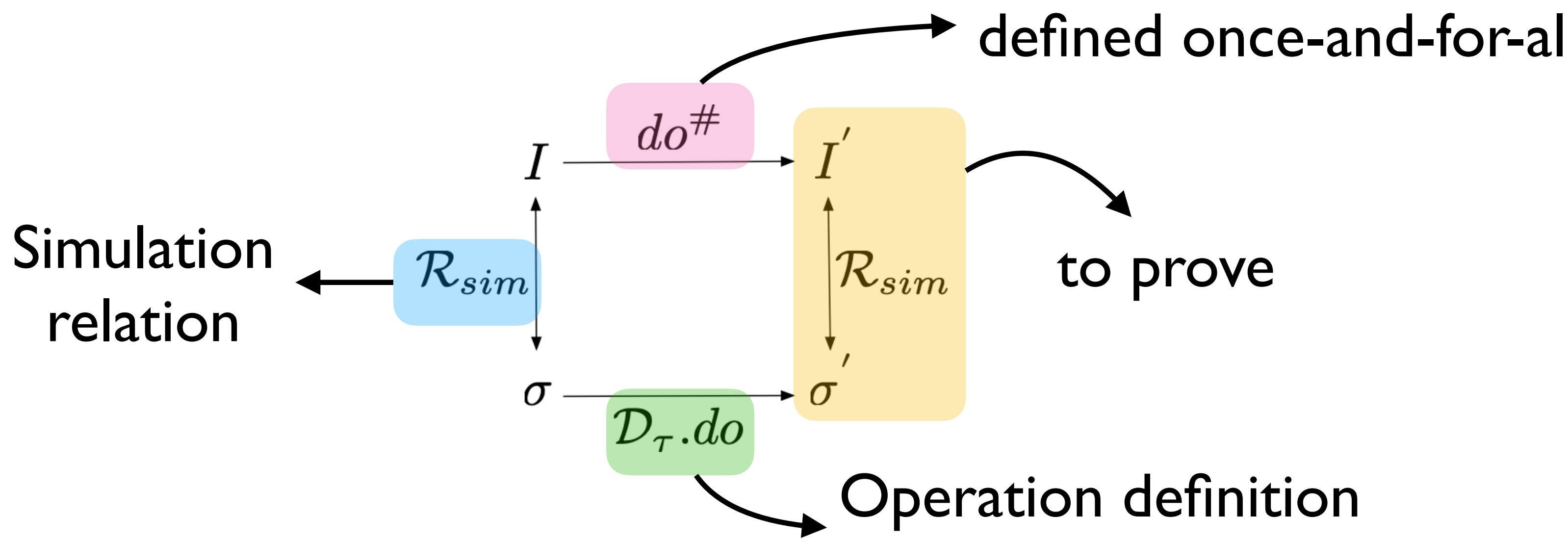
Verifying Operations

1. Show that the simulation holds for operations



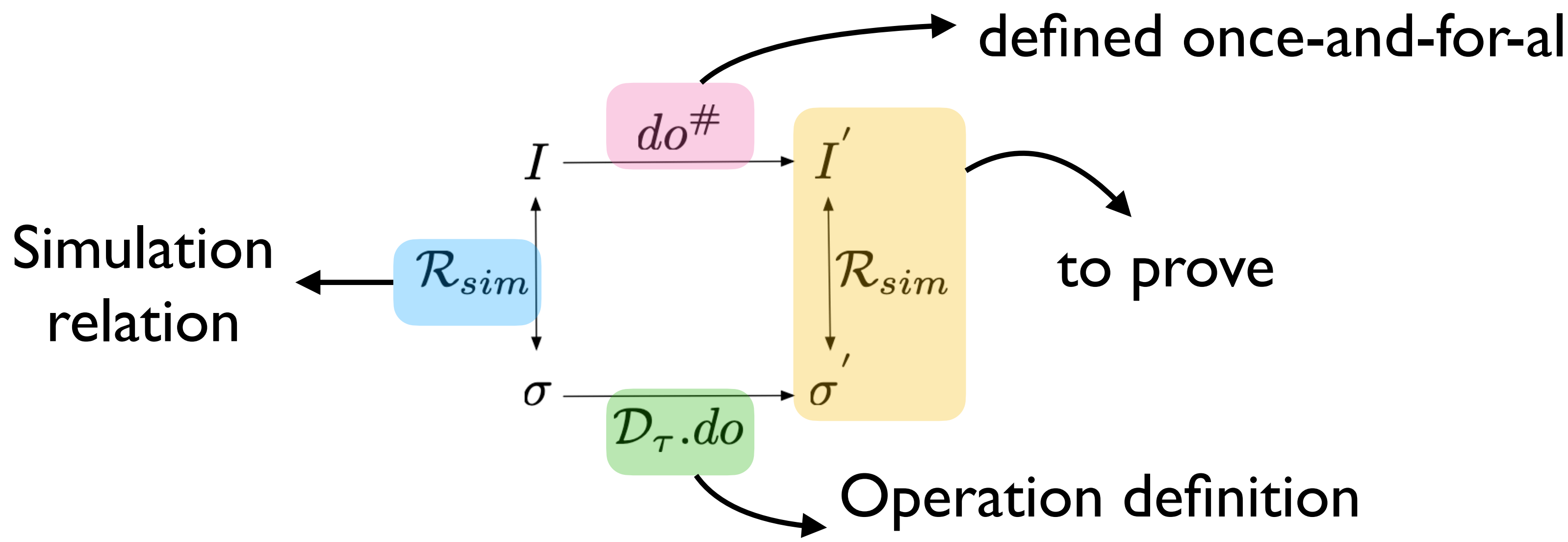
Verifying Operations

1. Show that the simulation holds for operations



Verifying Operations

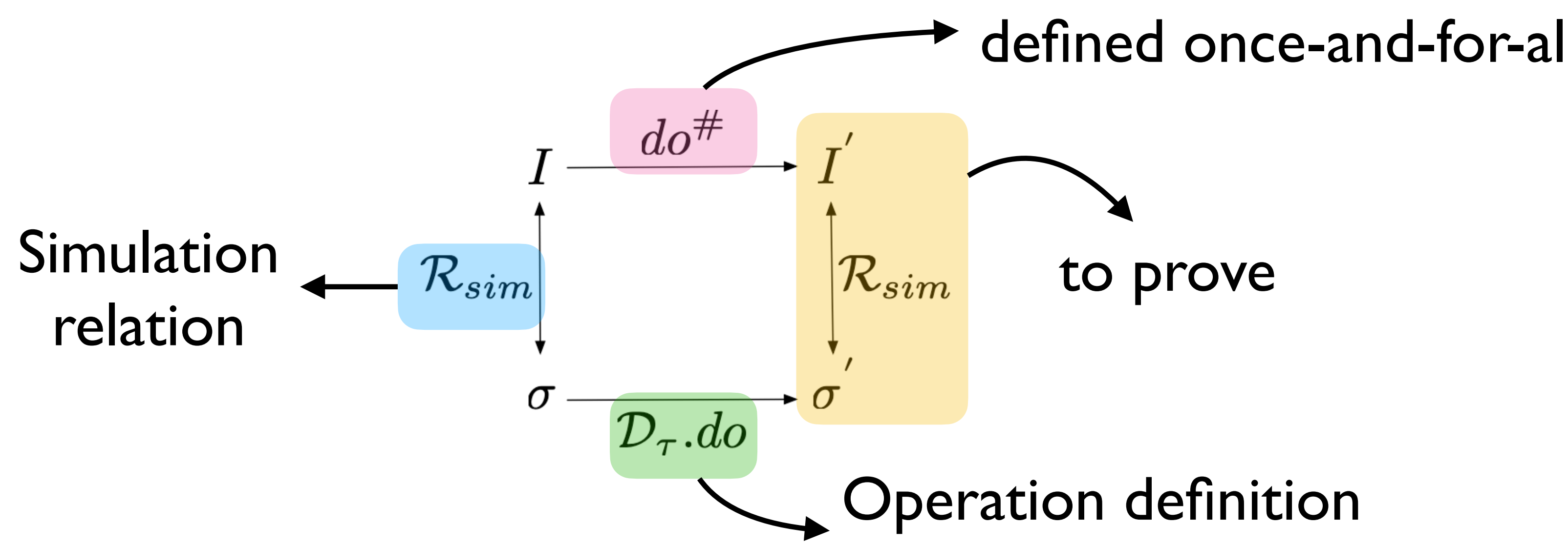
1. Show that the simulation holds for operations



