Conflict-Free Replicated Sets

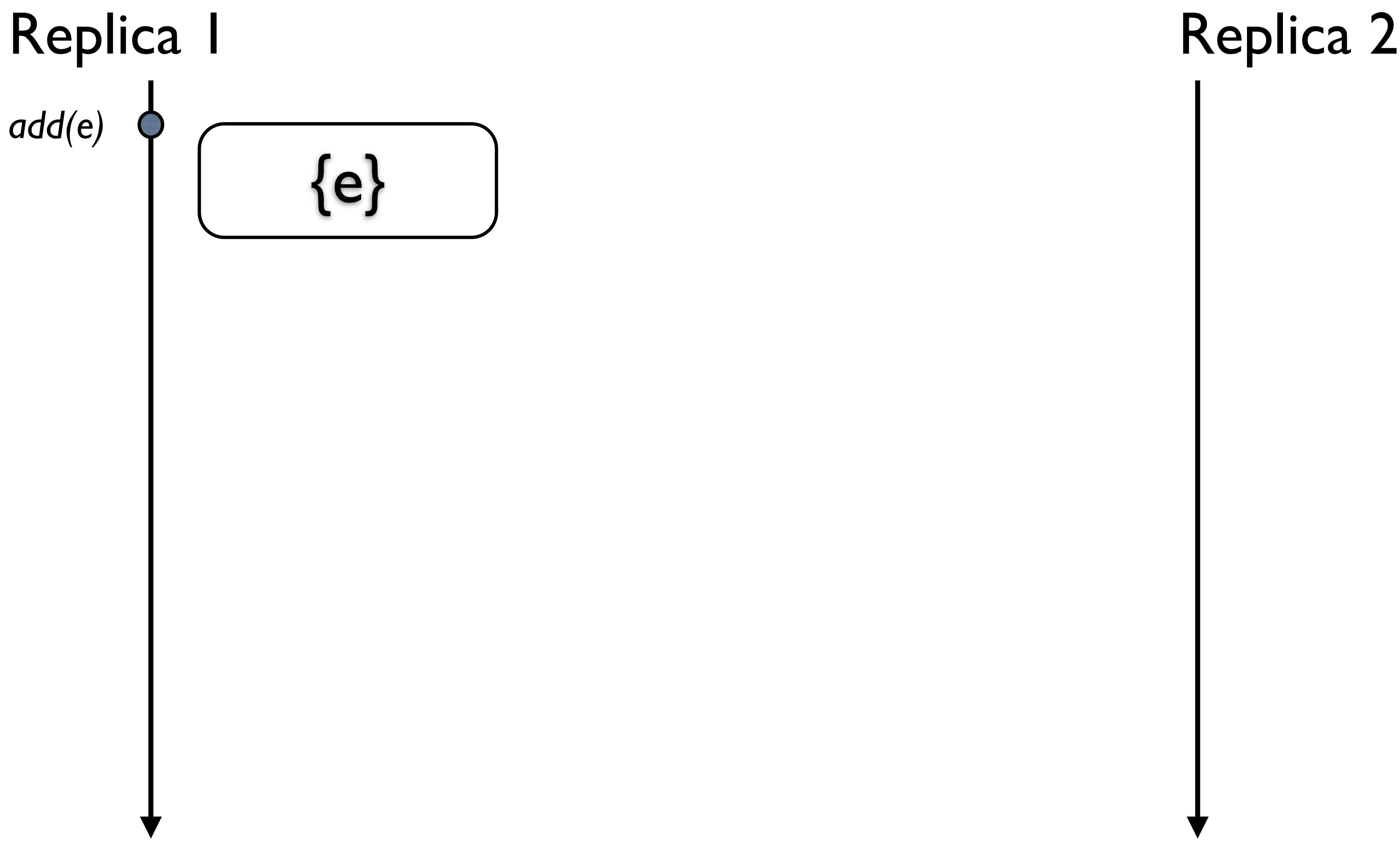
Annette Bieniusa
together with Nuno Preguiça, Marc Shapiro, Marek Zawirski,
Carlos Baquero, Valter Balegas

Replicated Sets

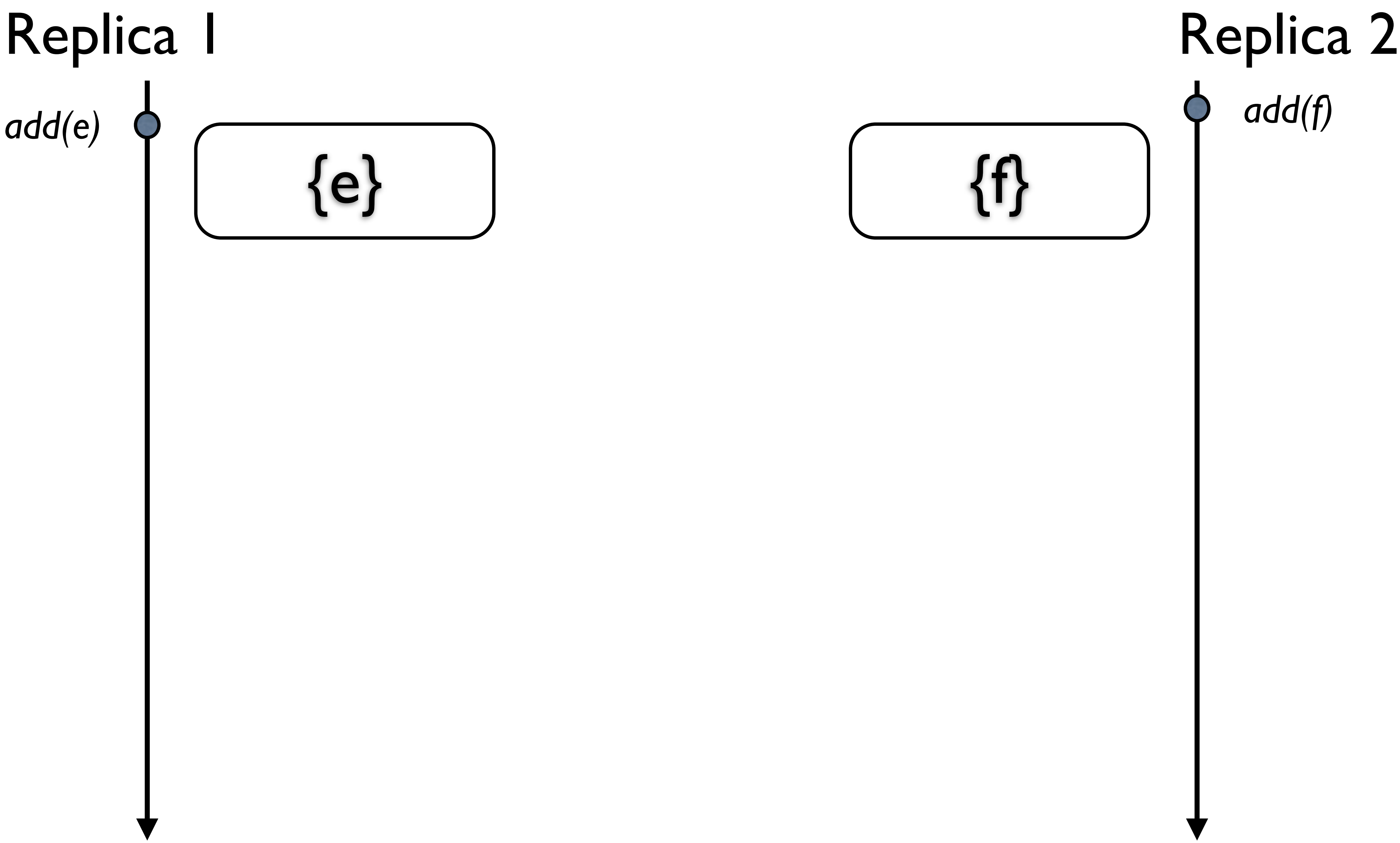
- Sets are standard container data type
- Building block for many other data types such as maps or graphs
- Previous replicated set designs seem flawed...

