Scalable consistency for replicated data

Annette Bieniusa

Joint work with
Marc Shapiro, Marek Zawirski (INRIA & LIP6)
Nuno Preguiça, Sérgio Duarte (UNL)
Carlos Baquero (U. Minho)

ConcoRDanT

Distributed systems are hot!

Everything is **distributed**!

- Social networks
- Games platforms
- Cloud computing
- Multicore (getting there)

Everything is **shared**!

- Processors, storage, network

Sharing resources

Small-scale local sharing demands **concurrency control**.

Large-scale (global) sharing demands **replication**.

- Replication is easy for immutable data, but hard for mutable data

Limitations

Objectives:

- Redundancy → fault tolerance
- Parallelism → high performance

Fisher, Lynch, Patterson (`85):

Consensus ∩ *Deterministic* ∩ *Asynchronous* ∩ *Faults* = ∅

CAP (Brewer `00; Gilbert & Lynch `06)

Strongly-Consistent ∩ *Available* ∩ *Partition-Tolerant* = ∅

Outline

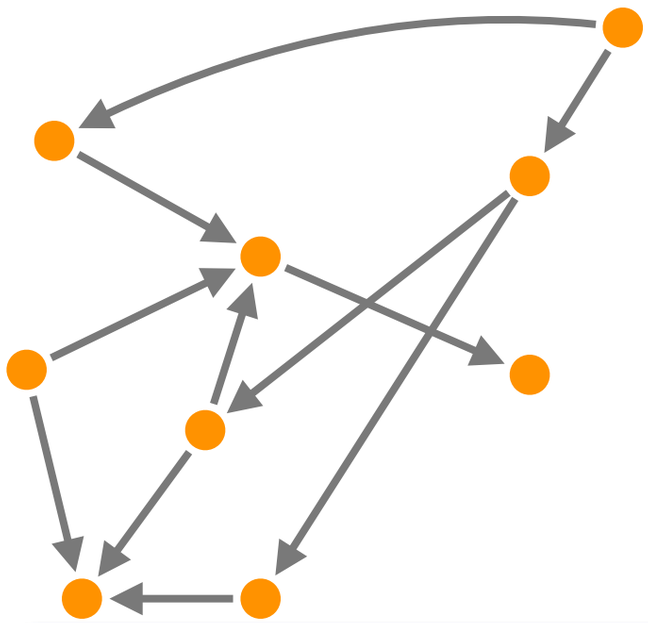
1. Strong vs. eventual consistency

2. Conflict-free replication

3. *SwiftCloud*: Geo-replication all the way to the edge

Strong vs. eventual consistency

Large shared data structure

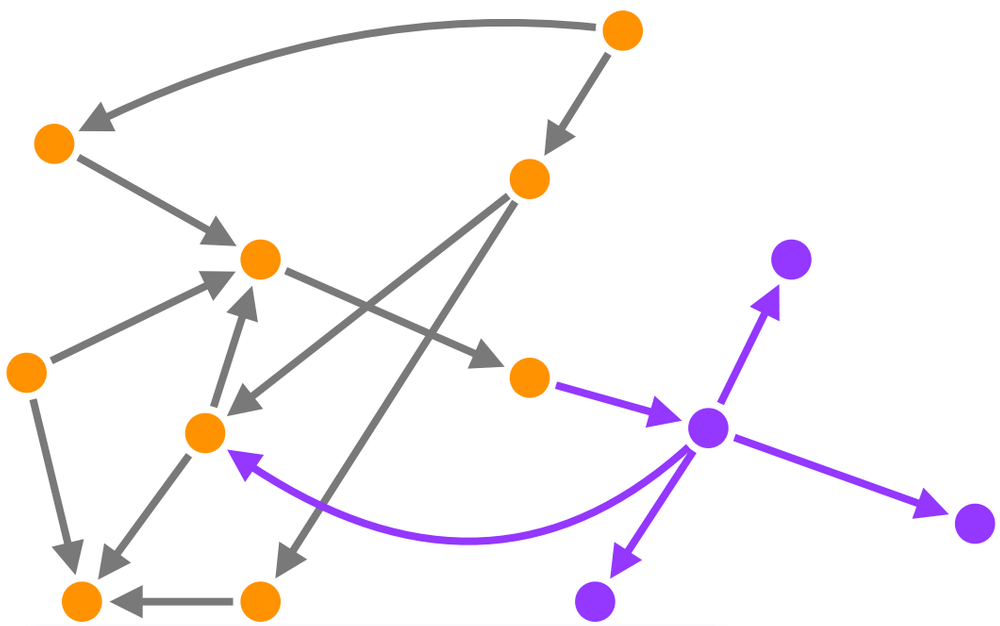


- Large, dynamic graph
- Incremental, parallel, asynchronous:
  - updates
  - processing

Wish list:

- Mutable
- Incremental
- Fast
- Fault tolerant
- Principled

Large shared data structure

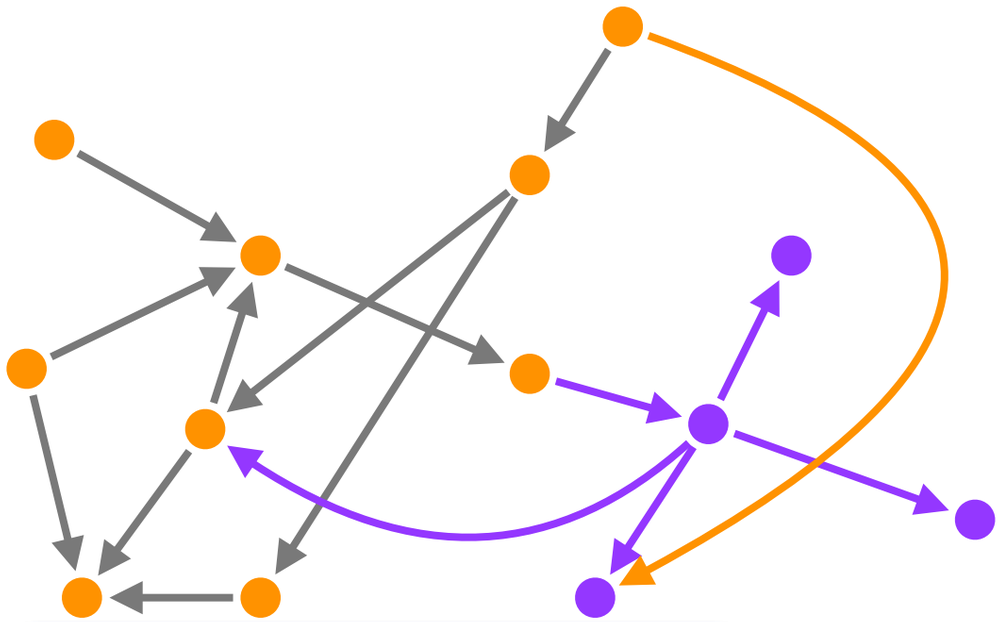


- Large, dynamic graph
- Incremental, parallel, asynchronous:
  - updates
  - processing

Wish list:

- Mutable
- Incremental
- Fast
- Fault tolerant
- Principled

Large shared data structure

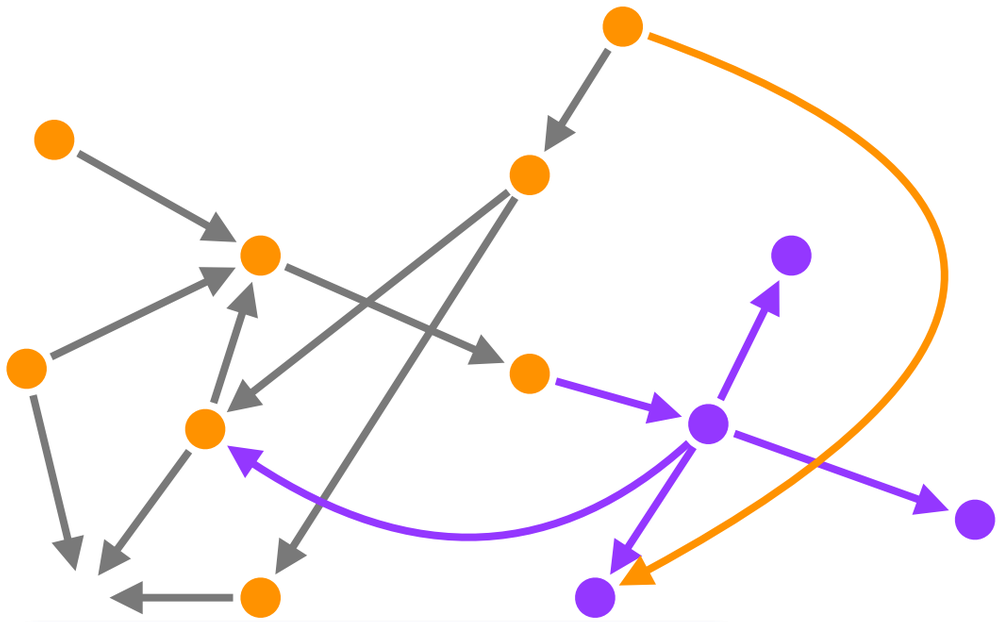


- Large, dynamic graph
- Incremental, parallel, asynchronous:
  - updates
  - processing

Wish list:

- Mutable
- Incremental
- Fast
- Fault tolerant
- Principled

Large shared data structure

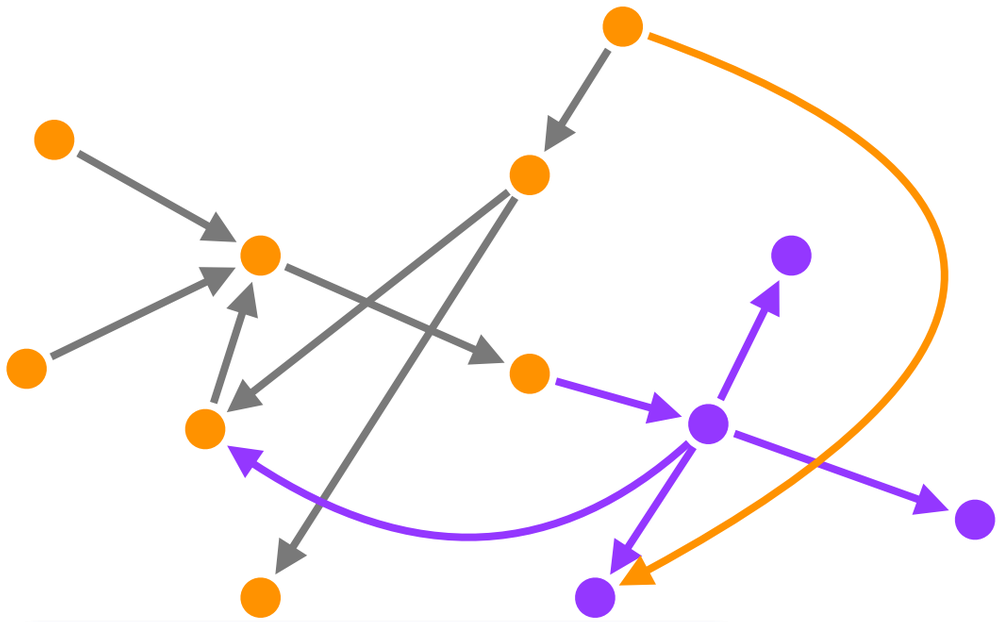


- Large, dynamic graph
- Incremental, parallel, asynchronous:
  - updates
  - processing

Wish list:

- Mutable
- Incremental
- Fast
- Fault tolerant
- Principled

Large shared data structure

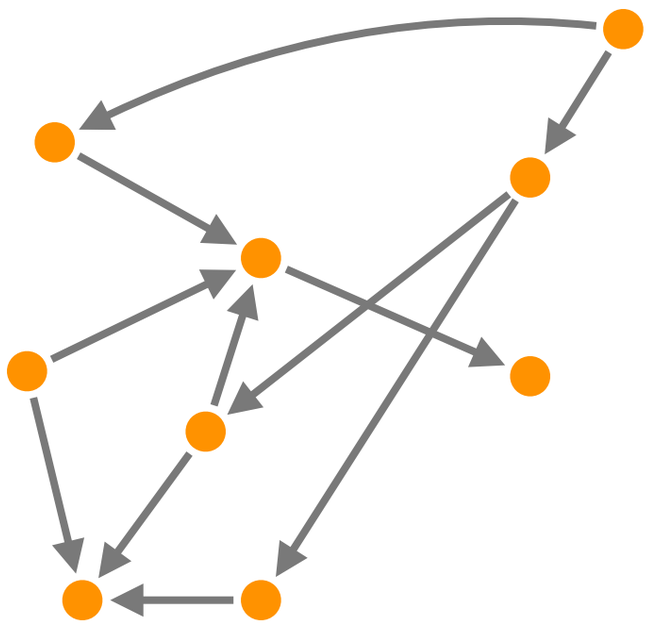


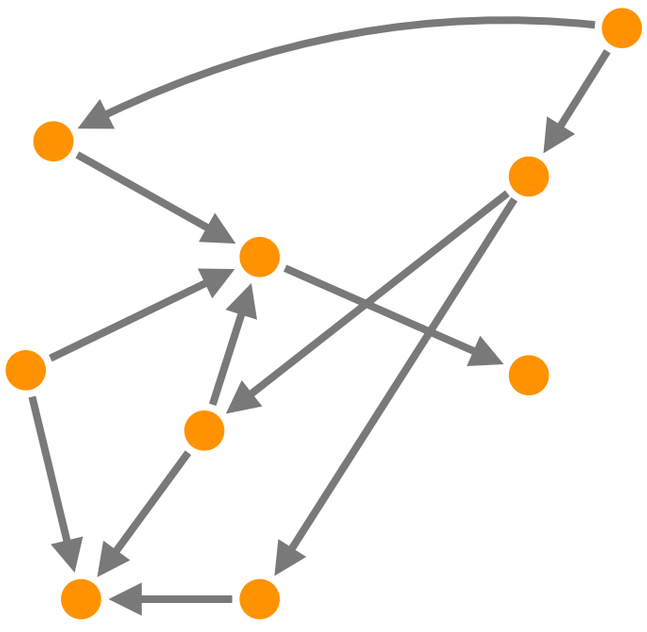
- Large, dynamic graph
- Incremental, parallel, asynchronous:
  - updates
  - processing

Wish list:

- Mutable
- Incremental
- Fast
- Fault tolerant
- Principled

State machine replication





- Conflict = concurrent access that violate an invariant
- SMR: no concurrency

Idea: Preclude conflict

- Single total order
- Requires consensus
- Serialisation bottleneck!

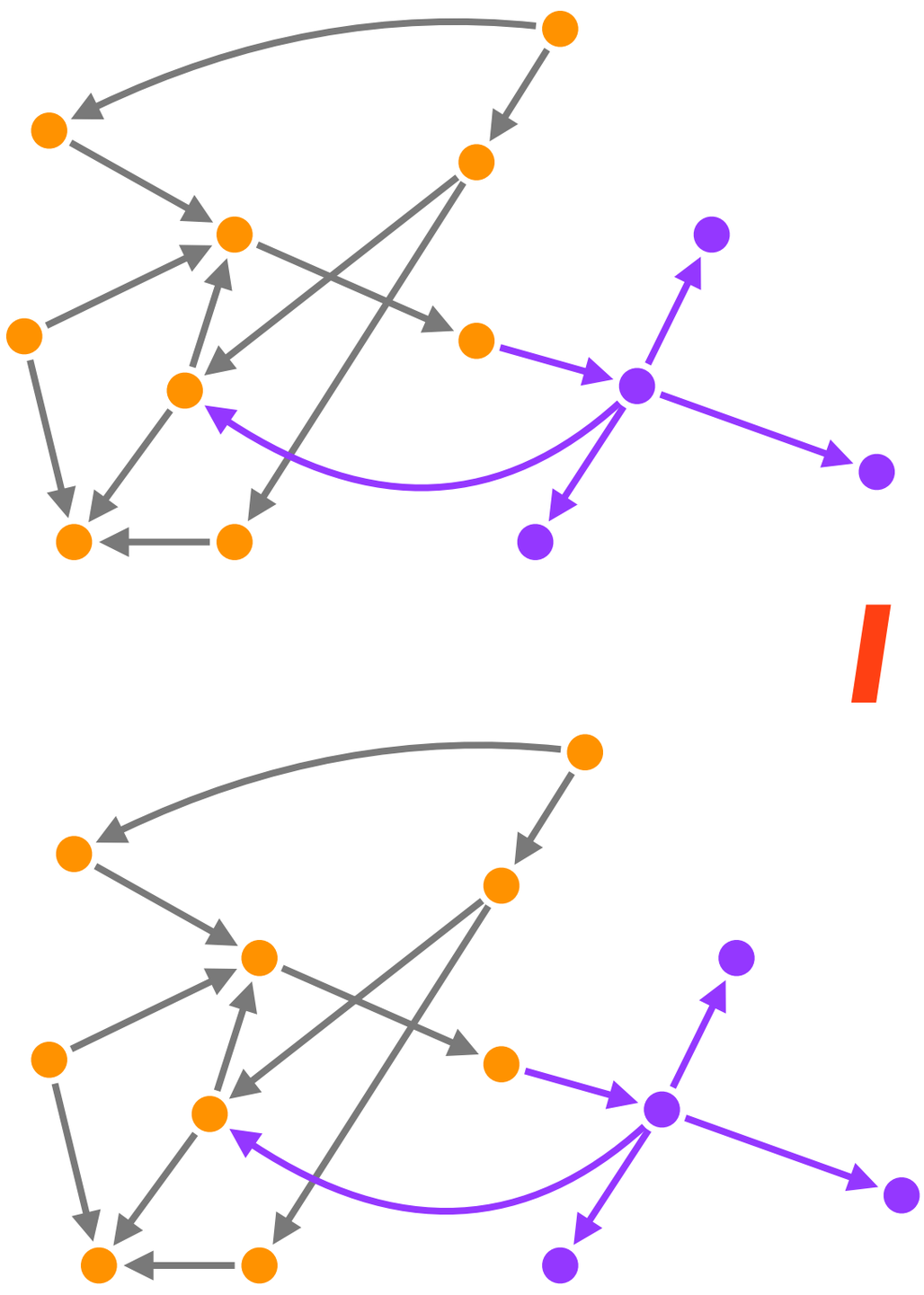
Very general

Strong consistency

- Simultaneous N-way agreement

- No faster than a single sequential computer
- Actually, slower

State machine replication



- Conflict = concurrent access that violate an invariant
- SMR: no concurrency

Idea: Preclude conflict

- Single total order
- Requires consensus
  - Serialisation bottleneck!

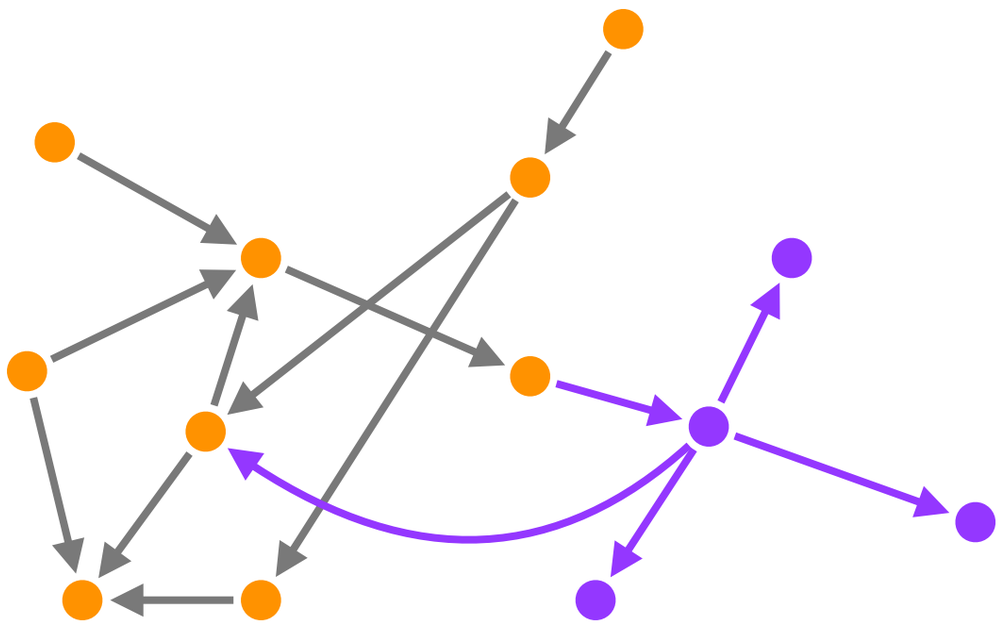
Very general

Strong consistency

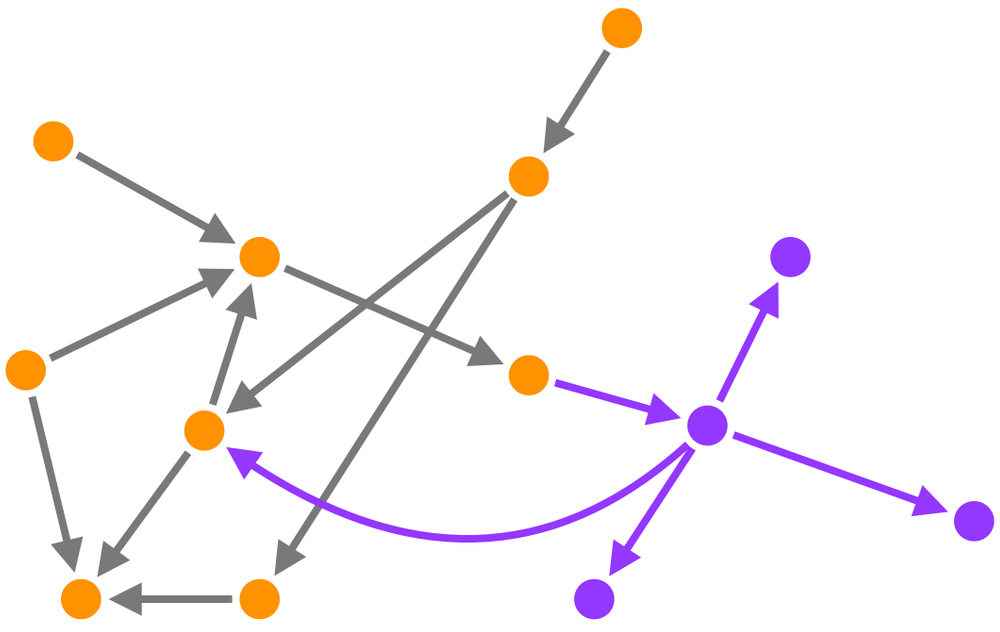
- Simultaneous N-way agreement

- No faster than a single sequential computer
- Actually, slower

State machine replication



2



- Conflict = concurrent access that violate an invariant
- SMR: no concurrency

Idea: Preclude conflict

- Single total order
- Requires consensus
  - Serialisation bottleneck!

