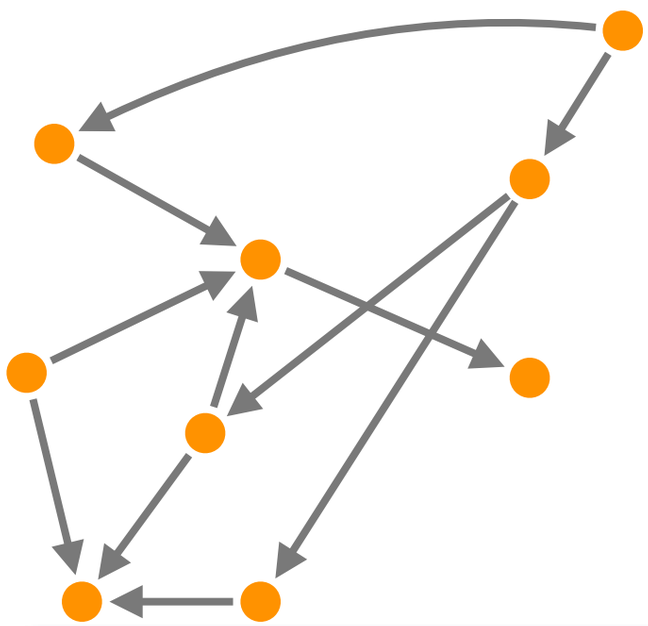
Conflict-free Replicated Data Types (CRDTs)

for collaborative environments

Marc Shapiro, INRIA & LIP6
Nuno Preguiça, U. Nova de Lisboa
Carlos Baquero, U. Minho
Marek Zawirski, INRIA & UPMC

Conflict-free objects for large-scale distribution



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

Shared Mutable data

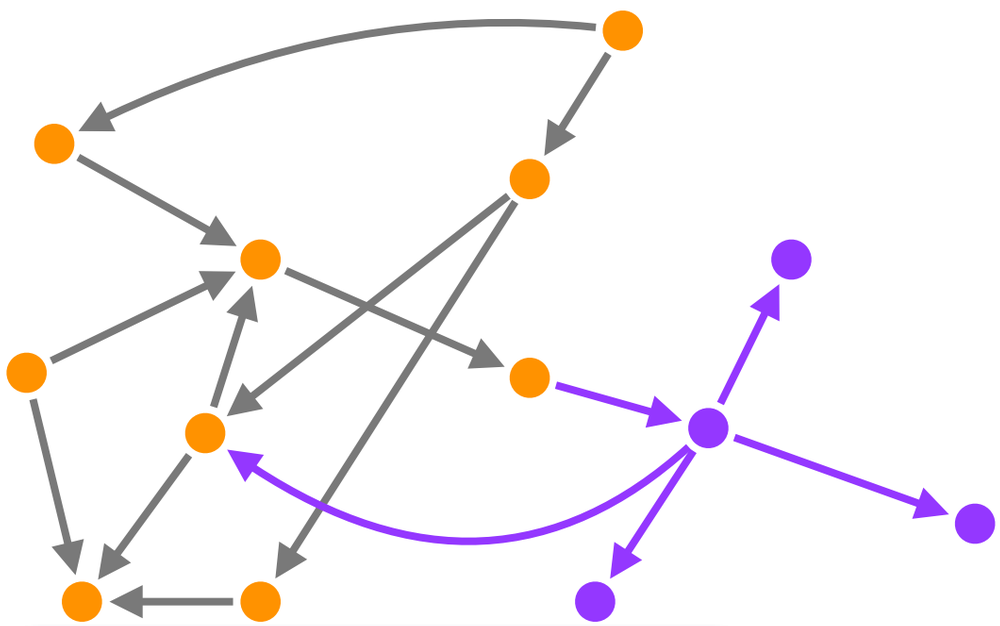
- Read ⇒ replicate
- Updates?

Novel, principled approach:
Conflict-free objects

Can we design useful object types *without any synchronisation* whatsoever?

Can we build practical systems from such objects?

Conflict-free objects for large-scale distribution



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

Shared Mutable data

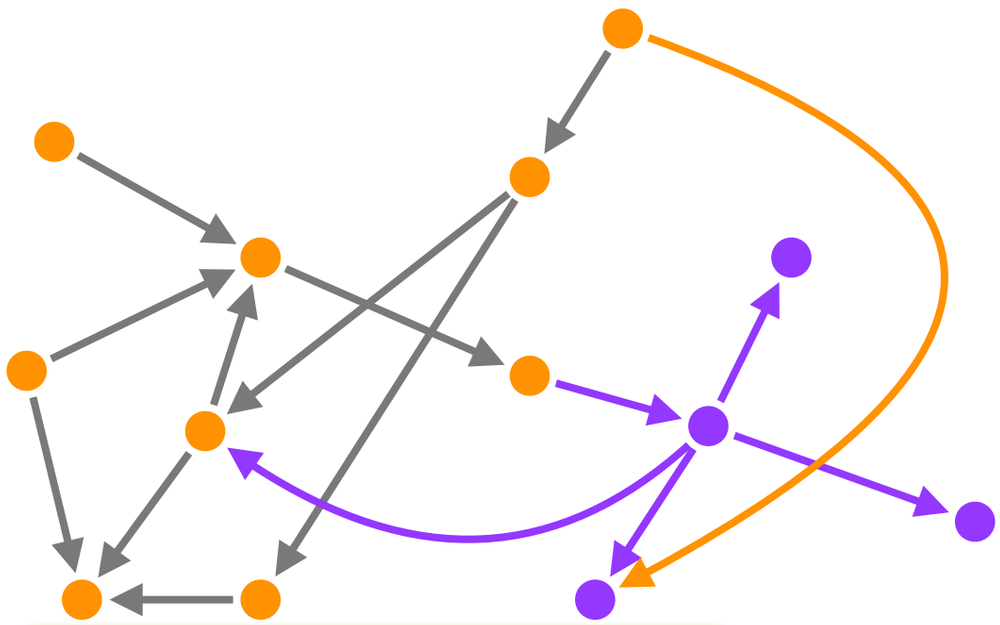
- Read ⇒ replicate
- Updates?

Novel, principled approach:
Conflict-free objects

Can we design useful object types *without any synchronisation* whatsoever?

Can we build practical systems from such objects?

Conflict-free objects for large-scale distribution



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

Shared Mutable data

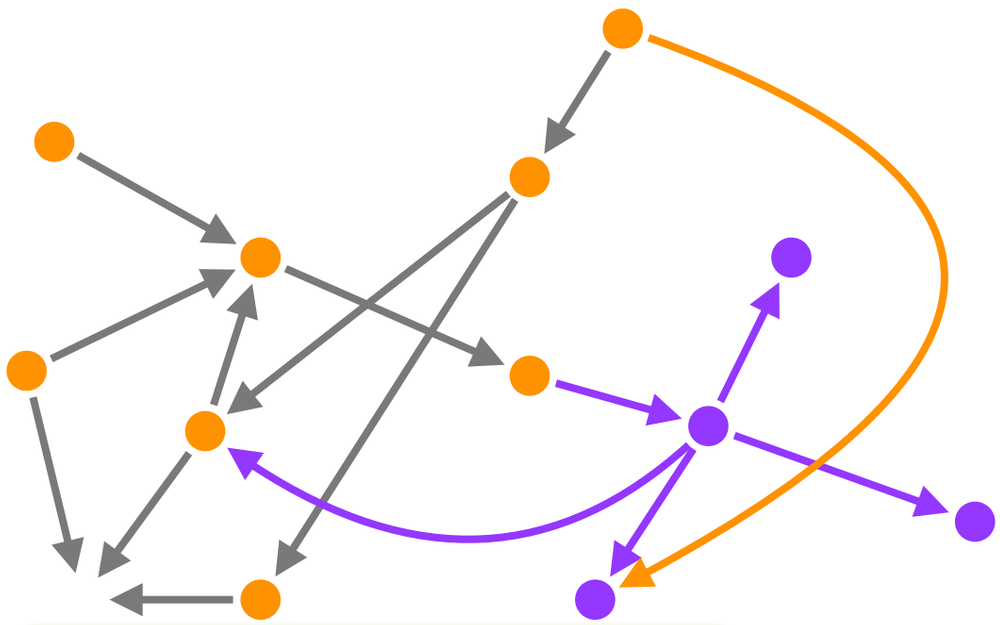
- Read ⇒ replicate
- Updates?

Novel, principled approach:
Conflict-free objects

Can we design useful object types *without any synchronisation* whatsoever?

Can we build practical systems from such objects?

Conflict-free objects for large-scale distribution



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

Shared Mutable data

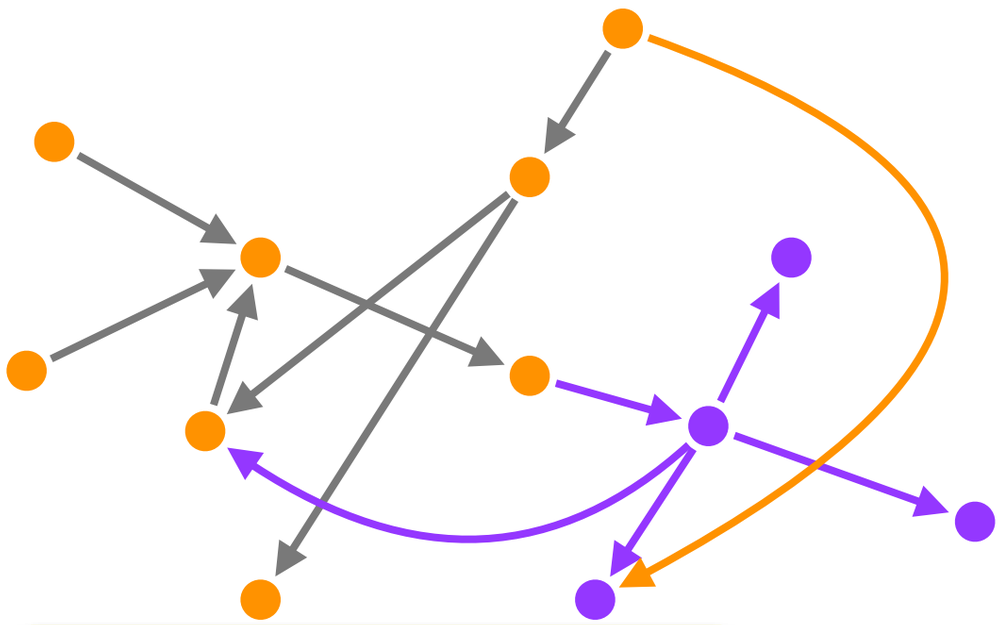
- Read ⇒ replicate
- Updates?

Novel, principled approach:
Conflict-free objects

Can we design useful object types *without any synchronisation* whatsoever?

Can we build practical systems from such objects?

Conflict-free objects for large-scale distribution



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

Shared Mutable data

- Read ⇒ replicate
- Updates?

Novel, principled approach:
Conflict-free objects

Can we design useful object types *without any synchronisation* whatsoever?

Can we build practical systems from such objects?

Replication for beginners

Replicated data

Share data ⇒ Replicate at many locations

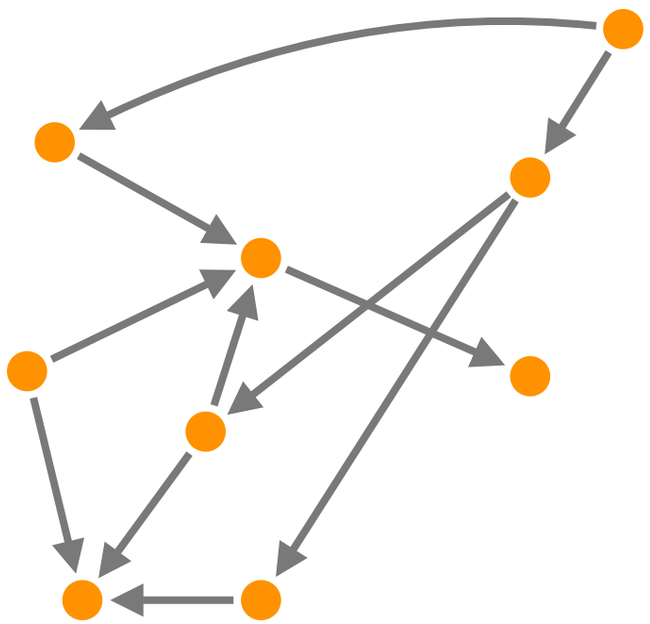
- Performance: local reads
- Availability: immune from network failure
- Fault-tolerance: replicate computation
- Scalability: load balancing

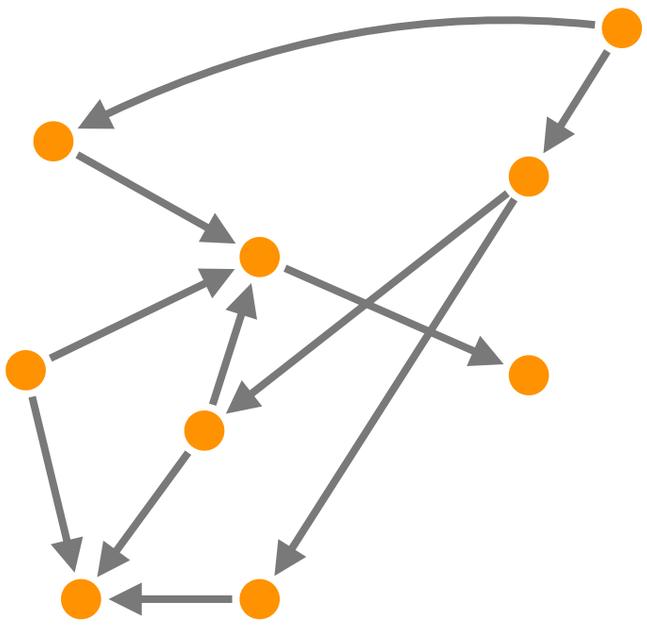
Updates

- Push to all replicas
- Conflicts: Consistency?

- Fault tolerance
- and parallelism too?
- Conflict!!