Amazon Dynamo Cart

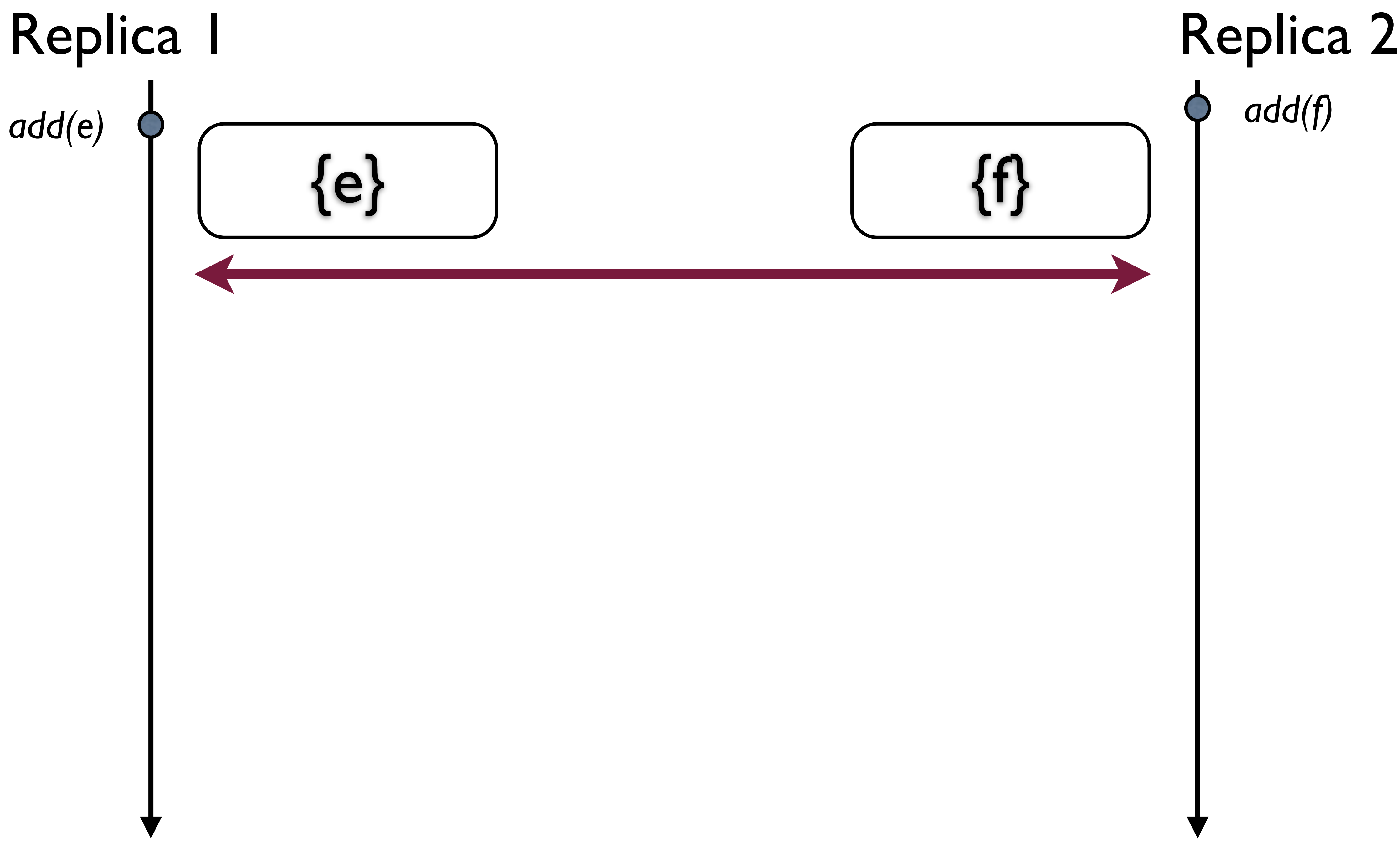
Amazon Dynamo Cart



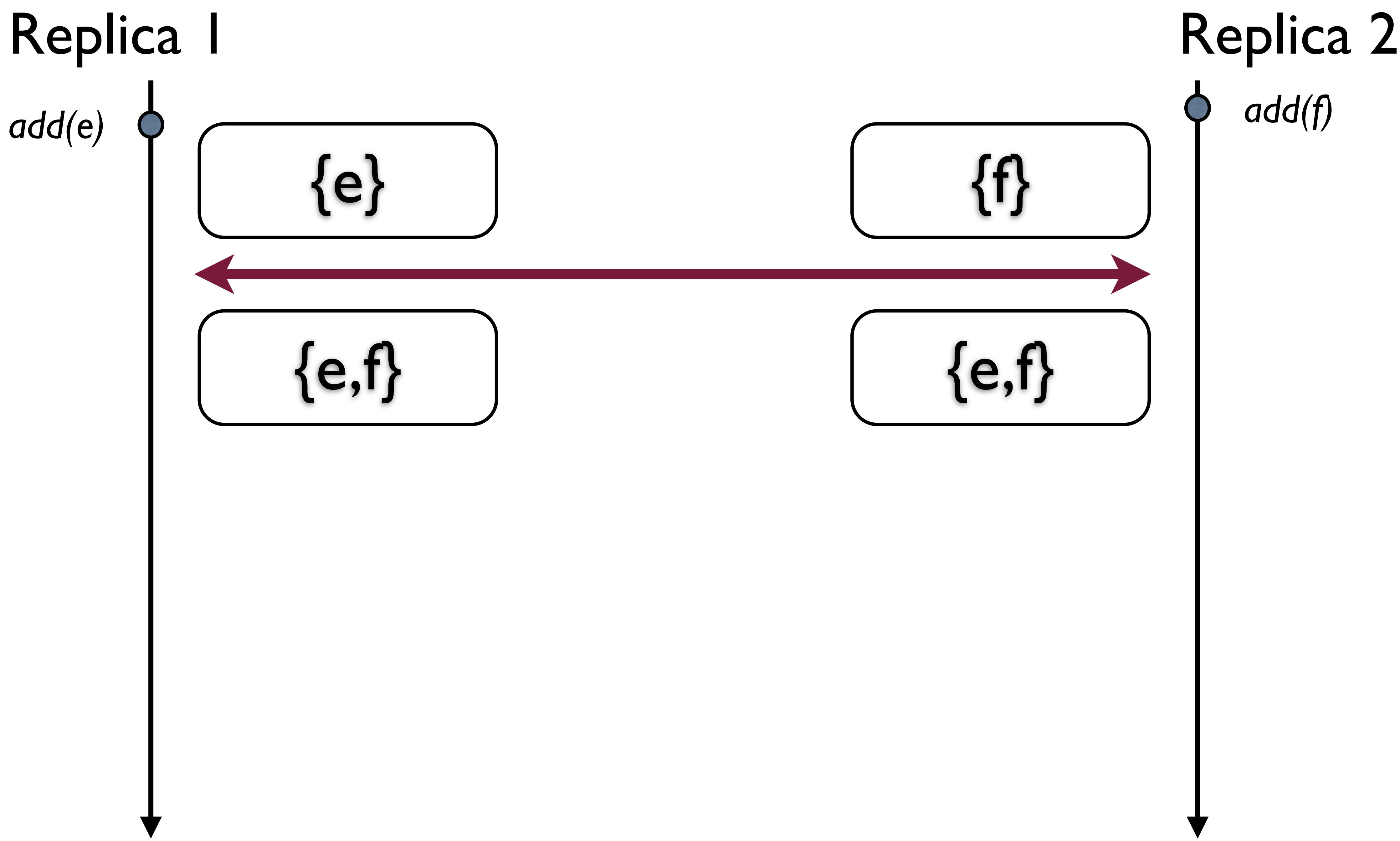
Amazon Dynamo Cart



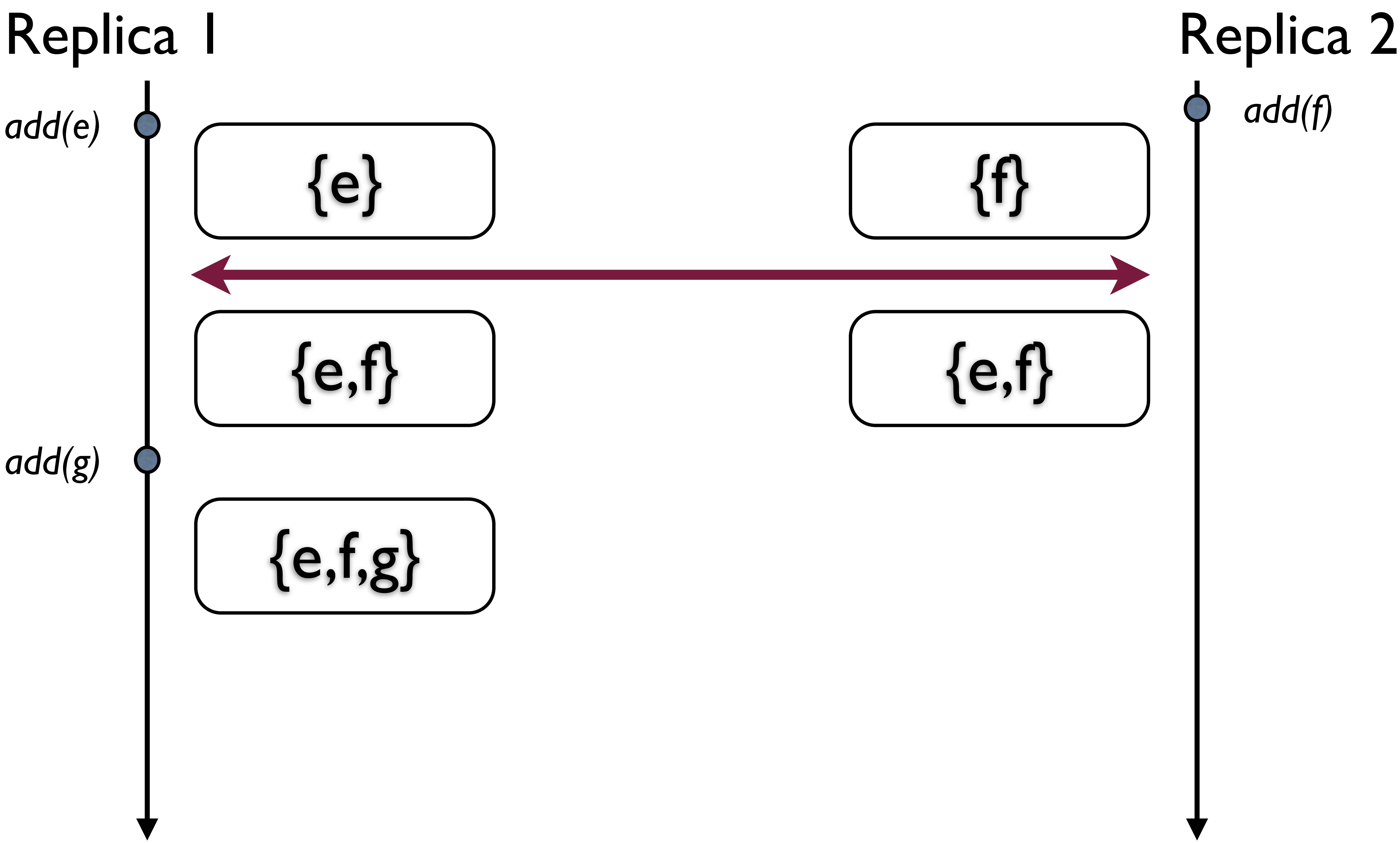
Amazon Dynamo Cart



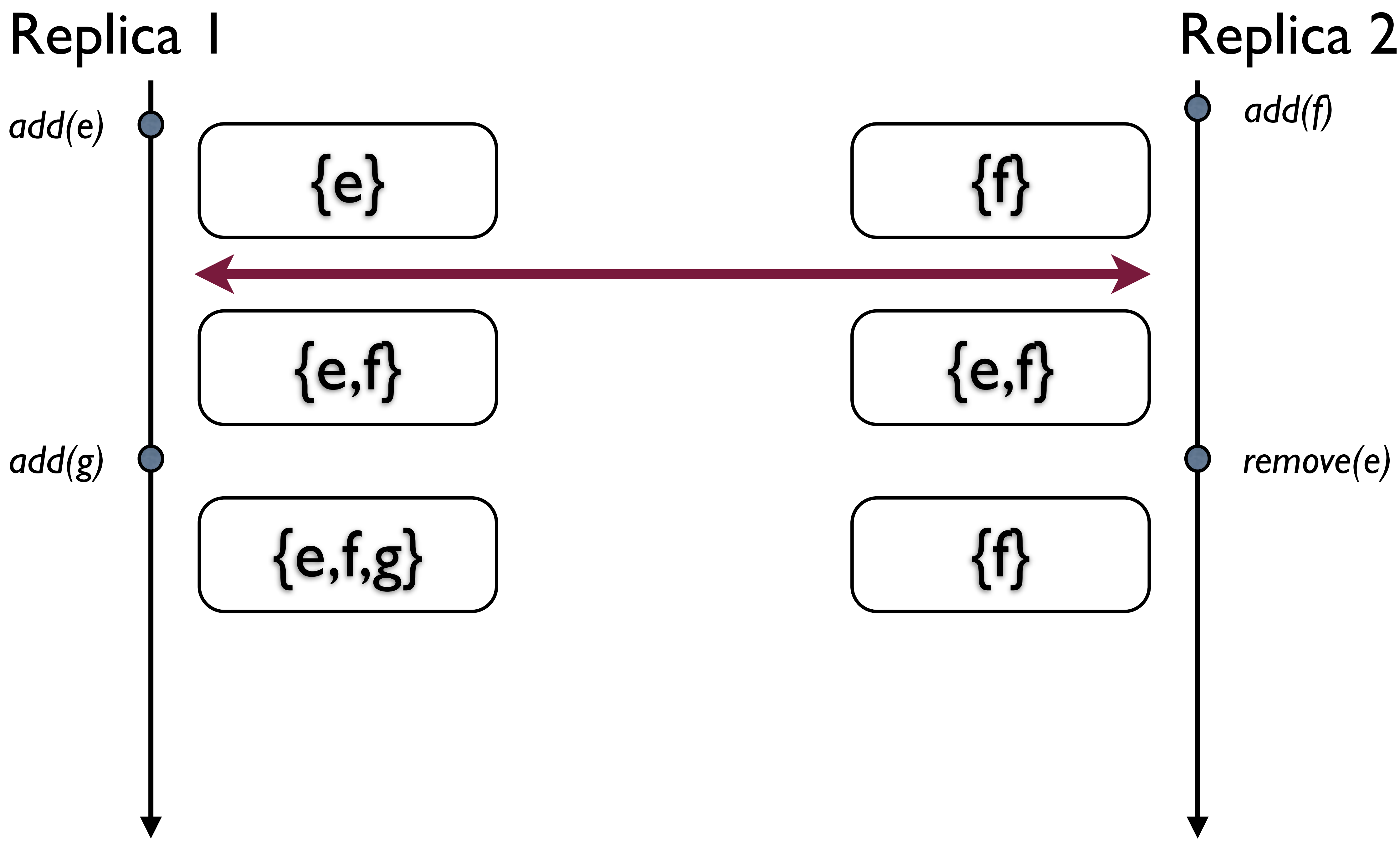
Amazon Dynamo Cart



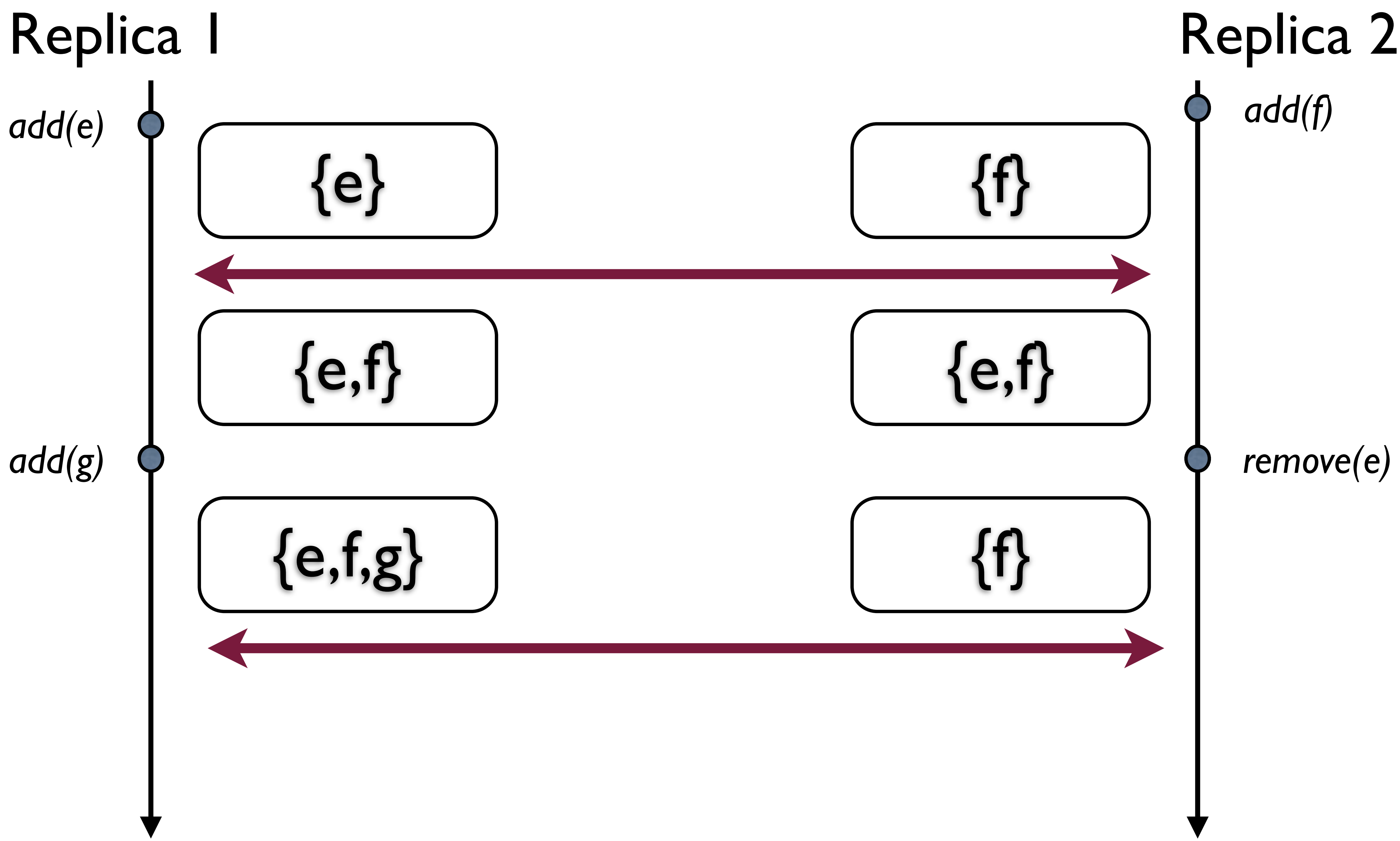
Amazon Dynamo Cart



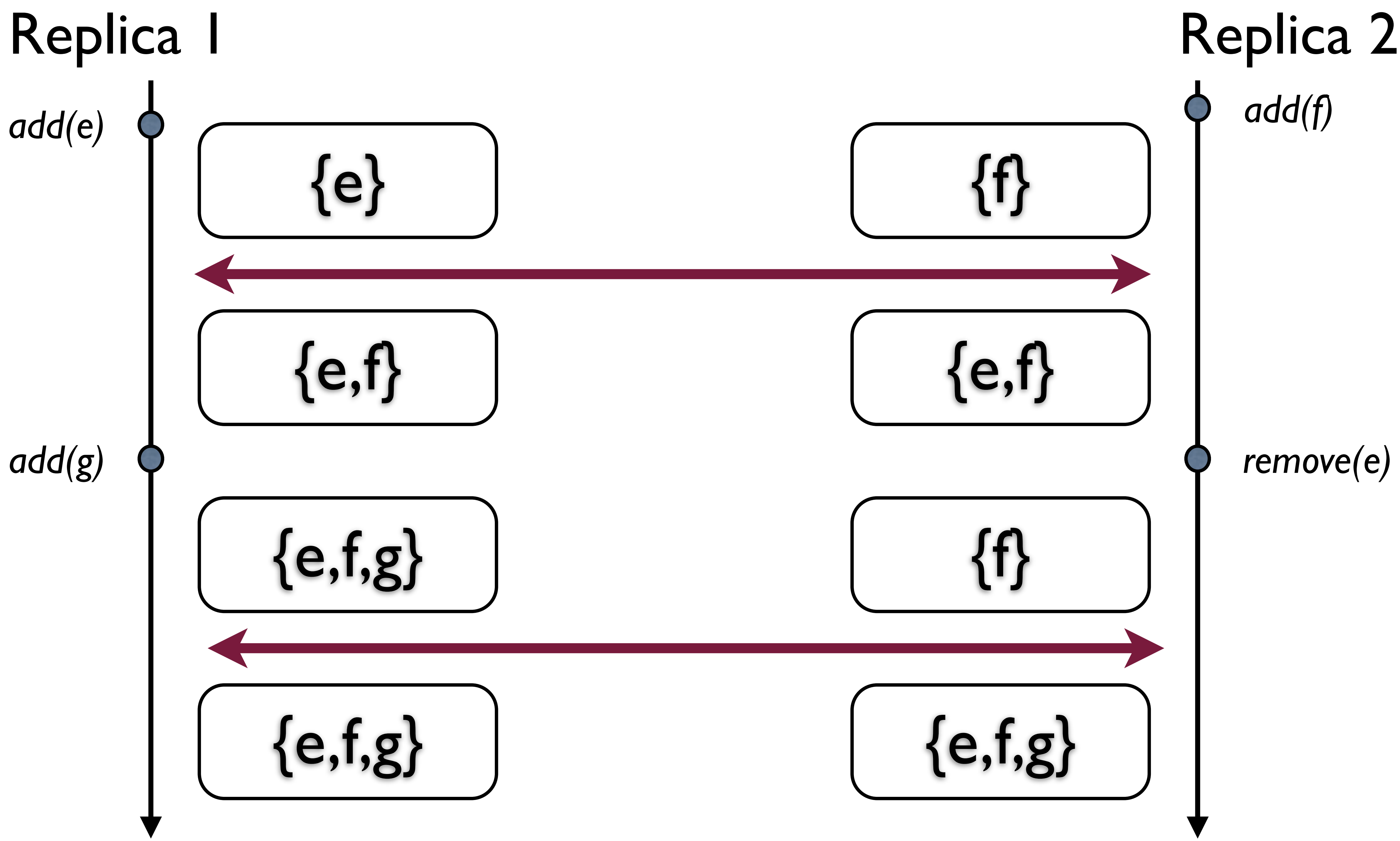
Amazon Dynamo Cart



Amazon Dynamo Cart

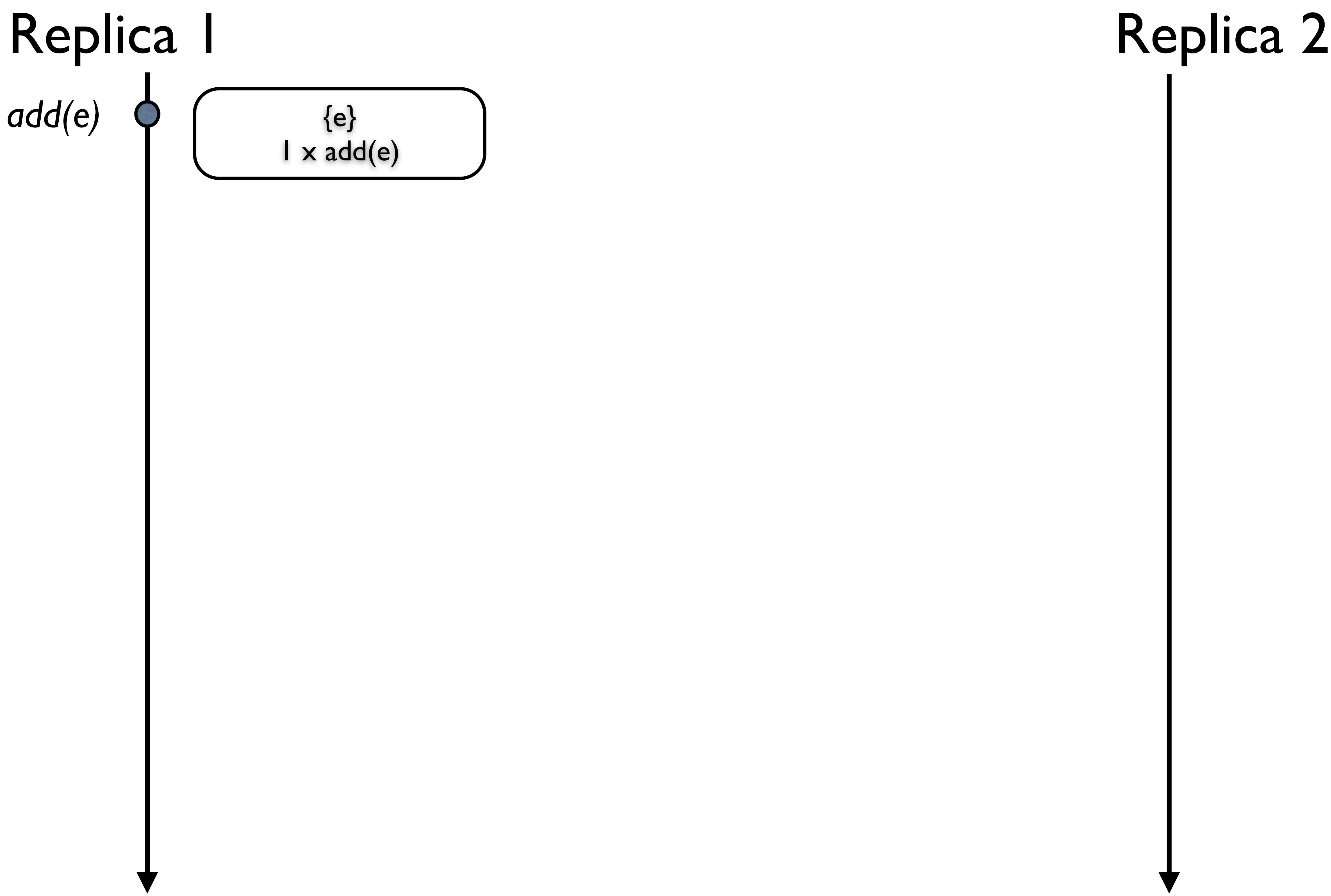


Amazon Dynamo Cart