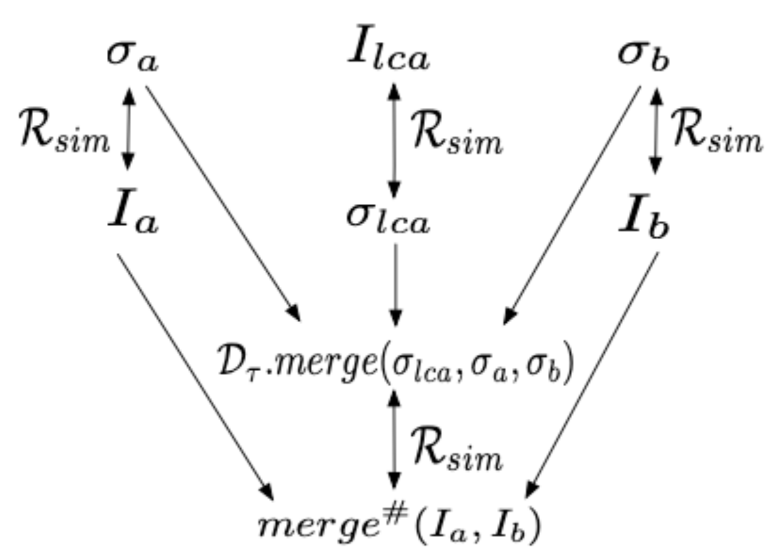
2. Show that the simulation holds for merge

Verifying Operations

1. Show that the simulation holds for operations

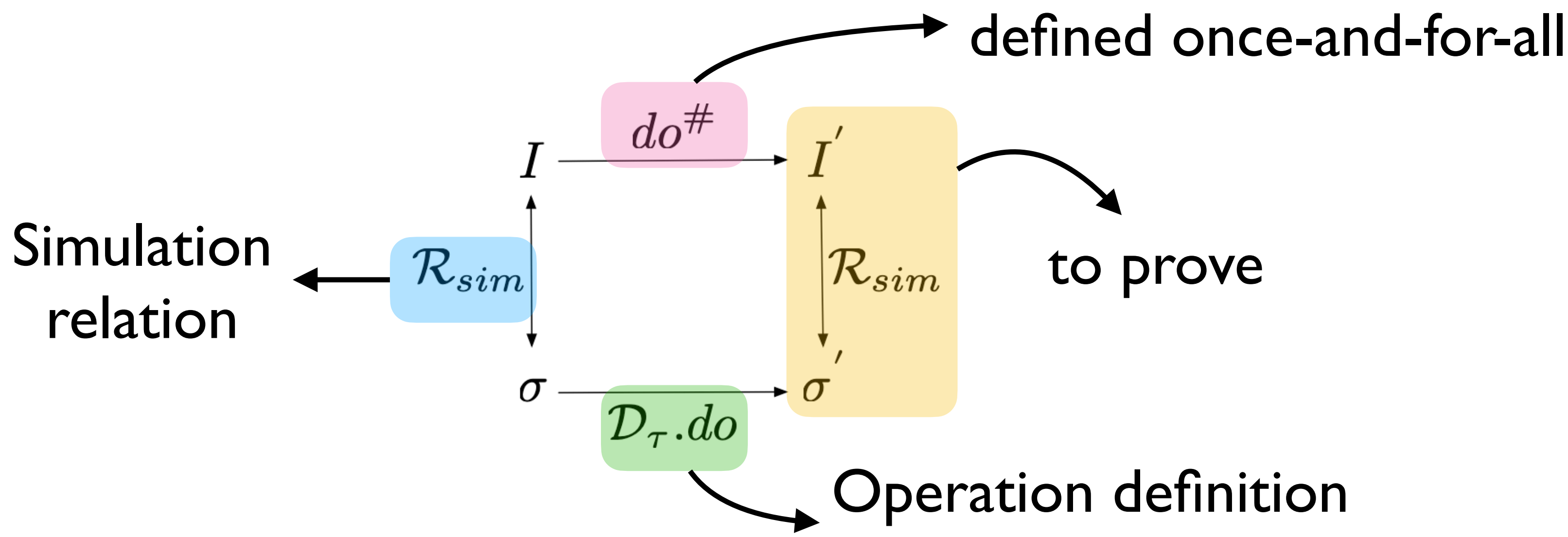


2. Show that the simulation holds for merge

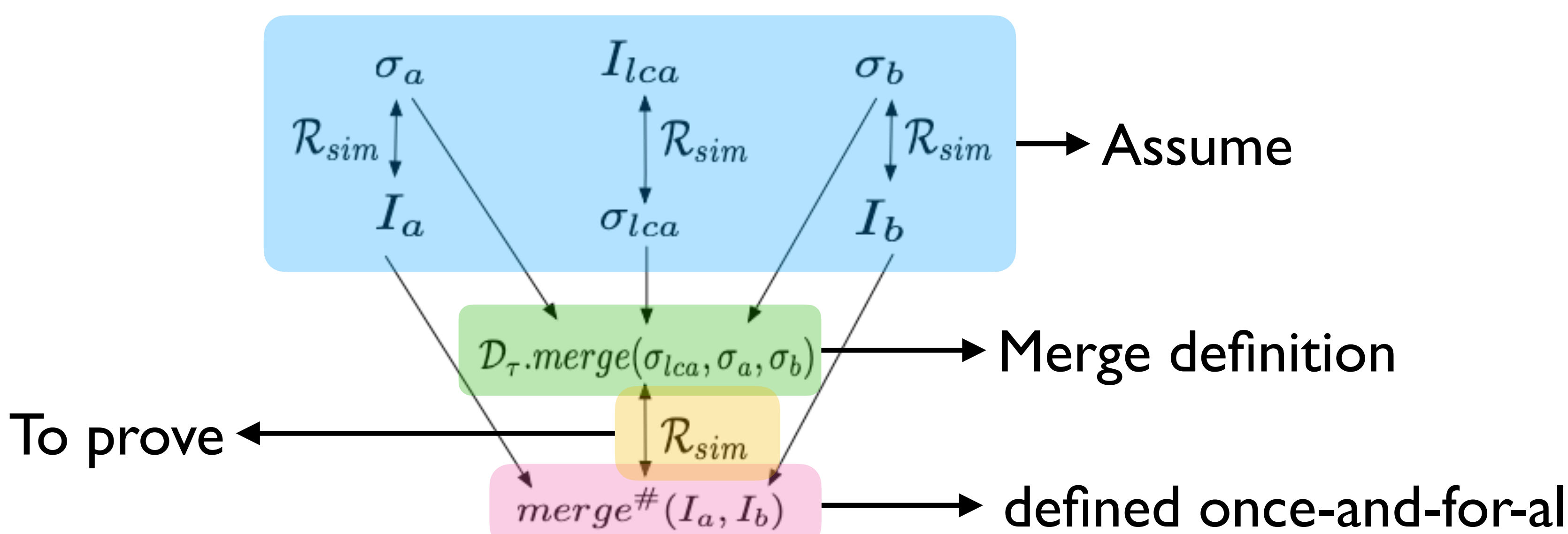


Verifying Operations

1. Show that the simulation holds for operations



2. Show that the simulation holds for merge



Verifying Operations

3. Show that the specification and the implementation agree on the return values of operations

$\Phi_{spec}(\mathcal{R}_{sim})$

$$\forall I, \sigma, e, op, a, t.\ \mathcal{R}_{sim}(I, \sigma) \wedge do^{\#}(I, e, op, a, t) = I^{'}$$
$$\wedge\ \mathcal{D}_{\tau}.do(op, \sigma, t) = (\sigma^{'}, a) \wedge \Psi_{ts}(I) \implies a = \mathcal{F}_{\tau}(o, I)$$

