Programming with weak synchronization models

Guest lecture ID2203

Peter Van Roy
Christopher Meiklejohn
Annette Bieniusa

LIGHTKONE

Overview of the lesson

- Motivation and principles
  - “As easy as strong consistency, as efficient as weak consistency”
  - A sweet spot: Strong Eventual Consistency
- Convergent data structures
  - Conflict-free replicated data types (CRDTs)
- Lasp
  - Programming language and platform based on composing CRDTs
- Antidote
  - Causal transactional database based on CRDTs

Guest lecturers

- This lesson is brought to you by:
  - Peter Van Roy, Université catholique de Louvain
  - Christopher Meiklejohn, Université catholique de Louvain
  - Annette Bieniusa, Technische Universität Kaiserslautern

- This research is being done in two European projects:

SyncFree

EU FP7
2013-2016

LightKone

EU H2020
2017-2019

Both easy and efficient

- One of the holy grails of distributed systems is to make them both easy to program and efficient to execute
- Strong consistency (linearizability) is easy to program but inefficient
- Eventual consistency (operations eventually complete) is efficient to execute but hard to program
- Can we get the best of both worlds?
  - Synchronization-free programming aims to combine the ease of strong consistency with the efficiency of eventual consistency
  - How can this work?

LIGHTKONE

Back to basics

- Distributed system = a collection of networked computing nodes that behaves like one system (= consistency model)
- To make this work, the nodes will coordinate with each other according to well-defined rules (= synchronization algorithm)
- For example, a reliable broadcast algorithm guarantees the all-or-none property: all correct nodes deliver, or none do

LIGHTKONE

How far can we go?

- We would like the consistency model to be as strong as possible (easy to program) and the synchronization algorithm to be as weak as possible (efficient to execute)
- Let’s try the extreme case: the weakest possible synchronization is no synchronization (no rules), which enforces no consistency at all!
  - So it’s clear we need *some* synchronization
  - How little can we get away with?

LIGHTKONE

A sweet spot: SEC

- Strong Eventual Consistency (SEC)
  - The data structure is defined so that n replicas that receive the same updates (in any order) have equivalent state
  - Synchronization is eventual replica-to-replica communication

- This consistency model is surprisingly powerful
  - It supports a programming model that resembles a concurrent form of functional programming
  - It handles both nondeterminism and nonmonotonicity
  - It has an efficient, resilient implementation

LIGHTKONE

Let’s exploit SEC!

- In the rest of the lesson we will see how far we can go with Strong Eventual Consistency
  - Convergent data structures (CRDTs)
  - Programming by composing CRDTs (Lasp)
  - Causally consistent transactions on CRDTs (Antidote)
  - Applications

LIGHTKONE

Convergent data structures

- We can define distributed data structures that obey Strong Eventual Consistency
  - One approach: Conflict-free Replicated Data Type (CRDT)
- Many CRDTs exist and have millions of users

CRDT definition

- A *state-based CRDT* is defined as a triple $((s_1, \ldots, s_n), m, q)$:
  - $(s_1, \ldots, s_n)$ is the configuration on n replicas, with $s_i \in S$ where S is a join semilattice
  - $q_i: S \longrightarrow V$ is a query function (read operation)
  - $m_i: S \longrightarrow S$ is a mutator (update operation) such that $s \sqsubseteq m(s)$
  - Periodically, replicas update each other's state: $\forall$ i,j: $s_i' = s_i \sqcup s_j$
- Because the mutator only inflates the value, and because of the periodic dissemination, all replicas will eventually converge to the same final value

LIGHTKONE

Join semilattice

- A *join-semilattice* is a partially ordered set S that has a least upper bound (join) for any nonempty finite subset:
  - Partial order: $\forall x, y, z \in S$:
    Reflexivity: $x \sqsubseteq x$
    Antisymmetry: $x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x=y$
    Transitivity: $x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
  - Least upper bound (join): $\forall x, y \in S$: $x \sqcup y \in S$
    - $z=x \sqcup y$ is an upper bound
    - All other upper bounds are at least as large as z

LIGHTKONE

CRDTs satisfy SEC

- Strong Eventual Consistency (SEC)
  - We assume eventual delivery: an update delivered at some correct replica is eventually delivered to all correct replicas
    - Eventual replica-to-replica communication satisfies this
  - An object is SEC if all correct replicas that have delivered the same updates have equivalent state
- Theorem: A state-based CRDT satisfies SEC
  - Proof by induction on the causal histories of deliveries at the replicas
  - Proof given in INRIA Research Report RR-7687 (see bibliography)

LIGHTKONE

Example: Grow-Only Counter

- Each replica i stores $s=(c_1, c_2, \ldots, c_i, \ldots, c_n)$ where $c_i \in N$ (natural)
- Each replica accepts inc, val, and $\sqcup$ (join) operations
  - inc_i: update s to s' where $s'=(c_1, c_2, \ldots, c_i+1, \ldots, c_n)$
  - val_i: return $\sum_{j \in i} s.j$
  - join: $s \sqcup s' = (\max(c_1,c_1'), \ldots, \max(c_n,c_n'))$

- How does this work?
  - The state vector stores the increments done at each replica
  - Eventually, all replicas' vectors will converge to know all increments

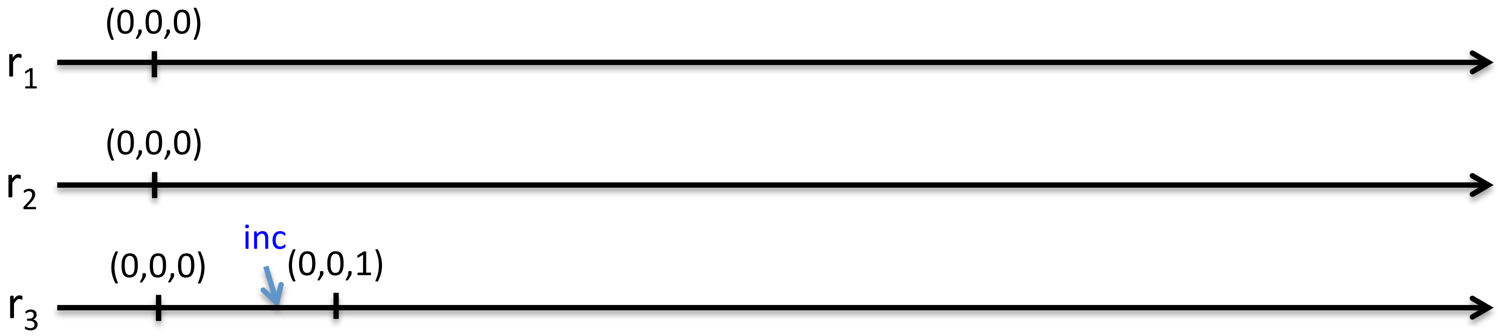
LIGHTKONE

Example execution

- Three replicas, each replica stores a 3-vector giving the increments it knows of at each replica

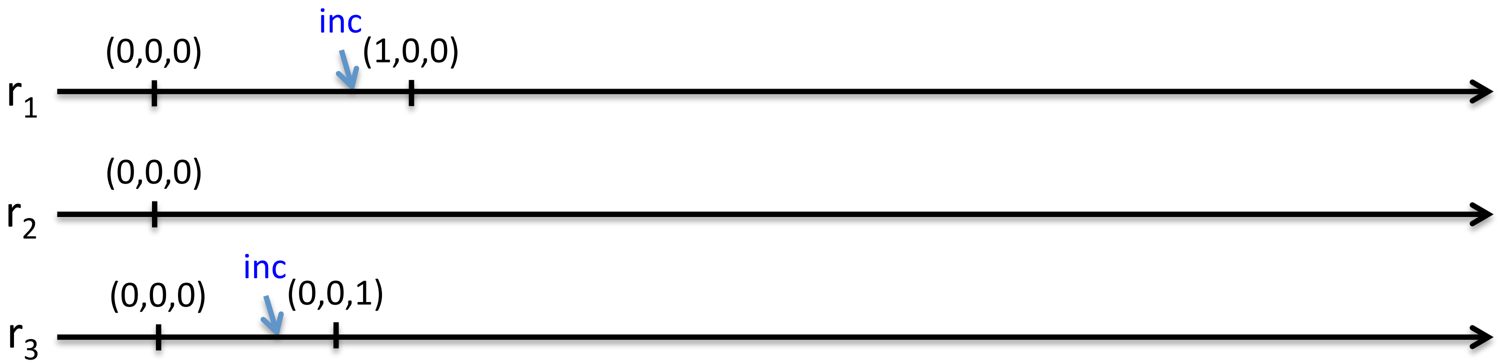
LIGHTKONE

Example execution



- Increment at replica 3, its vector becomes (0,0,1)