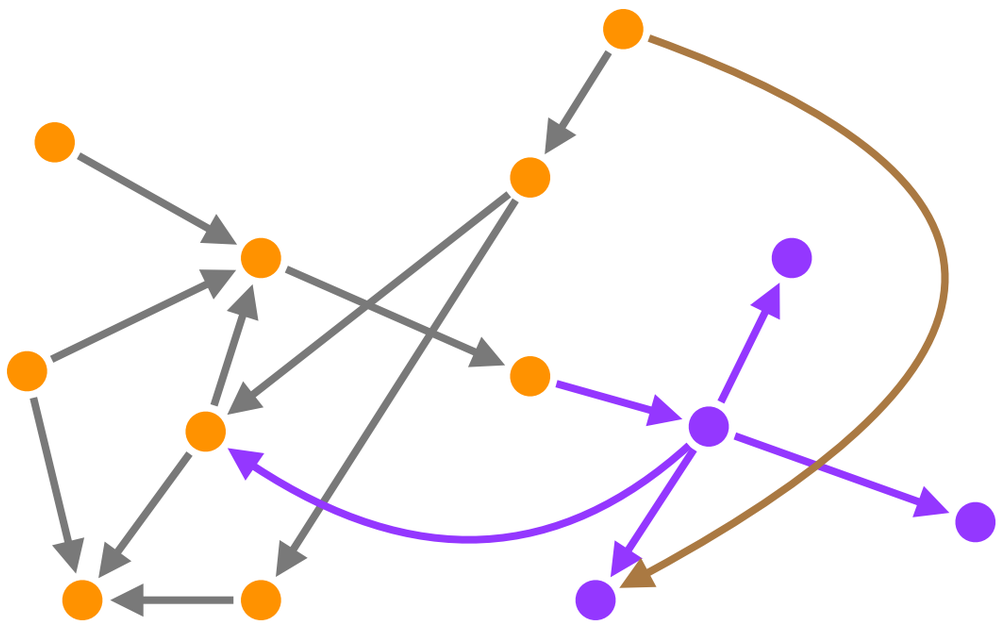
Very general

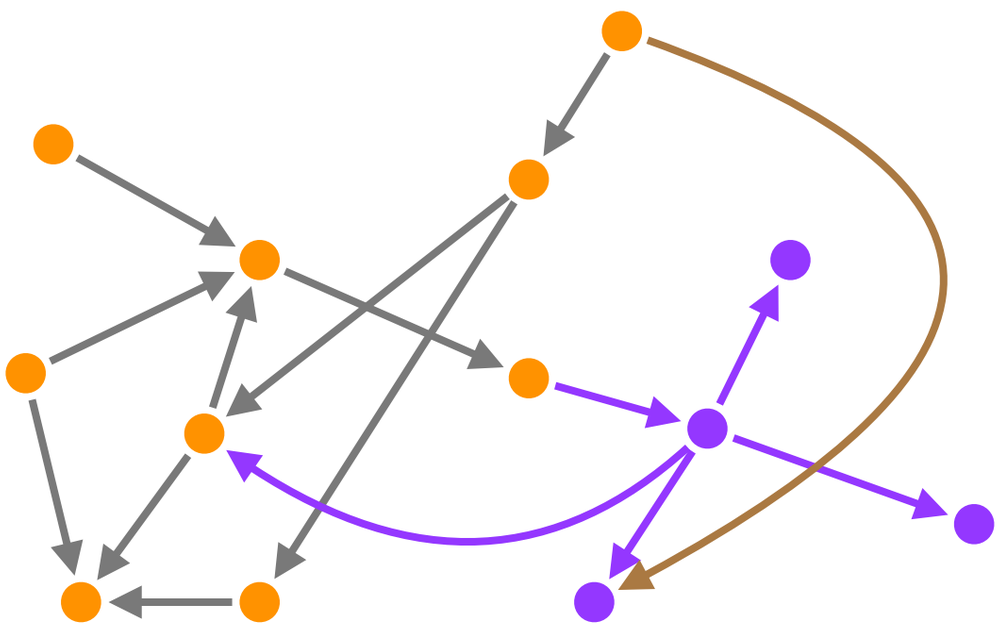
Strong consistency

- Simultaneous N-way agreement

- No faster than a single sequential computer
- Actually, slower

State machine replication





3

- Conflict = concurrent access that violate an invariant
- SMR: no concurrency

Idea: Preclude conflict

- Single total order
- Requires consensus
  - Serialisation bottleneck!

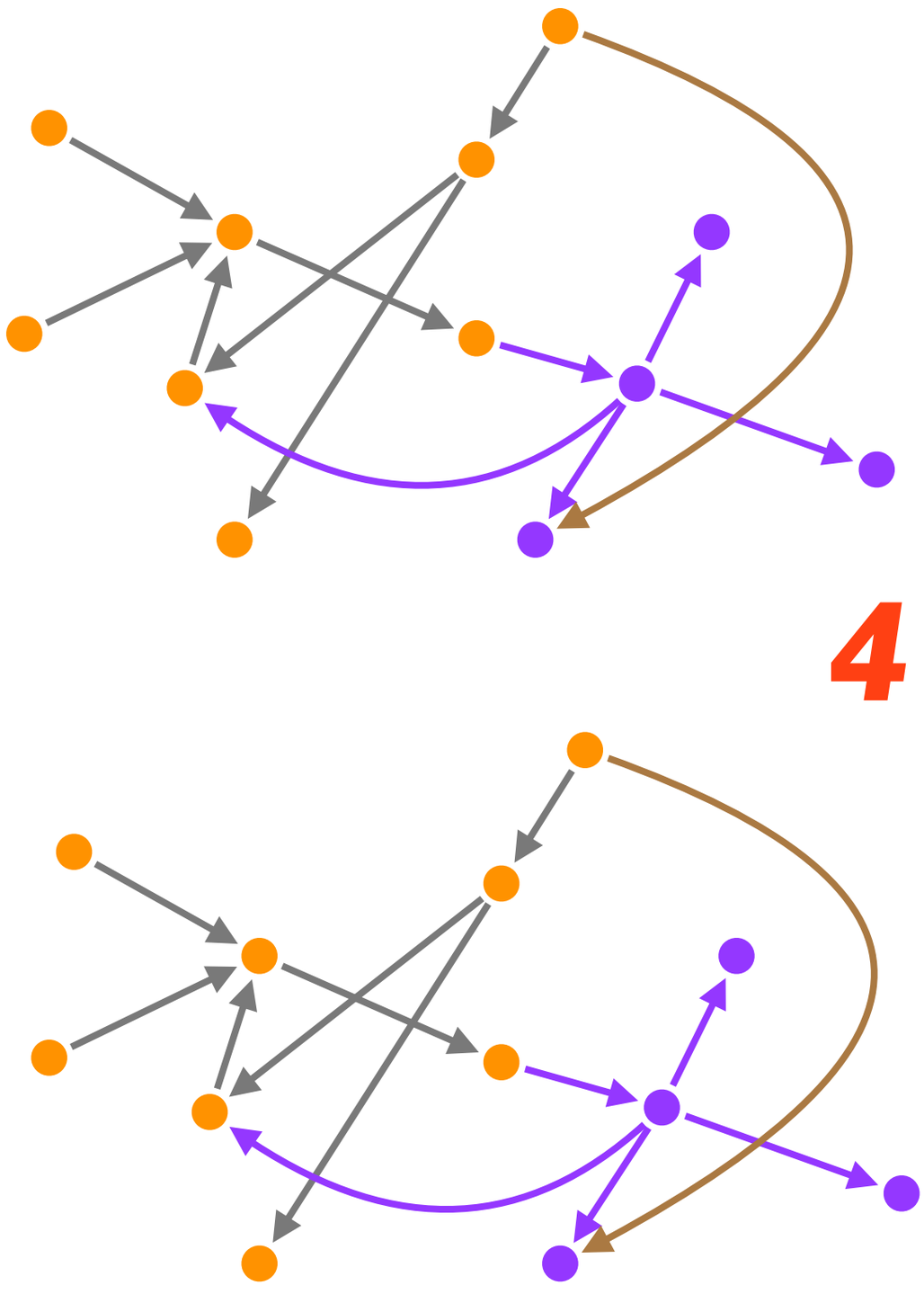
Very general

Strong consistency

- Simultaneous N-way agreement

- No faster than a single sequential computer
- Actually, slower

State machine replication



- Conflict = concurrent access that violate an invariant
- SMR: no concurrency

Idea: Preclude conflict

- Single total order
- Requires consensus
  - Serialisation bottleneck!

Very general

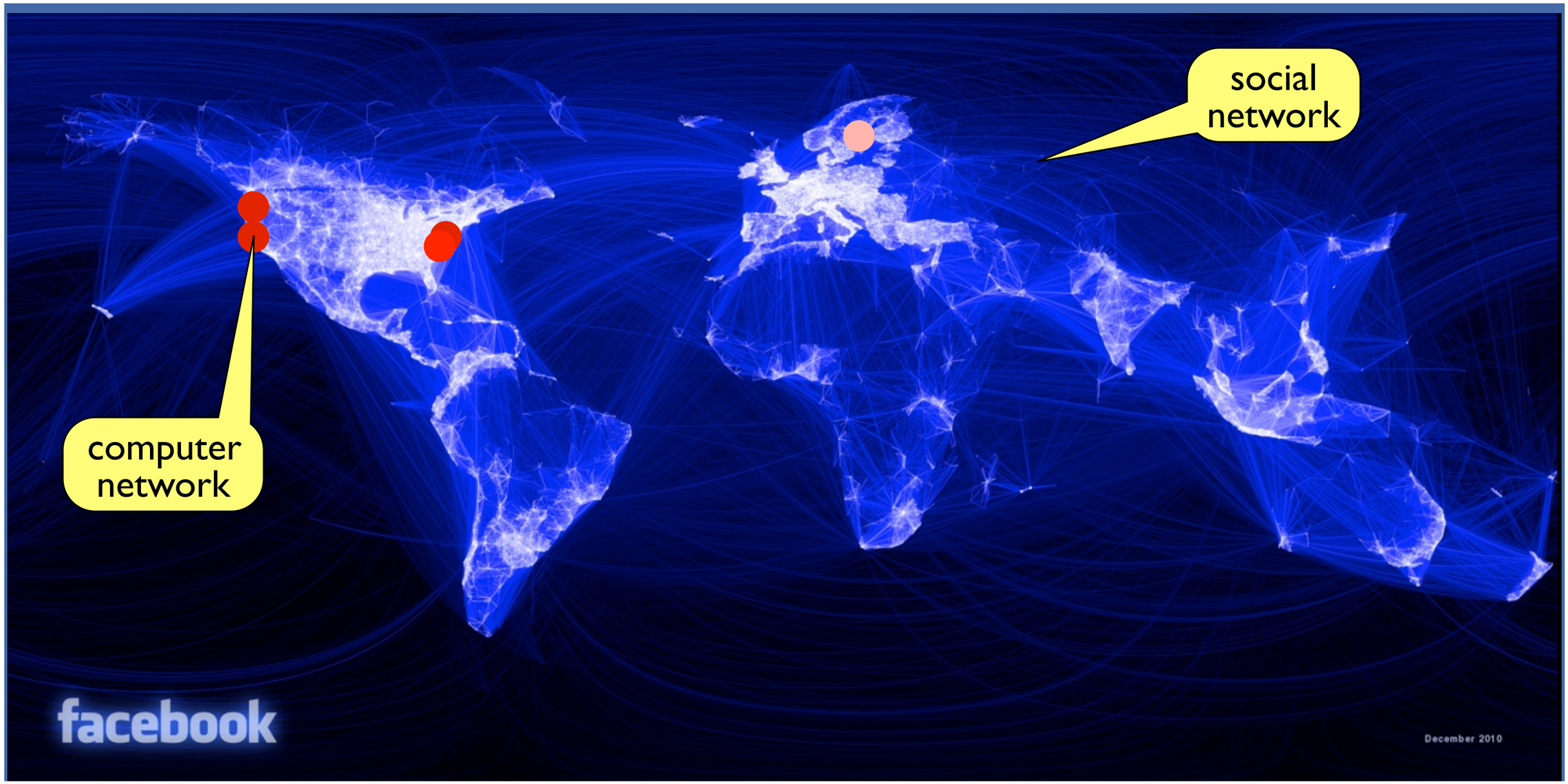
Strong consistency

- Simultaneous N-way agreement

- No faster than a single sequential computer
- Actually, slower

- Transactions
- social network: not partitionable

The Facebook networks



- 4 (5?) data centres: 1 RW, others RO
- Ad-hoc hacks for speed ≈ consistent?

Eventual Consistency

- parallel
- human in the loop)

Design point: mobile computing

- Availability, parallelism
- Crash-recovery fault model

- perfect failure detector

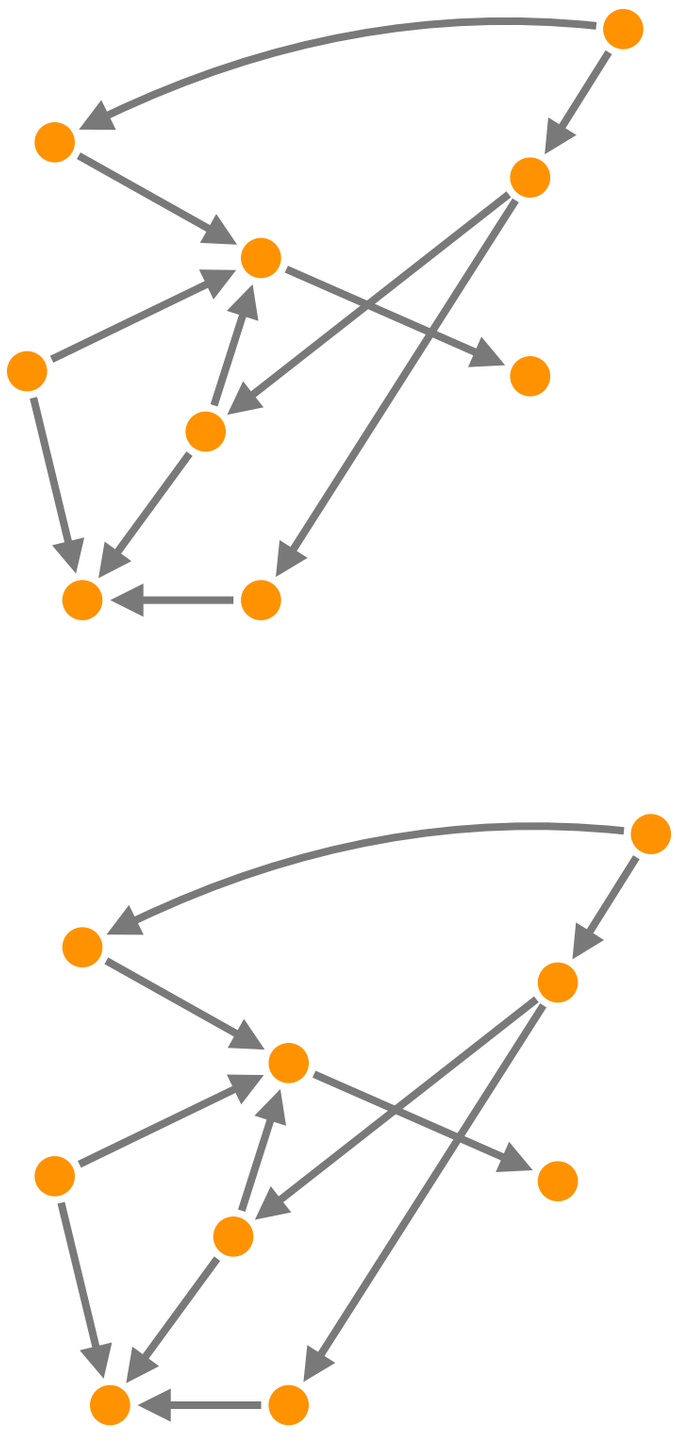
Update each replica independently

- Transport to other replicas
- Replay or merge

Guaranteed deliver: eventually, all replicas receive all updates

- Hopefully they converge...
- But order of updates differs!

Be optimistic!



Update local + propagate

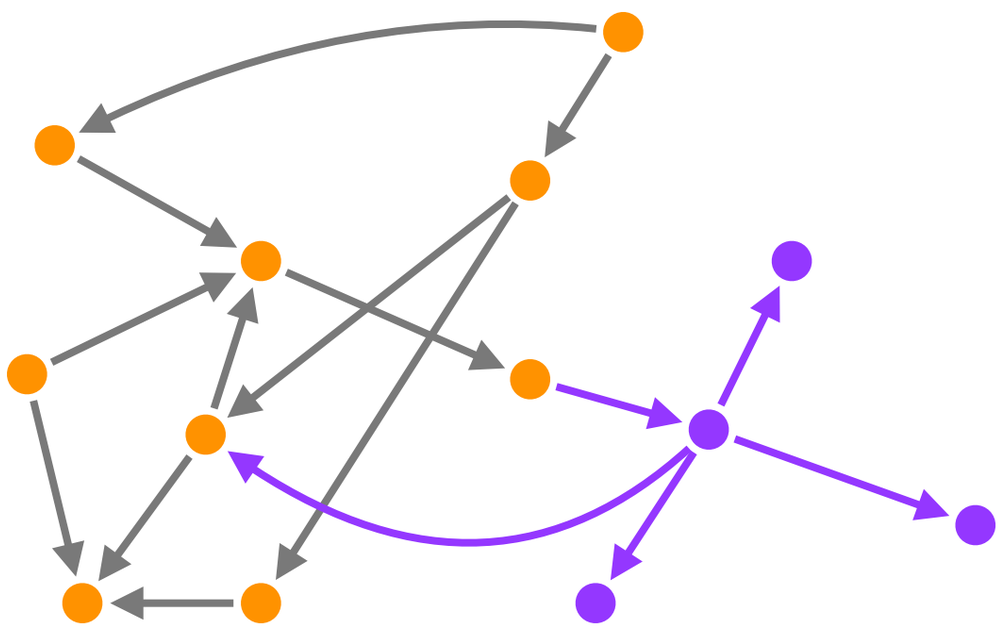
- No foreground synch
- Optimistic speculation
- Eventual, reliable delivery

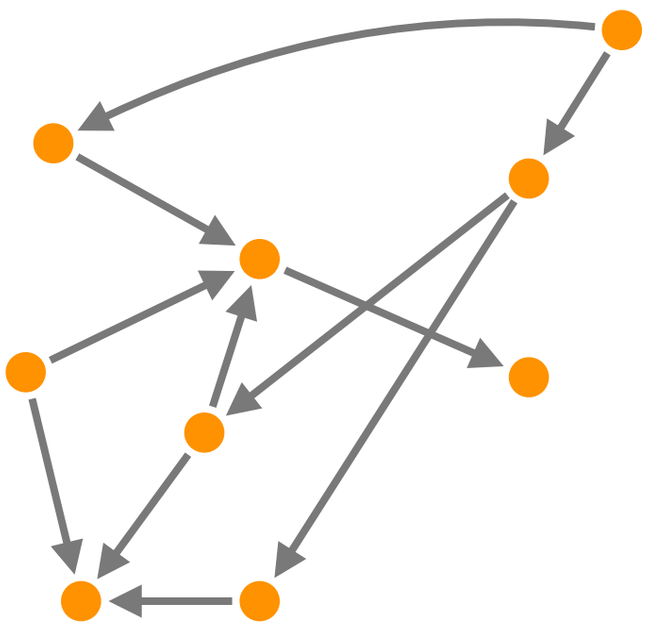
On conflict

- **Arbitrate** (in background)
- Roll-back

- ↗ available, ↗ responsive
- Complex and error-prone
- Expose tentative state

Be optimistic!





Update local + propagate

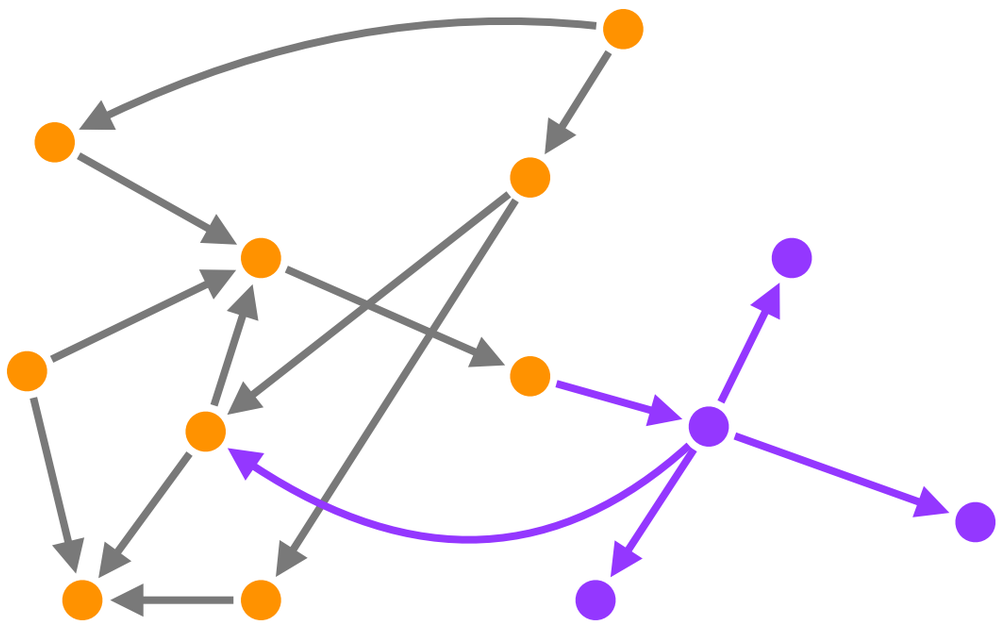
- No foreground synch
- Optimistic speculation
- Eventual, reliable delivery

On conflict

- **Arbitrate** (in background)
- Roll-back

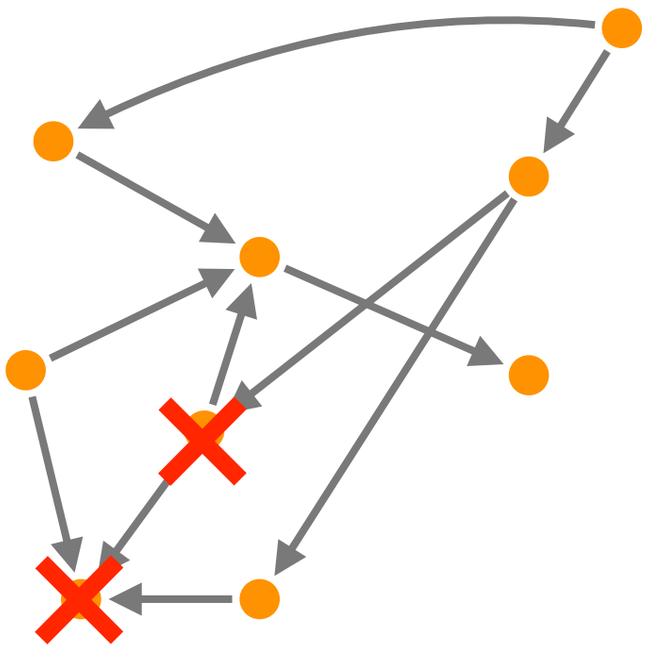
- ↗ available, ↗ responsive
- Complex and error-prone
- Expose tentative state

Be optimistic!



Update local + propagate

- No foreground synch
- Optimistic speculation
- Eventual, reliable delivery



On conflict

- **Arbitrate** (in background)
- Roll-back

- ↗ available, ↗ responsive
- Complex and error-prone
- Expose tentative state

Be optimistic!

Update local + propagate

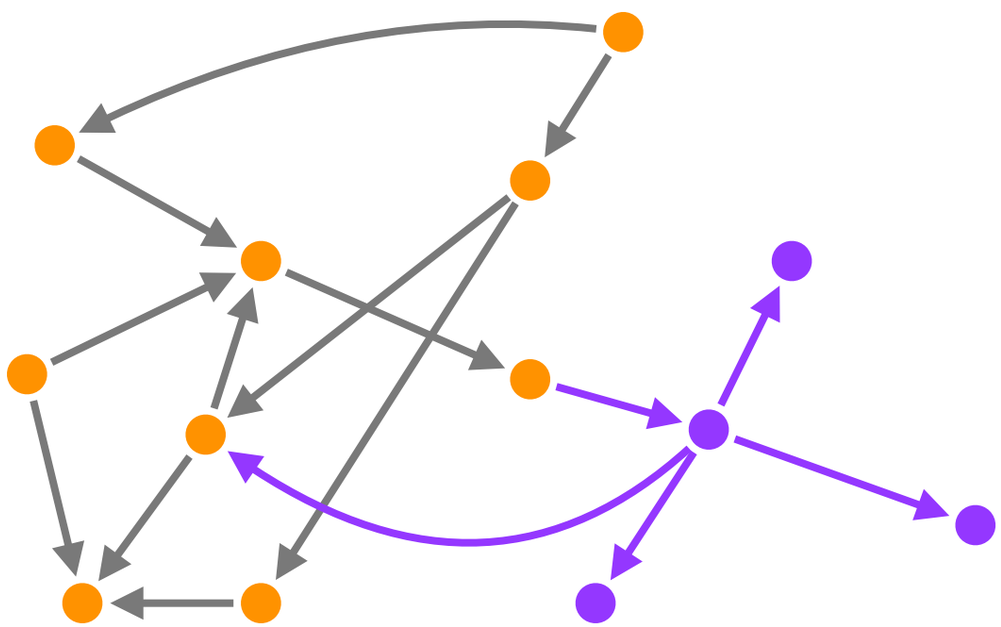
- No foreground synch
- Optimistic speculation
- Eventual, reliable delivery

On conflict

- **Arbitrate** (in background)
- Roll-back

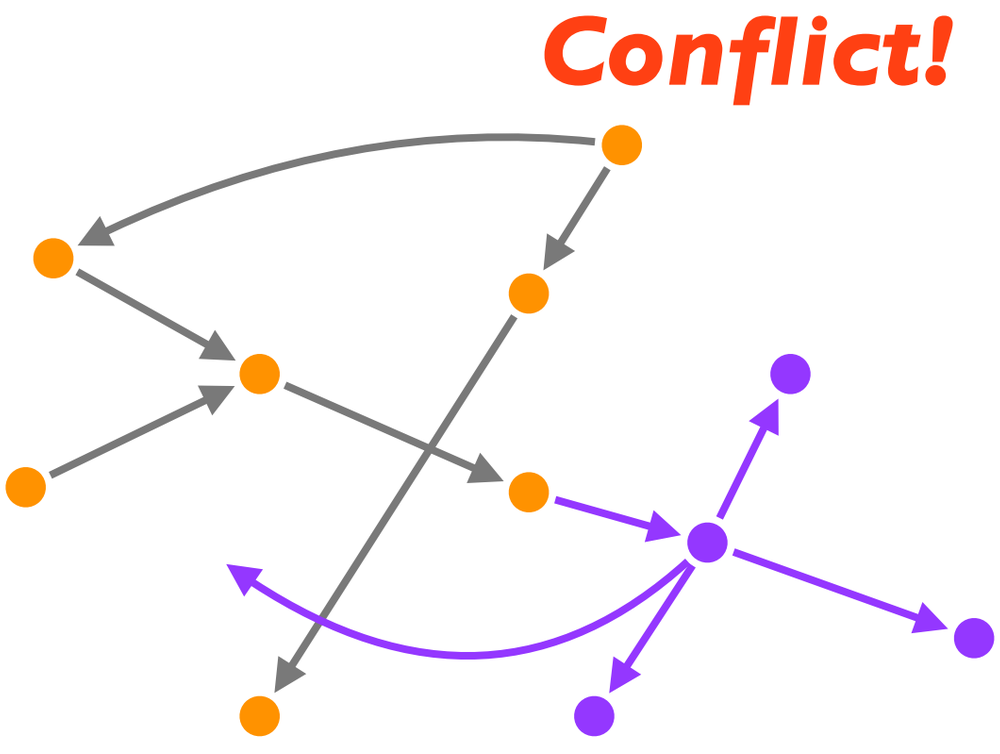
- ↗ available, ↗ responsive
- Complex and error-prone
- Expose tentative state

Be optimistic!