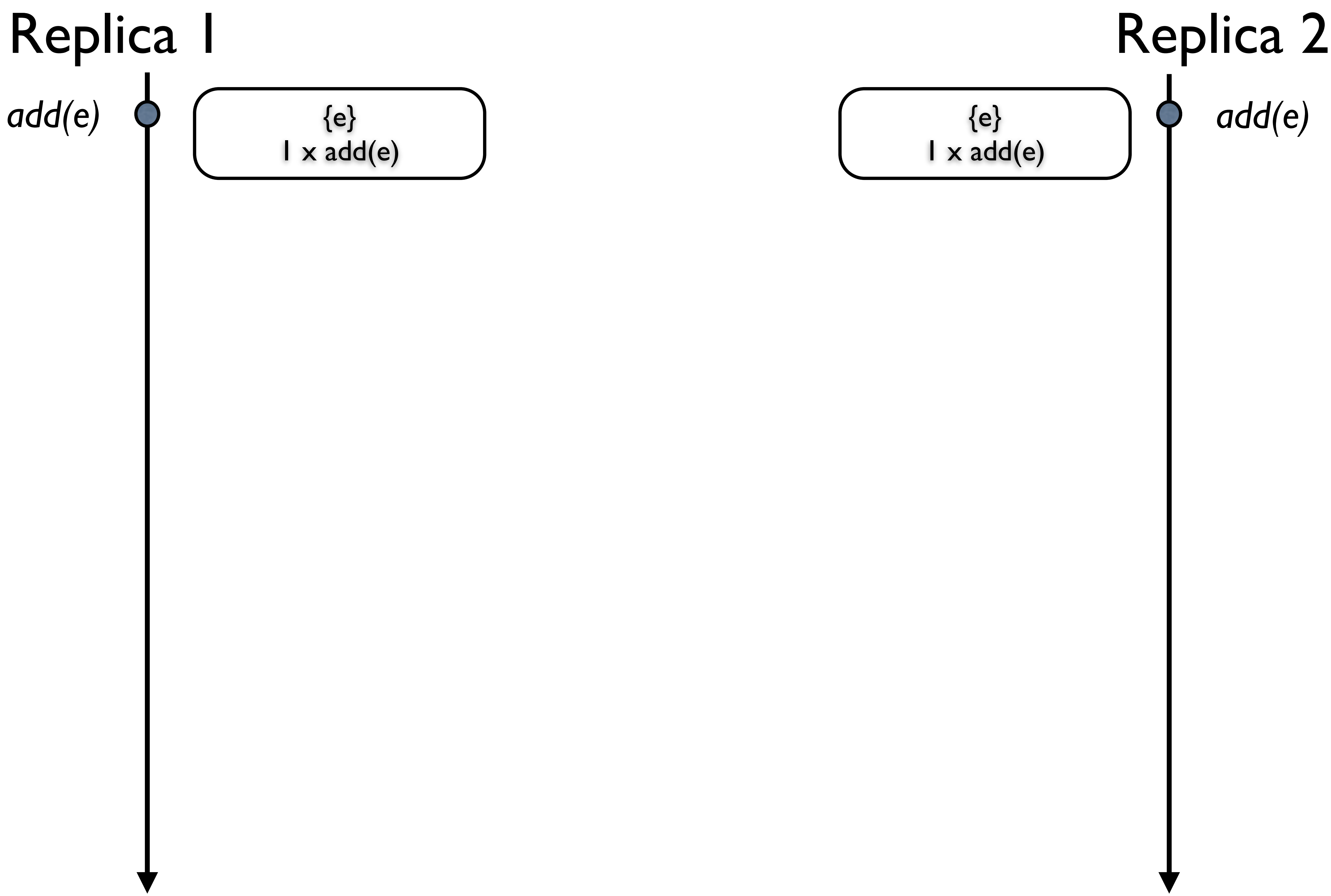


C-Set

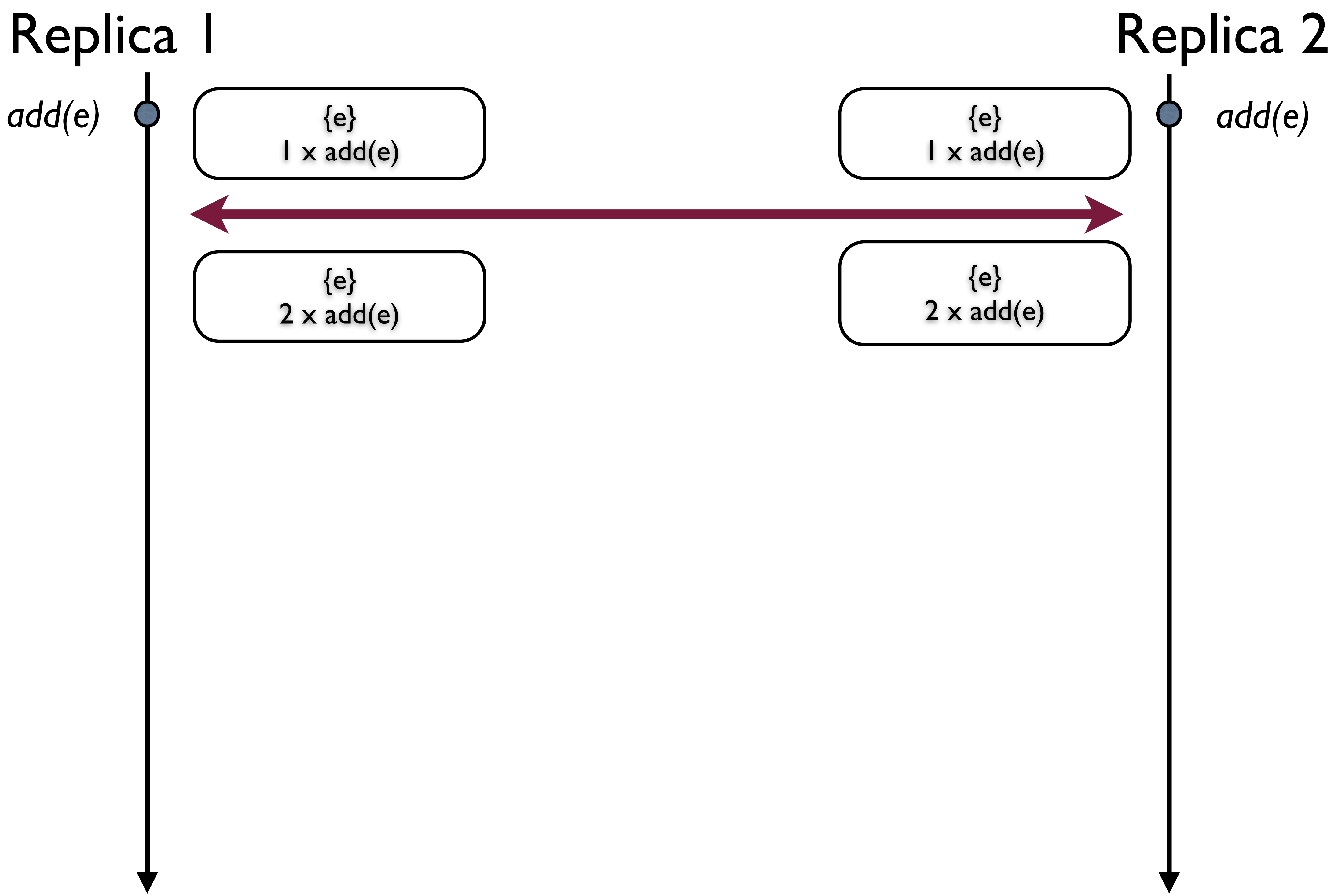
C-Set



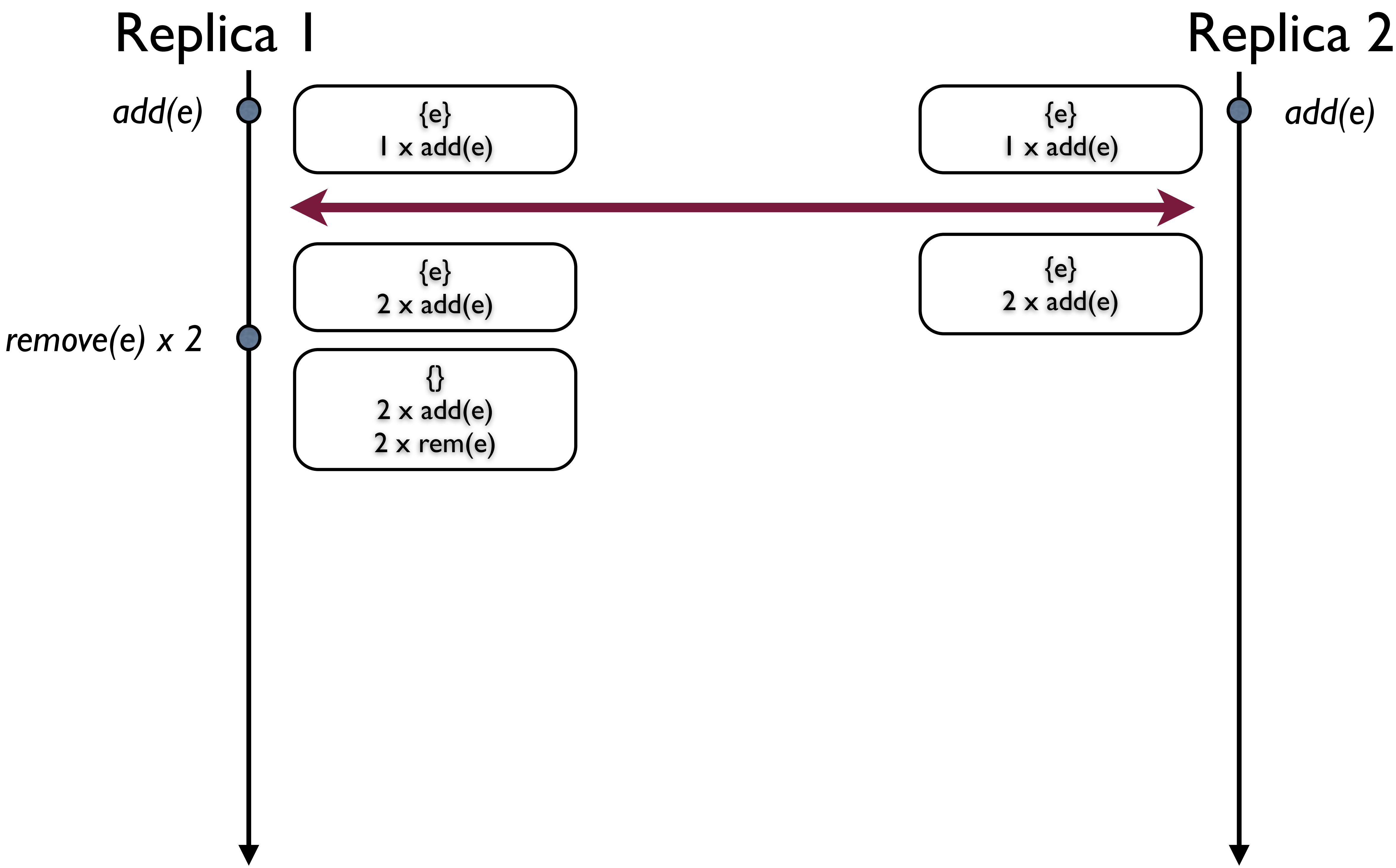
C-Set



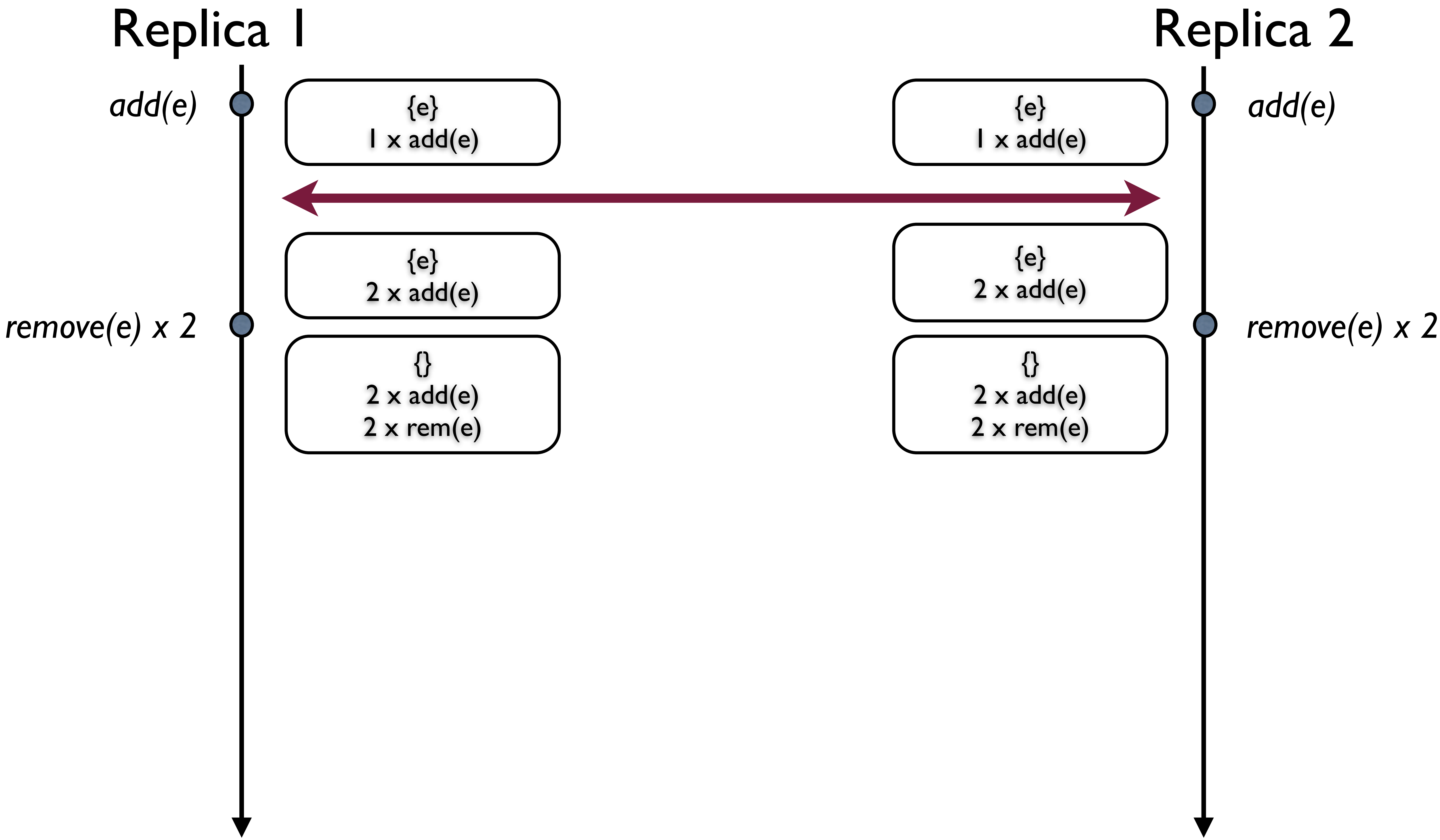
C-Set


Replica 1
Replica 2
add(e)
add(e)
{e}
1 x add(e)
{e}
1 x add(e)
{e}
2 x add(e)
{e}
2 x add(e)

C-Set

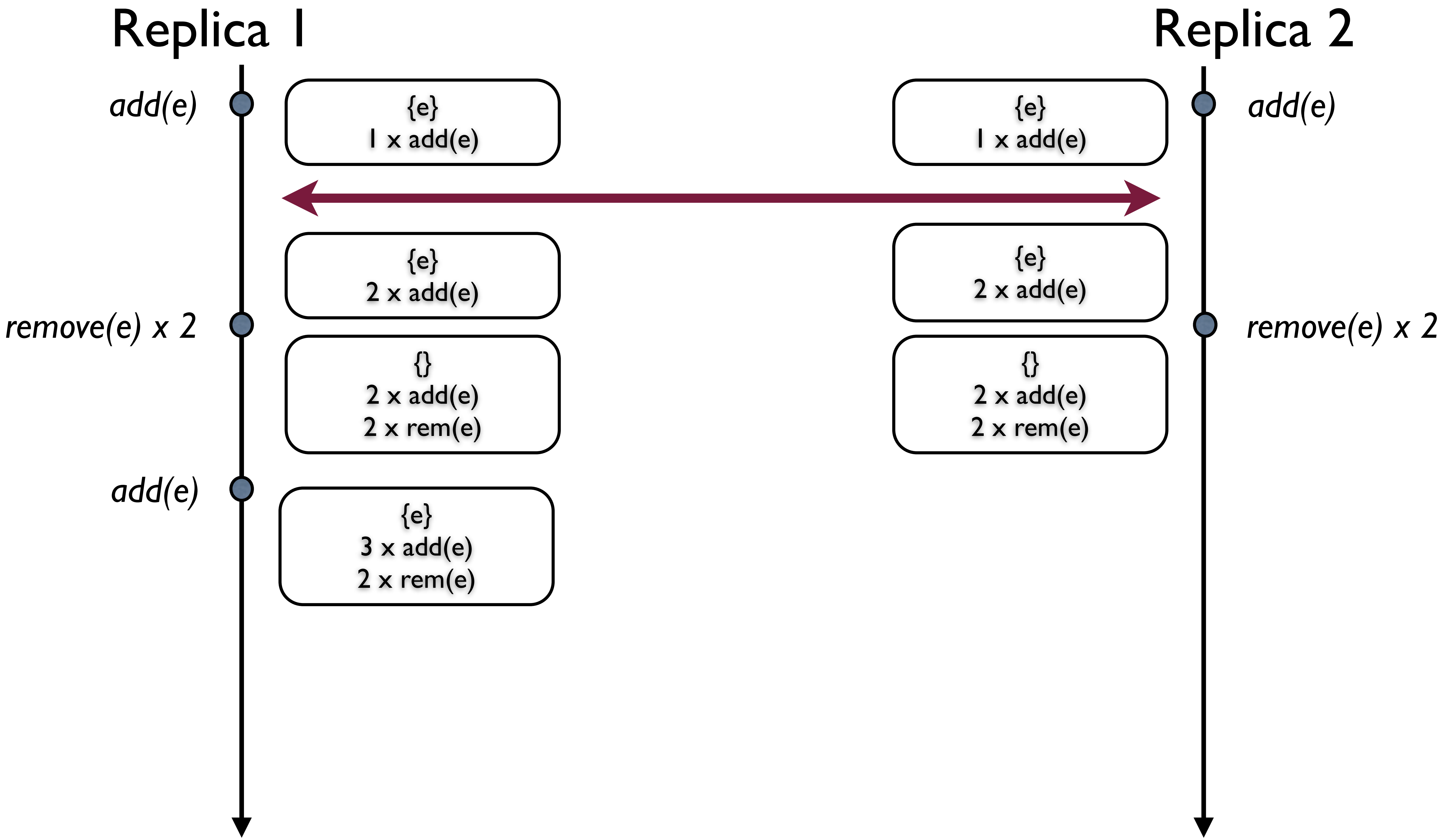

Replica 1
Replica 2
add(e)
{e}
1 x add(e)
{e}
1 x add(e)
add(e)
{e}
2 x add(e)
{e}
2 x add(e)
remove(e) x 2
{}
2 x add(e)
2 x rem(e)

