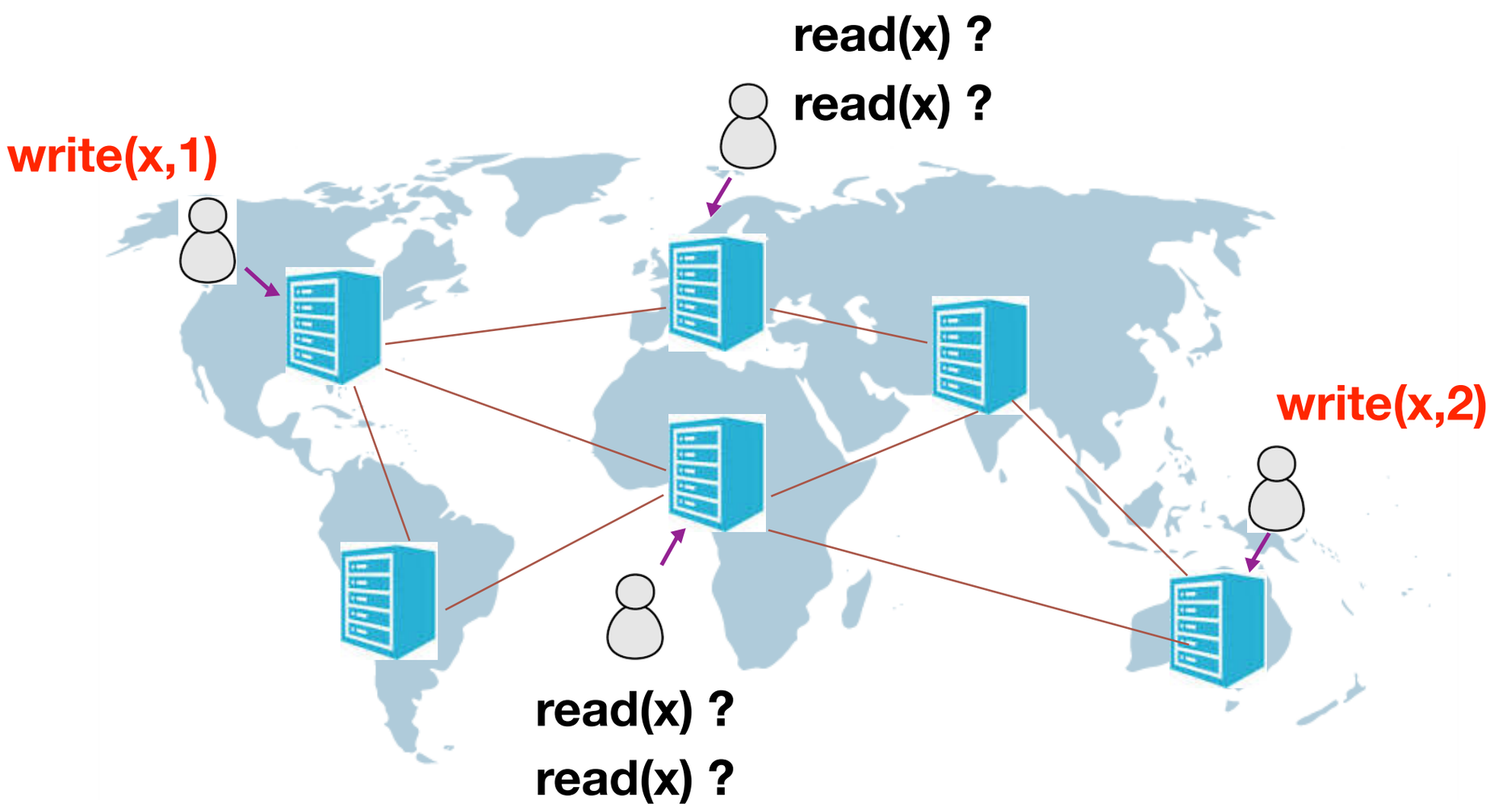
Weak Consistency

Constantin Enea
Ecole Polytechnique

Replicated objects

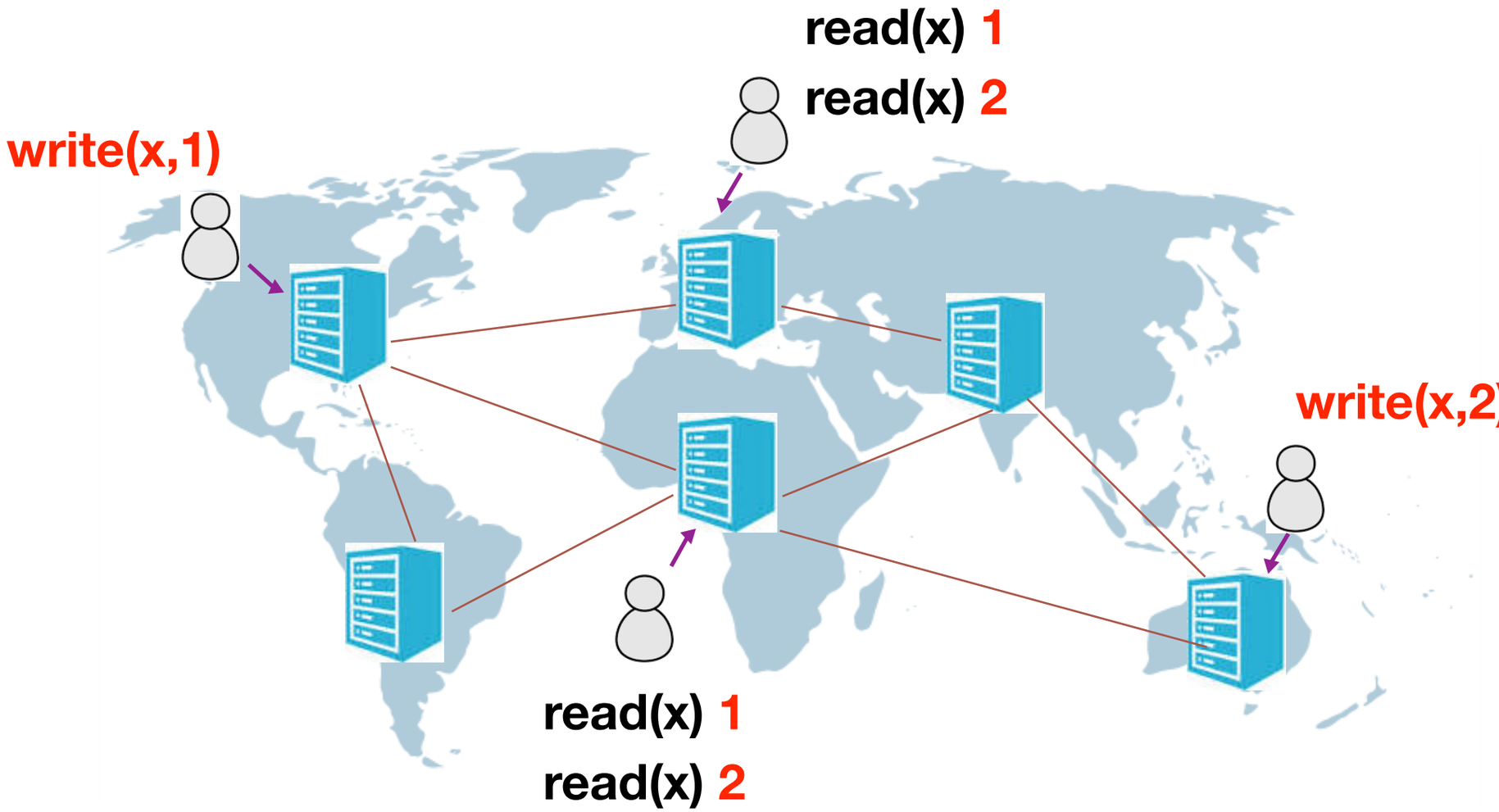
Distributed systems



Conflicting concurrent updates: how are they observed on different replicas ?