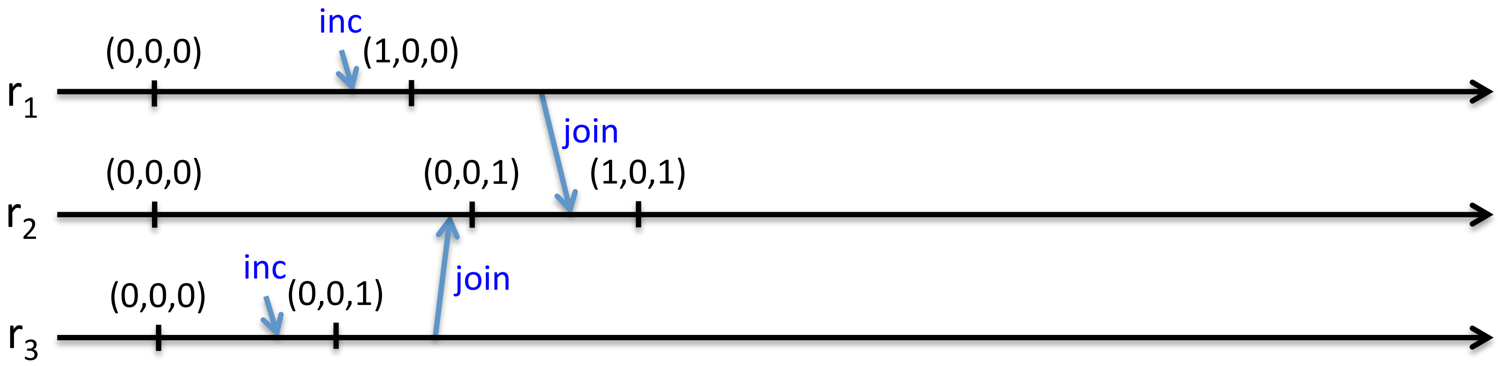
Example execution



- Increment at replica 1, its vector becomes (1,0,0)

LIGHTKONE

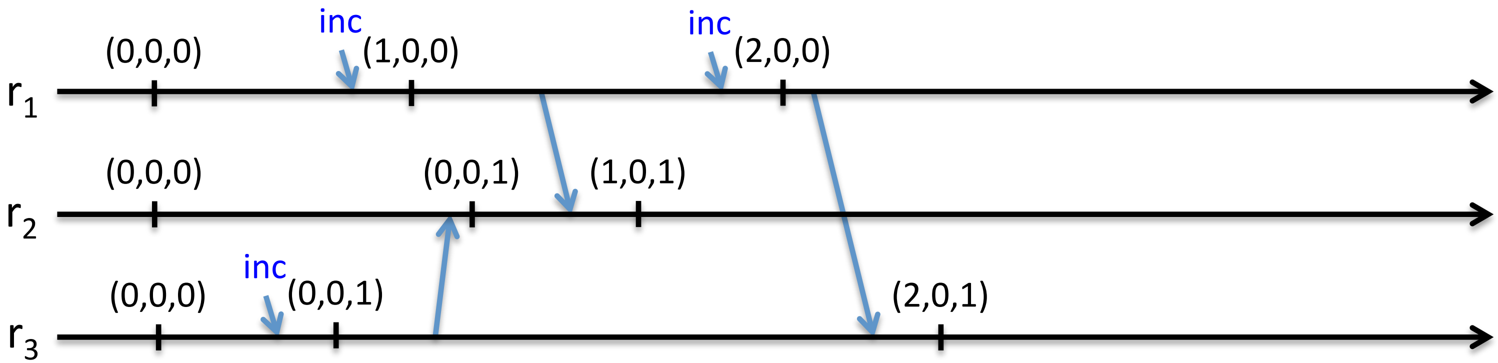
Example execution



- Join operations merge state from replica 1 and replica 3
- Replica 2's state is updated to (0,0,1) and then to (1,0,1)

LIGHTKONE

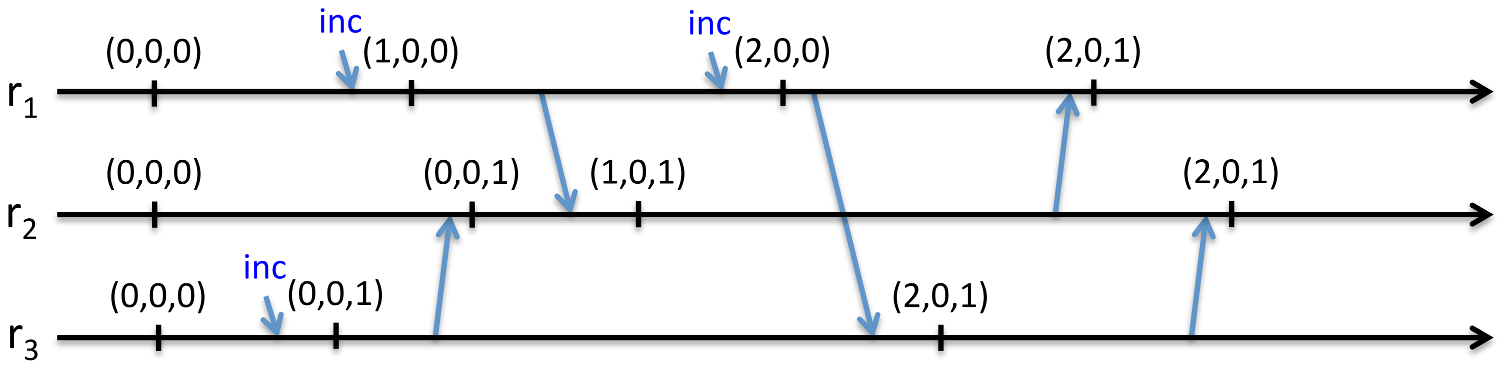
Example execution



- Another increment at replica 1 and a join to replica 3
- Replica 3's state becomes (2,0,1)
- Replica 1 is (2,0,0) and replica 2 is (1,0,1)

LIGHTKONE

Example execution



- Join operation from replica 2 to replica 1
- Join operation from replica 3 to replica 2
- All replicas have converged to the state (2,0,1)

LIGHTKONE

Carrying on

- The Grow-Only counter is one of the simplest CRDTs
  - Each replica stores information about all replicas, very much like a vector clock
- How expressive can a CRDT be?
  - Can we express counters that both increment and decrement?
  - Can we express sets where we can both add and remove elements?
- The answer is, yes, a CRDT can express all that and more
  - We will look at some smarter CRDTs in the next video

LIGHTKONE

More powerful CRDTs

- Let us now look at some more powerful CRDTs
- We show the Up-Down Counter and the Observed-Remove set
- Many more powerful CRDTs exists; we refer you to the bibliography to find out more
- We compare CRDTs with RSMs as a way to implement distributed data structures

LIGHTKONE

Up-Down Counter (PN Counter)

- Each replica i stores $s=(u_1, \ldots, u_n, d_1, \ldots, d_n)$ where $u_i, d_i \in N$ (natural)
- Each replica accepts inc, dec, val, and $\sqcup$ (join) operations
  - inc_i: update s to s' where $s'=(u_1, \ldots, u_i+1, \ldots, u_n, d_1, \ldots, d_n)$
  - dec_i: update s to s' where $s'=(u_1, \ldots, u_n, d_1, \ldots, d_i+1, \ldots, d_n)$
  - val_i: return $\sum_{1 \leq j \leq n} s.j - \sum_{n+1 \leq j \leq 2n} s.j$
  - join: $s \sqcup s' = (\max(u_1,u_1'), \ldots, \max(u_n,u_n'), \max(d_1,d_1'), \ldots, \max(d_n,d_n'))$

- How does this work?
  - Both inc and dec will inflate the value on the lattice
  - The val function calculates the correct value by doing a subtraction
  - Eventually all replicas will converge to the correct value, as before

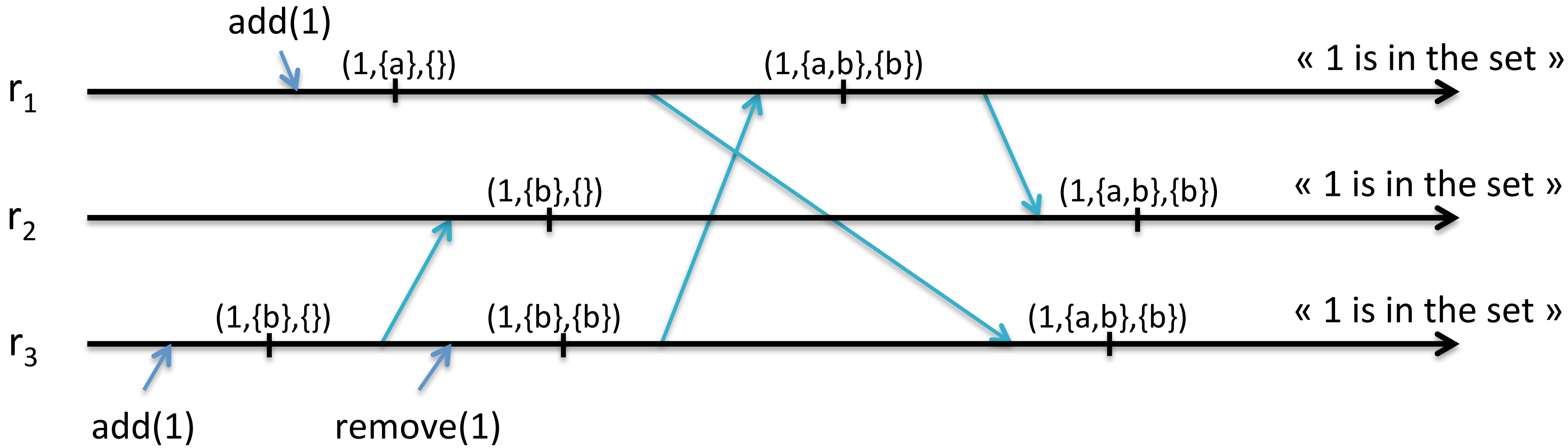
LIGHTKONE

Observed-Remove Set

- The OR-Set supports both adding and removing elements
  - The outcome of a sequence of adds and removes depends only on the causal history and conforms to the sequential specification of a set
  - In case of concurrent add and remove, the add has precedence
- The intuition is to tag each added element uniquely
  - The tag is not exposed when querying the set content
  - When removing an element, all tags are removed

