Replicated Data Types

Tutorial, ATVA 2025

Kartik Nagar
nagark@cse.iitm.ac.in
IIT Madras, India

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

SOSP 2007

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com

is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

- Solution: Replication and Object versioning.
- Synchronous replica co-ordination required for strong consistency
  - Providing the illusion of a centralized data store.
- Does not work under failure scenarios ("Network Partitions").

CAP Theorem:

Strong Consistency, Availability and Tolerance against Network Partitions cannot be simultaneously achieved.

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

SOSP 2007

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com

is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

- Optimistic Replication
- Every update at a replica generates a new version.
  - Writes are instantaneous and need no synchronization.
- Reads also return instantaneously.
  - May not return the most recently updated value.
- Replicas periodically merge their versions.

CAP Theorem:

Strong Consistency, Availability and Tolerance against Network Partitions cannot be simultaneously achieved.

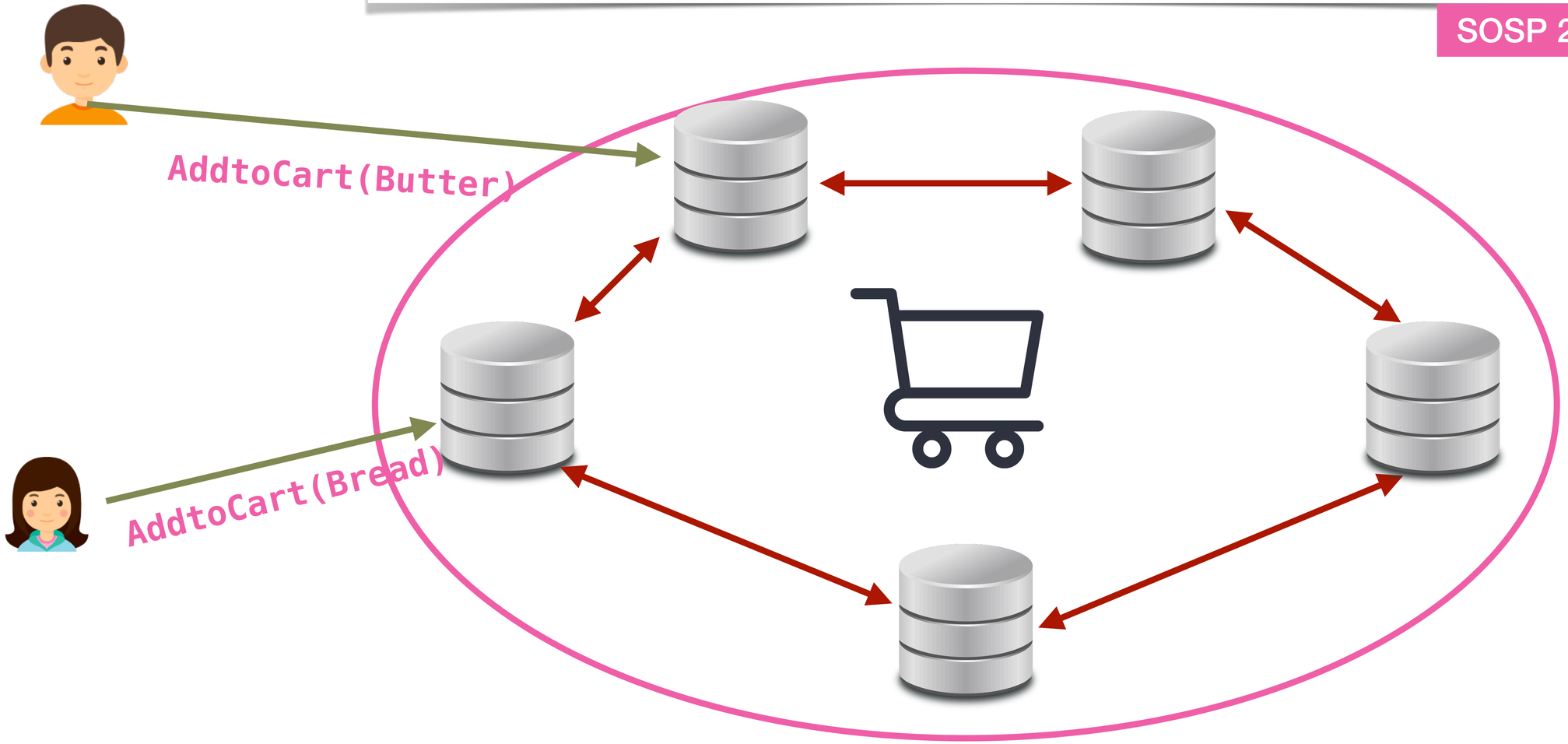
Eventual Consistency

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

SOSP 2007

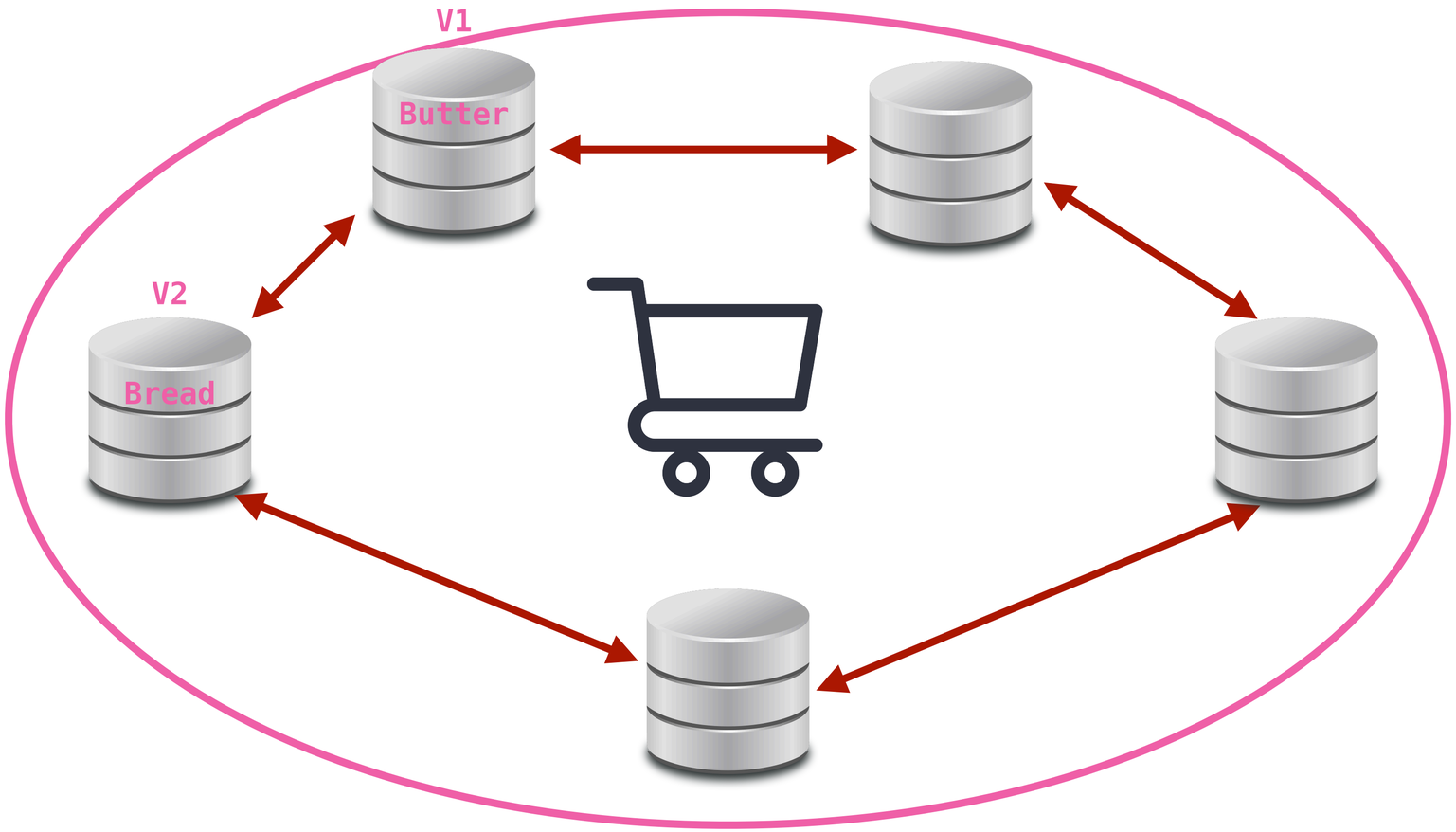


Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

SOSP 2007



Merging V1 and V2

- Classical conflict resolution strategy: Last writer wins
- Amazon's requirement: "Add to cart" operation can never be forgotten or rejected.
- Need for an customized merge operation which can cater to application requirements.
- Genesis for the "Add-wins" Set replicated data type.

Fast forward to today…

Replicated Data Types have been widely adopted by the industry, studied in academia and have found use-case in a number of collaborative applications

Academia

- New RDT Paradigms
  - State-based, Op-based, Delta-state
- Space and time-efficient RDT designs
- New datatypes
  - JSON, Rich textual format, Augmented Reality
- Automated Verification and Analysis

Industry

riak

aws

DynamoDB

Redis

Applications

Airtable

Google Docs

Figma

Notion

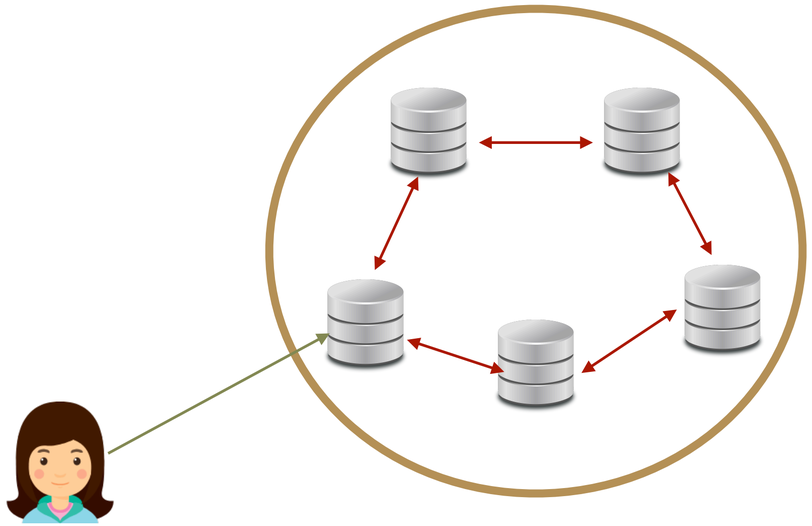
Overleaf

Part A: RDT Basics

RDT Basics

- Abstract Data Type interface for clients
- A set of operations for each data type
  - Register: {wr(v), rd}
  - Counter: {inc, dec, rd}
  - Set: {add(e), rem(e), lookup(e)}
  - List: {addAfter(a,b), remove(a), rd}
- Each operation is either a query or an update.
- The state is maintained at n different replicas or nodes.

RDT Basics



- Each replica is initialized to the same state.
- Client issues an operation at any replica, which is immediately executed at that replica.
- Periodically, a replica broadcasts a message to every other replica.
  - The contents of the message and the period depends on the RDT paradigm.
- When a replica receives a message, it updates its state.
- Different RDT paradigms also have different requirements on the network behavior, e.g. whether the messages can get lost, duplicated, re-ordered, etc.

Happens-before Order

- A replicated system is a distributed system where replicas send messages to each other to communicate updates.
- To define the concurrency semantics, two relations play a crucial role.
- An event e_1 happens-before e_2 (denoted by $e_1 \prec e_2$) if one of the following conditions hold:
  - e_1 occurs before e_2 on the same replica
  - e_1 sends a message m, and e_2 is the receive event corresponding to m
  - There exists another event e such that $e_1 \prec e$ and $e \prec e_2$.
- An update event u_1 happens-before u_2 if u_1 has already been applied on the replica where u_2 is applied.

Arbitration Order

- Arbitration order is a total order among update events.
- Should subsume happens-before order between update events.
- Useful to define last-writer-wins semantics.
- Can be implemented using Lamport time-stamps.

Formalism: Client Interaction

- We fix a data-type $\mathscr{D}$. Let Σ be the set of states maintained by $\mathscr{D}$.
  - Let σ_{init} be the initial state.
- Let O be the set of operations supported by $\mathscr{D}$.
  - O can be partitioned into O_u and O_q containing update and query operations respectively.
- $\mathscr{D}$ defines a function $do : \Sigma \times O_u \times \text{Timestamp} \rightarrow \Sigma$
- When a client issues an update operation o at a replica r with state σ:
  - $do(\sigma, o, t)$ is installed at r.
  - timestamp t is supplied by the replica.
- $\mathscr{D}$ also defines $ret : \Sigma \times O_q \rightarrow V$, which is called when a client issues a query operation.