Strong consistency





Preclude conflicts

- All replicas execute updates in same total order
- Any deterministic object

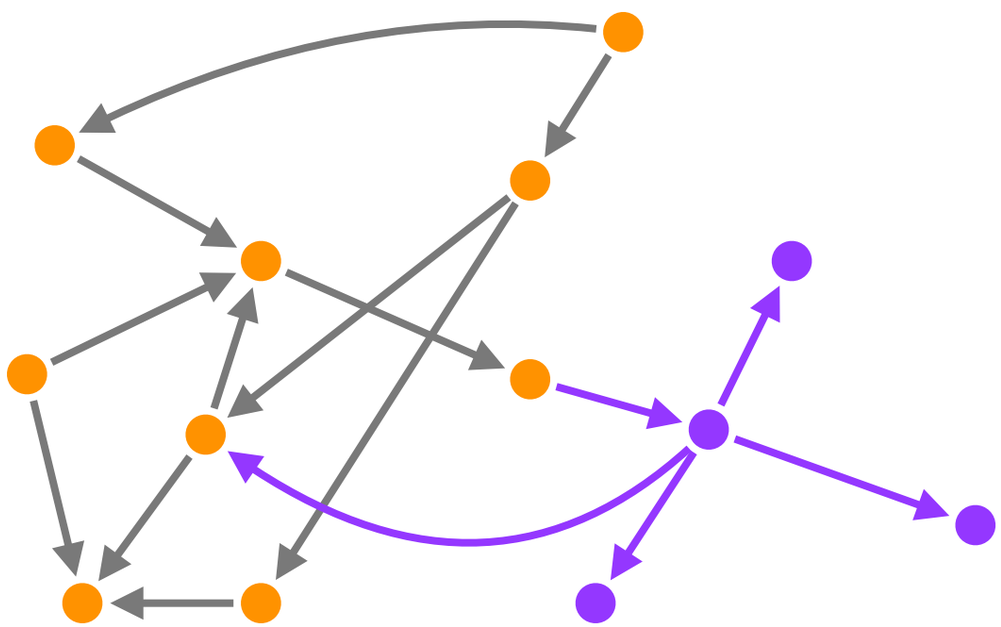
- Simultaneous N-way agreement

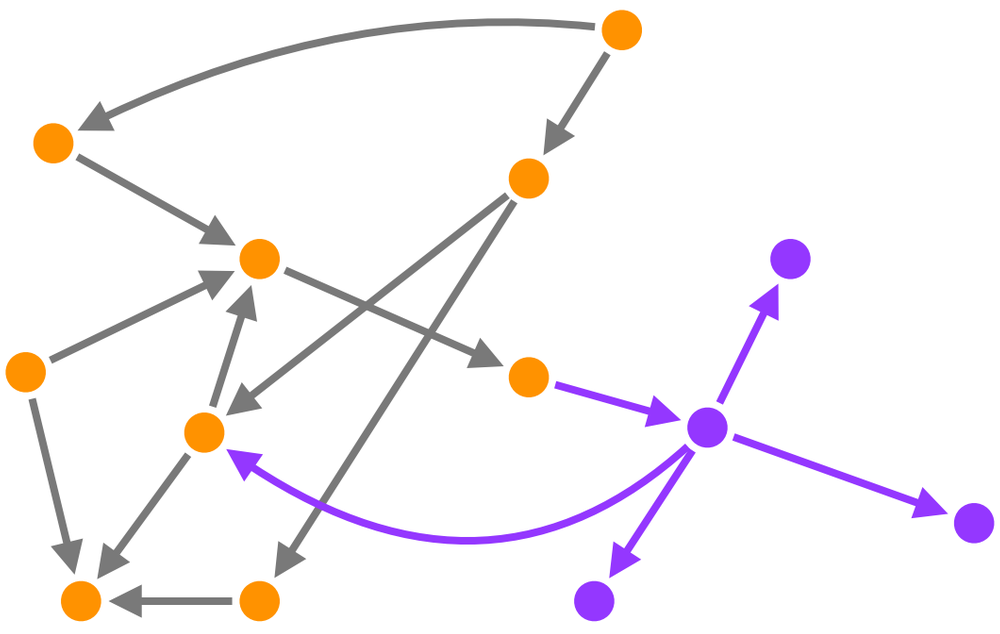
Consensus

- Serialisation bottleneck
- Tolerates < $n/2$ faults

- Very general
- Correct
- Doesn't scale

Strong consistency





Preclude conflicts

- All replicas execute updates in same total order
- Any deterministic object

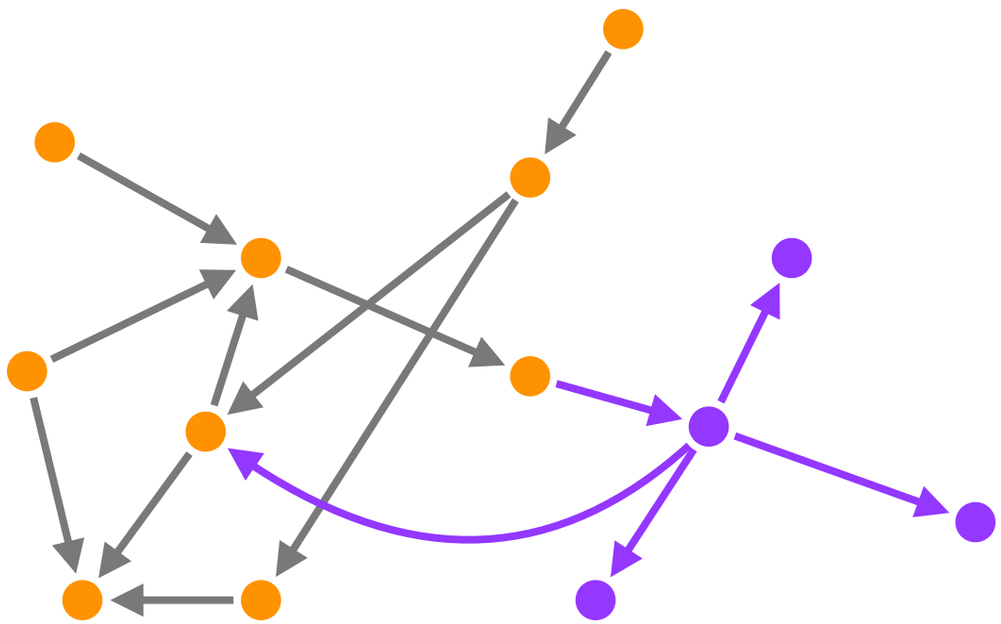
- Simultaneous N-way agreement

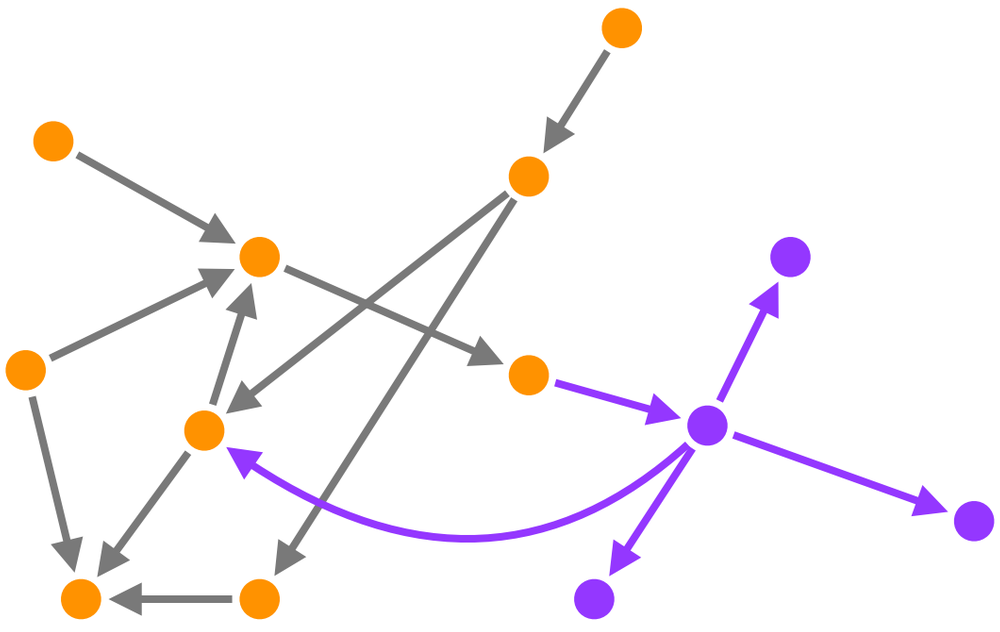
Consensus

- Serialisation bottleneck
- Tolerates < *n*/2 faults

- Very general
- Correct
- Doesn't scale

Strong consistency





Preclude conflicts

- All replicas execute updates in same total order
- Any deterministic object

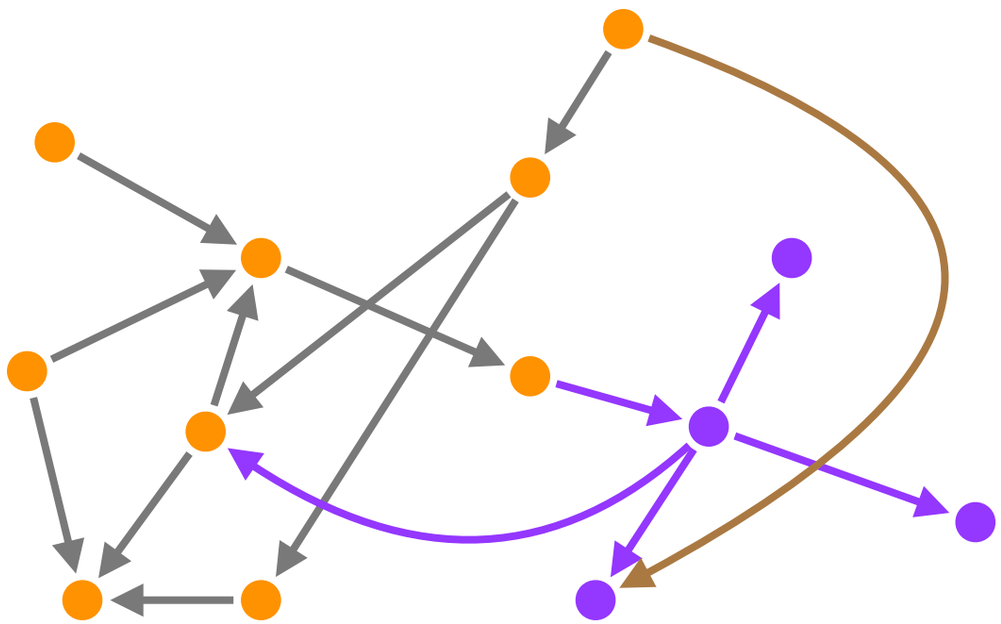
- Simultaneous N-way agreement

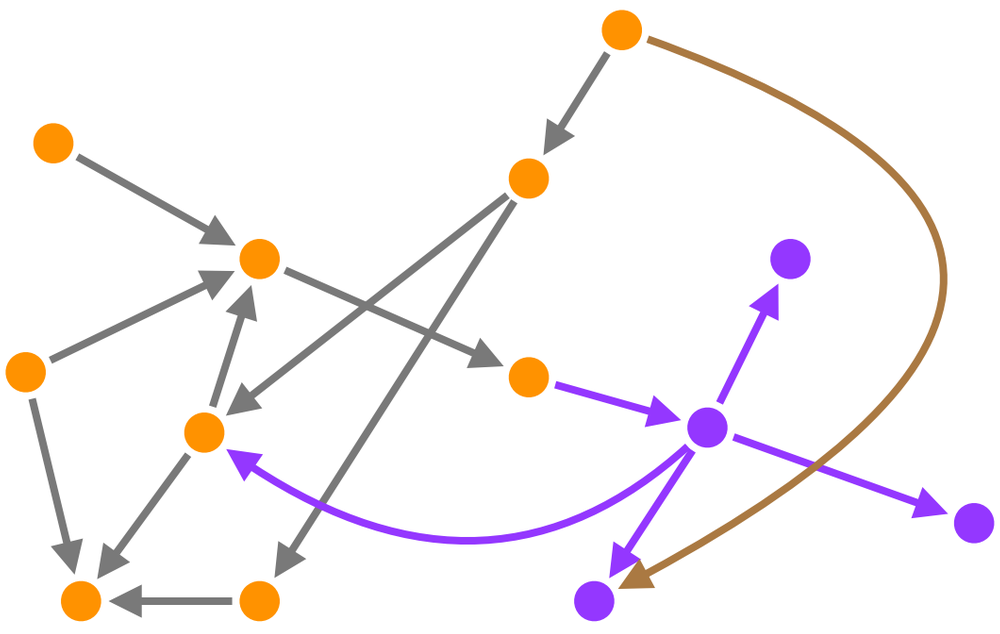
Consensus

- Serialisation bottleneck
- Tolerates < $n/2$ faults

- Very general
- Correct
- Doesn't scale

Strong consistency





3

Preclude conflicts

- All replicas execute updates in same total order
- Any deterministic object

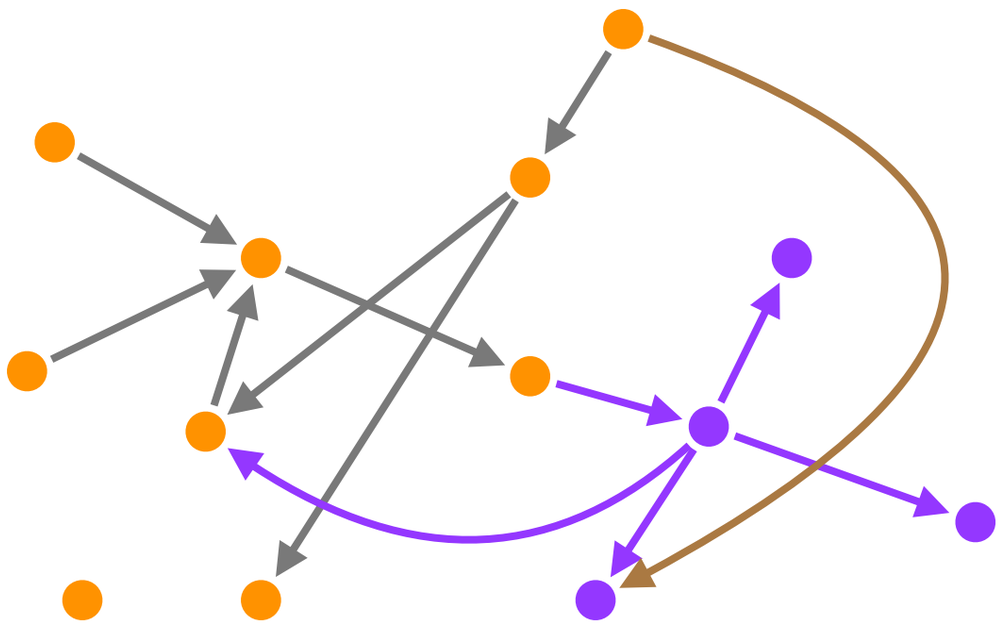
- Simultaneous N-way agreement

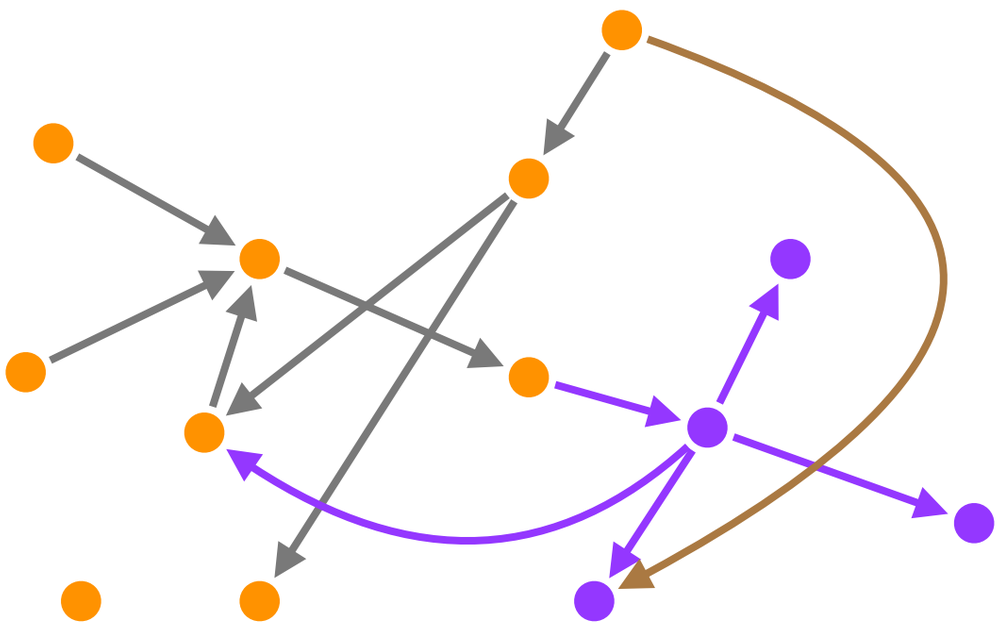
Consensus

- Serialisation bottleneck
- Tolerates < *n*/2 faults

- Very general
- Correct
- Doesn't scale

Strong consistency





Preclude conflicts

- All replicas execute updates in same total order
- Any deterministic object

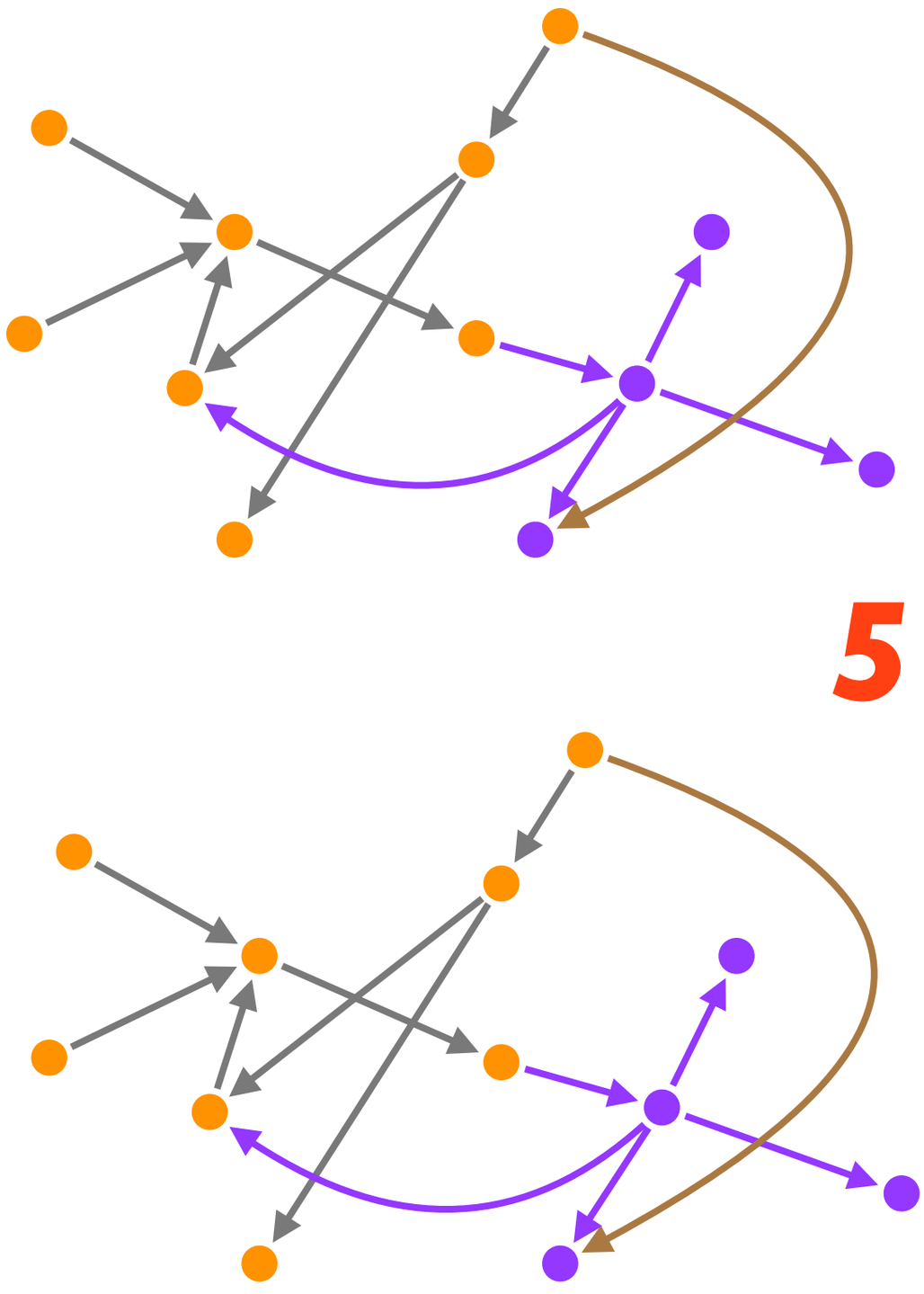
Simultaneous N-way agreement

Consensus

- Serialisation bottleneck
- Tolerates < *n*/2 faults

Very general
Correct
Doesn't scale

Strong consistency



Preclude conflicts

- All replicas execute updates in same total order
- Any deterministic object

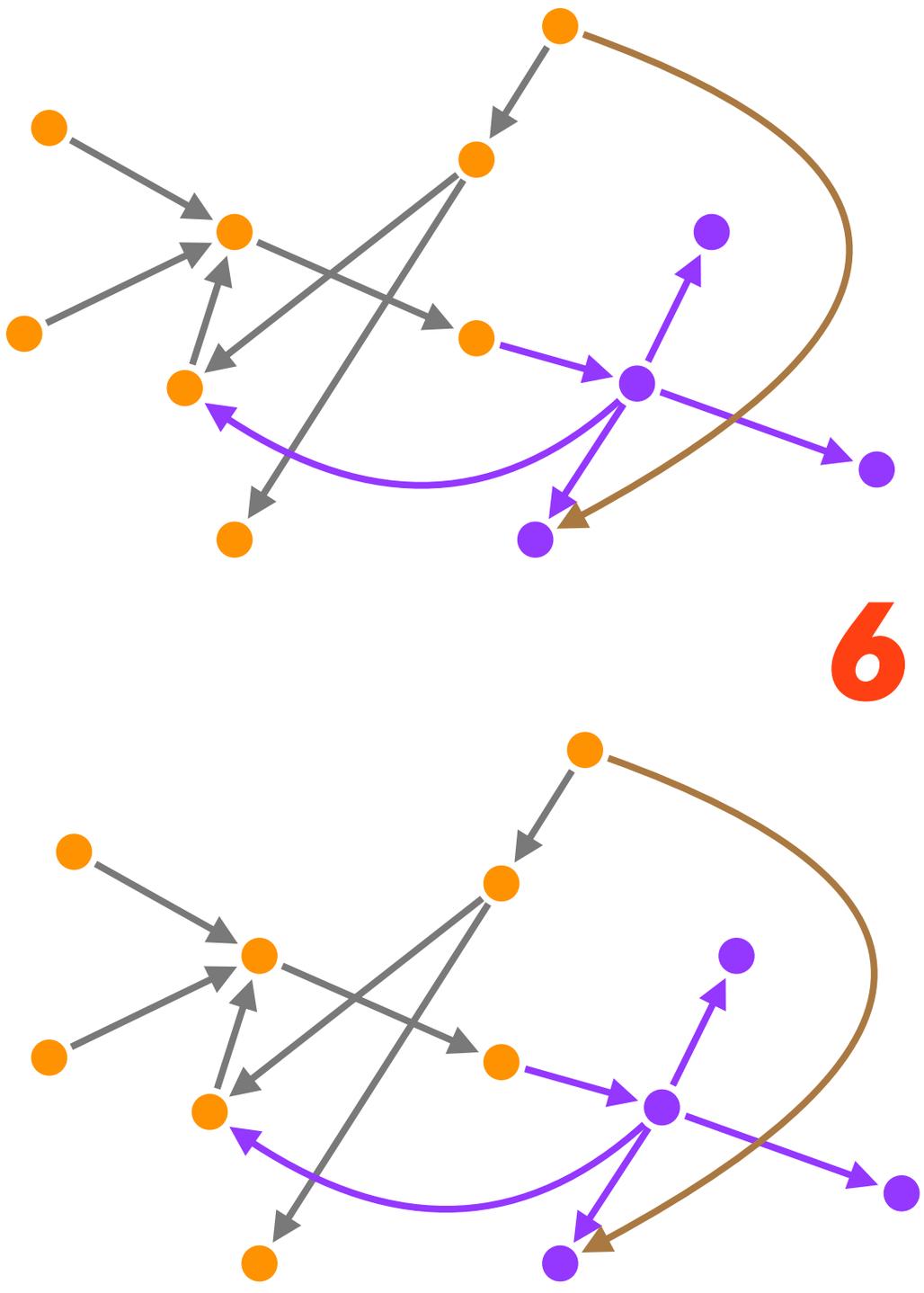
Simultaneous N-way agreement

Consensus

- Serialisation bottleneck
- Tolerates < *n*/2 faults

Very general
Correct
Doesn't scale

Strong consistency



Preclude conflicts

- All replicas execute updates in same total order
- Any deterministic object

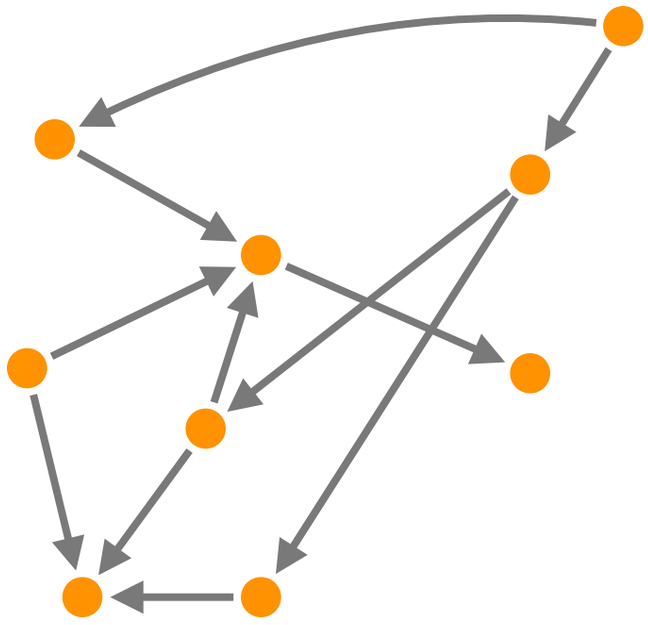
Simultaneous N-way agreement

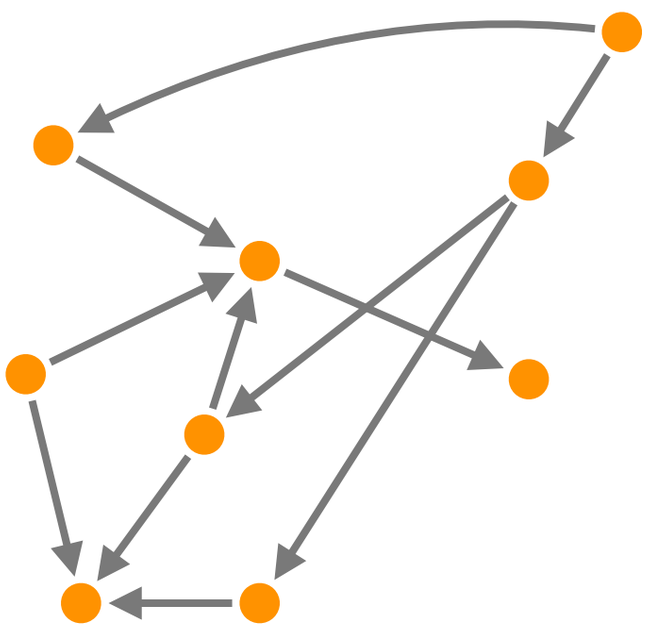
Consensus

- Serialisation bottleneck
- Tolerates < *n*/2 faults

Very general
Correct
Doesn't scale

Eventual Consistency





Update local + propagate

- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

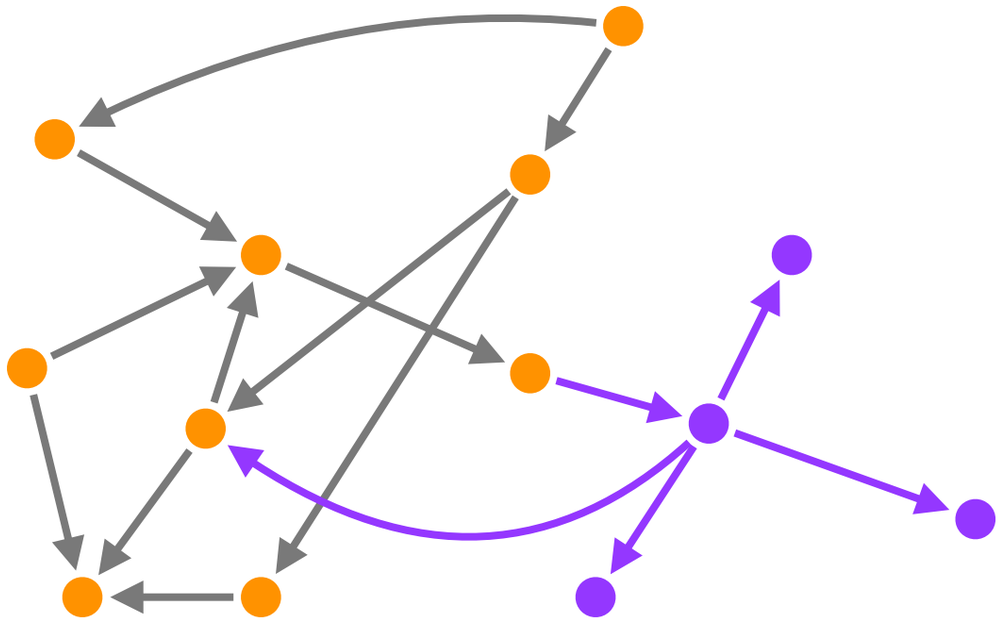
On conflict

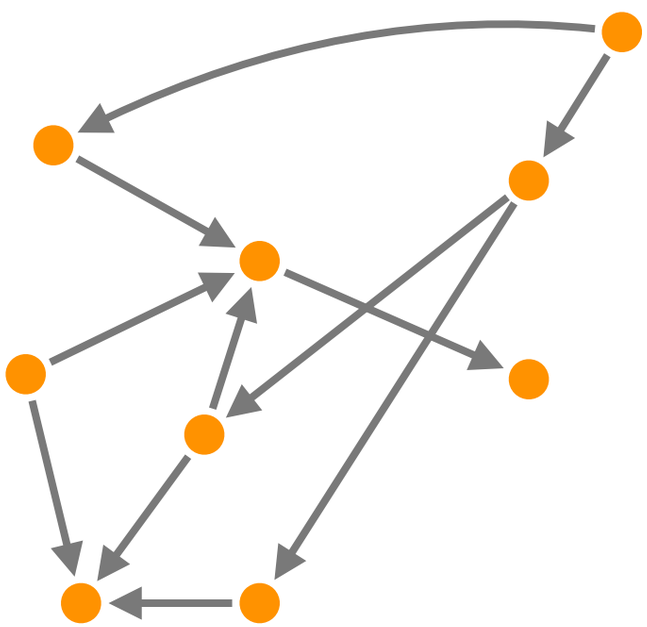
- Arbitrate
- Roll back

- Availability ++
- Parallelism++
- Latency --
- Complexity ++

Consensus moved to background

Eventual Consistency





Update local + propagate

- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

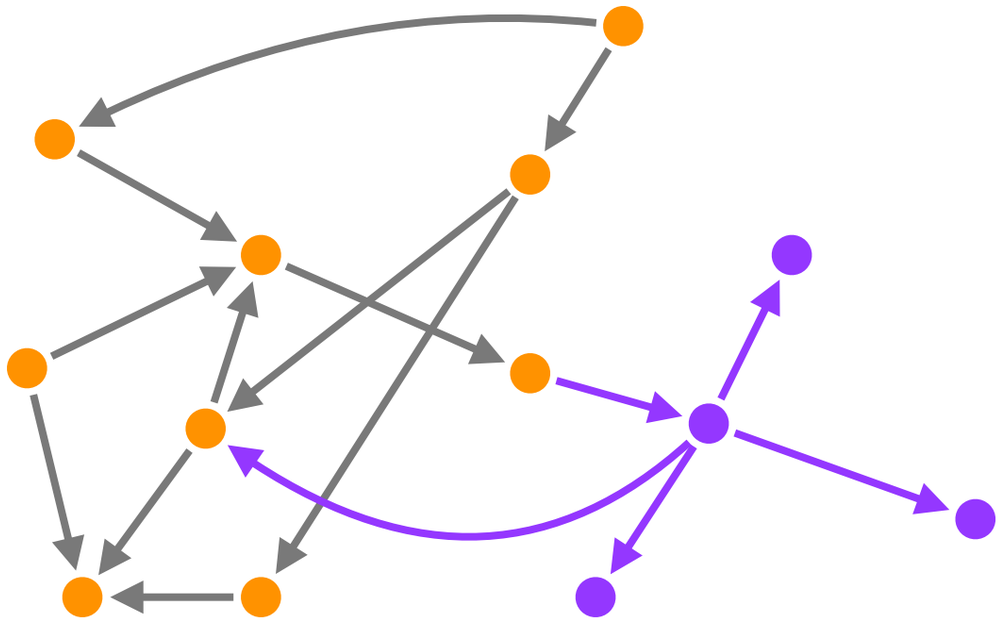
On conflict

- Arbitrate
- Roll back

- Availability ++
- Parallelism++
- Latency --
- Complexity ++

Consensus moved to background

Eventual Consistency



Update local + propagate

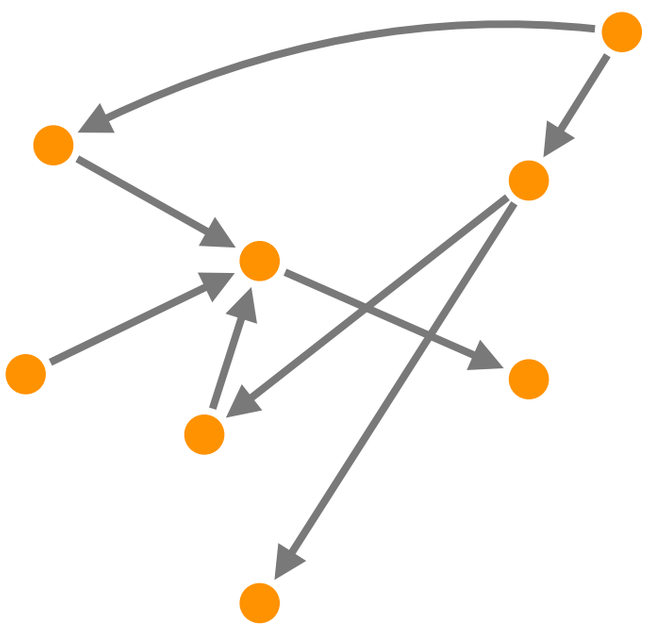
- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

On conflict

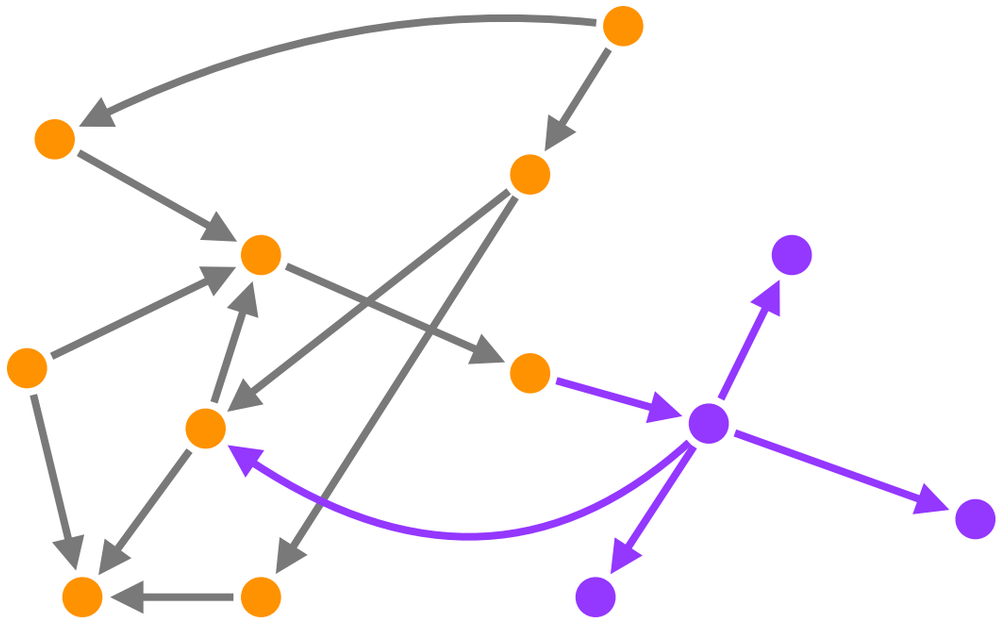
- Arbitrate
- Roll back

- Availability ++
- Parallelism++
- Latency --
- Complexity ++

Consensus moved to background



Eventual Consistency



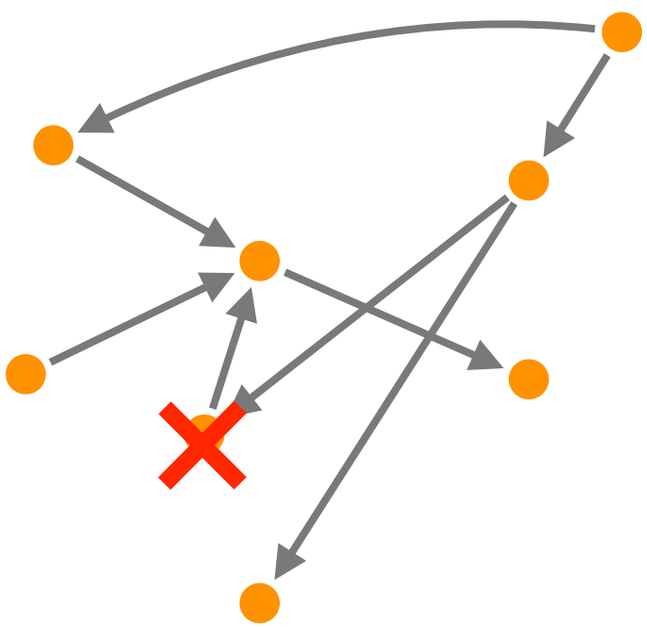
Update local + propagate

- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

On conflict

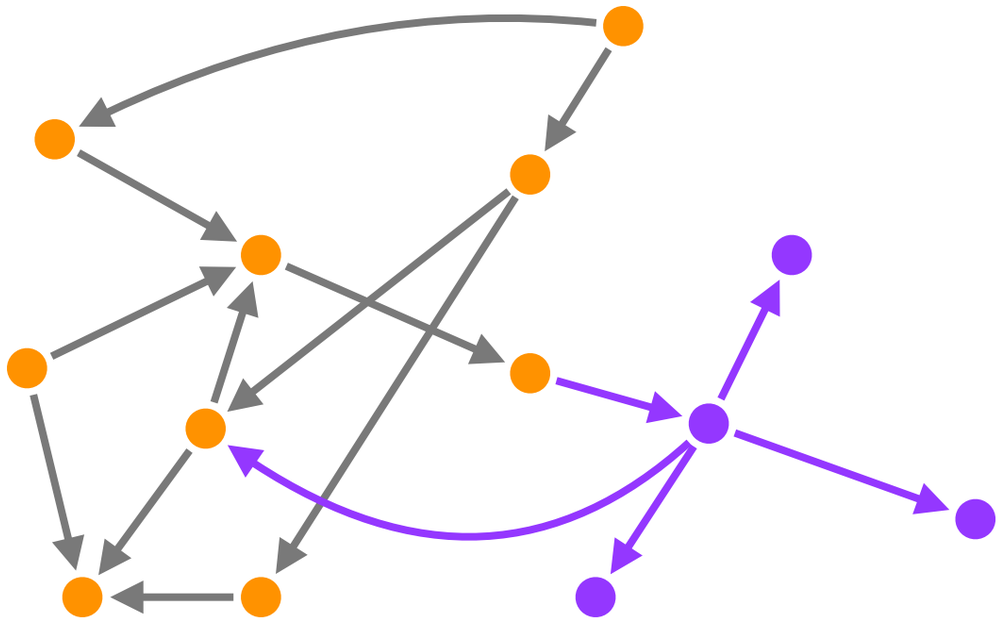
- Arbitrate
- Roll back

- Availability ++
- Parallelism++
- Latency --
- Complexity ++



Consensus moved to background

Eventual Consistency



Update local + propagate

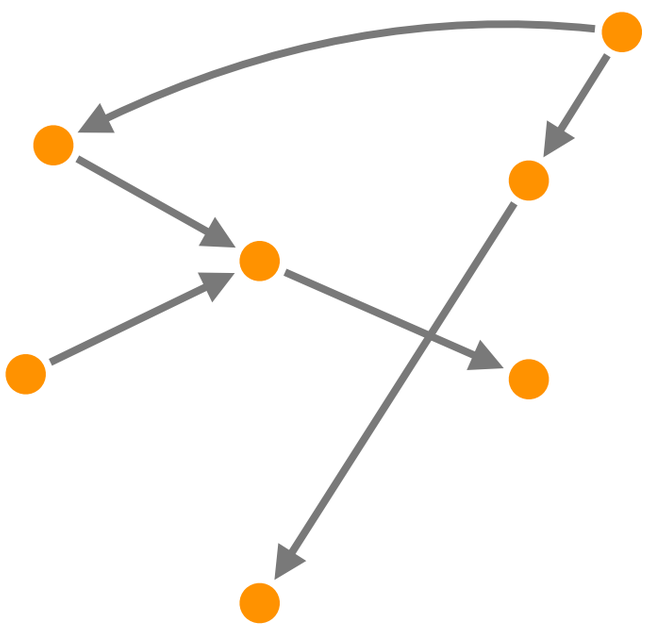
- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

On conflict

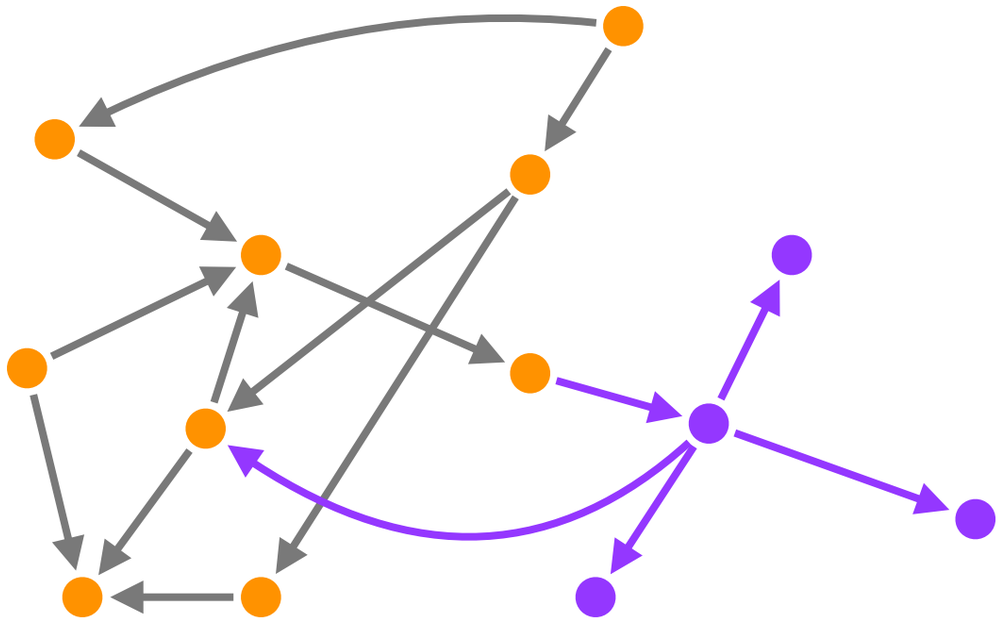
- Arbitrate
- Roll back

- Availability ++
- Parallelism++
- Latency --
- Complexity ++

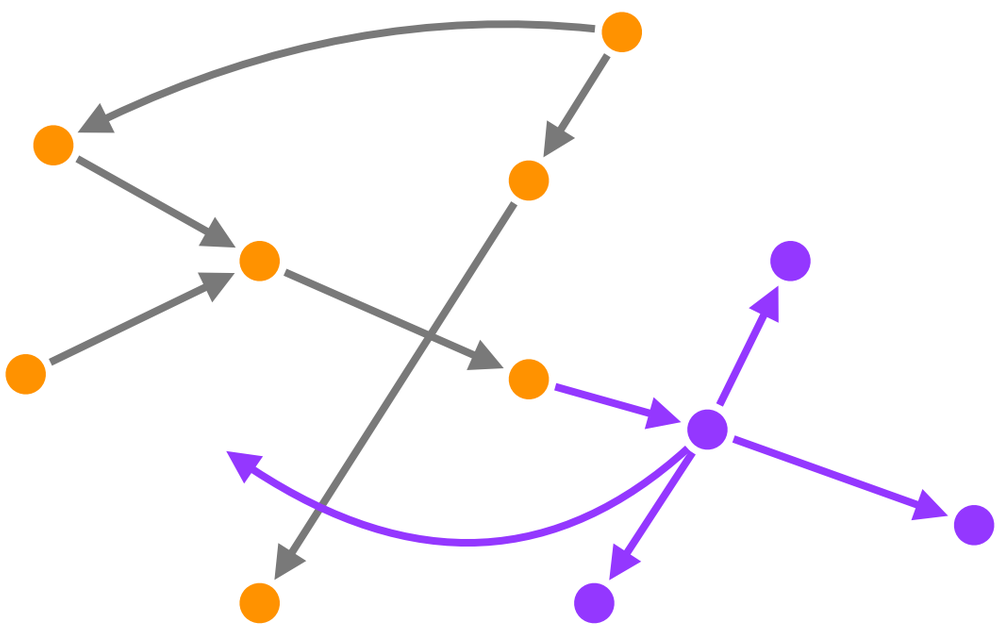
Consensus moved to background



Eventual Consistency



Conflict!