LIGHTKONE

Observed-Remove Set



- Each replica stores triples (e,a,r) where e is the element, a is the set of adds and r is the set of removes
- If (e,a,r) with a-r≠{} then e is in the set
  - All updates (both adds and removes) cause monotonic increases in (e,a,r)

LIGHTKONE

Other CRDTs

- Many CRDTs have been invented
  - Registers: last-writer wins, multi-value
  - Sets: grow-only, 2P, add-wins, remove-wins
  - Maps, Pairs (including recursive versions)
  - Counter: unlimited, restricted ≥0 (bounded)
  - Graph: directed, monotonic DAG, edit graph
  - Sequence / List

LIGHTKONE

Comparison CRDT $\longleftrightarrow$ RSM

- In the course we have now seen two ways to define replicated distributed data structures
  - Replicated State Machine (RSM) approach
  - CRDT approach
- What is the difference?
  - RSM approach ensures consistency of replicas after each update, at the cost of needing consensus (e.g., Paxos or Raft)
  - CRDT approach ensures consistency when replicas have received the same set of updates, which needs only node-to-node communication

LIGHTKONE

What’s the catch?

- Many companies and applications are using CRDTs, and their number is growing daily
  - But if CRDTs are so great, why isn’t everybody using them?
- Trade-offs for using CRDTs
  - CRDTs require meta-data to ensure monotonicity and causality, which grows with the number of replicas
  - State-based CRDTs have growing state (tombstones), which requires some form of (unsynchronized) garbage collection
  - Last-writer-wins with physical clocks undergoes clock skew

LIGHTKONE

Rest of the lesson

- My colleagues Chris and Annette will now explain two important directions of this work:
- Lasp: a programming language and platform based on strong eventual consistency
- Antidote: a causally consistent transactional database based on strong eventual consistency

LIGHTKONE

Programming **Weak Synchronization** Models

Christopher S. Meiklejohn
Université catholique de Louvain, Belgium
Instituto Superior Técnico, Portugal

Convergent Objects
Conflict-Free Replicated Data Types

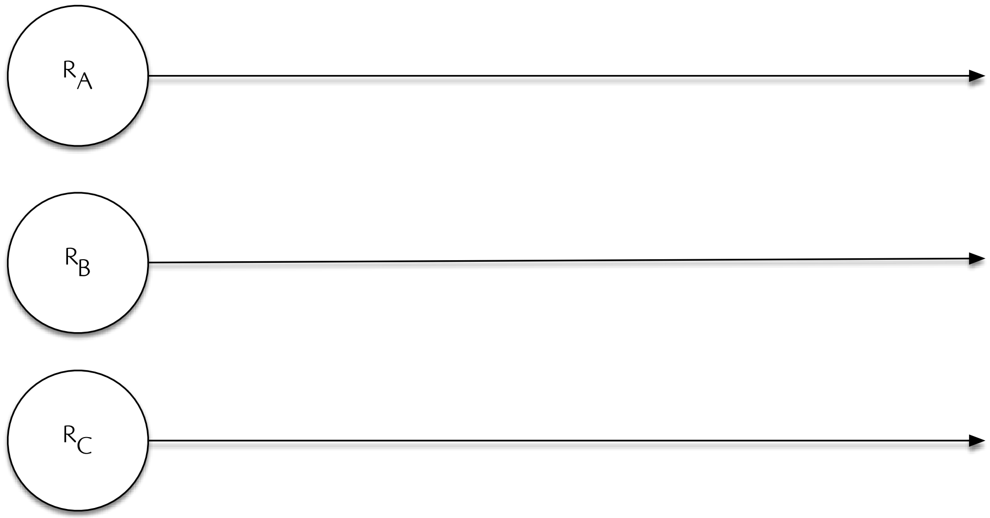
Conflict-Free Replicated Data Types

- **Many types exist with different properties**
  Sets, counters, registers, flags, maps

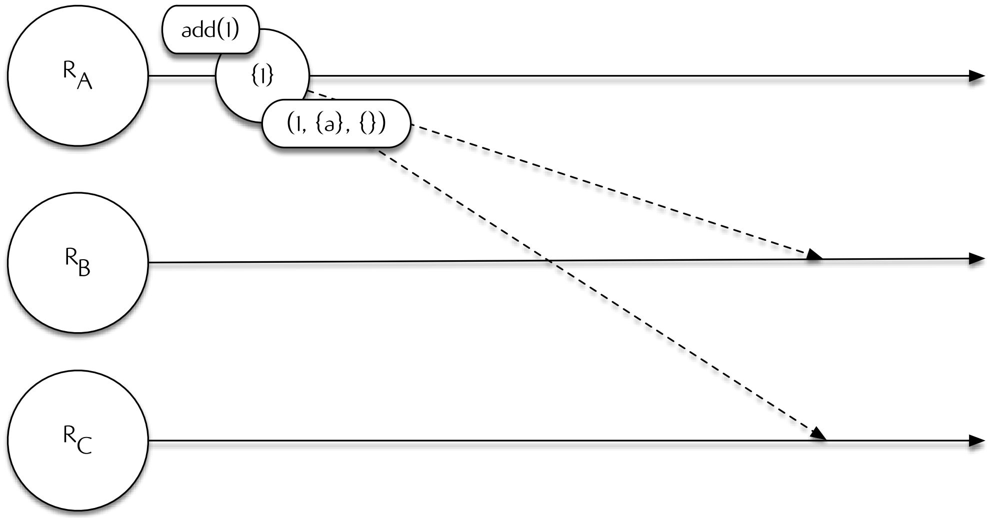
- **Strong Eventual Consistency**
  Instances satisfy SEC property per-object

- **Bounded join-semilattices**
  Formalized using bounded join-semilattices where the merge operation is the **join**