Adversarial environments: crashes, network partitions

Pessimistic Replication

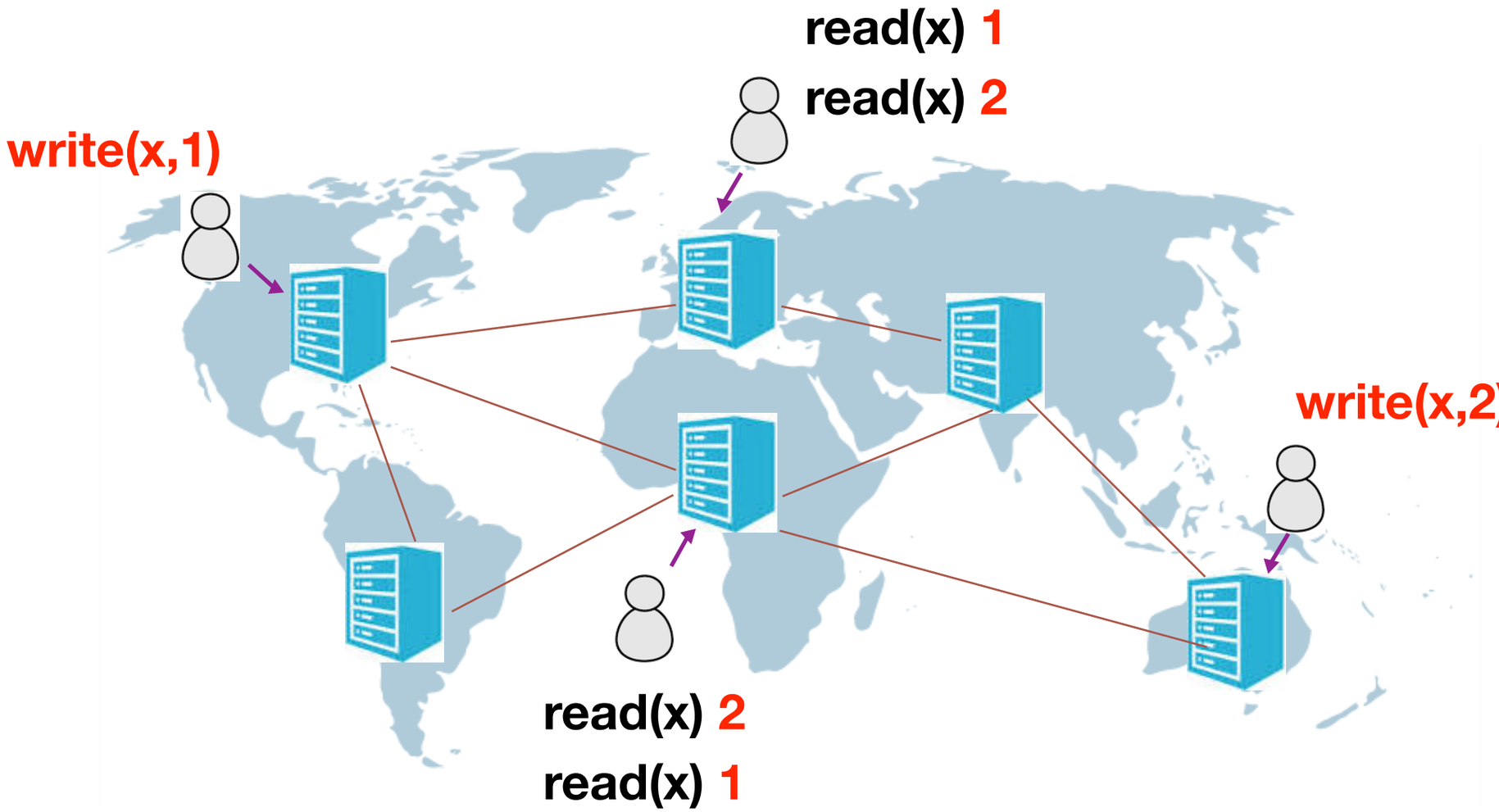


Using consensus algorithms to agree on an order between conflicting concurrent updates

✓ strong consistency ✗ availability

CAP theorem [Gilbert et al.'02]: strong cons. + availability + partition tolerance is impossible

Optimistic Replication



Each update is applied on the **local** replica and propagated **asynchronously** to other replicas

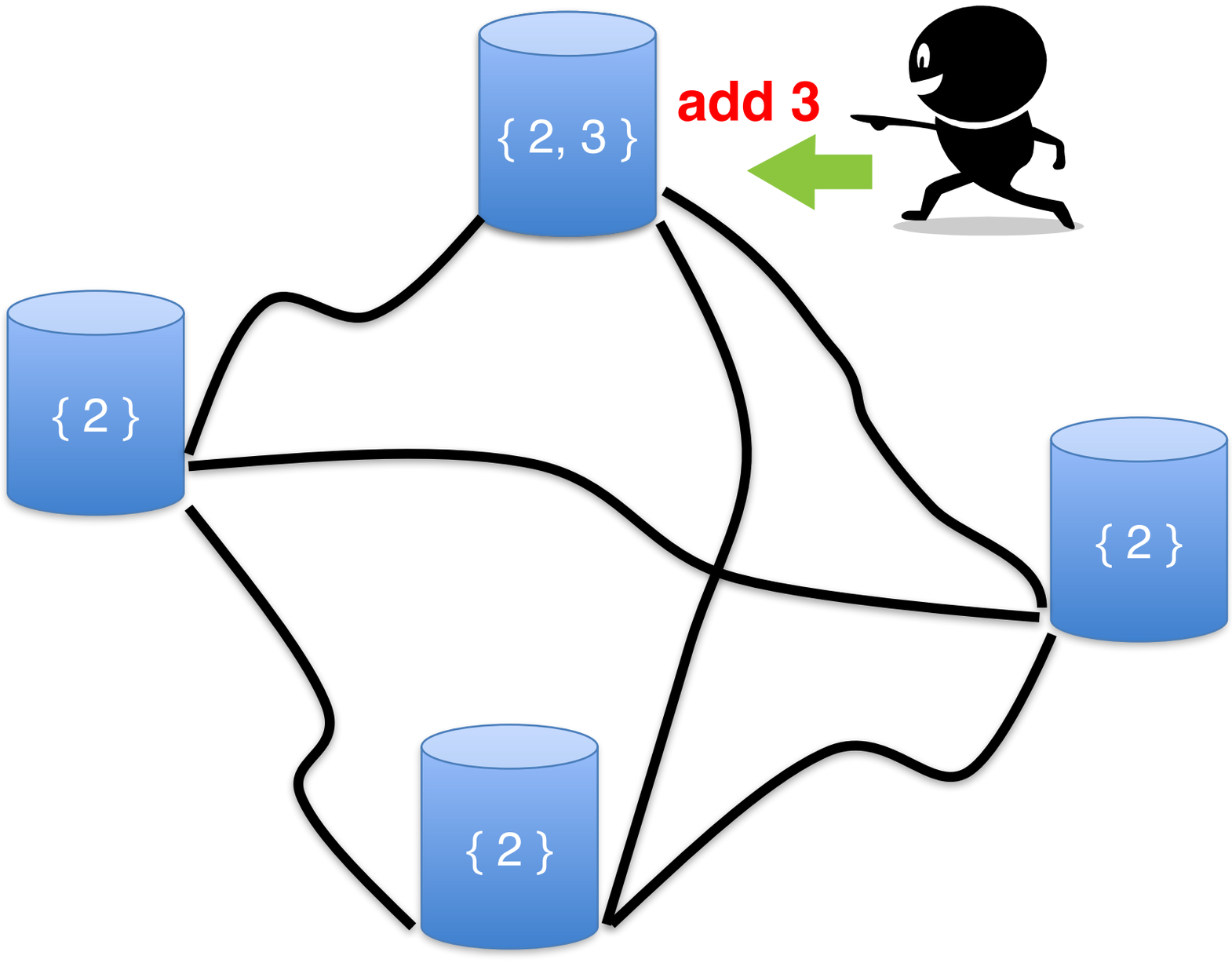
✗ strong consistency ✓ availability

Replicas may store different versions of data: **weak consistency**

Optimistic data replication

Optimistic replication: replicas are allowed to diverge

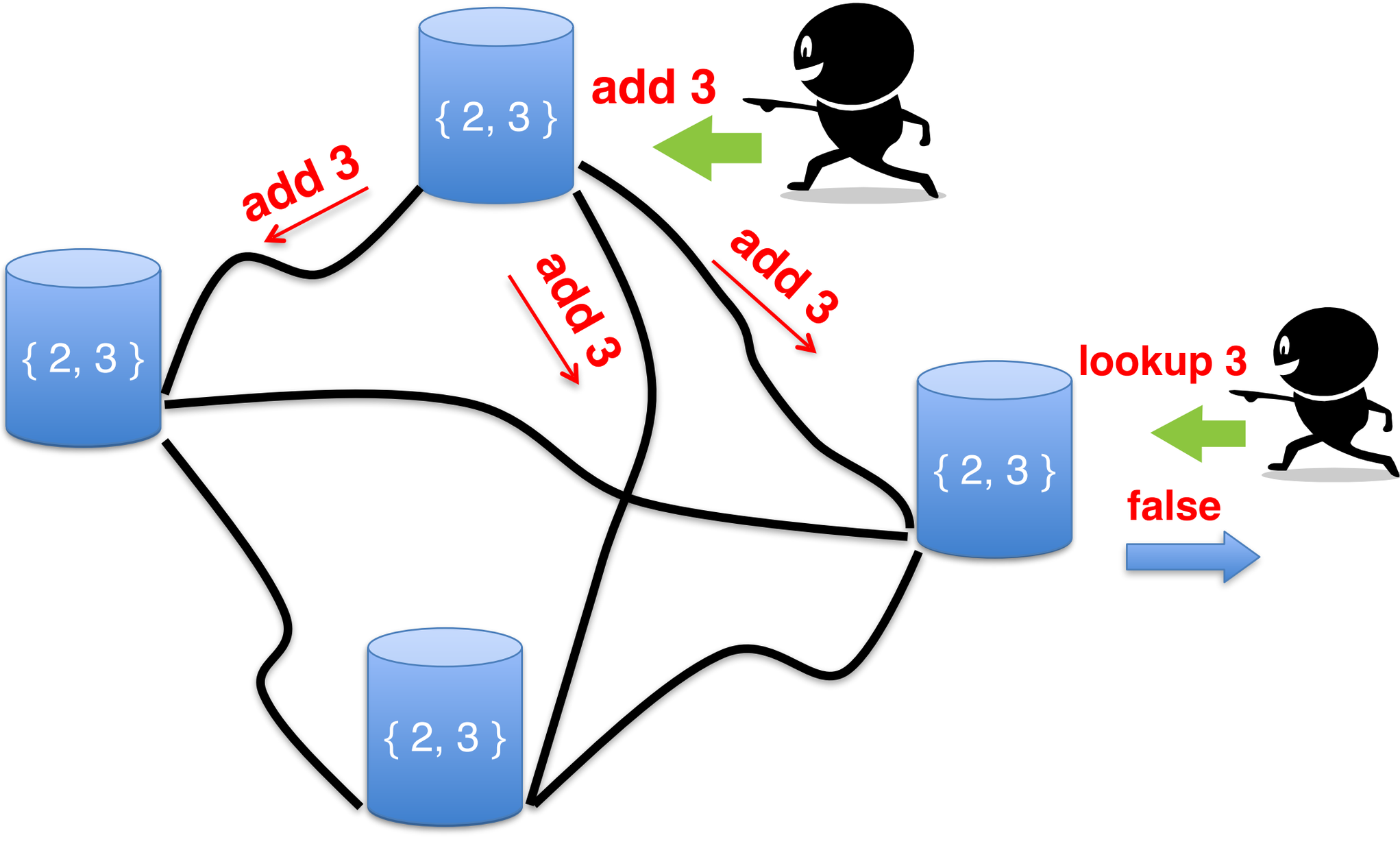
– operations are applied immediately at the submission site