Update local + propagate

- No foreground synch
- Optimistic speculation
- Eventual, reliable delivery

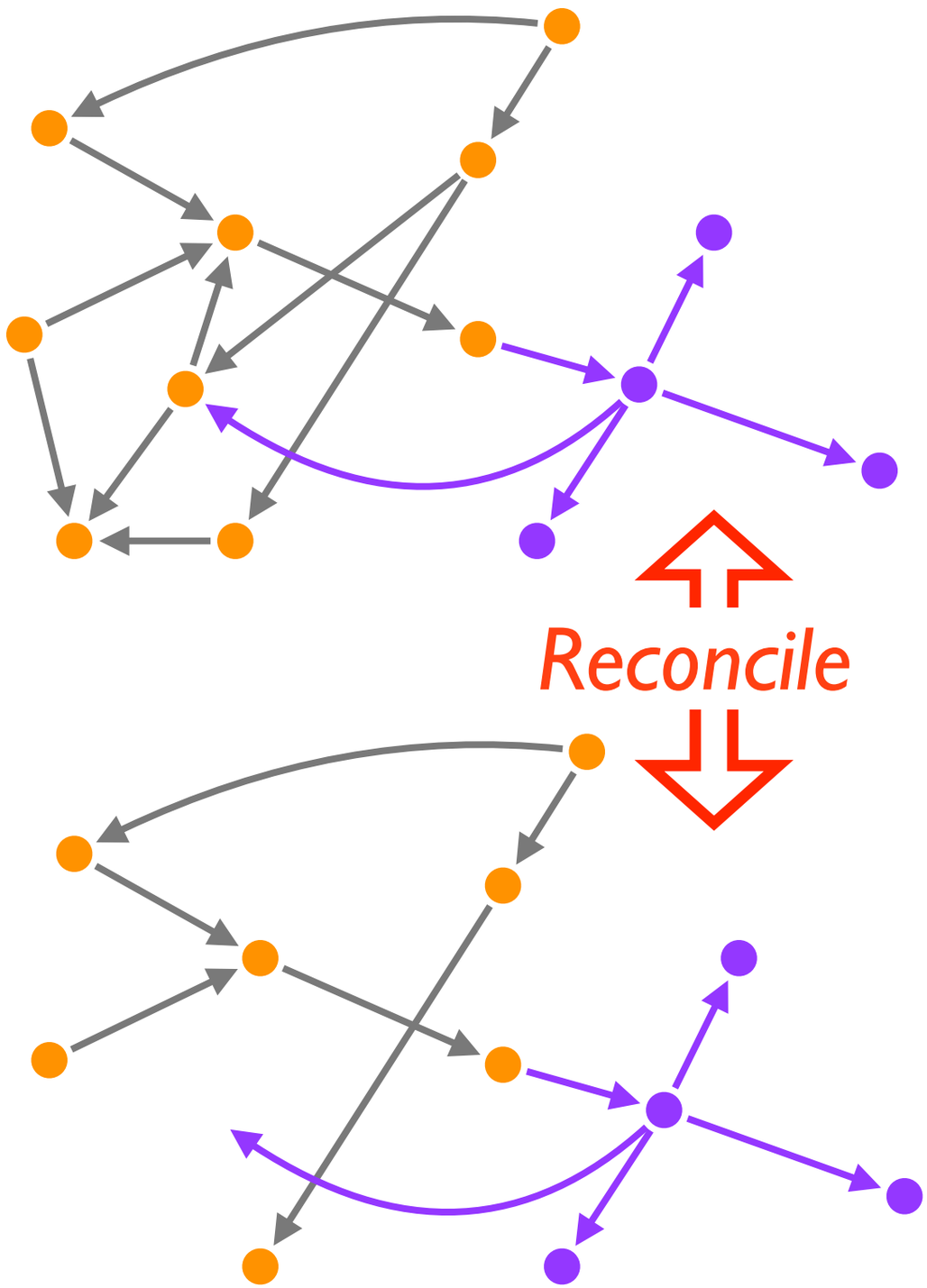


On conflict

- **Arbitrate** (in background)
- Roll-back

- ↗ available, ↗ responsive
- Complex and error-prone
- Expose tentative state

Be optimistic!



Update local + propagate

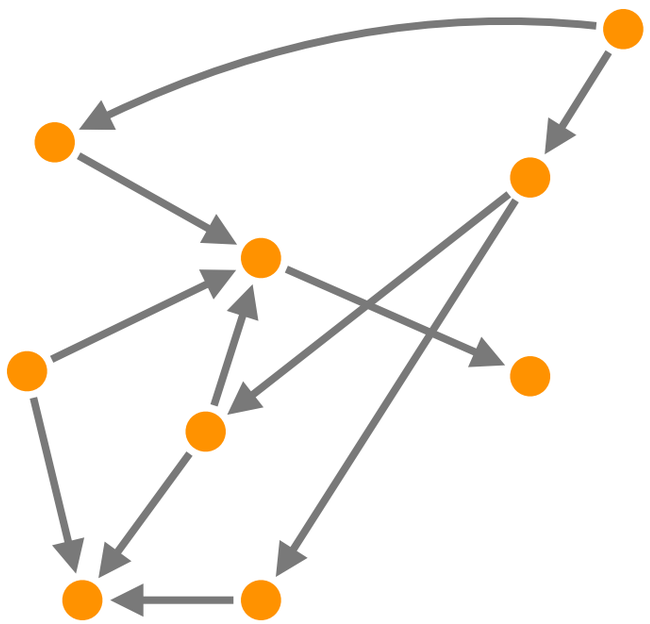
- No foreground synch
- Optimistic speculation
- Eventual, reliable delivery

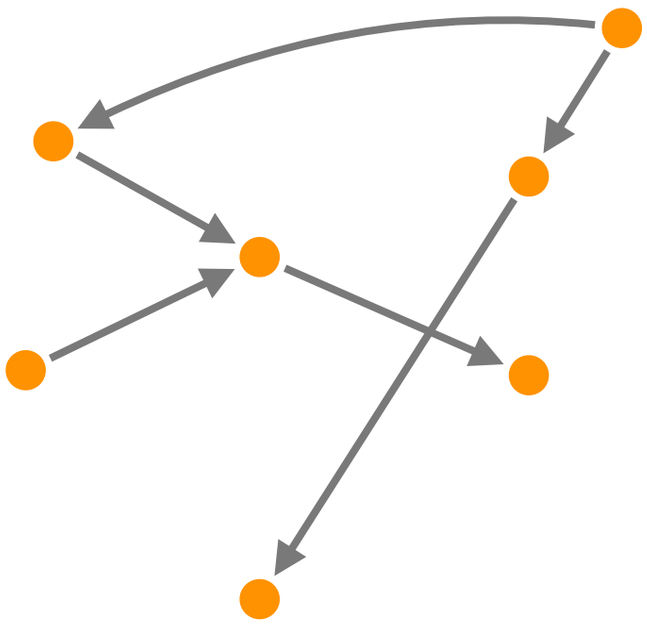
On conflict

- **Arbitrate** (in background)
- Roll-back

- ↗ available, ↗ responsive
- Complex and error-prone
- Expose tentative state

Be optimistic!





Update local + propagate

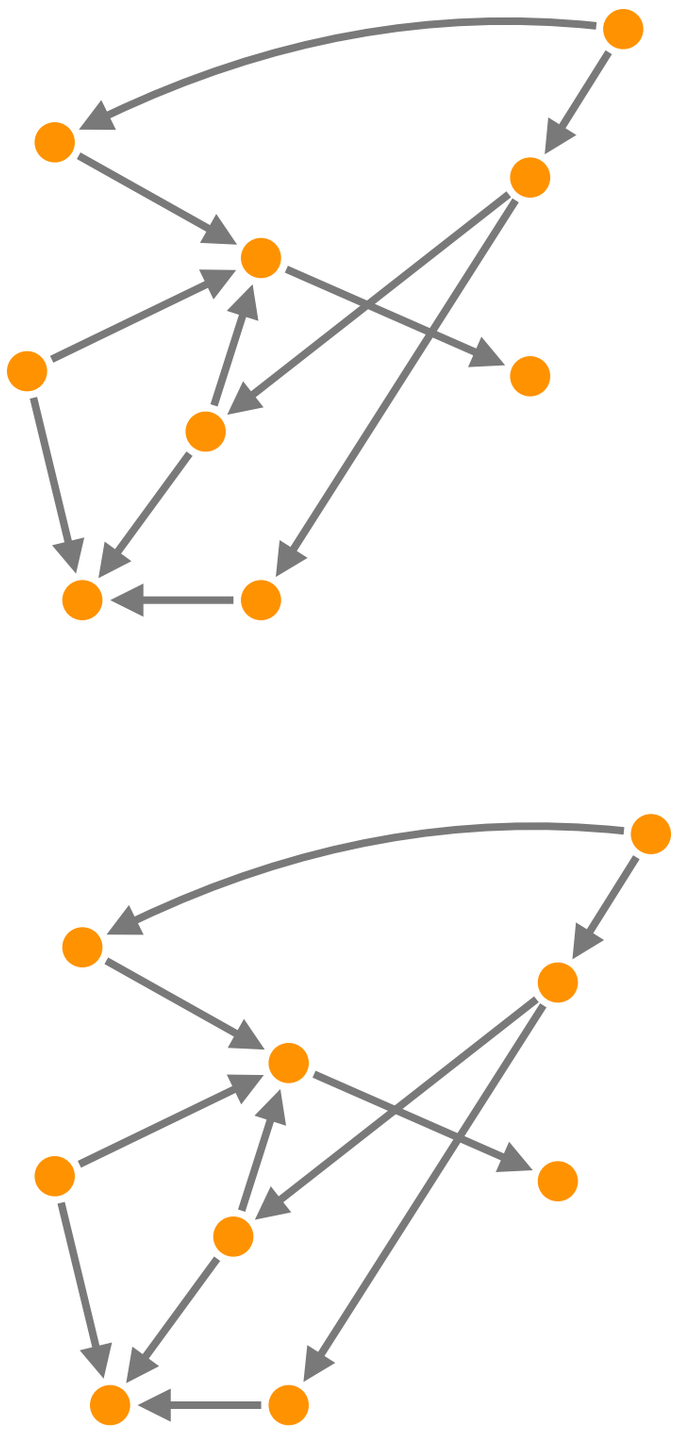
- No foreground synch
- Optimistic speculation
- Eventual, reliable delivery

On conflict

- **Arbitrate** (in background)
- Roll-back

- ↗ available, ↗ responsive
- Complex and error-prone
- Expose tentative state

Be optimistic!



Update local + propagate

- No foreground synch
- Optimistic speculation
- Eventual, reliable delivery

On conflict

- **Arbitrate** (in background)
- Roll-back

- ↗ available, ↗ responsive
- Complex and error-prone
- Expose tentative state

Why conflict-detection
or prevention...

if you can have it
conflict-free!

Conflict-free replication

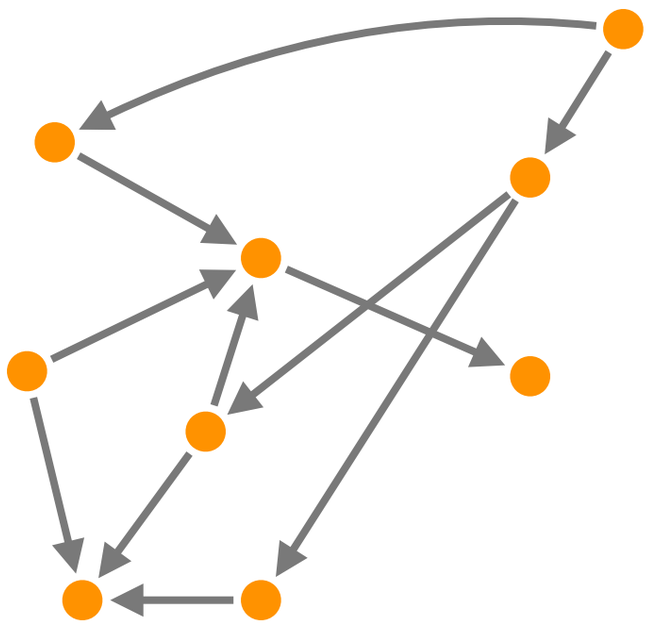
What we have seen so far

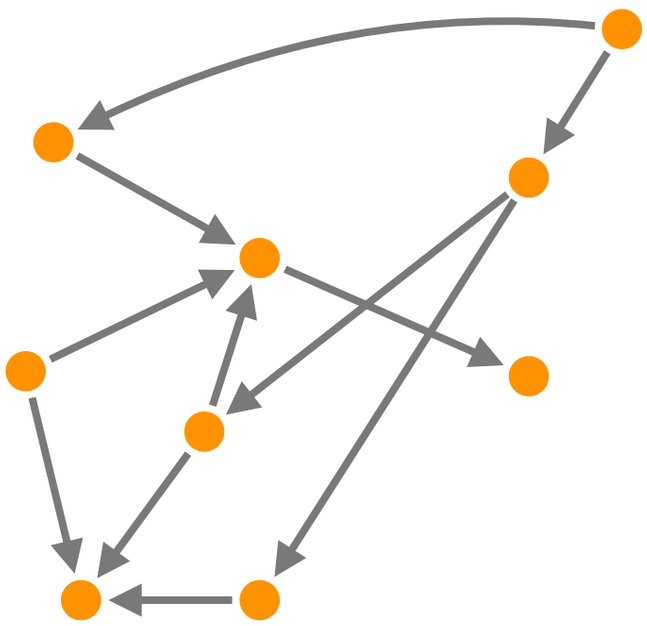
- Strong consistency doesn't scale
- Eventual consistency is complicated
- Main issue: Conflicts!

- Conflict: concurrent updates that violate an invariant

- No arbitration, no roll-back, no consensus
- Concurrent updates must have deterministic outcome