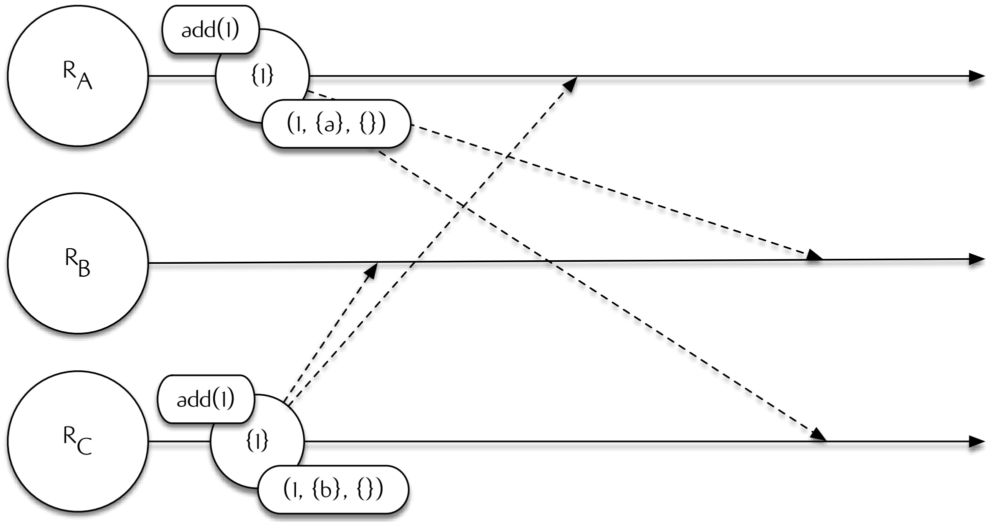
Convergent Objects
Observed-Remove Set



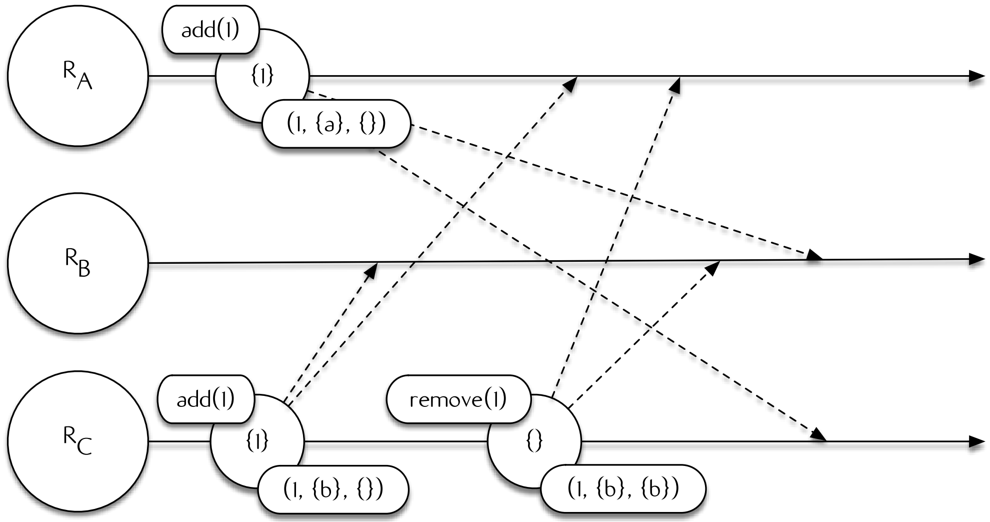
R_A
R_B
R_C



add(1)
{1}
(1, {a}, {})
R_A
R_B
R_C

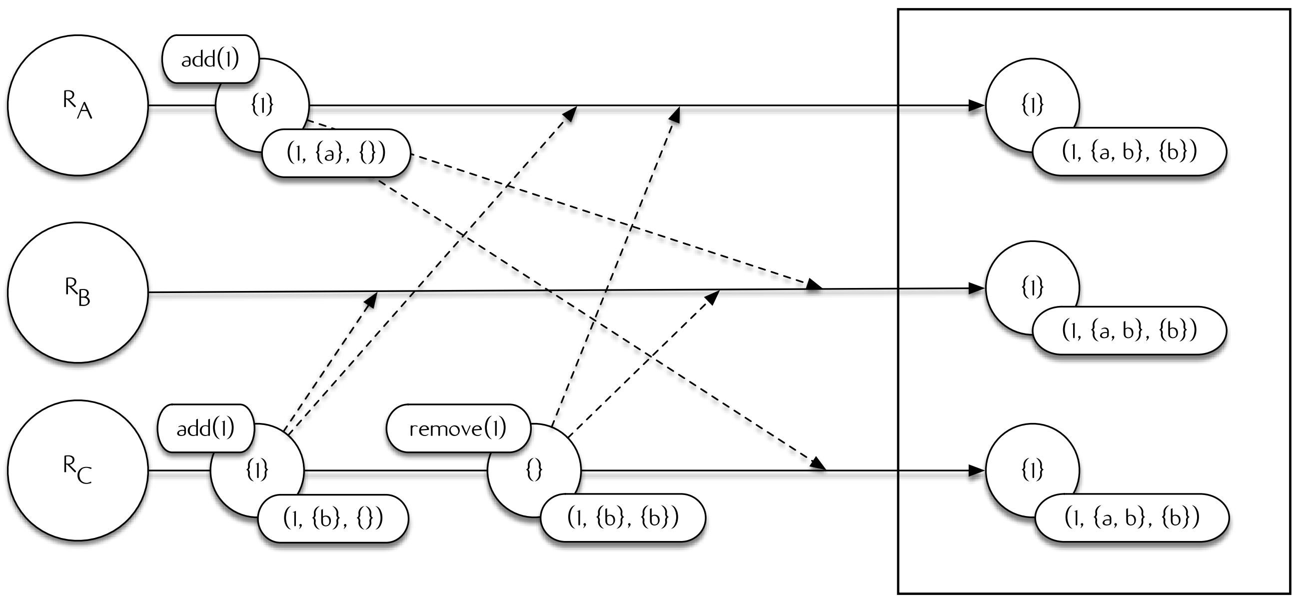


add(1)
R_A
{1}
(1, {a}, {})
R_B
add(1)
R_C
{1}
(1, {b}, {})



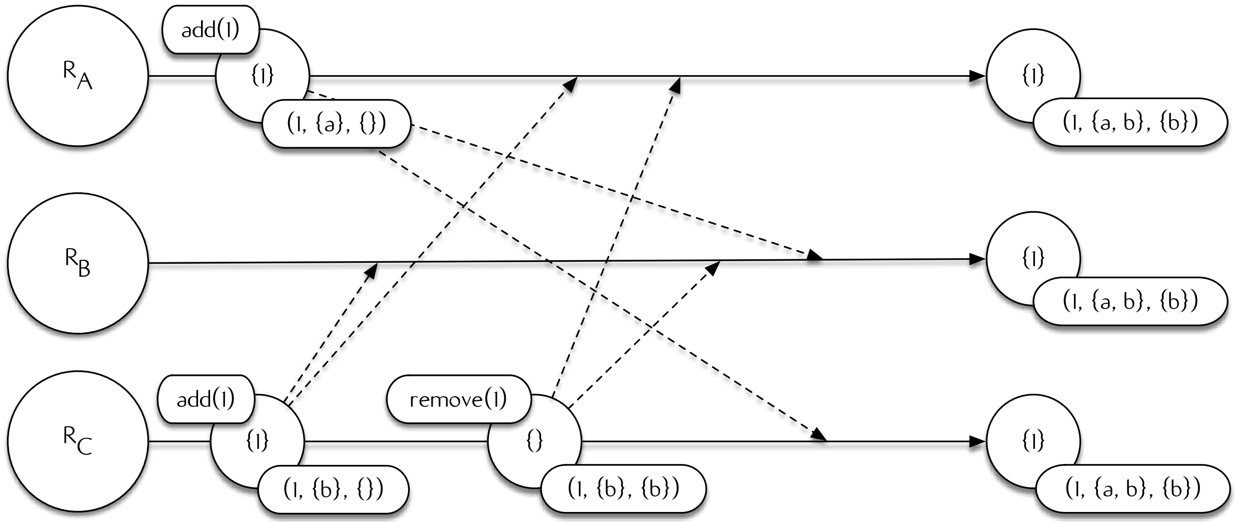
R_A
add(1)
{1}
(1, {a}, {})
R_B
R_C
add(1)
{1}
(1, {b}, {})
remove(1)
{}
(1, {b}, {b})

Convergence reached.