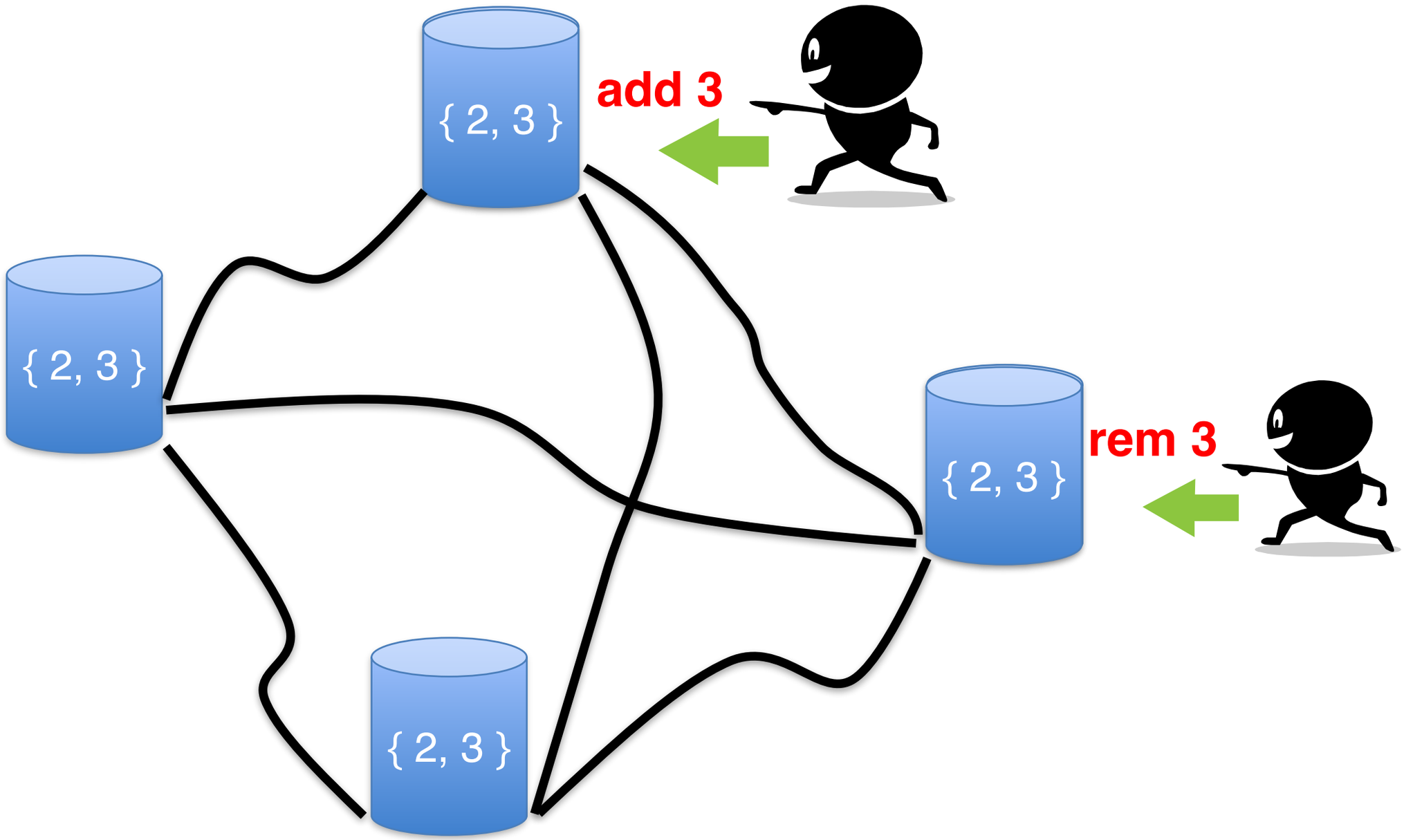
Optimistic data replication

Optimistic replication: replicas are allowed to diverge

- operations are applied immediately at the submission site
- in the background, sites exchange and apply remote operations