Example: Increment-only Counter

- $\Sigma = \mathbb{N}$
- $\sigma_{init} = 0$
- $O = \{inc, rd\}$
  - $O_u = \{inc\}$
  - $O_q = \{rd\}$
- $do(\sigma, inc, t) = \sigma + 1$
- $ret(\sigma, rd) = \sigma$

Formalism: Messaging between replicas

- Let M be the set of messages used by $\mathscr{D}$
- $\mathscr{D}$ defines a function $send : \Sigma \to M$
  - Periodically, a replica with state σ will broadcast $send(\sigma)$ to all other replicas.
- $\mathscr{D}$ defines a function $receive : \Sigma \times M \to \Sigma$
  - When a replica r with state σ receives message m, the state $receive(\sigma, m)$ will be installed at r.

Two major paradigms in RDT Design

- The paradigms differ in what and when messages are sent.
- State-based CRDTs
  - The entire state is sent as a message.
  - Formally, $M = \Sigma$, $send(\sigma) = \sigma$
  - Messages can be sent at any time.
  - $receive : \Sigma \times \Sigma \to \Sigma$, also known as $merge$.
- Op-based CRDTs
  - A function (also known as an effector) is sent as a message.
  - Formally, $M = \Sigma \to \Sigma$, typically $send(do(\sigma, o, t)) = \lambda\sigma' \,.\, do(\sigma', o, t)$
  - A message is sent after every update operation is performed.
  - $receive(\sigma, F) = F(\sigma)$

Network Assumptions

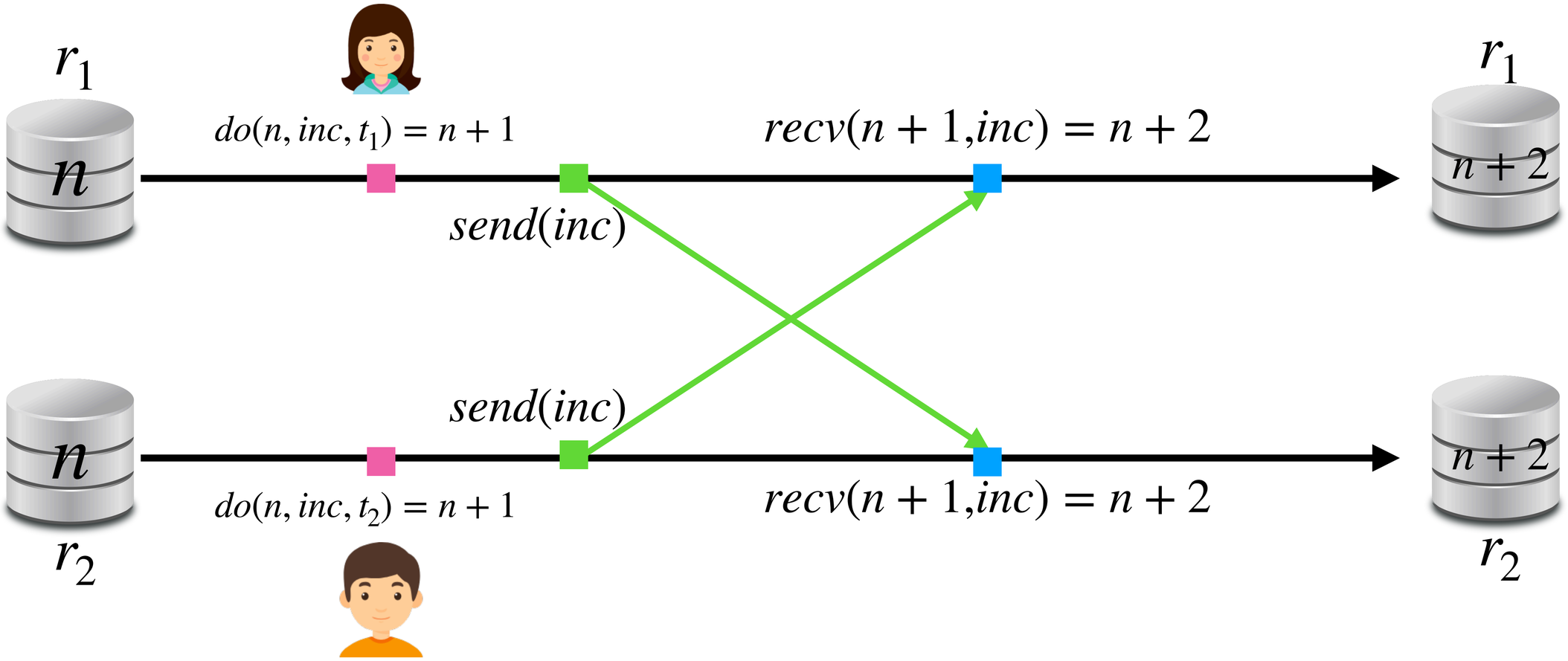
- State-based CRDTs require minimal constraints on the underlying network.
  - ✓ Messages can be lost, duplicated, or re-ordered.
  - They cannot be forged, i.e. any message received must have been sent by some replica.
- Op-based CRDTs require more stringent constraints on the network.
  - ✓ Messages can be lost.
  - Messages cannot be duplicated.
  - Messages cannot be forged.
  - Re-ordering is allowed to a limited extent: more on this later.

Op-based Increment-only Counter

- $\Sigma = \mathbb{N}$
- $O = \{inc, rd\}$
- $do(\sigma, inc, t) = \sigma + 1$
- $ret(\sigma, rd) = \sigma$
- $send(do(\sigma, o, t)) = \lambda\sigma' \,.\, do(\sigma', o, t)$
- $receive(\sigma, F) = F(\sigma)$

Ref: A comprehensive study of CRDTs, Shapiro et al.

An execution of the Op-based Counter



✳Commutativity of increment and decrement simplifies the design.

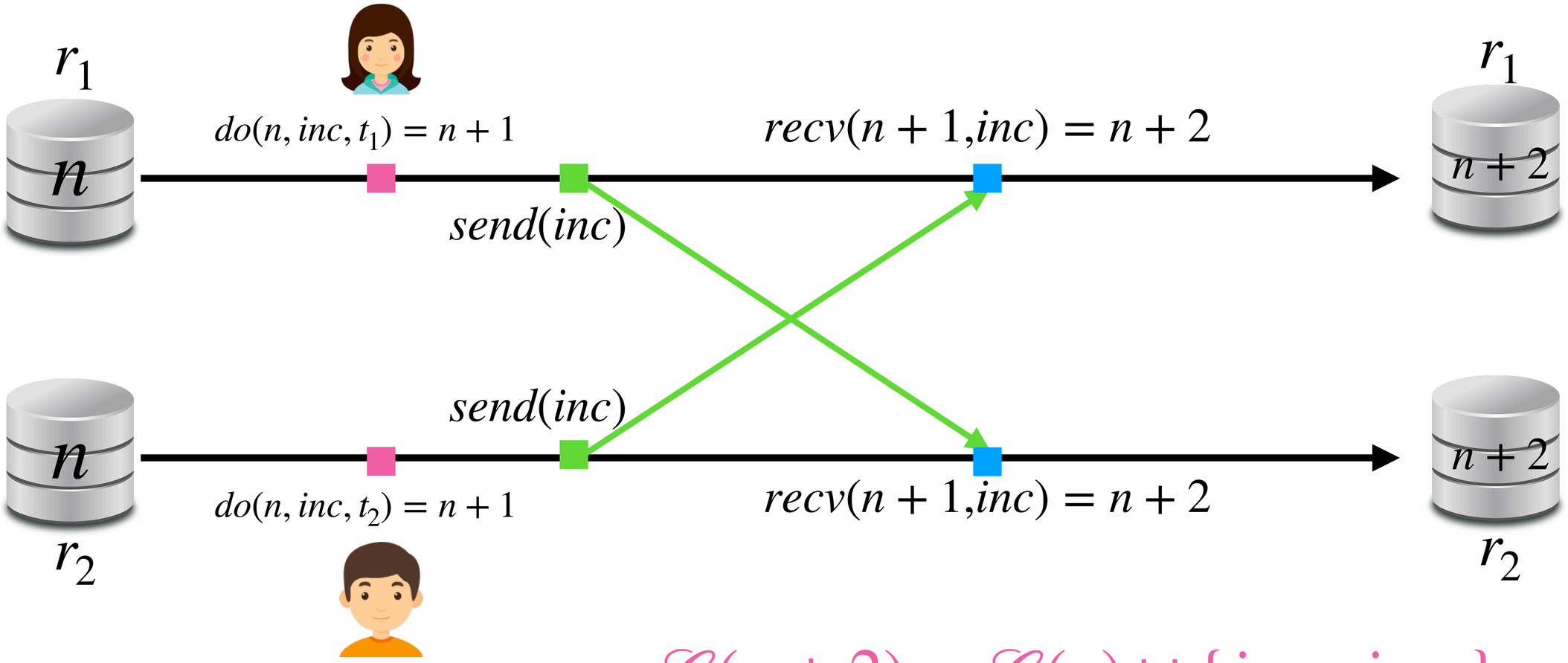
✳Note that “no duplication of messages” network assumption is critical.

Strong Eventual Consistency

- We associate an abstract state $\mathscr{C}(\sigma)$ with each state σ which collects the update operations applied directly or indirectly to get σ.
- For the initial RDT state σ_{init}, $\mathscr{C}(\sigma_{\text{init}}) = \emptyset$.
- On performing $do(\sigma, o, t)$, $\mathscr{C}(do(\sigma, o, t)) = \mathscr{C} \cup \{(o, t)\}$
- For state-based RDTs:
  - On performing $receive(\sigma_r, \sigma_m)$, $\mathscr{C}(receive(\sigma_r, \sigma_m)) = \mathscr{C}(\sigma_r) \cup \mathscr{C}(\sigma_m)$
- For op-based RDTs:
  - On performing $receive(\sigma_r, F)$, if (o, t) was the generator of F, then $\mathscr{C}(receive(\sigma_r, (o, t))) = \mathscr{C}(\sigma_r) \cup \{(o, t)\}$

A RDT $\mathscr{D}$ is strong eventually consistent if for any two states σ_1 and σ_2 present at any two replicas, $\mathscr{C}(\sigma_1) = \mathscr{C}(\sigma_2) \implies \sigma_1 = \sigma_2$

An execution of the Op-based Counter



$$\mathscr{C}(n+2) = \mathscr{C}(n) \cup \{inc_1, inc_2\}$$

State-based increment-only Counter

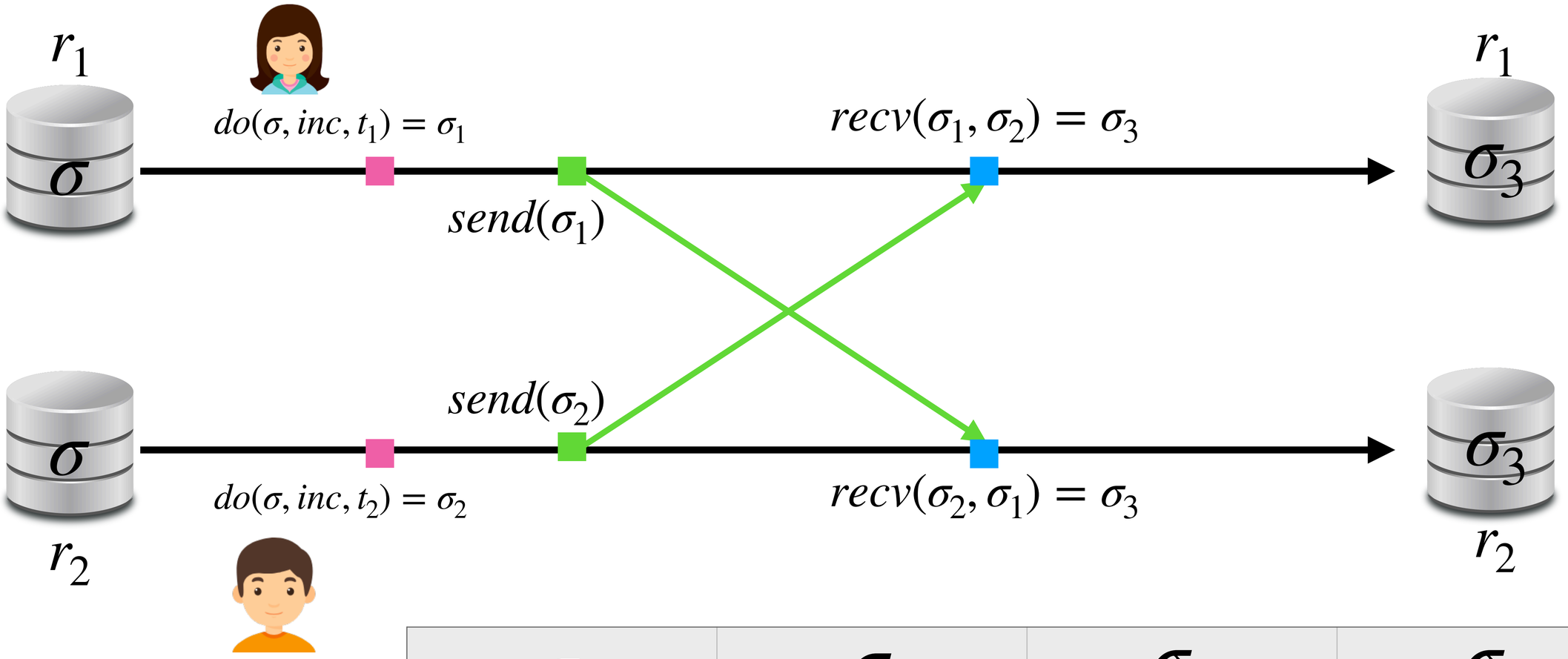
- $\Sigma = \mathscr{R} \rightarrow \mathbb{N}$ (assume $\mathscr{R}$ is the set of replicas)
- $O = \{inc, rd\}$
- At replica r, $do(\sigma, inc, t) = \sigma[r \mapsto \sigma(r) + 1]$
- $ret(\sigma, rd) = \Sigma_{r \in \mathbb{R}} \sigma(r)$
- $send(\sigma) = \sigma$
- $receive(\sigma_r, \sigma_m) = \lambda r\ .\ max(\sigma_r(r), \sigma_m(r))$