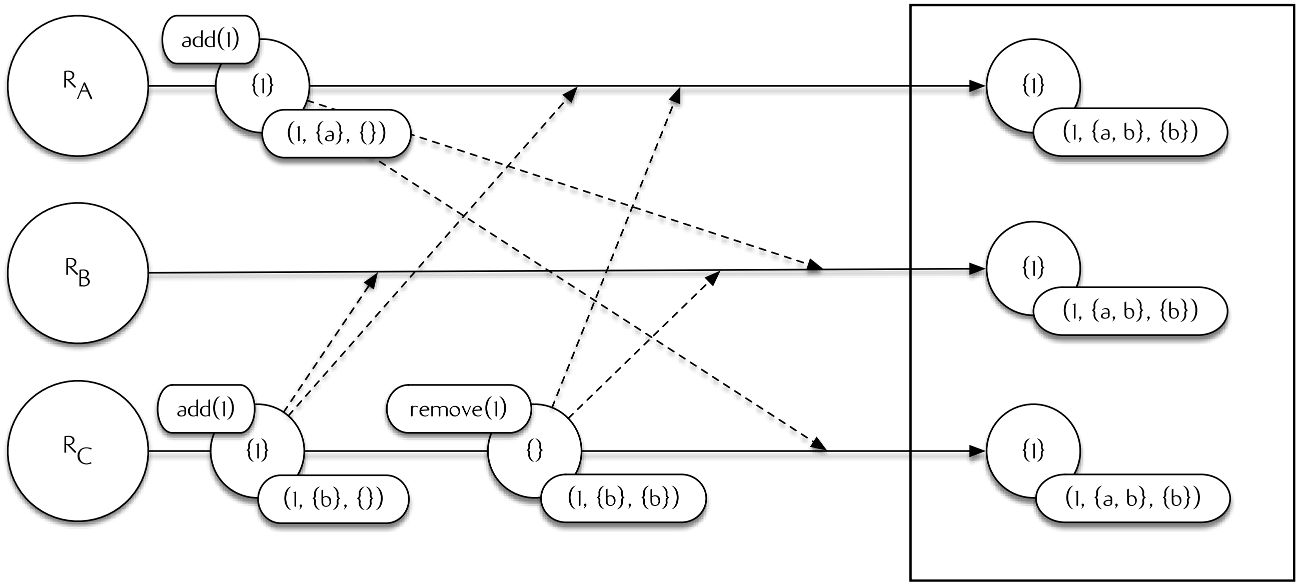
Convergent Objects

Nondeterminism



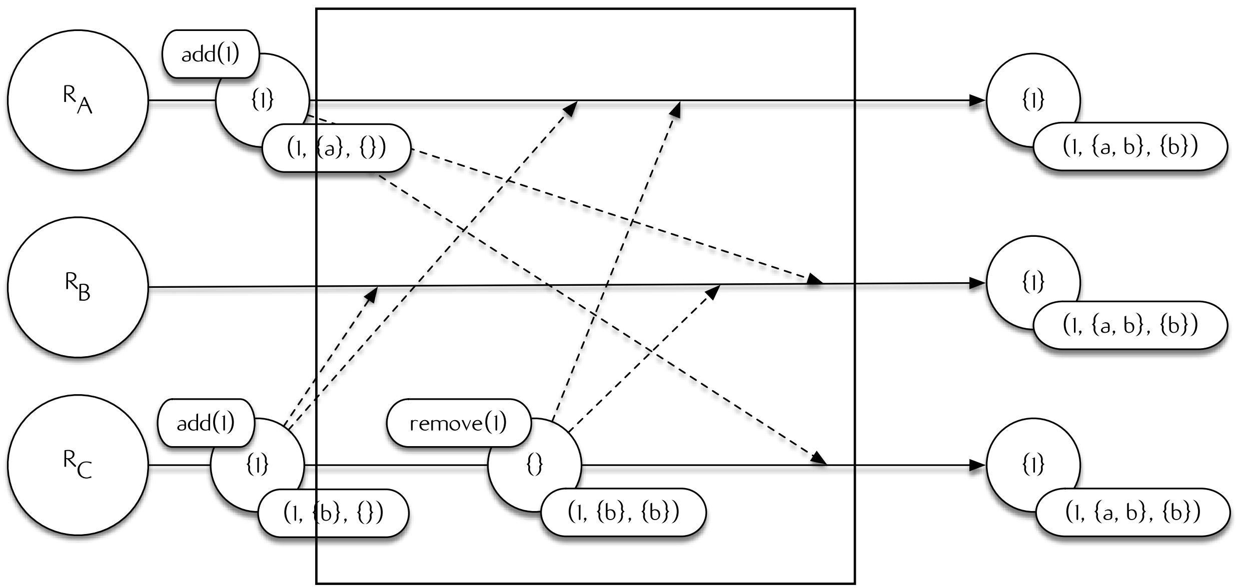
R_A
add(1)
{1}
(1, {a}, {})
{1}
(1, {a, b}, {b})
R_B
{1}
(1, {a, b}, {b})
R_C
add(1)
{1}
(1, {b}, {})
remove(1)
{}
(1, {b}, {b})
{1}
(1, {a, b}, {b})

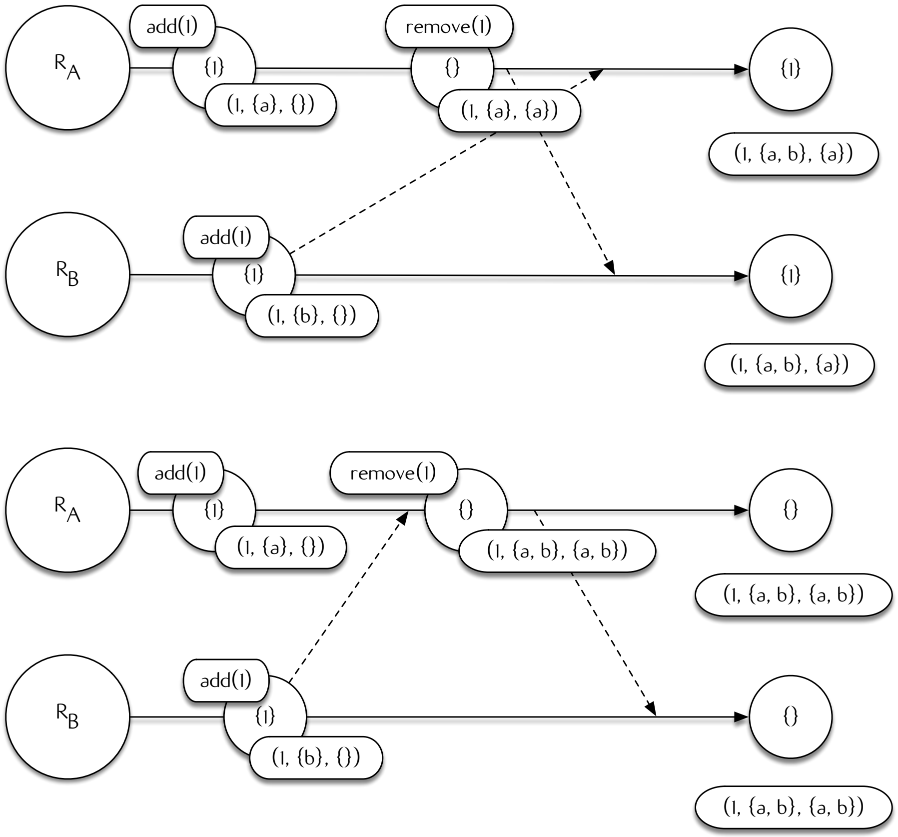
Convergence reached.



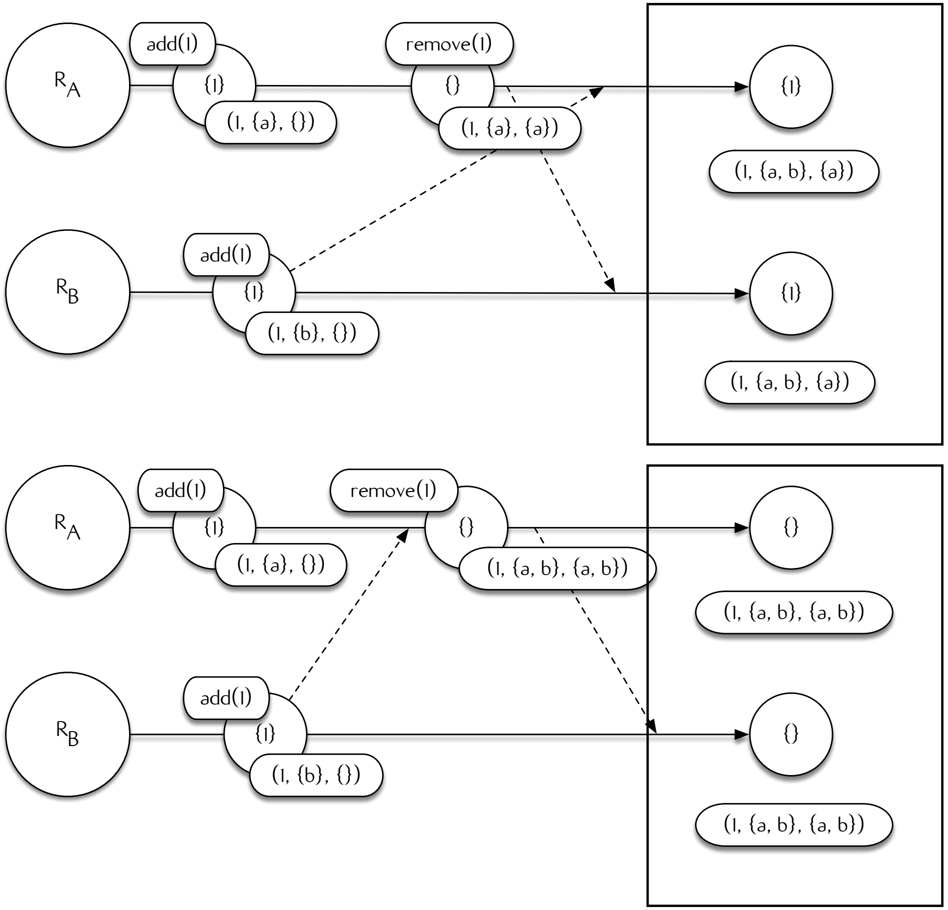
Desire:

The ability to reorder messages without impacting outcome.