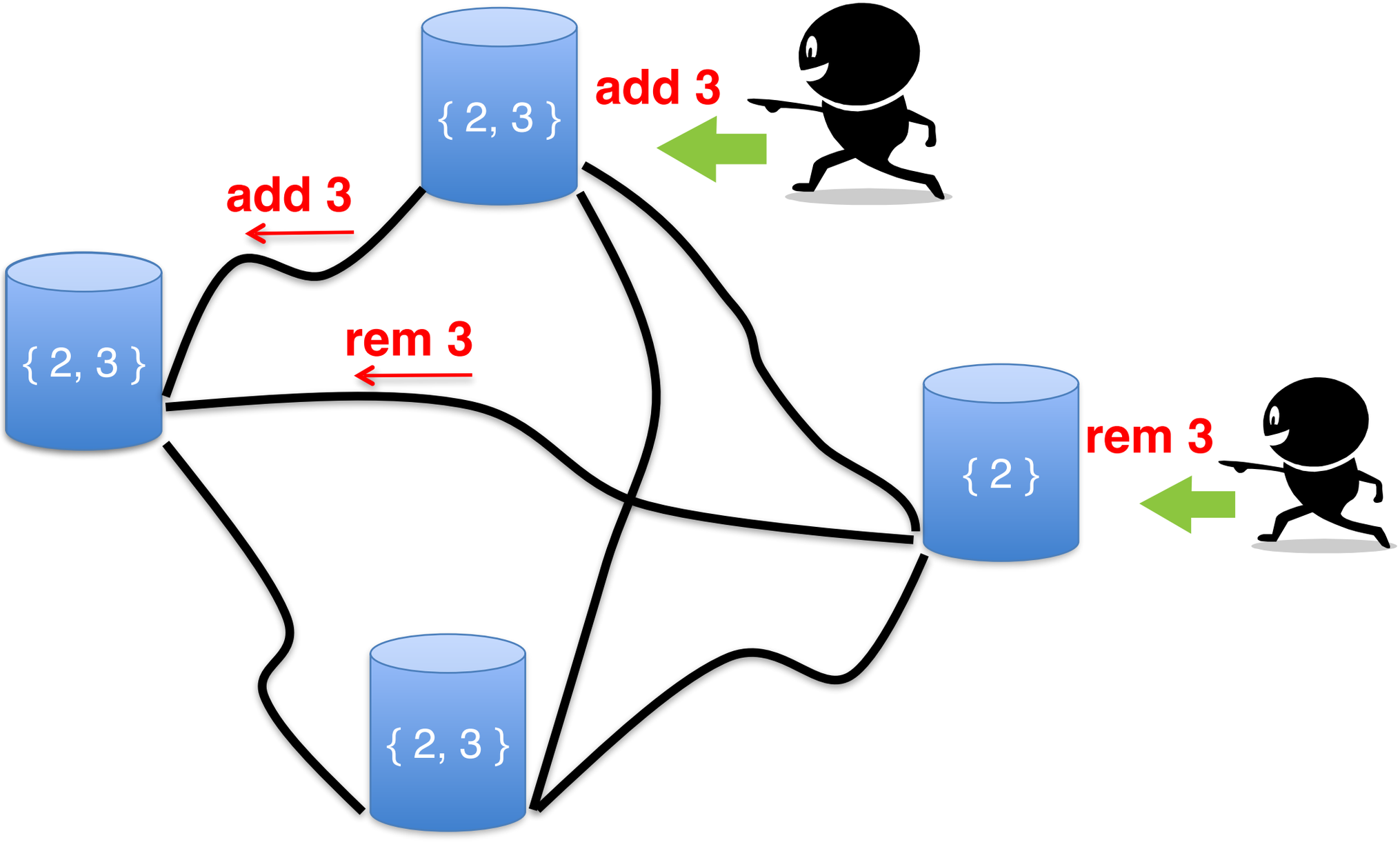
Concurrent operations



Concurrent operations

Solving conflicts between concurrent operations

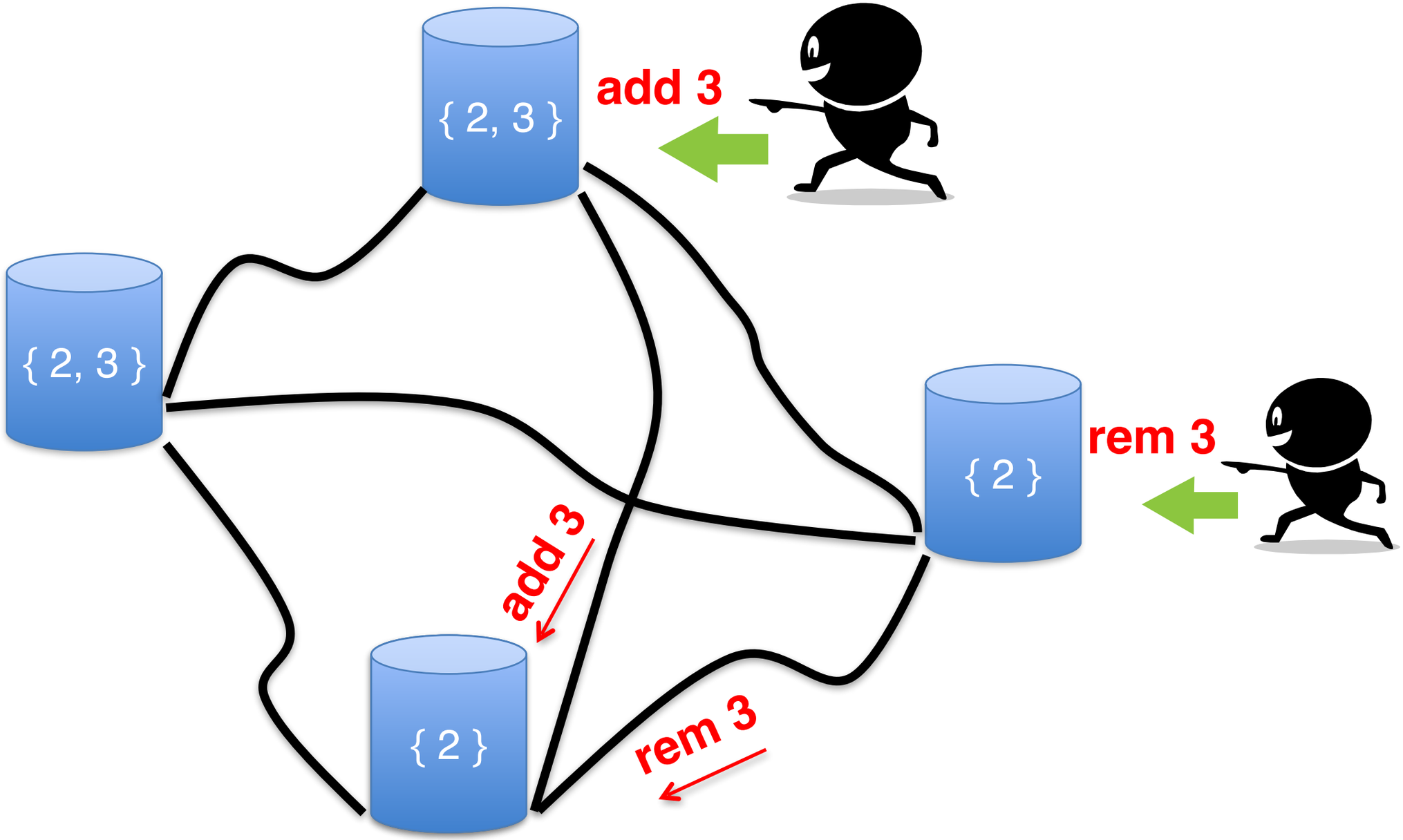
- **speculate and roll-back, e.g., Google App Engine Datastore**



Concurrent operations

Solving conflicts between concurrent operations

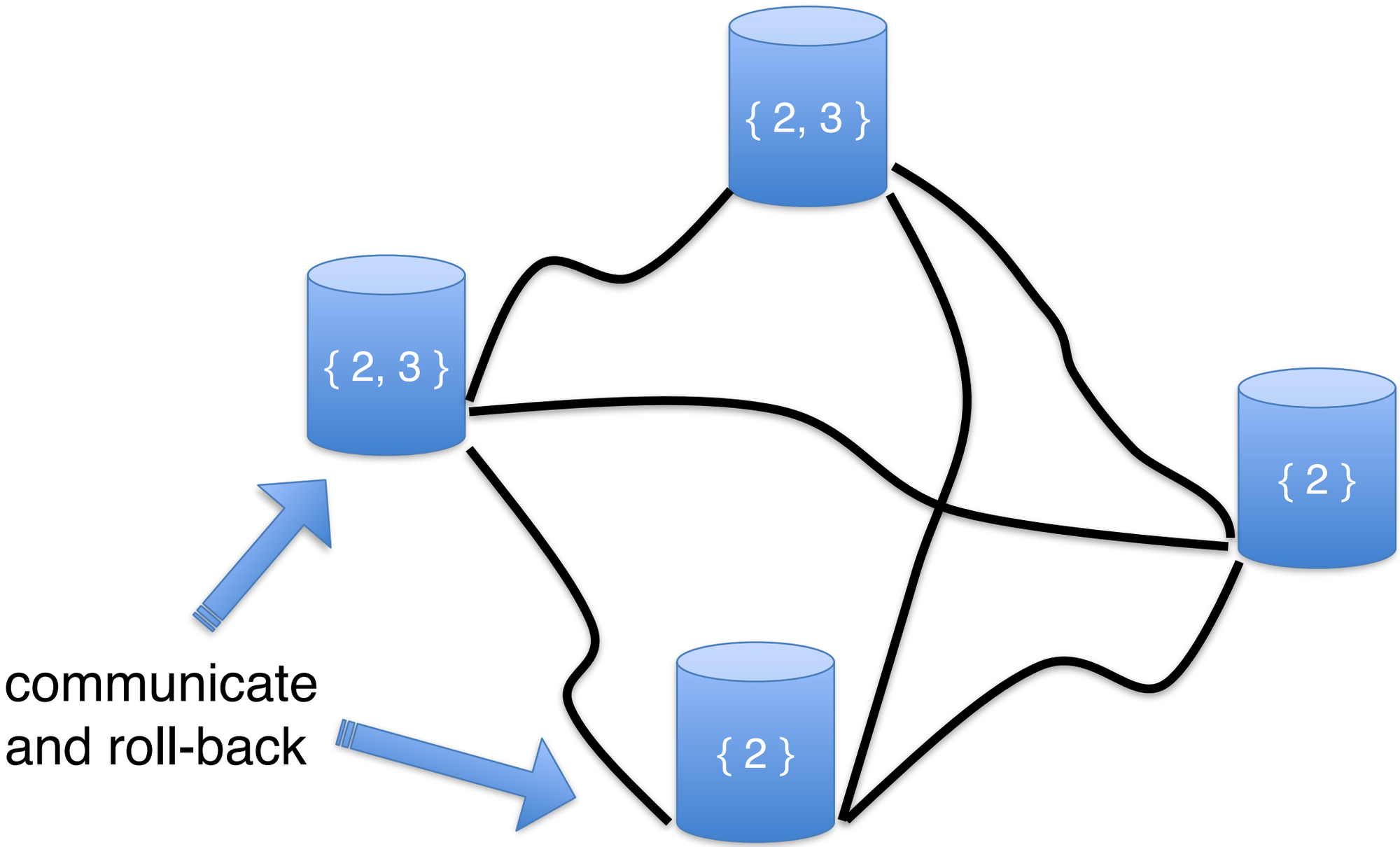
- **speculate and roll-back, e.g., Google App Engine Datastore**



Concurrent operations

Solving conflicts between concurrent operations

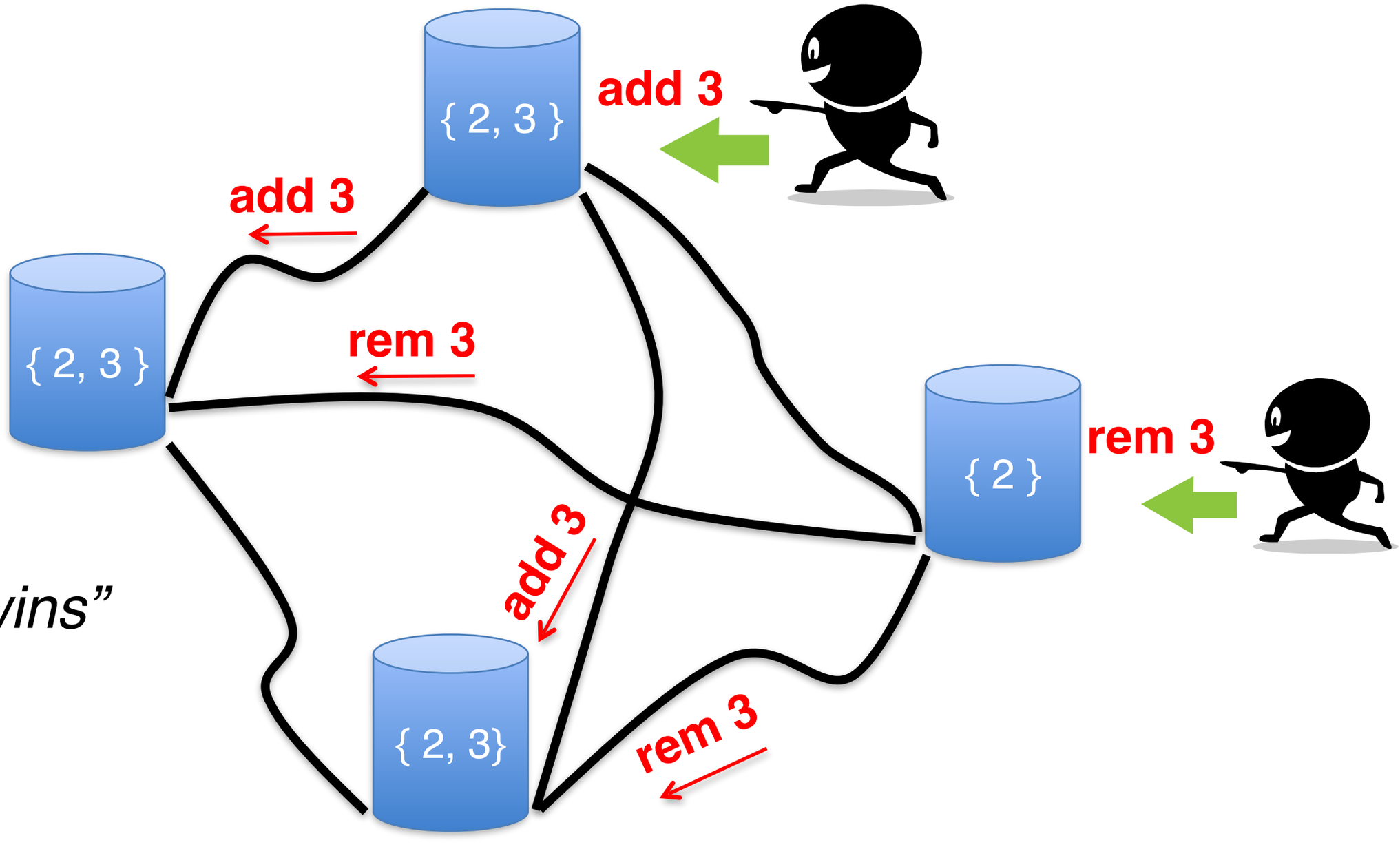
- **speculate and roll-back, e.g., Google App Engine Datastore**