C-Set



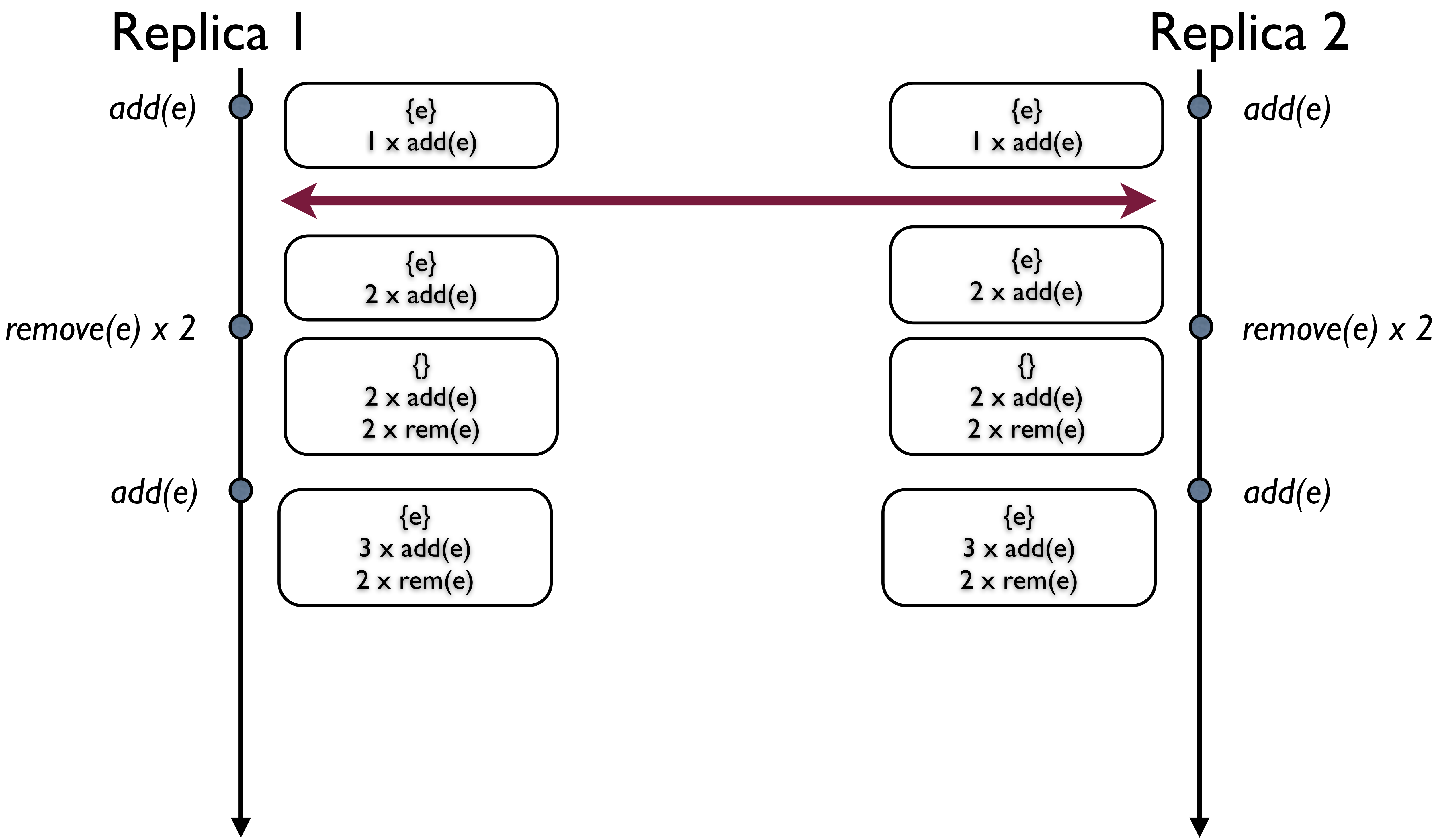
Replica 1
Replica 2
add(e)
{e}
1 x add(e)
{e}
1 x add(e)
add(e)
{e}
2 x add(e)
{e}
2 x add(e)
remove(e) x 2
remove(e) x 2
{}
2 x add(e)
2 x rem(e)
{}
2 x add(e)
2 x rem(e)

C-Set



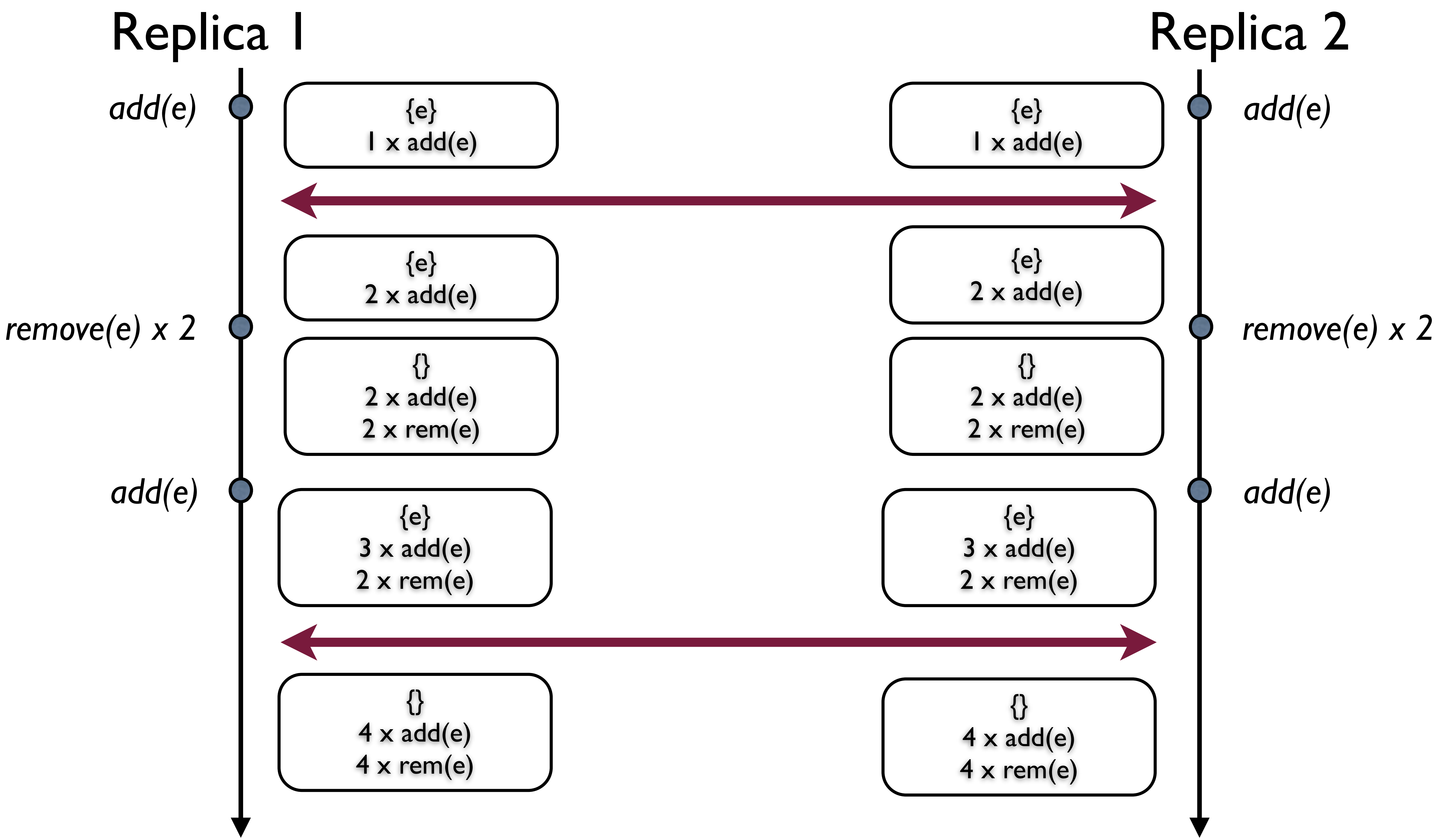
Replica 1
Replica 2
add(e)
{e}
1 x add(e)
{e}
1 x add(e)
add(e)
{e}
2 x add(e)
{e}
2 x add(e)
remove(e) x 2
remove(e) x 2
{}
2 x add(e)
2 x rem(e)
{}
2 x add(e)
2 x rem(e)
add(e)
{e}
3 x add(e)
2 x rem(e)