Strong Eventual Consistency





- Available, responsive
- More parallelism
- No conflicts
- No rollback

Update local + propagate

- No synchronization
- Update is durable
- Broadcast

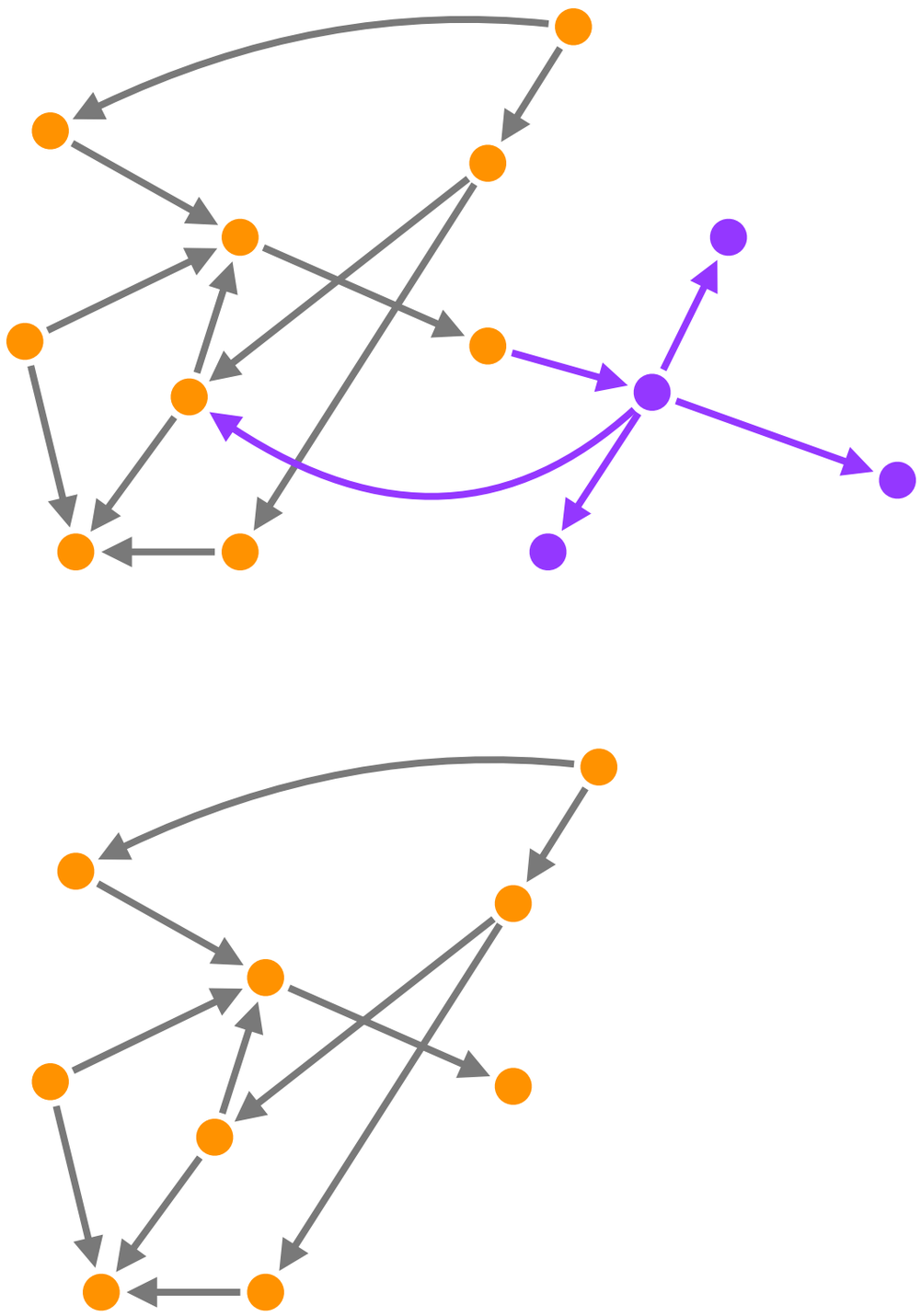
No conflict

- Unique outcome of concurrent updates

No consensus: ≤ n-1 faults

Fast, responsive

Strong Eventual Consistency



- Available, responsive
- More parallelism
- No conflicts
- No rollback

Update local + propagate

- No synchronization
- Update is durable
- Broadcast

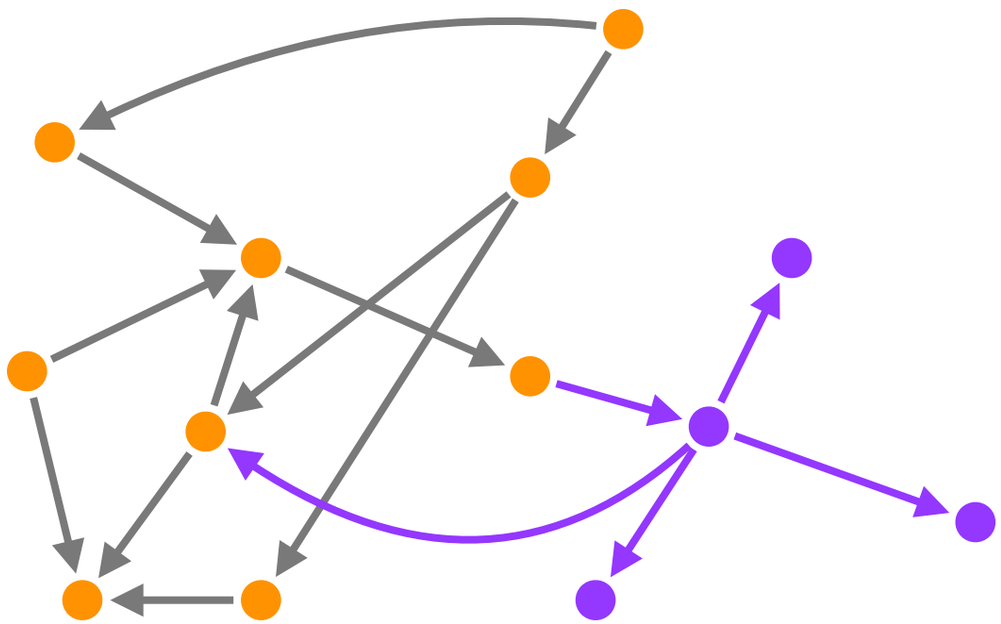
No conflict

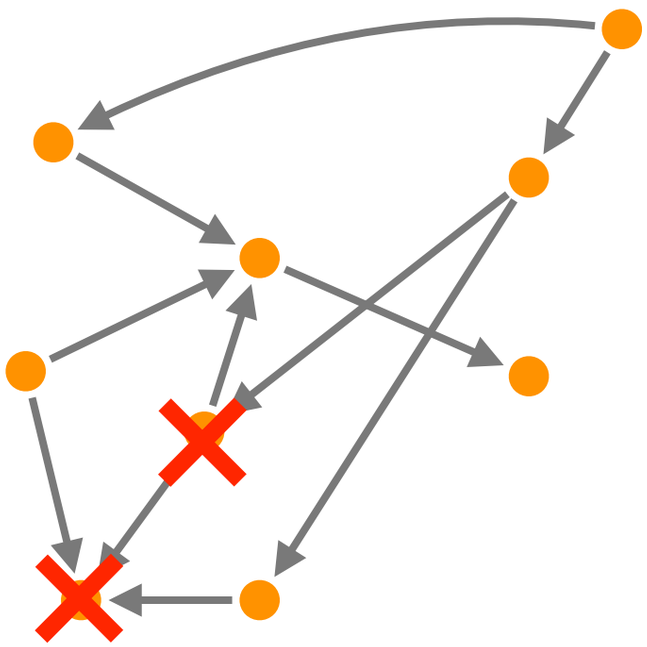
- Unique outcome of concurrent updates

No consensus: ≤ n-1 faults

Fast, responsive

Strong Eventual Consistency





- Available, responsive
- More parallelism
- No conflicts
- No rollback

Update local + propagate

- No synchronization
- Update is durable
- Broadcast

No conflict

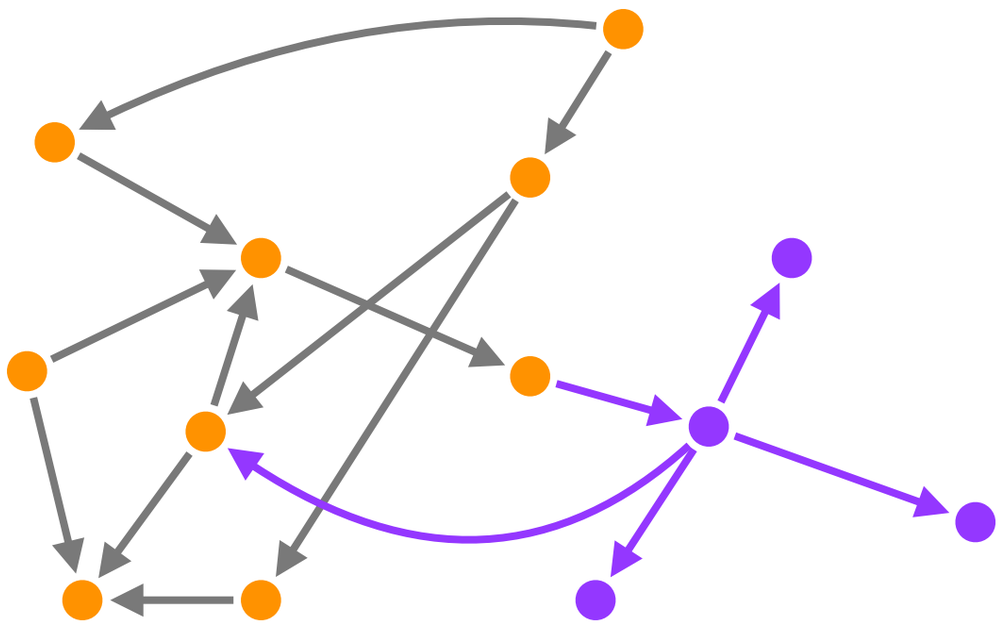
- Unique outcome of concurrent updates

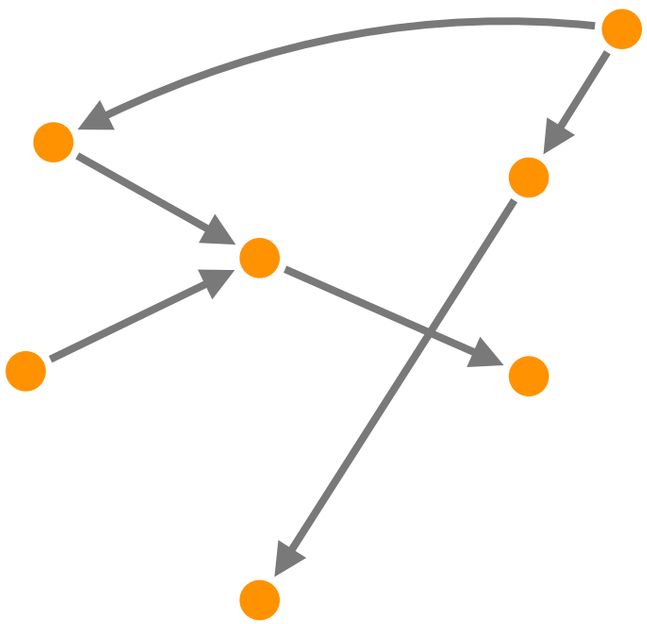
No consensus: ≤ n-1 faults

Fast, responsive

Strong Eventual Consistency

- Available, responsive
- More parallelism
- No conflicts
- No rollback





Update local + propagate

- No synchronization
- Update is durable
- Broadcast

No conflict

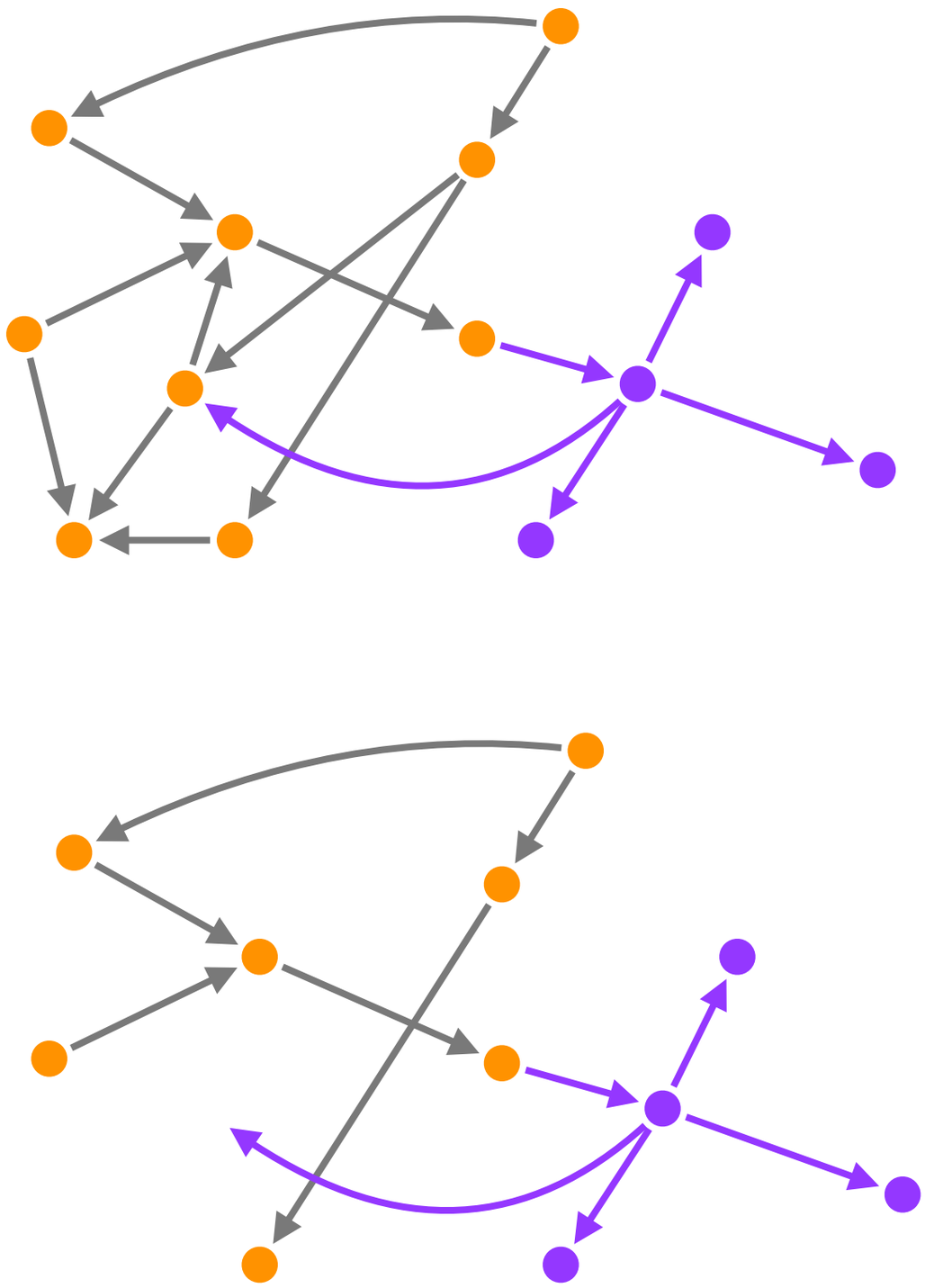
- Unique outcome of concurrent updates

No consensus: ≤ n-1 faults

Fast, responsive

Strong Eventual Consistency

- Available, responsive
- More parallelism
- No conflicts
- No rollback



Update local + propagate

- No synchronization
- Update is durable
- Broadcast

No conflict

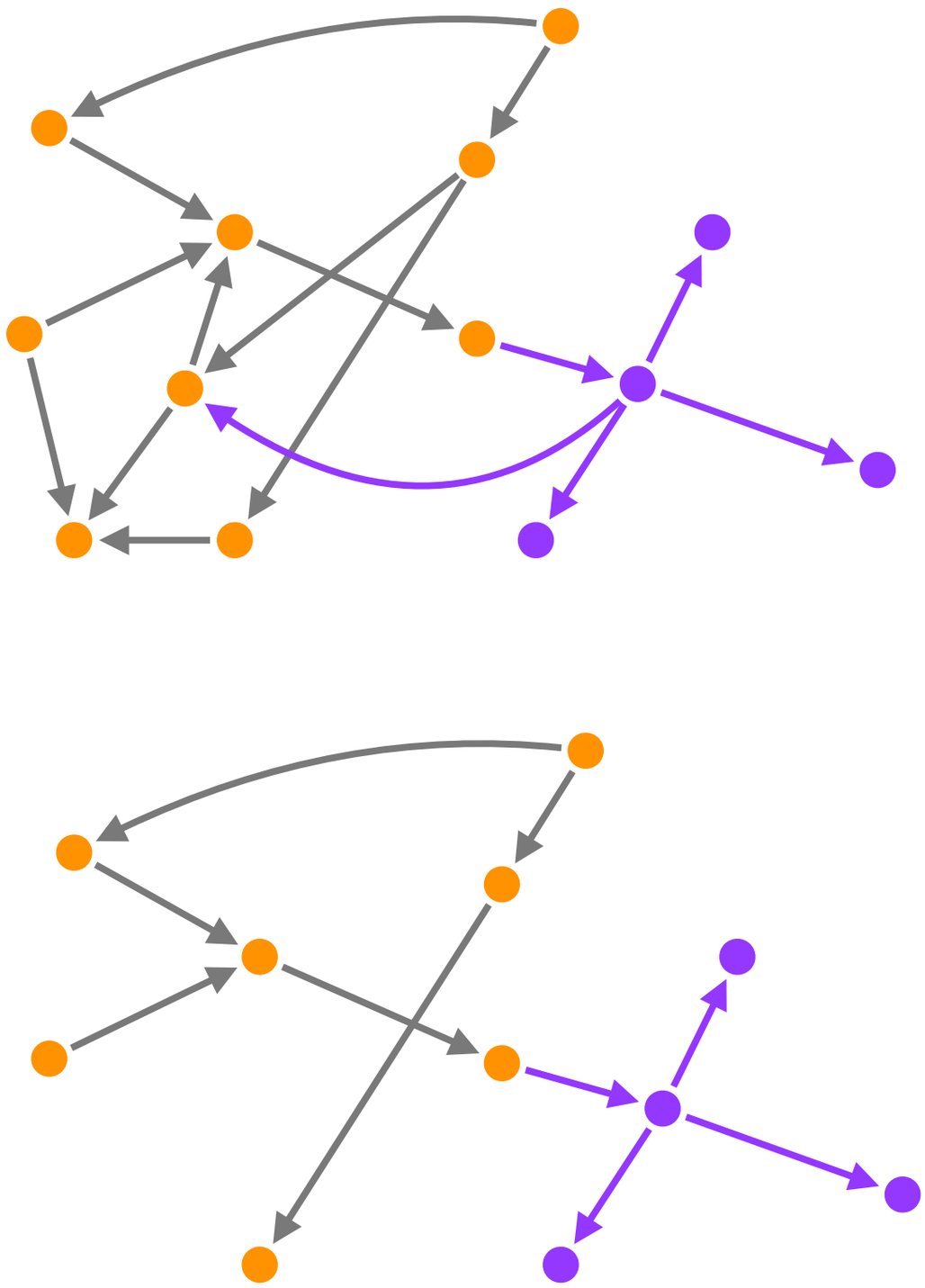
- Unique outcome of concurrent updates

No consensus: ≤ n-1 faults

Fast, responsive

Strong Eventual Consistency

- Available, responsive
- More parallelism
- No conflicts
- No rollback



Update local + propagate

- No synchronization
- Update is durable
- Broadcast

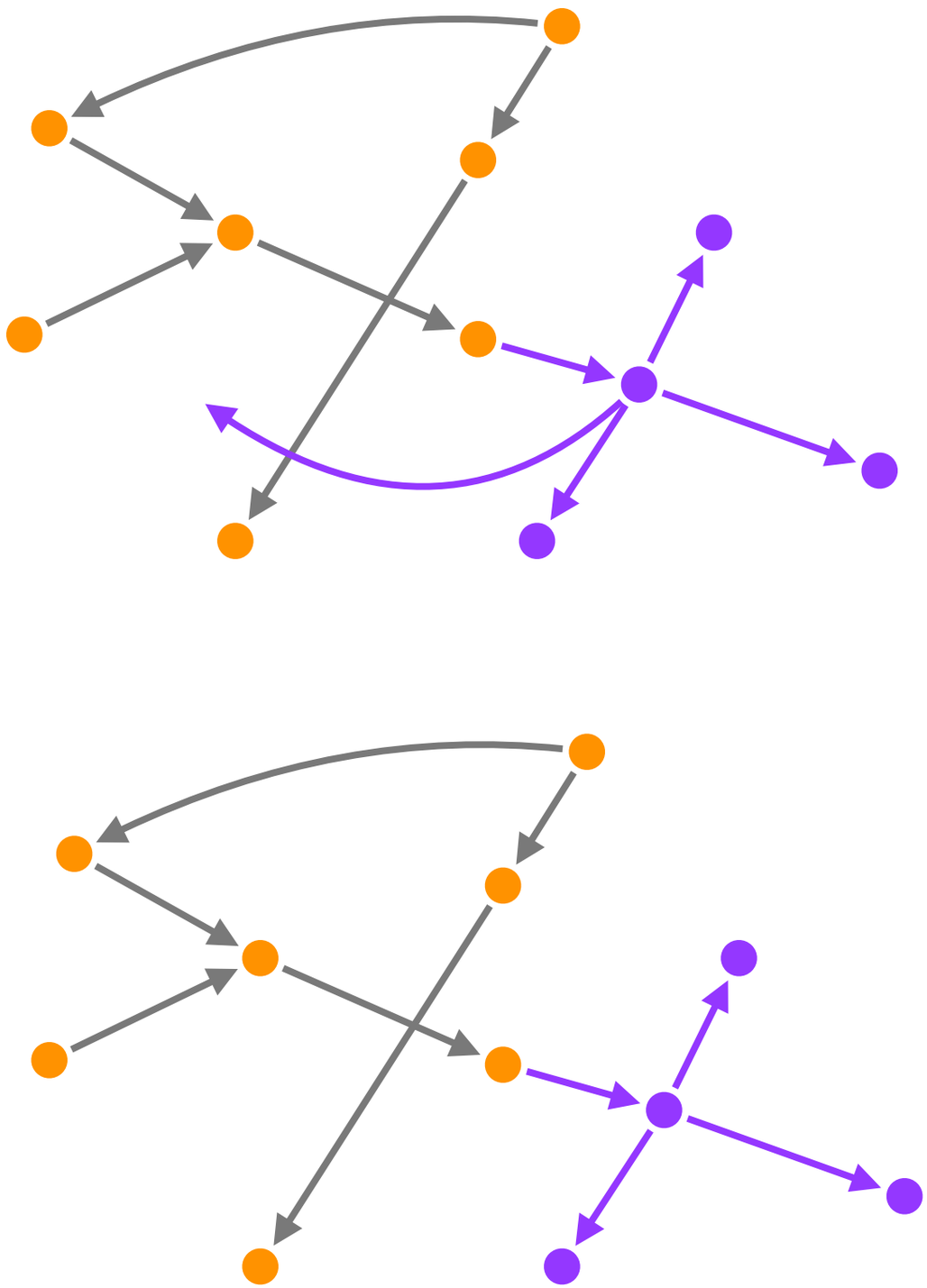
No conflict

- Unique outcome of concurrent updates

No consensus: ≤ n-1 faults

Fast, responsive

Strong Eventual Consistency



- Available, responsive
- More parallelism
- No conflicts
- No rollback

Update local + propagate

- No synchronization
- Update is durable
- Broadcast

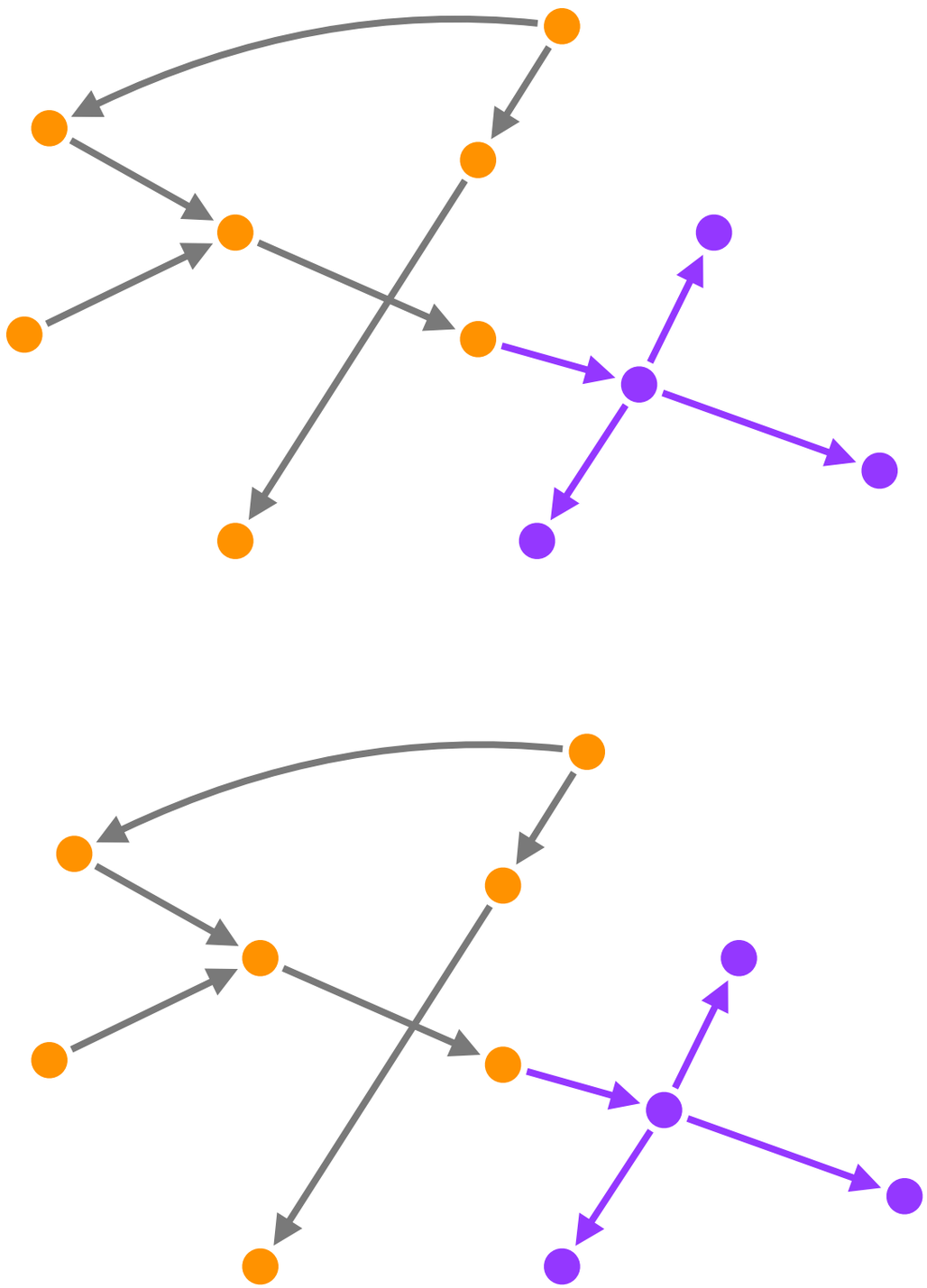
No conflict

- Unique outcome of concurrent updates

No consensus: ≤ n-1 faults

Fast, responsive

Strong Eventual Consistency



- Available, responsive
- More parallelism
- No conflicts
- No rollback

Update local + propagate

- No synchronization
- Update is durable
- Broadcast

No conflict

- Unique outcome of concurrent updates

No consensus: ≤ n-1 faults

Fast, responsive

Strong Eventual Consistency

Eventual delivery:

Every update eventually executes at all correct replicas.

Termination:

Every update terminates.

***Strong** Convergence:*

Correct replicas that have executed the same updates have equivalent state.

Strong Eventual Consistency

Eventual delivery:

Every update eventually executes at all correct replicas.

Termination:

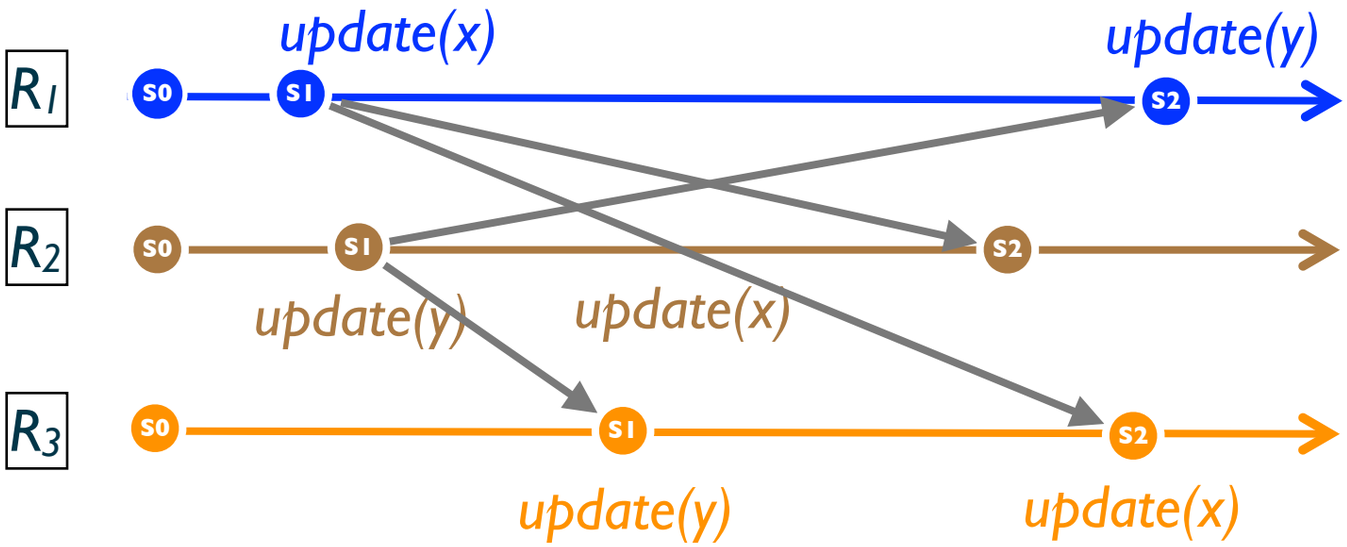
Every update terminates.

Strong *Convergence:*

Correct replicas that have executed the same updates have equivalent state.

Deterministic, context-independent outcome to concurrent updates

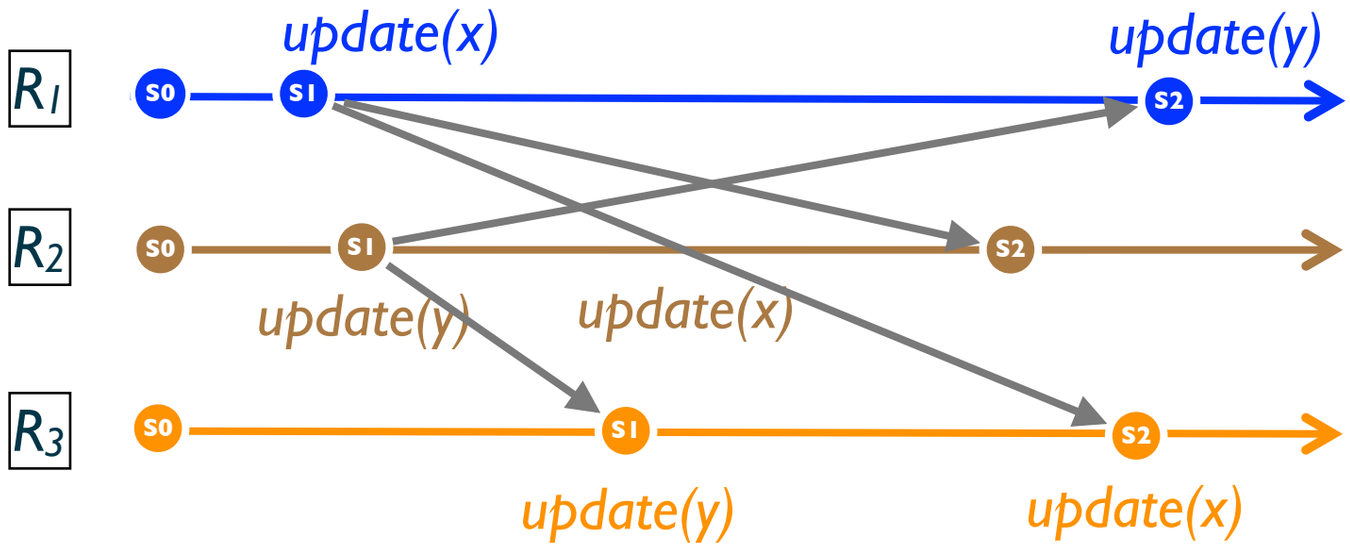
Operation-based updates



- Small messages, no information duplication
- Uses **causal broadcast**
  - Vector clock counts messages received / node
  - Size of vector clock ~ number of replicas
- Consensus not required

Causal broadcast
- small messages
- includes past, never goes back in time
- VC issues