Update local + propagate

- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

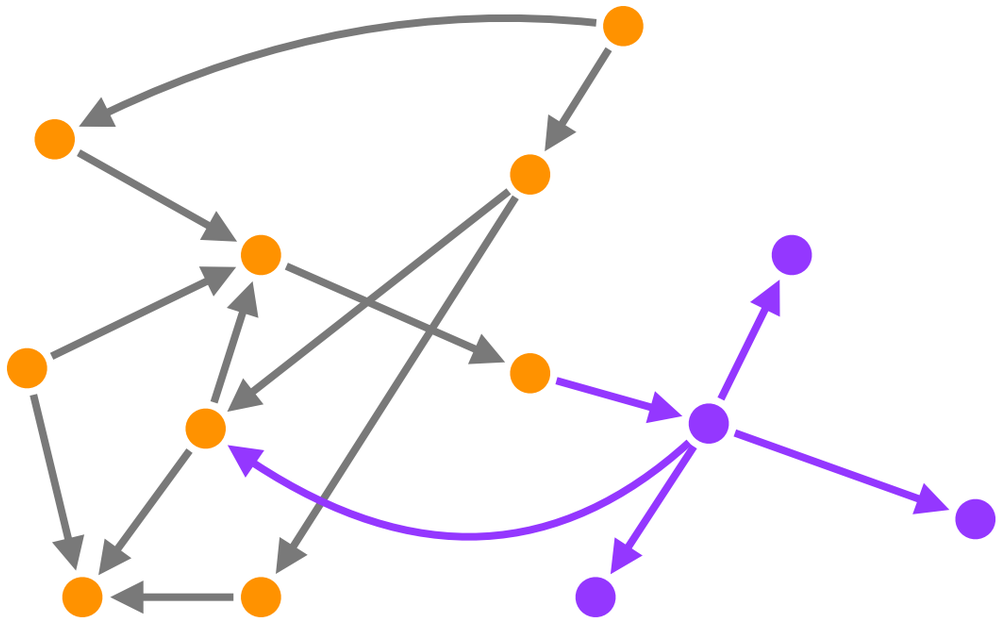
On conflict

- Arbitrate
- Roll back

- Availability ++
- Parallelism++
- Latency --
- Complexity ++

Consensus moved to background

Eventual Consistency



Update local + propagate

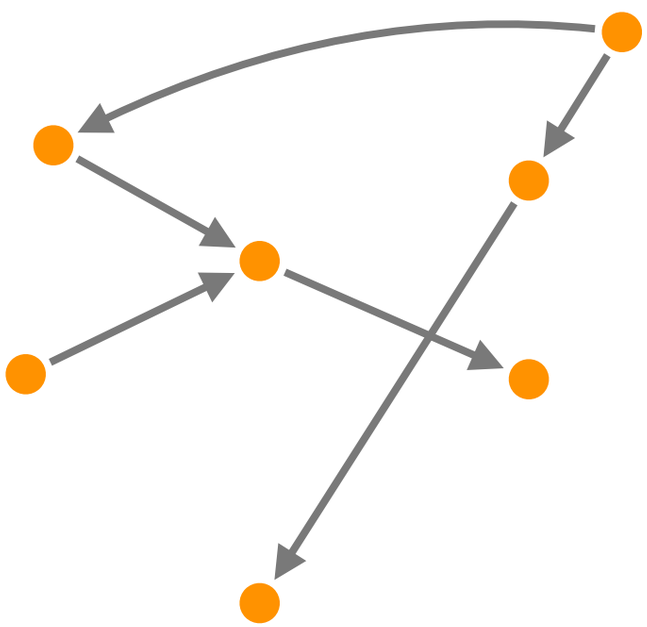
- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

On conflict

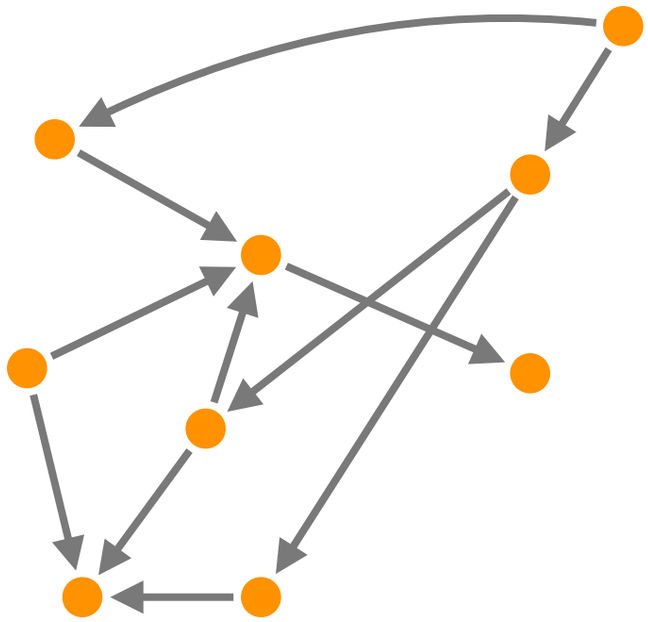
- Arbitrate
- Roll back

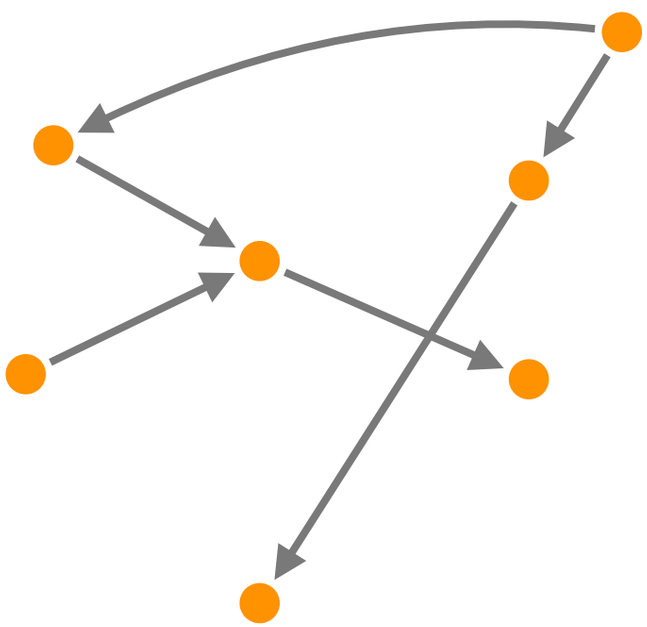
- Availability ++
- Parallelism++
- Latency --
- Complexity ++

Consensus moved to background



Eventual Consistency





Update local + propagate

- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

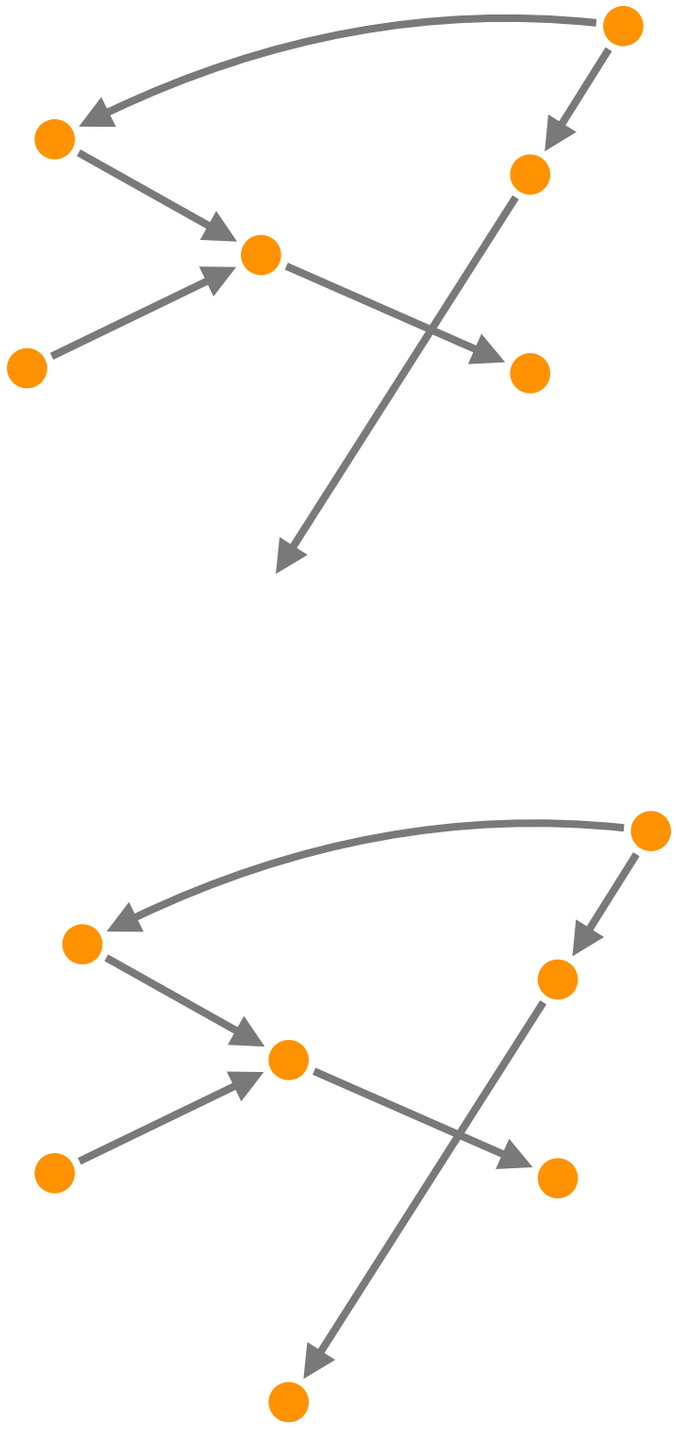
On conflict

- Arbitrate
- Roll back

- Availability ++
- Parallelism++
- Latency --
- Complexity ++

Consensus moved to background

Eventual Consistency



Update local + propagate

- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

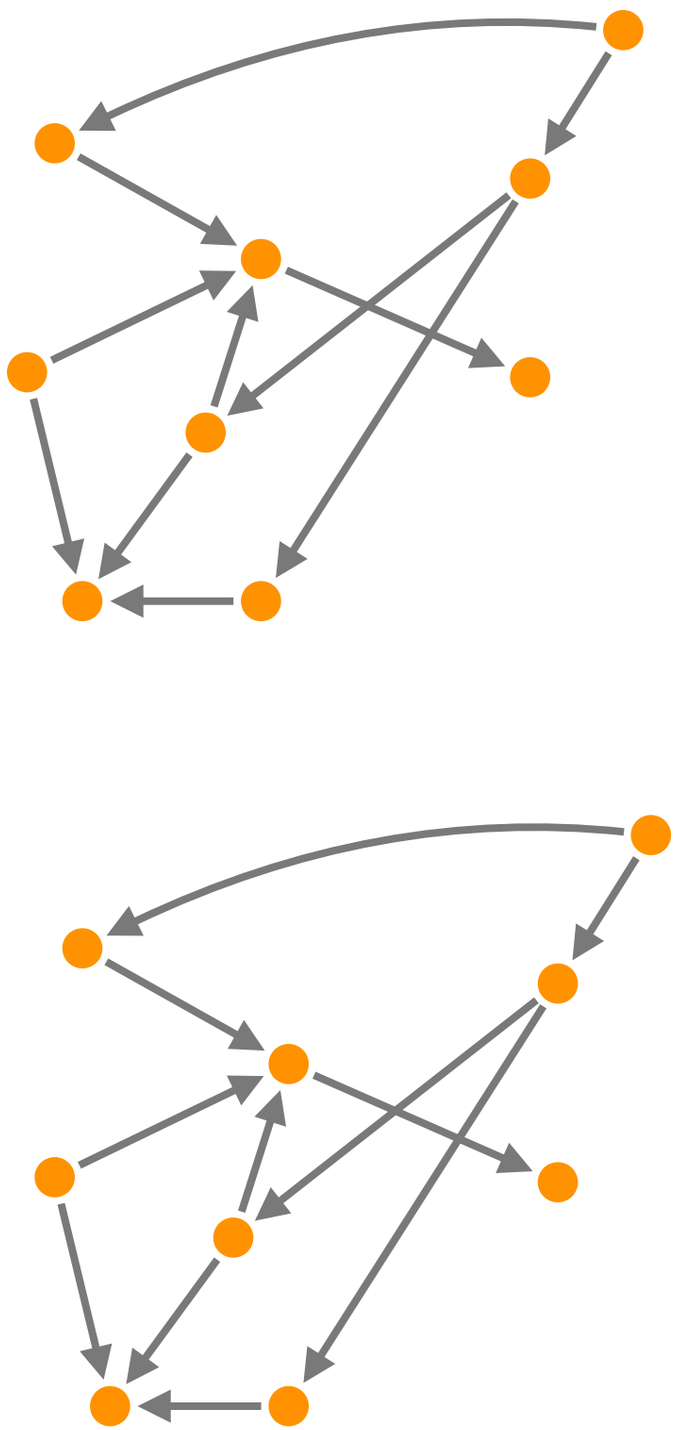
On conflict

- Arbitrate
- Roll back

- Availability ++
- Parallelism++
- Latency --
- Complexity ++

Consensus moved to background

Strong Eventual Consistency



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

Update local + propagate

- No synch
- Expose intermediate state
- Eventual, reliable delivery

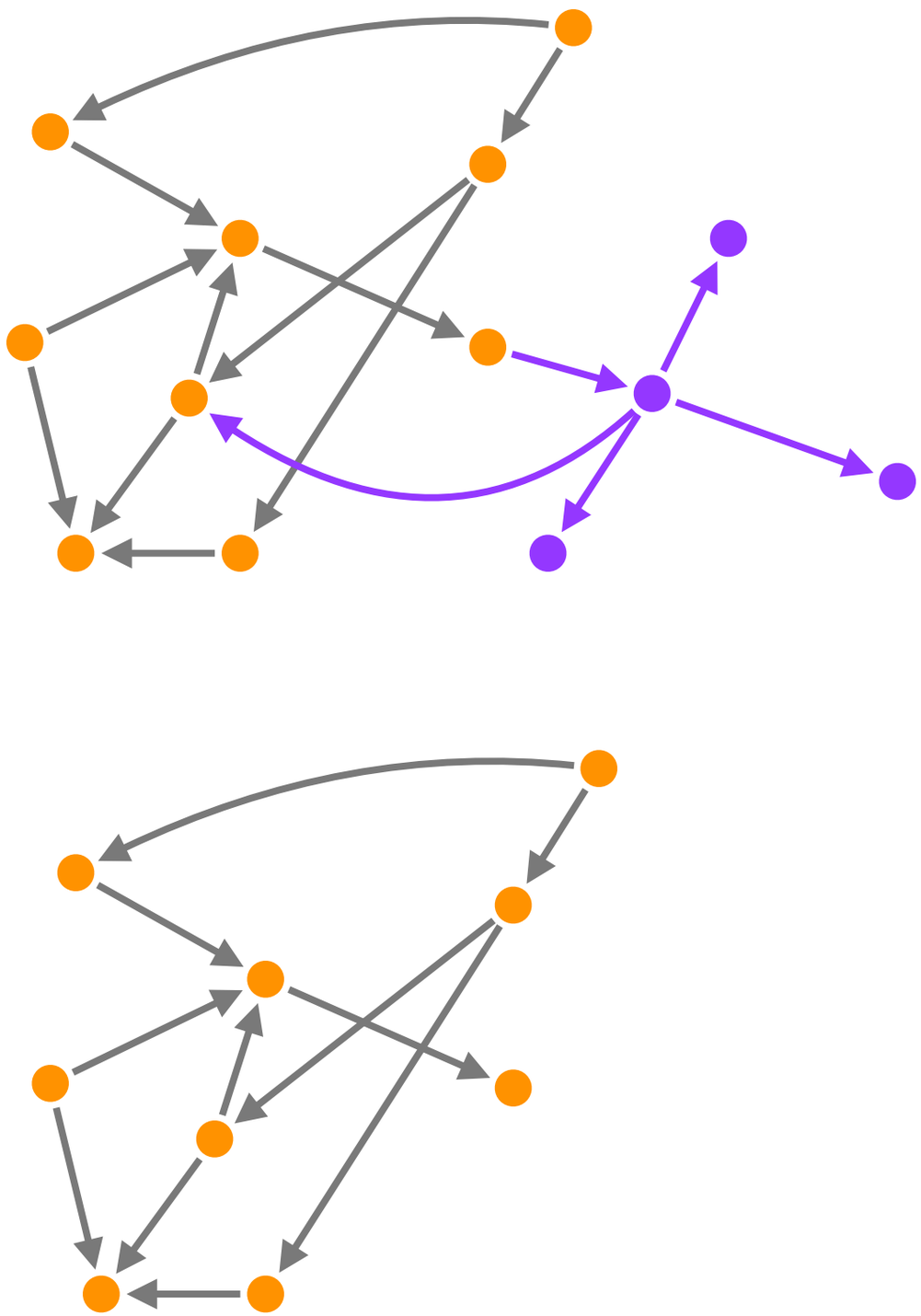
No conflict

- Deterministic outcome of concurrent updates

No consensus: ≤ n-1 faults
Not universal

Solves the CAP problem

Strong Eventual Consistency



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

Update local + propagate

- No synch
- Expose intermediate state
- Eventual, reliable delivery

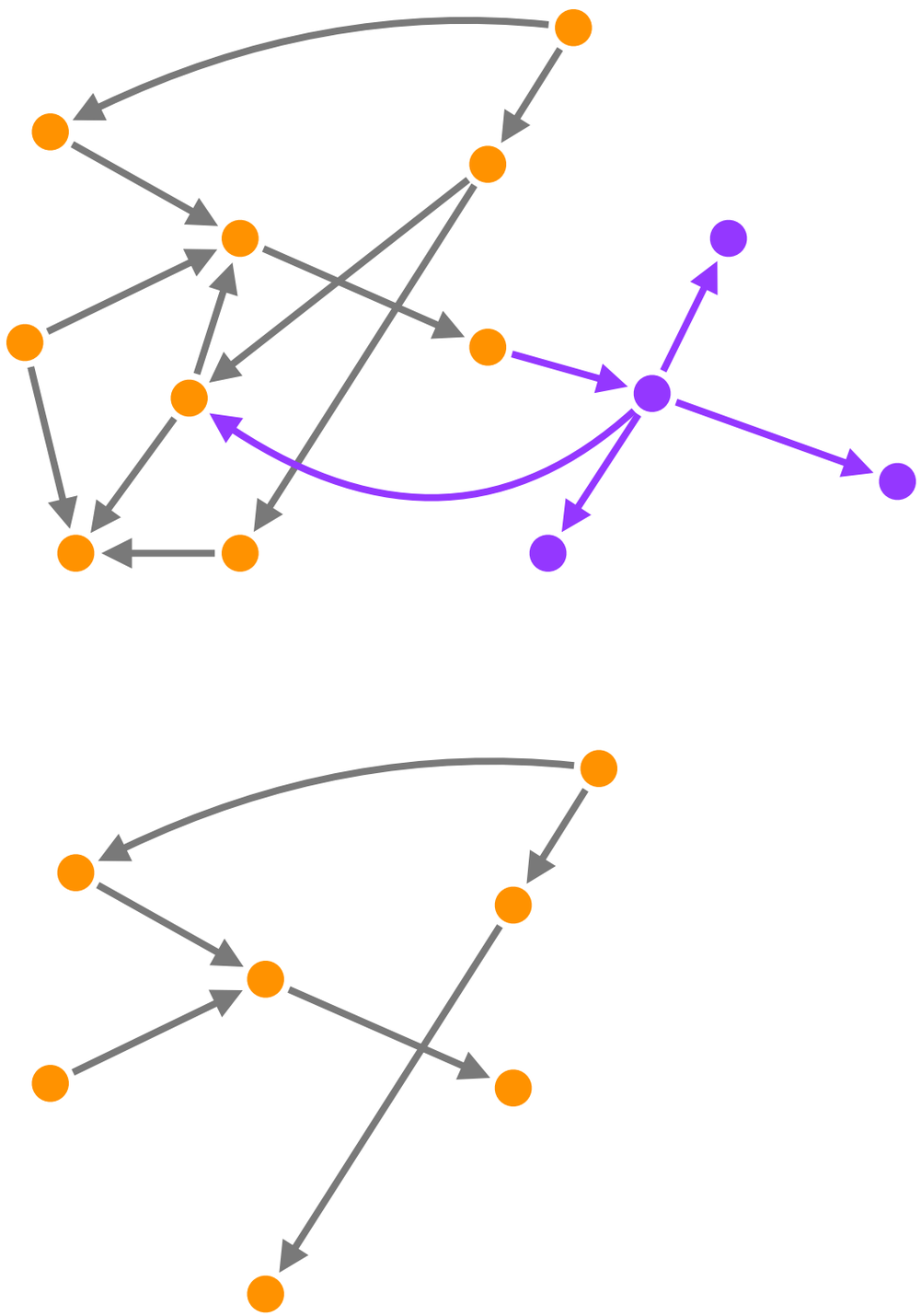
No conflict

- Deterministic outcome of concurrent updates

No consensus: ≤ n-1 faults
Not universal

Solves the CAP problem

Strong Eventual Consistency



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

Update local + propagate

- No synch
- Expose intermediate state
- Eventual, reliable delivery

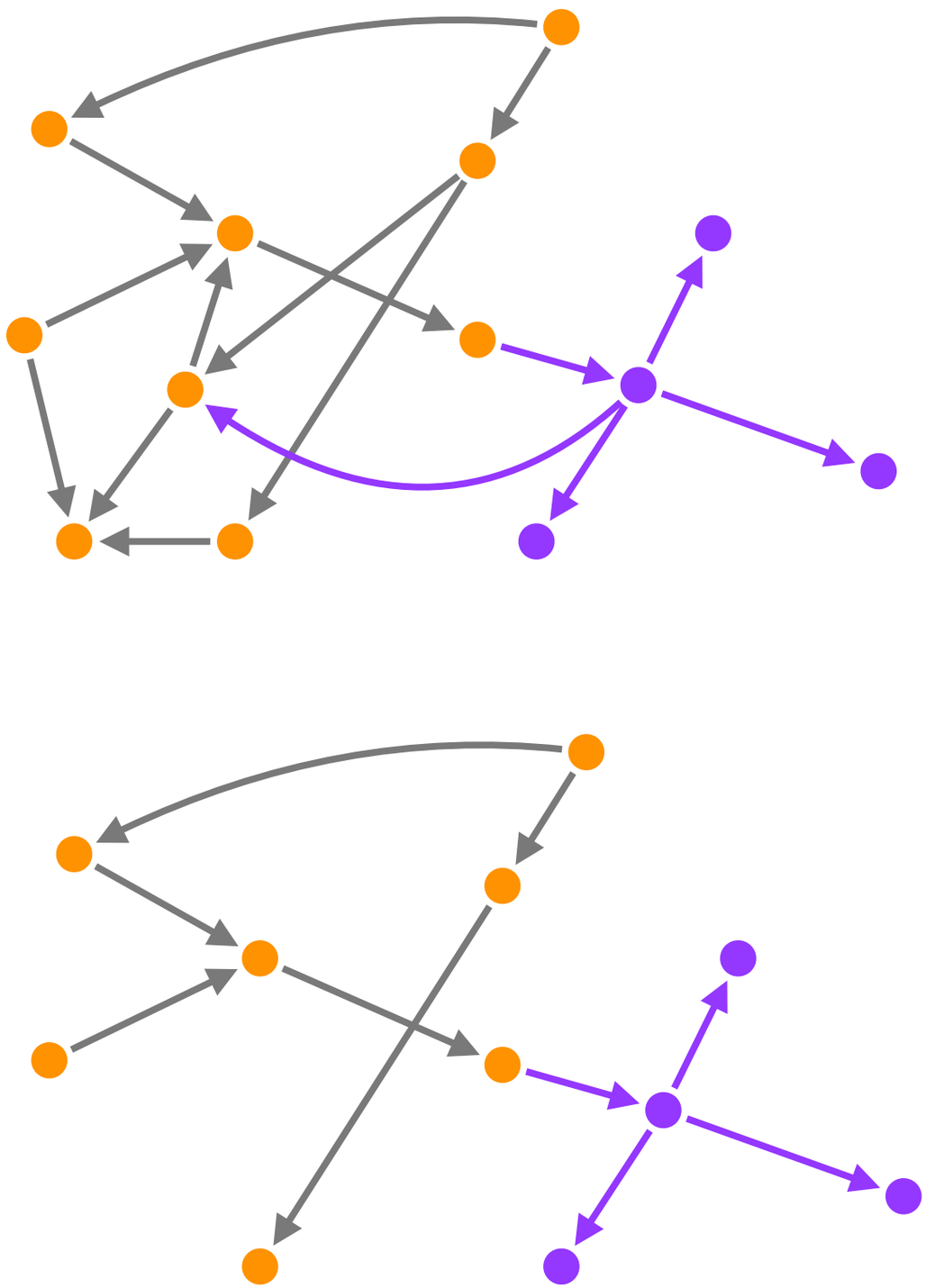
No conflict

- Deterministic outcome of concurrent updates

No consensus: ≤ n-1 faults
Not universal

Solves the CAP problem

Strong Eventual Consistency



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

Update local + propagate

- No synch
- Expose intermediate state
- Eventual, reliable delivery

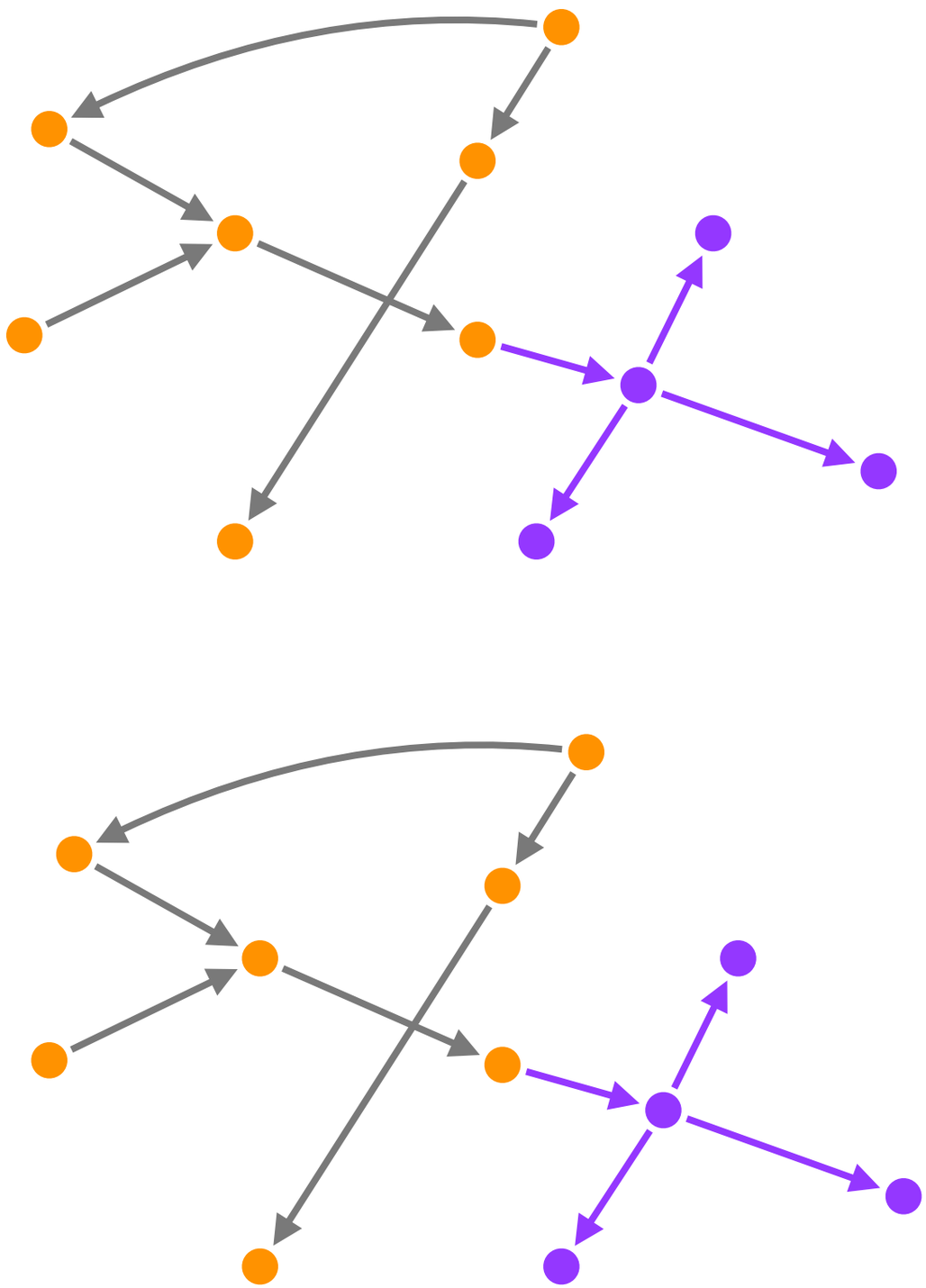
No conflict

- Deterministic outcome of concurrent updates

No consensus: ≤ n-1 faults
Not universal

Solves the CAP problem

Strong Eventual Consistency



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

Update local + propagate

- No synch
- Expose intermediate state
- Eventual, reliable delivery

No conflict

- Deterministic outcome of concurrent updates

No consensus: ≤ n-1 faults
Not universal

Solves the CAP problem

The challenge:

What interesting objects can we design with no synchronisation whatsoever?

Portfolio of CRDTs

Register
- Last-Writer Wins
- Multi-Value

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

Map
- Set of Registers

Counter
- Unlimited
- Non-negative

Graphs
- **Directed**
- Monotonic DAG
- Edit graph

Sequence
- **Edit sequence**

Set design alternatives

- linearisable: sequential order
- equivalent to real-time order
- Requires consensus

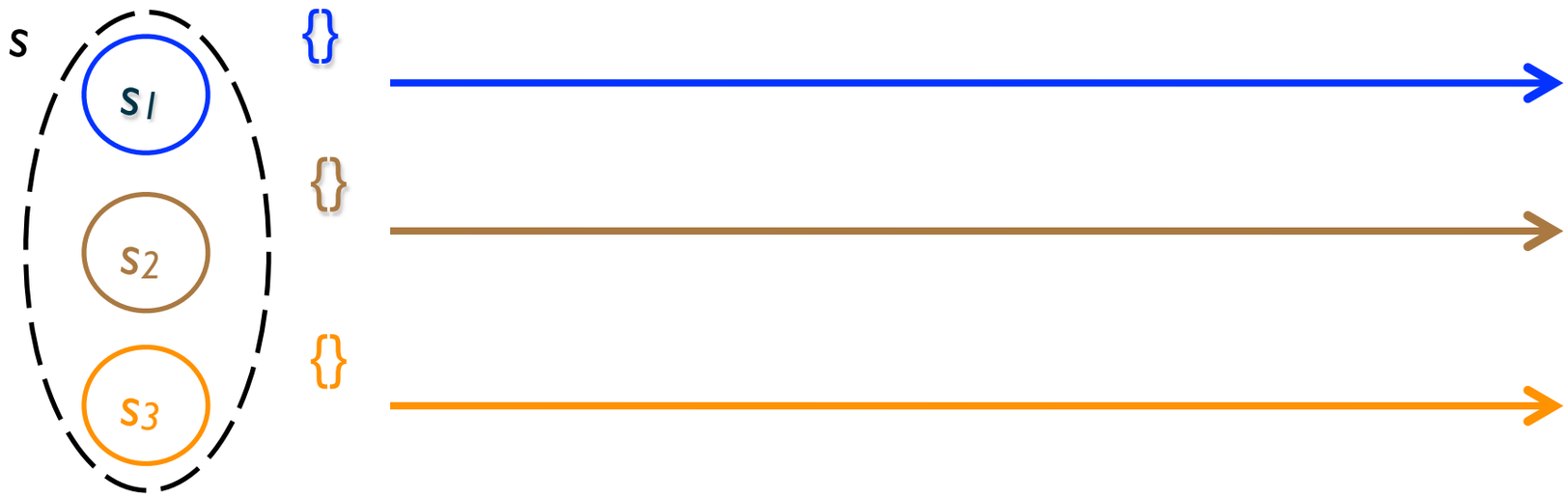
Sequential specification:

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

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

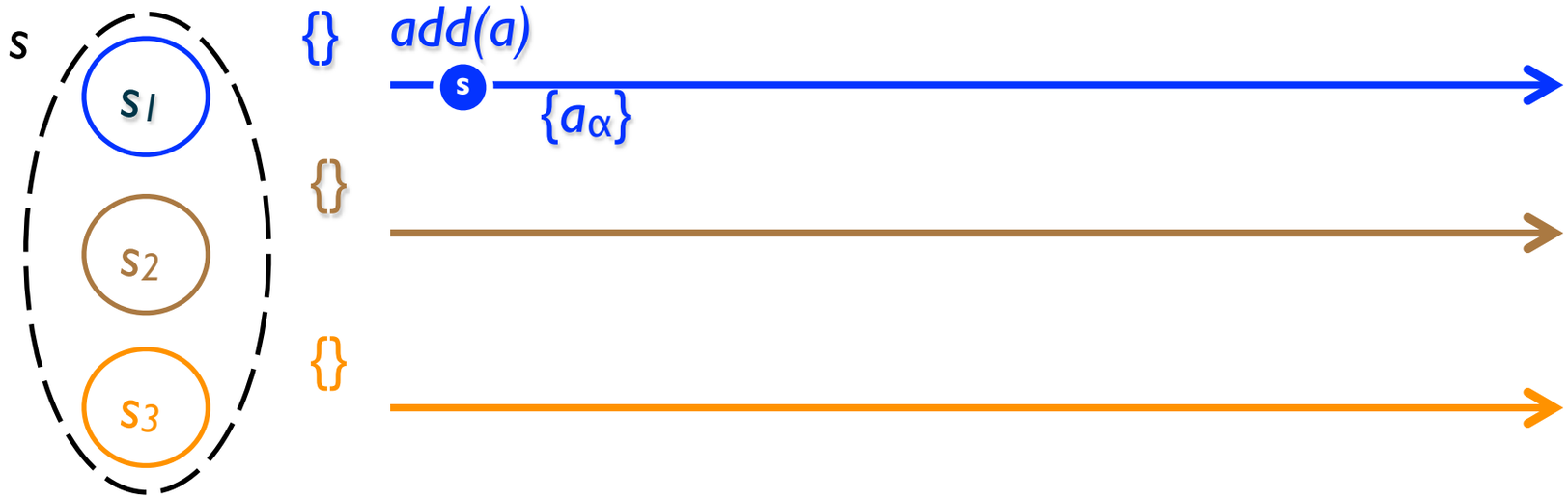
- ~~linearisable?~~
- add wins?
- remove wins?
- last writer wins?
- error state?