C-Set



Replica 1
Replica 2
add(e)
{e}
1 x add(e)
{e}
1 x add(e)
add(e)
{e}
2 x add(e)
{e}
2 x add(e)
remove(e) x 2
{}
2 x add(e)
2 x rem(e)
{}
2 x add(e)
2 x rem(e)
remove(e) x 2
add(e)
{e}
3 x add(e)
2 x rem(e)
{e}
3 x add(e)
2 x rem(e)
add(e)

C-Set



Replica 1
Replica 2
add(e)
{e}
1 x add(e)
{e}
1 x add(e)
add(e)
{e}
2 x add(e)
{e}
2 x add(e)
remove(e) x 2
remove(e) x 2
{}
2 x add(e)
2 x rem(e)
{}
2 x add(e)
2 x rem(e)
add(e)
add(e)
{e}
3 x add(e)
2 x rem(e)
{e}
3 x add(e)
2 x rem(e)
{}
4 x add(e)
4 x rem(e)
{}
4 x add(e)
4 x rem(e)

Specification of concurrent operations

- For independent operations, concurrent execution should yield the same result as the sequential execution

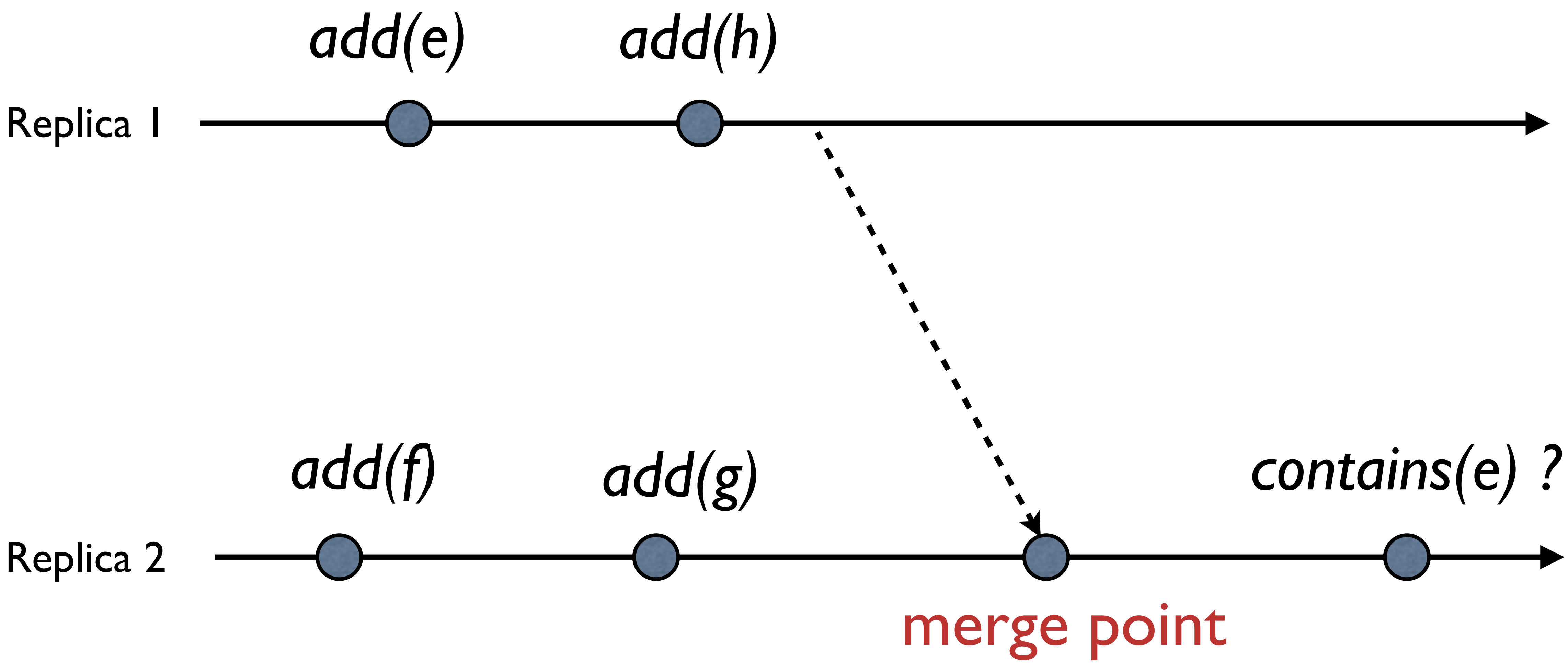
Specification of concurrent operations

- For independent operations, concurrent execution should yield the same result as the sequential execution

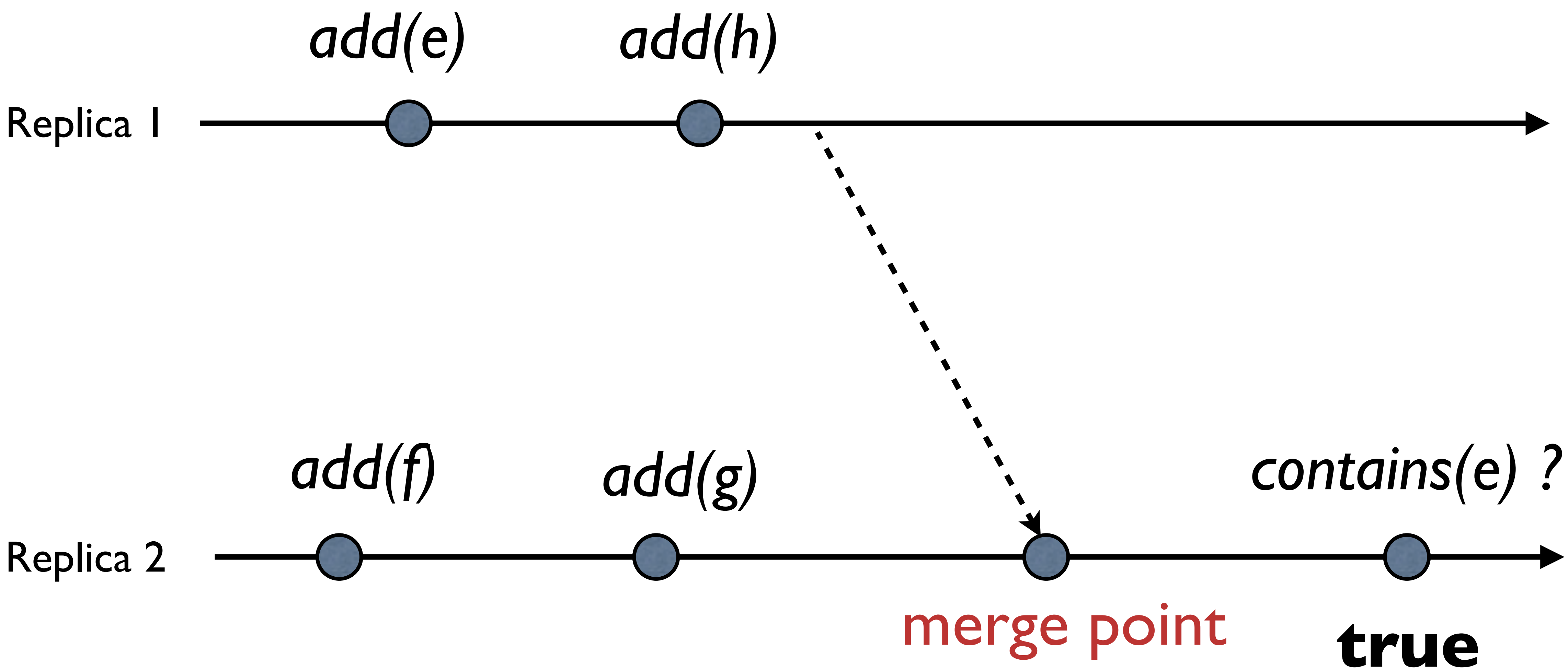
Principle of Permutation Equivalence

If *op;op'* and *op';op* have the same effect, then *op* || *op'* should also have the same effect.