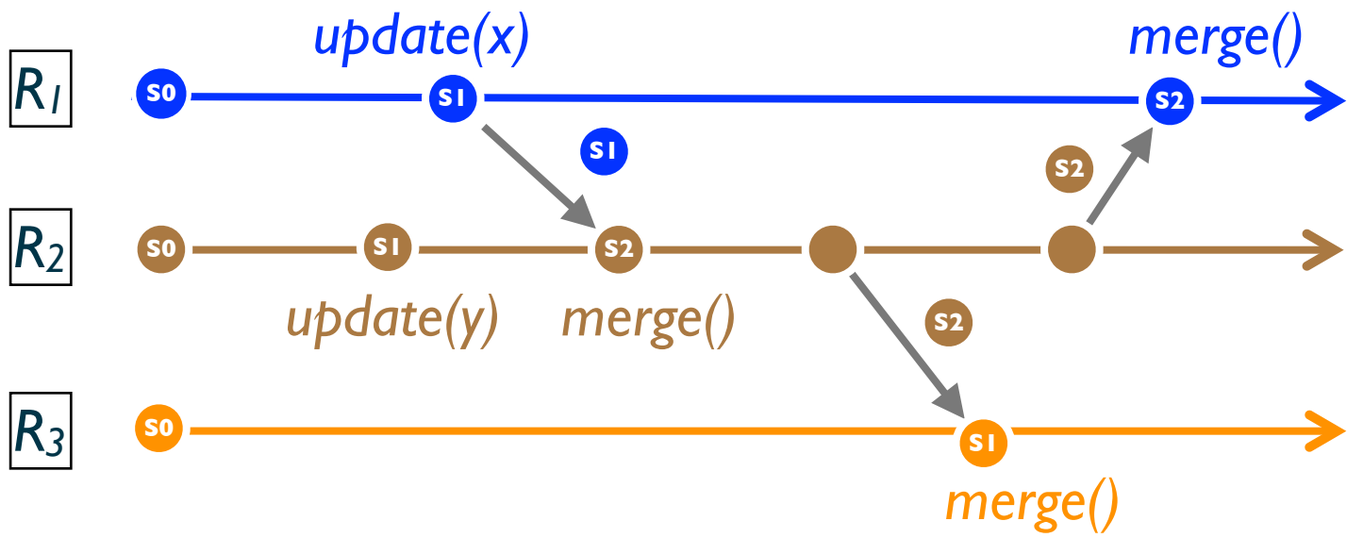
Operation-based CRDTs



- Example: Counter with incr and decr
- All replicas have equivalent state in the end
- Sufficient condition:
  - Reliable causal delivery ⇒ Vector clocks
  - Concurrent operations **commute**

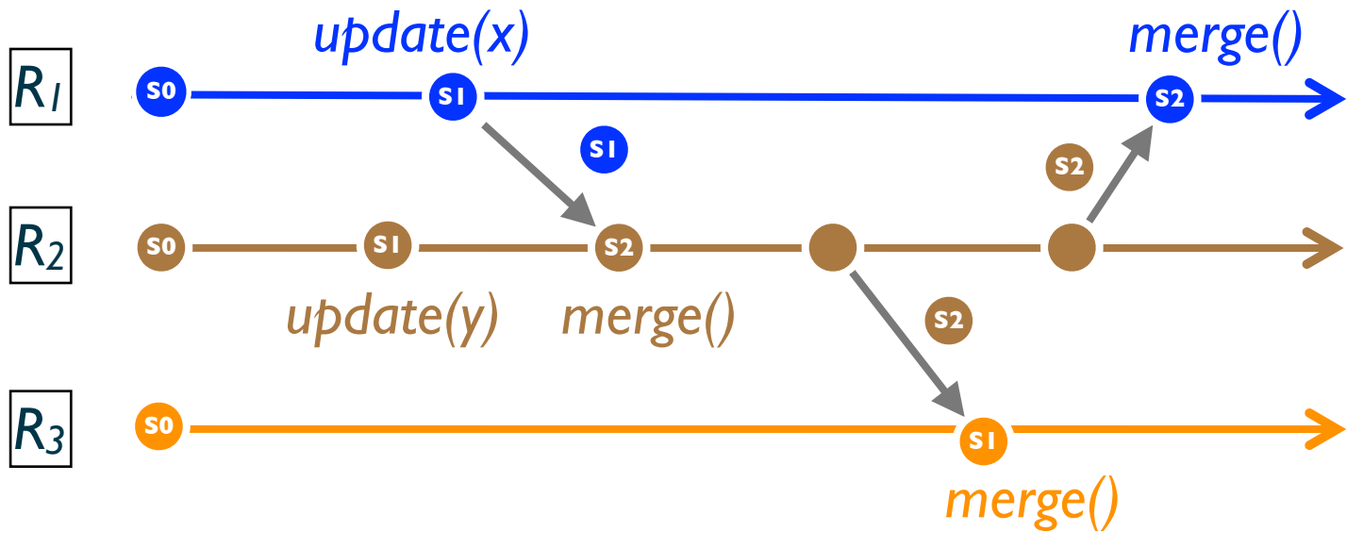
Causal broadcast
- small messages
- includes past, never goes back in time
- VC issues

State-based / data shipping



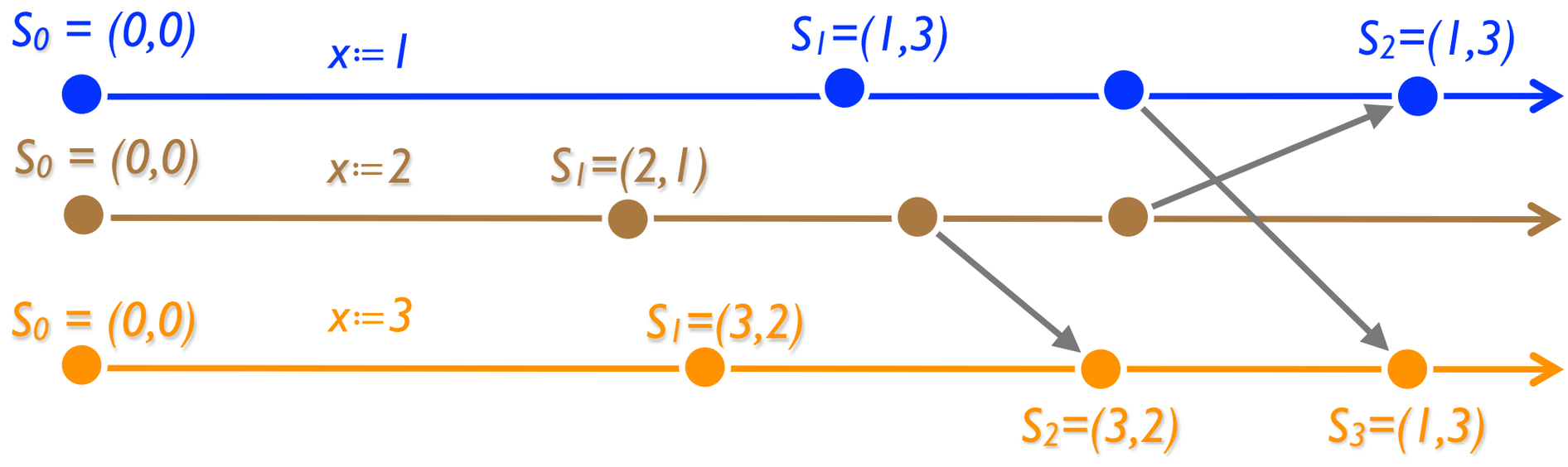
- **Epidemic propagation**: Flooding
- Eventual delivery
- Consensus not required
- Inefficient for large payload
- Convergence?

State-based CRDTs



- All replicas have equivalent state in the end
- Sufficient condition: ***Monotonic semi-lattice***
  - Partial order
  - Monotonic
  - *merge* computes Least Upper Bound
  - *merge* eventually delivered

Last-Writer-Wins Register



Payload	$S \stackrel{\text{def}}{=} (\textit{value } v, \textit{timestamp } ts)$
Update	$S \bullet [x := v] \stackrel{\text{def}}{=} (v, ts{+}{+})$
Merge	$S \bullet \textit{merge}(S') \stackrel{\text{def}}{=} S.ts < S'.ts \ ? \ S' : S$
Compare	$S \leq S' \stackrel{\text{def}}{=} S.ts \leq S'.ts$

Grow-only Set

Payload	$S \stackrel{\text{def}}{=} \{ e_0, ..., e_n \}$
Update	$S \bullet add(e) \stackrel{\text{def}}{=} S \cup \{ e \}$
Lookup	$S \bullet lookup(e) \stackrel{\text{def}}{=} e \in S$
Merge	$S \bullet merge(S') \stackrel{\text{def}}{=} S \cup S'$
Compare	$S \leq S' \stackrel{\text{def}}{=} S \subseteq S'$

2P-Set

Payload	$S \stackrel{\text{def}}{=} (A, R)$
Update	$S \bullet add(e) \stackrel{\text{def}}{=} (A \cup \{ e \}, R)$
	$S \bullet rmv(e) \stackrel{\text{def}}{=} (A, R \cup \{ e \})$
Lookup	$S \bullet lookup(e) \stackrel{\text{def}}{=} e \in A \wedge e \notin R$
Merge	$S \bullet merge(S') \stackrel{\text{def}}{=} (A \cup A', R \cup R')$
Compare	$S \leq S' \stackrel{\text{def}}{=} A \subseteq A' \wedge R \subseteq R'$

Conflict-free Replicated Data Types (CRDTs)

Register
- Last-Writer Wins
- Multi-Value

Set
- Grow-Only
- 2P
- Observed-Remove

Map

Counter
- Unlimited
- Non-negative

Graph
- Directed
- Monotonic DAG
- Edit graph

Sequence

Designing a CRDT

Sequential specification of Set:

- {*true*} add(e) {e ∈ *S*}
- {*true*} rmv(e) {e ∉ *S*}

Sequentially non ambiguous (e ≠ *f*):

- {*true*} add(e) || add(e) {e ∈ *S*}
- {*true*} rmv(e) || rmv(e) {e ∉ *S*}
- {*true*} add(e) || add(*f*) {e,*f* ∈ *S*}
- {*true*} rmv(e) || rmv(*f*) {e, *f* ∉ *S*}
- {*true*} add(e) || rmv(*f*) {e ∈ *S*, *f* ∉ *S*}

Ambiguous:

- {*true*} add(e) || rmv(e) {????}

- Principle of Non-Ambiguous Permutation Equivalence

Design alternatives for add(e) || rem(e)

- linearisable: requires consensus

Design alternatives for add(e) || rem(e)

{*true*} add(e) || rmv(e) {*????*}

- linearisable: requires consensus

Design alternatives for add(e) || rem(e)

{*true*} add(e) || rmv(e) {*????*}

- linearisable: requires consensus

Design alternatives for add(e) || rem(e)

{*true*} add(e) || rmv(e) {*????*}

- ~~linearisable?~~

- linearisable: requires consensus

Design alternatives for add(e) || rem(e)

{*true*} add(e) || rmv(e) {*????*}

- ~~linearisable?~~
- last writer wins? { add(e) < rmv(e) ⇒ e ∉ S
  ∧ rmv(e) < add(e) ⇒ e ∈ S }

- linearisable: requires consensus

Design alternatives for add(e) || rem(e)

{*true*} add(e) || rmv(e) {*????*}

- linearisable: requires consensus

- ~~linearisable?~~
- last writer wins? { add(e) < rmv(e) ⇒ e ∉ *S*
  ∧ rmv(e) < add(e) ⇒ e ∈ *S* }
- error state? {⊥e ∈ *S*}

Design alternatives for add(e) || rem(e)

{*true*} add(e) || rmv(e) {*????*}

- ~~linearisable?~~
- last writer wins? { add(e) < rmv(e) ⇒ e ∉ *S* ∧ rmv(e) < add(e) ⇒ e ∈ *S* }
- error state? {⊥e ∈ *S*}
- add wins? {e ∈ *S*}

- linearisable: requires consensus

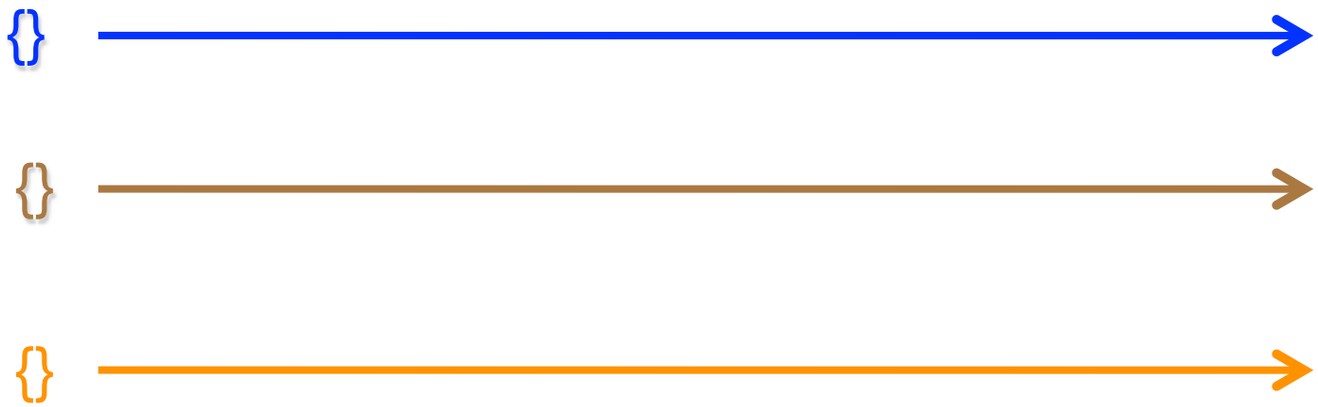
Design alternatives for add(e) || rem(e)

{*true*} add(e) || rmv(e) {*????*}

- linearisable: requires consensus

- ~~linearisable?~~
- last writer wins? { add(e) < rmv(e) ⇒ e ∉ S ∧ rmv(e) < add(e) ⇒ e ∈ S }
- error state? {$\perp_e \in S$}
- add wins? {e ∈ S}
- remove wins? {e ∉ S}

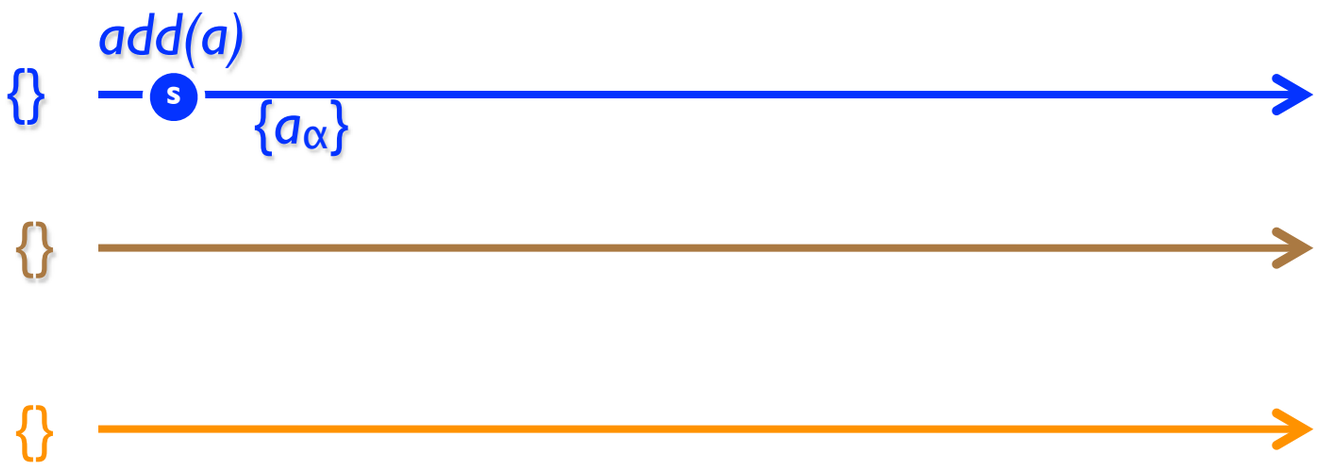
Observed-Remove Set



- Timestamps assumed unique
- Can never remove more tokens than exist
- Op order ⇒ removed

Payload	S	$\stackrel{\text{def}}{=} (A = \{(e, uid), \ldots\},\ R = \{(e', uid'), \ldots\})$
Update	$S \bullet add(e)$	$\stackrel{\text{def}}{=} (A \cup \{ (e,uid) \}, R)$
	$S \bullet rmv(e)$	$\stackrel{\text{def}}{=} (A \setminus T, R \cup T)$ *with* $T = \{ (e, \_) \in A \}$
Lookup	$S \bullet lookup(e)$	$\stackrel{\text{def}}{=} e \in A$
Merge	$S \bullet merge(S')$	$\stackrel{\text{def}}{=} (A \setminus R' \cup A' \setminus R, R \cup R')$
Compare	$S \leq S'$	$\stackrel{\text{def}}{=} A \cup R \subseteq A' \cup R' \wedge R \subseteq R'$

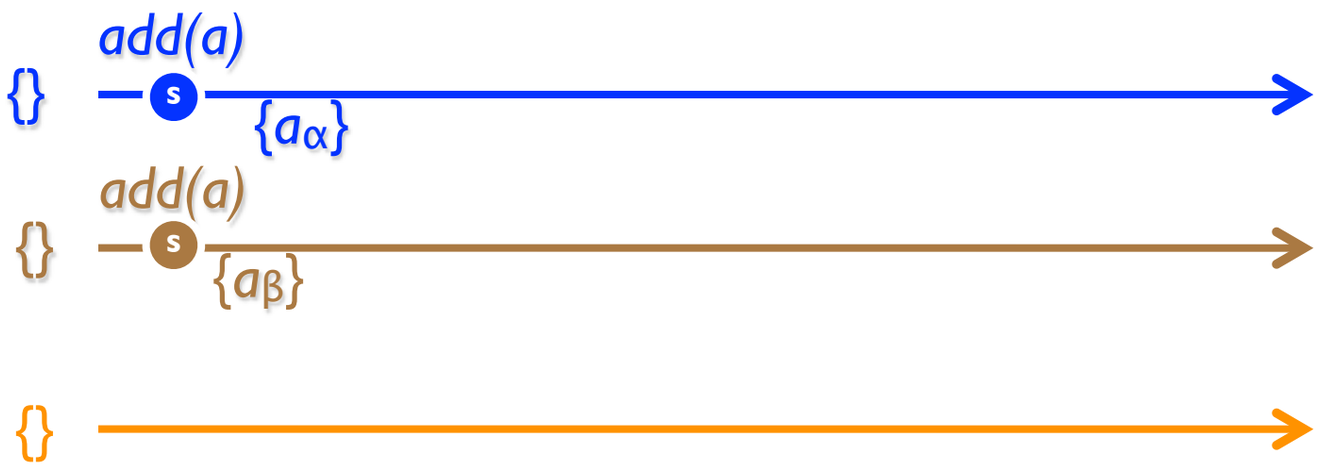
Observed-Remove Set



- Timestamps assumed unique
- Can never remove more tokens than exist
- Op order ⇒ removed

Payload	S	$\stackrel{\text{def}}{=} (A = \{(e, uid), \ldots\},\ R = \{(e', uid'), \ldots\})$
Update	$S \bullet add(e)$	$\stackrel{\text{def}}{=} (A \cup \{ (e,uid) \}, R)$
	$S \bullet rmv(e)$	$\stackrel{\text{def}}{=} (A \setminus T, R \cup T)$ with $T = \{ (e, \_) \in A \}$
Lookup	$S \bullet lookup(e)$	$\stackrel{\text{def}}{=} e \in A$
Merge	$S \bullet merge(S')$	$\stackrel{\text{def}}{=} (A \setminus R' \cup A' \setminus R, R \cup R')$
Compare	$S \leq S'$	$\stackrel{\text{def}}{=} A \cup R \subseteq A' \cup R' \wedge R \subseteq R'$

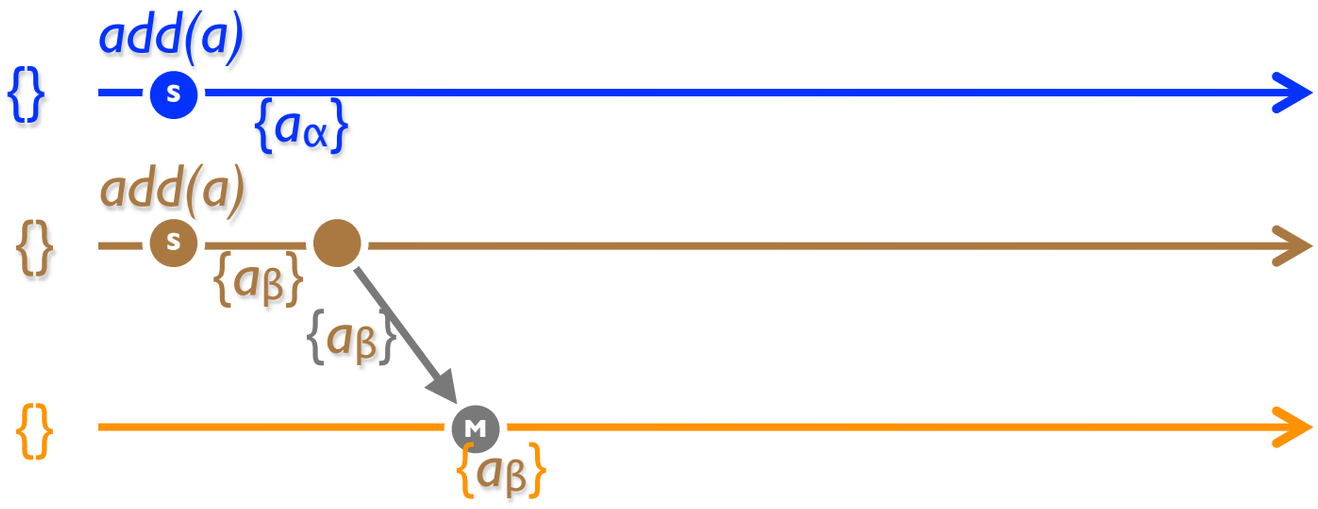
Observed-Remove Set



- Timestamps assumed unique
- Can never remove more tokens than exist
- Op order ⇒ removed

Payload	S	$\stackrel{\text{def}}{=} (A = \{(e, uid), \ldots\},\ R = \{(e', uid'), \ldots\})$
Update	$S \bullet add(e)$	$\stackrel{\text{def}}{=} (A \cup \{ (e,uid) \}, R)$
	$S \bullet rmv(e)$	$\stackrel{\text{def}}{=} (A \setminus T, R \cup T)$ with $T = \{ (e, \_) \in A \}$
Lookup	$S \bullet lookup(e)$	$\stackrel{\text{def}}{=} e \in A$
Merge	$S \bullet merge(S')$	$\stackrel{\text{def}}{=} (A \setminus R' \cup A' \setminus R, R \cup R')$
Compare	$S \leq S'$	$\stackrel{\text{def}}{=} A \cup R \subseteq A' \cup R' \wedge R \subseteq R'$

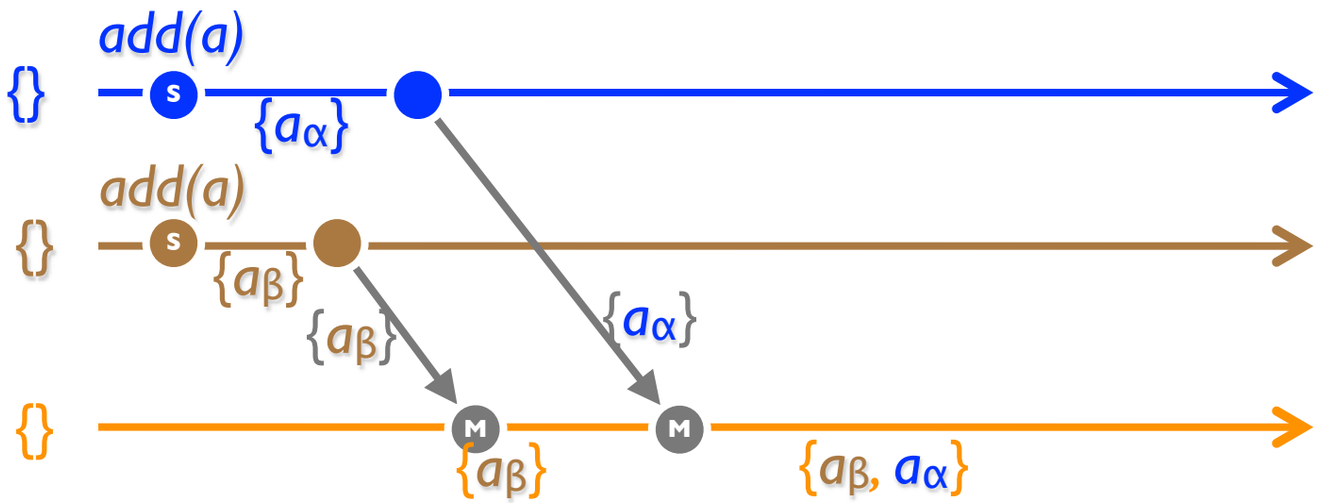
Observed-Remove Set



- Timestamps assumed unique
- Can never remove more tokens than exist
- Op order ⇒ removed

Payload	S	$\stackrel{\text{def}}{=} (A = \{(e, uid), \ldots\},\ R = \{(e', uid'), \ldots\})$
Update	$S \bullet add(e)$	$\stackrel{\text{def}}{=} (A \cup \{ (e,uid) \}, R)$
	$S \bullet rmv(e)$	$\stackrel{\text{def}}{=} (A \setminus T, R \cup T)$ with $T = \{ (e, \_) \in A \}$
Lookup	$S \bullet lookup(e)$	$\stackrel{\text{def}}{=} e \in A$
Merge	$S \bullet merge(S')$	$\stackrel{\text{def}}{=} (A \setminus R' \cup A' \setminus R, R \cup R')$
Compare	$S \leq S'$	$\stackrel{\text{def}}{=} A \cup R \subseteq A' \cup R' \wedge R \subseteq R'$