Concurrent operations

Solving conflicts between concurrent operations

- speculate and eventually, roll-back, e.g., Google App Engine Datastore
- **convergent conflict resolution, e.g., CRDTs [Shapiro et al.'11]**



"add wins"

Formal verification of Opt. Repl. Syst.

- Correct operations ? Allowed level of consistency between replicas ?
  - by CAP theorem [Gilbert et al.'02], achieving strong consistency (linearizability) is impossible
  - various correctness criteria: eventual consistency, causal consistency, etc

Example: Key-value map

```
struct Timestamp(number: nat ; rid: nat) ;
function lessthan(Timestamp(n1,rid1), Timestamp(n2,rid2)) : boolean {
  return (n1 < n2) ∨ (n1 == n2 ∧ rid1 < rid2) ;
}

message Update(key: Key, val: Val, ts: Timestamp) : reliable

role Replica(rid: nat) {
  var localclock: nat ;
  var store: pmap⟨Key, pair⟨Val,Timestamp⟩⟩ ;

  operation read(key: Key) {
    match store[key] with
      ⊥→ { return undef;}
      (val,ts) → { return val ; }
    }
}
operation write(key: Key, val: Val) {
  localclock++ ; // advance logical clock
  store[key] := (val,ts) ;
  send Update(key,val,Timestamp(localclock,rid)) ;
  return ok ;
}
receive Update(key,val,ts) {
  if (store[key] = ⊥∨ store[key].second.lessthan(ts))
        store[key] := (val, ts) ;
  if (ts.number > localclock) // keep up with time
        clock := ts.number ;
}
```

Example: OR-Set

payload set E, set T

initial $\varnothing, \varnothing$
query $contains$ (element e) : boolean b
    let $b = (\exists n : (e, n) \in E)$
query $elements$ () : set S
    let $S = \{e | \exists n : (e, n) \in E\}$
update add (element e)
    prepare (e)
        let $n = unique()$
    effect (e, n)
        $E := E \cup \{(e, n)\} \setminus T$
update $remove$ (element e)
    prepare (e)
        let $R = \{(e, n) | \exists n : (e, n) \in E\}$
    effect (R)
        $E := E \setminus R$
        $T := T \cup R$

Example: ABD register [Attiya, Bar-Noy, Dolev 1995]

1: **local variables:**
2: *sn*, initially 0 {for readers and writers, sequence number used to identify messages}
3: *val*, initially v_0 {for servers, latest register value}
4: *ts*, initially $(0, 0)$ {for servers, timestamp of current register value, (integer, process id) pair}

5: **function queryPhase():**
6: *sn*++
7: broadcast ⟨"query",*sn*⟩
8: wait for $\geq \frac{n+1}{2}$ reply msgs to this query msg
9: (*v*,*u*) := pair in reply msg with largest timestamp
10: return (*v*,*u*)

11: **when ⟨"query",*s*⟩ is received from *q*:**
12: send ⟨"reply",*val*,*ts*,*s*⟩ to *q*

13: **function updatePhase(*v*,*u*):**
14: *sn*++
15: broadcast ⟨"update",*v*, *u*, *sn*⟩
16: wait for $\geq \frac{n+1}{2}$ ack msgs for this update msg
17: return

18: **when ⟨"update",*v*,*u*,*s*⟩ is received from *q*:**
19: if $u > ts$ then (*val*,*ts*) := (*v*,*u*)
20: send ⟨"ack",*s*⟩ to *q*

21: **Read():**
22: (*v*,*u*) := queryPhase()
23: updatePhase(*v*,*u*) {write-back}
24: return *v*

25: **Write(*v*) for process with id *i*:**
26: $(-, (t, -))$:= queryPhase() {just need integer in timestamp}
27: updatePhase(v,$(t + 1, i)$)
28: return

Formal Definitions

- Histories
- Abstract Executions
- Operation Context
- Replicated Data Types
- Return-value Consistency

(Using the formal framework in “Principles of Eventual Consistency” by S. Burckhardt)

Histories

Definition 3.1 (History)**.** A *history* is an event graph $(E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss})$ where

(h1) $\mathsf{op} : E \to \mathsf{Operations}$ describes the operation of an event.

(h2) $\mathsf{rval} : E \to \mathsf{Values} \cup \{\nabla\}$ describes the value returned by the operation, or the special symbol ∇ ($\nabla \notin \mathsf{Values}$) to indicate that the operation never returns.

(h3) rb is a natural partial order on E, the *returns-before* order.

(h4) ss is an equivalence relation on E, the *same-session* relation.

Definition 3.2 (Well-formed History)**.** A history $(E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss})$ is *well-formed* if

(h5) $x \xrightarrow{\mathsf{rb}} y$ implies $\mathsf{rval}(x) \neq \nabla$ for all $x, y \in E$.

(h6) for all $a, b, c, d \in E$: $(a \xrightarrow{\mathsf{rb}} b \ \wedge \ c \xrightarrow{\mathsf{rb}} d) \Rightarrow (a \xrightarrow{\mathsf{rb}} d \ \vee \ c \xrightarrow{\mathsf{rb}} b)$.

(h7) For each session $[e] \in E/\approx_{ss}$, the restriction $\mathsf{rb}|_{[e]}$ is an enumeration.

Abstract Executions

Definition 3.3 (Abstract Executions)**.** An *abstract execution* is an event graph $(E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss}, \mathsf{vis}, \mathsf{ar})$ such that

(a1) $(E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss})$ is a history.

(a2) vis is an acyclic and natural relation.

(a3) ar is a total order.

Definition 4.4 (Operation Context)**.** An *operation context* is a finite event graph $C = (E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})$ where $\mathsf{op} : E \to \mathsf{Operations}$ describes the operation of each event, vis is an acyclic relation representing visibility among the elements of E, and ar is a total order representing arbitration of the elements in E. We let $\mathcal{C}$ be the set of all operation contexts.

Replicated Data Types

Definition 4.5 (Replicated Data Type)**.** A *replicated data type* $\mathcal{F}$ is a function $\mathsf{Operations} \times \mathcal{C} \quad \rightarrow \quad \mathsf{Values}$ that, given an operation o and an operation context C, specifies the expected return value $\mathcal{F}(o, C)$ to be used when performing o in context C, and which does not depend on the events, *i.e.* is the same for isomorphic (as in Definition 2.2) contexts: $C \simeq C' \;\Rightarrow\; \mathcal{F}(o, C) = \mathcal{F}(o, C')$ for all o, C, C'.

Replicated Counter: a read returns the number of increment operations in the context

$$\mathcal{F}_{ctr}(\mathsf{rd}, (E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})) = \left|\{e' \in E \mid \mathsf{op}(e') = \mathsf{inc}\}\right|$$

Last-Writer-Wins Register: a read returns the value of the last write in the context (w.r.t. arbitration order), or “undef”, if there is no write

$$\mathcal{F}_{reg}(\mathsf{rd}, (E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})) = \begin{cases} undef & \text{if } \mathsf{writes}(E) = \emptyset \\ v & \text{if } \mathsf{op}(\max_{\mathsf{ar}} \mathsf{writes}(E)) = \mathsf{wr}(v) \end{cases}$$

Multi-Value Register: a read returns a set of values, one for each write in the context that has not been overwritten by some other write

$$\mathcal{F}_{mvr}(\mathsf{rd}, (E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})) =$$
$$\{v \mid \exists e \in E : \mathsf{op}(e) = \mathsf{wr}(v) \text{ and } \forall e' \in \mathsf{writes}(E) : e \overset{\mathsf{vis}}{\not\longrightarrow} e'\}$$

OR-Set ?

payload set E, set T

initial $\varnothing, \varnothing$
query $contains$ (element e) : boolean b
    let $b = (\exists n : (e, n) \in E)$
query $elements$ () : set S
    let $S = \{e | \exists n : (e, n) \in E\}$
update add (element e)
    prepare (e)
        let $n = unique()$
    effect (e, n)
        $E := E \cup \{(e, n)\} \setminus T$
update $remove$ (element e)
    prepare (e)
        let $R = \{(e, n) | \exists n : (e, n) \in E\}$
    effect (R)
        $E := E \setminus R$
        $T := T \cup R$

Return-Value Consistency

Definition 4.8. For a replicated data type $\mathcal{F}$, we define the return value consistency guarantee as

$$\text{RVAL}(\mathcal{F}) \stackrel{\text{def}}{=} \quad \forall e \in E : \quad \mathsf{rval}(e) = \mathcal{F}(\mathsf{op}(e), \mathsf{context}(A, e))$$

where context is defined as follows:

Definition 4.9. Let $A = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss}, \mathsf{vis}, \mathsf{ar})$ be an abstract execution containing an event $e \in E$. Then

$$\mathsf{context}(A, e) \stackrel{\text{def}}{=} A|_{\mathsf{vis}^{-1}(e), \mathsf{op}, \mathsf{vis}, \mathsf{ar}}$$

Formal Definitions

$$\textsc{ReadMyWrites} \stackrel{\text{def}}{=} (\mathsf{so} \subseteq \mathsf{vis})$$

$$\textsc{MonotonicReads} \stackrel{\text{def}}{=} (\mathsf{vis}\,;\mathsf{so}) \subseteq \mathsf{vis}$$

$$\textsc{ConsistentPrefix} \stackrel{\text{def}}{=} (\mathsf{ar}\,;(\mathsf{vis} \cap \neg\mathsf{ss})) \subseteq \mathsf{vis}$$

$$\textsc{NoCircularCausality} \stackrel{\text{def}}{=} \mathsf{acyclic}(\mathsf{hb})$$

$$\textsc{CausalVisibility} \stackrel{\text{def}}{=} (\mathsf{hb} \subseteq \mathsf{vis})$$

$$\textsc{CausalArbitration} \stackrel{\text{def}}{=} (\mathsf{hb} \subseteq \mathsf{ar})$$

$$\mathsf{hb} \stackrel{\text{def}}{=} (\mathsf{so} \cup \mathsf{vis})^{+}$$

$$\textsc{Causality} \stackrel{\text{def}}{=} \textsc{CausalVisibility} \wedge \textsc{CausalArbitration}$$

$$\textsc{SingleOrder} \stackrel{\text{def}}{=} \exists E' \subseteq \mathsf{rval}^{-1}(\nabla) : \mathsf{vis} = \mathsf{ar} \setminus (E' \times E)$$

$$\textsc{Realtime} \stackrel{\text{def}}{=} \mathsf{rb} \subset \mathsf{ar}$$

$$\textsc{Linearizability}(\mathcal{F}) \stackrel{\text{def}}{=} \textsc{SingleOrder} \wedge \textsc{Realtime} \wedge \textsc{RVal}(\mathcal{F})$$

$$\textsc{SequentialConsistency}(\mathcal{F}) \stackrel{\text{def}}{=} \textsc{SingleOrder} \wedge \textsc{ReadMyWrites} \wedge \textsc{RVal}(\mathcal{F})$$

$$\textsc{CausalConsistency}(\mathcal{F}) \stackrel{\text{def}}{=} \textsc{EventualVisibility} \wedge \textsc{Causality} \wedge \textsc{RVal}(\mathcal{F})$$

$$\textsc{BasicEventualConsistency}(\mathcal{F}) \stackrel{\text{def}}{=} \textsc{EventualVisibility} \wedge \textsc{NoCircularCausality} \wedge \textsc{RVal}(\mathcal{F})$$

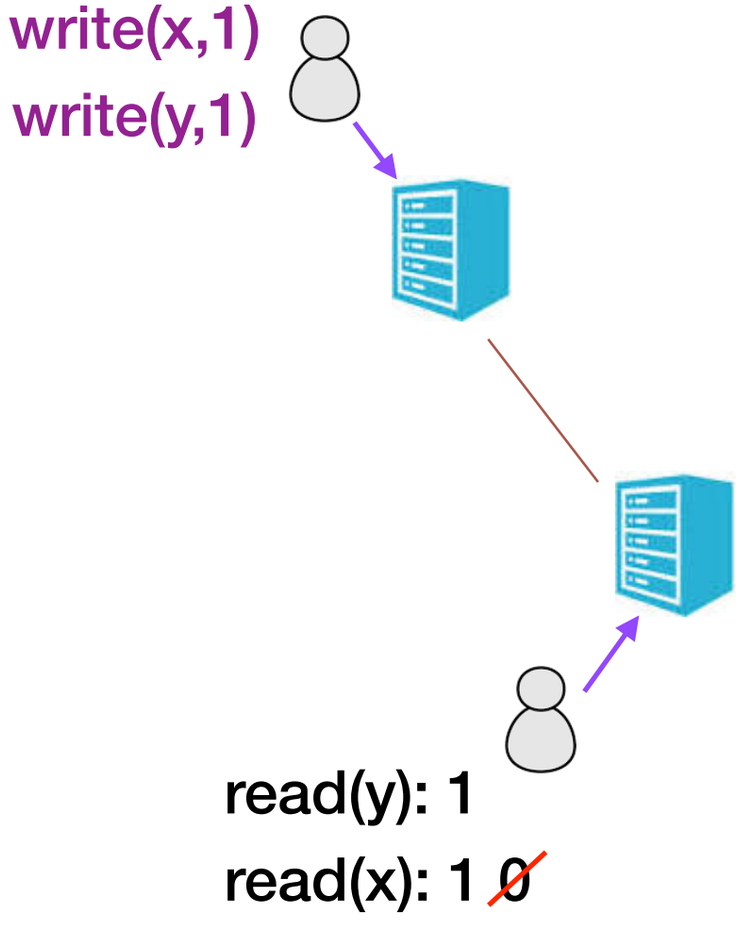
EventualVisibility: the nb. of operations not “seeing” an operation is finite

Causal Consistency (CC)

[Lamport'78]

If an update is visible, then all the updates is dependent on should be also visible

- **write(x,1)** and **write(y,1)** are **causally dependent**:

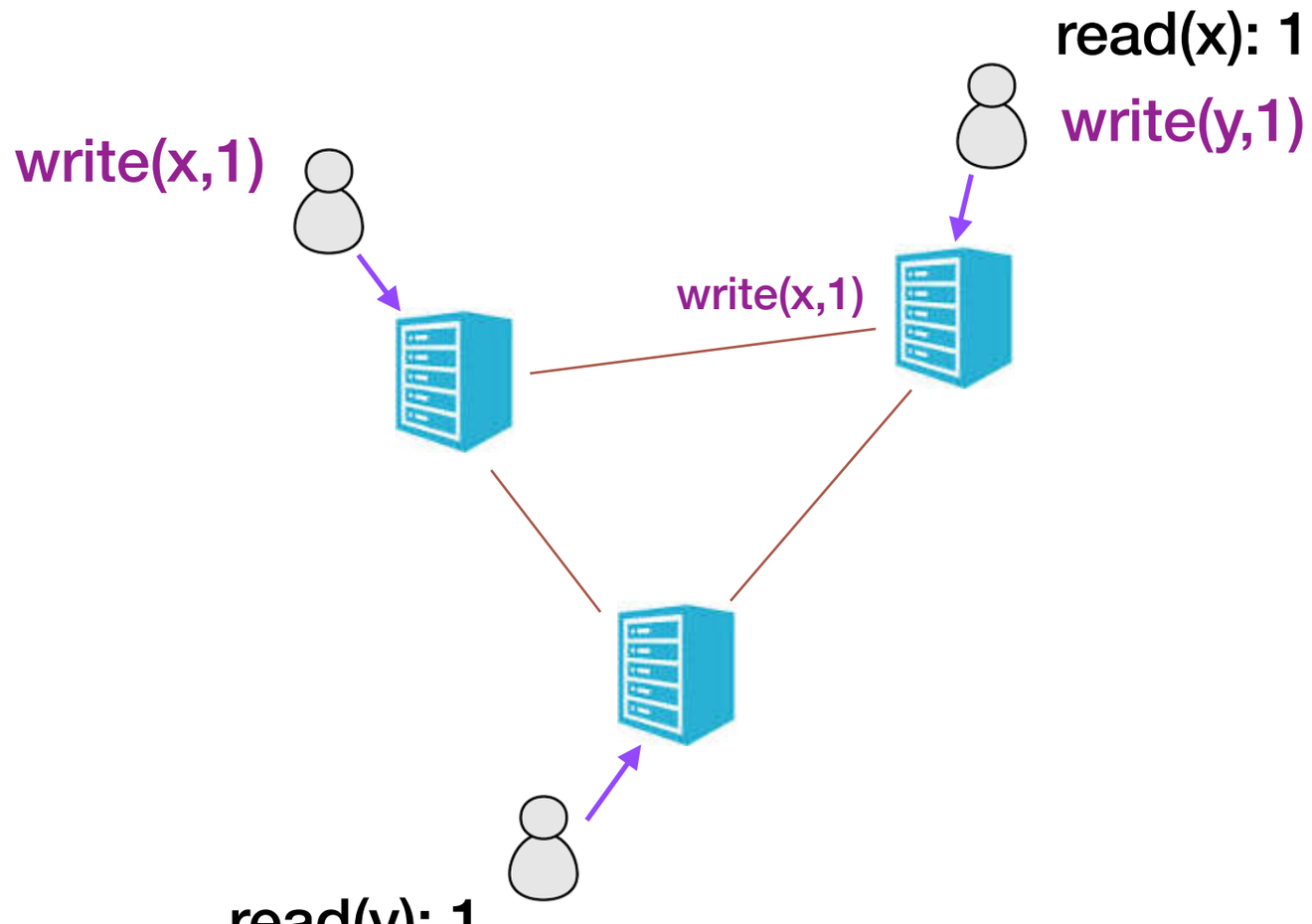


Causal Consistency (CC)

[Lamport'78]

If an update is visible, then all the updates is dependent on should be also visible

- **write(x,1)** and **write(y,1)** are **causally dependent**:

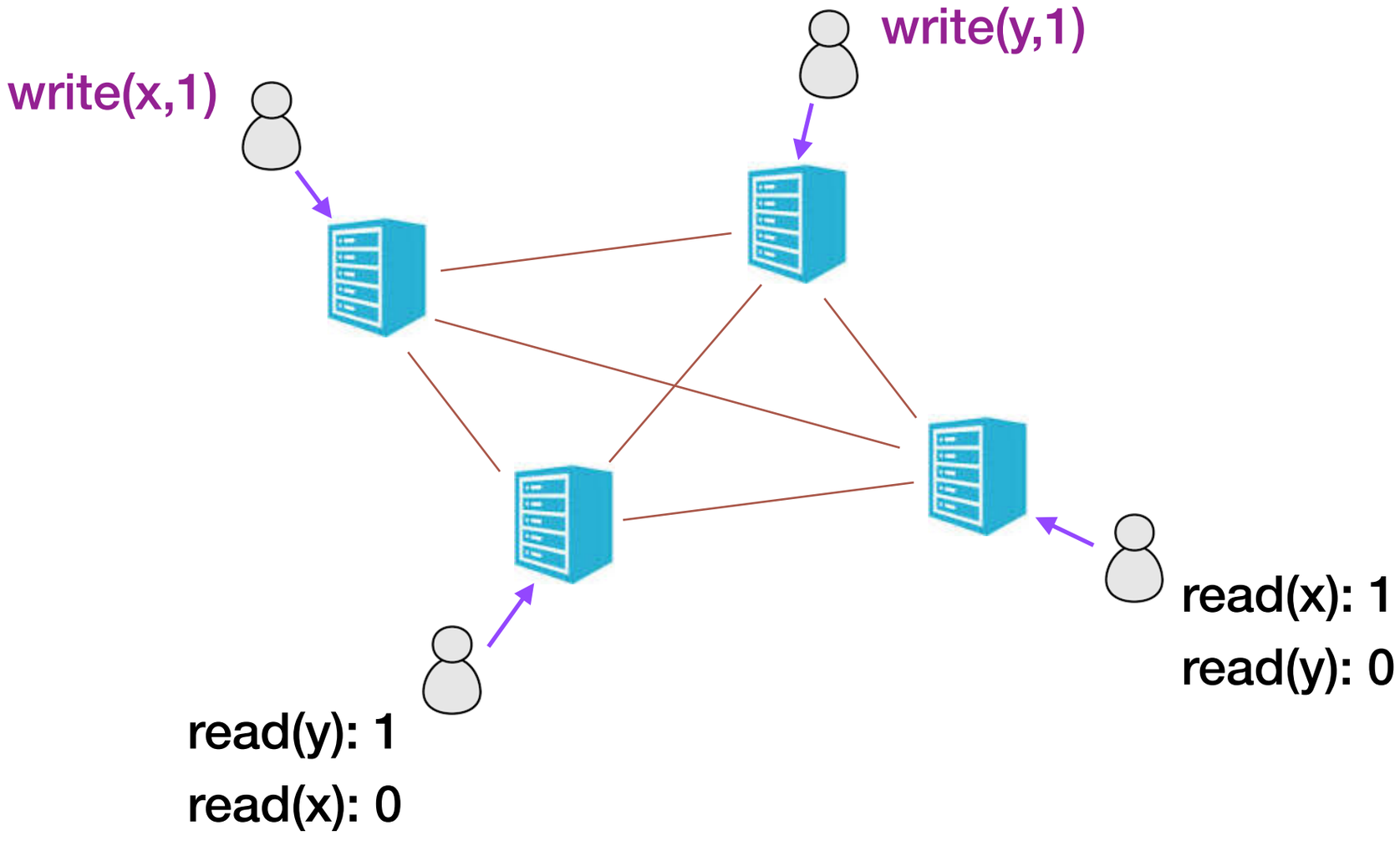


Causal Consistency (CC)

[Lamport'78]

If an update is visible, then all the updates is dependent on should be also visible

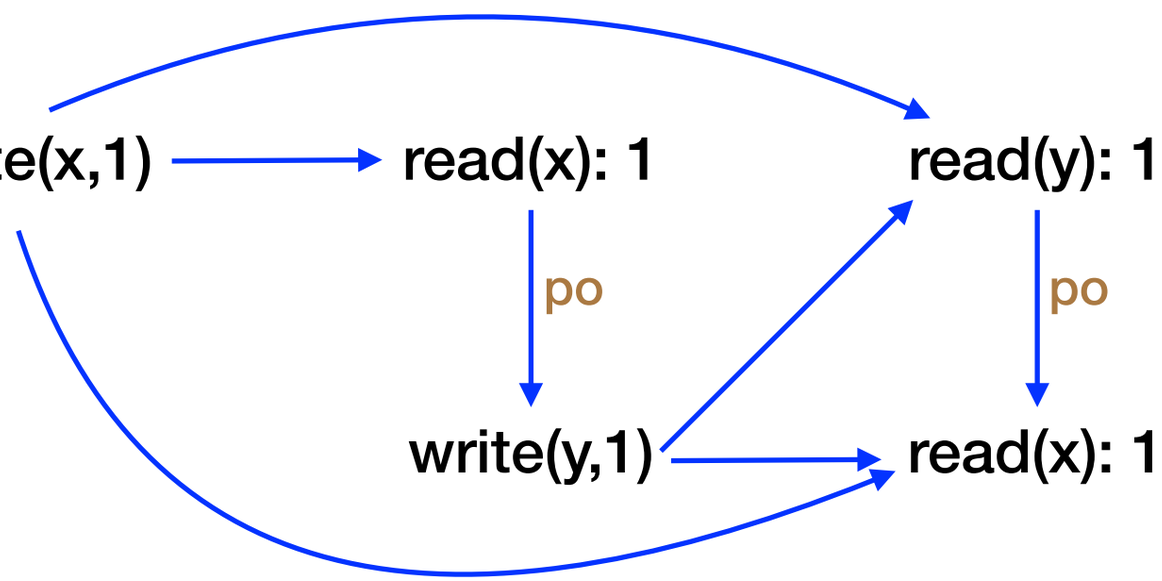
- **write(x,1)** and **write(y,1)** are **concurrent**:



Formalization: Visibility

[Burckhardt et al.'14]

A is visible to operation **B at replica R** if the effect of A had been included in R at the time when B was executed



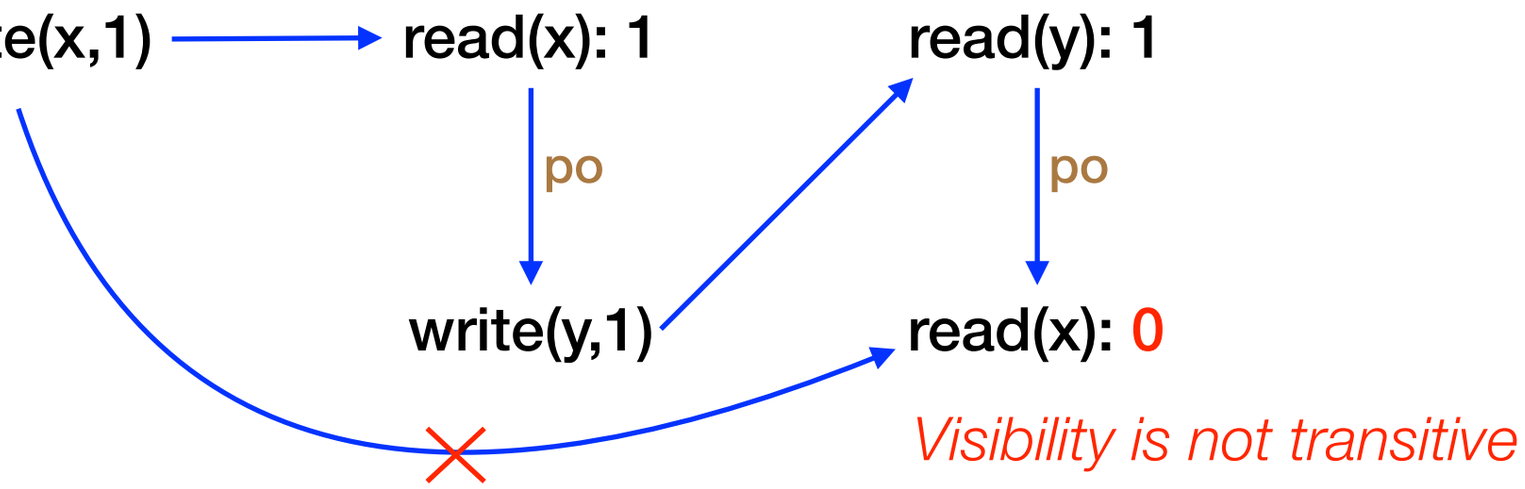
∃ **vis**. **vis** ⊇ **po** ∧ **vis** transitive

∧ ∀ read. ``**return value** is consistent with the set of visible ops"

Formalization: Visibility

[Burckhardt et al.'14]

A is visible to operation **B at replica R** if the effect of A had been included in R at the time when B was executed



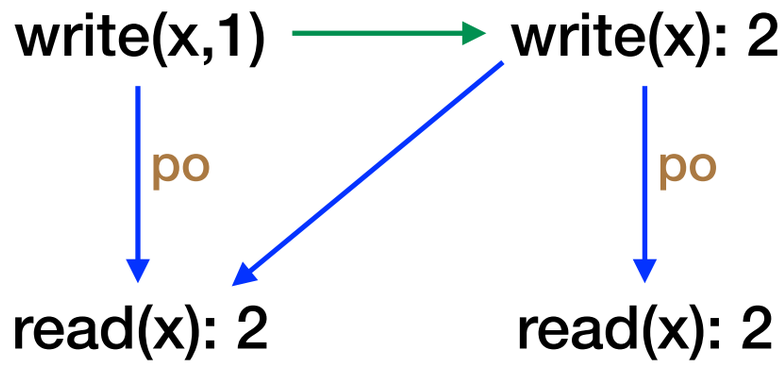
∃ **vis**. **vis** ⊇ **po** ∧ **vis** transitive