✳Extra meta-data as compared to Op-based counter

✳Works under more relaxed network assumptions: duplication, arbitrary re-ordering of messages

Ref: A comprehensive study of CRDTs, Shapiro et al.

An execution of the state-based Counter



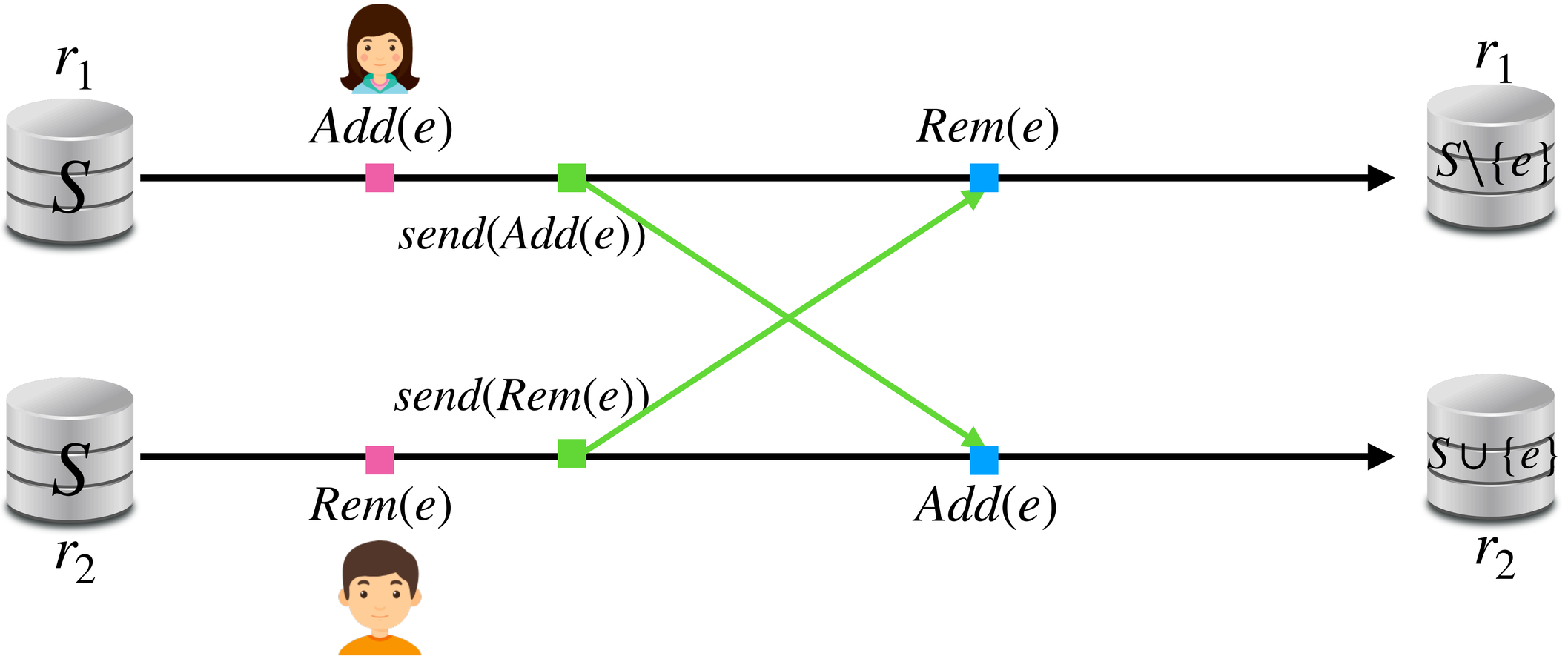
σ	σ_1	σ_2	σ_3
$r_1 \mapsto n$ $r_2 \mapsto n$	$r_1 \mapsto n+1$ $r_2 \mapsto n$	$r_1 \mapsto n$ $r_2 \mapsto n+1$	$r_1 \mapsto n+1$ $r_2 \mapsto n+1$

State-based LWW Register

- $\Sigma = V \times T$ (assume V is the value set)
- $O = \{set(v) \mid v \in V\} \cup \{get\}$
- $do((v', t'), set(v), t) = \begin{cases} (v, t) & \text{if } t > t' \\ (v', t') & \text{otherwise} \end{cases}$
- $ret((v, t), get) = v$
- $send(\sigma) = \sigma$
- $receive((v_r, t_r), (v_m, t_m)) = \begin{cases} (v_r, t_r) & \text{if } t_r > t_m \\ (v_m, t_m) & \text{otherwise} \end{cases}$

Ref: A comprehensive study of CRDTs, Shapiro et al.

Designing an Op-based Set RDT

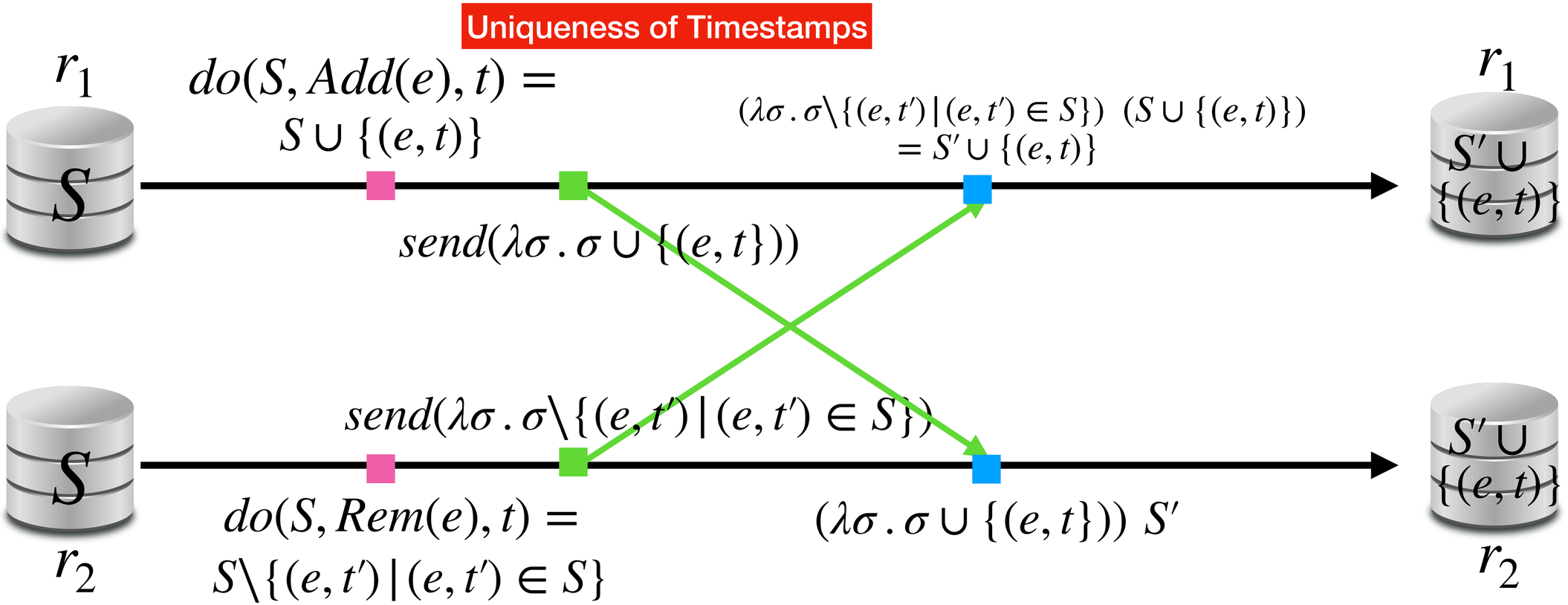


$do(\sigma, Add(e), t) = \sigma \cup \{e\}$

$do(\sigma, Rem(e), t) = \sigma \backslash \{e\}$

Violation of Strong Eventual Consistency

Designing an Op-based Set RDT



Add-wins Set

amazon 😄

$S' = S\backslash\{(e,t') \mid (e,t') \in S\}$

Op-based Add-wins Set

- $\Sigma = \mathbb{P}(V \times T)$
- $O = \{Add(v) \mid v \in V\} \cup \{Rem(v) \mid v \in V\} \cup \{lookup(v) \mid v \in V\}$
- $do(\sigma, Add(v), t) = \sigma \cup \{(v, t)\}$
- $do(\sigma, Rem(v), t) = \sigma \backslash \{(v, t') \mid (v, t') \in \sigma\}$
- $ret(\sigma, lookup(v)) = \exists t\,.\ (v, t) \in \sigma$
- $send(do(\sigma, Add(v), t)) = \lambda\sigma'\,.\,\sigma' \cup \{(v, t)\}$
- $send(do(\sigma, Rem(v), t)) = \lambda\sigma'\,.\,\sigma' \backslash \{(e, t') \mid (e, t') \in \sigma\}$
- $receive(\sigma, F) = F(\sigma)$

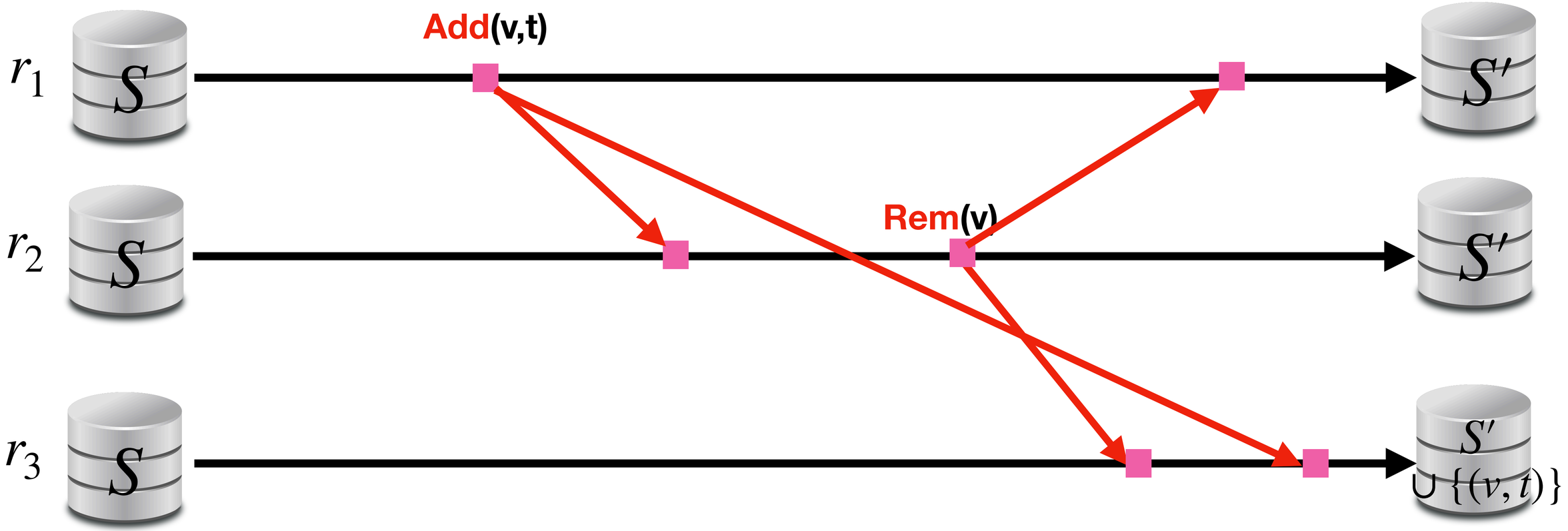
✳Note that the Rem effector is slightly different from the do function of Rem.