Verifying Operations

3. Show that the specification and the implementation agree on the return values of operations

$$\Phi_{spec}(\mathcal{R}_{sim}) \qquad \begin{array}{c} \forall I, \sigma, e, op, a, t.\ \mathcal{R}_{sim}(I, \sigma) \wedge do^{\#}(I, e, op, a, t) = I' \\ \wedge\ \mathcal{D}_{\tau}.do(op, \sigma, t) = (\sigma', a) \wedge \Psi_{ts}(I) \implies a = \mathcal{F}_{\tau}(o, I) \end{array}$$

4. Convergence

$$\Phi_{con}(\mathcal{R}_{sim}) \qquad \forall I, \sigma_a, \sigma_b.\ \mathcal{R}_{sim}(I, \sigma_a) \wedge \mathcal{R}_{sim}(I, \sigma_b) \implies \sigma_a \sim \sigma_b$$

Verifying Operations

3. Show that the specification and the implementation agree on the return values of operations

$$\Phi_{spec}(\mathcal{R}_{sim}) \qquad \forall I, \sigma, e, op, a, t.\ \mathcal{R}_{sim}(I, \sigma) \wedge do^{\#}(I, e, op, a, t) = I' \wedge \mathcal{D}_{\tau}.do(op, \sigma, t) = (\sigma', a) \wedge \Psi_{ts}(I) \implies a = \mathcal{F}_{\tau}(o, I)$$

4. Convergence

$$\Phi_{con}(\mathcal{R}_{sim}) \qquad \forall I, \sigma_a, \sigma_b.\ \mathcal{R}_{sim}(I, \sigma_a) \wedge \mathcal{R}_{sim}(I, \sigma_b) \implies \sigma_a \sim \sigma_b$$

- ✦ Permits the different replicas to converge to states that are *observationally equal* but not *structurally equal*
  - ✤ Example: differently balanced BSTs

Verifying Operations

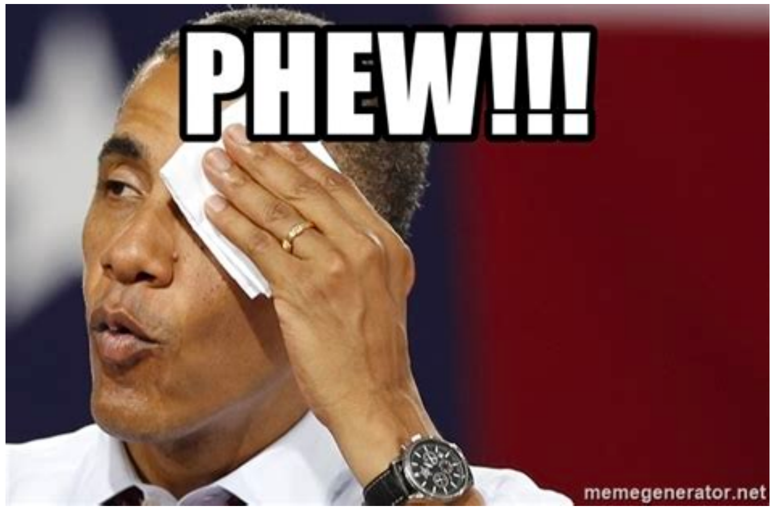
3. Show that the specification and the implementation agree on the return values of operations

$$\Phi_{spec}(\mathcal{R}_{sim}) \qquad \begin{array}{c} \forall I, \sigma, e, op, a, t.\ \mathcal{R}_{sim}(I,\sigma) \wedge do^{\#}(I, e, op, a, t) = I' \\ \wedge\ \mathcal{D}_{\tau}.do(op, \sigma, t) = (\sigma', a) \wedge \Psi_{ts}(I) \implies a = \mathcal{F}_{\tau}(o, I) \end{array}$$

4. Convergence

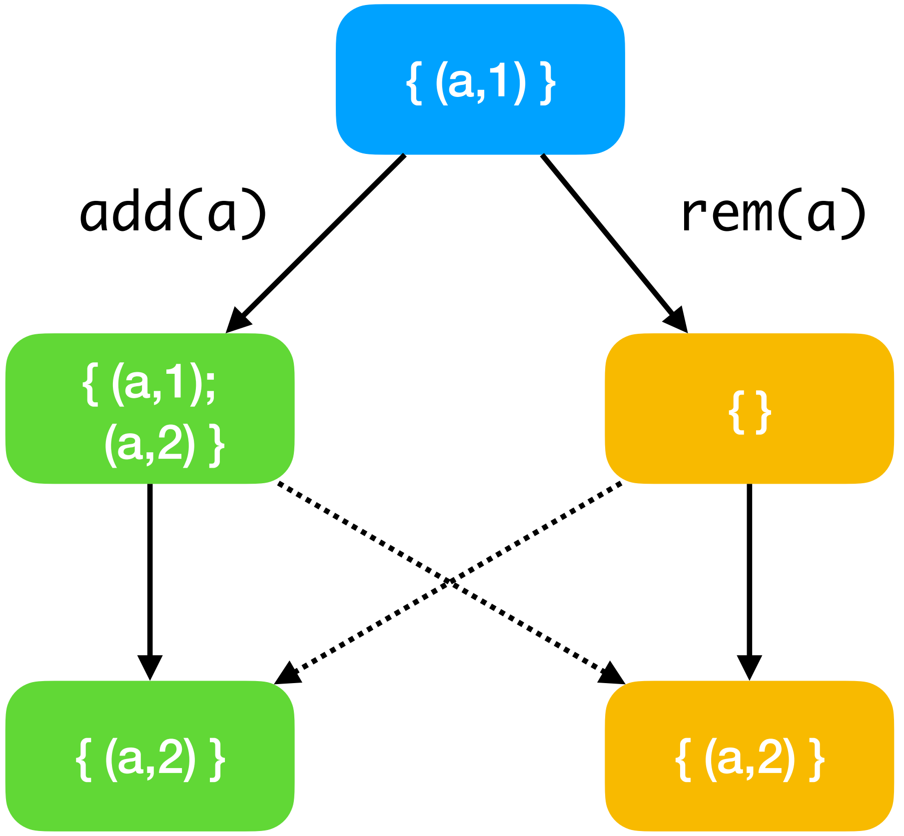
$$\Phi_{con}(\mathcal{R}_{sim}) \qquad \forall I, \sigma_a, \sigma_b.\ \mathcal{R}_{sim}(I, \sigma_a) \wedge \mathcal{R}_{sim}(I, \sigma_b) \implies \sigma_a \sim \sigma_b$$

- ✦ Permits the different replicas to converge to states that are *observationally equal* but not *structurally equal*
  - ✤ Example: differently balanced BSTs



Space-efficient OR-Set

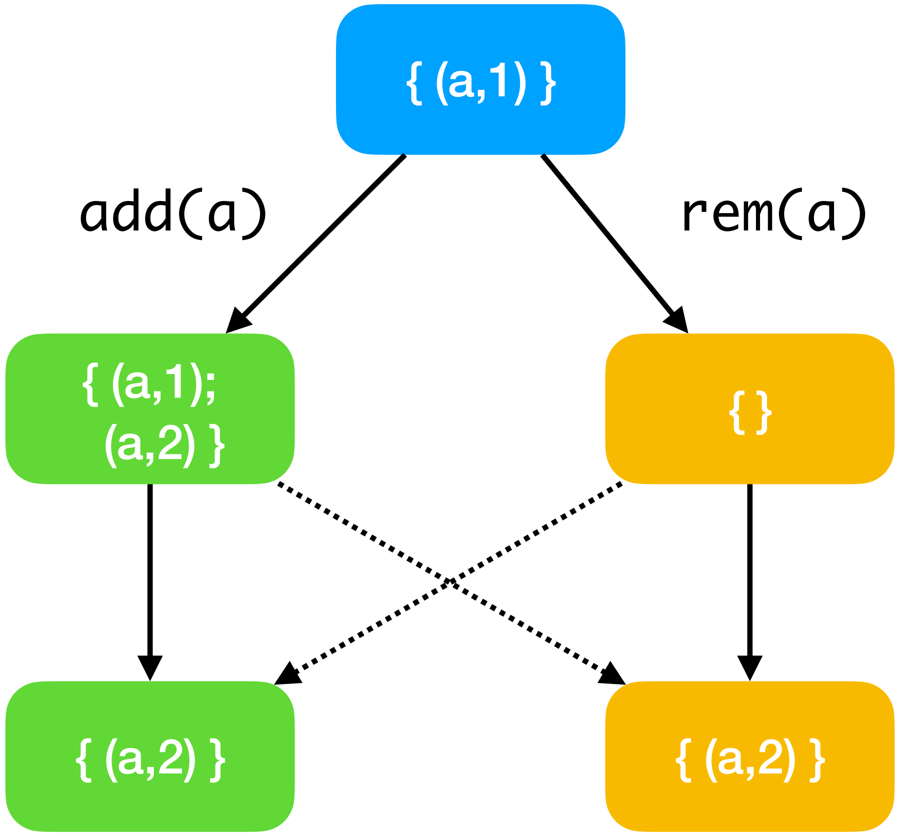
- Recall that the OR-set has duplicates



- How can we remove them?