Observed-Remove Set



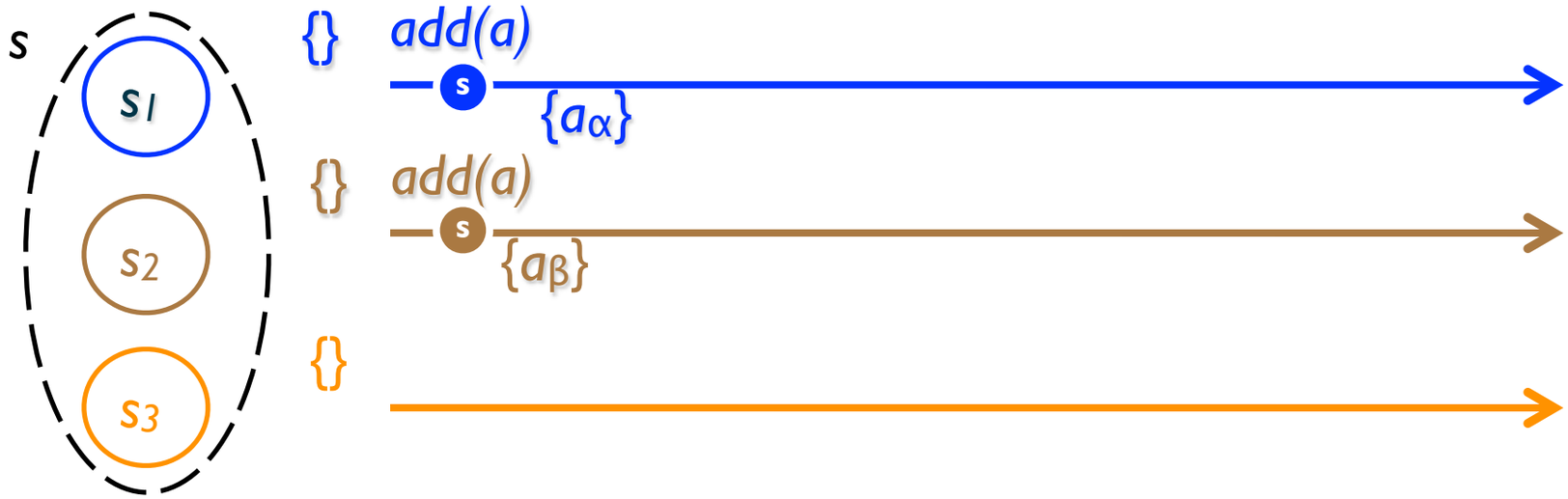
- Can never remove more tokens than exist
- Op order ⇒ removed tokens have been previously added

Observed-Remove Set



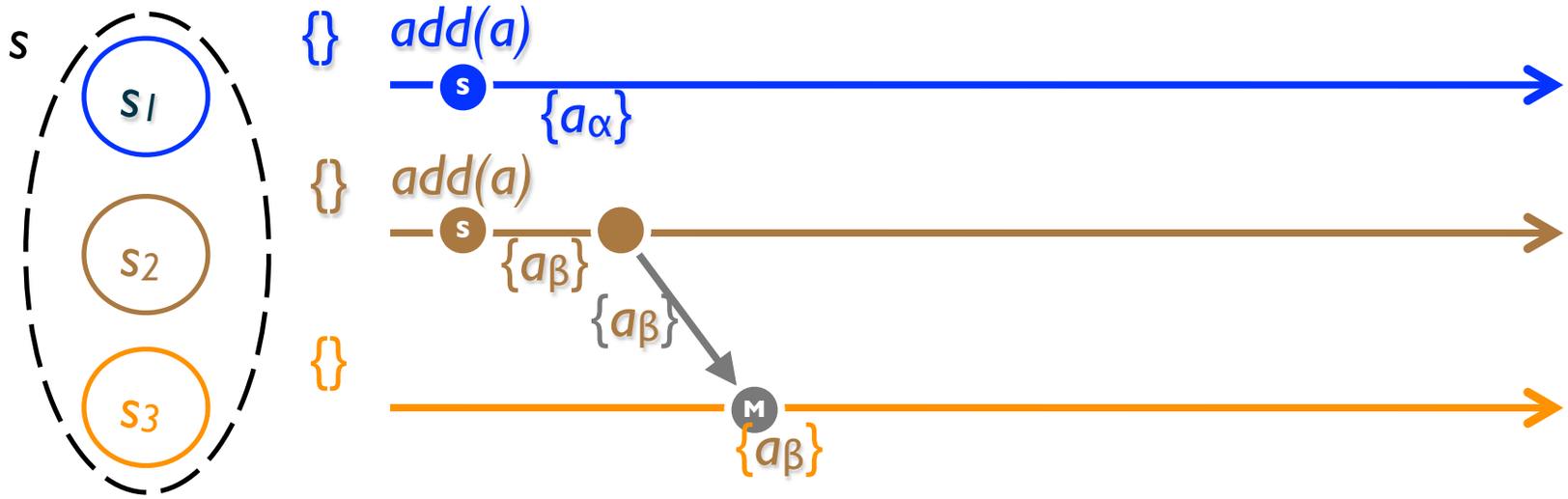
- Can never remove more tokens than exist
- Op order ⇒ removed tokens have been previously added

Observed-Remove Set



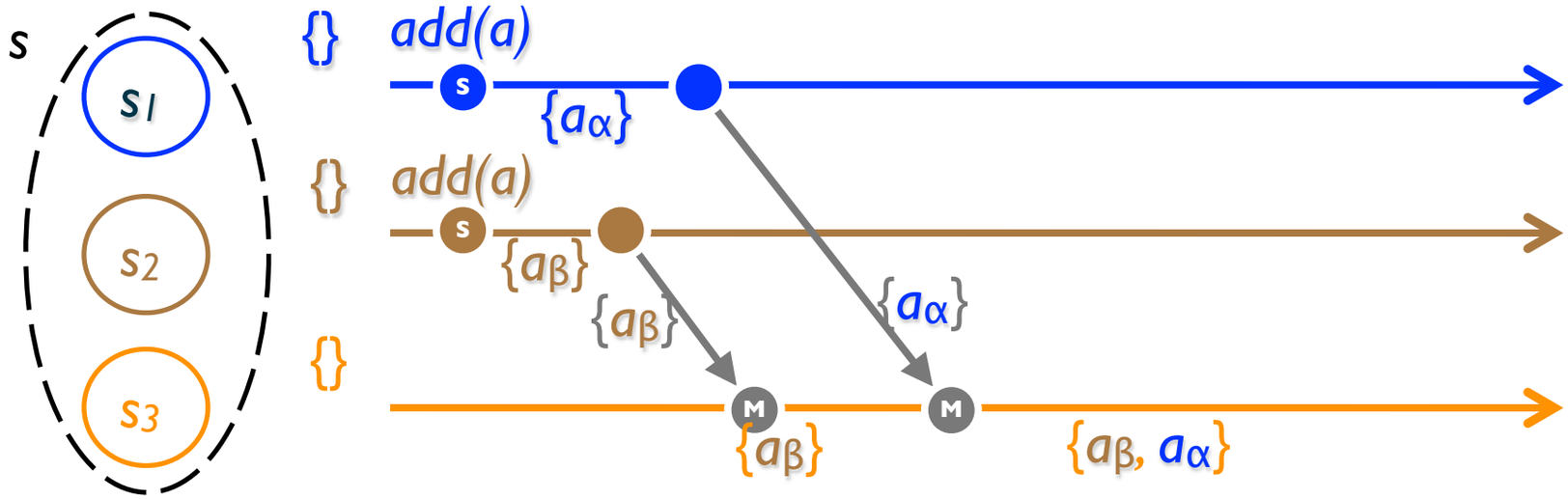
- Can never remove more tokens than exist
- Op order ⇒ removed tokens have been previously added

Observed-Remove Set



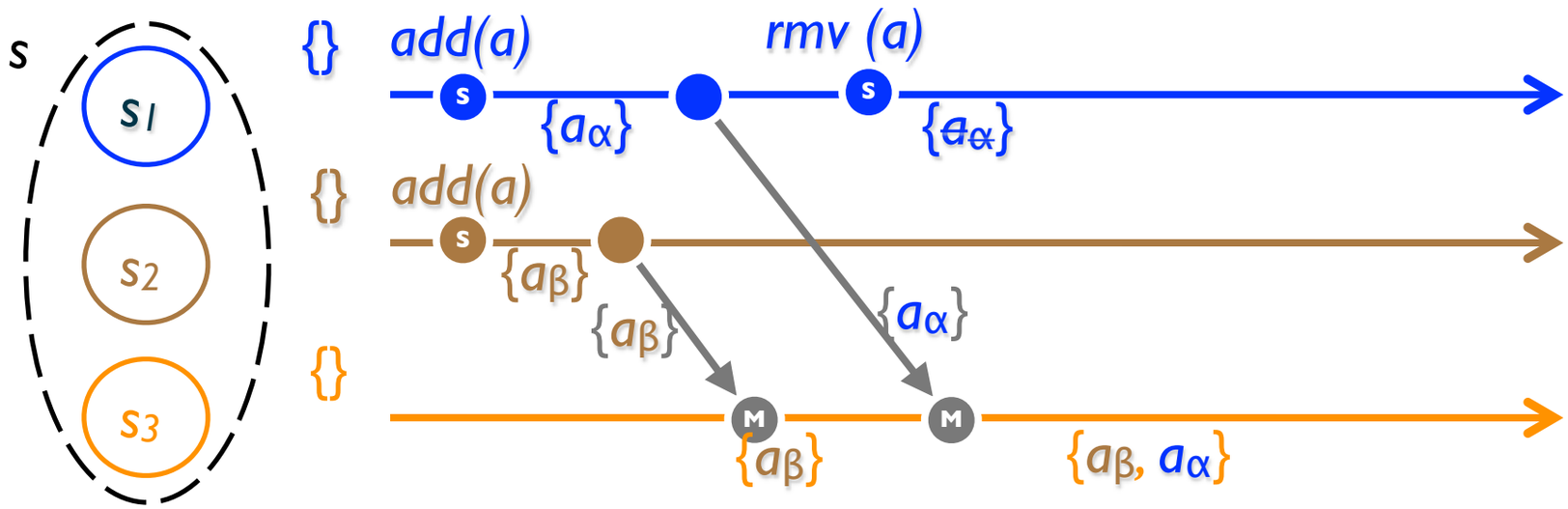
- Can never remove more tokens than exist
- Op order ⇒ removed tokens have been previously added

Observed-Remove Set



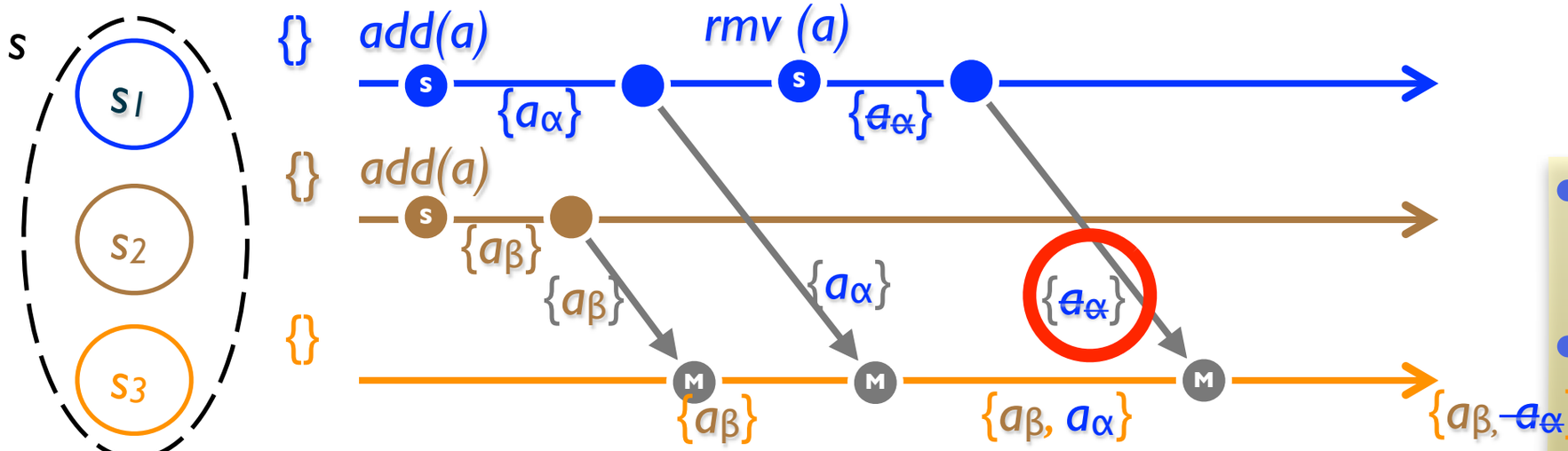
- Can never remove more tokens than exist
- Op order ⇒ removed tokens have been previously added

Observed-Remove Set



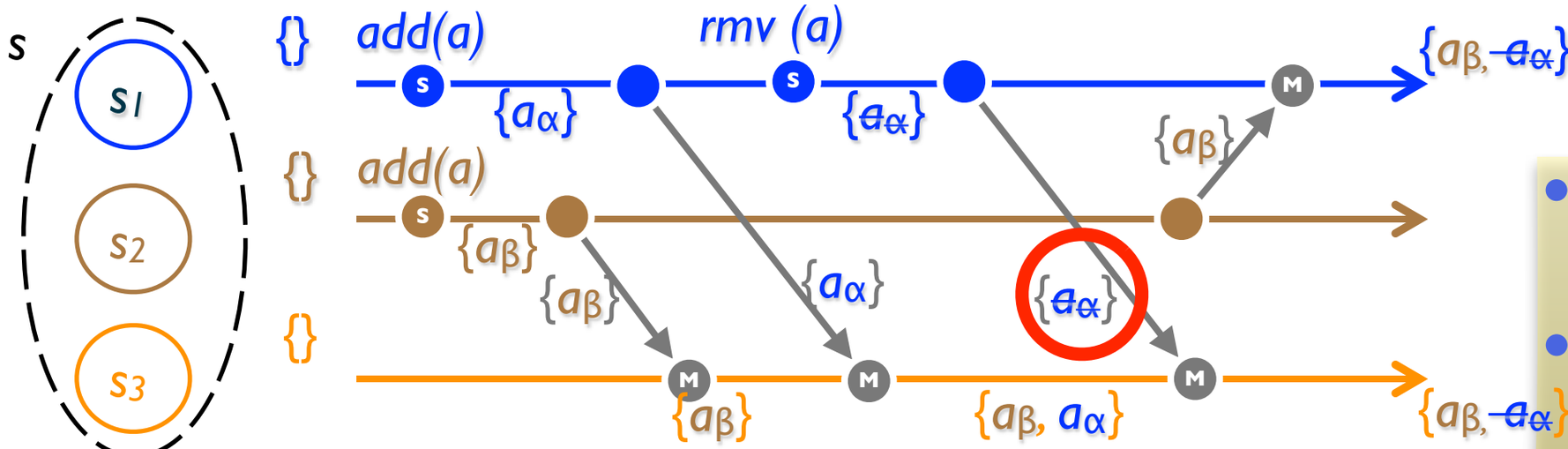
- Can never remove more tokens than exist
- Op order ⇒ removed tokens have been previously added

Observed-Remove Set



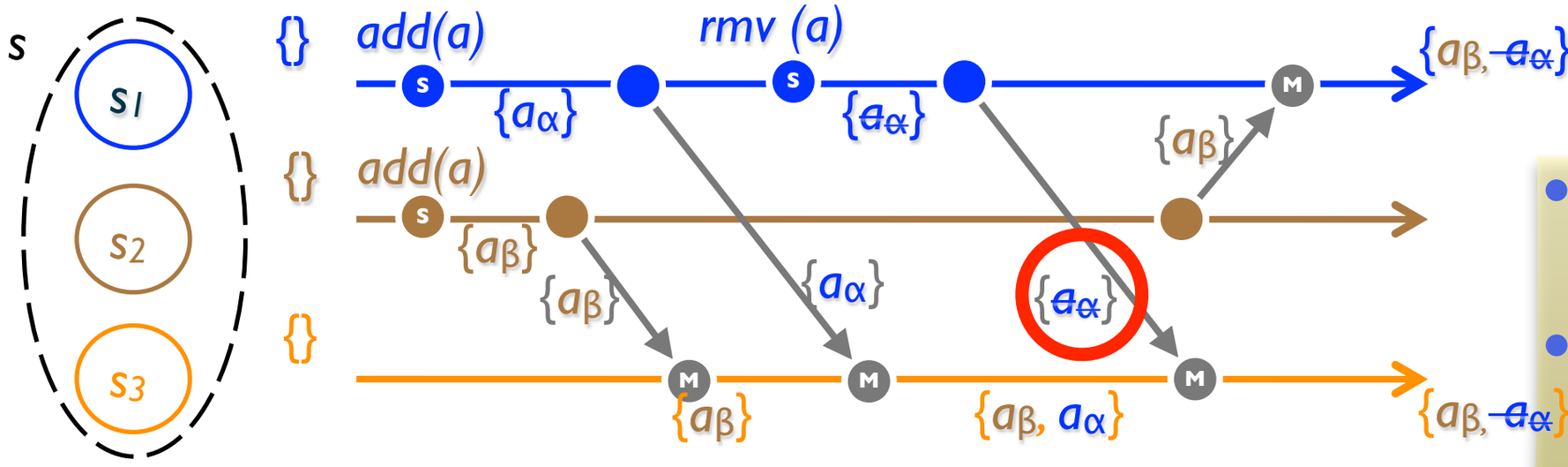
- Can never remove more tokens than exist
- Op order ⇒ removed tokens have been previously added

Observed-Remove Set



- Can never remove more tokens than exist
- Op order ⇒ removed tokens have been previously added

Observed-Remove Set



- Payload: added, removed (*element, unique-token*)
  add(e) = $A := A \cup \{(e, \alpha)\}$
- Remove: all unique elements observed
  remove(e) = $R := R \cup \{ (e, –) \in A\}$
- *lookup*(e) = $\exists\ (e, –) \in A \setminus R$
- *merge* $(S, S') = (A \cup A', R \cup R')$
- $\{true\}$ add(e) || remove(e) $\{e \in S\}$

OR-Set

Set: solves Dynamo Shopping Cart anomaly

Optimisations

- Just mark tombstones
- Garbage-collect tombstones
- Operation-based approach

Graph design alternatives

Graph = (V, E) where $E \subseteq V \times V$

Sequential specification:

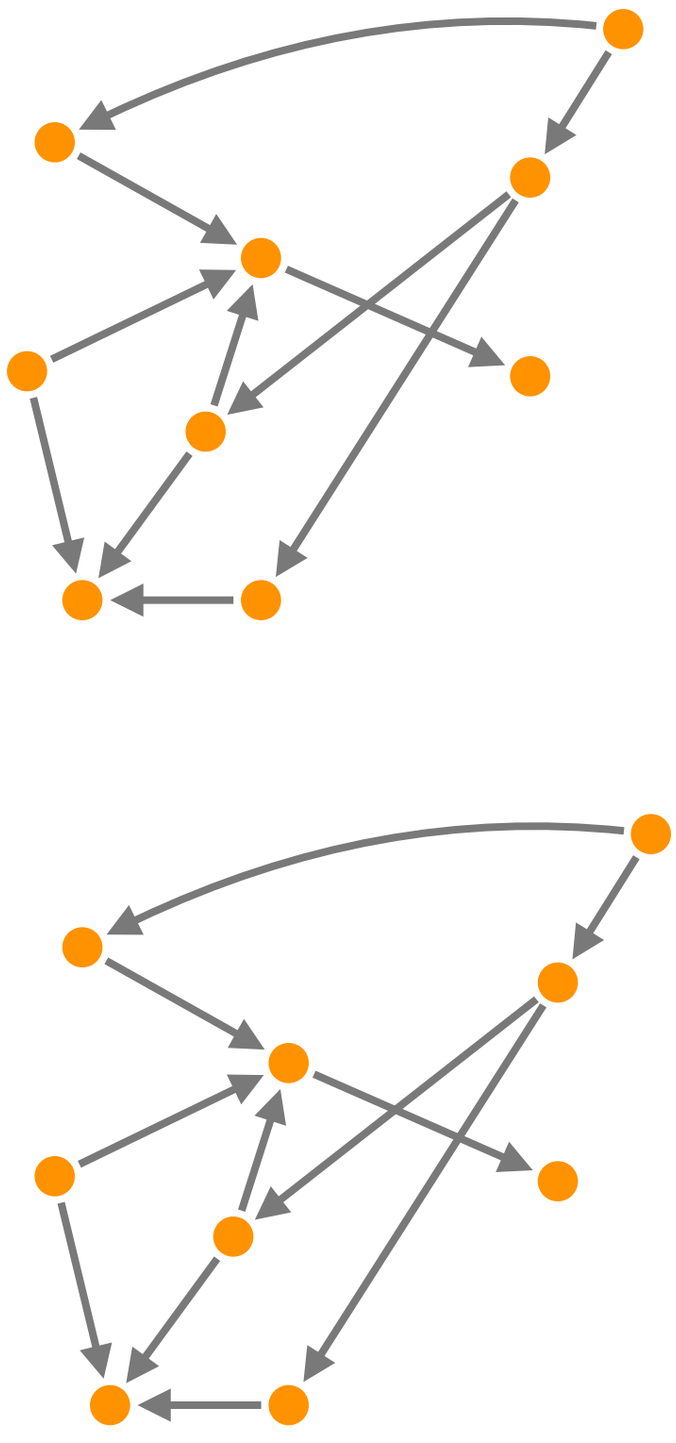
- $\{v,v' \in V\}$ addEdge(v,v') $\{\ldots\}$
- $\{\nexists(v,v') \in E\}$ removeVertex(v) $\{\ldots\}$

Concurrent: removeVertex(v') || addEdge(v,v')

- ~~linearisable?~~
- addEdge wins?
- removeVertex wins?
- etc.

- for our Web Search Engine application, removeVertex wins
- Do not check precondition at add/remove

Graph



Payload = OR-Set *V*, OR-Set *E*

Updates add/remove to *V*, *E*

- *addVertex(v)*, *removeVertex(v)*
- *addEdge(v,v')*, *removeEdge(v,v')*

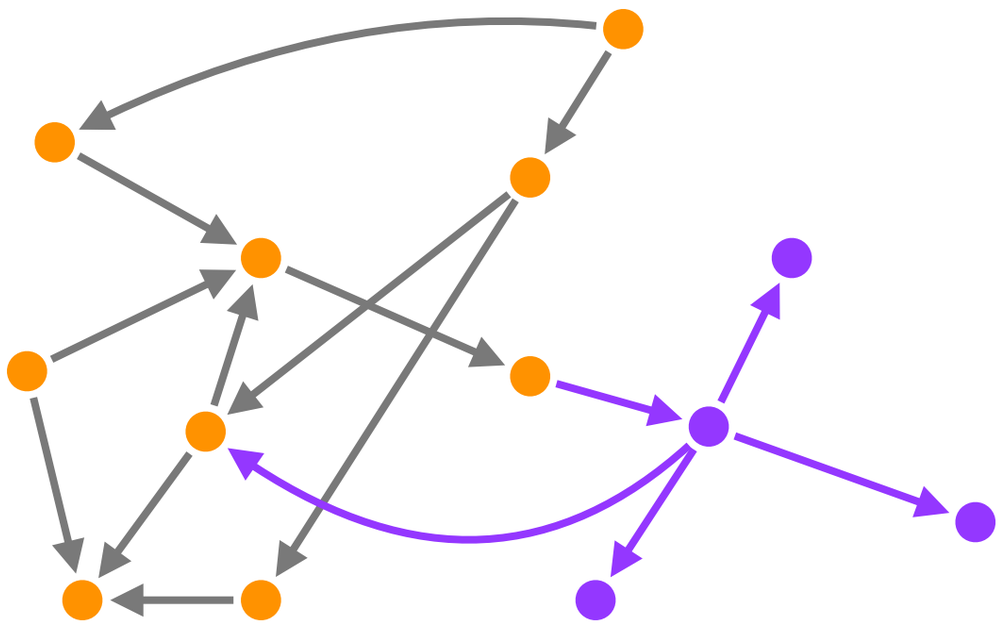
Do not enforce invariant a priori

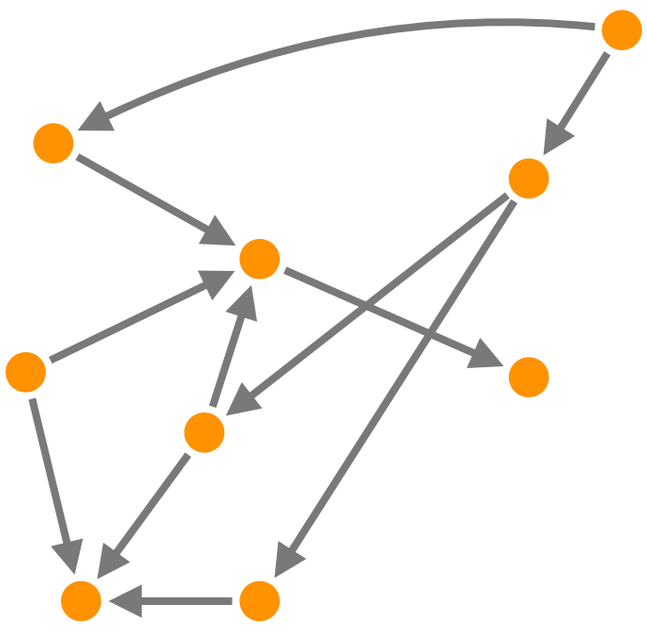
- lookupEdge(*v*,*v'*) = $(v,v') \in E \wedge v \in V \wedge v' \in V$

removeVertex(v') || *addEdge(v,v')*

- remove wins

Graph





Payload = OR-Set *V*, OR-Set *E*

Updates add/remove to *V, E*

- *addVertex(v), removeVertex(v)*
- *addEdge(v,v'), removeEdge(v,v')*

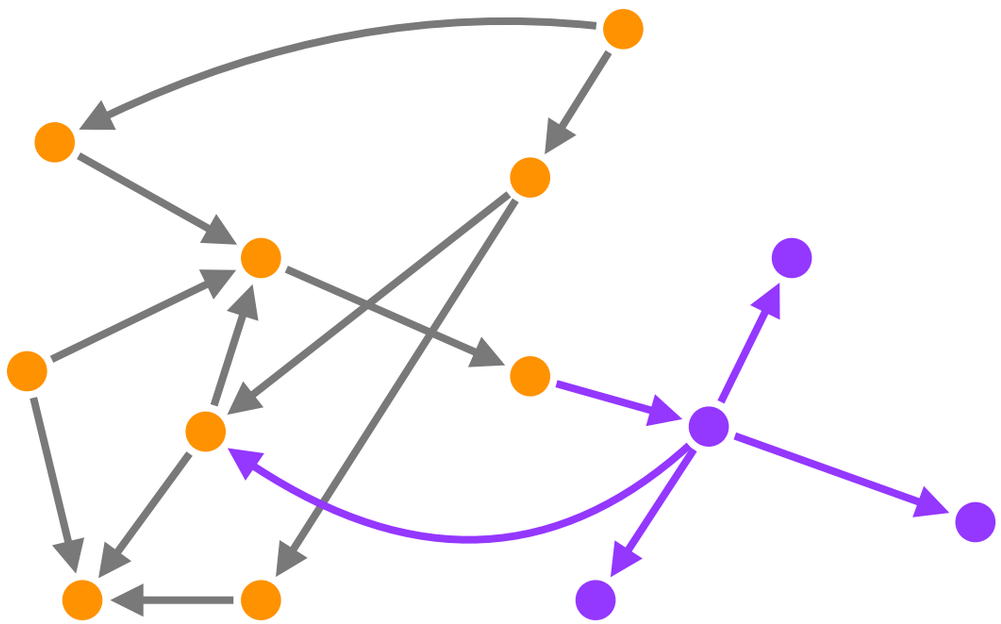
Do not enforce invariant a priori

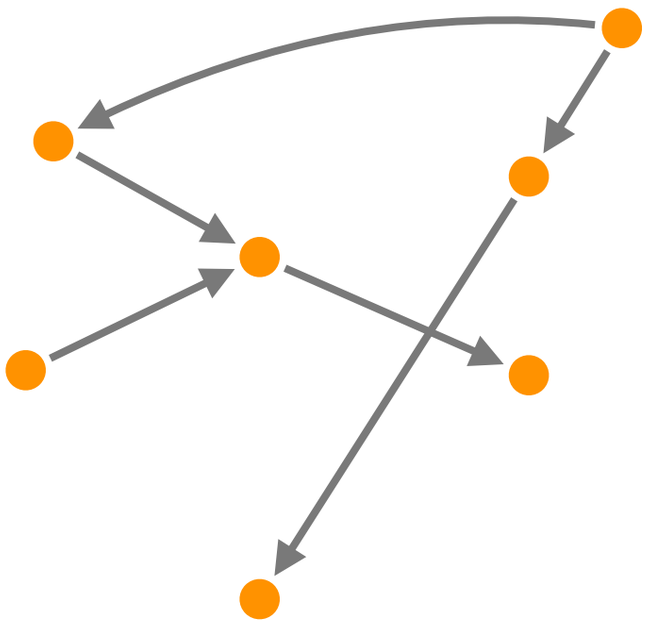
- lookupEdge(*v,v'*) = $(v,v') \in E \wedge v \in V \wedge v' \in V$