✳However, the do function is an application of the effector on the source replica

Ref: A comprehensive study of CRDTs, Shapiro et al.

Does this always work?
Unfortunately not!
Needs stronger network assumptions

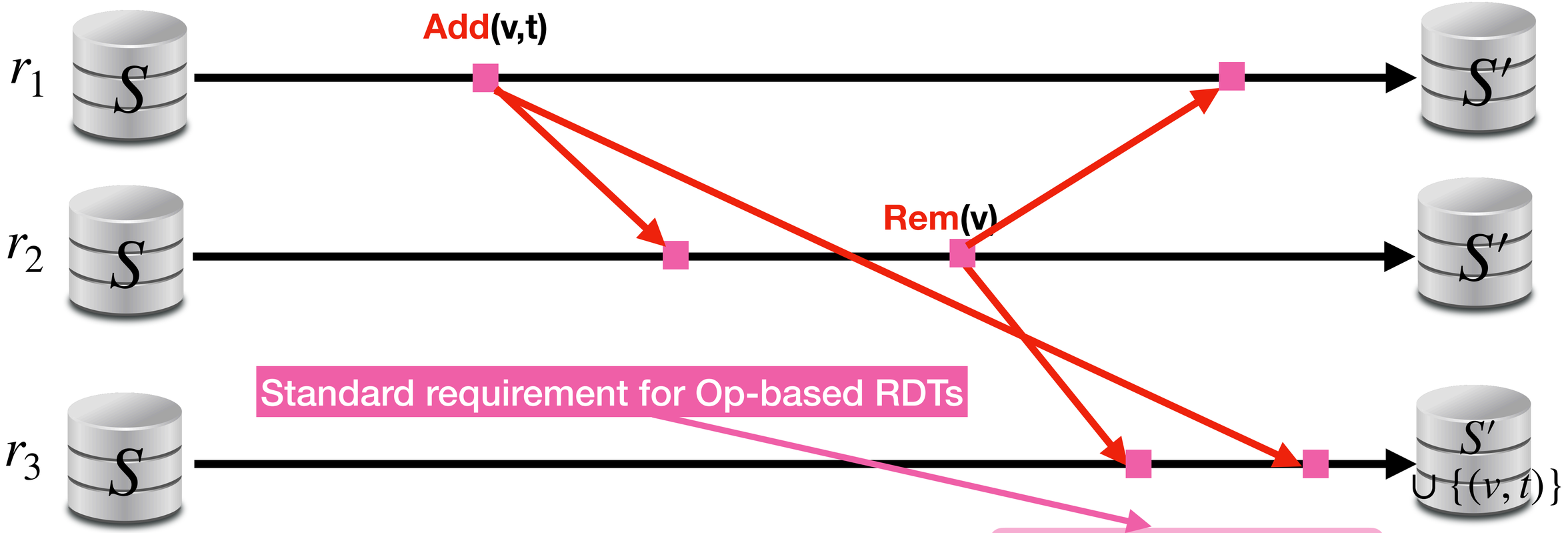
A diverging execution of the Add-wins Set



Violation of Strong Eventual Consistency

$$S' = S \setminus \{(v,t') \mid (v,t') \in S\}$$

A diverging execution of the Add-wins Set

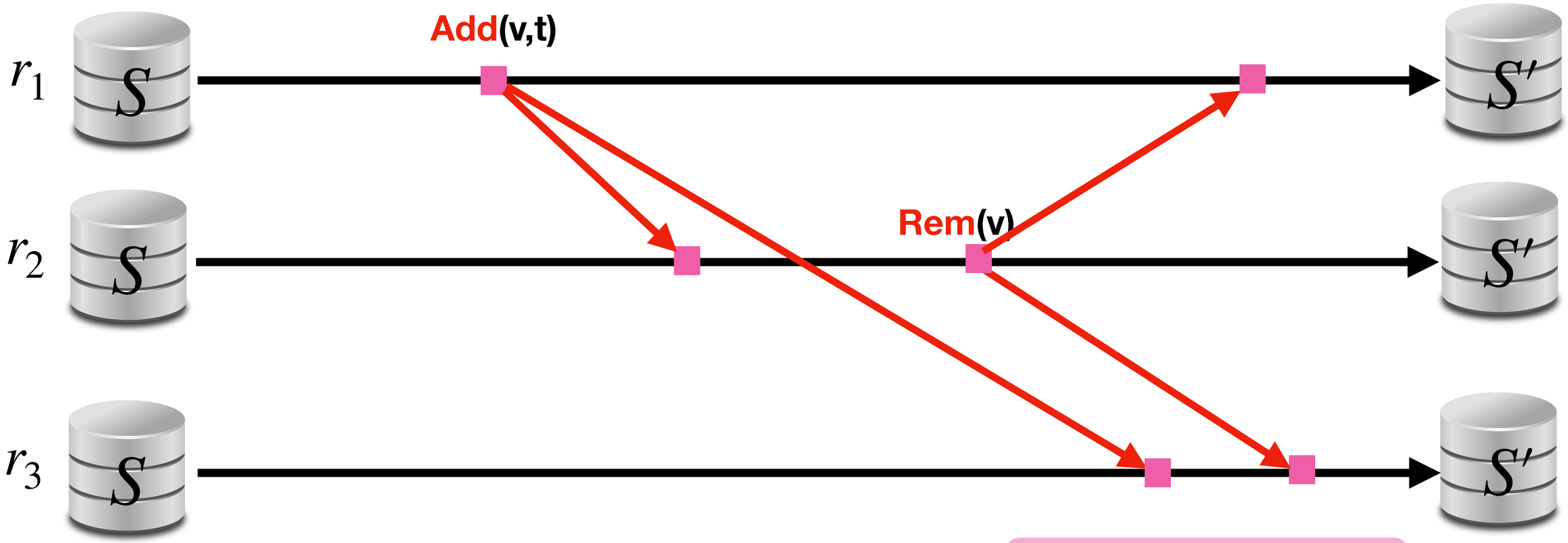


Need a stronger network guarantee: Causal consistency

If $send(m_1)$ happens-before $send(m_2)$, then m_1 must be received before m_2 at each replica

$$S' = S \setminus \{(v,t') \mid (v,t') \in S\}$$

A execution of the Add-wins Set under Causal Consistency



Need a stronger network guarantee: Causal consistency

If $send(m_1)$ happens-before $send(m_2)$, then m_1 must be received before m_2 at each replica

$$S' = S \setminus \{(v, t') \mid (v, t') \in S\}$$

Other RDTs

- Multi-value register
- State-based PN Counter
- Remove-wins set
- List (Replicated Growable Array)
- Update-wins/Remove-wins Map
- Graphs
- JSON (Automerge)
- …

Other RDT Paradigms: MRDT

- Mergeable Replicated Data Types (MRDT)
  - A variant of state-based RDTs
  - Uses a 3-way merge function to update the state on receipt of a message.
  - $merge(\sigma_{lca}, \sigma_1, \sigma_2)$: σ_1 is the state at the receiving replica, σ_2 is the state of the sending replica, and σ_{lca} is the lowest common ancestor state of σ_1 and σ_2.
  - Can lead to very efficient RDT implementations.
  - Inspired by distributed version control systems such as Git.

Increment-only Counter MRDT

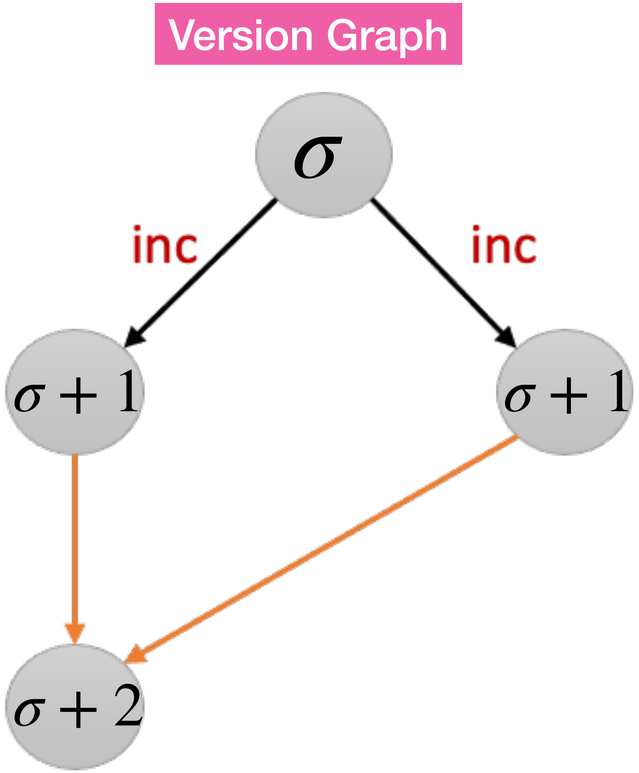
- $\Sigma = \mathbb{N}$
- $O = \{inc, rd\}$
- At replica r, $do(\sigma, inc, t) = \sigma + 1$
- $ret(\sigma, rd) = \sigma$
- $send(\sigma) = \sigma$
- $merge(\sigma_l, \sigma_r, \sigma_m) = \sigma_l + (\sigma_r - \sigma_l) + (\sigma_m - \sigma_l)$

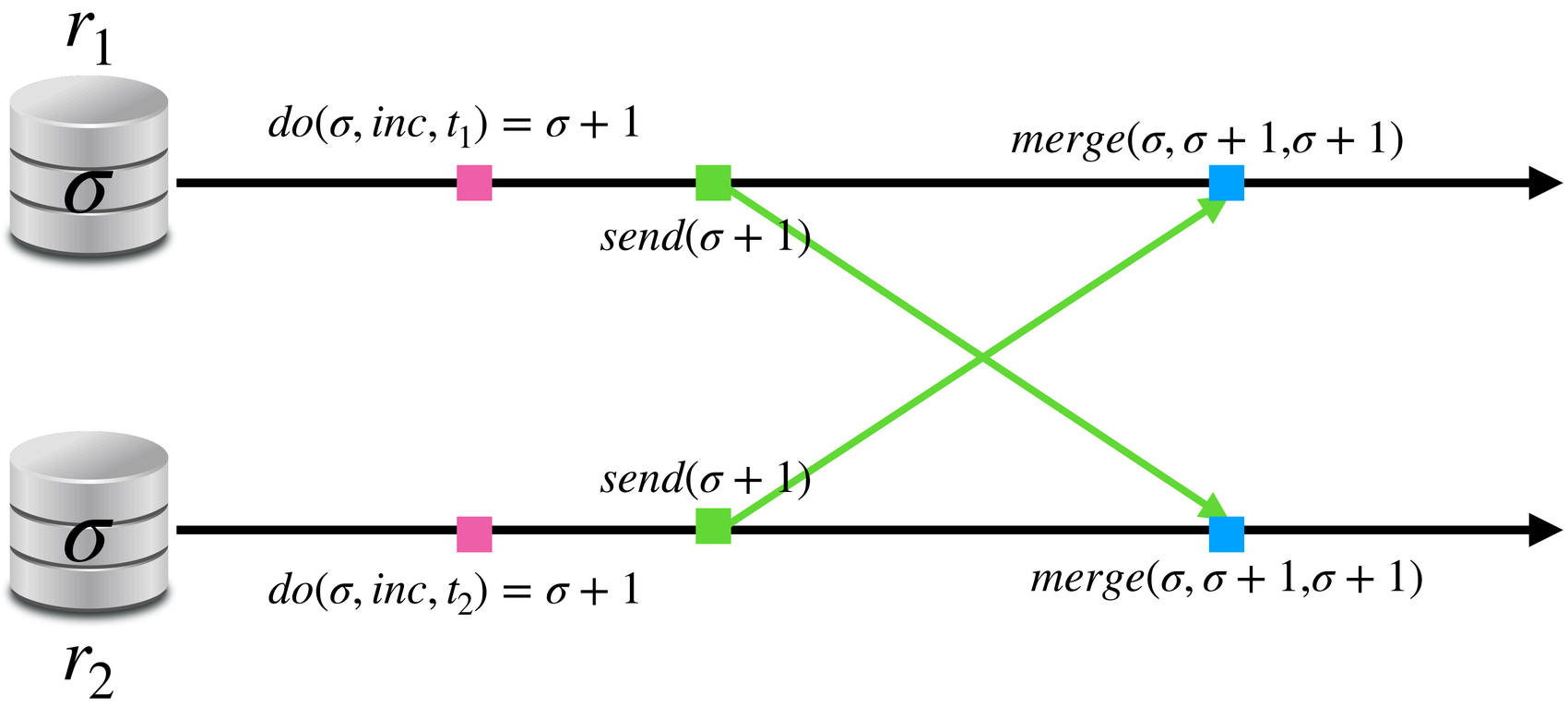
Best of both worlds

✳No extra meta-data (space and time complexity same as the Op-based Counter)