Space-efficient OR-Set

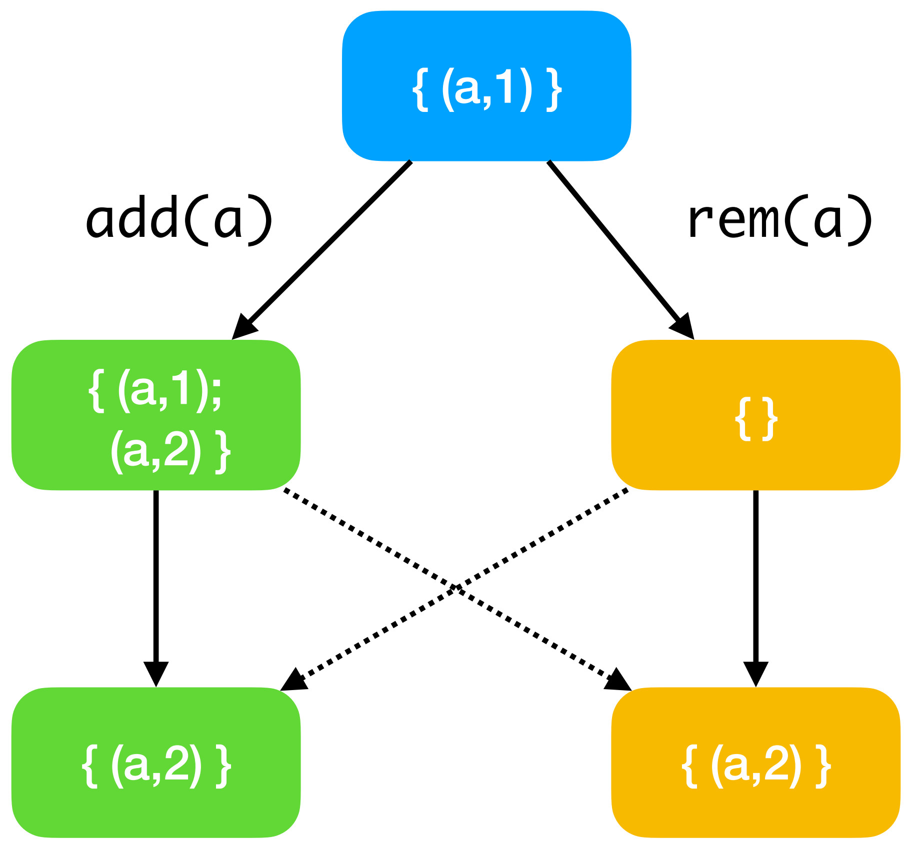
- Recall that the OR-set has duplicates



- How can we remove them?
- **Idea**
  - ★ On addition, replace existing element's timestamp with the new timestamp
  - ★ On merge, pick the larger timestamp