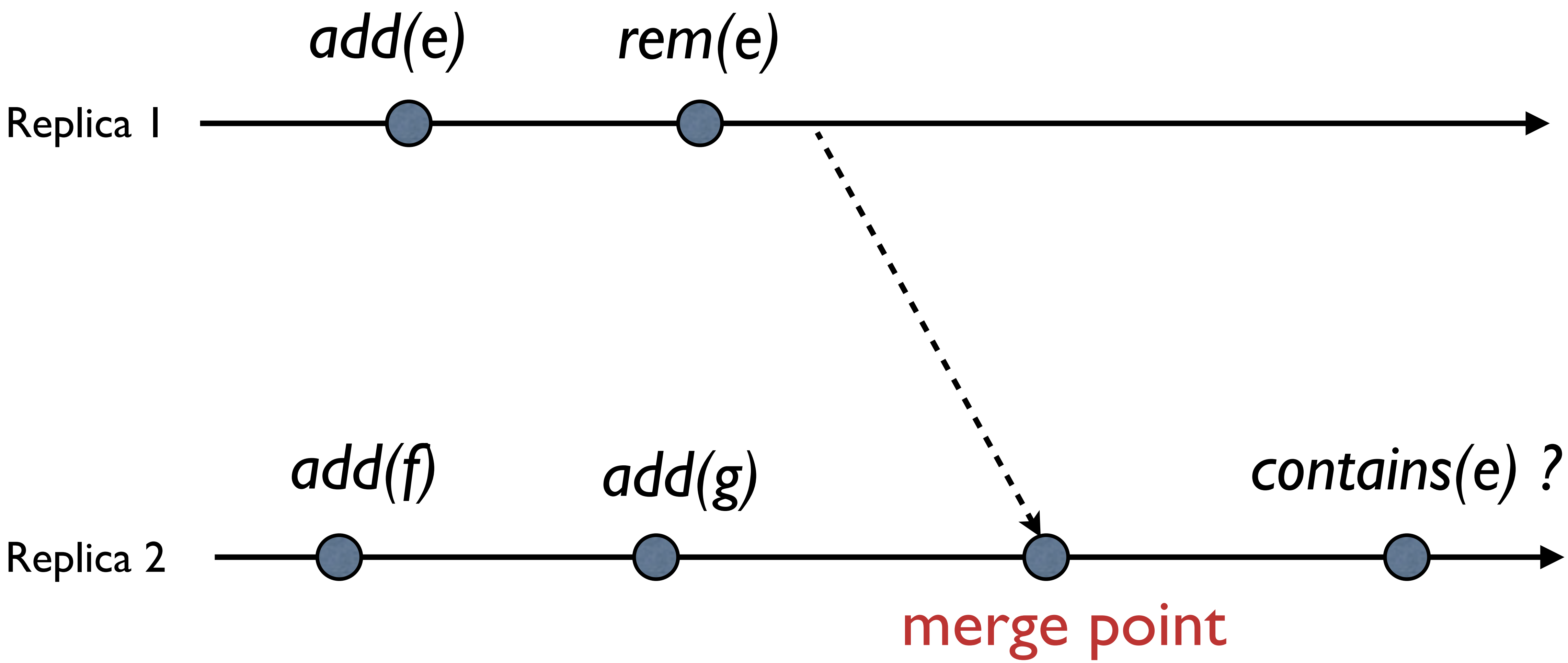
Applying the principle



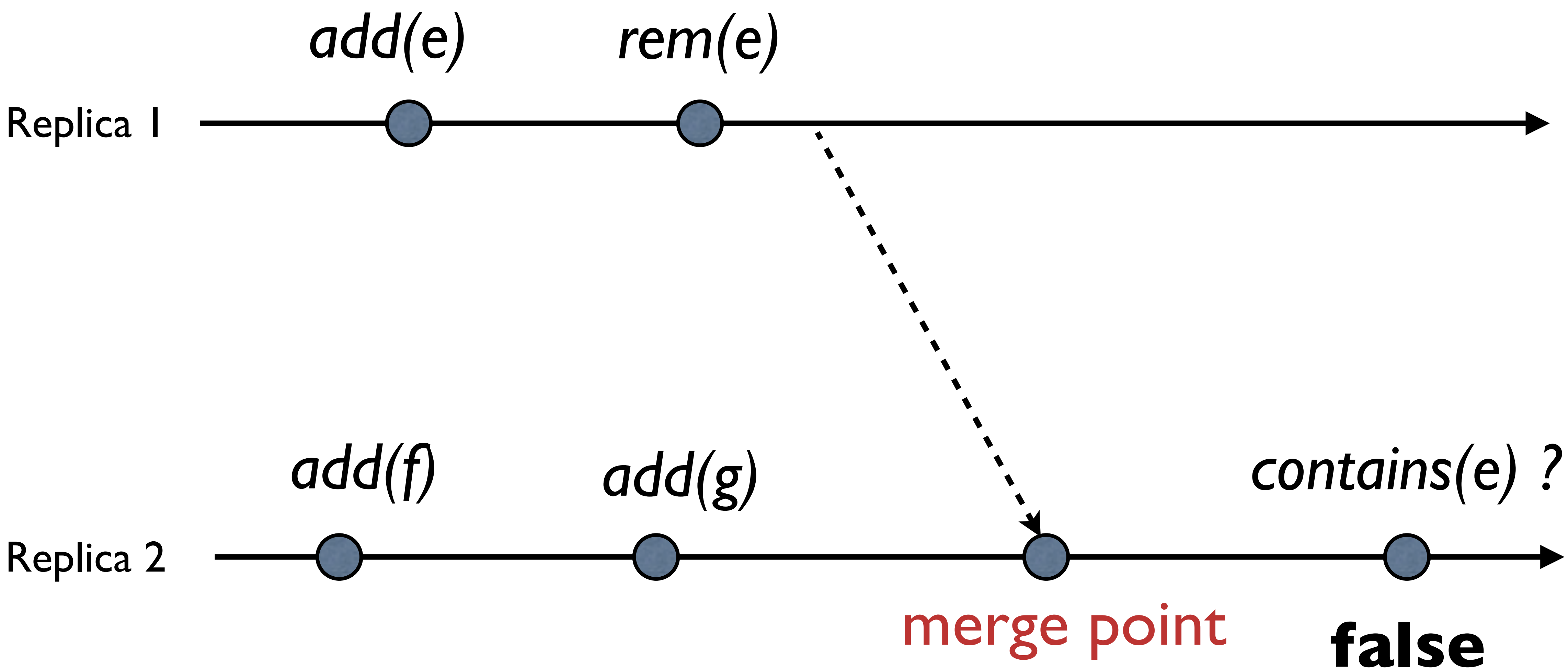
Applying the principle



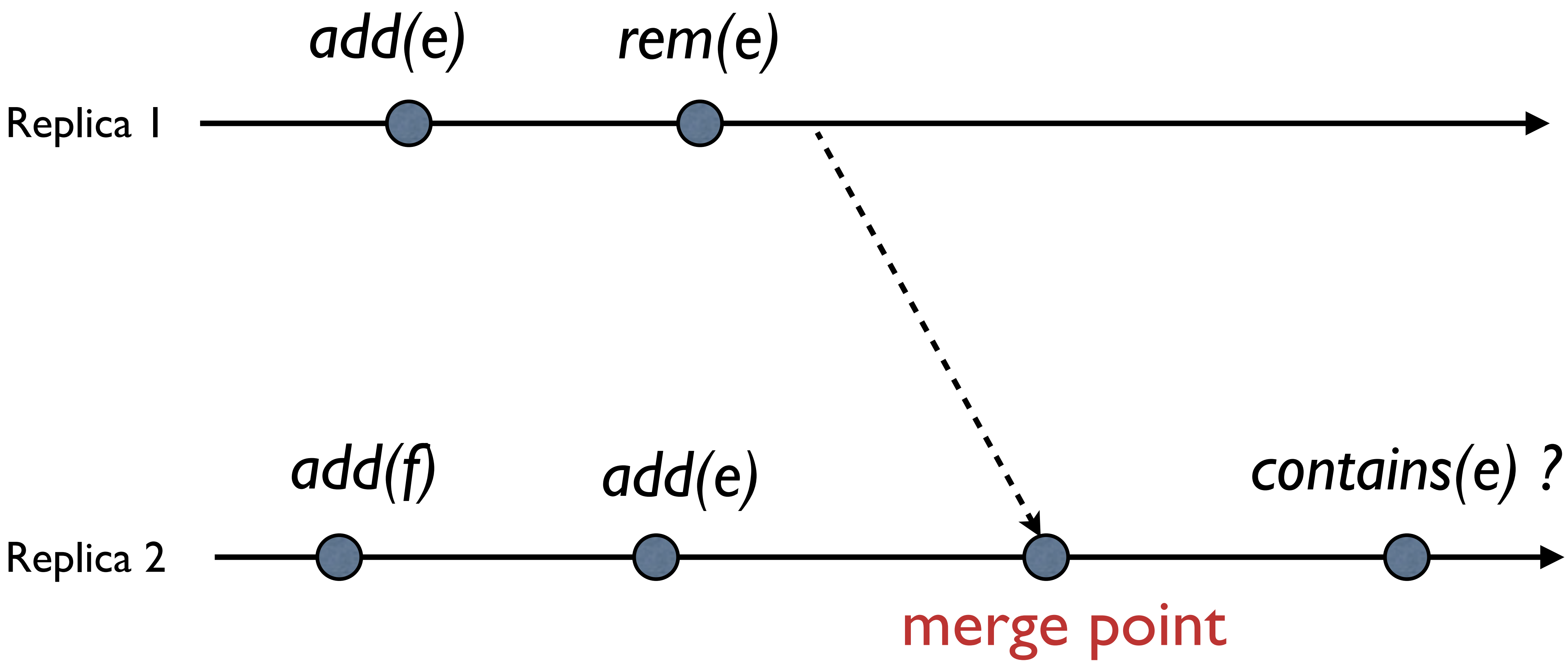
Applying the principle



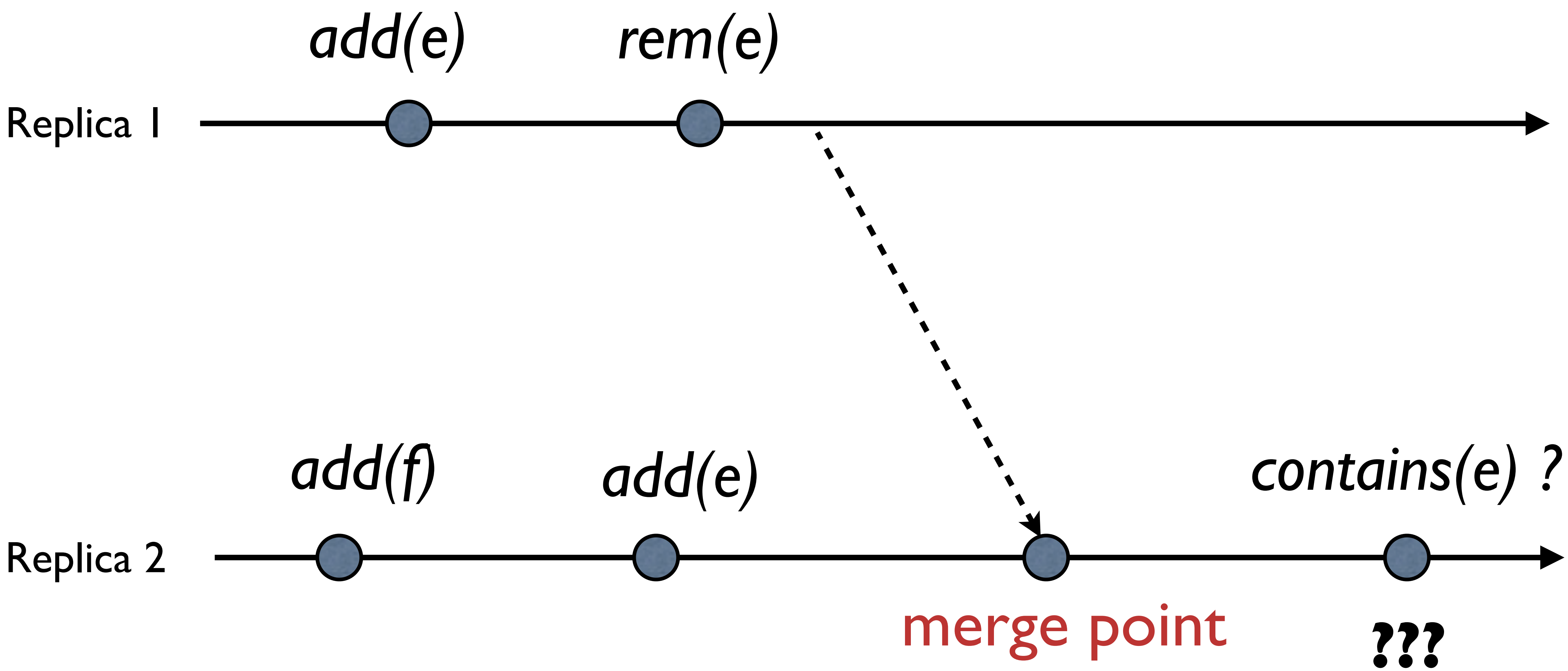
Applying the principle



Applying the principle



Applying the principle



Set semantics

The principle of permutation equivalence requires that

Set semantics

The principle of permutation equivalence requires that

{true} add(e) || add(f) {e,f ∈ S}

Set semantics

The principle of permutation equivalence requires that

{true}	add(e) || add(f)	$\{e,f \in S\}$
{true}	add(e) || rem(f)	$\{e \in S \wedge f \notin S\}$

Set semantics

The principle of permutation equivalence requires that

{true}	add(e) || add(f)	$\{e,f \in S\}$
{true}	add(e) || rem(f)	$\{e \in S \wedge f \notin S\}$
{true}	rem(e) || rem(f)	$\{e,f \notin S\}$

Set semantics

The principle of permutation equivalence requires that

{true}	add(e) || add(f)	$\{e, f \in S\}$
{true}	add(e) || rem(f)	$\{e \in S \wedge f \notin S\}$
{true}	rem(e) || rem(f)	$\{e, f \notin S\}$
{true}	add(e) || add(e)	$\{e \in S\}$

Set semantics

The principle of permutation equivalence requires that

{true}	add(e) || add(f)	$\{e,f \in S\}$
{true}	add(e) || rem(f)	$\{e \in S \wedge f \notin S\}$
{true}	rem(e) || rem(f)	$\{e,f \notin S\}$
{true}	add(e) || add(e)	$\{e \in S\}$
{true}	rem(e) || rem(e)	$\{e \notin S\}$

Semantics of add(e) || rem(e)

Semantics of add(e) || rem(e)

$\{\perp_e \in S\}$

Error indicator

Version control systems

Semantics of add(e) || rem(e)

$\{\bot_e \in S\}$	Error indicator	Version control systems
$\{e \in S\}$	Add wins	Shopping cart

Semantics of add(e) || rem(e)

$\{\bot_e \in S\}$	Error indicator	Version control systems
$\{e \in S\}$	Add wins	Shopping cart
$\{e \notin S\}$	Remove wins	Shopping cart

Semantics of add(e) || rem(e)

$\{\perp_e \in S\}$	Error indicator	Version control systems
$\{e \in S\}$	Add wins	Shopping cart
$\{e \notin S\}$	Remove wins	Shopping cart
$\{add(e) > rem(e) \Rightarrow e \in S$ $\wedge$ $rem(e) > add(e) \Rightarrow e \notin S\}$	Last Writer wins	Read/write registers

OR-SET

payload set E, set T -- elements and tombstones: sets of pairs (element, id)

query **contains** (element e) : boolean b
let b = (∃n : (e, n) ∈ E)

update **add** (element e)
prepare (e)
let n = unique() -- unique() returns a unique tag
effect (e, n)
E := E ∪ {(e,n)} \ T

update **remove** (element e)
prepare (e)
let R = {(e, n) | ∃n : (e, n) ∈ E} -- collect all unique pairs containing e
effect (R)
E := E \ R
T := T ∪ R -- remove pairs observed at source

OR-SET

payload set E, set T

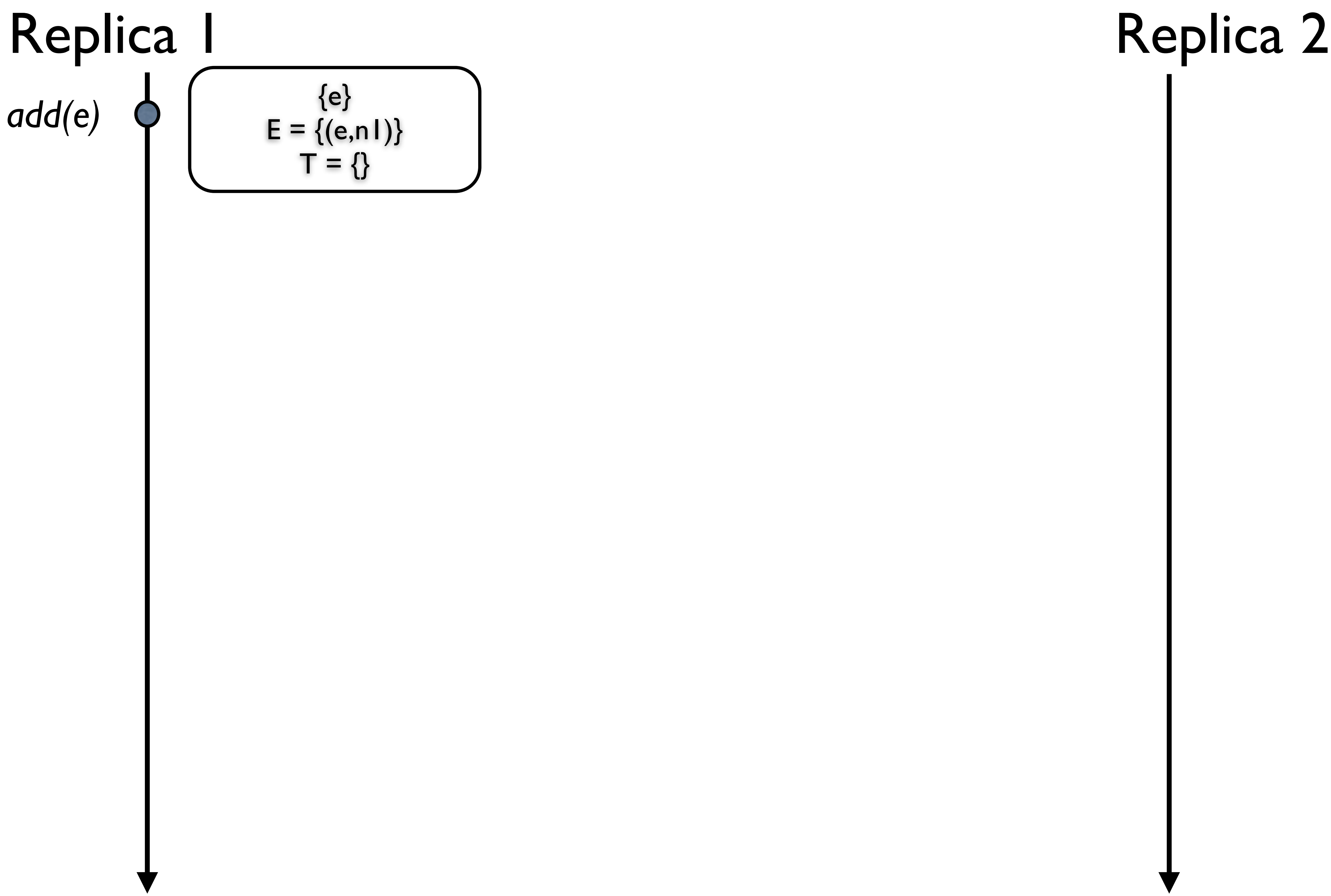
compare (ORSet B) : boolean b
  let b = $(E \cup T) \subseteq (B.E \cup B.T) \ \wedge \ T \subseteq B.T$

merge (OrSet B)
  $E := (E \setminus B.T) \cup (B.E \setminus T)$
  $T := T \cup B.T$

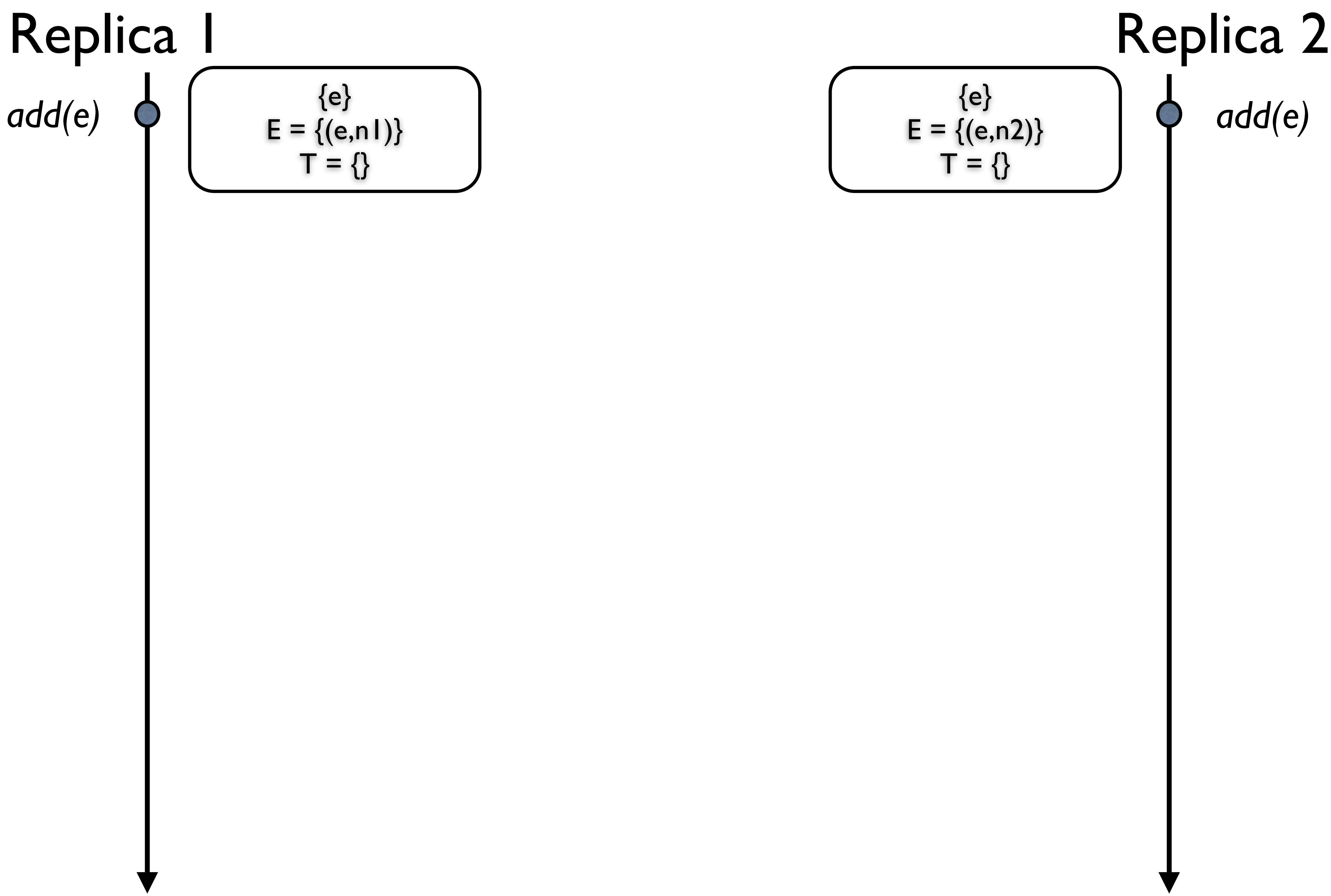
- Specification supports mixing of state-based and operation-based updating!