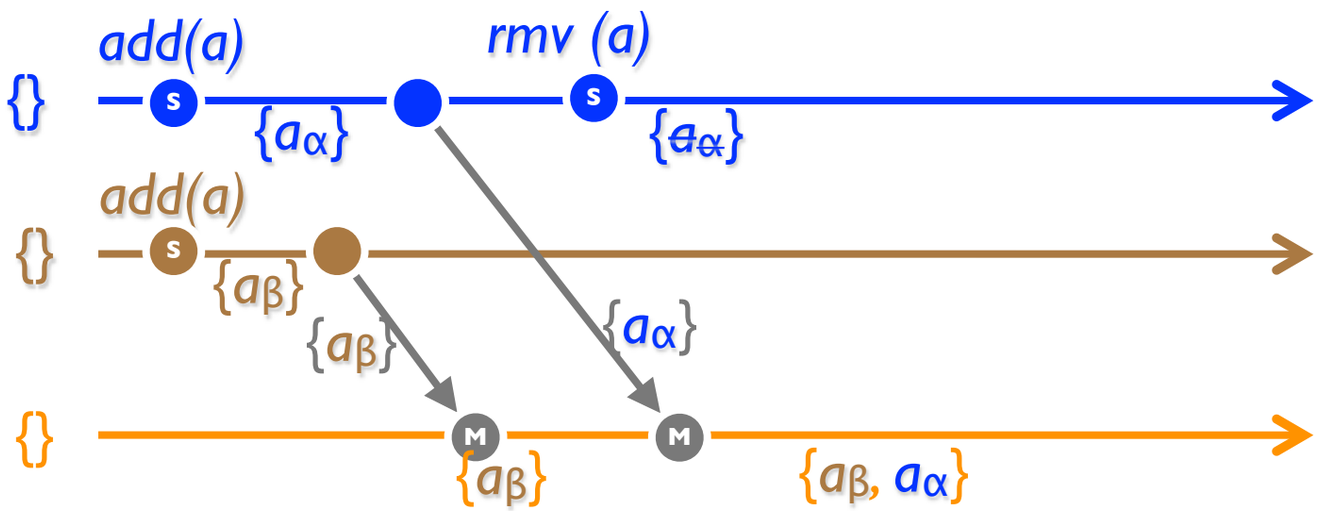
Observed-Remove Set



- Timestamps assumed unique
- Can never remove more tokens than exist
- Op order ⇒ removed

Payload	S	$\stackrel{\text{def}}{=} (A = \{(e, uid), \ldots\},\ R = \{(e', uid'), \ldots\})$
Update	$S \bullet add(e)$	$\stackrel{\text{def}}{=} (A \cup \{ (e,uid) \}, R)$
	$S \bullet rmv(e)$	$\stackrel{\text{def}}{=} (A \setminus T, R \cup T)$ with $T = \{ (e, \_) \in A \}$
Lookup	$S \bullet lookup(e)$	$\stackrel{\text{def}}{=} e \in A$
Merge	$S \bullet merge(S')$	$\stackrel{\text{def}}{=} (A \setminus R' \cup A' \setminus R, R \cup R')$
Compare	$S \leq S'$	$\stackrel{\text{def}}{=} A \cup R \subseteq A' \cup R' \wedge R \subseteq R'$

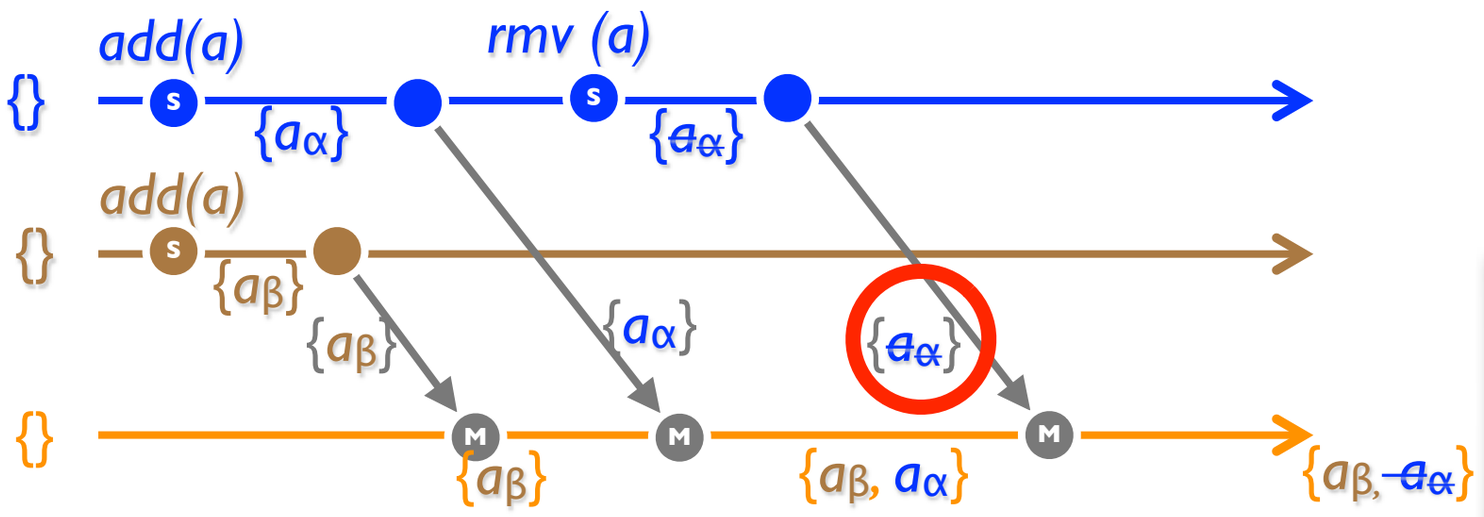
Observed-Remove Set



- Timestamps assumed unique
- Can never remove more tokens than exist
- Op order ⇒ removed

Payload	S	$\stackrel{\text{def}}{=} (A = \{(e, uid), \ldots\},\ R = \{(e', uid'), \ldots\})$
Update	$S \bullet add(e)$	$\stackrel{\text{def}}{=} (A \cup \{ (e,uid) \}, R)$
	$S \bullet rmv(e)$	$\stackrel{\text{def}}{=} (A \setminus T, R \cup T)$ with $T = \{ (e, \_) \in A \}$
Lookup	$S \bullet lookup(e)$	$\stackrel{\text{def}}{=} e \in A$
Merge	$S \bullet merge(S')$	$\stackrel{\text{def}}{=} (A \setminus R' \cup A' \setminus R, R \cup R')$
Compare	$S \leq S'$	$\stackrel{\text{def}}{=} A \cup R \subseteq A' \cup R' \wedge R \subseteq R'$

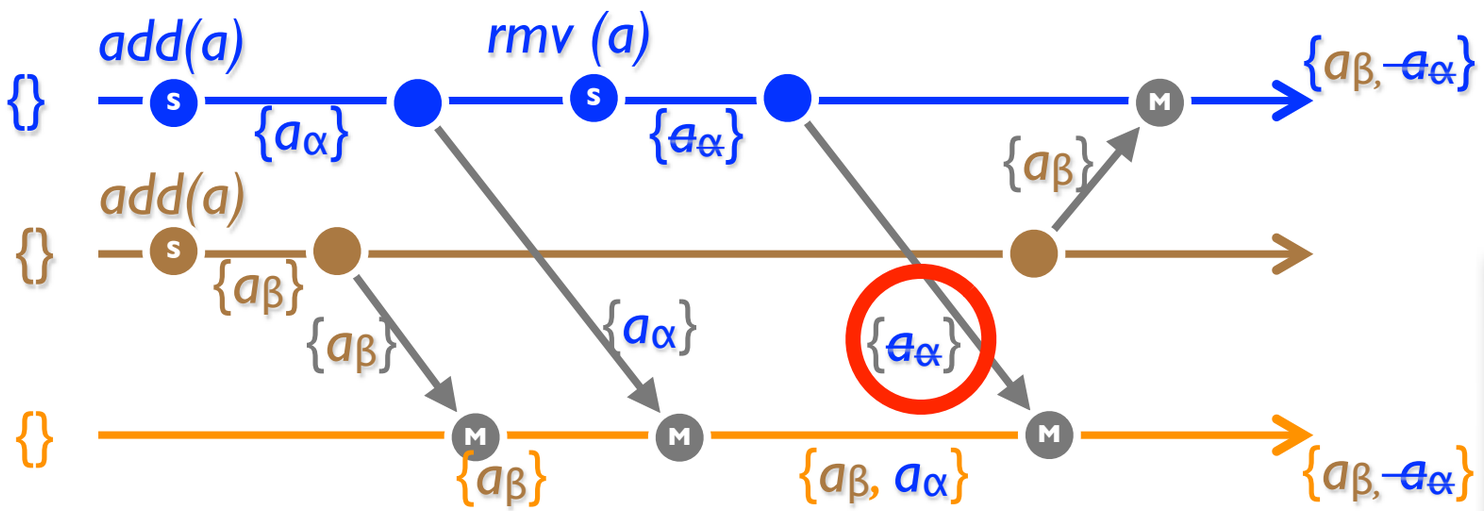
Observed-Remove Set



- Timestamps assumed unique
- Can never remove more tokens than exist
- Op order ⇒ removed

Payload	S	$\stackrel{\text{def}}{=} (A = \{(e, uid), \ldots\},\ R = \{(e', uid'), \ldots\})$
Update	$S \bullet add(e)$	$\stackrel{\text{def}}{=} (A \cup \{ (e,uid) \}, R)$
	$S \bullet rmv(e)$	$\stackrel{\text{def}}{=} (A \setminus T, R \cup T)$ with $T = \{ (e, \_) \in A \}$
Lookup	$S \bullet lookup(e)$	$\stackrel{\text{def}}{=} e \in A$
Merge	$S \bullet merge(S')$	$\stackrel{\text{def}}{=} (A \setminus R' \cup A' \setminus R, R \cup R')$
Compare	$S \leq S'$	$\stackrel{\text{def}}{=} A \cup R \subseteq A' \cup R' \wedge R \subseteq R'$

Observed-Remove Set



- Timestamps assumed unique
- Can never remove more tokens than exist
- Op order ⇒ removed

Payload	S	$\stackrel{\text{def}}{=} (A = \{(e, uid), \ldots\},\ R = \{(e', uid'), \ldots\})$
Update	$S \bullet add(e)$	$\stackrel{\text{def}}{=} (A \cup \{ (e,uid) \}, R)$
	$S \bullet rmv(e)$	$\stackrel{\text{def}}{=} (A \setminus T, R \cup T)$ with $T = \{ (e, \_) \in A \}$
Lookup	$S \bullet lookup(e)$	$\stackrel{\text{def}}{=} e \in A$
Merge	$S \bullet merge(S')$	$\stackrel{\text{def}}{=} (A \setminus R' \cup A' \setminus R, R \cup R')$
Compare	$S \leq S'$	$\stackrel{\text{def}}{=} A \cup R \subseteq A' \cup R' \wedge R \subseteq R'$

Summary: CRDT

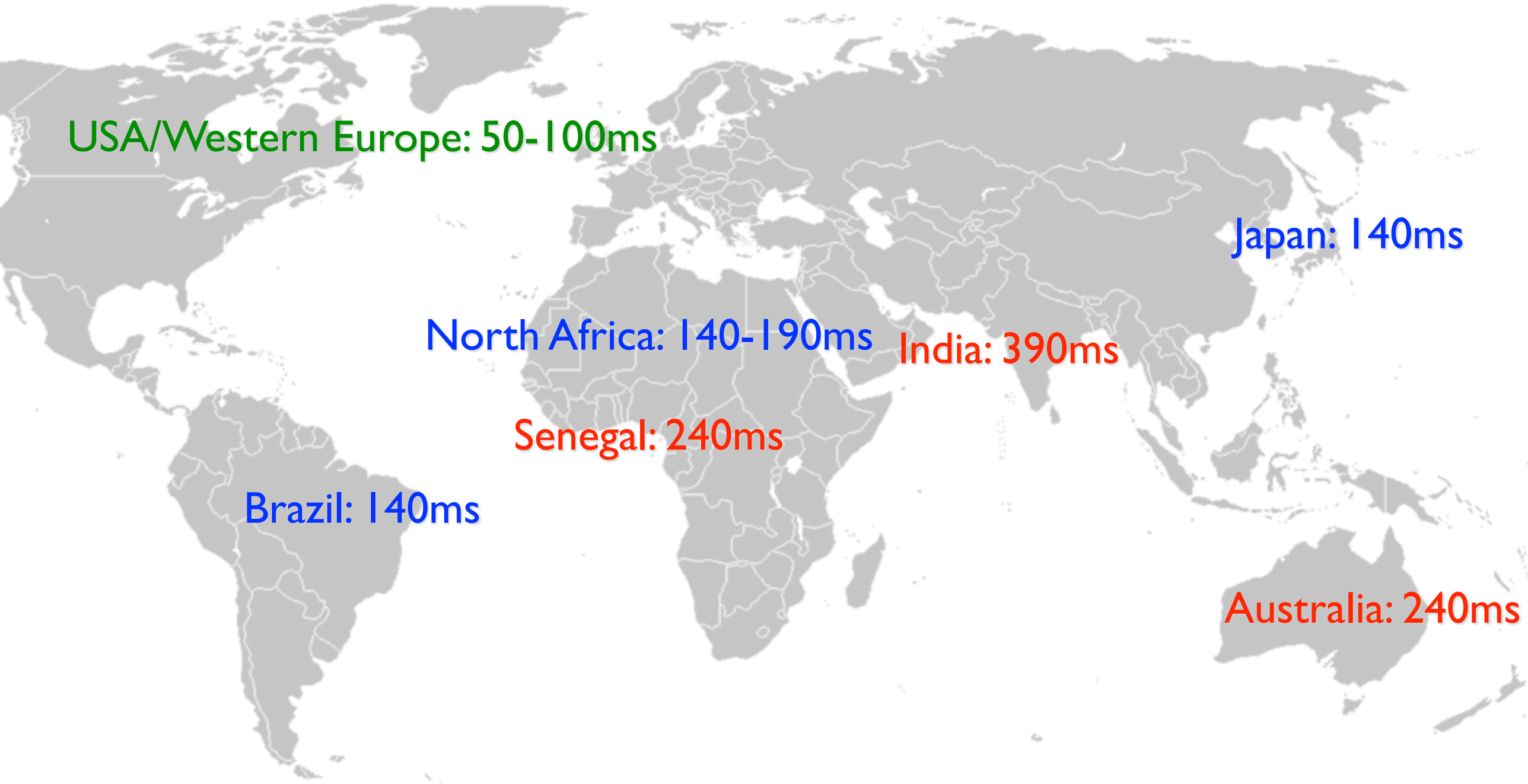
- Concurrent updates have deterministic outcome
- Sufficient conditions:
  - State-based: epidemic, monotonic semi-lattice
  - Op-based: causal, concurrent ⇒ commute

- CRDTs
  - don‘t lose updates
  - converge eventually
  - have durable updates, no rollbacks
  - support unlimited (crash-recovery) failures

SwiftCloud: Geo-replication all the way to the edge

Study by Jay *et al.[`07]:*

„*I*nteractive response times > 50ms are annoying to users“



Closer to the client!

- Less latency
- Improved availability
- Support for disconnected operation
- Decreased network traffic by notification
- Current trend in Ajax and HTML 5

Scalability objectives

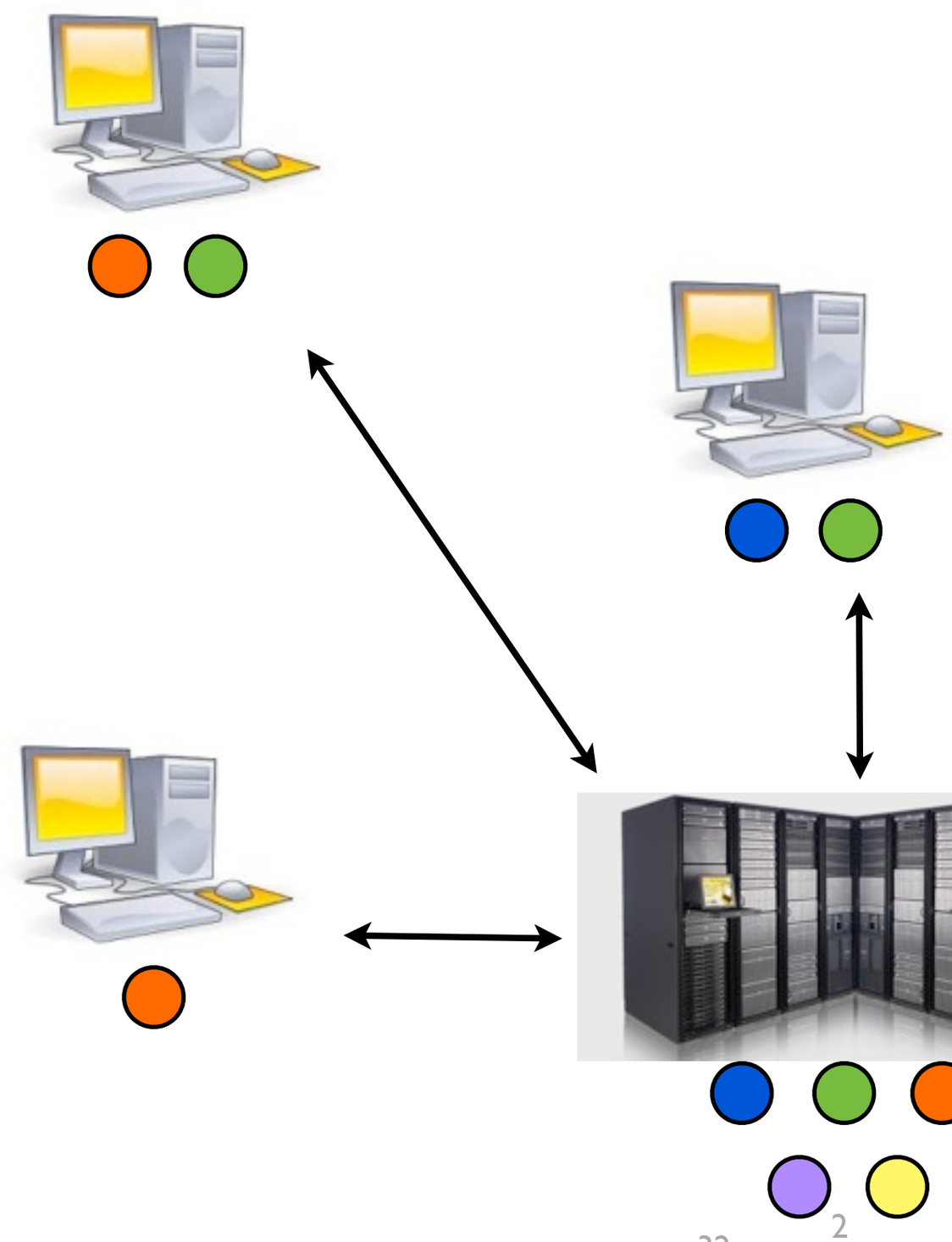
100–1000 data centres

» 10^6 client-side replicas

Fast updates, high rates

WAN latency

Network failures



SwiftCloud

Large-scale CRDT store

- Library of CRDT types
- Application can use composition of CRDTs
- CRDT transactions
- Tiered architecture:
  - Core: ≈100 data centres
  - Nebula: > 106–108 scouts
- Commit & persistence in core
  ⇒ small version vectors

- Geo-Replication:
- Failure independence
- abstractly, 1 RDC == 1 crash-free server

Core

Core

Core

Sequencers

Data nodes

Storage nodes

Surrogate nodes

Scouts & clients

Conflict-free transaction

- Multiple updates, multiple objects
- All commute ⇒ commit always succeeds
- Asynchronous
- ACID (from the perspective of any replica)
- Configurations:
  - *Strictly Most Recent / Cached mode*
  - *Conflict-Free Snapshot Isolation / Repeatable Reads*
- Multi-version CRDTs + causally-consistent snapshot
- Send updates in a single packet

Example: SwiftSocial

Similar to WaltSocial [Sovran et al. (SOSP'11)]

High-level operations modelled as transactions

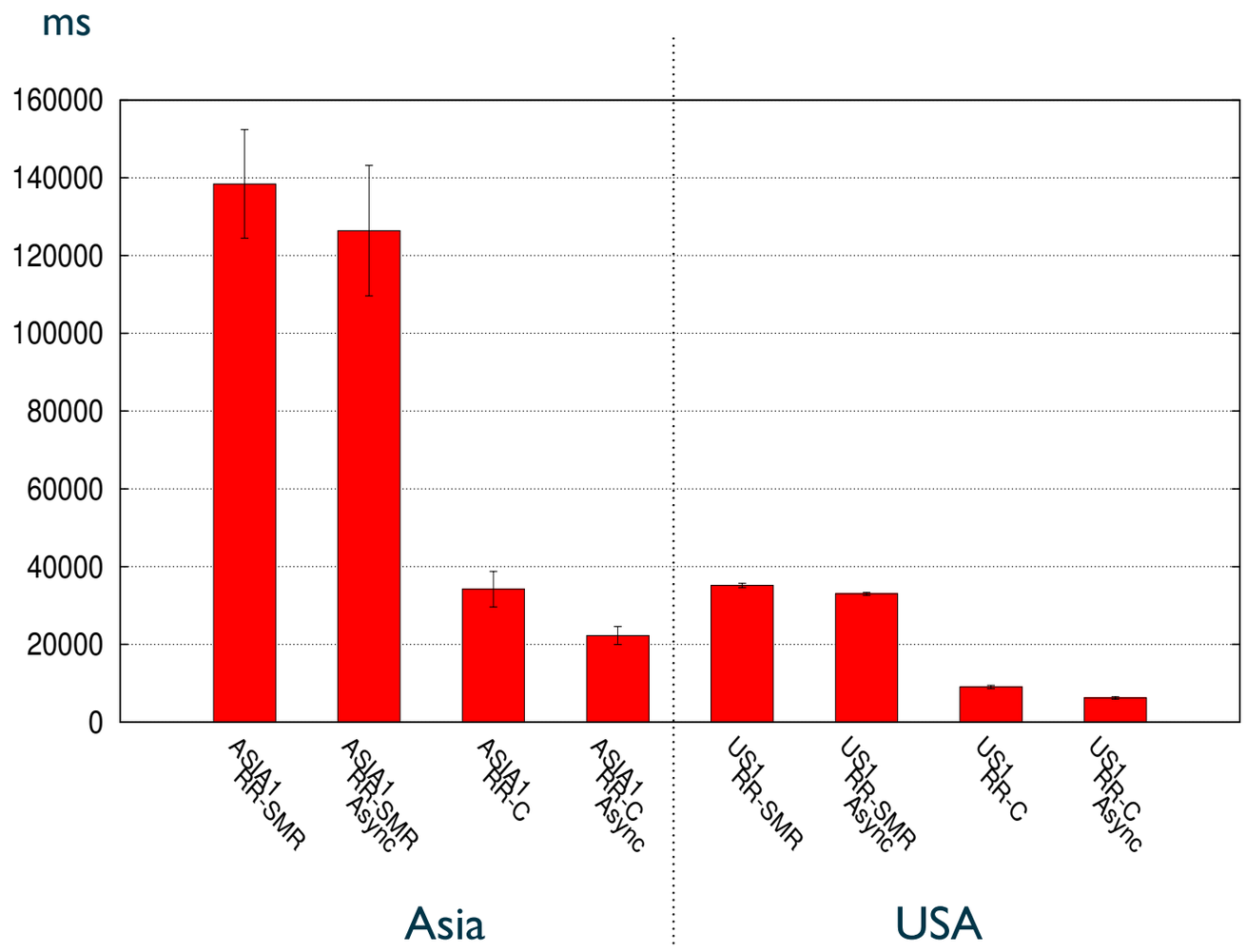
- Registering user, Login/Logout
- Post status update
- Send message
- View wall
- Friendship management