✳Works under more relaxed network assumptions: duplication, arbitrary re-ordering of messages

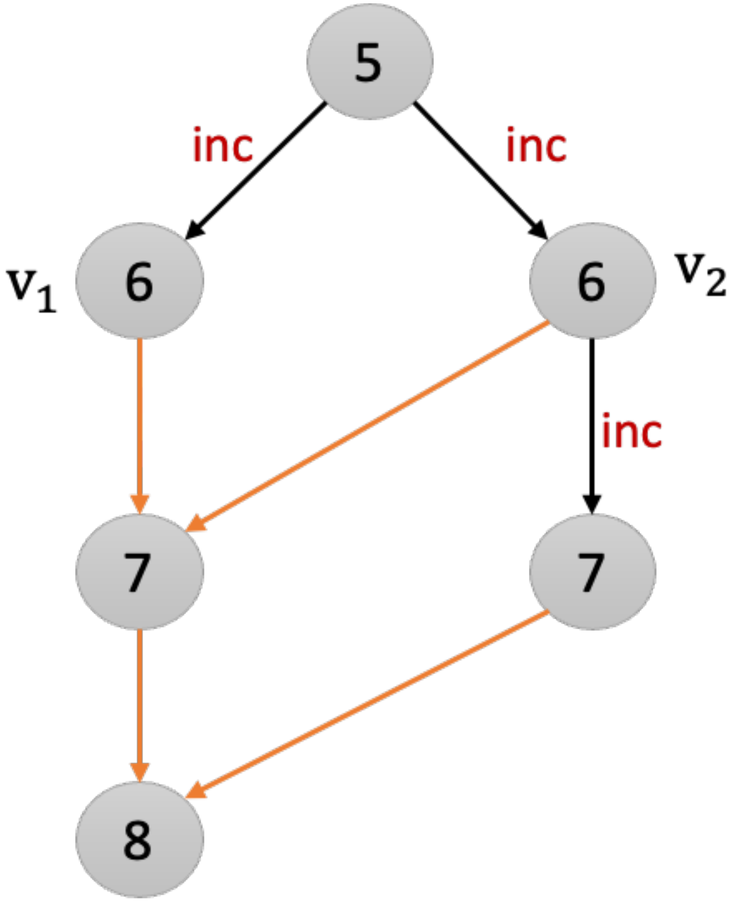
An execution of the Counter MRDT



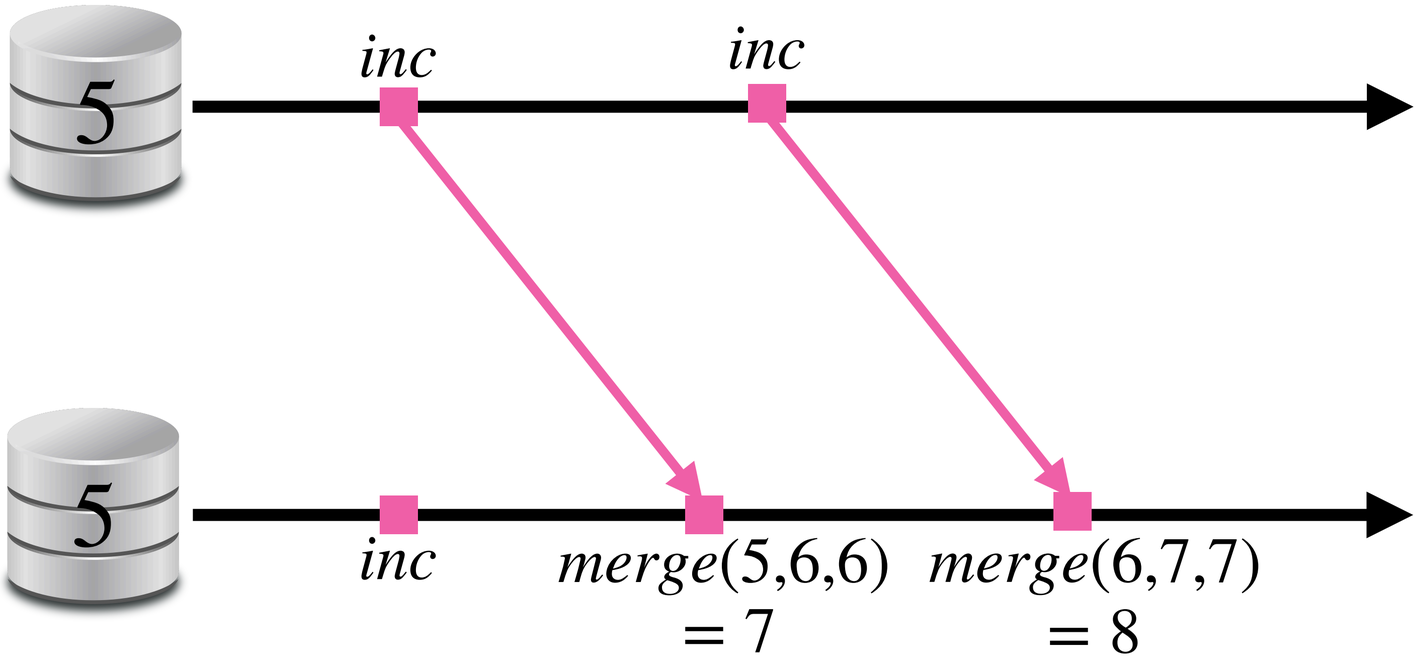


$$merge(\sigma, \sigma + 1, \sigma + 1) = \sigma + (\sigma + 1 - \sigma) + (\sigma + 1 - \sigma) = \sigma + 2$$

Another execution of the Counter MRDT

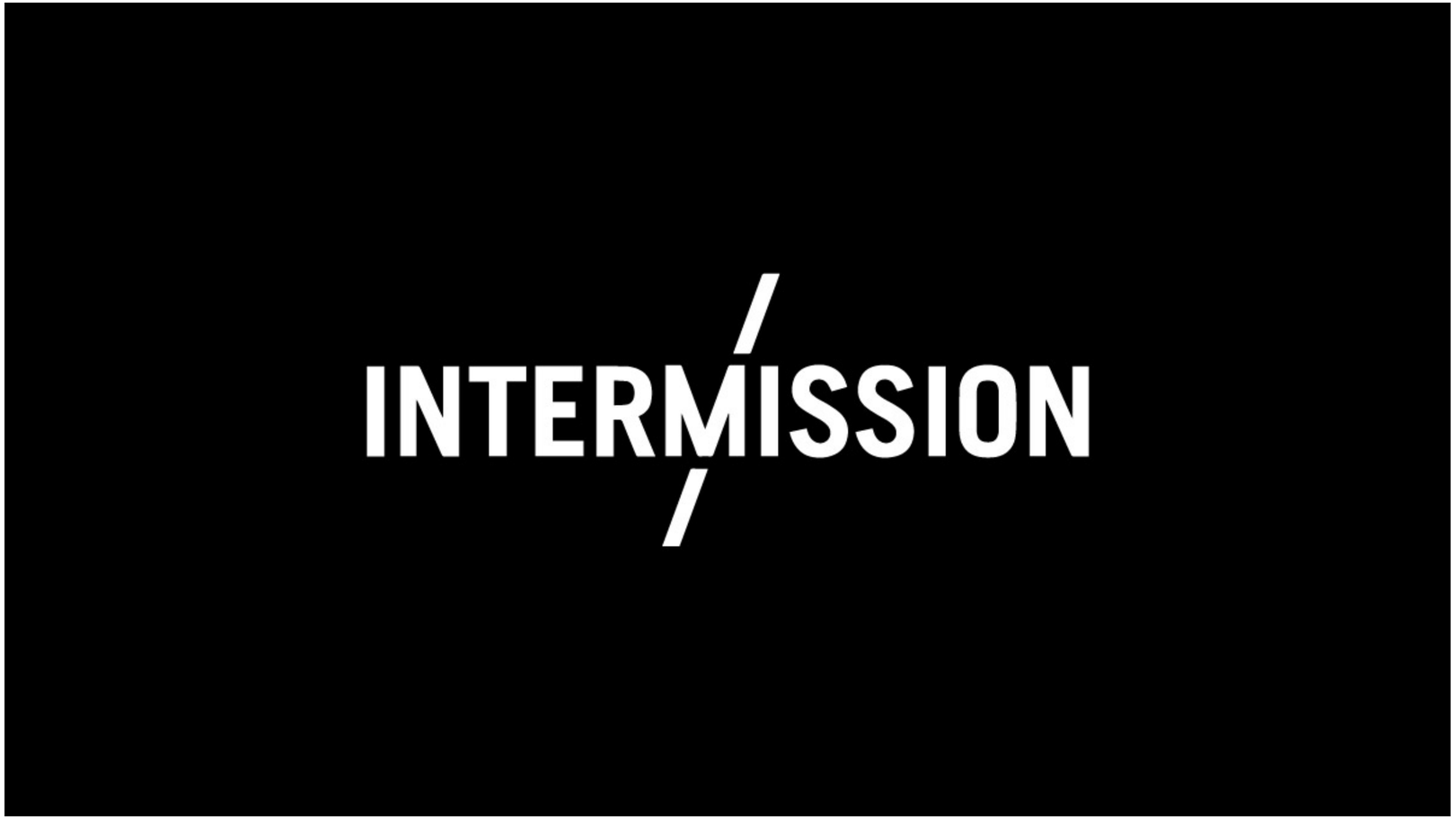


$$\text{merge}\,(\text{lca}, v_1, v_2) = \text{lca} + (v_1 - \text{lca}) + (v_2 - \text{lca})$$



Other RDT Paradigms: Delta-state CRDTs

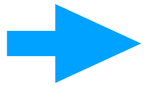
- Sending the entire state as a message could get expensive and may be unnecessary.
  - E.g. in state-based Counter CRDT, an increment operation only updates a small part of the overall state.
- Delta-state CRDTs send delta-mutators as messages, which encodes all the changes that have been made at a replica since its last communication.



After the break, Specification and Verification of RDTs…

Part B: Specification and Verification of RDTs

Different forms of Specifications



- Strong Eventual Consistency (also known as convergence)
- Declarative RDT Specification
- Replication-aware Linearizability

Recall: Strong Eventual Consistency

- We associate an abstract state $\mathscr{C}(\sigma)$ with each state σ which collects the update operations applied directly or indirectly to get σ.
- For the initial RDT state σ_{init}, $\mathscr{C}(\sigma_{\text{init}}) = \emptyset$.
- On performing $do(\sigma, o, t)$, $\mathscr{C}(do(\sigma, o, t)) = \mathscr{C} \cup \{(o, t)\}$
- For state-based RDTs:
  - On performing $receive(\sigma_r, \sigma_m)$, $\mathscr{C}(receive(\sigma_r, \sigma_m)) = \mathscr{C}(\sigma_r) \cup \mathscr{C}(\sigma_m)$
- For op-based RDTs:
  - On performing $receive(\sigma_r, F)$, if (o, t) was the generator of F, then $\mathscr{C}(receive(\sigma_r, (o, t))) = \mathscr{C}(\sigma_r) \cup \{(o, t)\}$

A RDT $\mathscr{D}$ is strong eventually consistent if for any two states σ_1 and σ_2 present at any two replicas, $\mathscr{C}(\sigma_1) = \mathscr{C}(\sigma_2) \implies \sigma_1 = \sigma_2$

Strong Eventual Consistency for state-based CRDTs

- Concepts from Lattice Theory can be used to verify convergence of state-based CRDTs.
- The first thing we need is for Σ to be a join semi-lattice.
- Let D be a set and $\leq \subseteq D \times D$ be a partial order on D.
- For any $x, y \in D$, $x \sqcup y$ is the least upper bound of x and y, i.e.
  - $x \leq x \sqcup y, y \leq x \sqcup y$ and
  - For any $z \in D$ such that $x \leq z, y \leq z, x \sqcup y \leq z$.
- For any $x, y \in D$, if $x \sqcup y$ exists, it must be unique.
- $(D, \leq, \sqcup)$ is a join semi-lattice if $x \sqcup y$ exists for all elements $x, y \in D$.