removeVertex(v') || *addEdge(v,v')*

- remove wins

Graph





Payload = OR-Set *V*, OR-Set *E*

Updates add/remove to *V, E*

- *addVertex(v), removeVertex(v)*
- *addEdge(v,v'), removeEdge(v,v')*

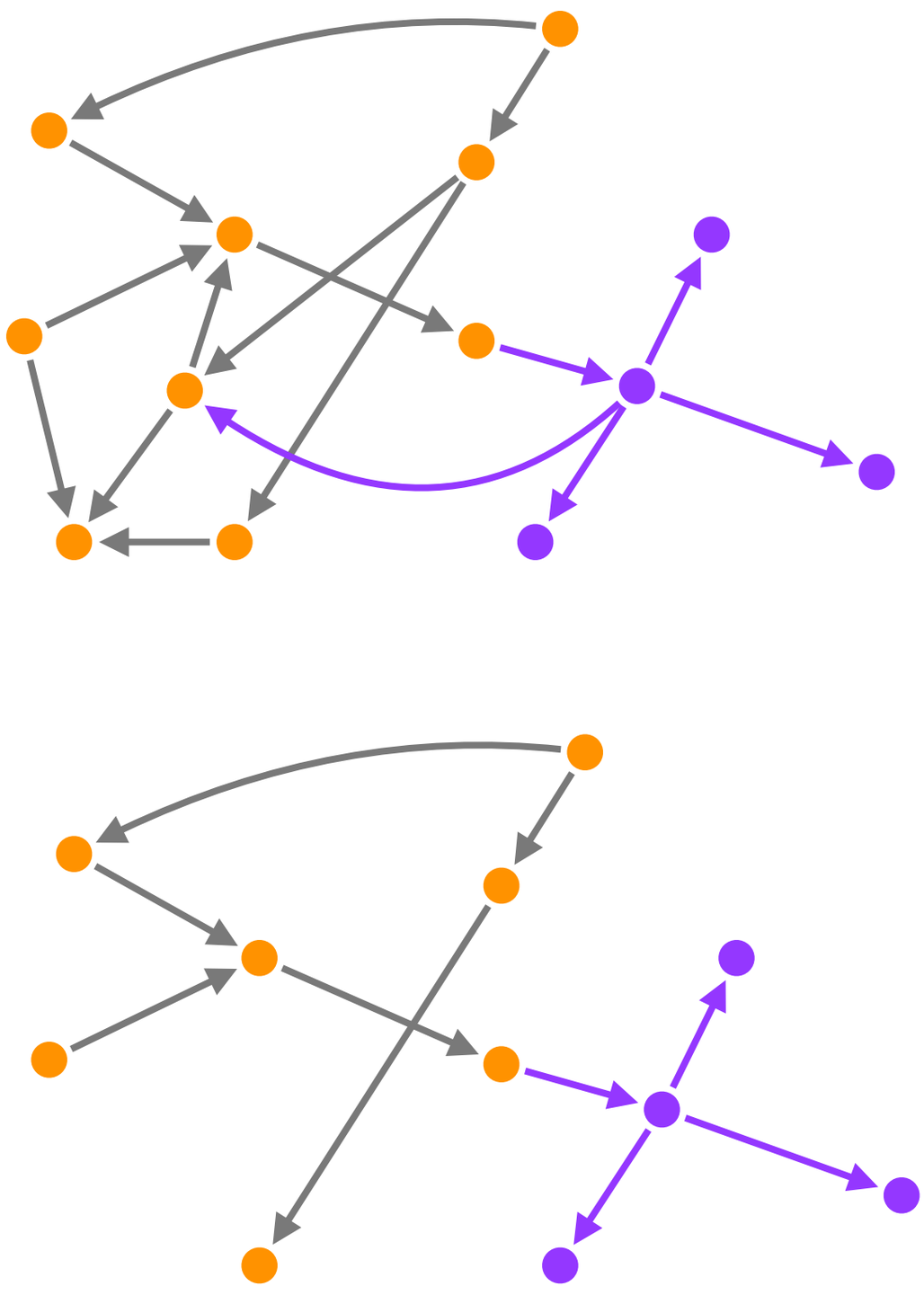
Do not enforce invariant a priori

- lookupEdge(*v,v'*) = $(v,v') \in E \wedge v \in V \wedge v' \in V$

removeVertex(v') || *addEdge(v,v')*

- remove wins

Graph



Payload = OR-Set *V*, OR-Set *E*

Updates add/remove to *V*, *E*

- *addVertex(v)*, *removeVertex(v)*
- *addEdge(v,v')*, *removeEdge(v,v')*

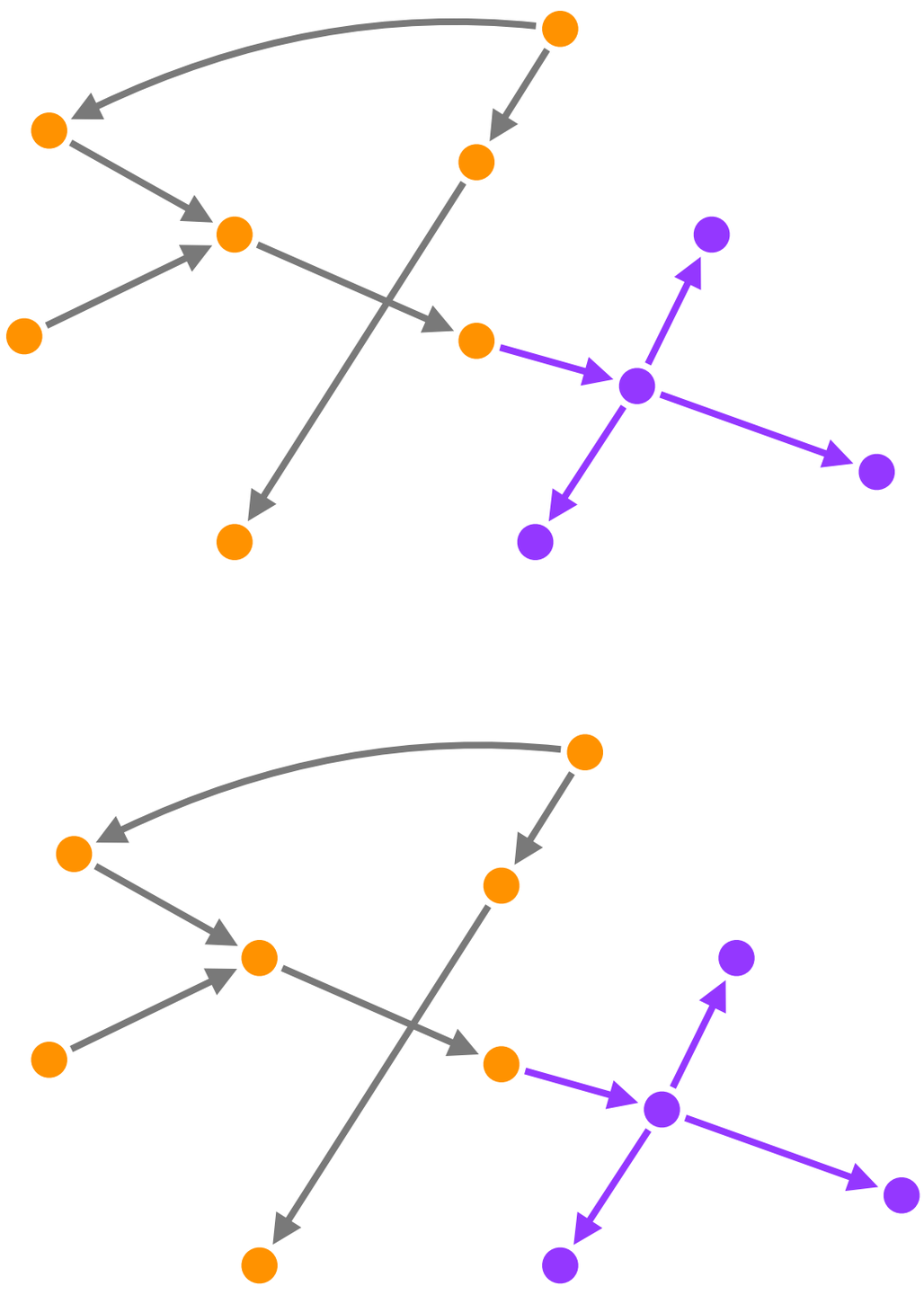
Do not enforce invariant a priori

- lookupEdge(*v*,*v'*) = $(v,v') \in E \wedge v \in V \wedge v' \in V$

removeVertex(v') || *addEdge(v,v')*

- remove wins

Graph



Payload = OR-Set *V*, OR-Set *E*

Updates add/remove to *V*, *E*

- *addVertex(v)*, *removeVertex(v)*
- *addEdge(v,v')*, *removeEdge(v,v')*

Do not enforce invariant a priori

- lookupEdge(*v*,*v'*) = $(v,v') \in E \wedge v \in V \wedge v' \in V$

removeVertex(v') || *addEdge(v,v')*

- remove wins

Co-operative editing

$\{x,z \in graph_i \wedge x < z\}$

add-between$_i$ *(x, y, z)*

$\{y \in graph_i \wedge x<y<z\}$

- Local constraint implies globally acyclic

- Deep (internal) view:
- DAG representing insert order

- This spec is implemented directly by WOOT
- Clean

- begin and end sentinels

- strong order takes precedence over weak

- add: between already-ordered elements

- Surface view: summarises total order

I	N	R	A

$\vdash \quad \alpha \quad \beta \quad \gamma \quad \varepsilon \quad \dashv$

- ensure consistent ordering at all replicas

Co-operative editing

$\{x,z \in graph_i \wedge x < z\}$

add-between$_i$ *(x, y, z)*

$\{y \in graph_i \wedge x<y<z\}$

- Local constraint implies globally acyclic

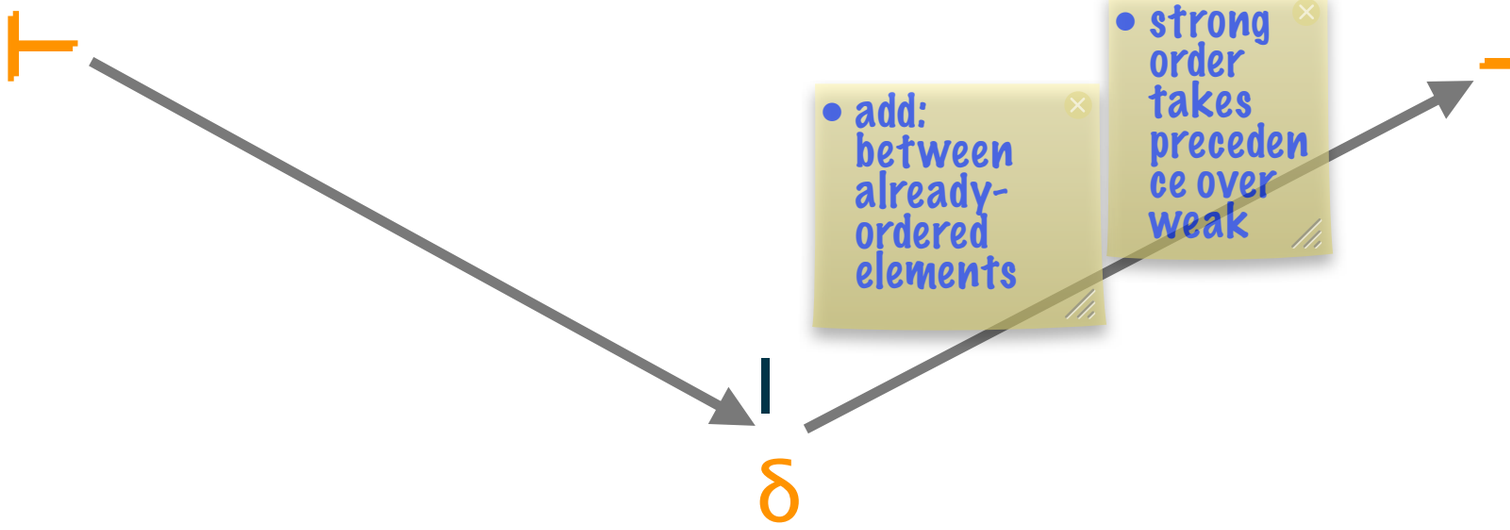
- Deep (internal) view:
- DAG representing insert order

- This spec is implemented directly by WOOT
- Clean

- begin and end sentinels

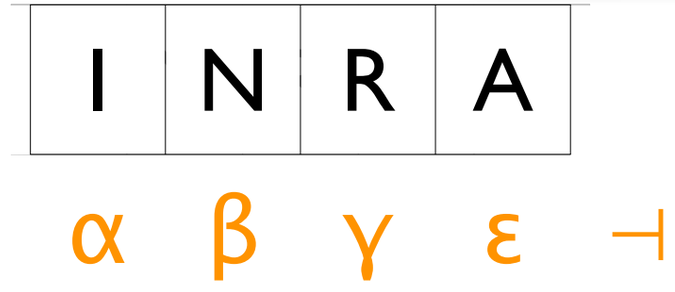
- strong order takes precedence over weak

- add: between already-ordered elements



- ensure consistent ordering at all replicas

- Surface view: summarises total order



Co-operative editing

$\{x,z \in graph_i \wedge x < z\}$

add-between$_i$ *(x, y, z)*

$\{y \in graph_i \wedge x<y<z\}$

- Local constraint implies globally acyclic

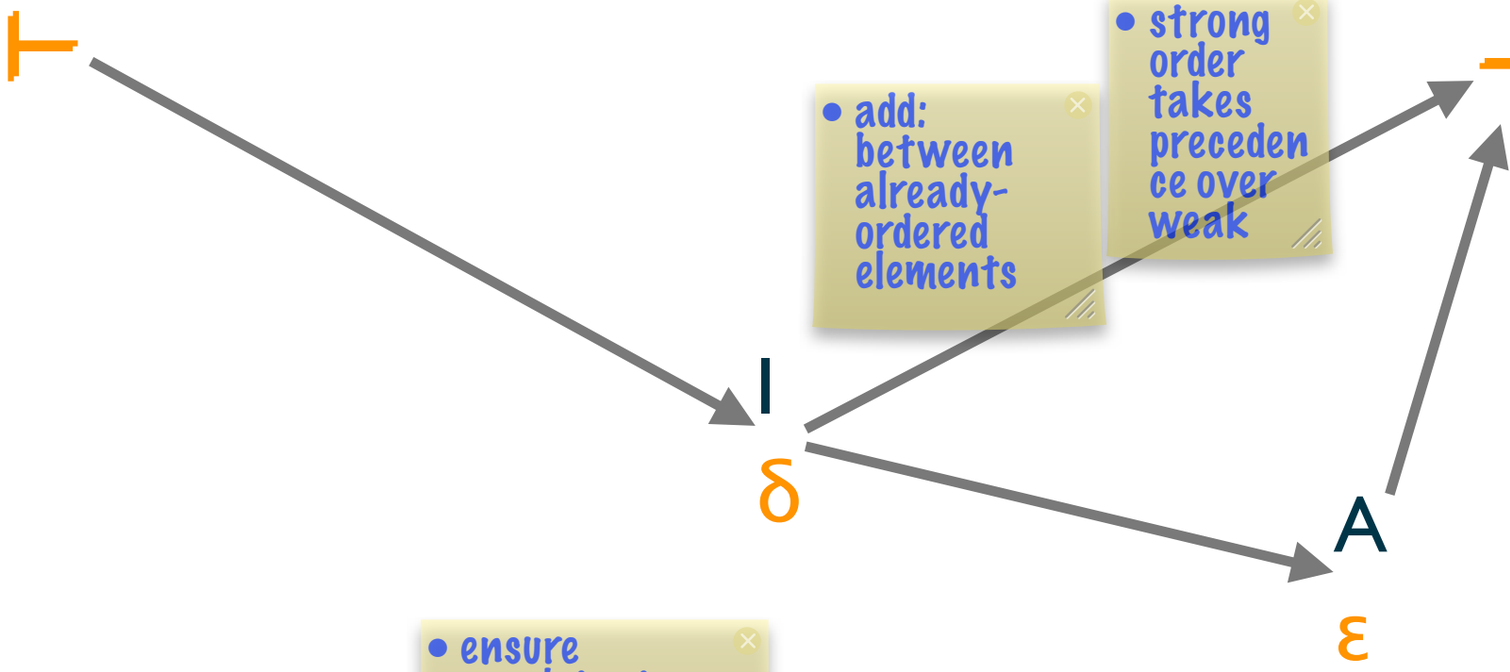
- Deep (internal) view:
- DAG representing insert order

- This spec is implemented directly by WOOT
- Clean

- begin and end sentinels

- strong order takes precedence over weak

- add: between already-ordered elements



- ensure consistent ordering at all replicas

- Surface view: summarises total order

I	N	R	A

⊢ α β γ ε ⊣

Co-operative editing

$\{x,z \in graph_i \wedge x < z\}$

add-between$_i$ *(x, y, z)*

$\{y \in graph_i \wedge x<y<z\}$

- Local constraint implies globally acyclic

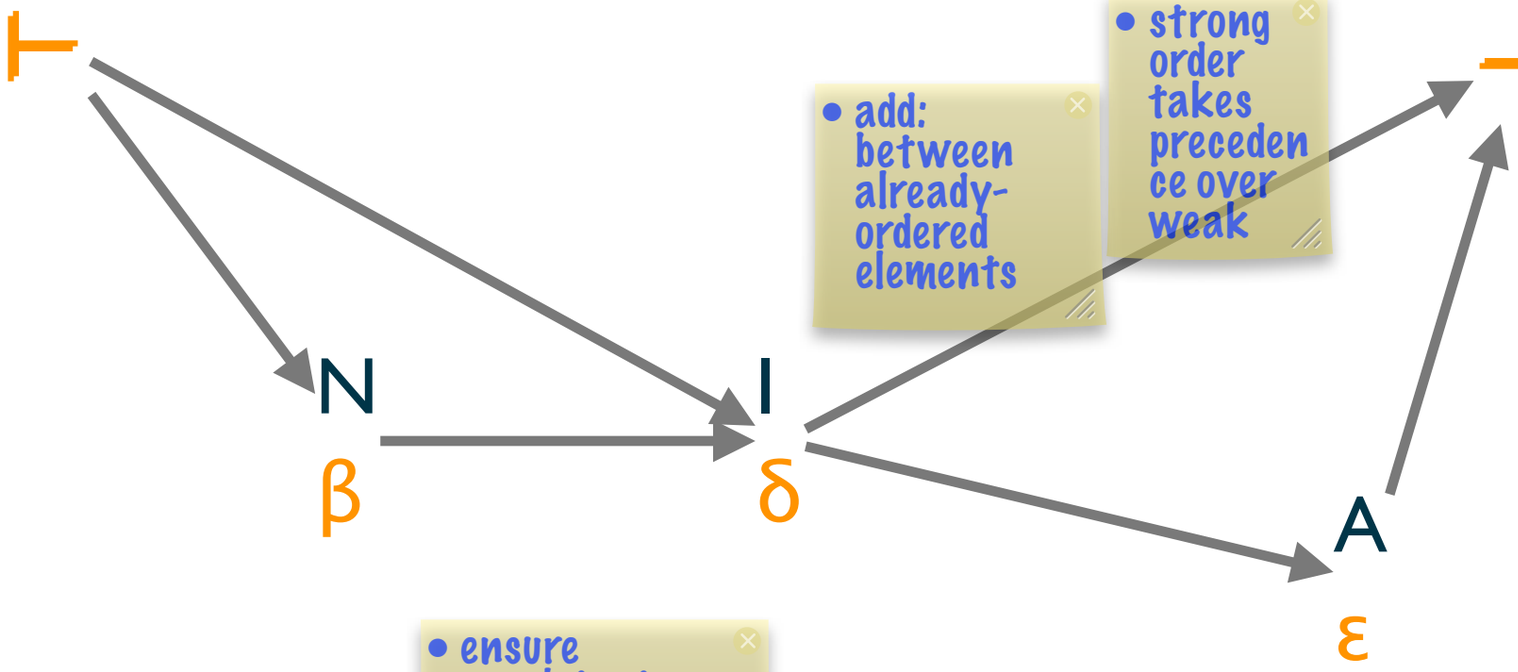
- Deep (internal) view:
- DAG representing insert order

- This spec is implemented directly by WOOT
- Clean

- begin and end sentinels

- strong order takes precedence over weak

- add: between already-ordered elements



- ensure consistent ordering at all replicas

- Surface view: summarises total order

I	N	R	A

⊢ α β γ ε ⊣

Co-operative editing

$$\{x,z \in graph_i \wedge x < z\}$$

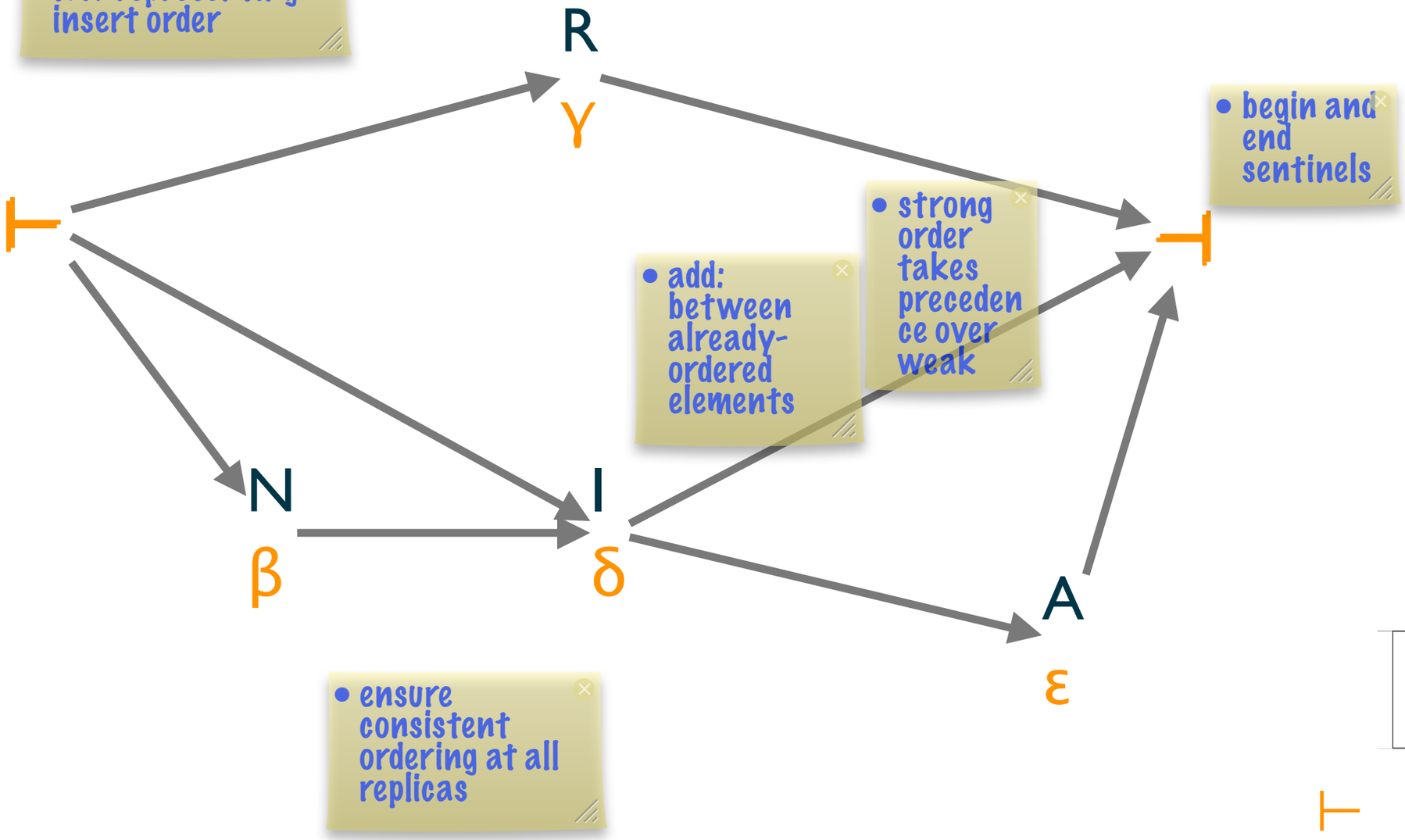
$$add\text{-}between_i\ (x, y, z)$$

$$\{y \in graph_i \wedge x<y<z\}$$

- Local constraint implies globally acyclic

- Deep (internal) view:
- DAG representing insert order

- This spec is implemented directly by WOOT
- Clean



- begin and end sentinels
- strong order takes precedence over weak
- add: between already-ordered elements
- ensure consistent ordering at all replicas

- Surface view: summarises total order

I	N	R	A

⊢ α β γ ε ⊣

Co-operative editing

$$\{x,z \in graph_i \wedge x < z\}$$

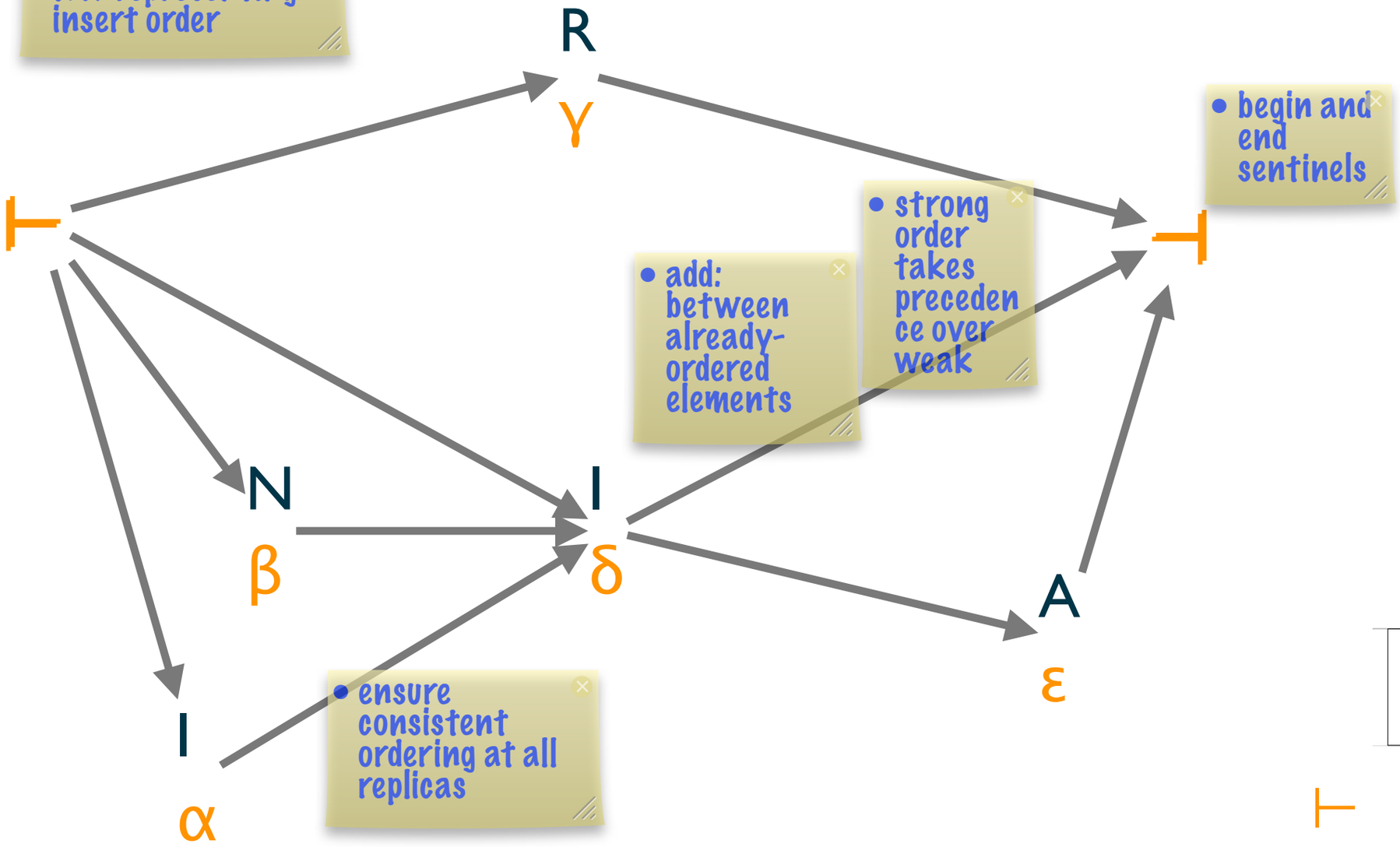
$$add\text{-}between_i\ (x, y, z)$$

$$\{y \in graph_i \wedge x<y<z\}$$

- Local constraint implies globally acyclic

- Deep (internal) view:
- DAG representing insert order

- This spec is implemented directly by WOOT
- Clean



- Surface view: summarises total order

I	N	R	A

⊢ α β γ ε ⊣

Co-operative editing

$\{x,z \in graph_i \wedge x < z\}$

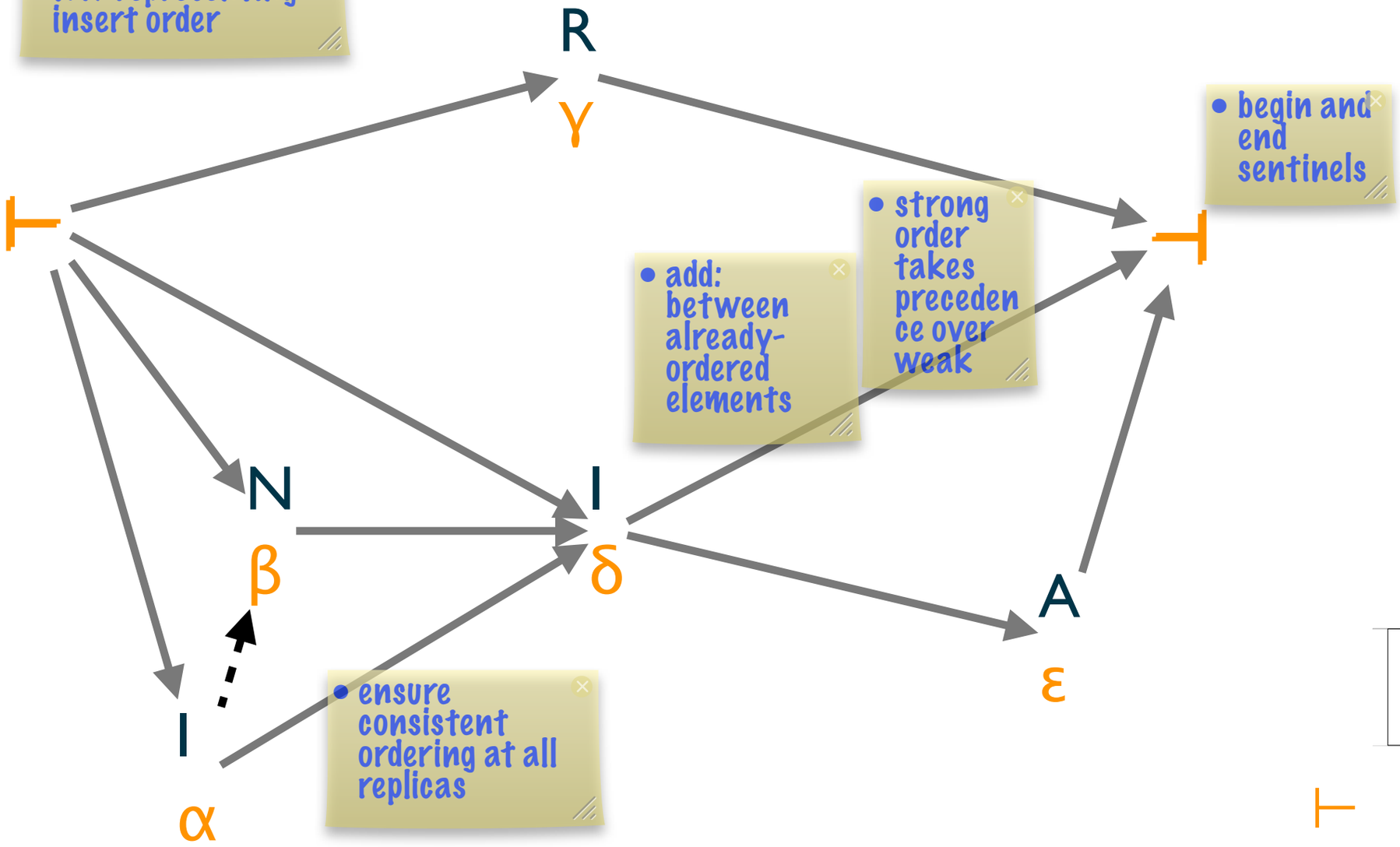
add-between$_i$ *(x, y, z)*

$\{y \in graph_i \wedge x<y<z\}$

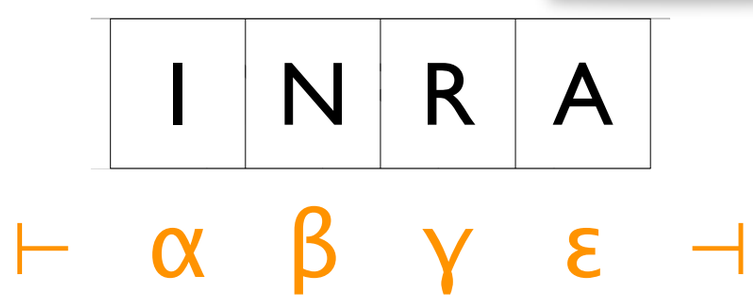
- Local constraint implies globally acyclic