∧ ∀ read. ``**return value** is consistent with the set of visible ops"

Formalization: Arbitration

[Burckhardt et al.'14]

Arbitration: conflict resolution between concurrent writes using timestamps



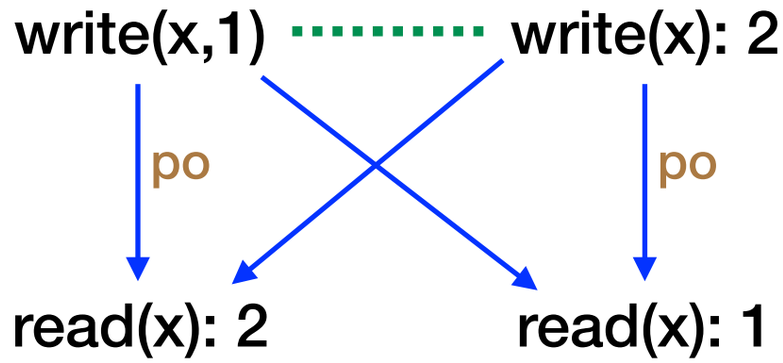
∃ **vis**, **arb**. **arb** ⊇ **vis** ⊇ **po** ∧ **vis** transitive ∧ **arb** total order

∧ ∀ read. ``**return value** = value of the last visible write in arbitration order"

Formalization: Arbitration

[Burckhardt et al.'14]

Arbitration: conflict resolution between concurrent writes using timestamps



No valid arbitration

∃ **vis**, **arb**. **arb** ⊇ **vis** ⊇ **po** ∧ **vis** transitive ∧ **arb** total order

∧ ∀ read. ``**return value** = value of the last visible write in arbitration order"

Checking CC

[POPL'17]

Checking CC for a single execution is NP-complete (proof on whiteboard)

Checking CC for a finite-state implementation (given as a regular language) and a regular specification is undecidable

Characterizing CC

[POPL'17]

∃ **vis**, **arb**. **arb** ⊇ **vis** ⊇ **po** ∧ **vis** transitive ∧ **arb** total order

∧ ∀ read. ``**return value** = value of the last visible write in arbitration order"

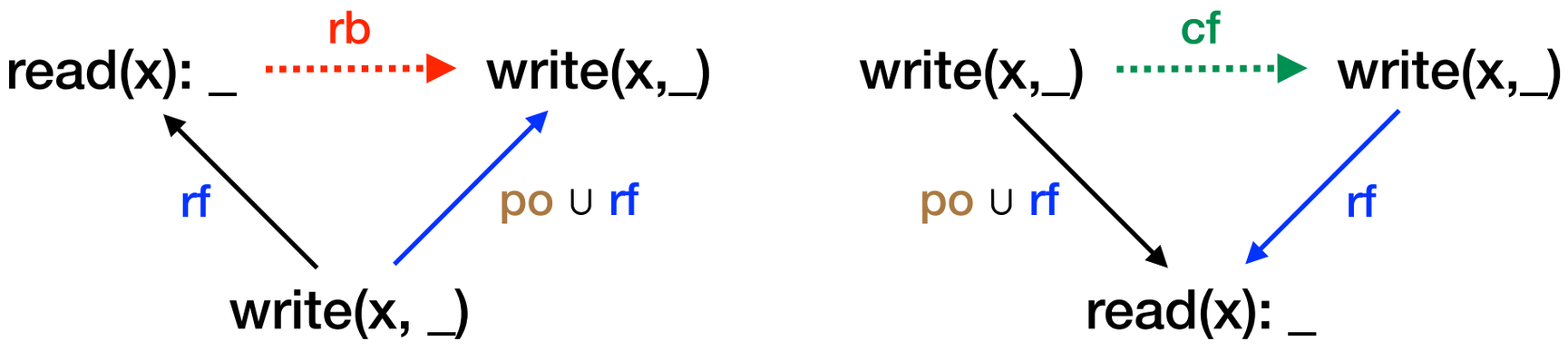
is equivalent to

∃ **rf**. **po** ∪ **rf** ∪ **rb** is acyclic **and** **po** ∪ **rf** ∪ **cf** is acyclic

rf (read-from): relating each read to a write with the same variable/value

rb (read-before): relating each read to a write that overwrites the read value

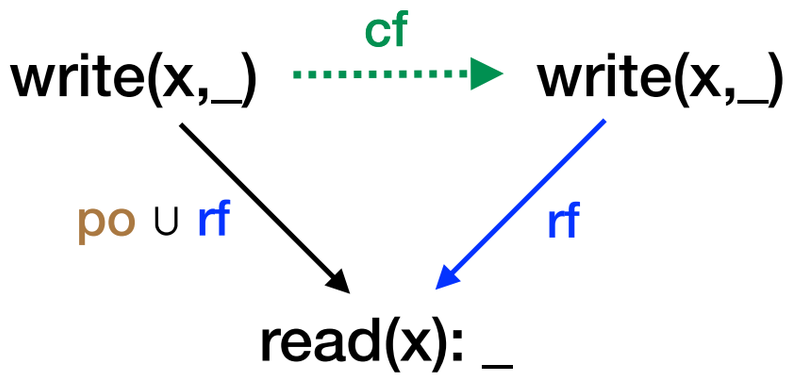
cf (conflict): a necessary subset of **arb** derived from **po** and **rf**



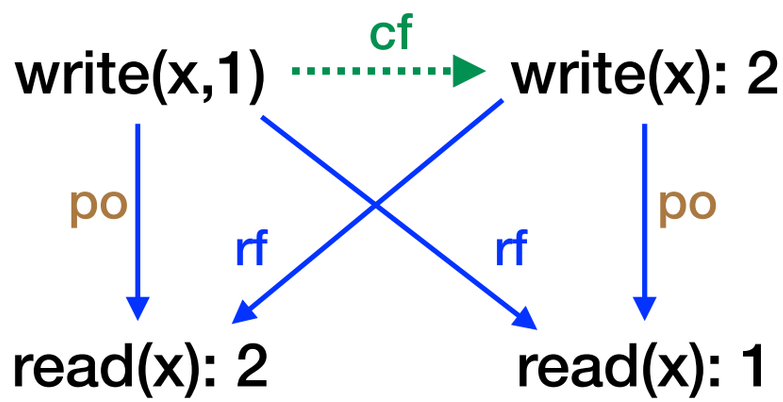
Characterizing CC

[POPL'17]

CC ≡ ∃ **rf**. **po** ∪ **rf** ∪ **rb** is acyclic **and**
po ∪ **rf** ∪ **cf** is acyclic



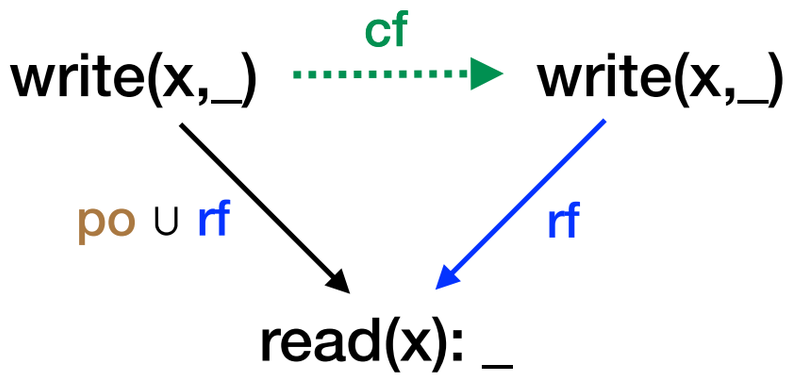
Example:



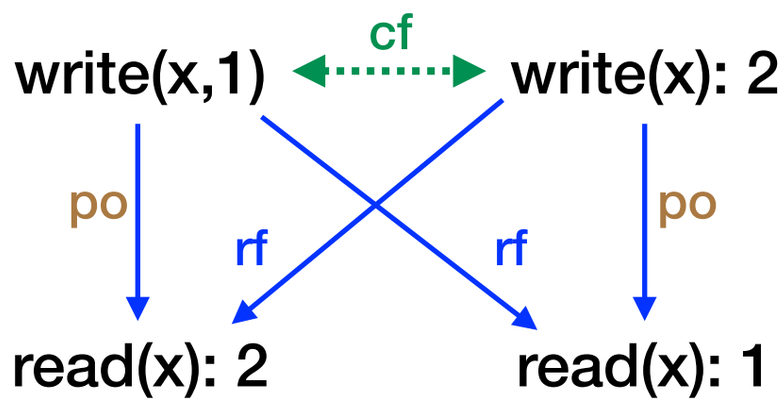
Characterizing CC

[POPL'17]

CC ≡ ∃ **rf**. **po** ∪ **rf** ∪ **rb** is acyclic **and**
po ∪ **rf** ∪ **cf** is acyclic



Example:



Checking CC

[POPL'17]

CC ≡ ∃ **rf**. **po** ∪ **rf** ∪ **rb** is acyclic **and**
po ∪ **rf** ∪ **cf** is acyclic

Each value is written once (data independence) => **fixed read-from**

Testing: acyclicity can be checked in polynomial time

Verification: using finite-state automata to represent "cyclic" executions