Example

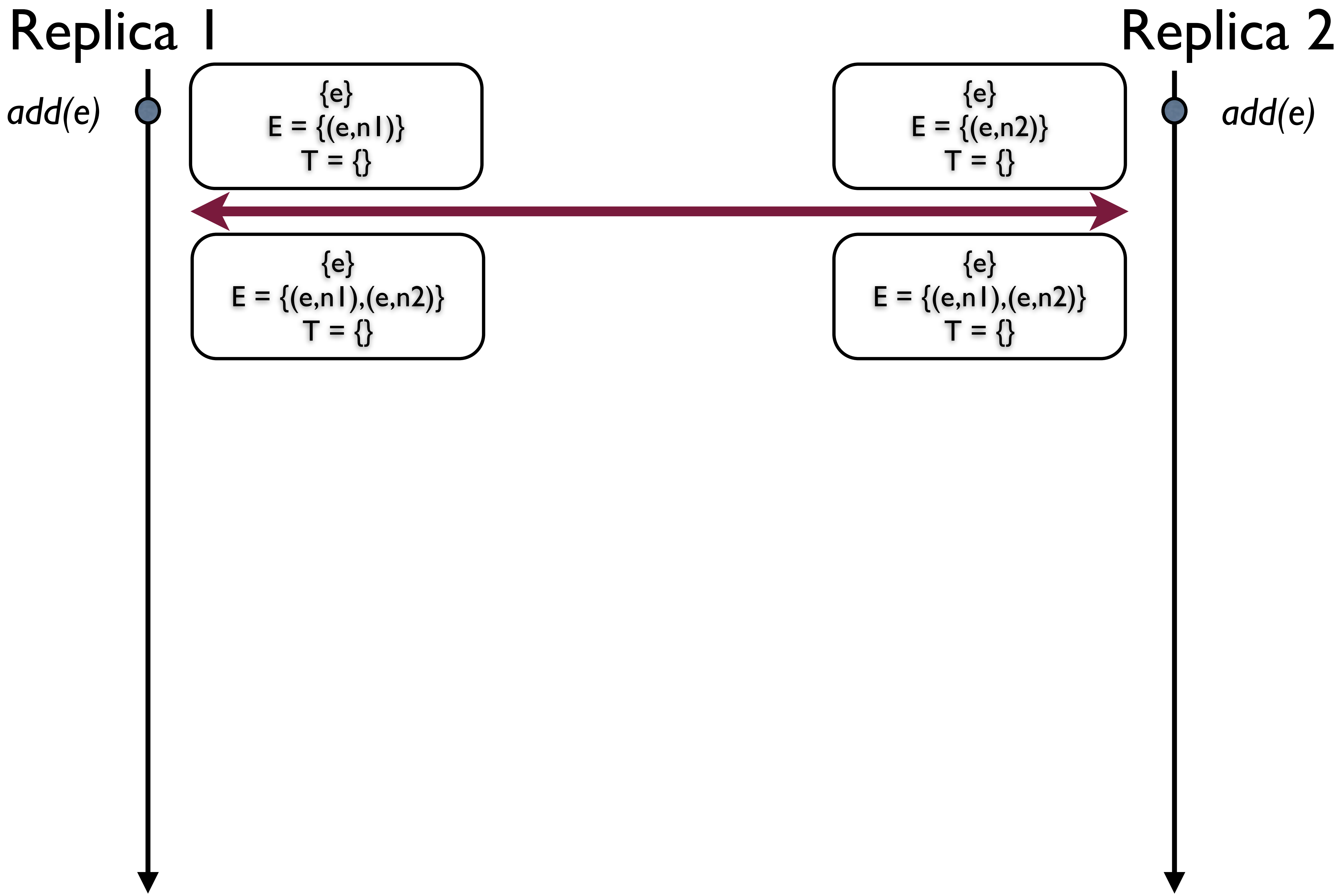
Example



Example

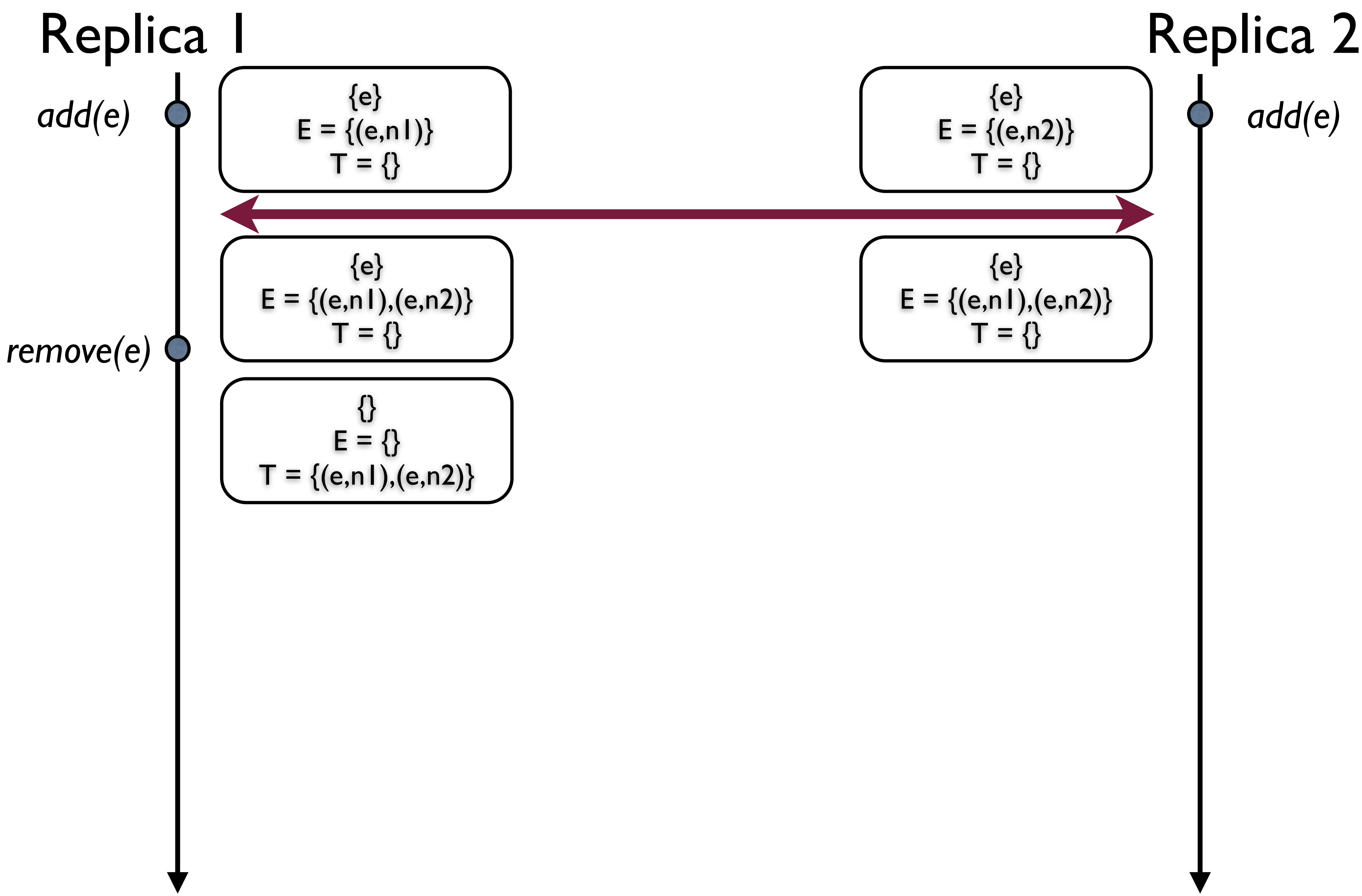


Example

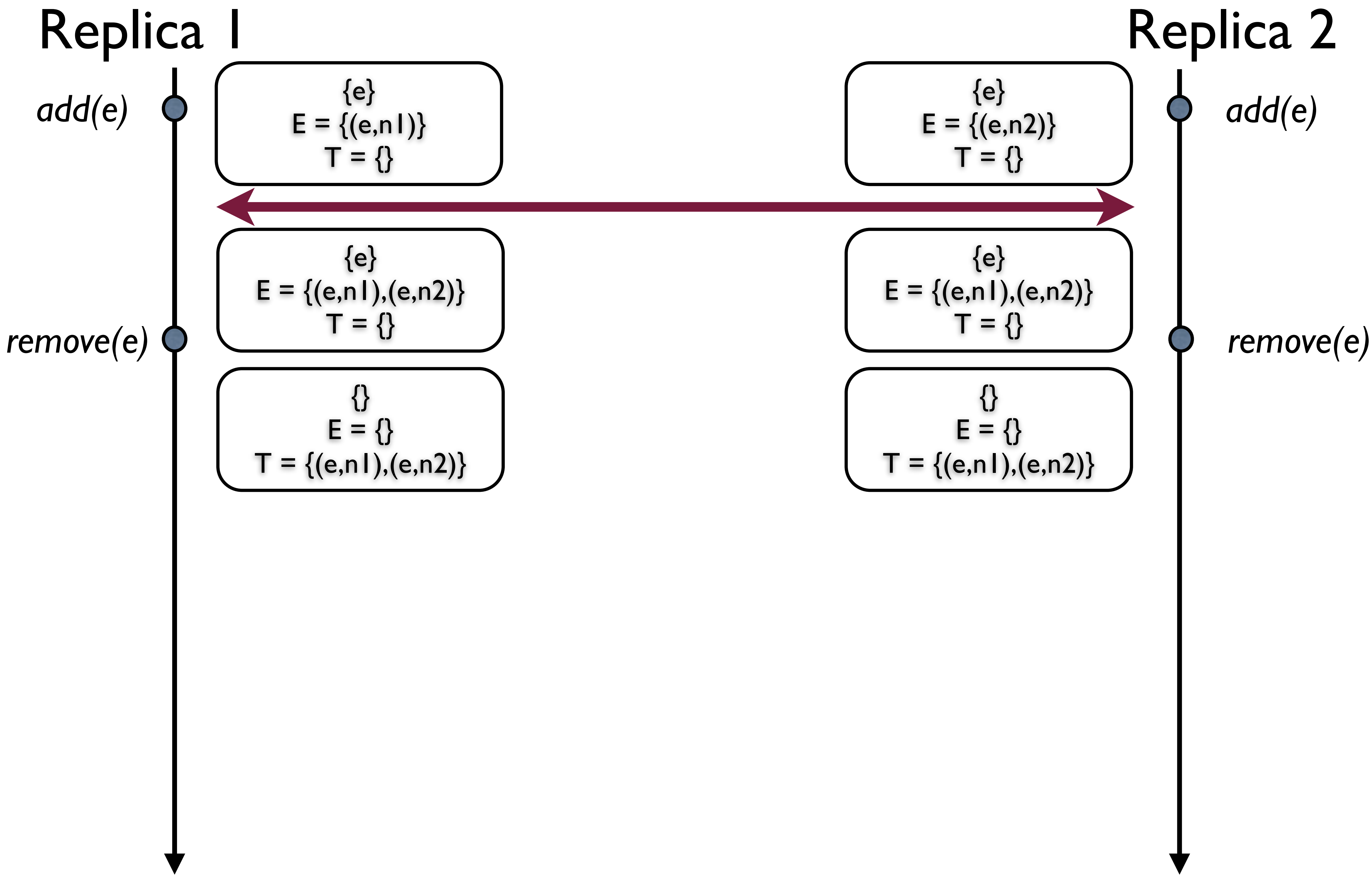


Replica 1
Replica 2
add(e)
add(e)
{e}
E = {(e,n1)}
T = {}
{e}
E = {(e,n2)}
T = {}
{e}
E = {(e,n1),(e,n2)}
T = {}
{e}
E = {(e,n1),(e,n2)}
T = {}

Example

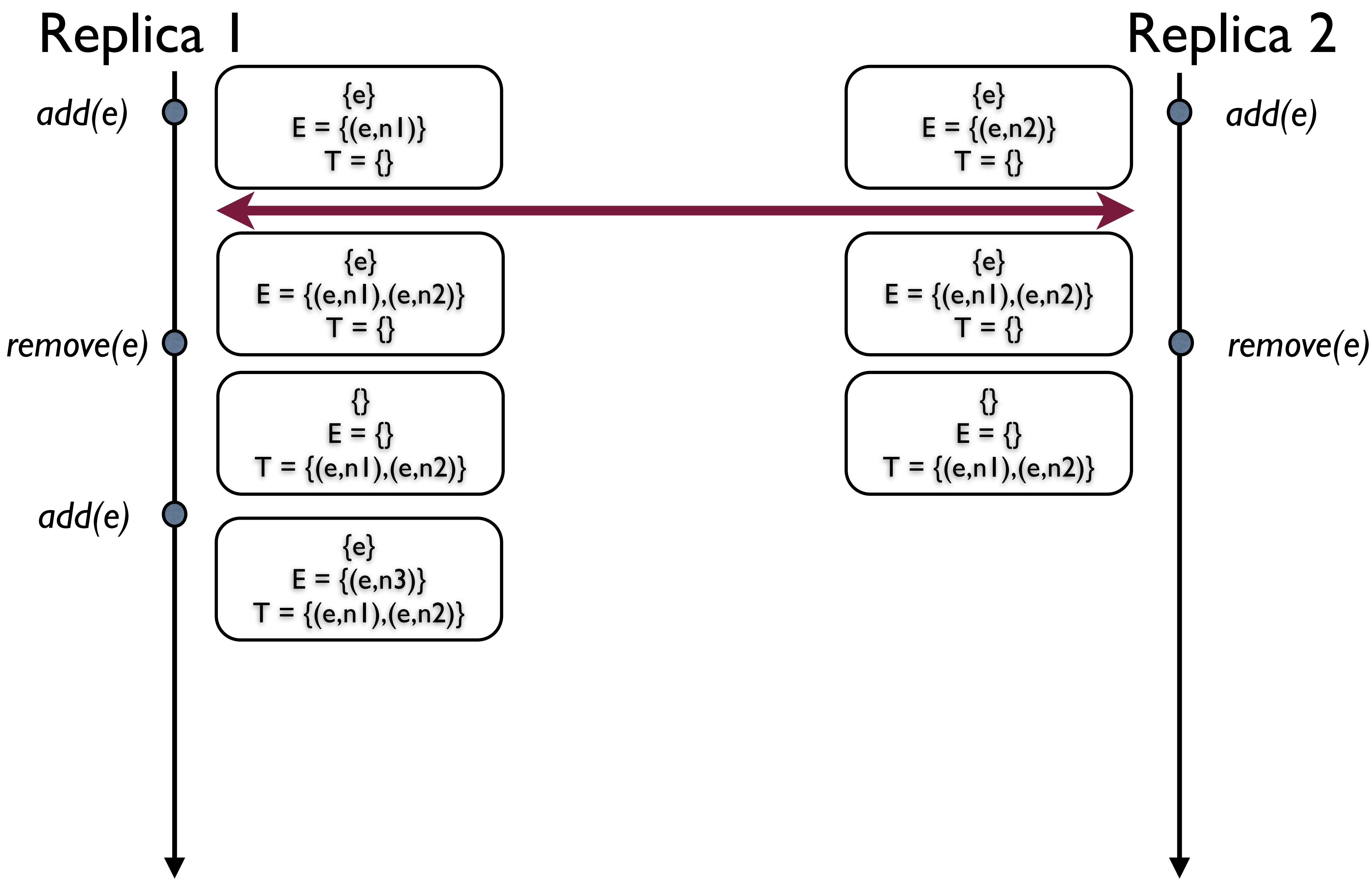

Replica 1
Replica 2
add(e)
{e}
E = {(e,n1)}
T = {}
{e}
E = {(e,n2)}
T = {}
add(e)
{e}
E = {(e,n1),(e,n2)}
T = {}
{e}
E = {(e,n1),(e,n2)}
T = {}
remove(e)
{}
E = {}
T = {(e,n1),(e,n2)}