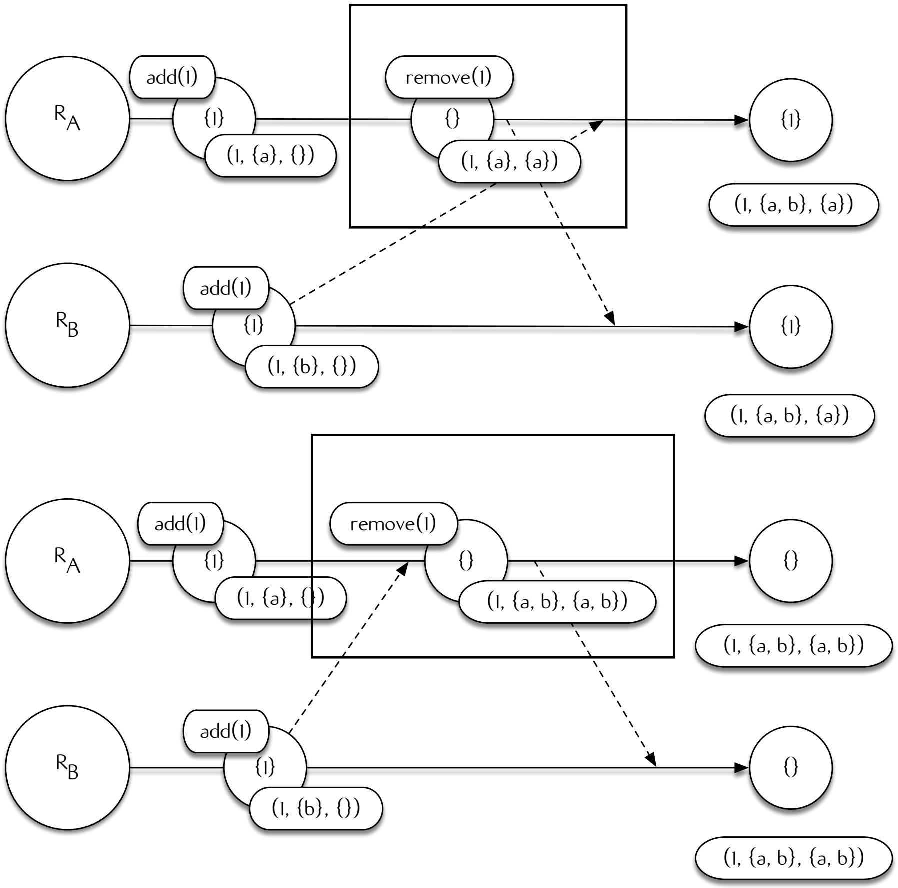


R_A
add(1)
{1}
(1, {a}, {})
remove(1)
{}
(1, {a}, {a})
{1}
(1, {a, b}, {a})
R_B
add(1)
{1}
(1, {b}, {})
{1}
(1, {a, b}, {a})
R_A
add(1)
{1}
(1, {a}, {})
remove(1)
{}
(1, {a, b}, {a, b})
{}
(1, {a, b}, {a, b})
R_B
add(1)
{1}
(1, {b}, {})
{}
(1, {a, b}, {a, b})



Convergence reached.

Different **synchronization schedules** can reach different outcomes.



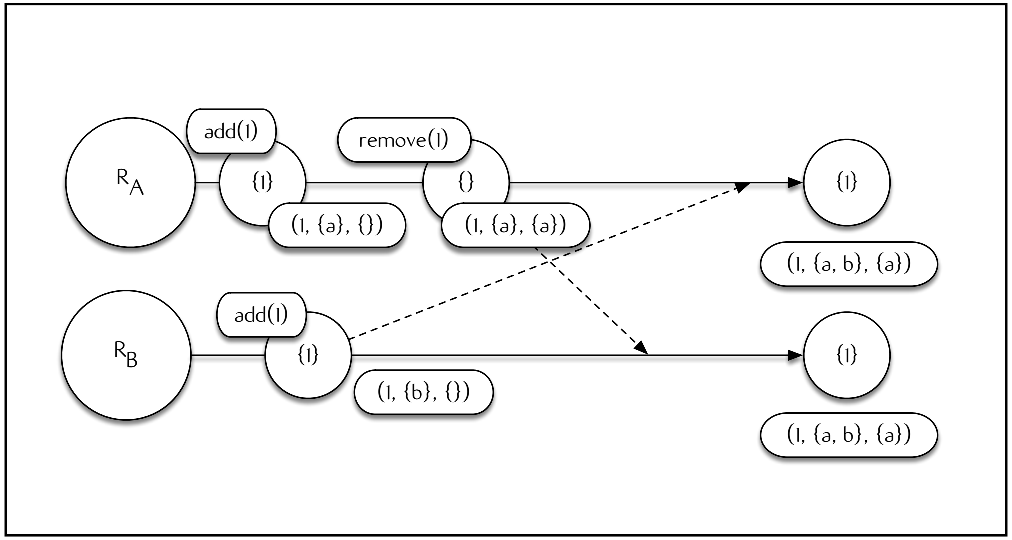
Reordering must be compatible with **causality.**

Each of these removes differ by their **causal** "influences."

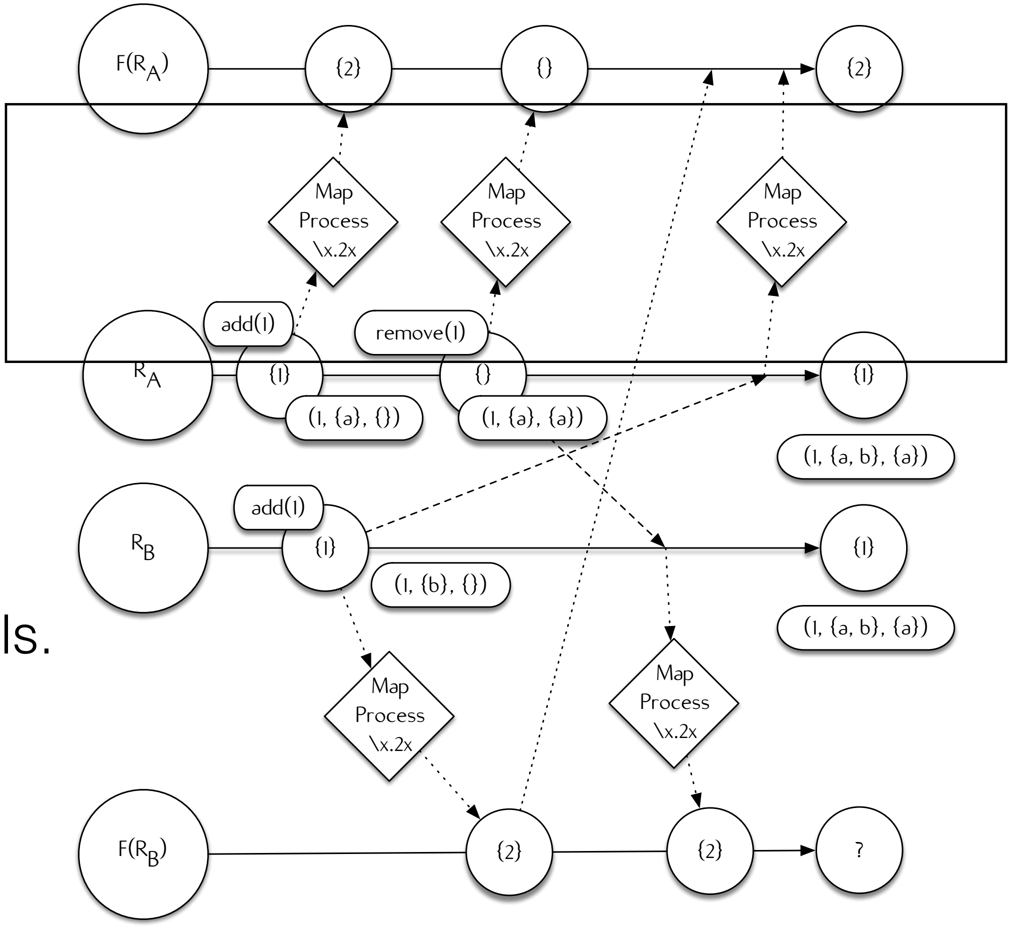
Convergent Objects

Composition

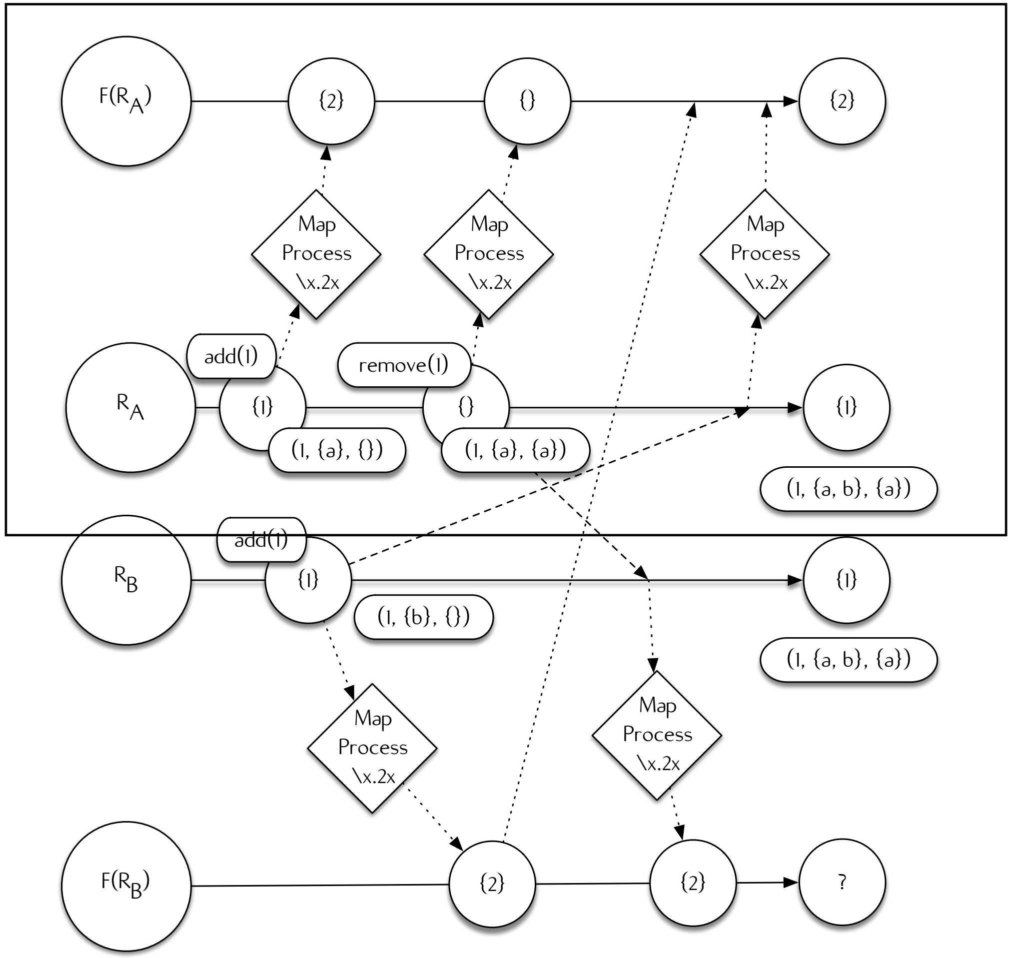
Replicated
set of naturals
across two nodes.