- Deep (internal) view:
- DAG representing insert order

- This spec is implemented directly by WOOT
- Clean



- Surface view: summarises total order



Co-operative editing

$$\{x,z \in graph_i \wedge x < z\}$$

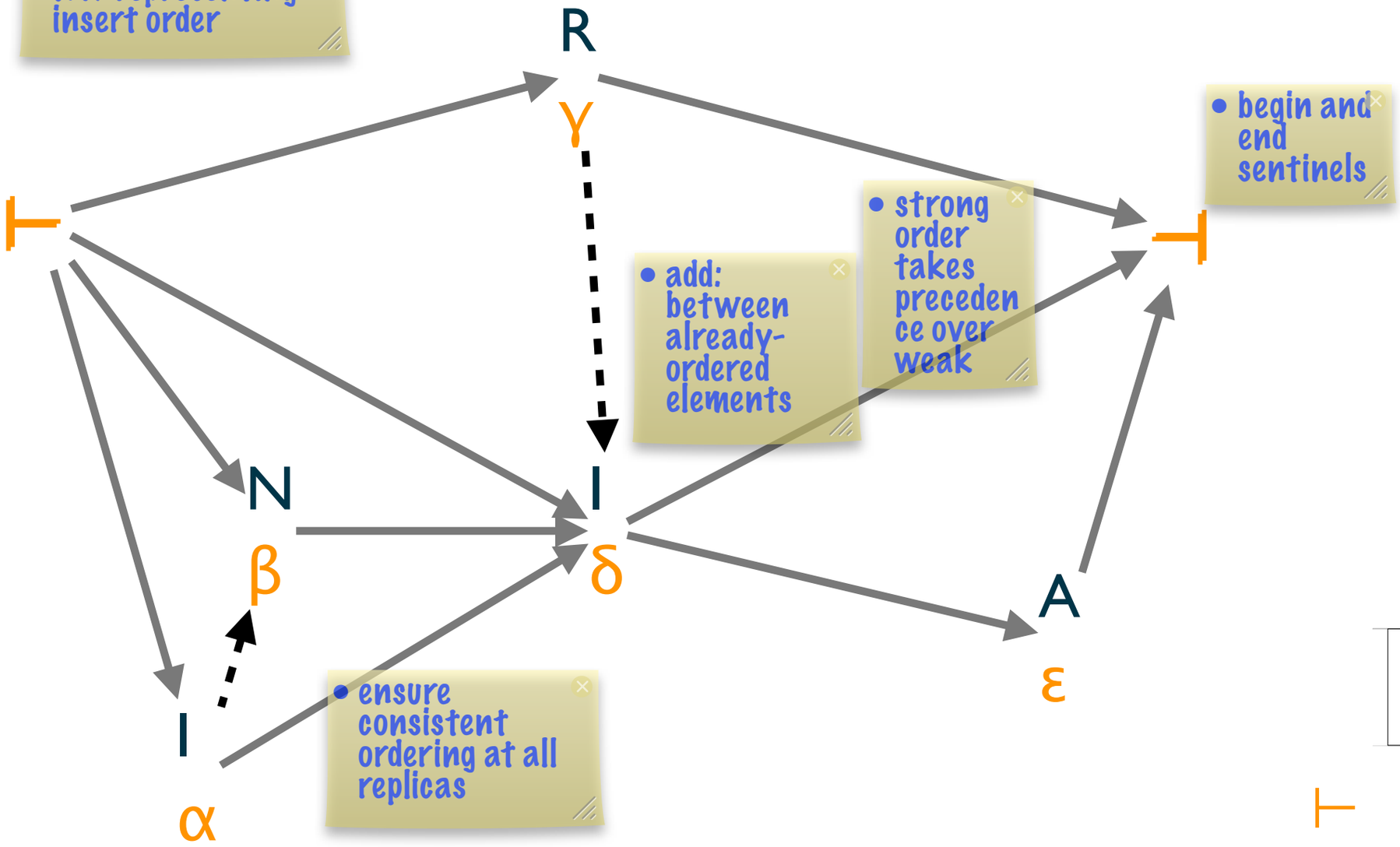
$$add\text{-}between_i\ (x, y, z)$$

$$\{y \in graph_i \wedge x<y<z\}$$

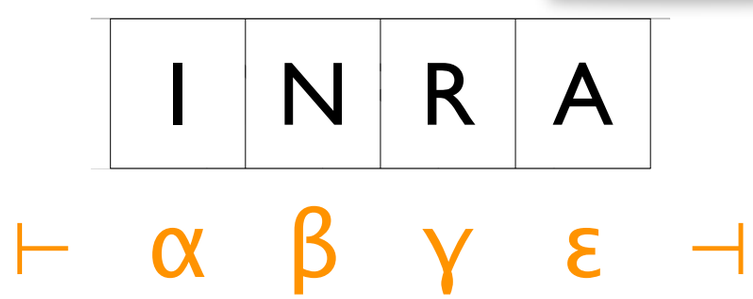
- Local constraint implies globally acyclic

- Deep (internal) view:
- DAG representing insert order

- This spec is implemented directly by WOOT
- Clean



- Surface view: summarises total order



Co-operative editing

$$\{x,z \in graph_i \wedge x < z\}$$

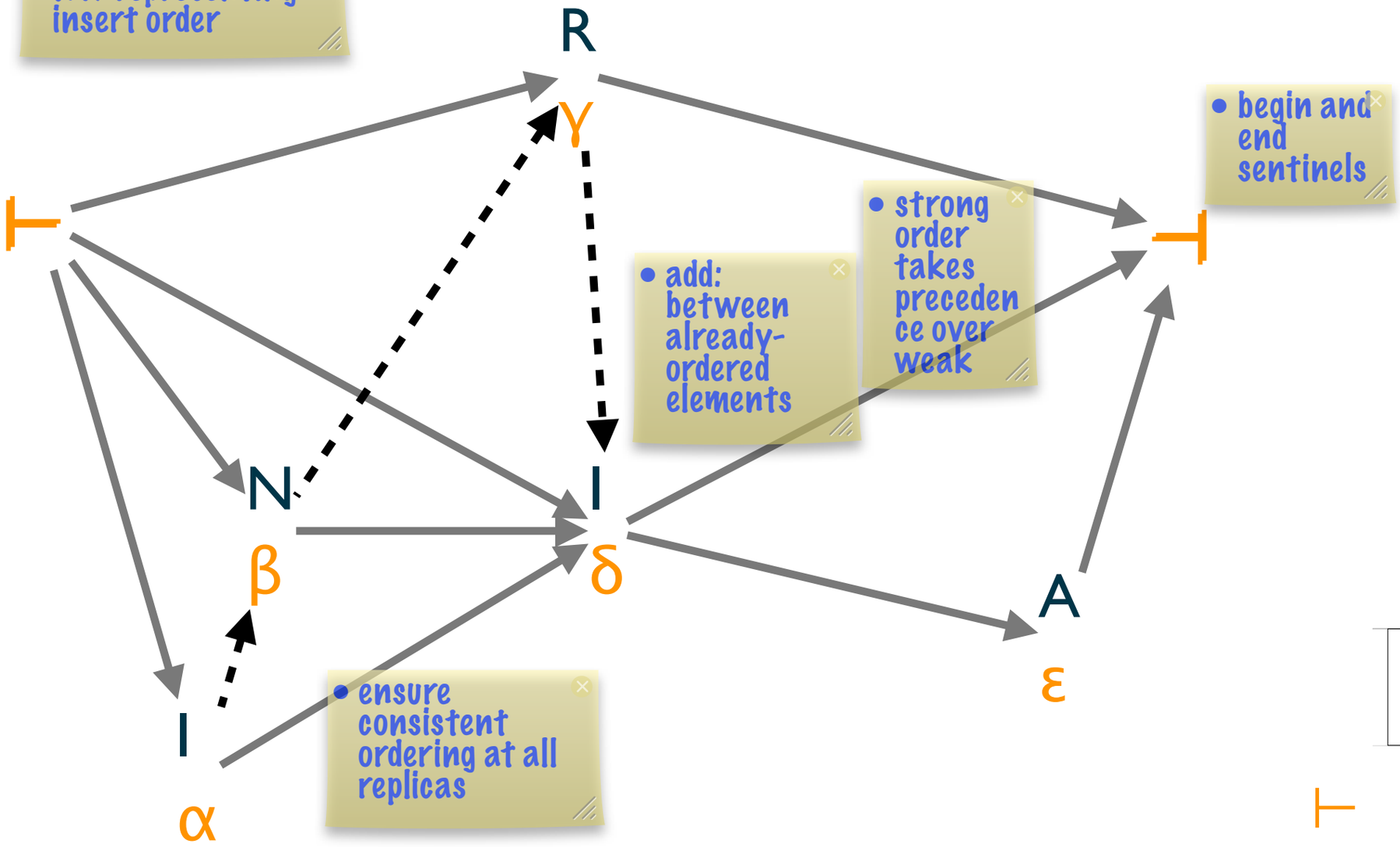
$$add\text{-}between_i\ (x, y, z)$$

$$\{y \in graph_i \wedge x<y<z\}$$

- Local constraint implies globally acyclic

- Deep (internal) view:
- DAG representing insert order

- This spec is implemented directly by WOOT
- Clean



- Surface view: summarises total order

I	N	R	A

⊢ α β γ ε ⊣

Co-operative editing

$\{x,z \in graph_i \wedge x < z\}$

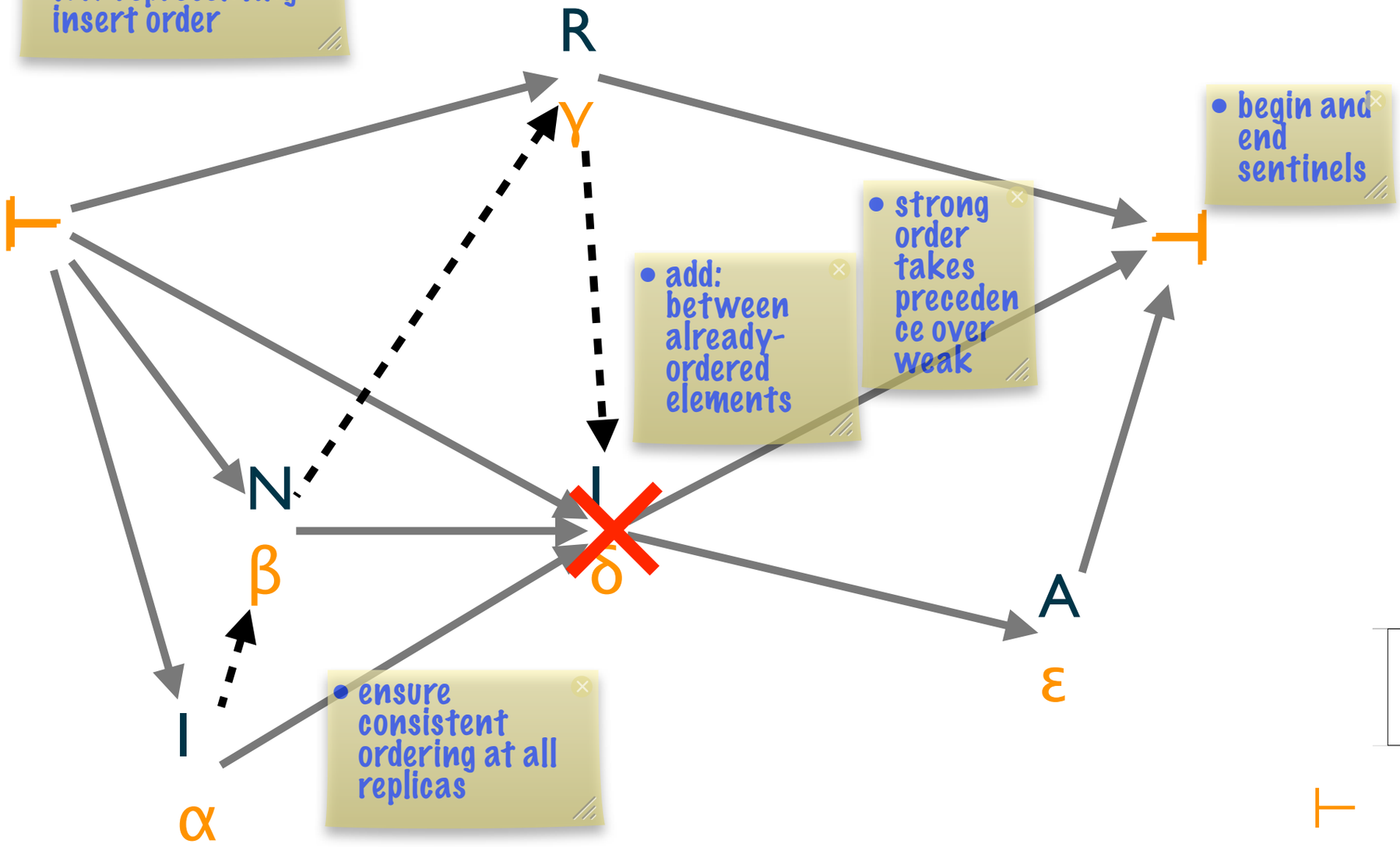
add-between$_i$ *(x, y, z)*

$\{y \in graph_i \wedge x<y<z\}$

- Local constraint implies globally acyclic

- Deep (internal) view:
- DAG representing insert order

- This spec is implemented directly by WOOT
- Clean



- Surface view: summarises total order

⊢	α	β	γ	ε	⊣

Continuum

Assign each element a unique real number

- *position*

Real numbers not appropriate

- approximate by tree

Continuum

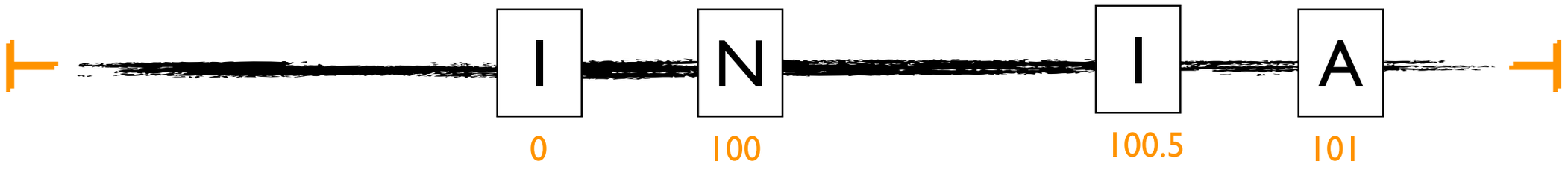
Assign each element a unique real number

- *position*

Real numbers not appropriate

- approximate by tree

Continuum



Assign each element a unique real number

- *position*

Real numbers not appropriate

- approximate by tree

Continuum

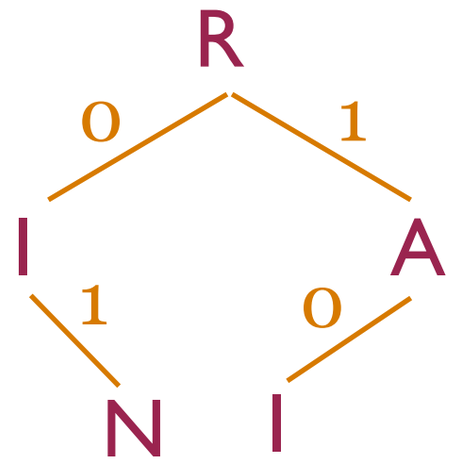
Assign each element a unique real number

- *position*

Real numbers not appropriate

- approximate by tree

Treedoc binary tree



= L ' I N R I

- Compact, low-arity tree
- In the following slides, will

- Low arity: binary, quaternary

Binary naming tree:

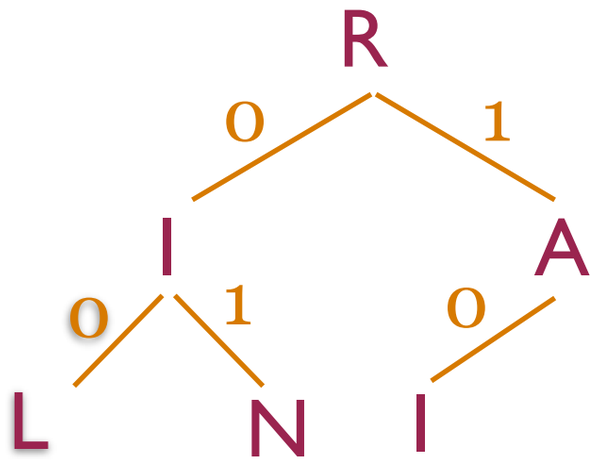
- compact, self-adjusting
- Logarithmic properties

- logarithmic: assuming tree

add appends leaf ⇒ non-destructive, IDs don't change

remove: tombstone, IDs don't change

Treedoc binary tree



= L ’ I N R I

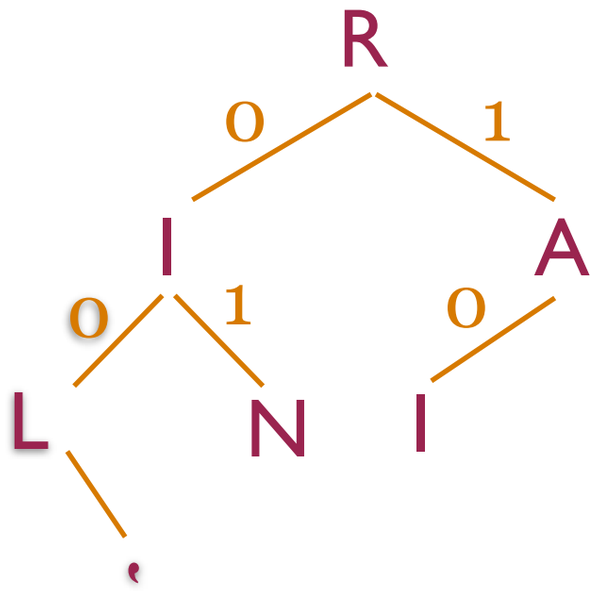
- Binary naming tree:
  - compact, self-adjusting
  - Logarithmic properties
- *add* appends leaf ⇒ non-destructive, IDs don’t change
- *remove*: tombstone, IDs don't change

- Compact, low-arity tree
- In the following slides, will

- Low arity: binary, quaternary

- logarithmic: assuming tree

Treedoc binary tree



= L ’ I N R I

- Compact, low-arity tree
- In the following slides, will

- Low arity: binary, quaternary

Binary naming tree:

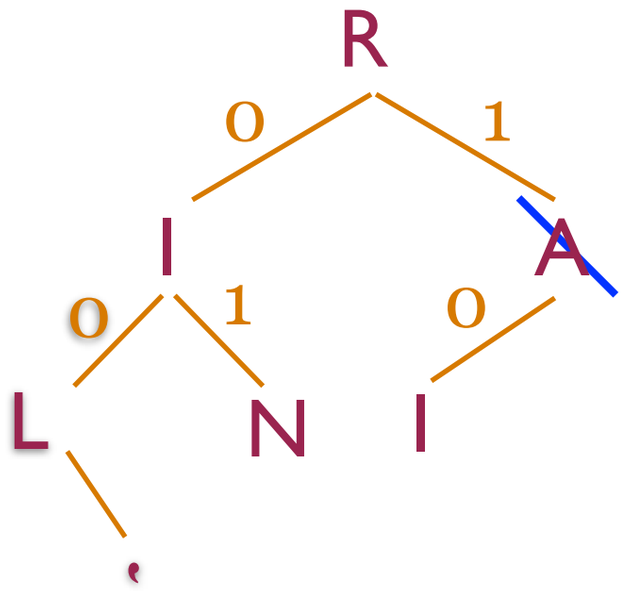
- compact, self-adjusting
- Logarithmic properties

- logarithmic: assuming tree

add appends leaf ⇒ non-destructive, IDs don’t change

remove: tombstone, IDs don't change

Treedoc binary tree



= L ' I N R I

Binary naming tree:

- compact, self-adjusting
- Logarithmic properties

add appends leaf ⇒ non-destructive, IDs don't change

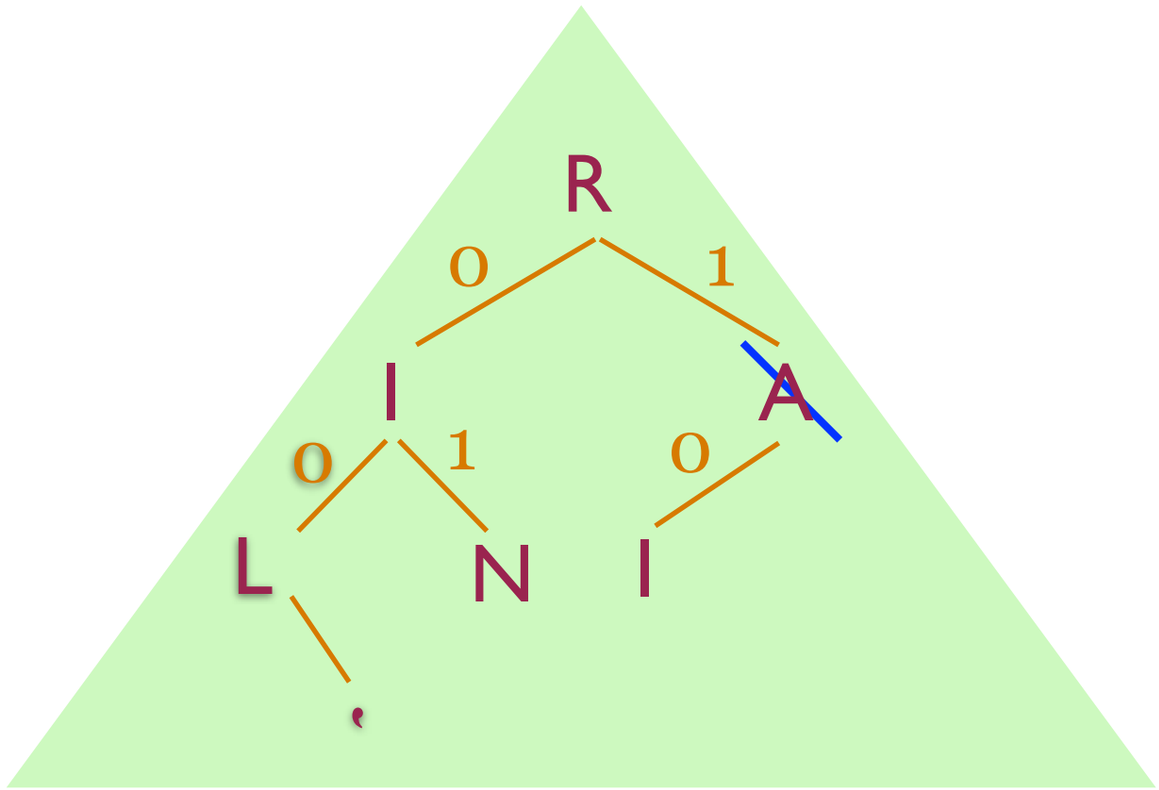
remove: tombstone, IDs don't change

- Compact, low-arity tree
- In the following slides, will

- Low arity: binary, quaternary

- logarithmic: assuming tree

Treedoc binary tree



= L ' I N R I

- Compact, low-arity tree
- In the following slides, will

- Low arity: binary, quaternary

Binary naming tree:

- compact, self-adjusting
- Logarithmic properties

- logarithmic: assuming tree

add appends leaf ⇒ non-destructive, IDs don't change

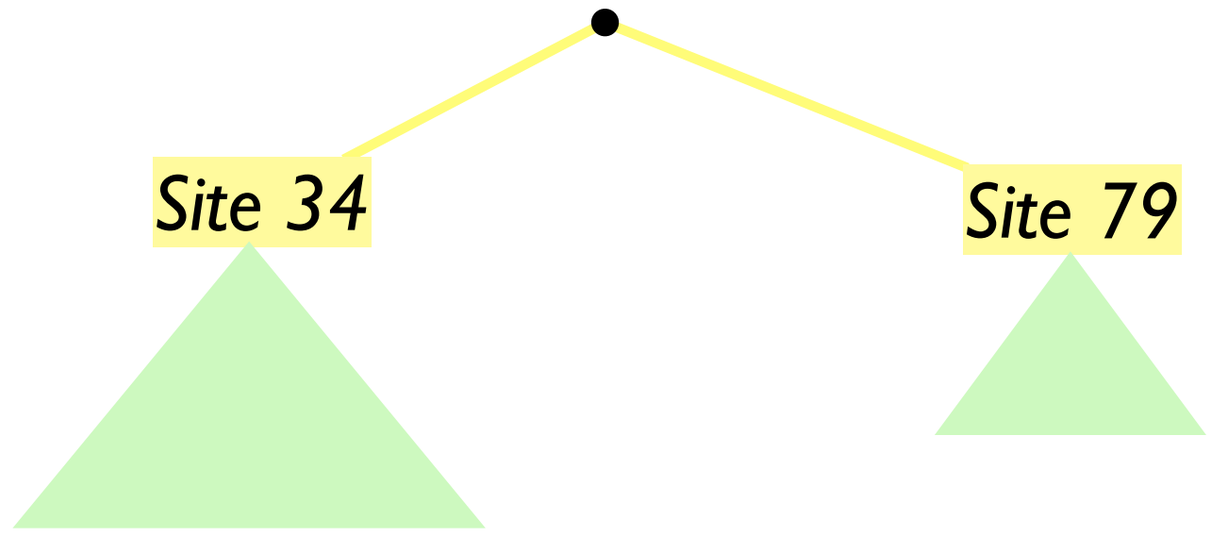
remove: tombstone, IDs don't change

Layered Treedoc

binary tree

Layered Treedoc

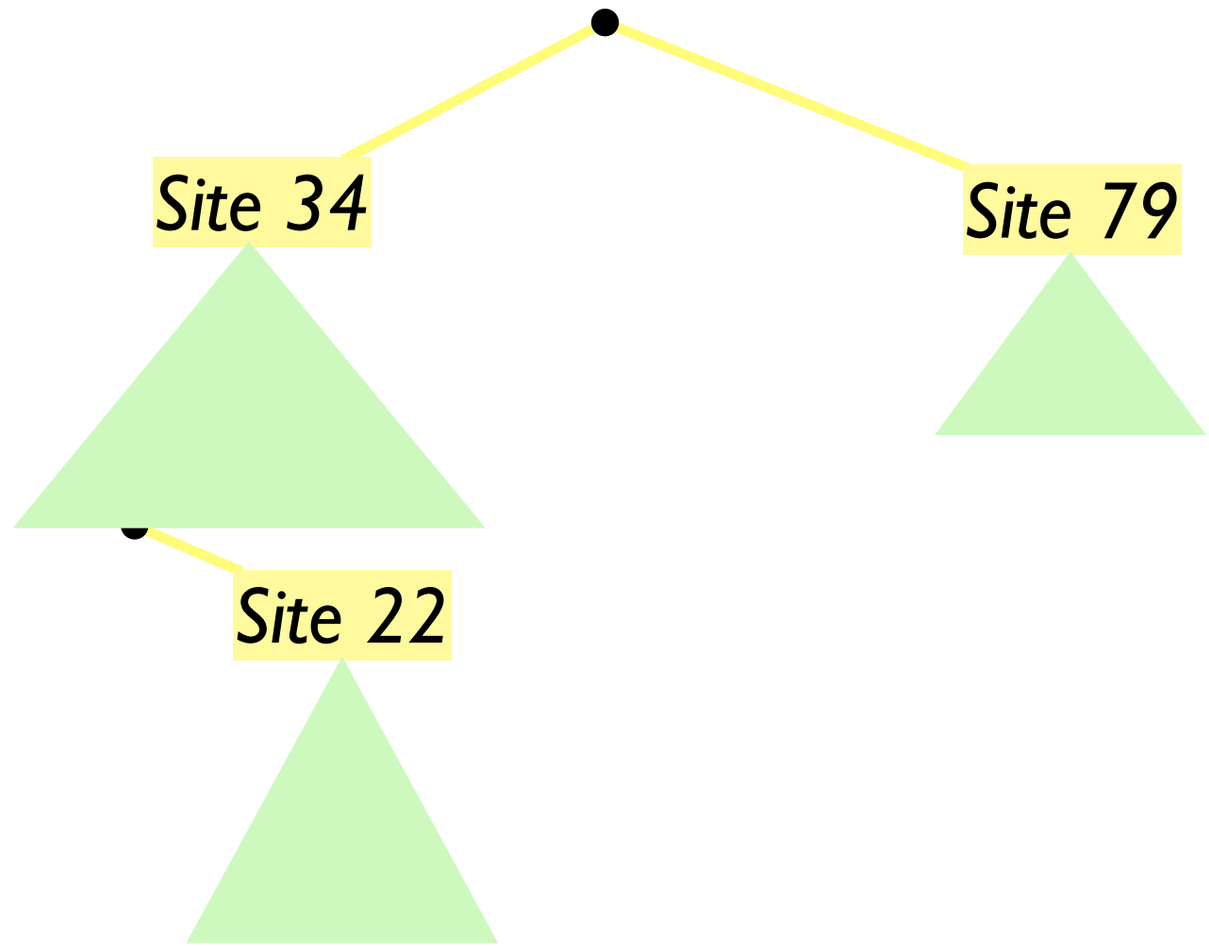
sparse 8^{64}-
-ary tree



binary tree

Layered Treedoc

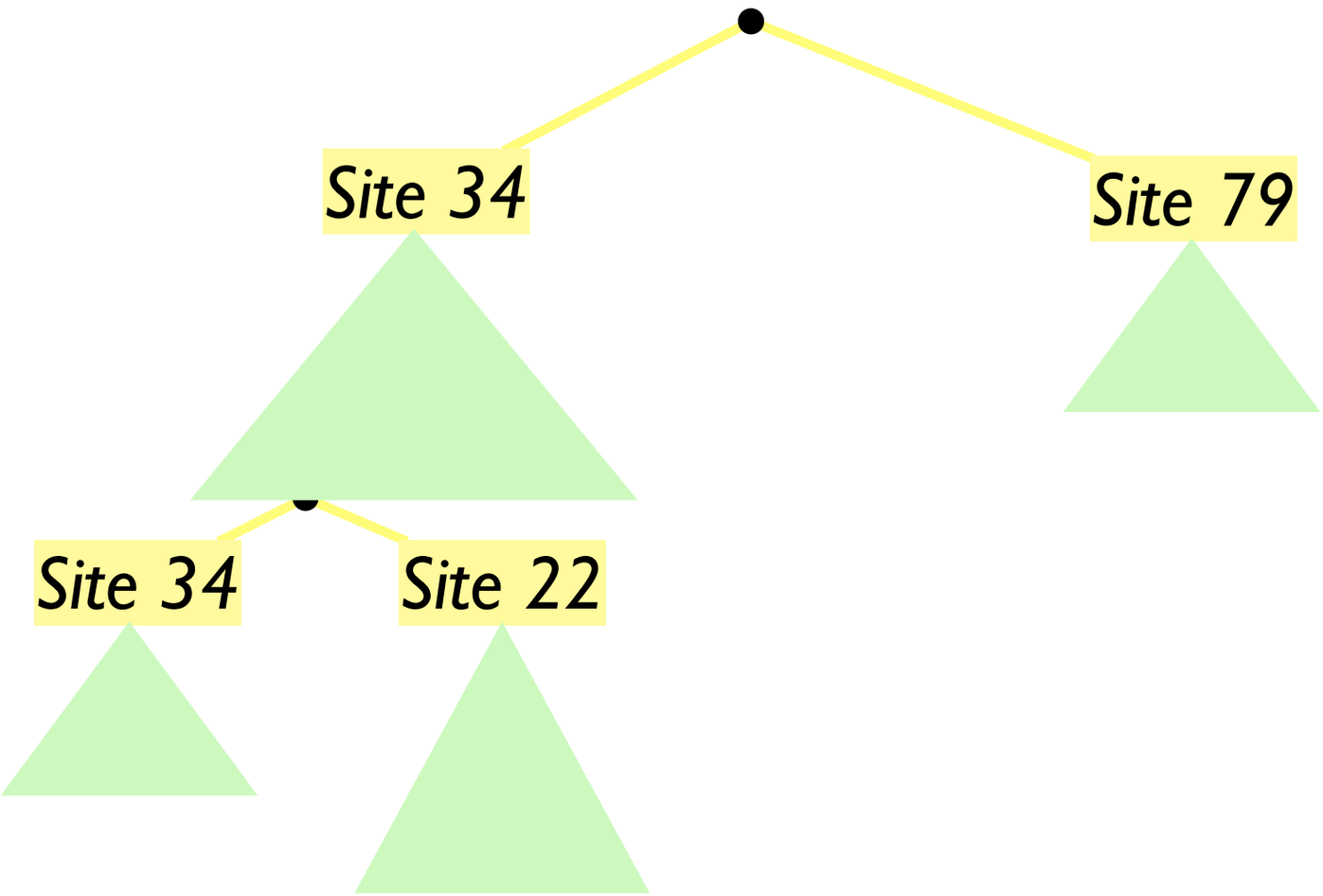
sparse 8^{64}-
-ary tree



binary tree

Layered Treedoc

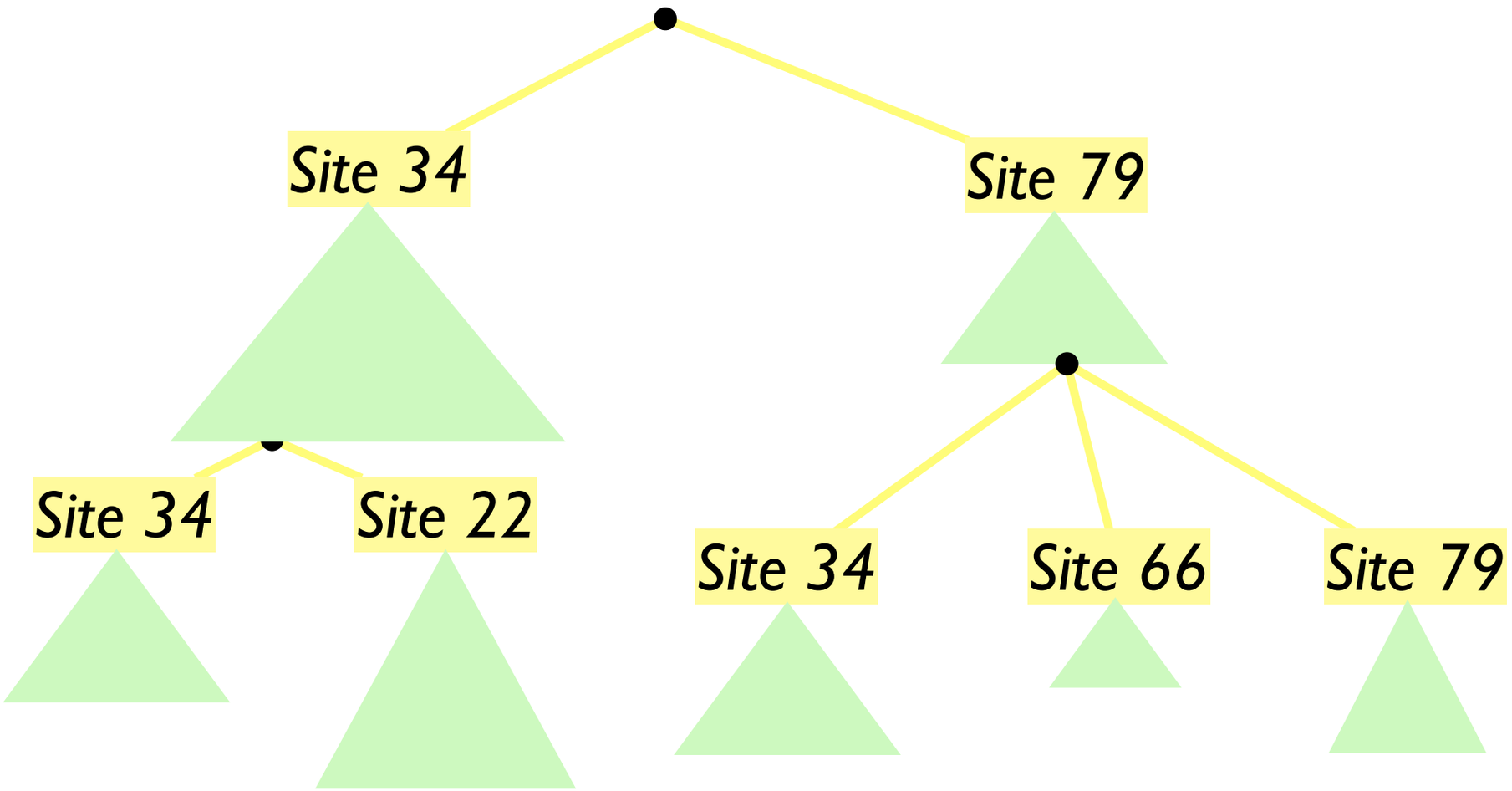
sparse 8^{64}-
-ary tree



binary tree

Layered Treedoc

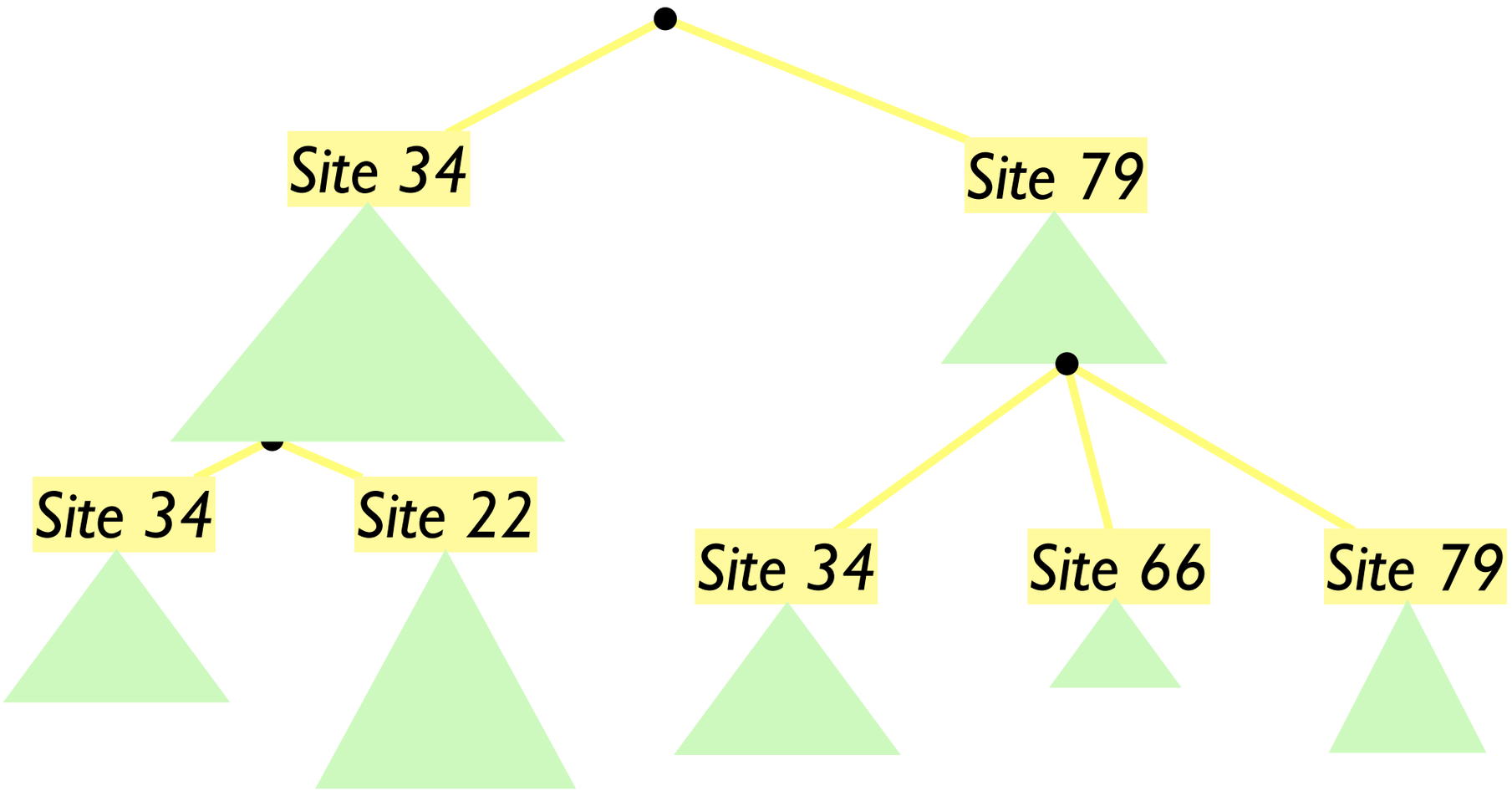
sparse 8^{64}-
-ary tree



binary tree

Layered Treedoc

sparse 8^{64}-
-ary tree



binary tree

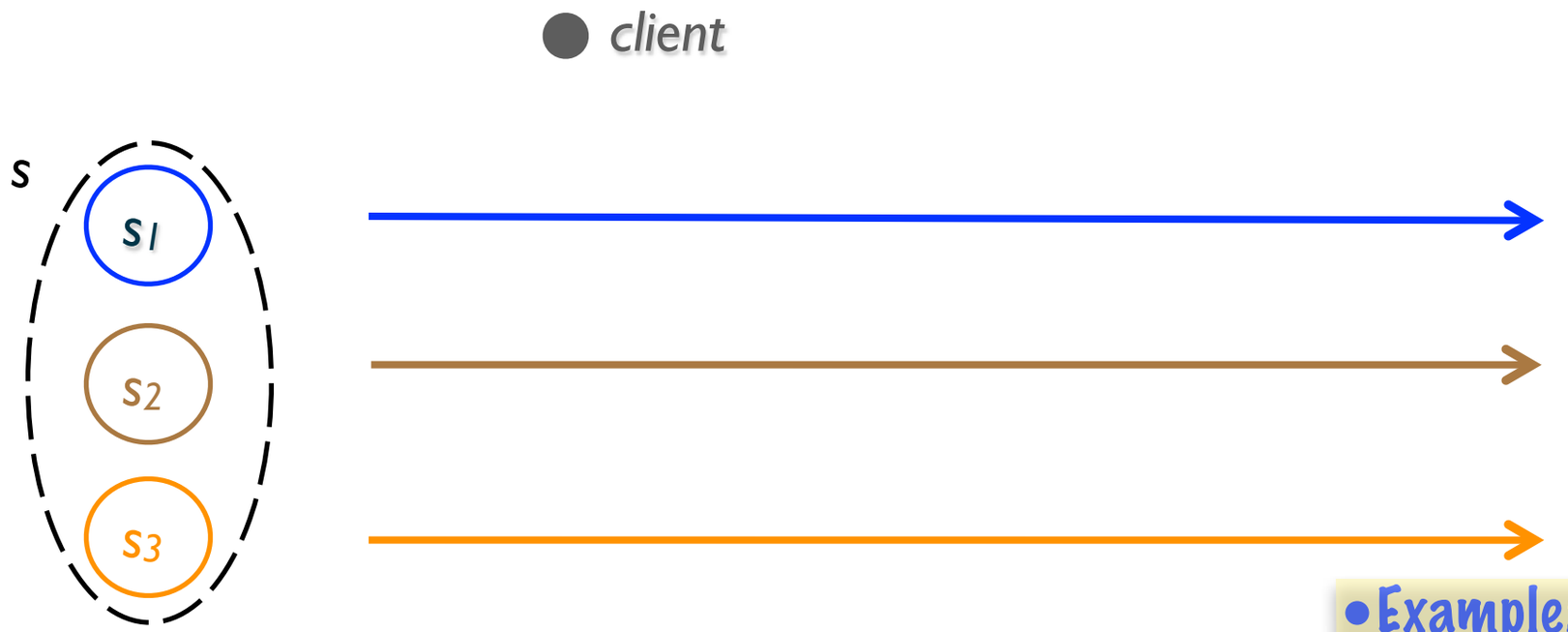
Edit: Binary tree

Concurrency: Sparse tree

The theory

Two simple conditions for correctness without synchronisation

Query

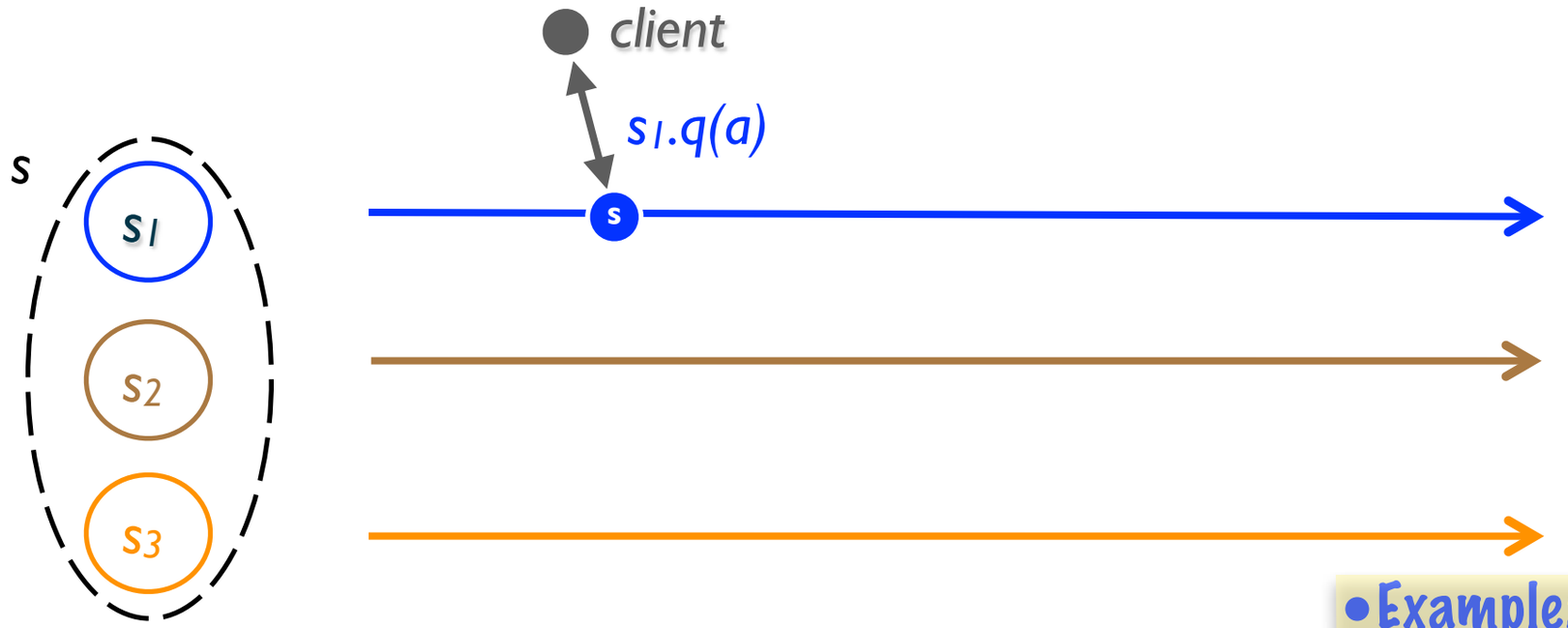


- Example: Amazon shopping cart is replicated
- unspecified client, e.g., Web front-end
- One or more
- load-balancer, failures may direct client to different replicas

Local at *source* replica

- Client's choice

Query

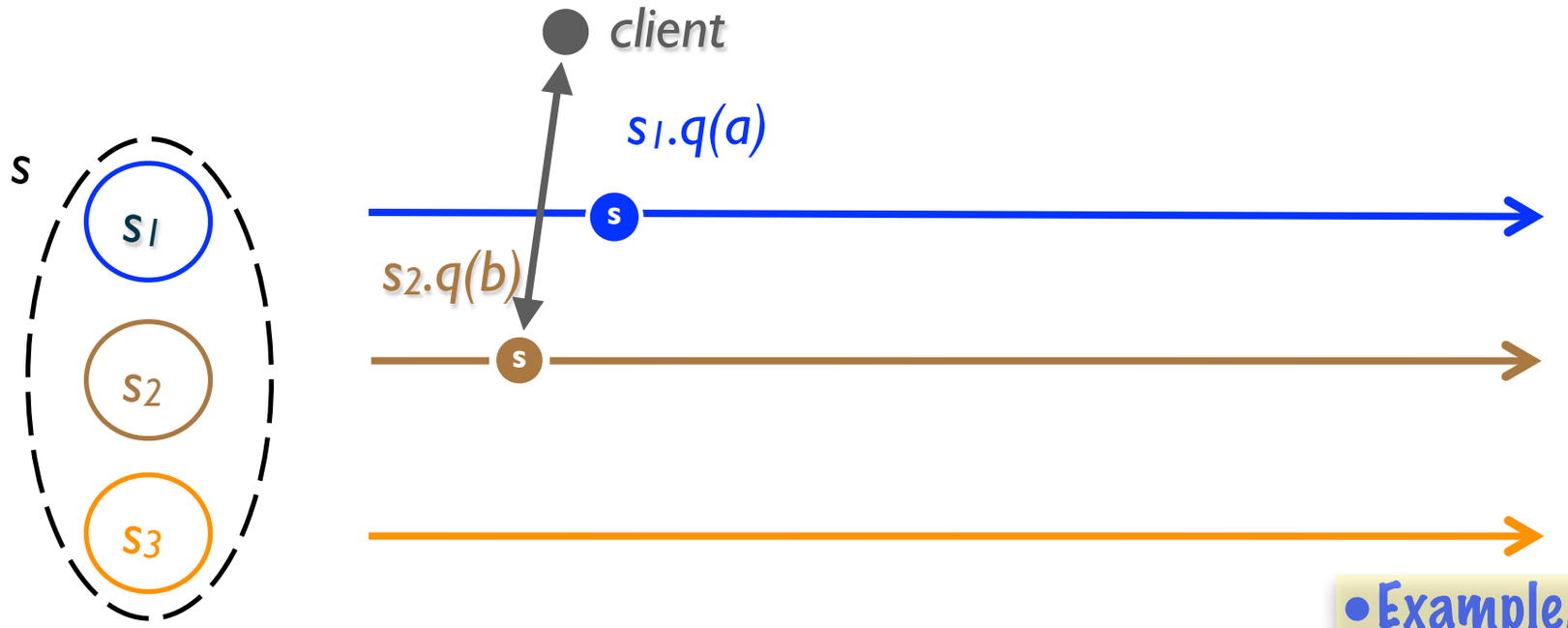


- Example: Amazon shopping cart is replicated
- unspecified client, e.g., Web front-end
- One or more
- load-balancer, failures may direct client to different replicas

Local at *source* replica

- Client's choice

Query

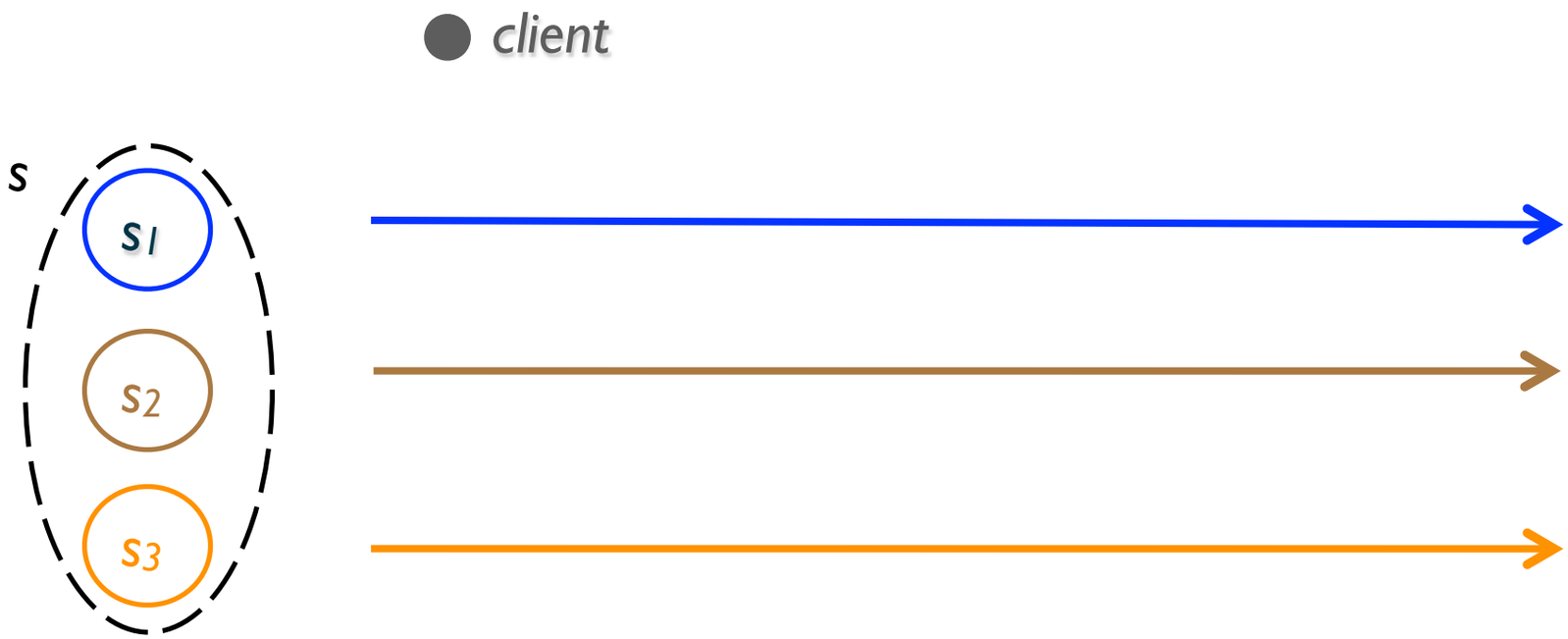


- Example: Amazon shopping cart is replicated
- unspecified client, e.g., Web front-end
- One or more
- load-balancer, failures may direct client to different replicas

Local at *source* replica

- Client's choice

State-based replication



Local at source $s_1.u(a)$, $s_2.u(b)$, …

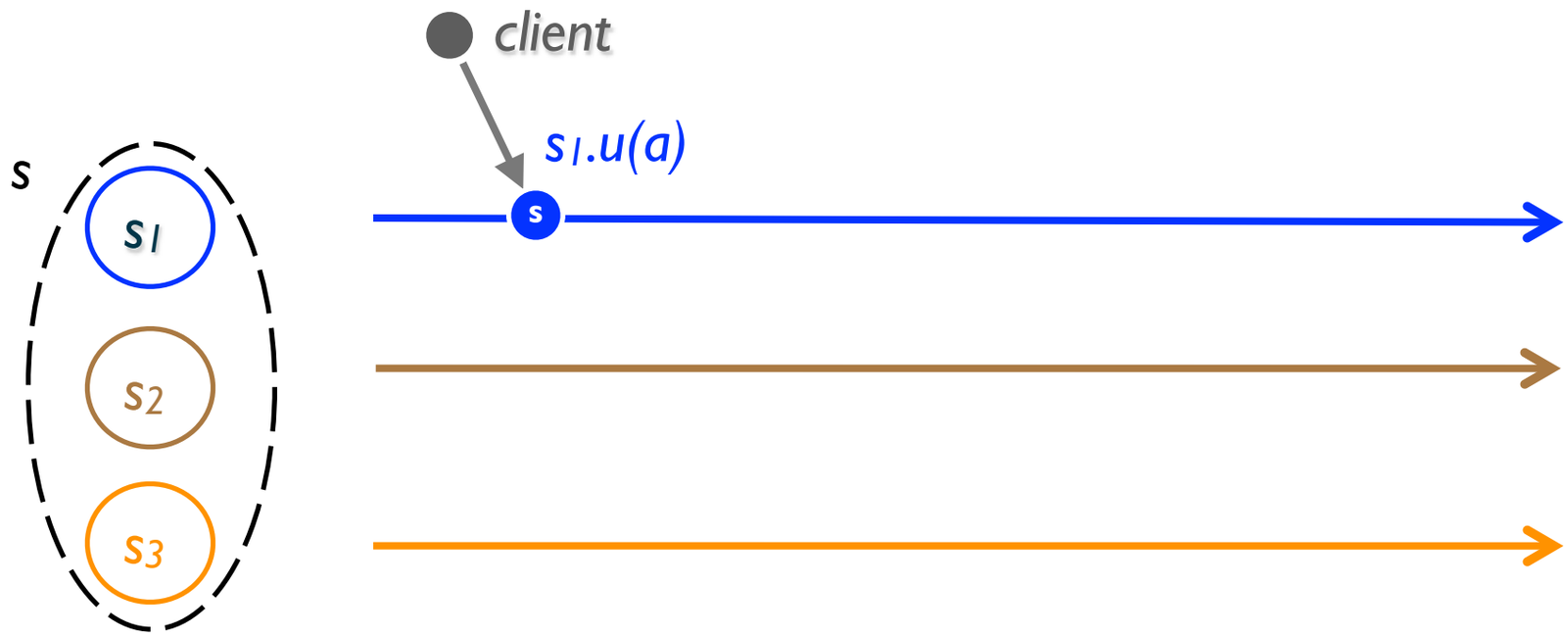
- Compute
- Update local payload

Convergence:

- Episodically: send s_i payload
- On delivery: *merge* payloads *m*

- merge two valid states
- produce valid state
- no historical info available
- Inefficient if payload is large

State-based replication



Local at source $s_1.u(a)$, $s_2.u(b)$, …

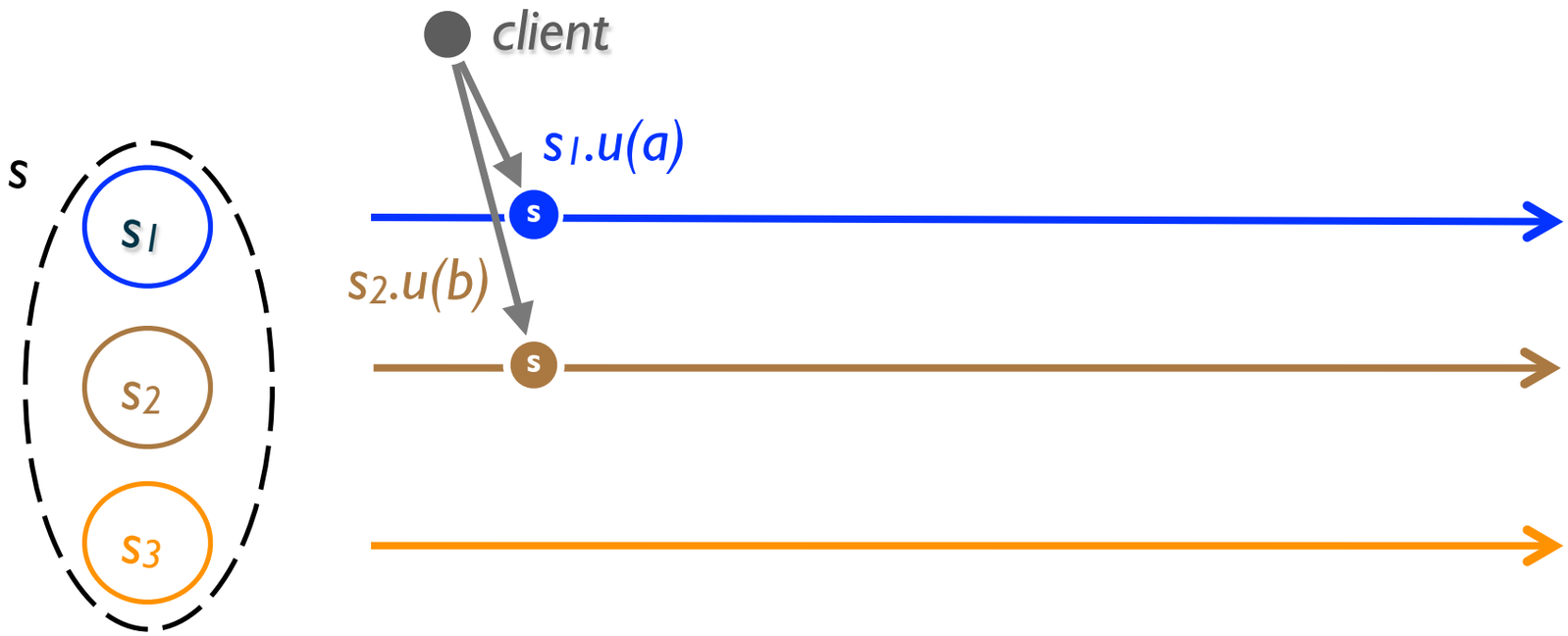
- Compute
- Update local payload

Convergence:

- Episodically: send s_i payload
- On delivery: *merge* payloads *m*

- merge two valid states
- produce valid state
- no historical info available
- Inefficient if payload is large

State-based replication



Local at source $s_1.u(a)$, $s_2.u(b)$, …

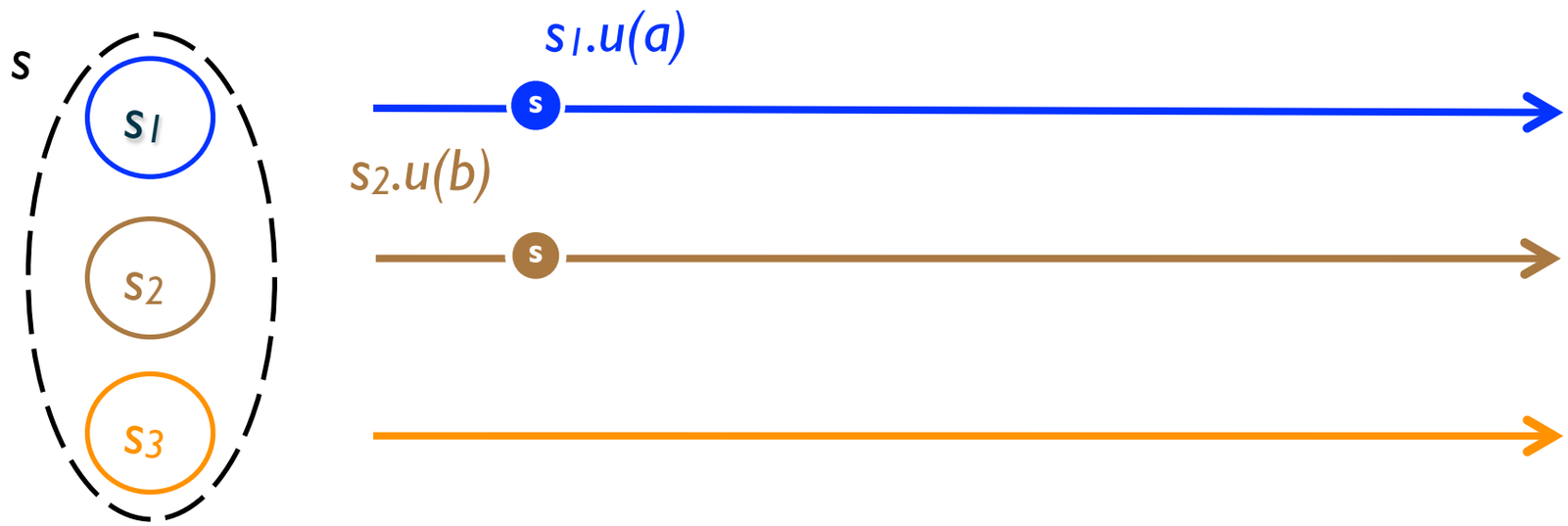
- Compute
- Update local payload

Convergence:

- Episodically: send s_i payload
- On delivery: *merge* payloads *m*

- merge two valid states
- produce valid state
- no historical info available
- Inefficient if payload is large