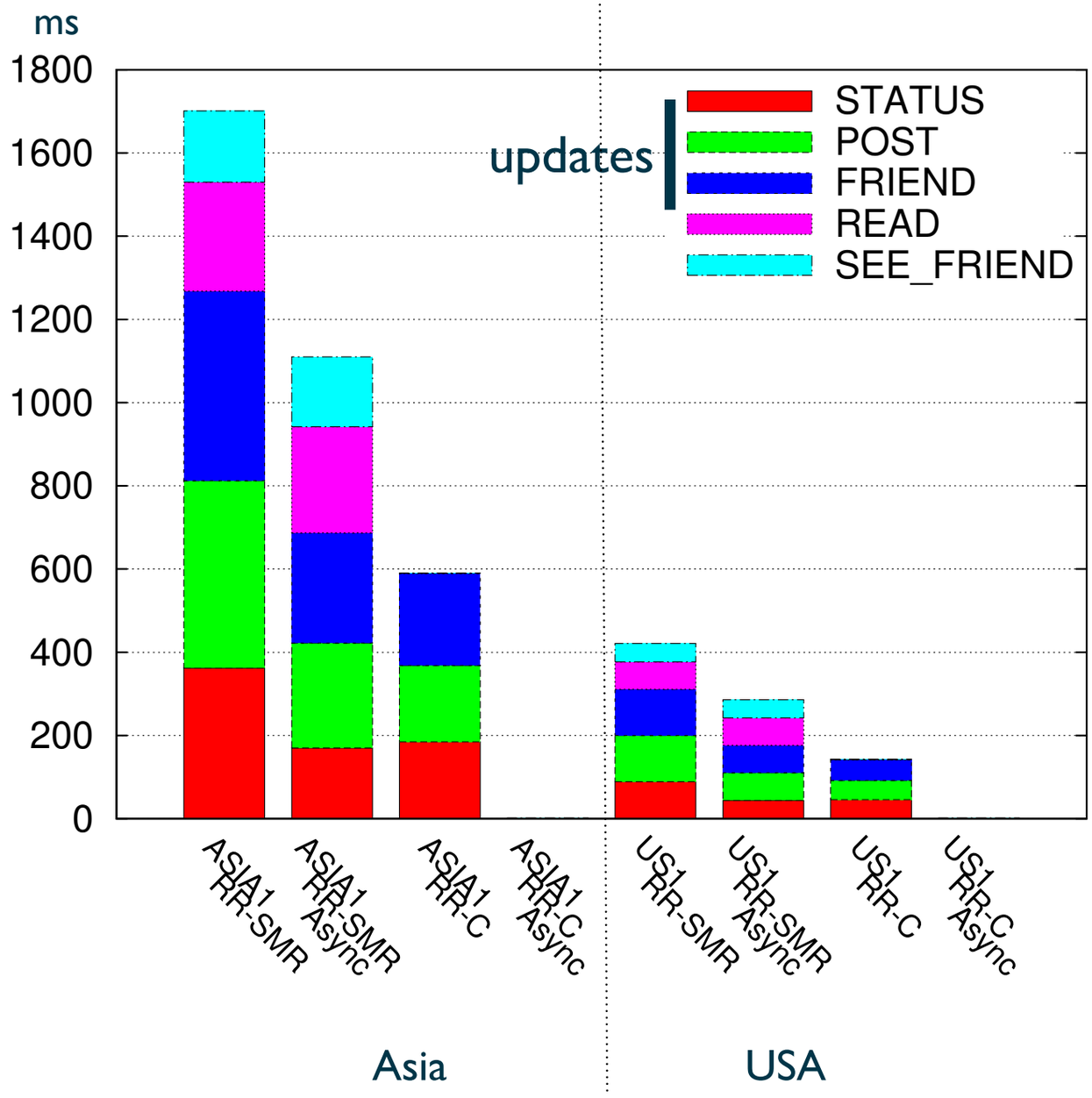
Set CRDT for messages and friends

Register CRDT for user data

(Counter CRDT for polls - not implemented yet)

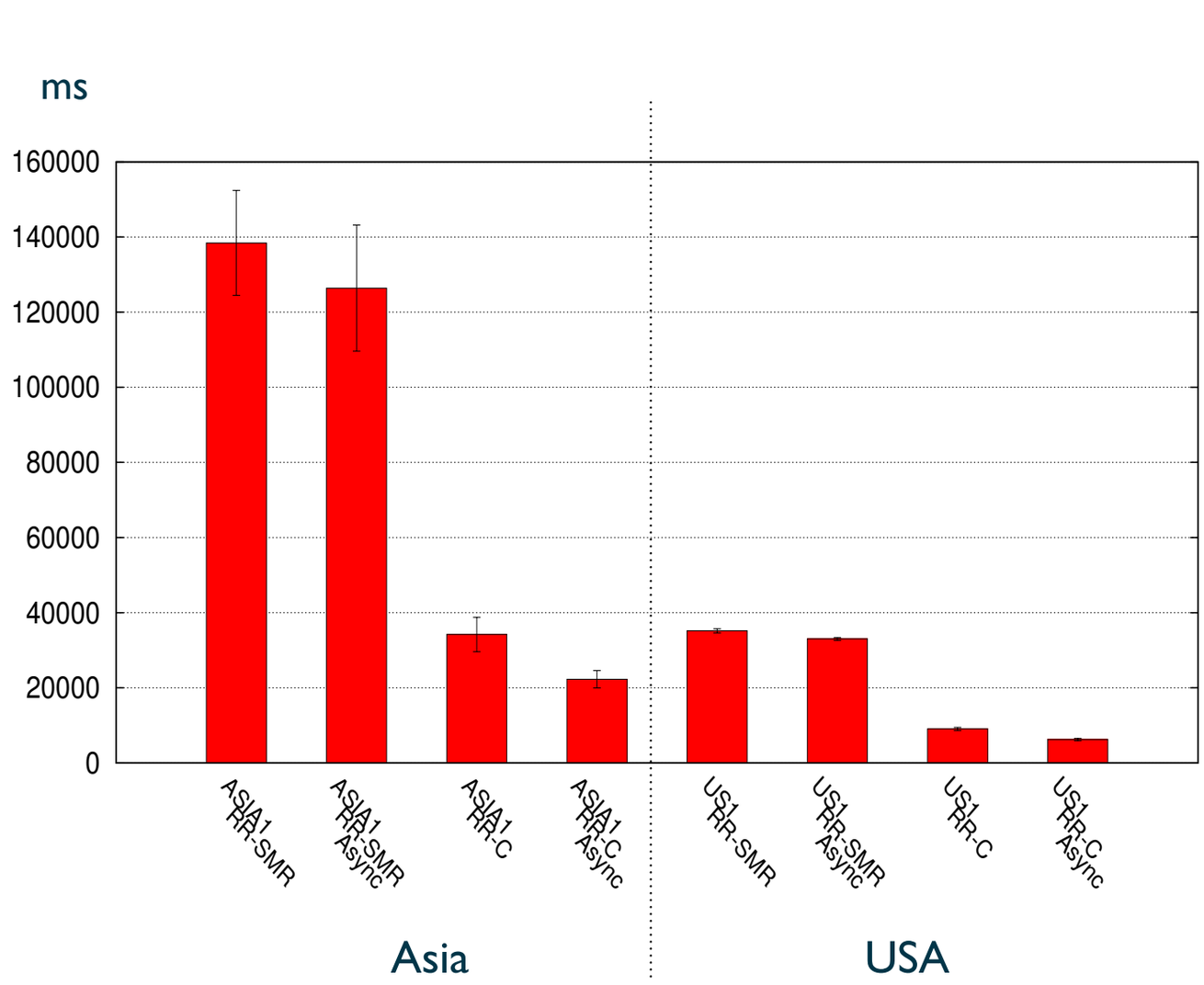
CRDT transactions vs. synchronous

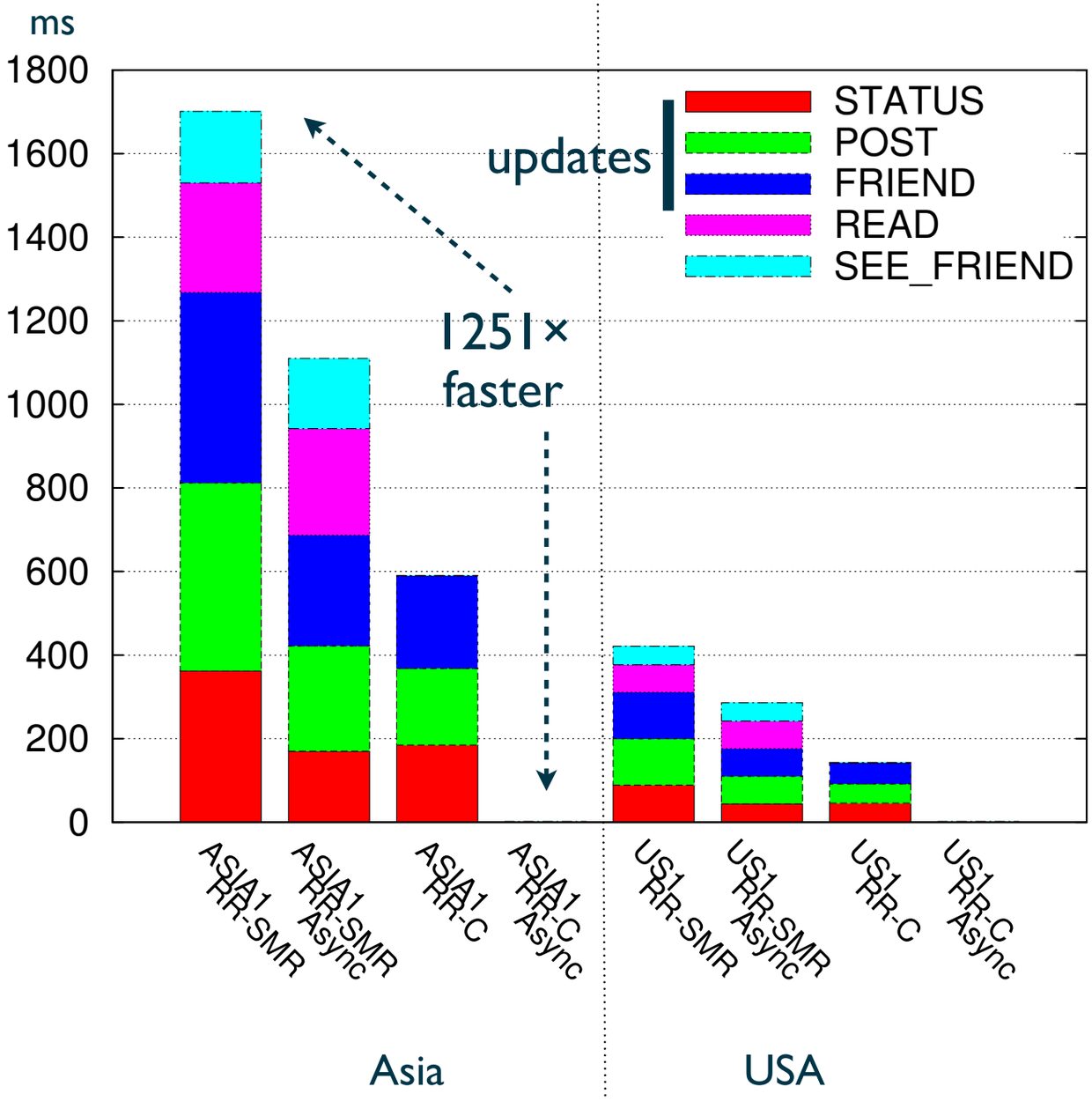




5000 users, each user has 50 associated friends
500 txns, 1s waiting time (not included in figure)
90% of txns on user‘s and friends‘ data
90% read-only txns

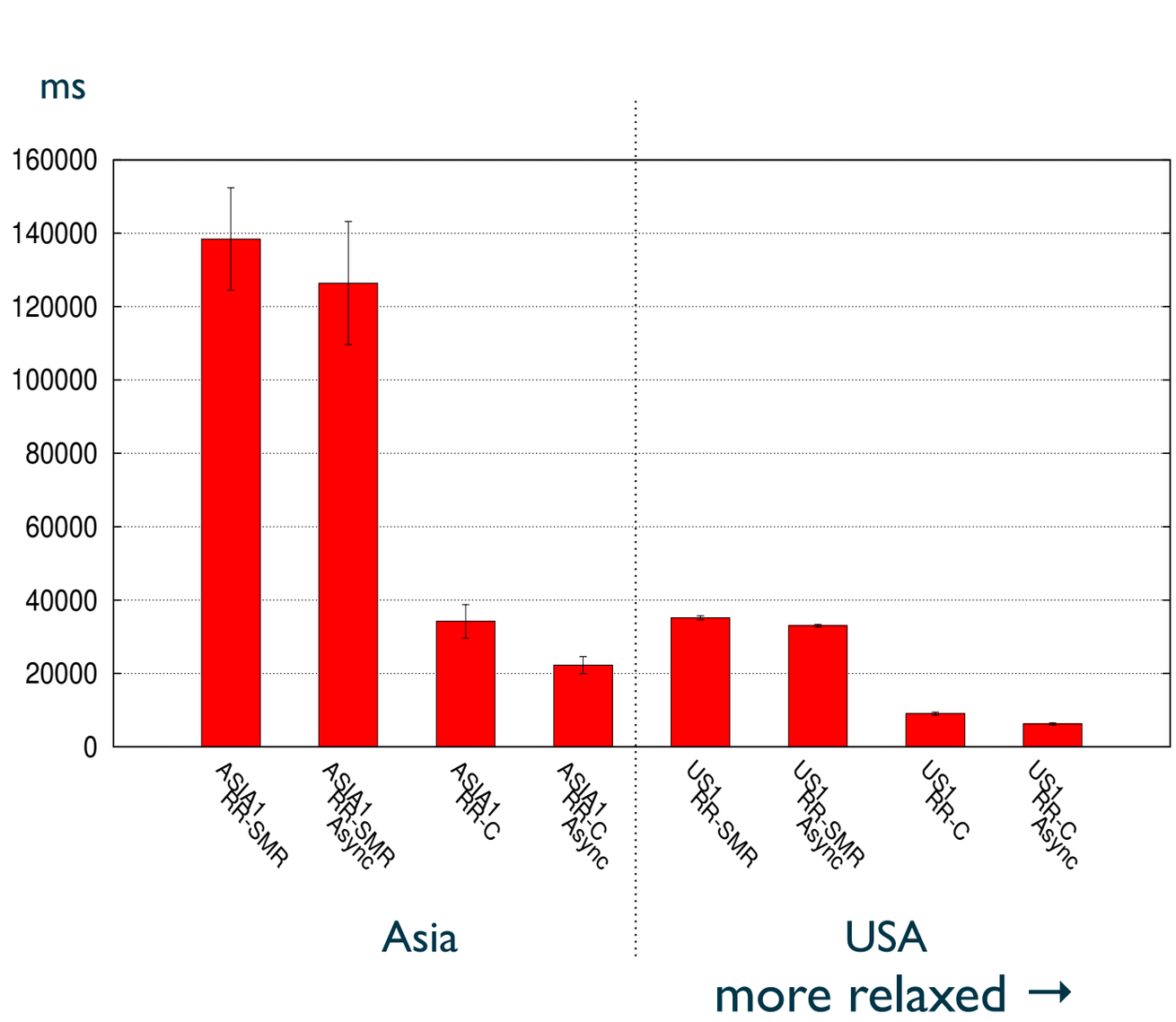
CRDT transactions vs. synchronous

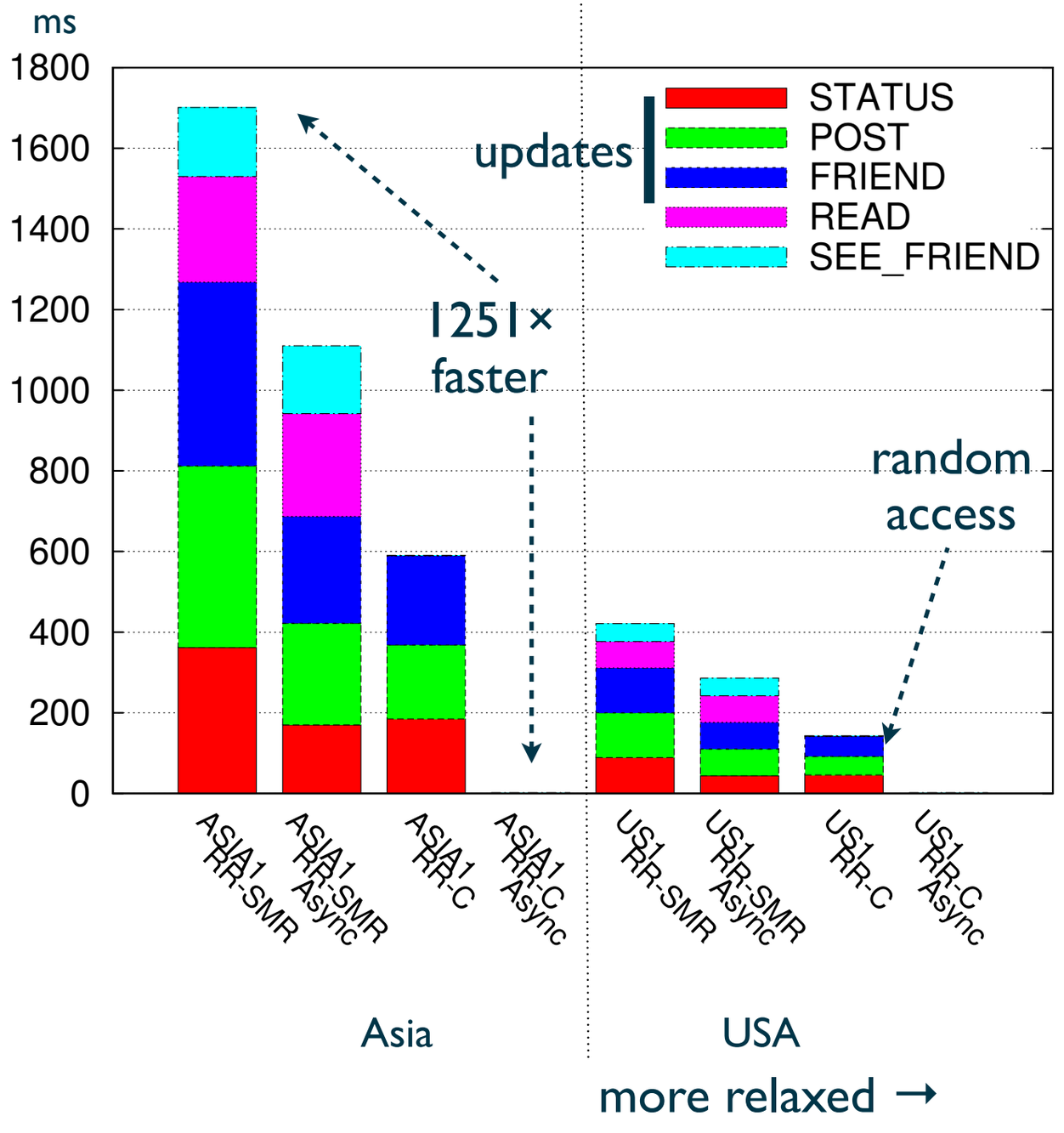




5000 users, each user has 50 associated friends
500 txns, 1s waiting time (not included in figure)
90% of txns on user‘s and friends‘ data
90% read-only txns

CRDT transactions vs. synchronous

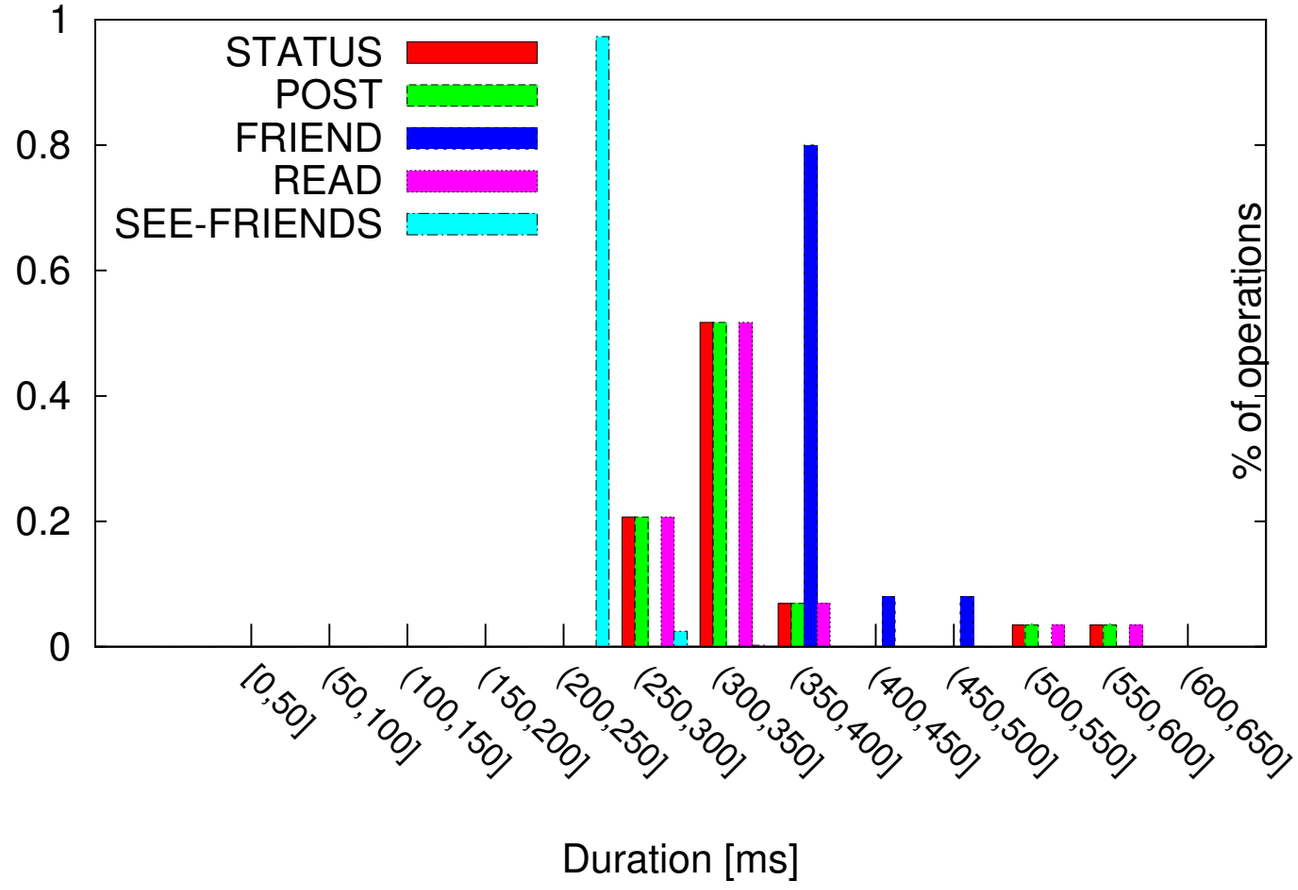




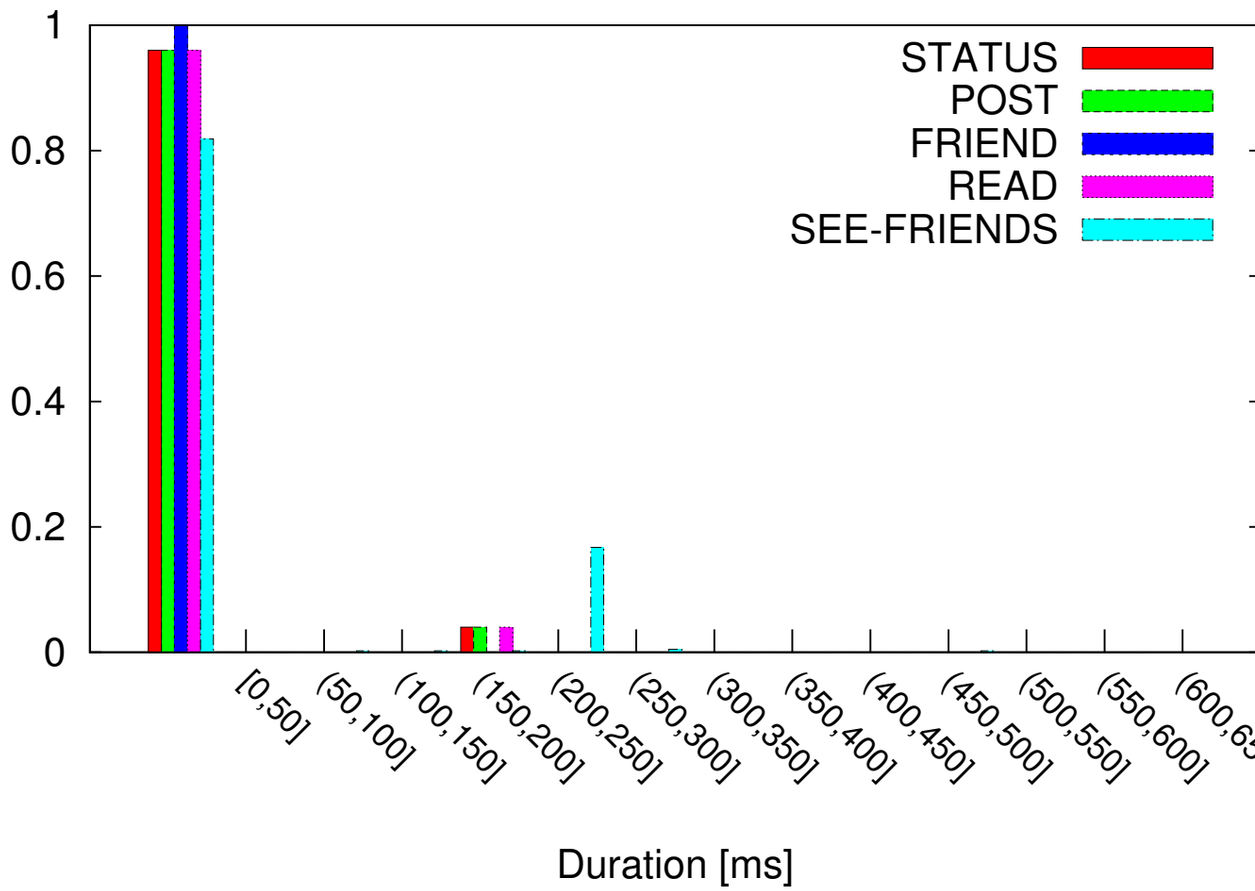
5000 users, each user has 50 associated friends
500 txns, 1s waiting time (not included in figure)
90% of txns on user‘s and friends‘ data
90% read-only txns

Caching for read *and* updates

Strictly-Most-Recent



Cached



Distribution for one user session

Distributed file system

- Based on recursive Directory CRDT
- Represented as map: (name, type) → object
- Operations:
  - *create_entry(n, t, v)*
    - Concurrent: merge subdirs recursively
  - *modify_file(n, t, v)*
    - Concurrent: merge file content
  - *remove_entry(n,t)*
    - Concurrent: deletion dominates
    - Changes can be retrieved from history

Summary: Swiftcloud

- Large-scale replication of mutable shared data structures
- Updates at the edge of the network
- Conflict-free transactions
- Efficient caching for improved availability and responsiveness

Open questions

What is expressible with CRDTs and transactions?