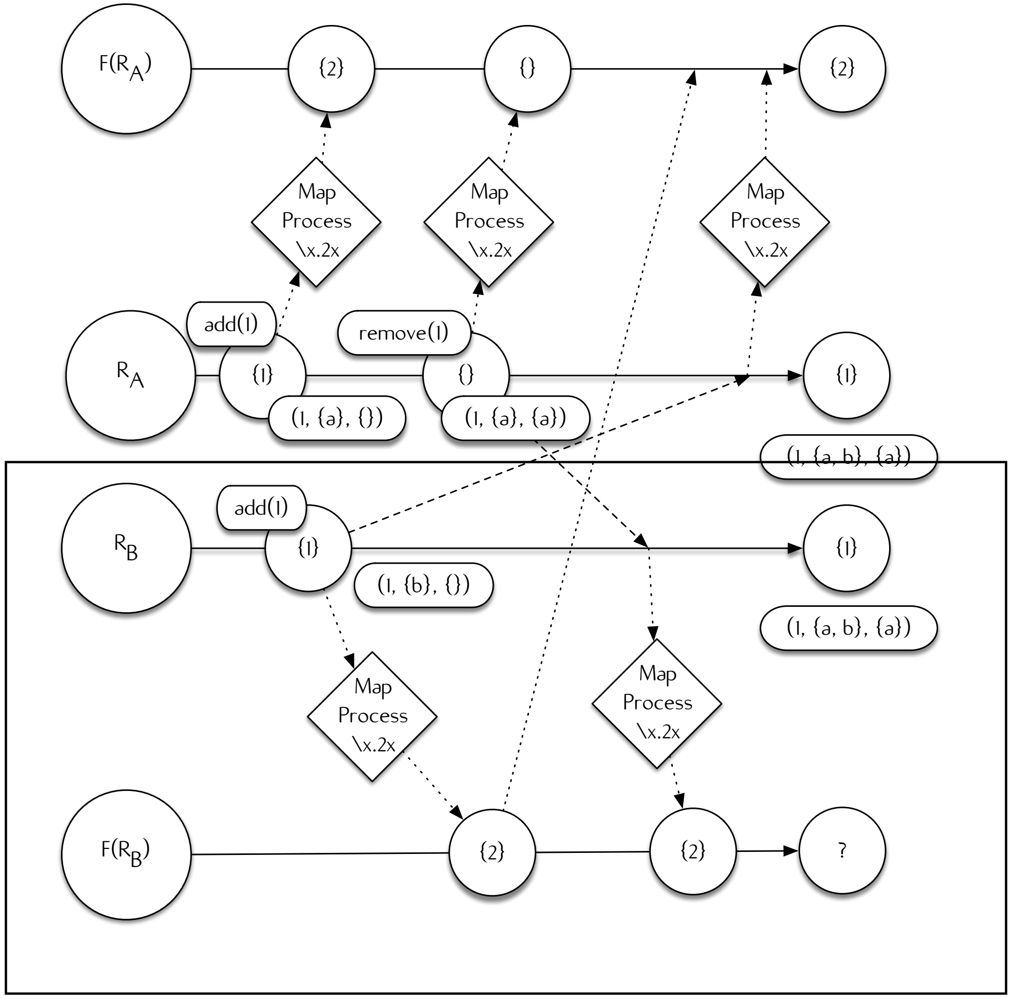
Map \x.2x
over a set of naturals.



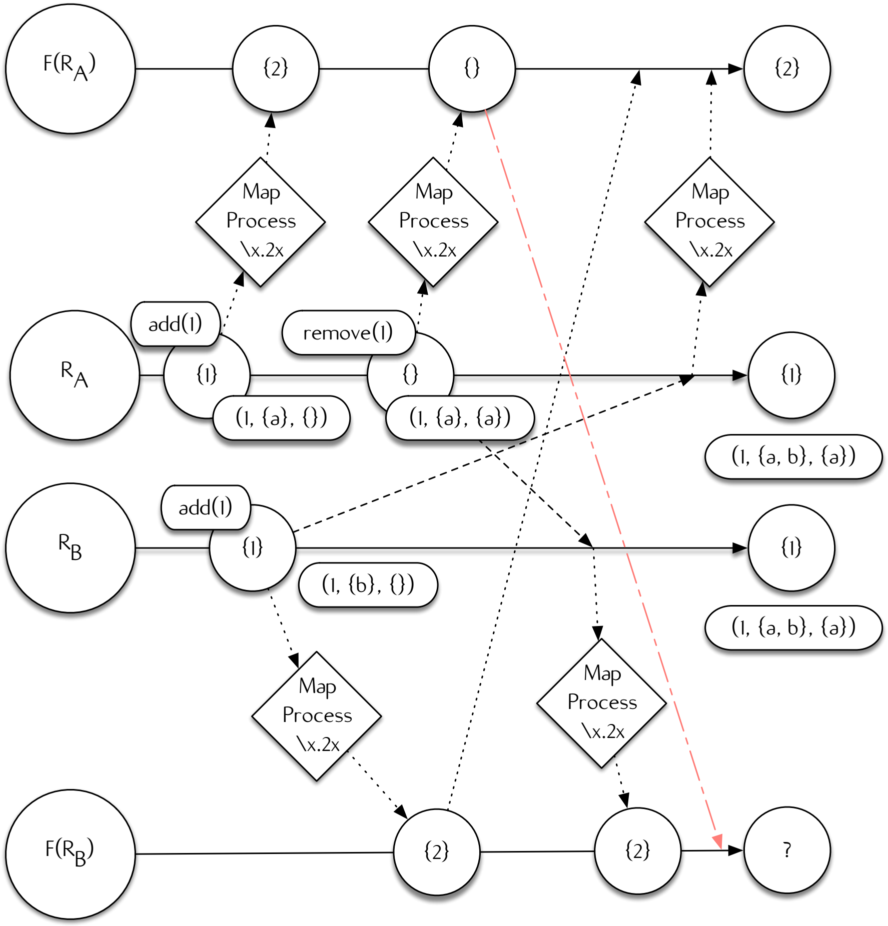
One **node**…



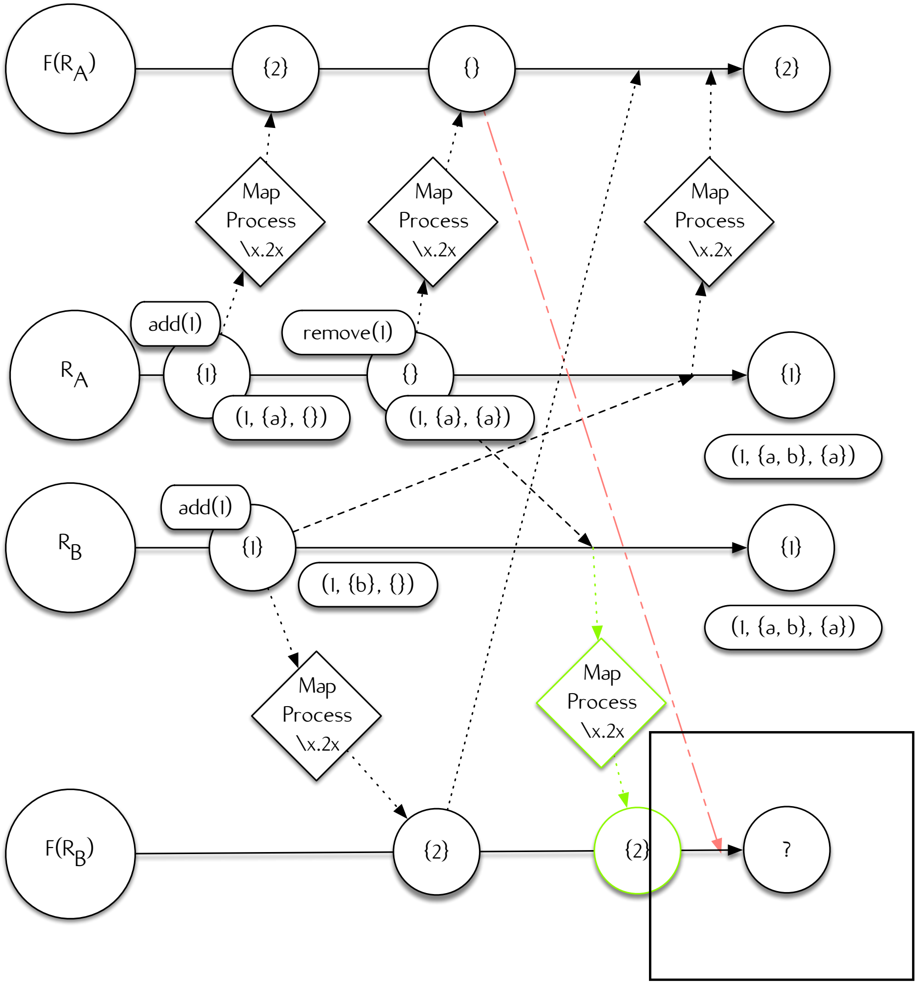
F(R_A)
{2}
{}
{2}
Map Process \x.2x
Map Process \x.2x
Map Process \x.2x
add(1)
remove(1)
R_A
{1}
{}
{1}
(1, {a}, {})
(1, {a}, {a})
(1, {a, b}, {a})
add(1)
R_B
{1}
{1}
(1, {b}, {})
(1, {a, b}, {a})
Map Process \x.2x
Map Process \x.2x
F(R_B)
{2}
{2}
?