State-based replication



Local at source $s_1.u(a)$, $s_2.u(b)$, …

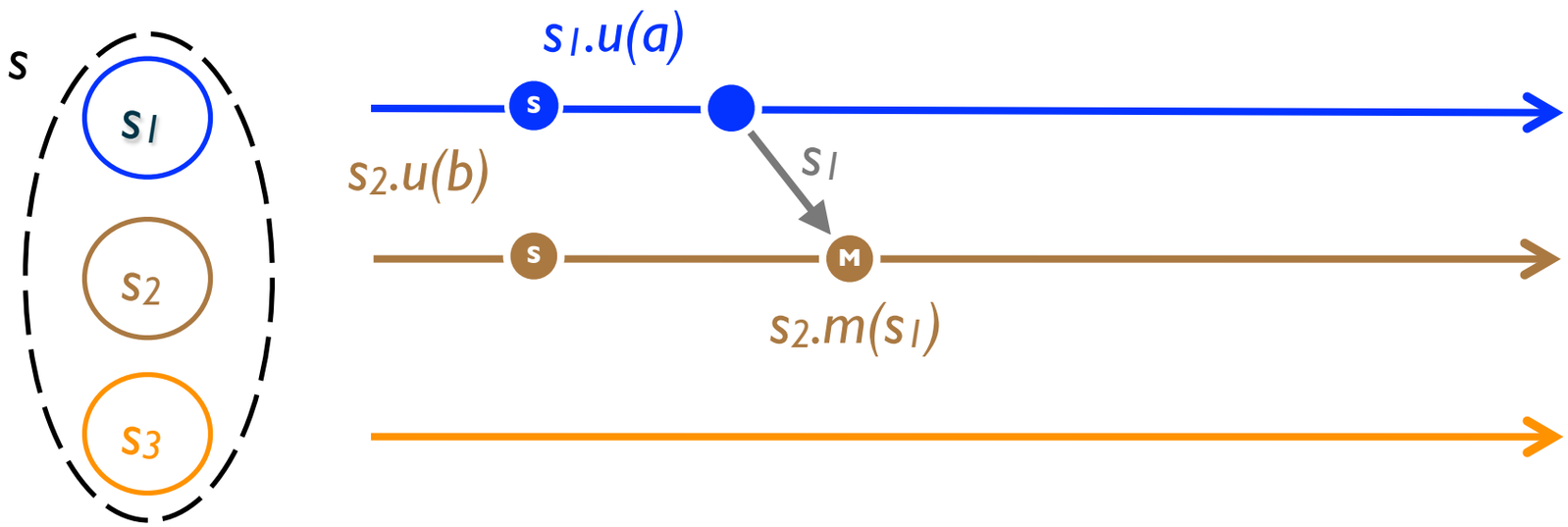
- Compute
- Update local payload

Convergence:

- Episodically: send s_i payload
- On delivery: *merge* payloads *m*

- merge two valid states
- produce valid state
- no historical info available
- Inefficient if payload is large

State-based replication



Local at source $s_1.u(a)$, $s_2.u(b)$, …

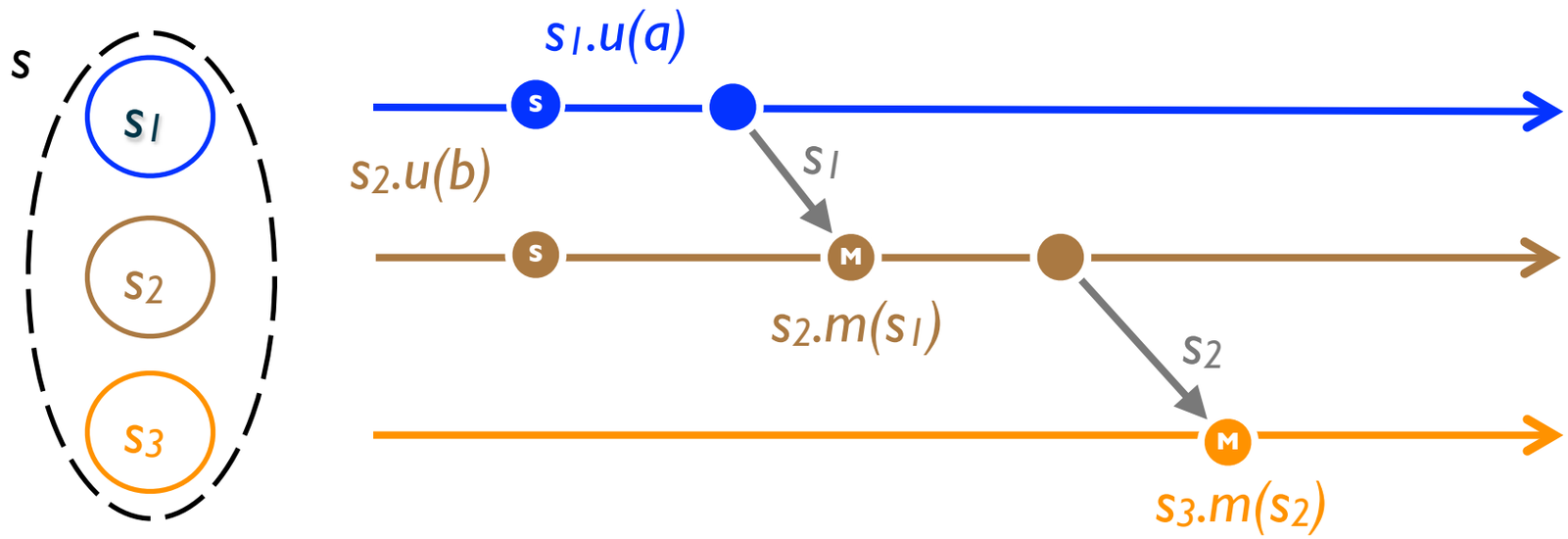
- Compute
- Update local payload

Convergence:

- Episodically: send s_i payload
- On delivery: *merge* payloads *m*

- merge two valid states
- produce valid state
- no historical info available
- Inefficient if payload is large

State-based replication



Local at source $s_1.u(a)$, $s_2.u(b)$, …

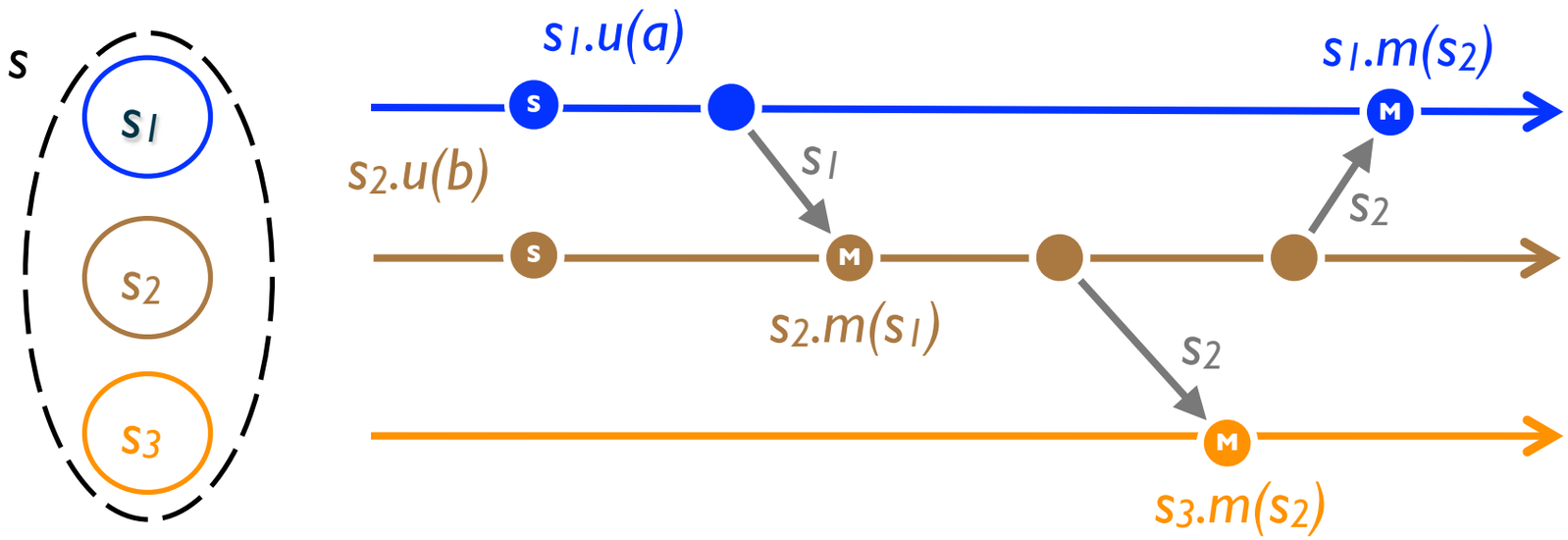
- Compute
- Update local payload

Convergence:

- Episodically: send s_i payload
- On delivery: *merge* payloads *m*

- merge two valid states
- produce valid state
- no historical info available
- Inefficient if payload is large

State-based replication



Local at source $s_1.u(a)$, $s_2.u(b)$, …

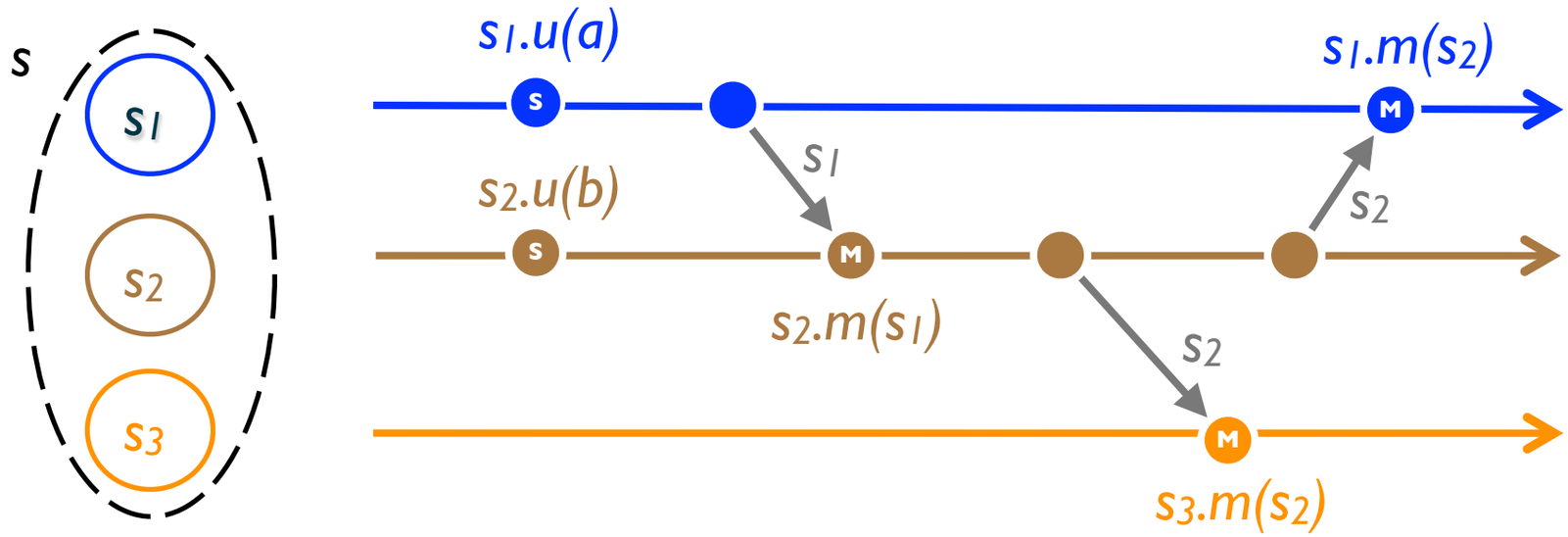
- Compute
- Update local payload

Convergence:

- Episodically: send s_i payload
- On delivery: *merge* payloads *m*

- merge two valid states
- produce valid state
- no historical info available
- Inefficient if payload is large

State-based: monotonic semi-lattice ⇒ CRDT



If

- payload type forms a semi-lattice
- updates are increasing
- *merge* computes Least Upper Bound

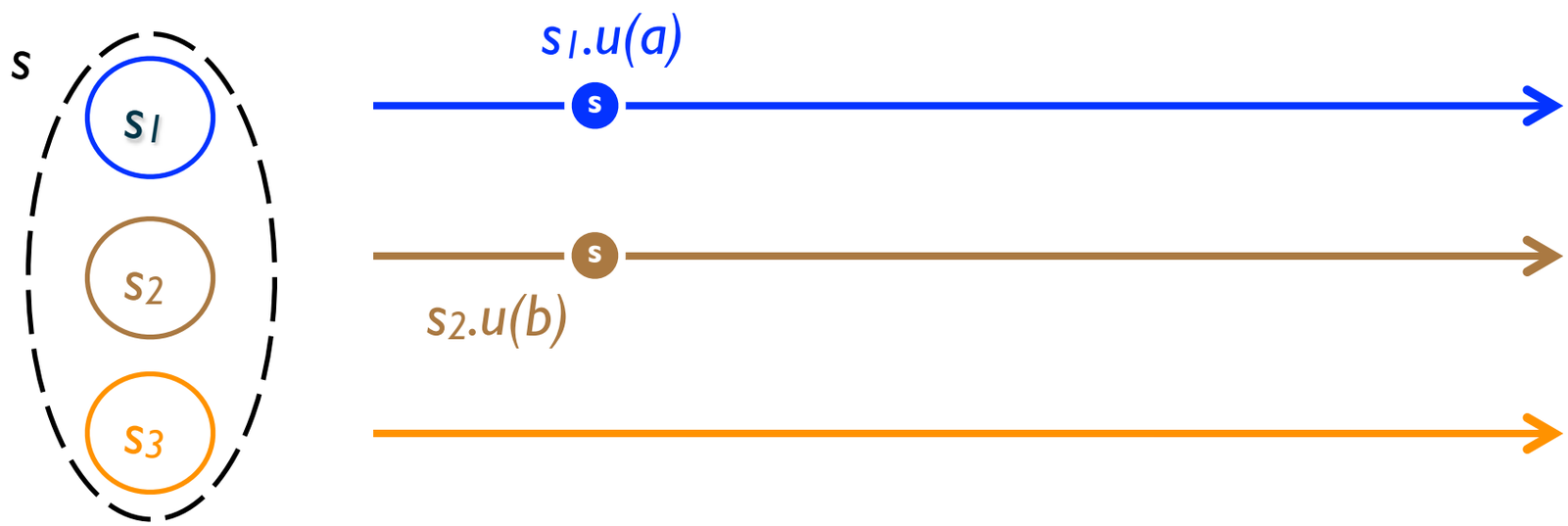
then replicas converge to LUB of last values

Example: Payload = int, *merge* = *max*

- ⊔ = Least Upper Bound
  LUB = merge

- no reference to history

Operation-based replication



- push to all replicas eventually
- push small updates
  - more efficient than state-based

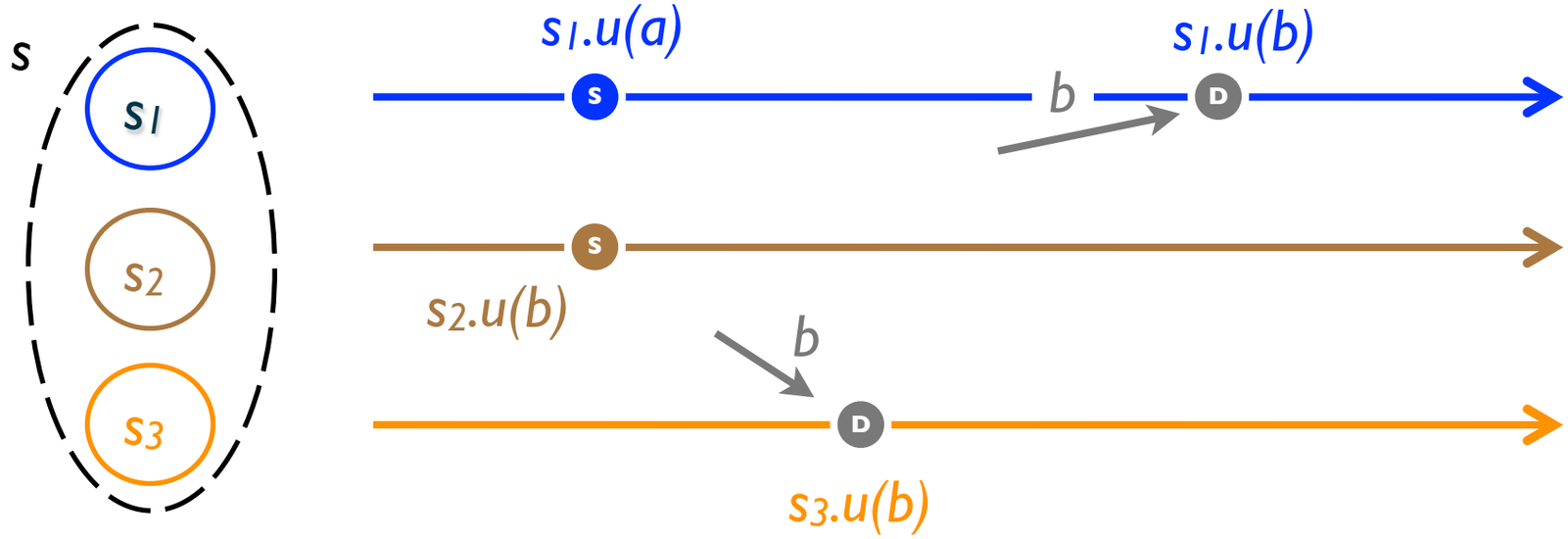
At source:

- prepare
- broadcast to all replicas

Eventually, at all replicas:

- update local replica

Operation-based replication



- push to all replicas eventually
- push small updates
  - more efficient than state-based

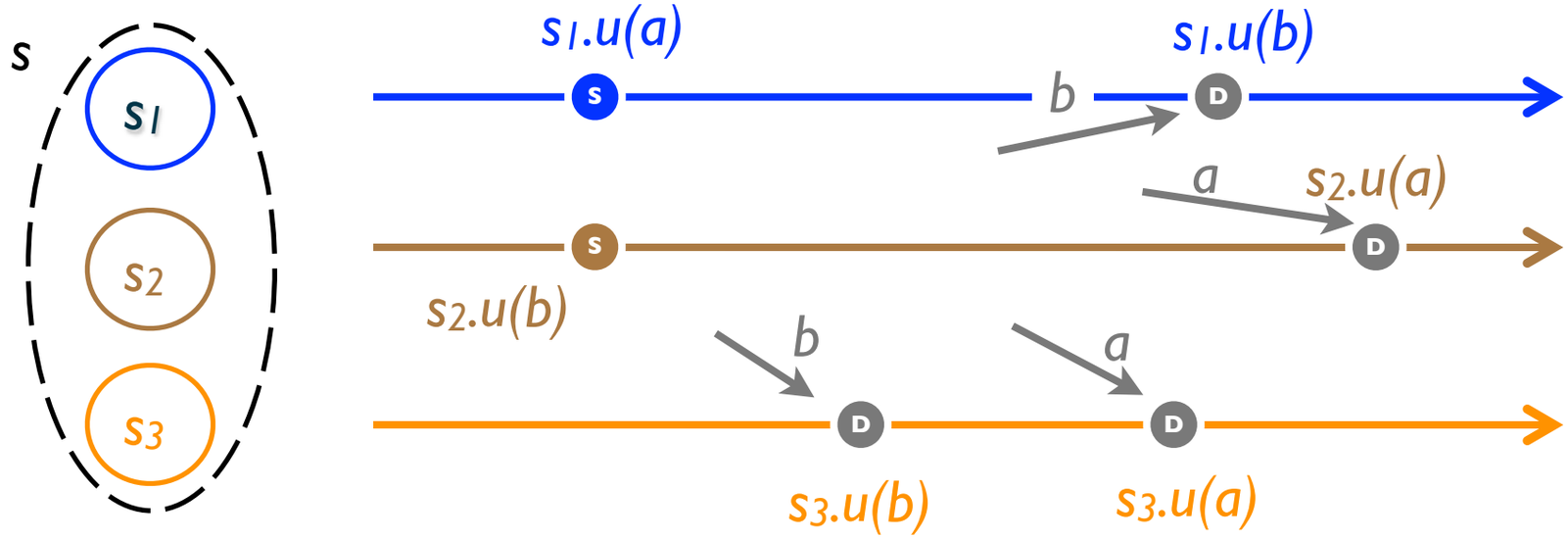
At source:

- prepare
- broadcast to all replicas

Eventually, at all replicas:

- update local replica

Operation-based replication



- push to all replicas eventually
- push small updates
  - more efficient than state-based

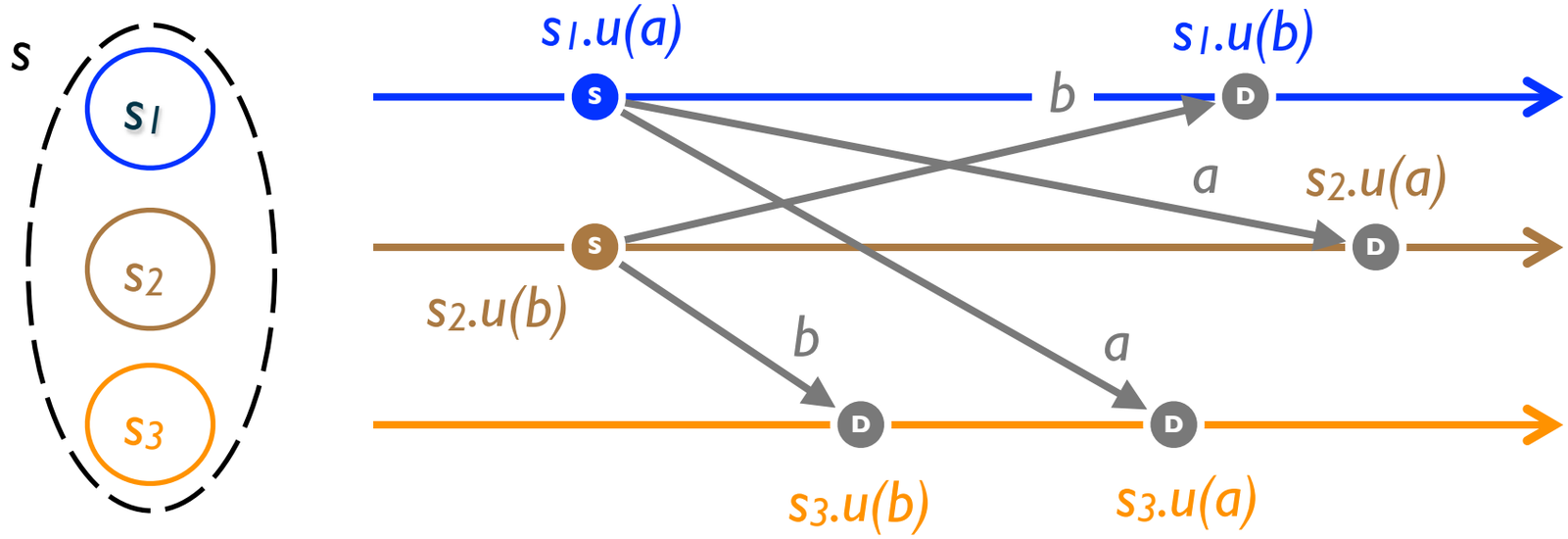
At source:

- prepare
- broadcast to all replicas

Eventually, at all replicas:

- update local replica

Op-based: commute ⇒ CRDT



- Delivery order ≃ ensures downstream precondition
- happened-before or weaker

If: • (*Liveness*) all replicas execute all operations in delivery order
• (*Safety*) concurrent operations all commute
Then: replicas converge

Monotonic semi-lattice ⇔ commutative

- Systematic transformation
- Inefficient
- ⇒ Hand-crafted op-based implementation

1. A state-based object can emulate an operation-based object, and vice-versa
2. State-based emulation of a CvRDT is a CmRDT
3. Operation-based emulation of a CvRDT is a CmRDT

Operation-based OR-Set

Payload: S = { (e,α), (e, β), (e', γ), … }
where $\alpha, \beta, \ldots$ unique

Operations:

- *lookup*$(e) = \exists\ \alpha : (e, \alpha) \in S$

- Set of IDs associated with a in the old state
- As observed by source!

Operation-based OR-Set

Payload: S = { (e,α), (e, β), (e', γ), … }
where $\alpha, \beta, \ldots$ unique

Operations:

- *lookup*$(e) = \exists\ \alpha : (e, \alpha) \in S$
- *add*$(e) = S := S \cup \{(e, \alpha)\}$ where α fresh

- Set of IDs associated with a in the old state
- As observed by source!

Operation-based OR-Set

Payload: S = { (e,α), (e, β), (e', γ), … }
where α, β,… unique

Operations:

- *lookup*(e) = $\exists\ \alpha: (e, \alpha) \in S$
- *add*(e) = $S \coloneqq S \cup \{(e, \alpha)\}$ where α fresh
- *remove* (e) =
  (at source) $R = \{(e, \alpha) \in S\}$
  (downstream) $S \coloneqq S \setminus R$

- Set of IDs associated with a in the old state
- As observed by source!

Operation-based OR-Set

Payload: S = { (e,α), (e, β), (e', γ), … }
where α, β,… unique

Operations:

- *lookup*(e) = $\exists\ \alpha: (e, \alpha) \in S$
- *add*(e) = $S \coloneqq S \cup \{(e, \alpha)\}$ where α fresh
- *remove* (e) =
  (at source) $R = \{(e, \alpha) \in S\}$
  (downstream) $S \coloneqq S \setminus R$

No tombstones

- Set of IDs associated with a in the old state
- As observed by source!

Operation-based OR-Set

Payload: S = { (e,α), (e, β), (e', γ), … }
where $\alpha, \beta, \ldots$ unique

Operations:

- *lookup*(e) = $\exists\ \alpha: (e, \alpha) \in S$
- *add*(e) = $S \coloneqq S \cup \{(e, \alpha)\}$ where α fresh
- *remove* (e) =
  (at source) $R = \{(e, \alpha) \in S\}$
  (downstream) $S \coloneqq S \setminus R$

No tombstones

$\{true\}$ *add*(e) || *remove*(e) $\{e \in S\}$

- Set of IDs associated with a in the old state
- As observed by source!

Ongoing work

CRDTs for P2P & Cloud Computing

ConcoRDanT: ANR 2010–2013

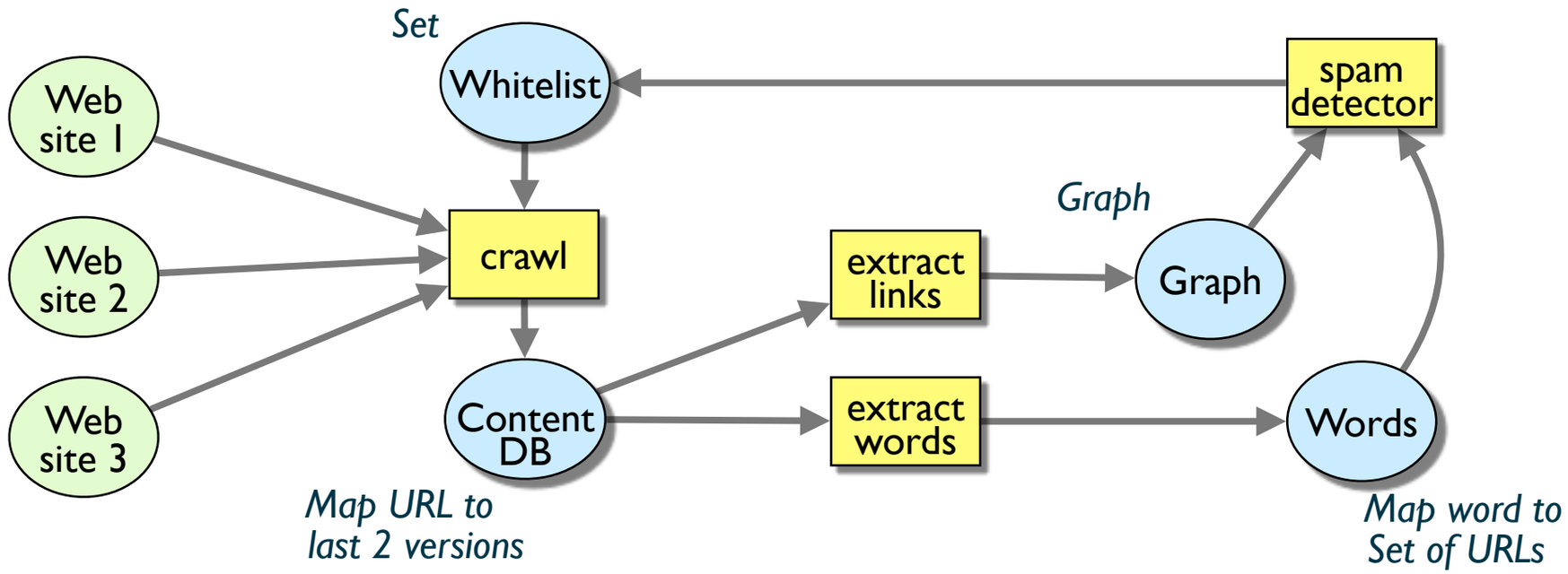
Systematic study of conflict-free design space

- Theory and practice
- Characterise invariants
- Library of data types

Not universal

- Conflict-free vs. conflict semantics
- Move consensus off critical path, non-critical ops

CRDT + dataflow



Incremental, asynchronous processing

- Replicate, shard CRDTs near the edge
- Propagate updates ≈ dataflow
- Throttle according to QoS metrics (freshness, availability, cost, etc.)

Scale: sharded

Synchronous processing: snapshot, at centre

OR-Set + Snapshot

Read consistent snapshot

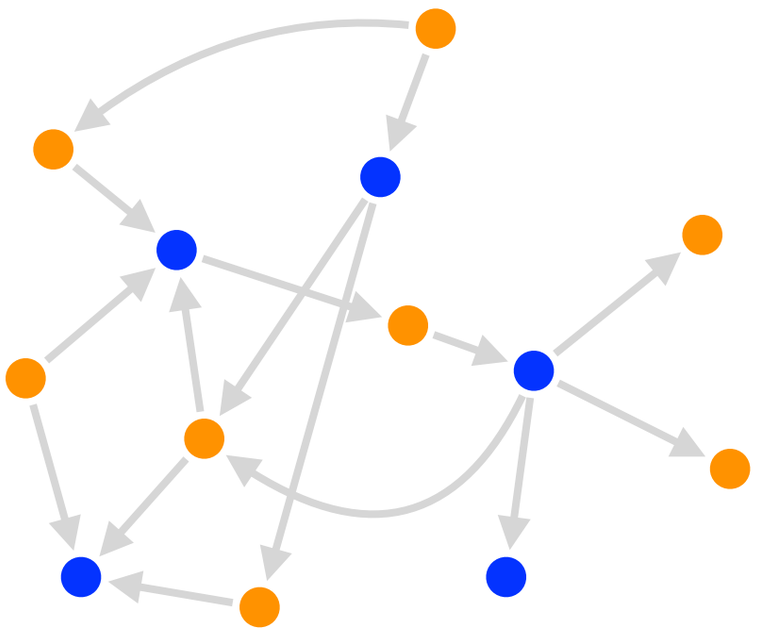
- Despite concurrent, incremental updates

Unique token = time (vector clock)

- α = Lamport *(process i, counter t)*
- UIDs identify snapshot version
- Snapshot: vector clock value
- Retain tombstones until not needed

$lookup(e, t) = \exists\ (e, i, t') \in A : t' > t \wedge \nexists\ (e, i, t') \in R : t' > t$

Sharded OR-Set



Very large objects

- Independent *shards*
- Static: hash

(Dynamic: requires consensus to rebalance)

Statically-Sharded CRDT

- Each shard is a CRDT
- Update: single shard
- No cross-object invariants
- The combination remains a CRDT

Statically Sharded OR-Set

- Combination of smaller OR-Sets
- Snapshot: clock across shards

Take aways

Principled approach

- Strong Eventual Consistency

Two sufficient conditions:

- State: monotonic semi-lattice
- Operation: commutativity

Useful CRDTs

- Register, Counter, Set, Map (KVS), Graph, Monotonic DAG, Sequence

Future work

- Snapshot, sharding, dataflow
- A wee bit of synchronisation

Portfolio of CRDTs

Register
- Last-Writer Wins
- Multi-Value

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

Map
- Set of Registers

Counter
- Unlimited
- Non-negative

Graphs
- Directed
- Monotonic DAG
- Edit graph

Sequence
- Edit sequence