Example

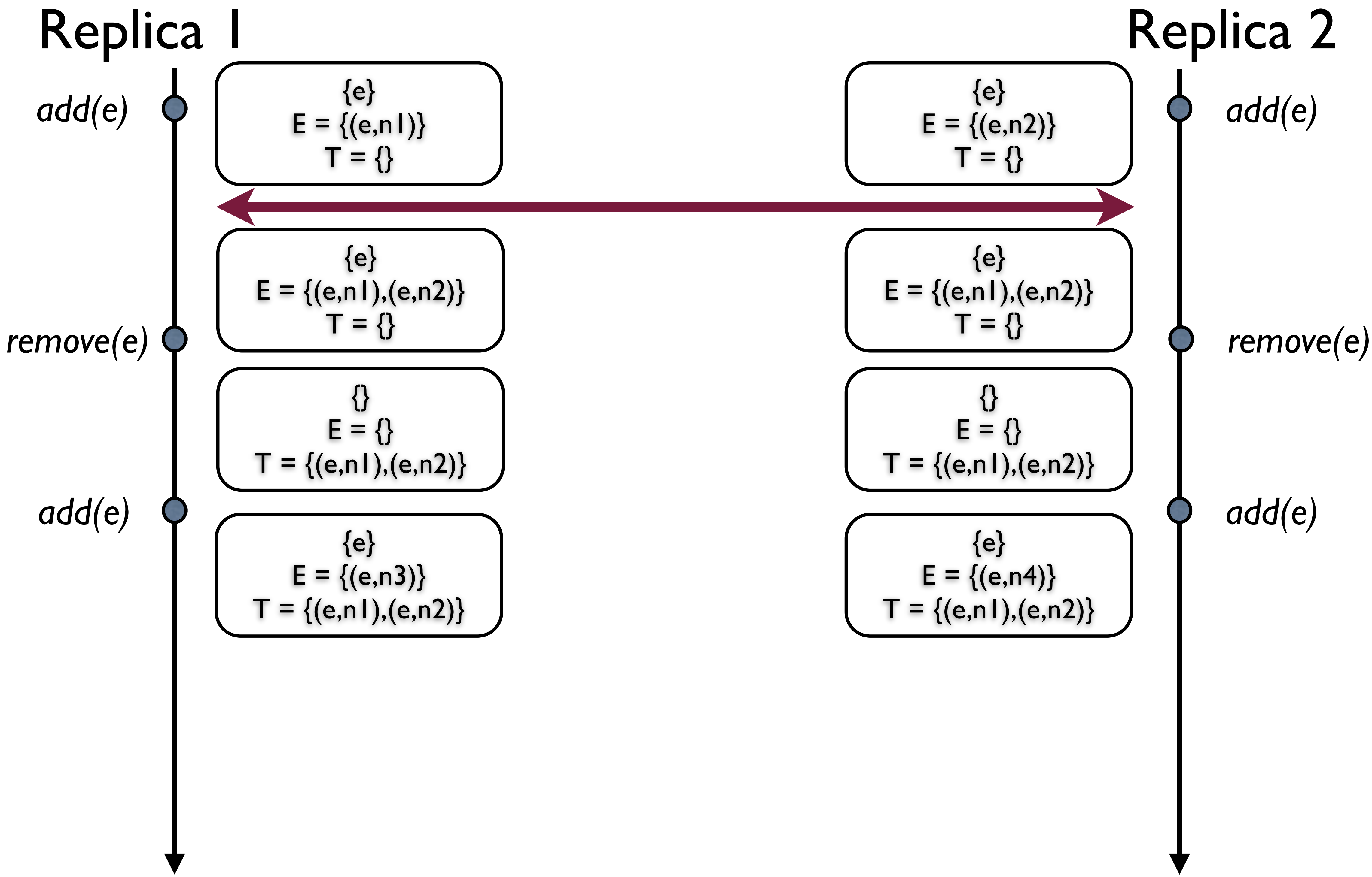


Replica 1
Replica 2
add(e)
{e}
E = {(e,n1)}
T = {}
{e}
E = {(e,n2)}
T = {}
add(e)
{e}
E = {(e,n1),(e,n2)}
T = {}
{e}
E = {(e,n1),(e,n2)}
T = {}
remove(e)
remove(e)
{}
E = {}
T = {(e,n1),(e,n2)}
{}
E = {}
T = {(e,n1),(e,n2)}

Example

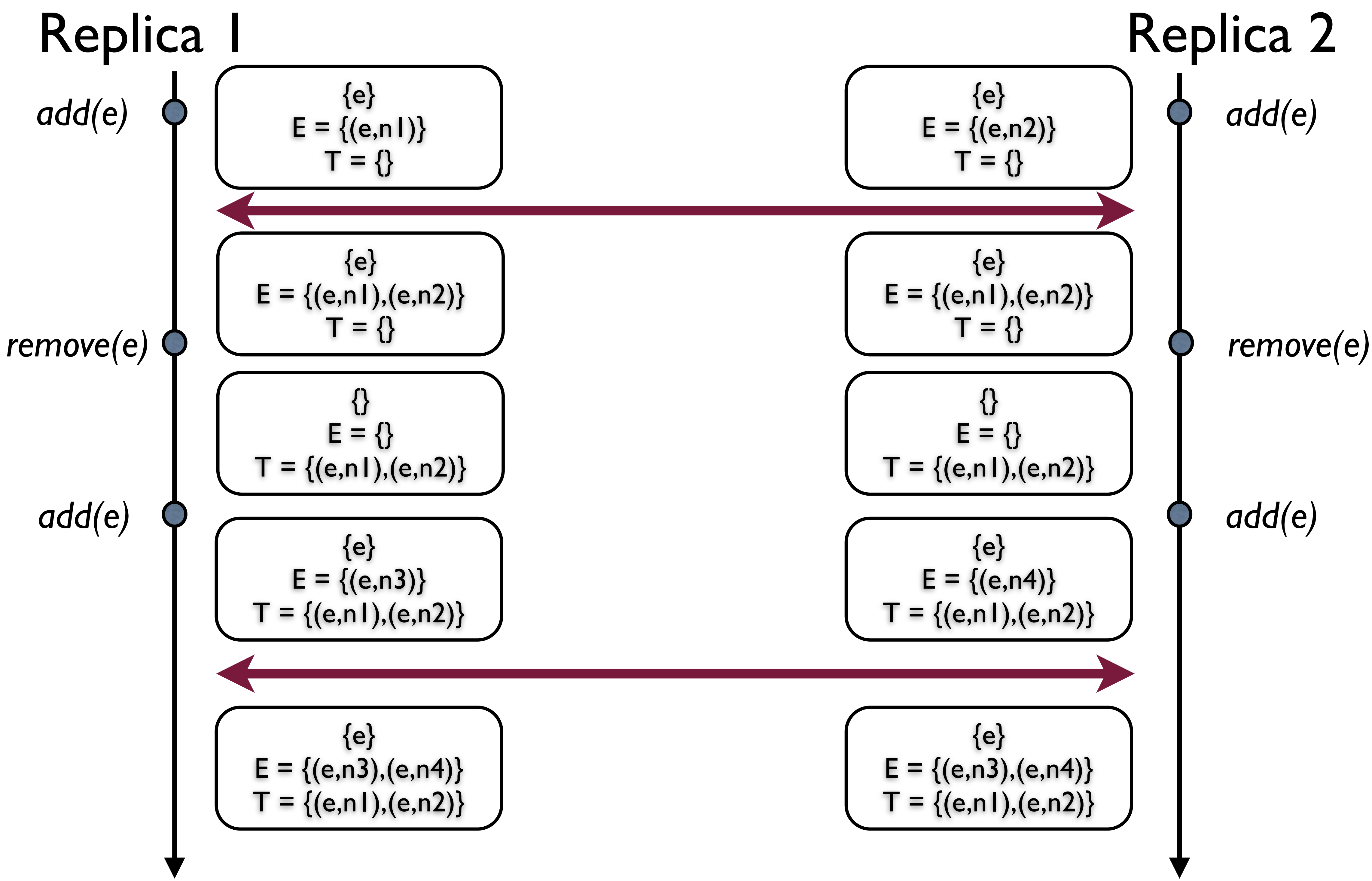

Replica 1
Replica 2
add(e)
{e}
E = {(e,n1)}
T = {}
{e}
E = {(e,n2)}
T = {}
add(e)
{e}
E = {(e,n1),(e,n2)}
T = {}
{e}
E = {(e,n1),(e,n2)}
T = {}
remove(e)
remove(e)
{}
E = {}
T = {(e,n1),(e,n2)}
{}
E = {}
T = {(e,n1),(e,n2)}
add(e)
{e}
E = {(e,n3)}
T = {(e,n1),(e,n2)}

Example



Replica 1
Replica 2
add(e)
{e}
E = {(e,n1)}
T = {}
{e}
E = {(e,n2)}
T = {}
add(e)
{e}
E = {(e,n1),(e,n2)}
T = {}
{e}
E = {(e,n1),(e,n2)}
T = {}
remove(e)
remove(e)
{}
E = {}
T = {(e,n1),(e,n2)}
{}
E = {}
T = {(e,n1),(e,n2)}
add(e)
add(e)
{e}
E = {(e,n3)}
T = {(e,n1),(e,n2)}
{e}
E = {(e,n4)}
T = {(e,n1),(e,n2)}

Example



Replica 1
Replica 2
add(e)
{e}
E = {(e,n1)}
T = {}
{e}
E = {(e,n2)}
T = {}
add(e)
{e}
E = {(e,n1),(e,n2)}
T = {}
{e}
E = {(e,n1),(e,n2)}
T = {}
remove(e)
remove(e)
{}
E = {}
T = {(e,n1),(e,n2)}
{}
E = {}
T = {(e,n1),(e,n2)}
add(e)
add(e)
{e}
E = {(e,n3)}
T = {(e,n1),(e,n2)}
{e}
E = {(e,n4)}
T = {(e,n1),(e,n2)}
{e}
E = {(e,n3),(e,n4)}
T = {(e,n1),(e,n2)}
{e}
E = {(e,n3),(e,n4)}
T = {(e,n1),(e,n2)}

Tombstone management

- Idea: Garbage-collect tombstones that have been delivered everywhere
  - Requires reliable membership service to deliver acknowledgements
  - Increases distributed processing
- Can we do better?

Tombstone management

- Insight: Adding an element (e,n) always happens-before removing (e,n)
  - For operation-based ORSet, no tombstones required!
  - To support merge, we need to record the happens-before information using version vectors (requires causal delivery)
- Trick can also be applied to other CRDTs

ORSWOT

```
payload set E, vector v        -- E: elements, set of triples (element e, timestamp c, replica i)
initial ∅,[0,...,0]            -- v: vector (summary) of received process identifiers, indexed by replica

query contains (element e) : boolean b
    let b = (∃c, i : (e, c, i) ∈ E)

update add (element e)
    prepare (e)
        let r = myID()               -- r = source replica
        let c = v[r] + 1
    effect (e, c, r)                 -- requires causal delivery
        if c > v[r] then
            v[r] := c
            E := E ∪ {(e, c, r)}

update remove (element e)
    prepare (e)
        let R = {(e, c, i) ∈ E}      -- Collect all unique triples containing e
    effect (R)
        E := E \ R
```

ORSWOT

compare (A, B) : boolean b

let R = $\{(c,i) \mid 0 < c \leq A.v[i] \wedge \nexists\, e : (e,c,i) \in A.E\}$ -- Compare sets of removed ids

let R′ = $\{(c,i) \mid 0 < c \leq B.v[i] \wedge \nexists\, e : (e,c,i) \in B.E\}$

let b = $A.v \leq B.v \wedge R \subseteq R'$

merge (B)

let M = $E \cap B.E$

let M′ = $\{(e,c,i) \in E \setminus B.E \mid c > B.v[i]\}$

let M′′ = $\{(e,c,i) \in B.E \setminus E \mid c > v[i]\}$

$E := M \cup M' \cup M''$

$v := [\max(v[0], B.v[0]), \ldots, \max(v[n], B.v[n])]$

Preliminary evaluation

	Write probability					
	0	0,2	0,4	0,6	0,8	1
HashSet	1228	1251	1266	1239	1249	1235
C-Set	1228	980	849	818	697	672
ORSWOT	1241	1223	1167	1110	1074	1036

Throughput (K operations per second) on single machine

Summary

- Principle of Permutation Equivalence
- Concurrency semantics for set operations
- Several specifications for sets unifying state- and operation-based approach
- Efficient tombstone management
- Results are available as Technical Report

Remove-Wins Set

payload set E, set T (*initial* ∅, ∅) -- sets of pairs { (element e, unique-tag n), . . . }

query **contains** (element e) : boolean b
let b = (∃ (e,n) ∈ E ∧ ∄n′ : (e,n′) ∈ T)

update **add** (element e)
prepare (e) -- Collect all unique pairs of tombstones containing e
let R′ = {(e, n) | ∃n : (e, n) ∈ T }
if R′ = ∅ then R = {(e, unique())}
else R = R′
effect (R) -- Remove pairs of tombstones observed at source
E := E ∪ R
T := T \ R

update **remove** (element e)
prepare (e)
let n = unique() -- unique() returns a unique tag
effect (e,n)
T := T ∪ {(e,n)} \ E

Last-Writer-Wins Set

payload set E (*initial* $\emptyset$) -- sets of tuples { (element e, time t, boolean inSet), . . . }

query **contains** (element e) : boolean b
let b = ($\exists$ (e, t, true) $\in$ E)

update **add** (element e)
prepare (e)
let t = unique() -- unique() returns unique time
effect (e, t)
T := {(e,t′,v) $\in$ E}
if $\forall$ (e, t′, v) $\in$ T : t′ < t then E := E \ T $\cup$ {(e, t, true)}

update **remove** (element e)
prepare (e)
let t = unique() -- unique() returns unique time
effect (e, t)
T := {(e,t′,v) $\in$ E}
if $\forall$(e, t′, v) $\in$ T : t′ < t then E := E \ T $\cup$ {(e, t, false)}