Strong Eventual Consistency for state-based CRDTs

- The first thing we need is for Σ to be a join semi-lattice.
- $merge : \Sigma \times \Sigma \rightarrow \Sigma$ will be the lattice join function ($\sqcup$).
- The $\leq$ ordering can be induced from the join function:
  - $\sigma \leq \sigma' \Leftrightarrow merge(\sigma, \sigma') = \sigma'$
- $\leq$ will be a partial order only if
  - $merge$ is idempotent, commutative and associative.

$(\Sigma, \leq, merge)$ is a join semi-lattice if $merge$ is idempotent, commutative and associative.

Strong Eventual Consistency for state-based CRDTs

- A function $f : D \to D$ is monotonic if $\forall x, y \in D, x \leq y \implies f(x) \leq f(y)$.
- We say that all update functions of a CRDT $\mathscr{D}$ are monotonic, if for every update operation $o \in O_u$, for all timestamps t, $do(o, t) : \Sigma \to \Sigma$ is monotonic.
- A state-based CRDT $\mathscr{D}$ is convergent if
  - Σ is a join semi-lattice with $merge$ being the join function.
  - Every update operation of $\mathscr{D}$ is monotonic.
- Intuitively, the uniqueness of join guarantees convergence.

Ref: A comprehensive study of CRDTs, Shapiro et al.

State-based increment-only Counter

- $\Sigma = \mathbb{R} \to \mathbb{N}$ (assume $\mathbb{R}$ is the set of replicas)
- $O = \{inc, rd\}$
- At replica r, $do(\sigma, inc, t) = \sigma[r \mapsto \sigma(r) + 1]$
- $ret(\sigma, rd) = \Sigma_{r \in \mathbb{R}} \sigma(r)$
- $send(\sigma) = \sigma$
- $merge(\sigma_r, \sigma_m) = \lambda r \,.\, max(\sigma_r(r), \sigma_m(r))$

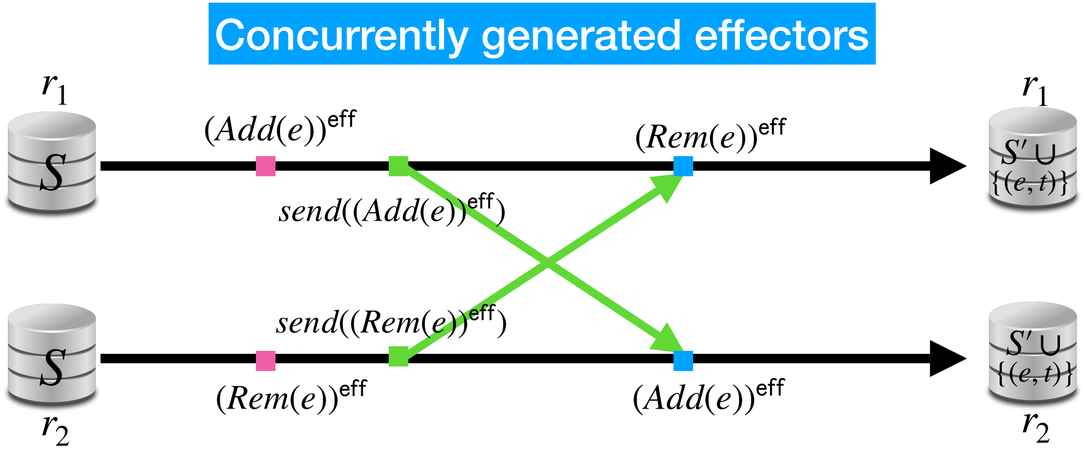
$merge$ is idempotent, commutative and associative

$$\sigma_1 \leq \sigma_2 \Leftrightarrow \forall r \in \mathbb{R} \,.\, \sigma_1(r) \leq \sigma_2(r)$$

$(\Sigma, \leq, merge)$ is a join semi-lattice

$do(inc, t)$ is a monotonic function

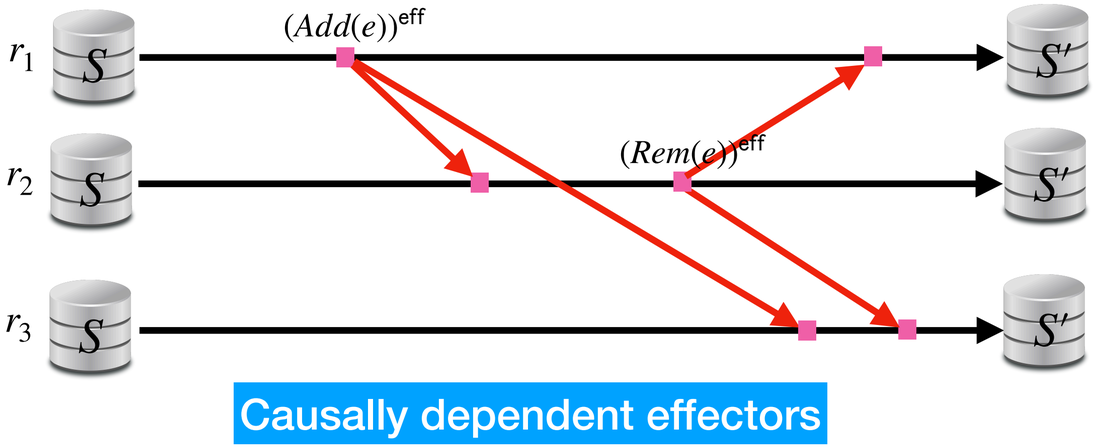
Strong Eventual Consistency for Op-based CRDTs



Effectors can be applied in either order, hence they must commute.

$$(Add(e))^{\text{eff}} \circ (Rem(e))^{\text{eff}} = (Rem(e))^{\text{eff}} \circ (Add(e))^{\text{eff}}$$

Not always though!



Causal consistency ensures that some effectors are always applied in the same order

Such effectors need not commute with each other

By o^{eff}, we mean applying the effector of o

Strong Eventual Consistency for Op-based CRDTs

- Commutativity of effectors which are concurrently generated is enough to ensure convergence of Op-based RDTs.
  - Commutativity modulo consistency policy
- The behavior of an effector depends on the state of the generating replica.
  - For the Op-based Add wins Set,

$$send(do(\sigma, Rem(v), t)) = \lambda\sigma' \,.\, \sigma' \setminus \{(e, t') \mid (e, t') \in \sigma\}$$

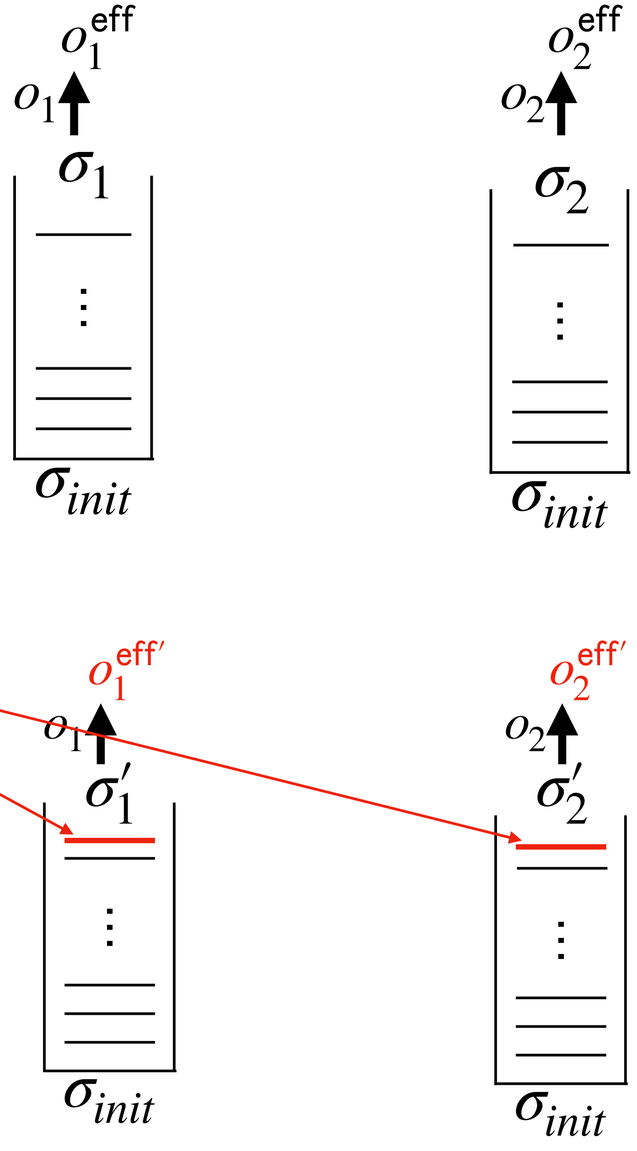
$$Rem(v)^{\text{eff}}$$

➡Effectively an infinite set of effectors

➡ We need to show commutativity for every pair of such concurrently generated effectors

Ref: Automated Parameterized Verification of CRDTs. Nagar and Jagannathan.

Verifying commutativity modulo consistency policy



Inductive Check:

If o_1^{eff} and o_2^{eff} commute, then $o_1^{eff'}$ and $o_2^{eff'}$ also commute.

Non-interference to commutativity

Fully Automated Approach

Both Base case and inductive case encoded using SMT

- The behaviour of an effector depends on the state of the generating replica.
- But the state of the generating replica itself is obtained by applying a sequence of effectors.
  - We can use induction on this sequence.

Quite effective in practice

 Ref: Automated Parameterized Verification of CRDTs, Nagar and Jagannathan.

Different forms of Specifications

- Strong Eventual Consistency (also known as convergence)
- Declarative RDT Specification
- Replication-aware Linearizability

Declarative Sequential Specifications

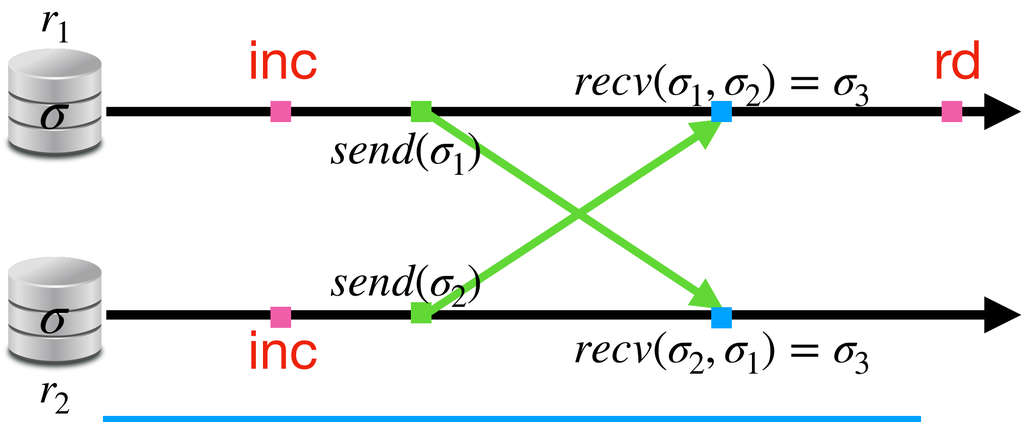
- Specification over sequences of update operations.
- Constrains the return value of a query operation occurring after a sequence of update operations.
  - Formally, $\mathcal{S}_{\mathscr{D}} : O_u^* \times O_q \to V$
- Examples:
  - For $\mathscr{D} =$ counter, $\mathcal{S}_{\mathscr{D}}(\pi, rd) = |\pi|$ where π is a sequence of inc operations.
  - For $\mathscr{D} =$ register, $\mathcal{S}_{\mathscr{D}}(\pi \cdot set(v), get) = v$
  - For $\mathscr{D} =$ set, $\mathcal{S}_{\mathscr{D}}(\pi, lookup(v)) = True \Leftrightarrow \pi$ contains an $add(v)$ operation not followed by a $rem(v)$ operation.