Space-efficient OR-Set

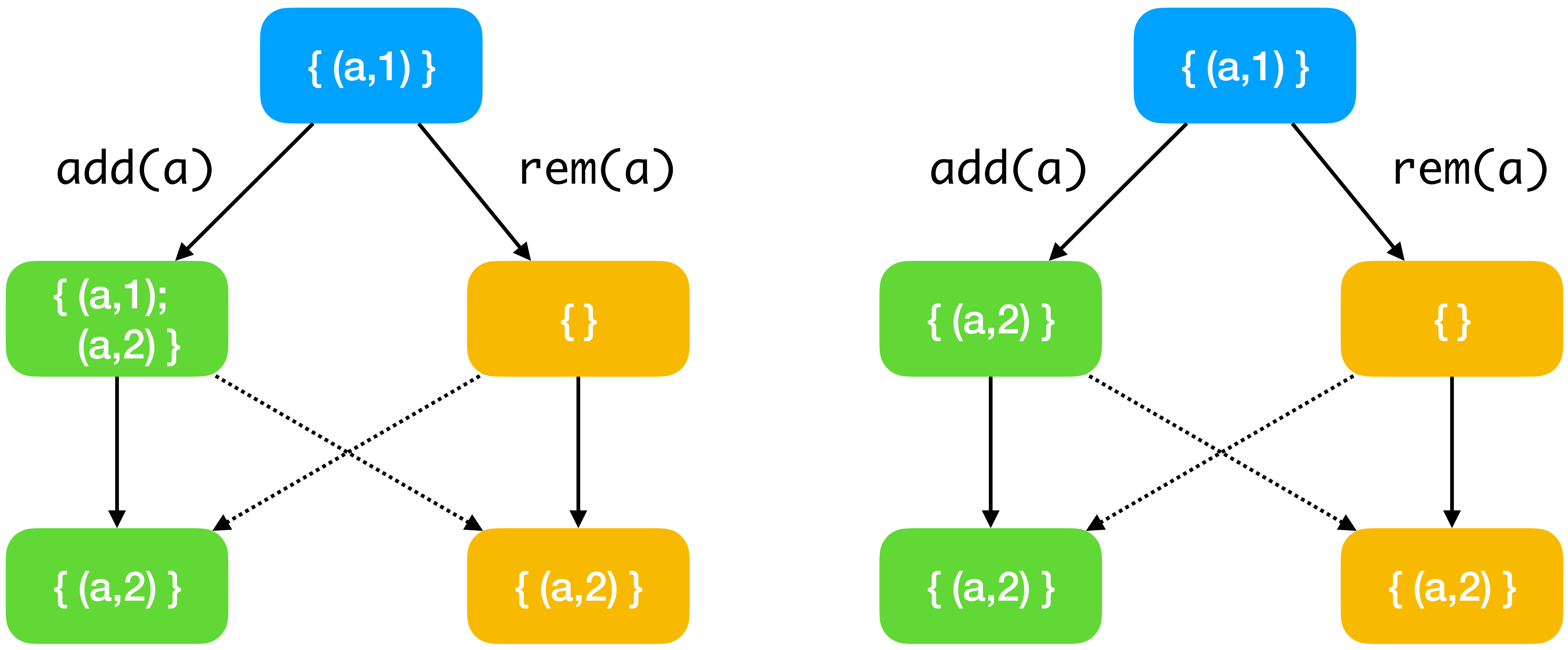
- Recall that the OR-set has duplicates



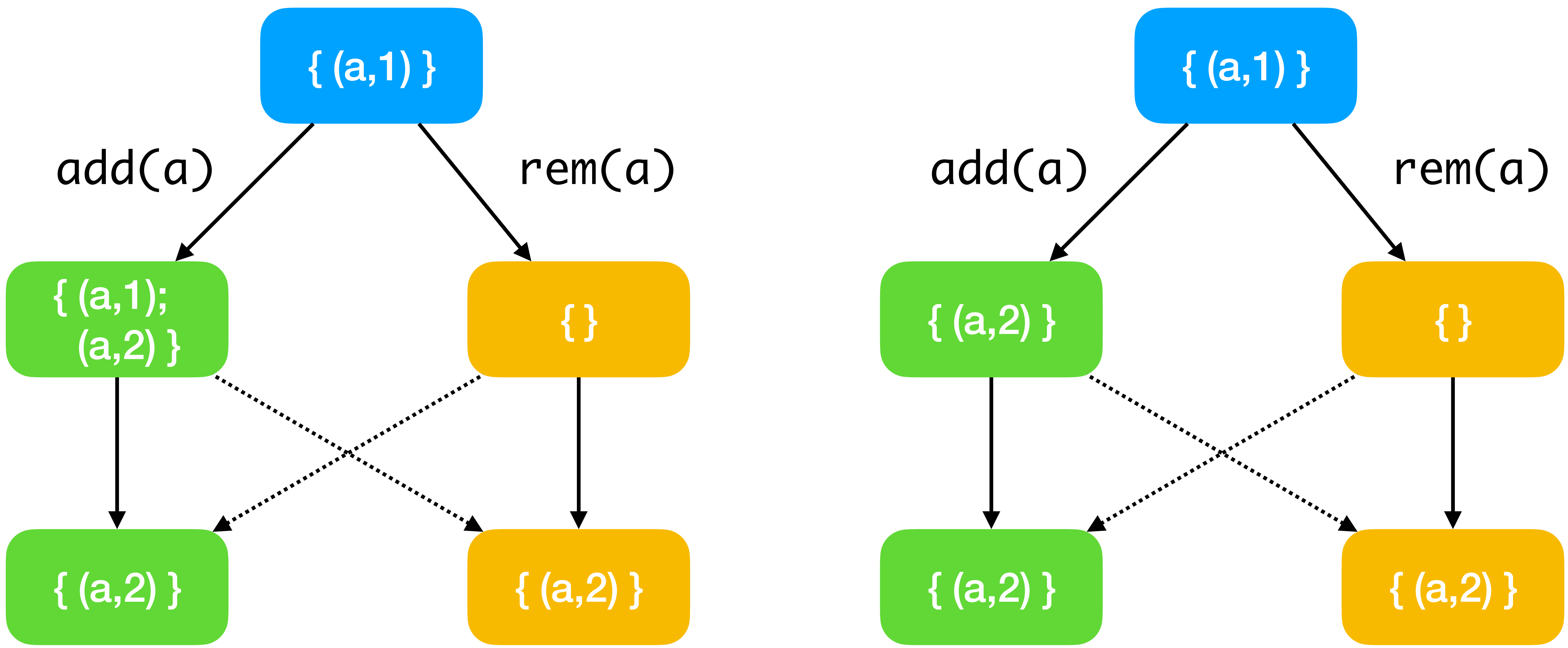
- How can we remove them?
- **Idea**
    - ★ On addition, replace existing element's timestamp with the new timestamp
    - ★ On merge, pick the larger timestamp