- Invariants on CRDT objects?
- Adding strong synchronization?

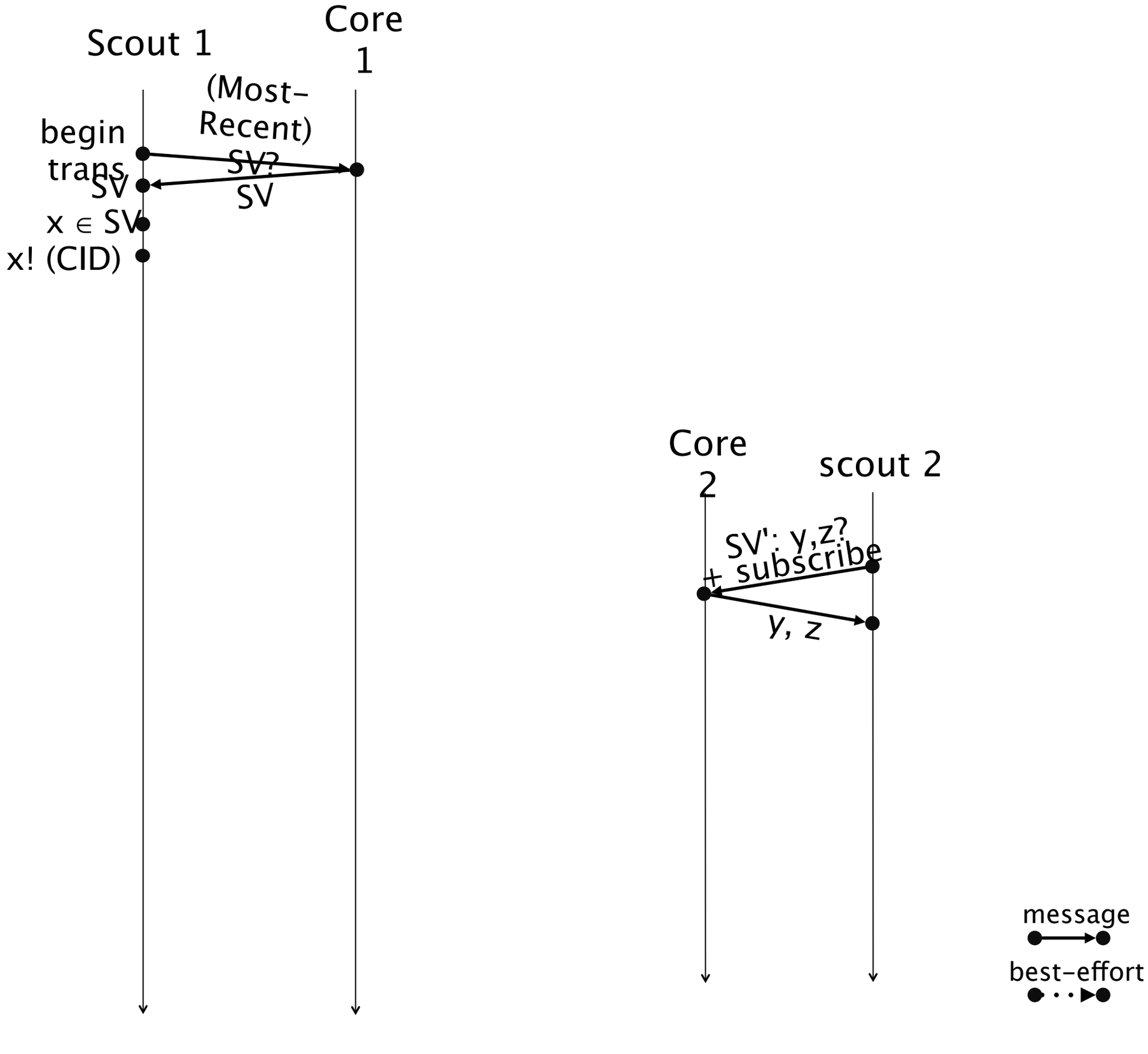
How can we make CRDTs secure?

- Re-introduce encapsulation and modularity
- Abstraction from internal state

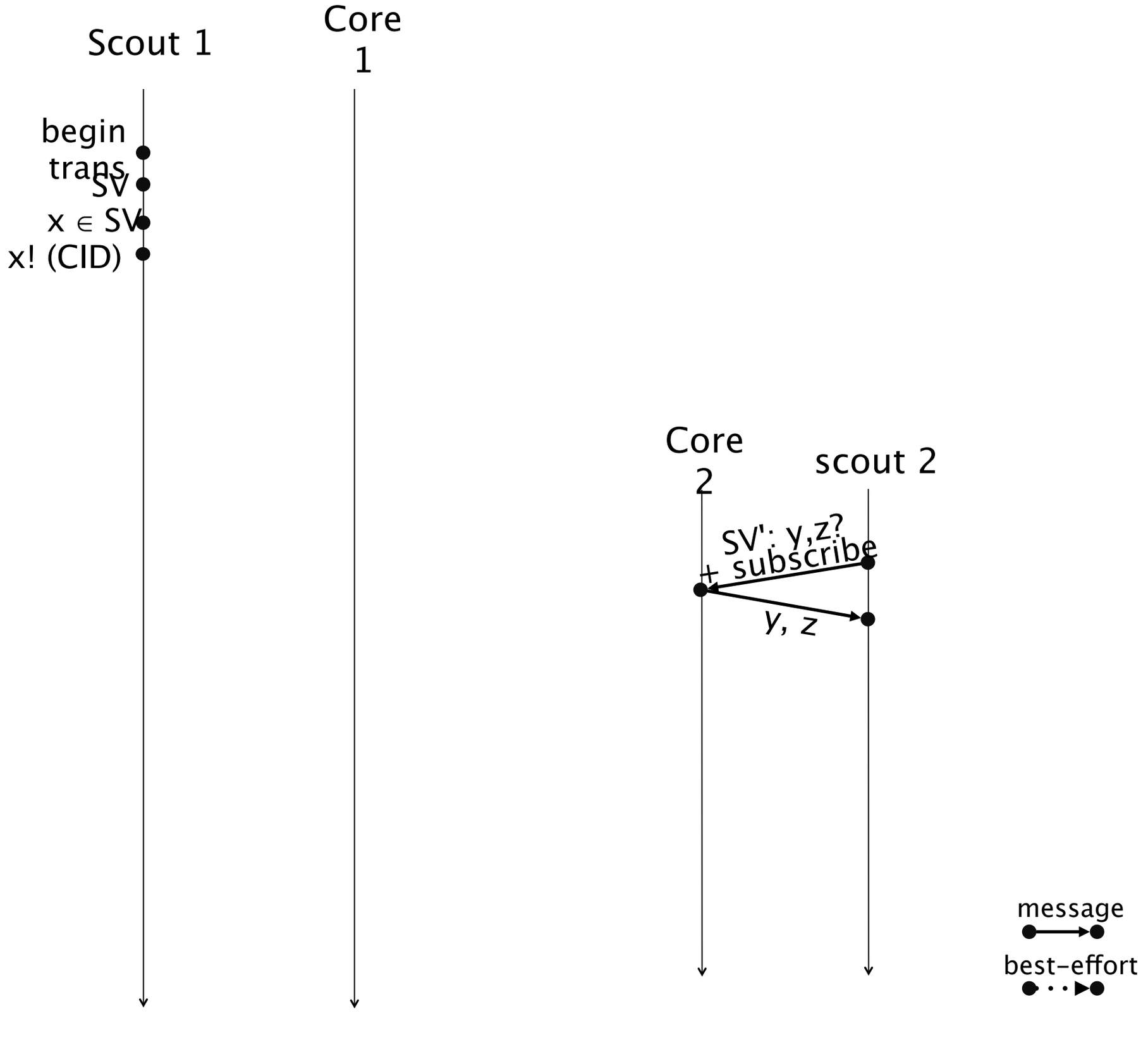
How can we optimize the platform?

- Pruning of object state
- Improved causaility tracking

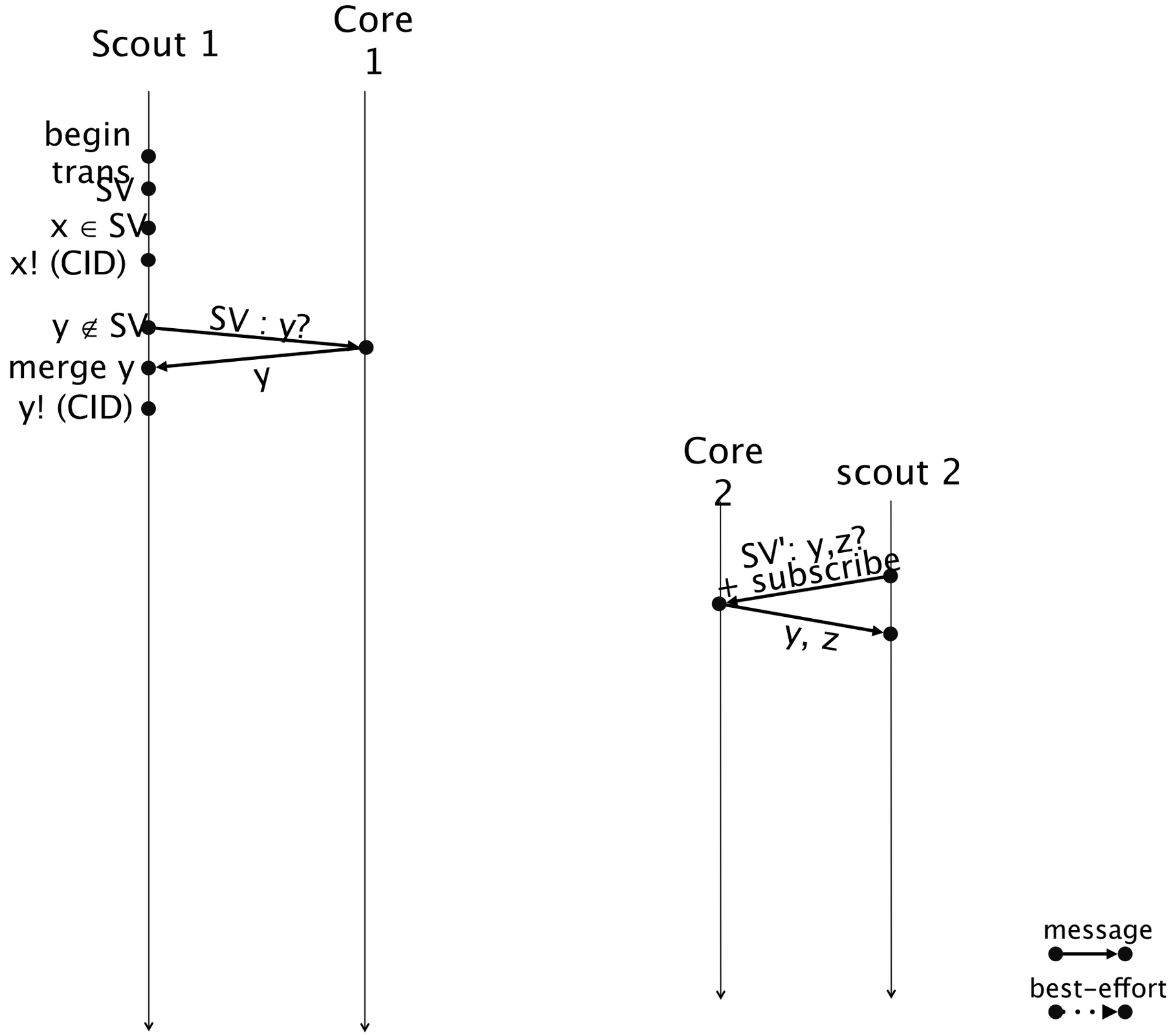
Transaction hand-over



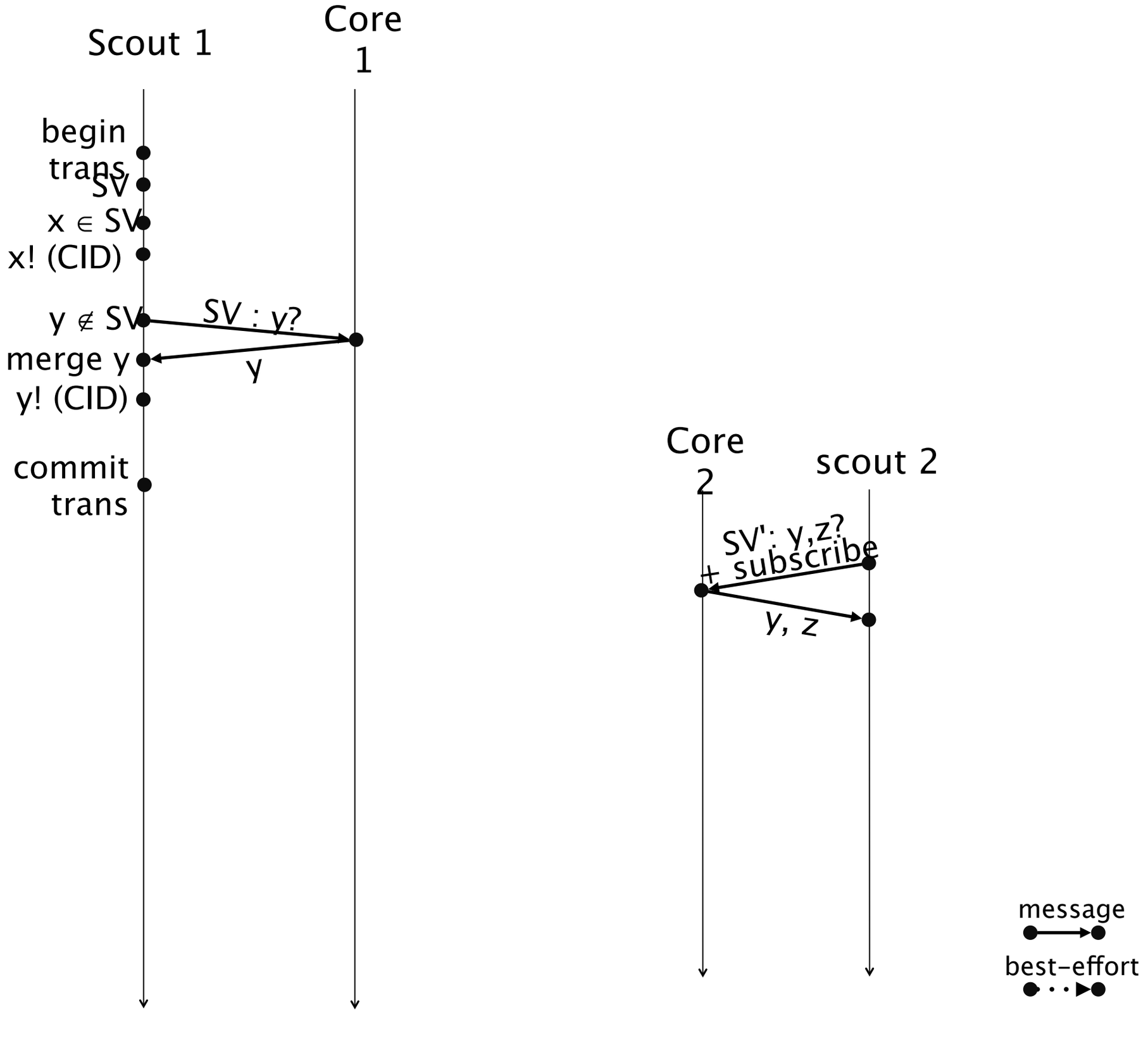
Transaction hand-over



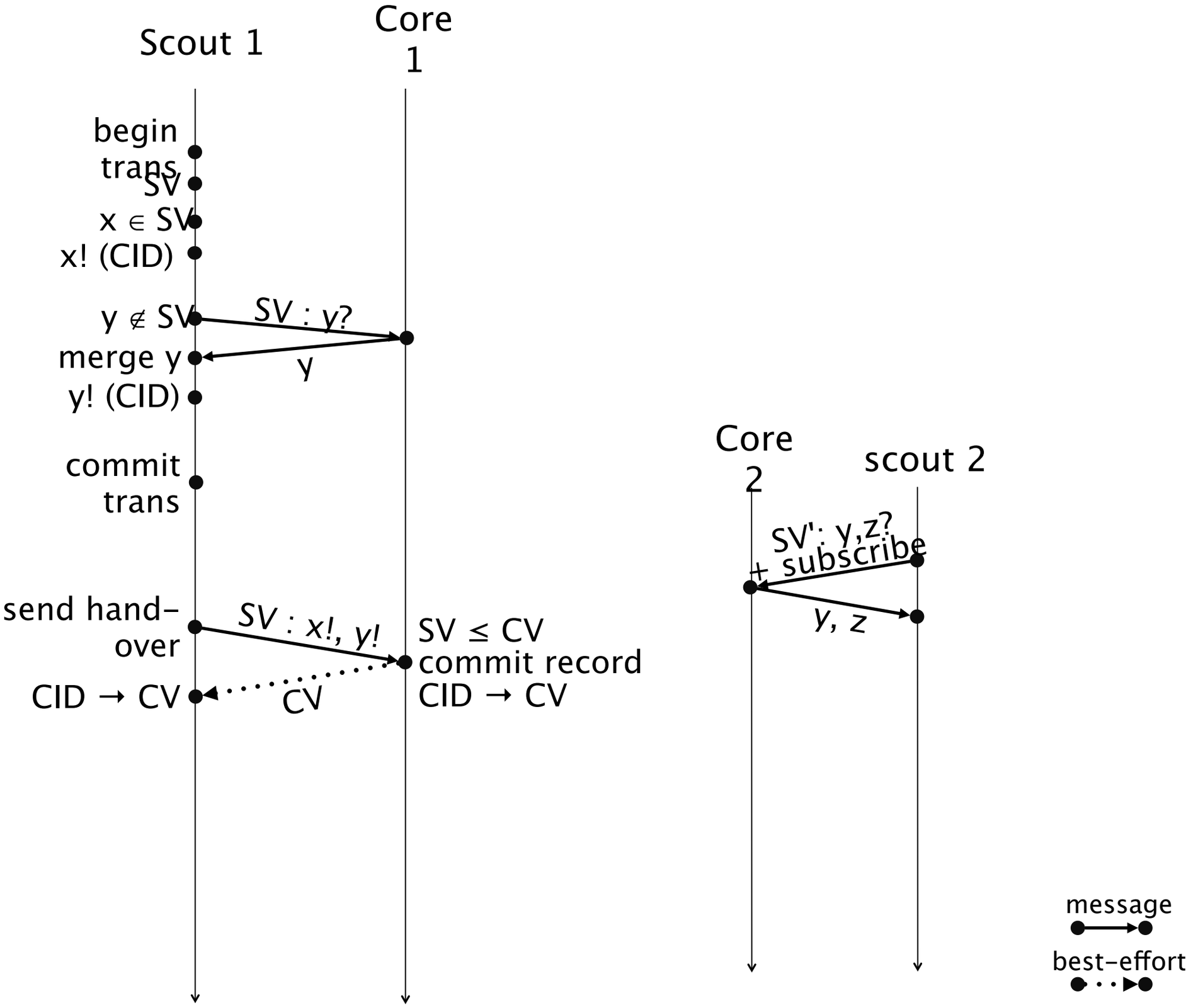
Transaction hand-over



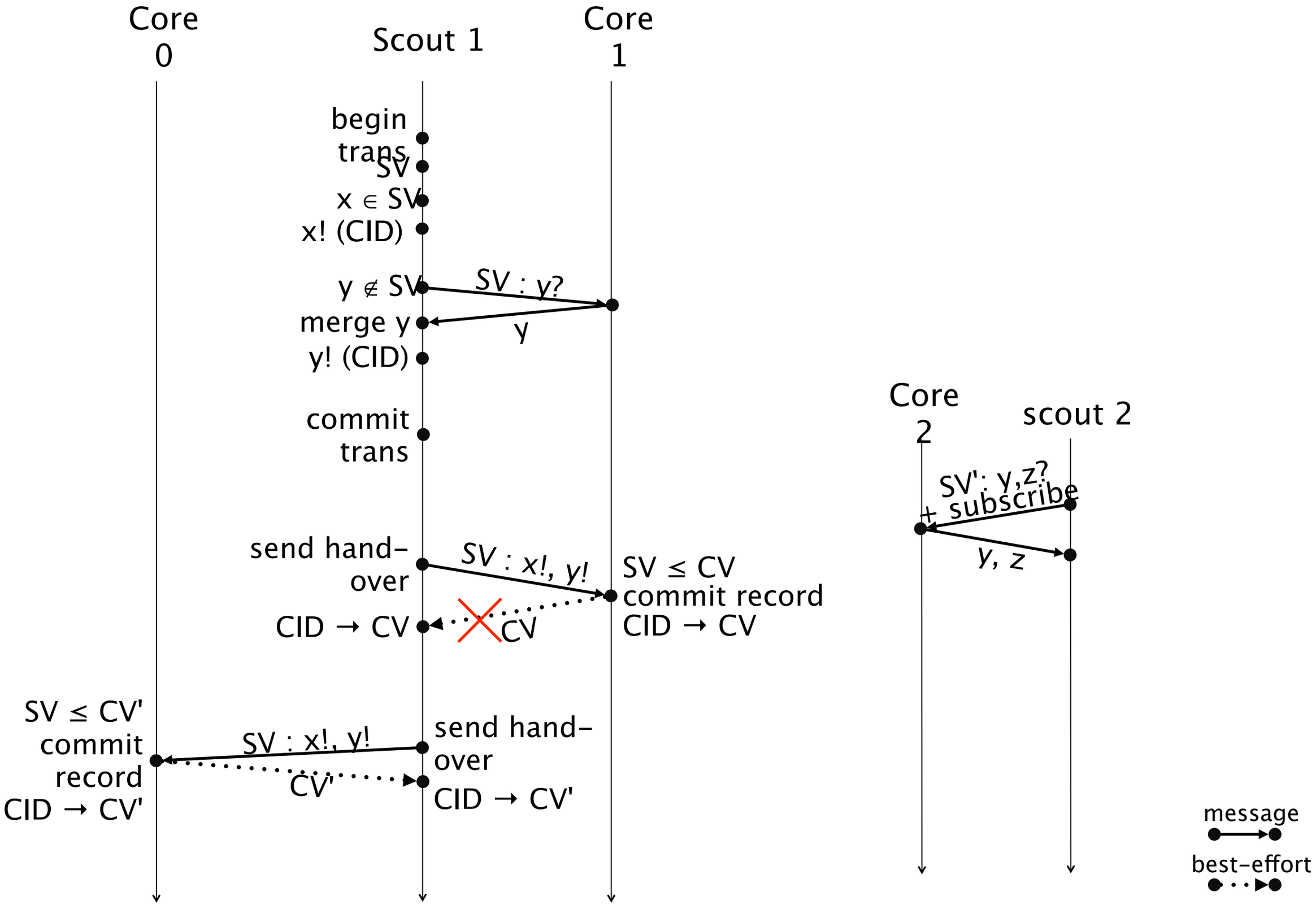
Transaction hand-over



Transaction hand-over



Transaction hand-over



Transaction hand-over