Declarative RDT Specification

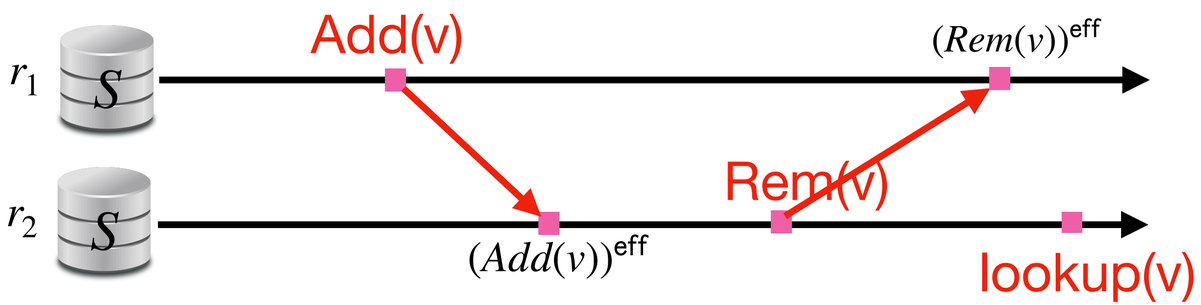
- Instead of using a sequence of update operations, RDT specifications are over an operation context $(E, oper, time, hb)$
  - E is a set of events
  - $oper : E \to O_u$
  - $time : E \to$ Timestamp
  - $hb \subseteq E \times E$ is an acyclic relation.
- RDT Specification $\mathscr{F}(\mathscr{D})$ is defined as a function which takes as input an operation context L and a query operation and returns a value.
  - $L.E$ is the abstract state $\mathscr{C}(r)$ of the replica r at which the query operation is performed.

Ref: Replicated Data Types: Specification, Verification and Optimality. Burckhardt et al.

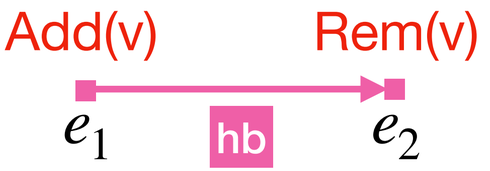
Operation Context: Examples



An execution of the state-based Counter



An execution of the op-based Set



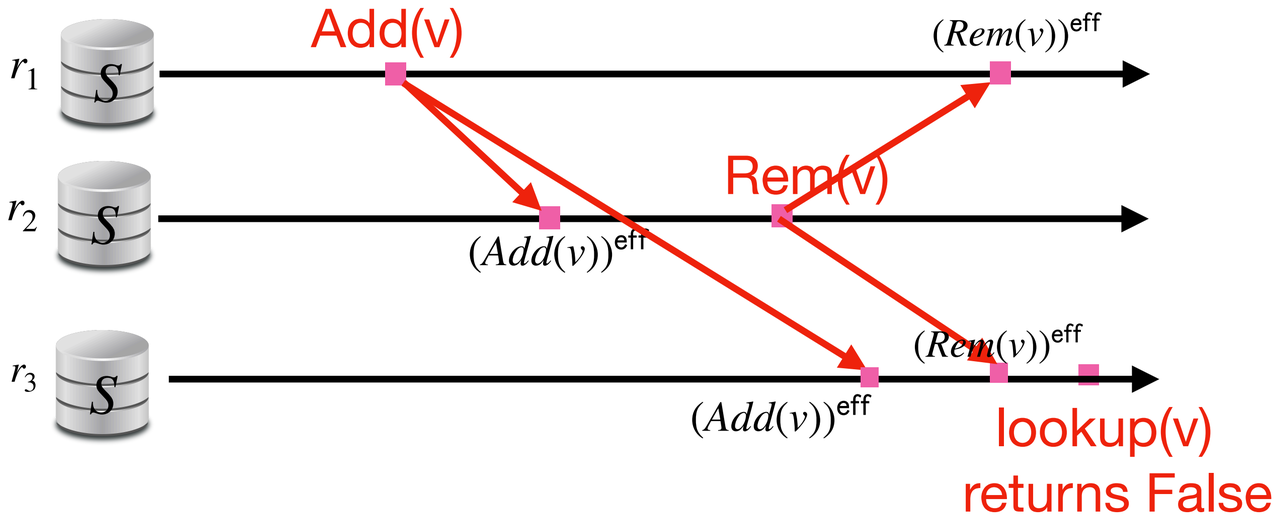
Declarative RDT Specification: Examples

- For $\mathscr{D}$ = counter, $\mathscr{F}_{\mathscr{D}}(L, rd) = |L.E|$.
- For $\mathscr{D}$ = LWW register, $\mathscr{F}_{\mathscr{D}}(L, get) = \mathscr{S}_{\mathscr{D}}(L.E^{\text{time}}, get)$ where $L.E^{\text{time}}$ is the sequence obtained by ordering operations in E in increasing order of their timestamps.
- For $\mathscr{D}$ = Add-wins set,

$$\mathscr{F}_{\mathscr{D}}(L, lookup(v)) = True \Leftrightarrow \exists e \in L.E.\ oper(e) = Add(v)$$
$$\wedge \neg(\exists f.\ oper(f) = Rem(v) \wedge L.hb(e, f))$$

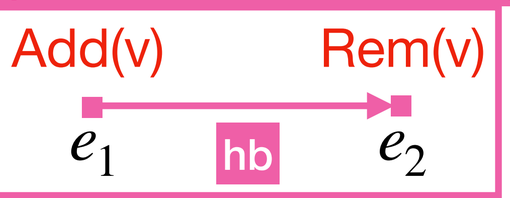
Correctness using Declarative RDT Specifications

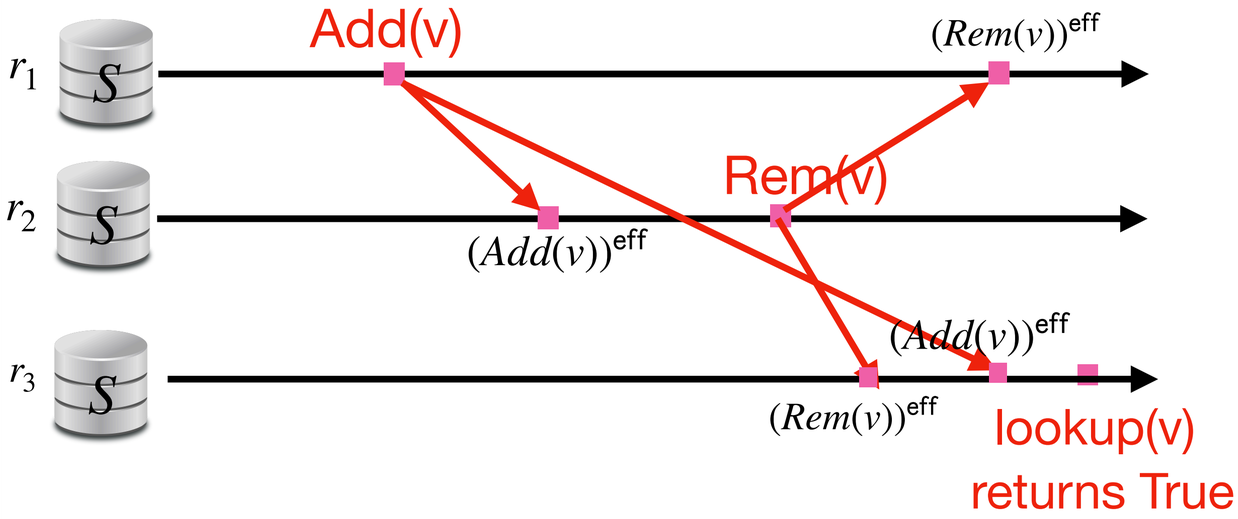
A RDT implementation $\mathscr{D}$ is correct if for every execution E and every query operation q in E, the return value of q matches $\mathscr{F}_{\mathscr{D}}(L, q)$ where L is the operation context of q



A correct execution of the op-based Add-wins Set

Operation context of lookup(v)





A incorrect execution of the op-based Add-wins Set

Actually prohibited by Causal Consistency

Verifying Declarative RDT Specifications for state-based RDTs

- A Replication-aware Simulation $\mathscr{R}_{\text{sim}}(L, \sigma)$ relation relates an operation context L with the concrete state σ.
- Verification using $\mathscr{R}_{\text{sim}}(L, \sigma)$ is carried in two steps:
  1. We show that $\mathscr{R}_{\text{sim}}(L, \sigma)$ holds inductively at all replicas r in any execution where L is the operation context at r and σ is the concrete state at r.
  2. We show that $\mathscr{R}_{\text{sim}}(L, \sigma)$ is sufficient to discharge the RDT specification.

Ref: Replicated Data Types: Specification, Verification and Optimality. Burckhardt et al.

Examples of $\mathscr{R}_{\text{sim}}$

- For the state-based Counter RDT:

$$\mathscr{R}_{\text{sim}}(L, \sigma) \Leftrightarrow |L\,.\,E| = \Sigma_{r \in \mathbb{R}} \sigma(r)$$

- $\Sigma = \mathbb{R} \rightarrow \mathbb{N}$ (assume $\mathbb{R}$ is the set of replicas)
- $O = \{inc, rd\}$
- At replica r,
  $do(\sigma, inc, t) = \sigma[r \mapsto \sigma(r) + 1]$
- $ret(\sigma, rd) = \Sigma_{r \in \mathbb{R}} \sigma(r)$
- $send(\sigma) = \sigma$
- $receive(\sigma_r, \sigma_m) = \lambda r\,.\; max(\sigma_r(r), \sigma_m(r))$

State-based Counter

Examples of $\mathscr{R}_{\text{sim}}$

- For the state-based LWW Register:

$$\mathscr{R}_{\text{sim}}(L,(v,t)) \Leftrightarrow \exists e \in L.E.\ oper(e) = set(v) \\ \wedge (\forall f \in L.E.\ time(f) \leq time(e))$$

- $\Sigma = V \times T$ (assume V is the value set)
- $O = \{set(v) \mid v \in V\} \cup \{get\}$
- $do((v',t'), set(v), t) = \begin{cases} (v,t) & \text{if } t > t' \\ (v',t') & \text{otherwise} \end{cases}$
- $ret((v,t), get) = v$
- $send(\sigma) = \sigma$
- $receive((v_r,t_r),(v_m,t_m)) = \begin{cases} (v_r,t_r) & \text{if } t_r > t_m \\ (v_m,t_m) & \text{otherwise} \end{cases}$

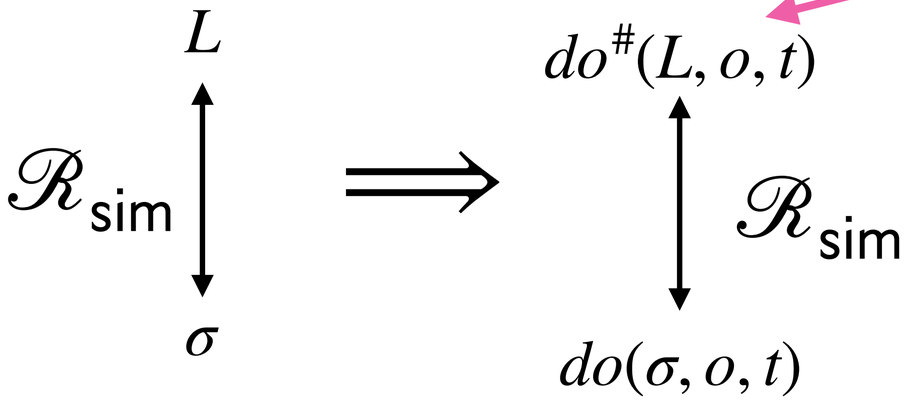
State-based LWW Register

Verification using $\mathscr{R}_{\text{sim}}$: Step-1

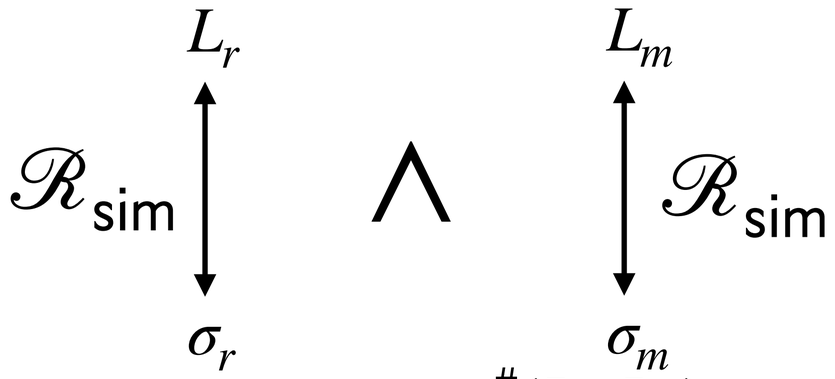
We show that $\mathscr{R}_{\text{sim}}$ holds inductively at every step in every execution