Correctness argument is tricky

Space-efficient OR-Set

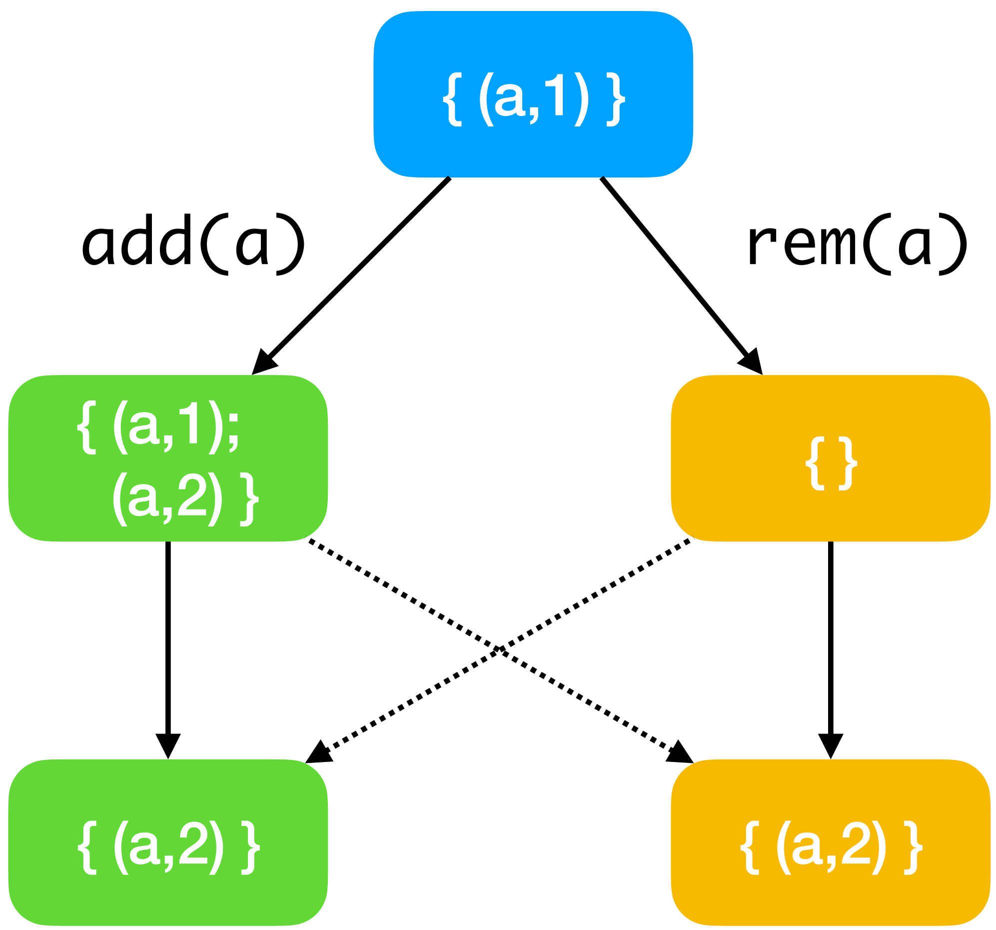


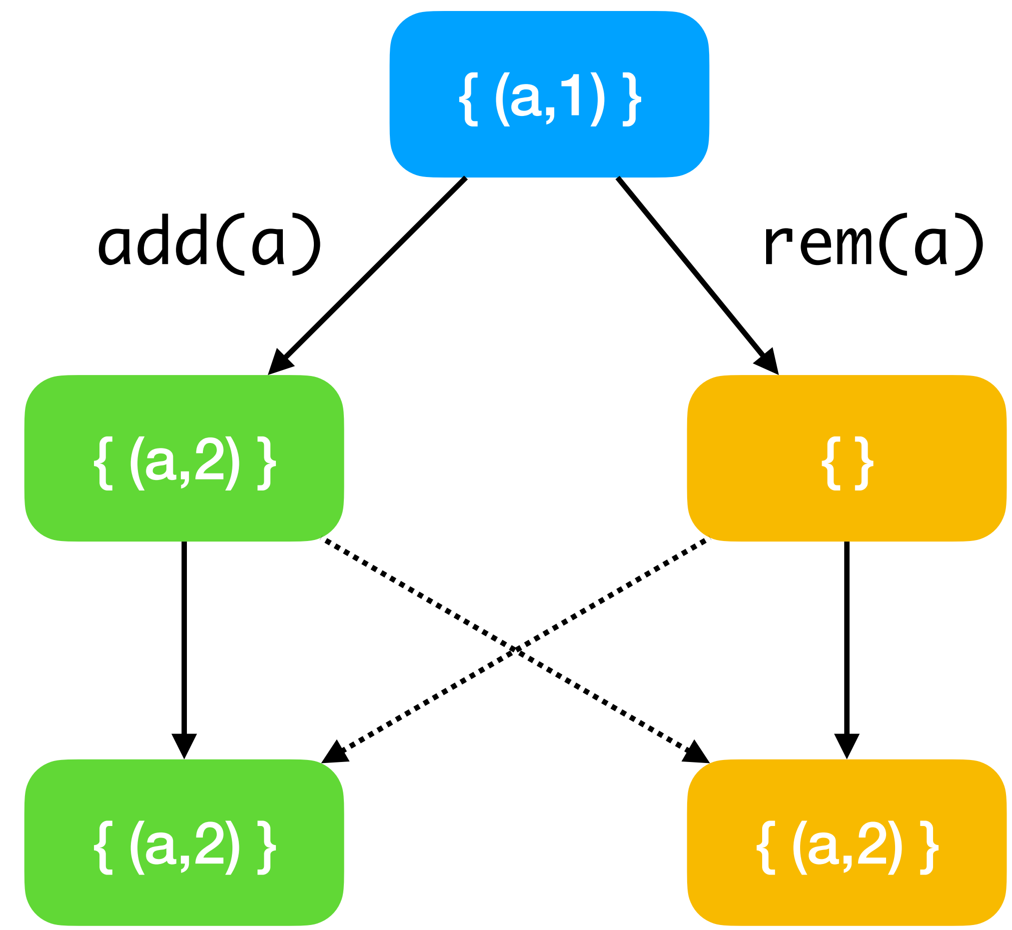
Space-efficient OR-Set



$$
\begin{aligned}
&\mathcal{R}_{sim}((E, oper, rval, time, vis), \sigma) \iff \\
&(\forall (a,t) \in \sigma \implies (\exists e \in E.\ oper(e) = add(a) \land time(e) = t \\
&\land \neg(\exists r \in E.\ oper(r) = remove(a) \land e \xrightarrow{vis} r)) \land \\
&(\forall e \in E.\ oper(e) = add(a) \land \neg(\exists r \in E.oper(r) = remove(a) \\
&\land e \xrightarrow{vis} r) \implies t \geq time(e))) \land \\
&(\forall e \in E. \forall a \in \mathbb{N}.\ oper(e) = add(a) \\
&\land \neg(\exists r \in E.\ oper(r) = remove(a) \land e \xrightarrow{vis} r) \implies (a, \_) \in \sigma)
\end{aligned}
$$

Space-efficient OR-Set





$$
\begin{aligned}
&\mathcal{R}_{sim}((E, oper, rval, time, vis), \sigma) \iff \\
&(\forall (a,t) \in \sigma \implies (\exists e \in E.\ oper(e) = add(a) \wedge time(e) = t \\
&\wedge \neg(\exists r \in E.\ oper(r) = remove(a) \wedge e \xrightarrow{vis} r)) \wedge \\
&(\forall e \in E.\ oper(e) = add(a) \wedge \neg(\exists r \in E. oper(r) = remove(a) \\
&\wedge e \xrightarrow{vis} r) \implies t \geq time(e))) \wedge \\
&(\forall e \in E. \forall a \in \mathbb{N}.\ oper(e) = add(a) \\
&\wedge \neg(\exists r \in E.\ oper(r) = remove(a) \wedge e \xrightarrow{vis} r) \implies (a, \_) \in \sigma)
\end{aligned}
$$

Simulation relation is more intricate as one would expect

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 CRDTs 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
  - ★ For simplicity, channels and messages cannot be deleted
- Represent application state as a grow-only map with string (channel name) keys and mergeable-log as values
- **Goal:**
  - ★ *map* and *log* proved correct separately
  - ★ Use the proof of underlying RDTs to prove chat application correctness

Generic Map MRDT

- Specification

Generic Map MRDT

- Specification

$$\mathcal{F}_{\alpha-map}(get(k, o_\alpha), I) = \text{let } I_\alpha = project(k, I) \text{ in } \mathcal{F}_\alpha(o_\alpha, I_\alpha)$$

where

$$project\ k\ I_{\alpha-map} = I_\alpha$$

Generic Map MRDT

- Specification

$$\mathcal{F}_{\alpha-map}(get(k, o_\alpha), I) = \\ \quad \text{let } I_\alpha = project(k, I) \text{ in } \mathcal{F}_\alpha(o_\alpha, I_\alpha)$$

where

$$project\ k\ I_{\alpha-map} = I_\alpha$$

- Project *filters* the abstract state of the map on the key *k* and returns an abstract state of the underlying data type
  - ★ Provided by the user once for a generic MRDT

Generic Map MRDT

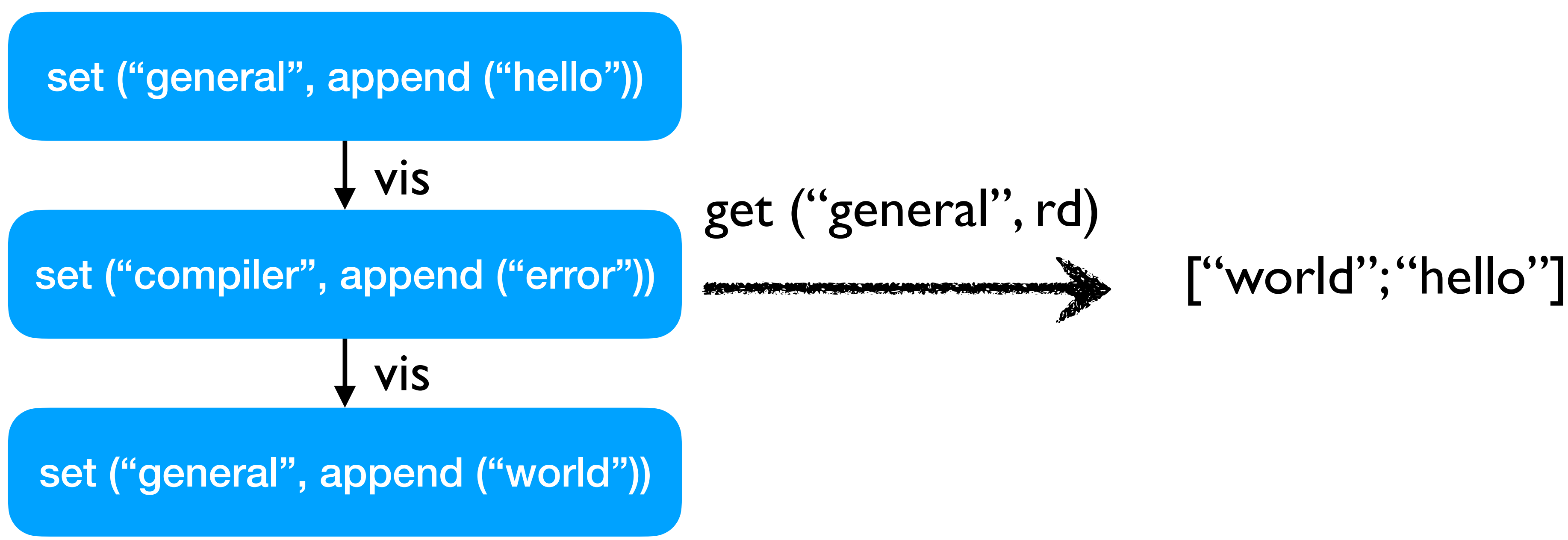
- Specification

$$\mathcal{F}_{\alpha-map}(get(k, o_\alpha), I) = \text{let } I_\alpha = project(k, I) \text{ in } \mathcal{F}_\alpha(o_\alpha, I_\alpha)$$

where

$$project\ k\ I_{\alpha-map} = I_\alpha$$

- Project *filters* the abstract state of the map on the key *k* and returns an abstract state of the underlying data type
  - ★ Provided by the user once for a generic MRDT



Generic Map MRDT

Implementation

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

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

2: $\sigma_0 = \{\}$

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

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

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

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

Simulation Relation

$$\mathcal{R}_{sim-\alpha-map}(I, \sigma) \iff \forall k.$$

1: $(k \in dom(\sigma) \iff \exists e \in I.E.\ oper(e) = set(k, \_)) \wedge$

2: $\quad \mathcal{R}_{sim-\alpha}\ (project(k, I),\ \delta(\sigma, k))$

Generic Map MRDT

Implementation

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

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

2: $\sigma_0 = \{\}$

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

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

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

Get applies given operation on the value at key k and returns the value

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

Simulation Relation

$$\mathcal{R}_{sim-\alpha-map}(I, \sigma) \iff \forall k.$$

1: $(k \in dom(\sigma) \iff \exists e \in I.E.\ oper(e) = set(k, \_)) \wedge$

2: $\quad \mathcal{R}_{sim-\alpha}\ (project(k, I),\ \delta(\sigma, k))$

Generic Map MRDT

Implementation

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

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

$$\sigma_0 = \{\}$$

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

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

Set is Get + update the map with the new state

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

Get applies given operation on the value at key k and returns the value

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

Simulation Relation

$$\mathcal{R}_{sim-\alpha-map}(I, \sigma) \iff \forall k.$$

$$(k \in dom(\sigma) \iff \exists e \in I.E.\ oper(e) = set(k, \_)) \land$$

$$\mathcal{R}_{sim-\alpha}\ (project(k, I),\ \delta(\sigma, k))$$

Generic Map MRDT

Implementation

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

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

$$\sigma_0 = \{\}$$

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

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

Set is Get + update the map with the new state

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

Get applies given operation on the value at key k and returns the value

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

Merge uses the merge of the underlying value type!

Simulation Relation

$$\mathcal{R}_{sim-\alpha-map}(I, \sigma) \iff \forall k.$$

$$(k \in dom(\sigma) \iff \exists e \in I.E.\ oper(e) = set(k, \_)) \wedge$$

$$\mathcal{R}_{sim-\alpha}\ (project(k, I),\ \delta(\sigma, k))$$

Generic Map MRDT

Implementation

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

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

$$\sigma_0 = \{\}$$

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

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

Set is Get + update the map with the new state

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

Get applies given operation on the value at key k and returns the value

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

Merge uses the merge of the underlying value type!

Simulation Relation

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

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

Composing IRC-style chat

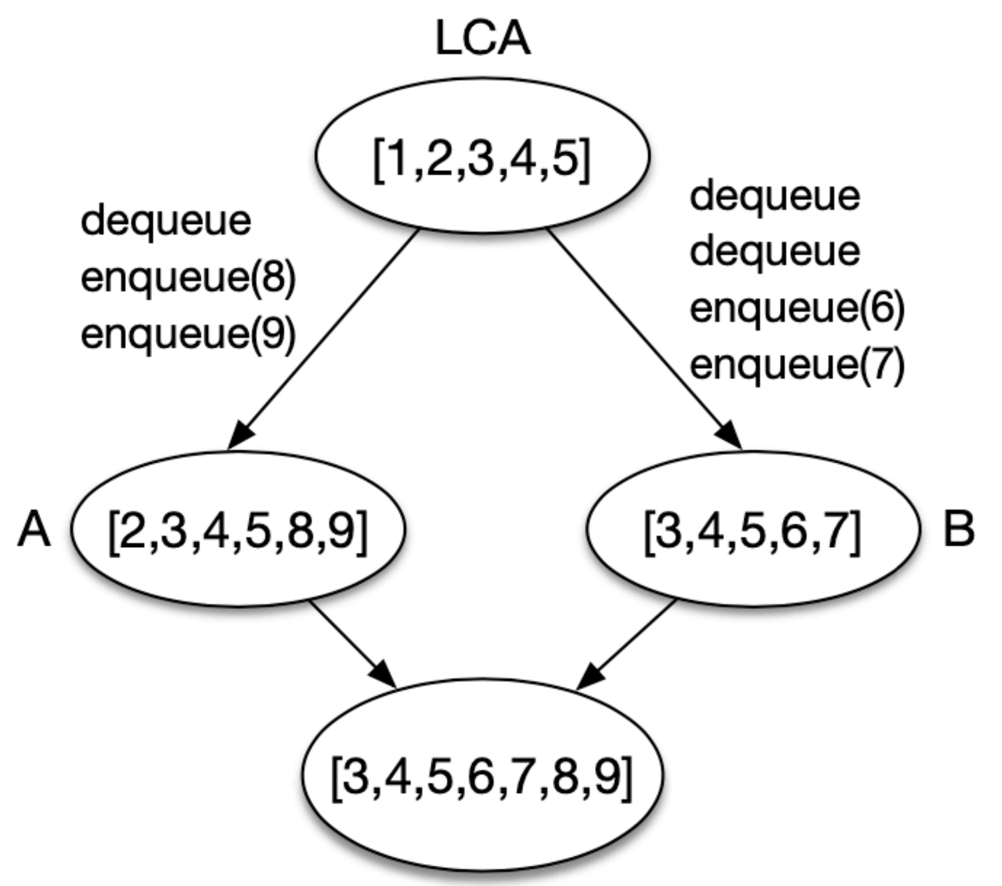
- Program state is constructed by instantiating *generic map* with *mergeable log*
    - ★ The proof of correctness of the chat application directly follows from the composition!

Mergeable Queues

- Replicated queue with *at-least-once* dequeue semantics
  - ★ First verified queue RDT!

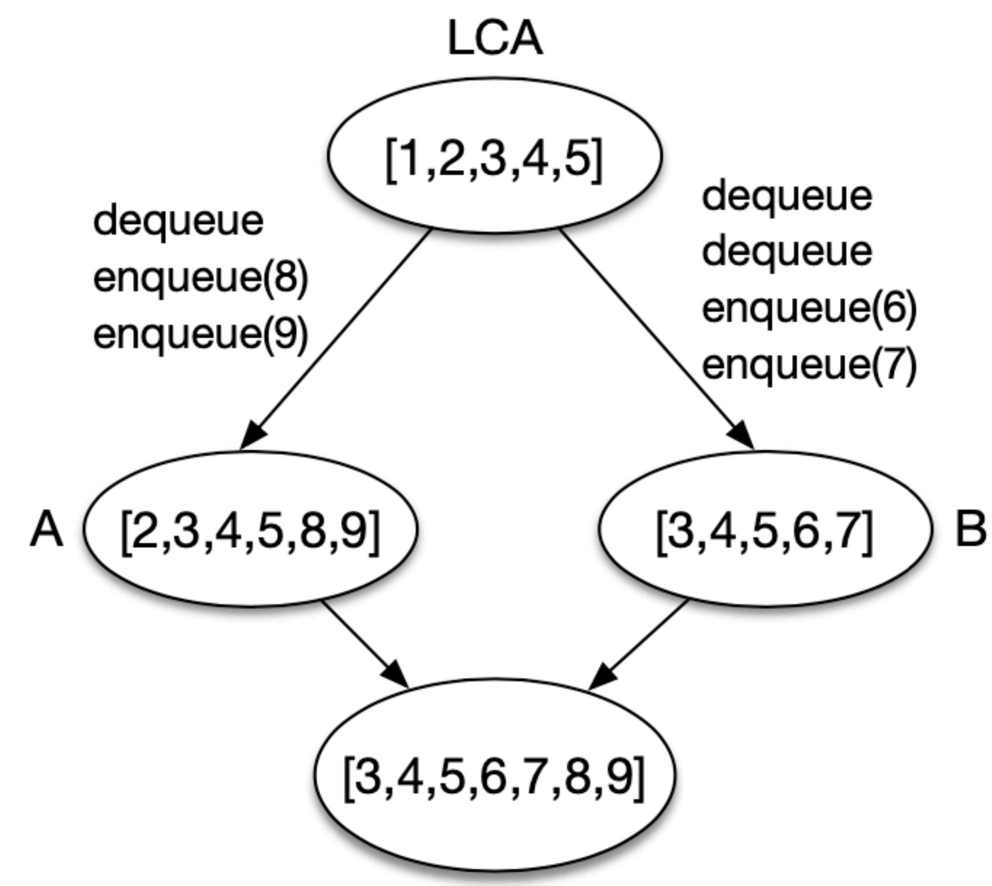
Mergeable Queues

- Replicated queue with *at-least-once* dequeue semantics
  - ★ First verified queue RDT!