1. Verifying Operations

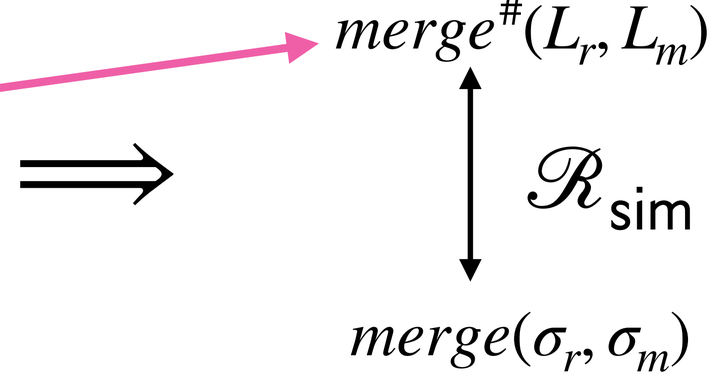
Creates a new operation context by adding a new event for the operation (o, t)



2. Verifying Merge

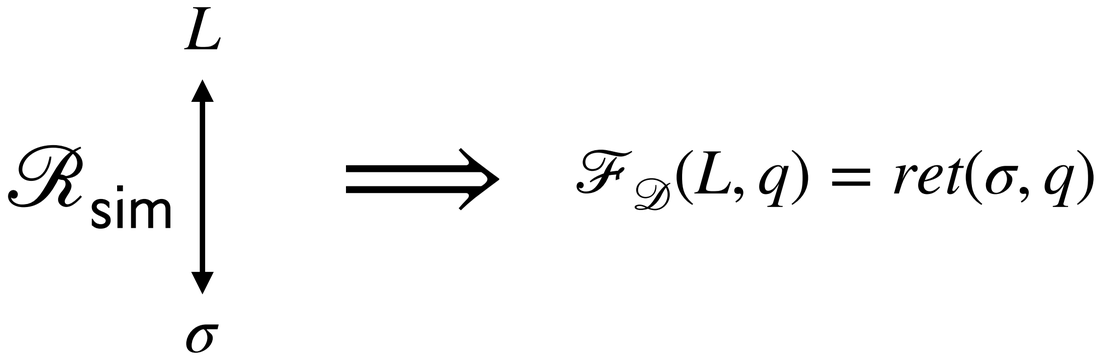


Creates a new operation context by taking a union of the contexts L_r and L_m

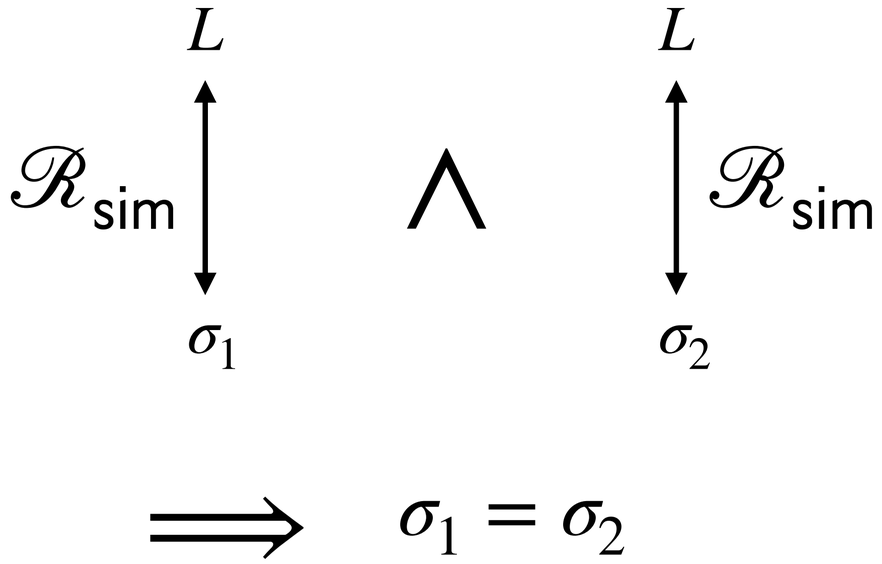


Verification using $\mathscr{R}_{\text{sim}}$: Step-2

3. Verifying RDT specification



Bonus Step: Verifying convergence



Semi-automated approach: programmer needs to provide the simulation relation

Different forms of Specifications

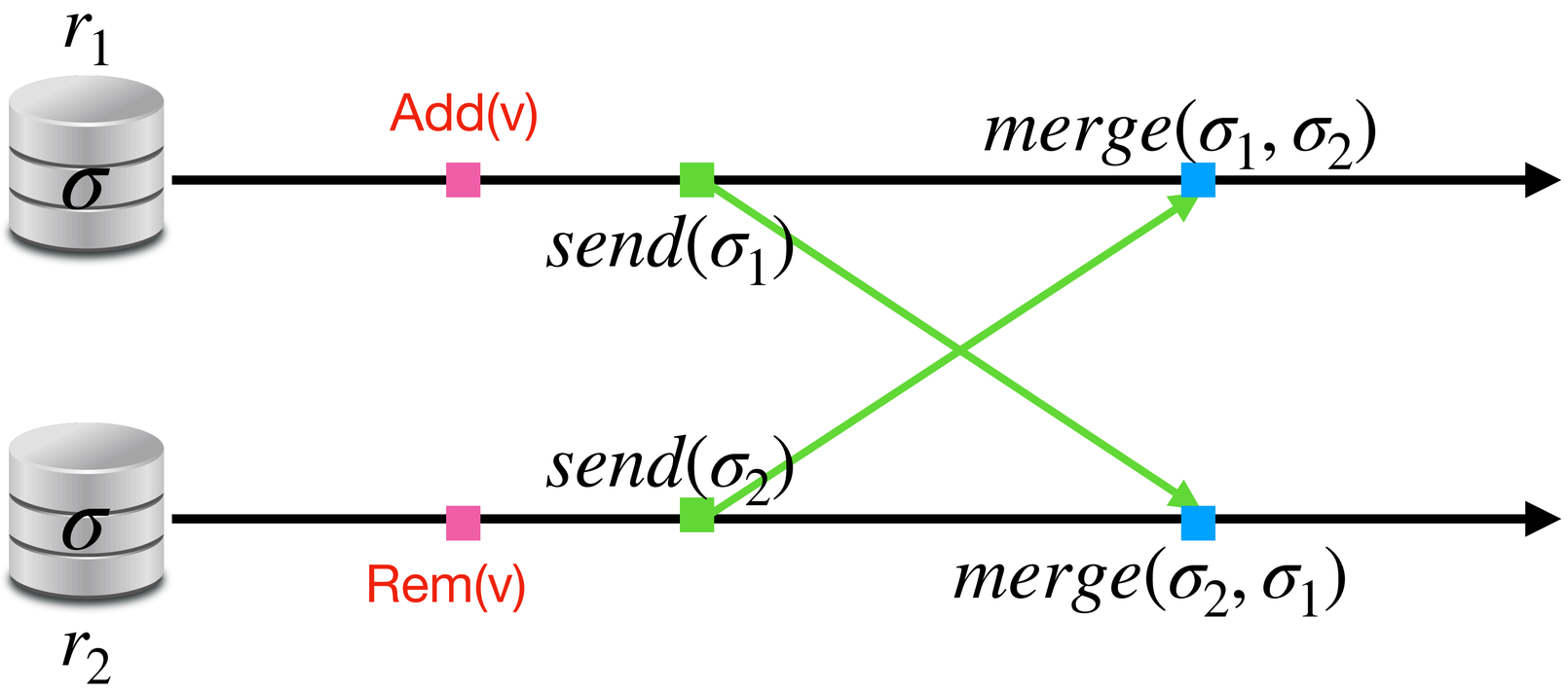
- Strong Eventual Consistency (also known as convergence)
- Declarative RDT Specification
- Replication-aware Linearizability

Replication aware Linearizability

- Inspired by linearizability in shared-memory concurrent library implementations.
- Basic idea: Each replica r's state should always be a linearisation of the updates performed at the replica, i.e. $\mathscr{C}(r)$.
- The linearisation order lo between update events should obey following properties
  - lo between two updates should remain the same at all replicas throughout the execution.
  - $hb \subseteq lo$
- In addition, for concurrent update events, a light-weight specification can be provided to always order them in a specific way.
  - Expressed as a relation (rc) over non-commutative operations.
  - E.g. for add-wins set, $rc = \{(Rem(v), Add(v)) \,|\, v \in V\}$
  - $rc \subseteq lo$

Ref: Replication aware linearizability. Enea et al.

Example : RA Linearizability for Add-wins Set



$$merge(\sigma_1, \sigma_2) = merge(\sigma_2, \sigma_1) = Add(v) \cdot Rem(v) \cdot \sigma$$

Shorthand for $do(\sigma, Rem(v), \_)$

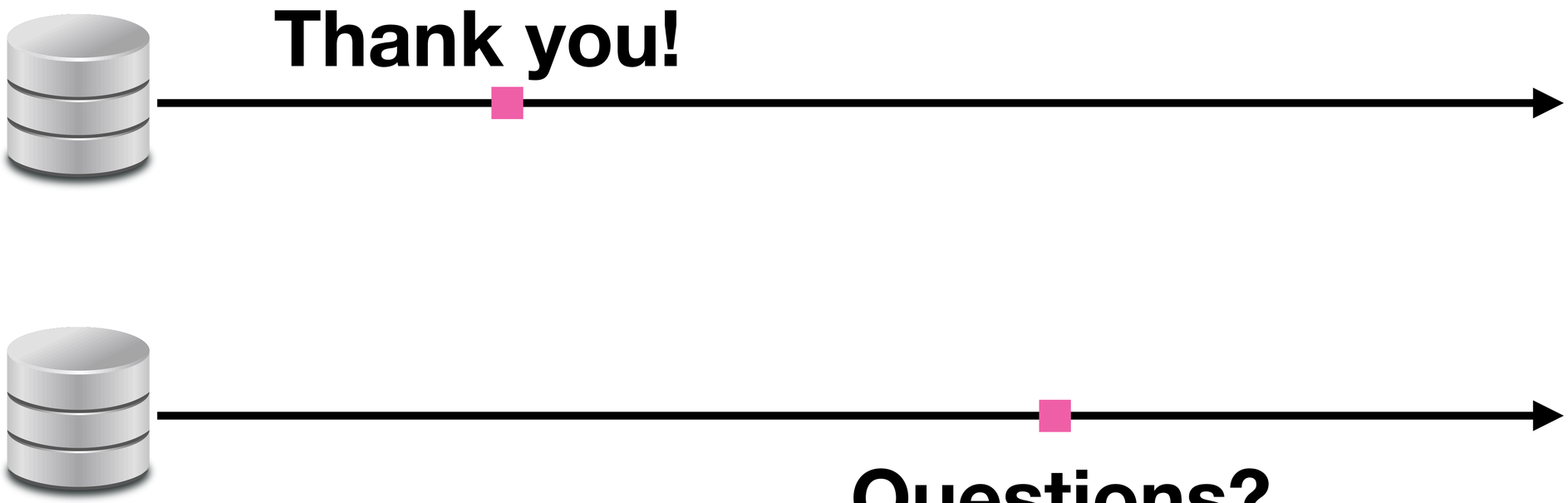
Verifying RA Linearizability using algebraic properties of $merge$

- Commutativity, associativity and idempotence of $merge$ are not sufficient.
- We require commutativity of $merge$ and do:
  - $merge(Add(v) \cdot \sigma_1, Rem(v) \cdot \sigma_2) = Add(v) \cdot merge(\sigma_1, Rem(v) \cdot \sigma_2)$
  - In general, $merge(e_1 \cdot \sigma_1, e_2 \cdot \sigma_2) = e_1 \cdot merge(\sigma_1, e_2 \cdot \sigma_2)$.
  - Applicable when $(e_2, e_1) \in rc$ or e_1 and e_2 commute with each other.
- We develop an inductive approach called Bottom-up linearisation to automatically prove such algebraic properties.

More details in our paper: Automatically Verifying Replication-aware Linearizability. OOPSLA 2025

Conclusion

- RDTs provide an elegant solution to an inherent inability of distributed systems to provide strong consistency as captured by the CAP theorem.
- Even though strong consistency cannot be achieved, RDTs nevertheless guarantee some strong correctness properties.
  - Strong Eventual Consistency
  - Declarative RDT specifications
  - Replication-aware linearizability
- Reasoning about correctness of RDTs is quite non-trivial due to infinite state-space + message-passing based semantics.
  - Still an unsolved problem as existing approaches are either not fully automated, or not complete and often fail while encountering new RDT implementations.
  - Not much is known about the decidability/complexity of the verification problem.


Thank you!
Questions?