Mergeable Queues

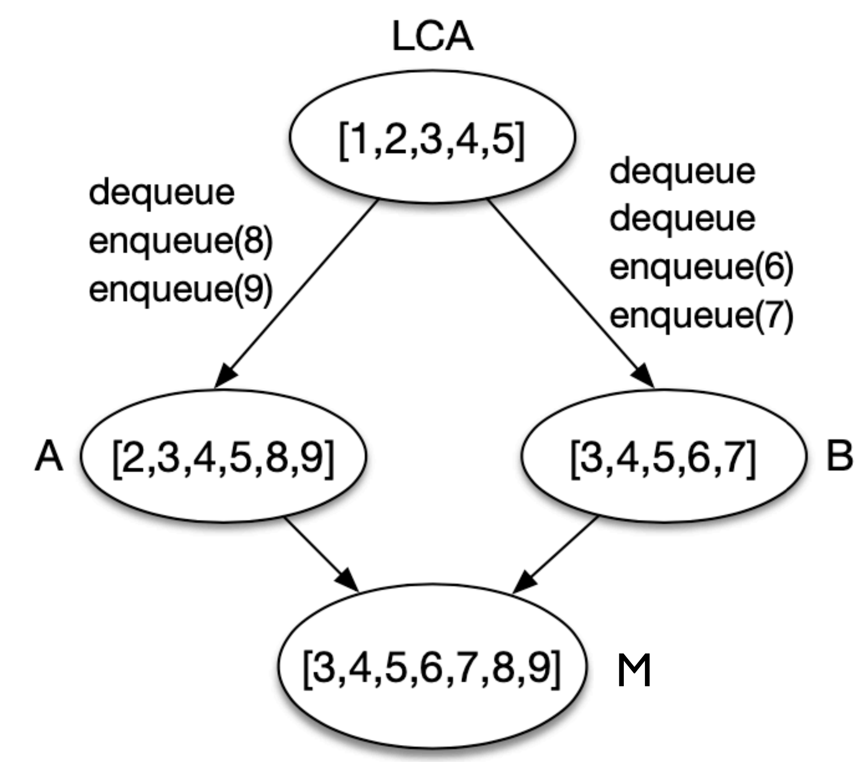
- Replicated queue with *at-least-once* dequeue semantics
  - ★ First verified queue RDT!



- Our aim is to have O(1) enqueue and dequeue and O(n) merge

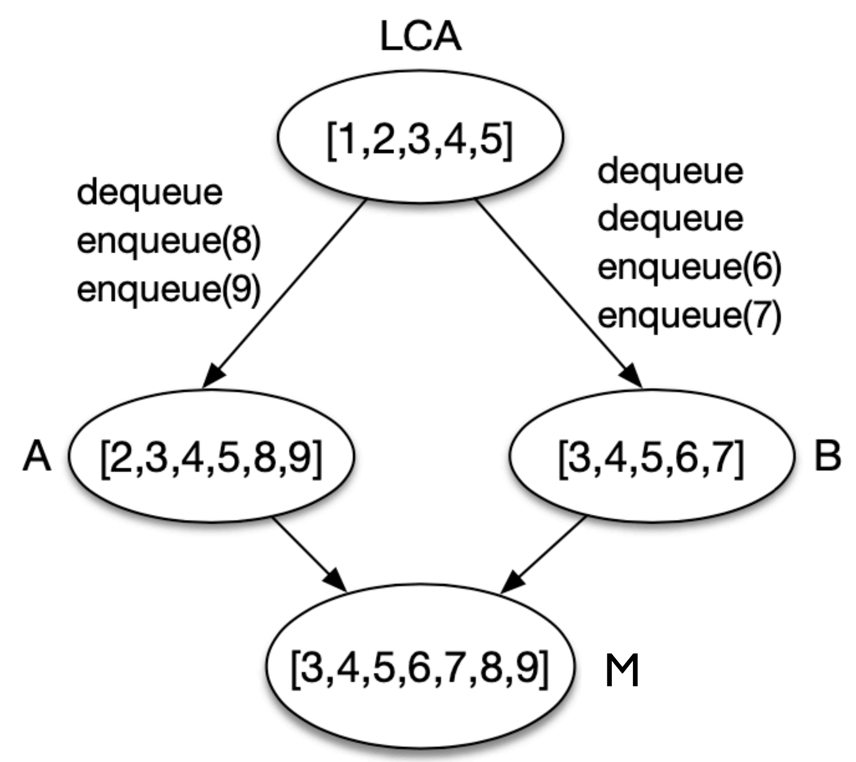
Mergeable Queues

- Implementation
  - ★ Uses *two-list functional queue* implementation
    - ✦ amortised O(1) enqueue and dequeue operations
  - ★ Merge uses *longest common contiguous subsequence* algorithm — O(n)



Mergeable Queues

- Implementation
  - ★ Uses *two-list functional queue* implementation
    - ✦ amortised O(1) enqueue and dequeue operations
  - ★ Merge uses *longest common contiguous subsequence* algorithm — O(n)



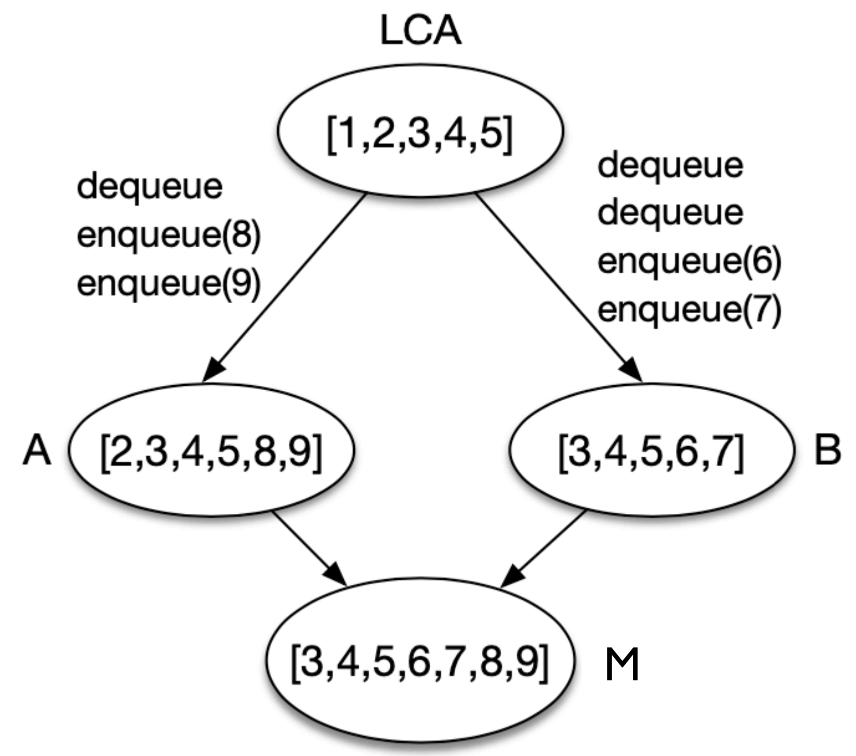
- Specification

1. Any element *popped* in either A or B does not remain in M
2. Any element *pushed* into either A or B appears in M
3. An element that remains *untouched* in LCA, A, B remains in M
4. *Order* of pairs of elements in LCA, A, B must be preserved in M, if those elements are present in M.

Mergeable Queues

- Implementation
  - ★ Uses *two-list functional queue* implementation
    - ✦ amortised O(1) enqueue and dequeue operations
  - ★ Merge uses *longest common contiguous subsequence* algorithm — O(n)
- Specification

1. Any element *popped* in either A or B does not remain in M
2. Any element *pushed* into either A or B appears in M
3. An element that remains *untouched* in LCA, A, B remains in M
4. *Order* of pairs of elements in LCA, A, B must be preserved in M, if those elements are present in M.



Implementation far removed from the specification!

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

Summary

Summary

- Programming and proving with RDTs is complicated due to *concurrency* and the lack of suitable *programming abstractions*

Summary

- Programming and proving with RDTs is complicated due to *concurrency* and the lack of suitable *programming abstractions*
- MRDTs simplify RDTs by implementing them as extensions of sequential data types
  - ★ Reasoning about correctness is still hard

Summary

- Programming and proving with RDTs is complicated due to *concurrency* and the lack of suitable *programming abstractions*
- MRDTs simplify RDTs by implementing them as extensions of sequential data types
  - ★ Reasoning about correctness is still hard
- Peepul is an F* library for certified MRDTs
  - ★ Replication-aware simulation for proving complex MRDTs
  - ★ Complex MRDTs can be constructed and proved using simpler MRDTs

Summary

- Programming and proving with RDTs is complicated due to *concurrency* and the lack of suitable *programming abstractions*
- MRDTs simplify RDTs by implementing them as extensions of sequential data types
  - ★ Reasoning about correctness is still hard
- Peepul is an F* library for certified MRDTs
  - ★ Replication-aware simulation for proving complex MRDTs
  - ★ Complex MRDTs can be constructed and proved using simpler MRDTs
- F* allows us to strike a balance between automated and interactive proofs
  - ★ Extract to OCaml and run on Irmin!

Backup Slides

Queue Performance