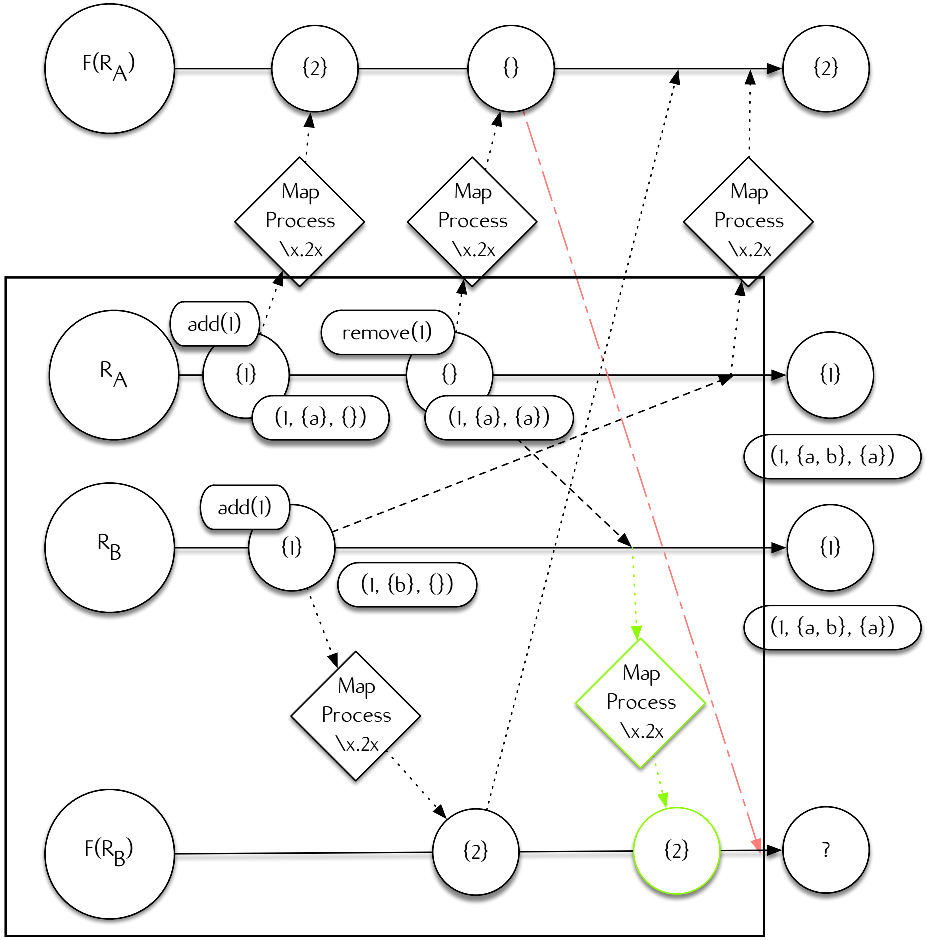
…another **node**.



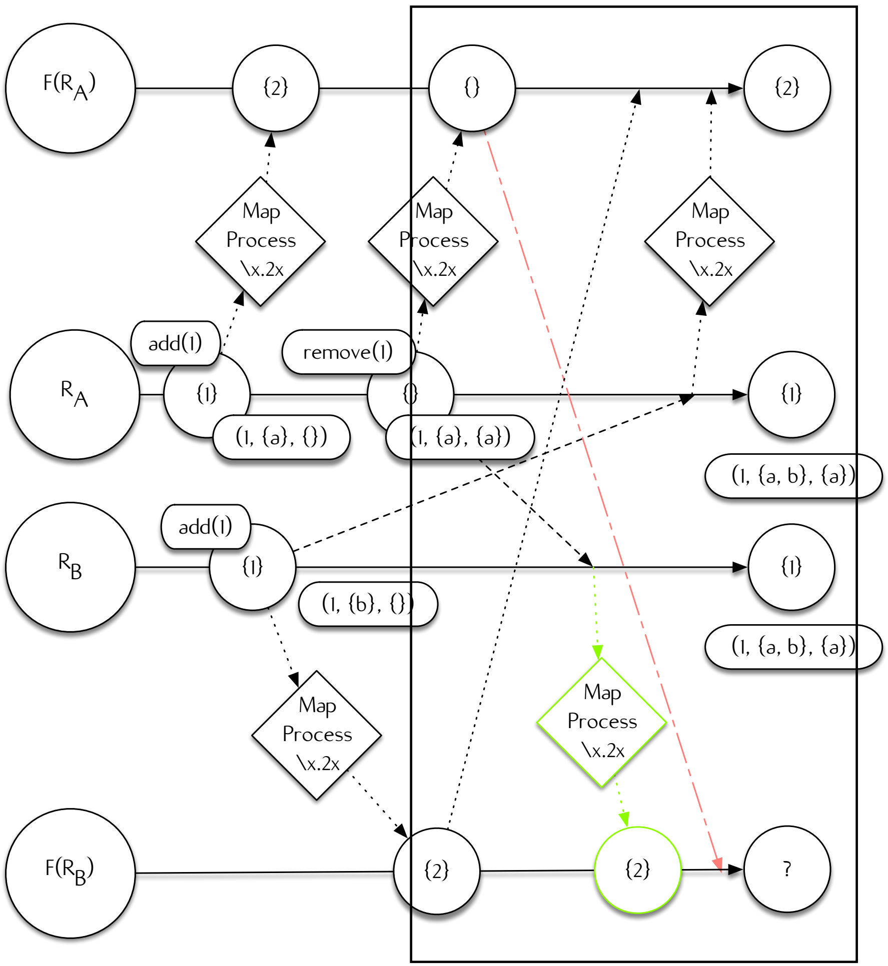
F(R_A)
{2}
{}
{2}
Map
Process
\x.2x
Map
Process
\x.2x
Map
Process
\x.2x
add(1)
remove(1)
R_A
{1}
{}
{1}
(1, {a}, {})
(1, {a}, {a})
(1, {a, b}, {a})
add(1)
R_B
{1}
{1}
(1, {b}, {})
(1, {a, b}, {a})
Map
Process
\x.2x
Map
Process
\x.2x
F(R_B)
{2}
{2}
?



Nondeterministic outcome.



Correct output that's seen all updates.



"Earlier" value that's been **delayed.**

Programming **Weak Synchronization** Models

Christopher S. Meiklejohn
Université catholique de Louvain, Belgium
Instituto Superior Técnico, Portugal

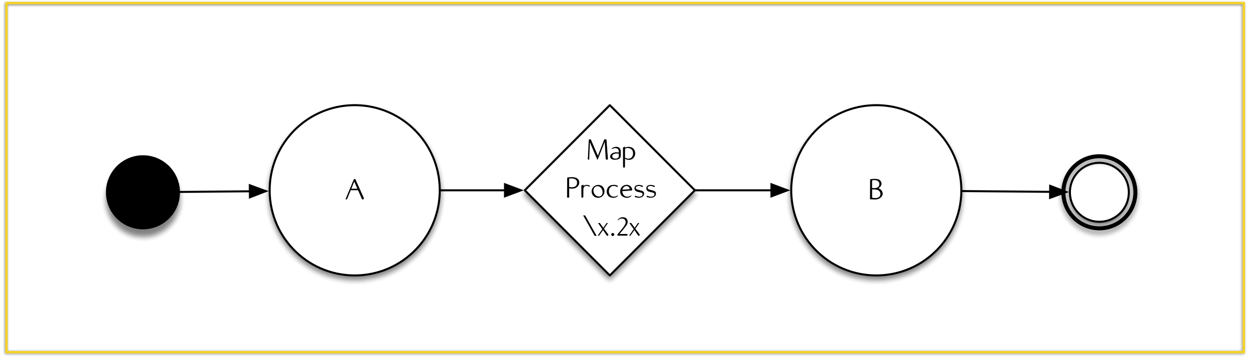
Convergent Programs

Lattice Processing

Lattice Processing

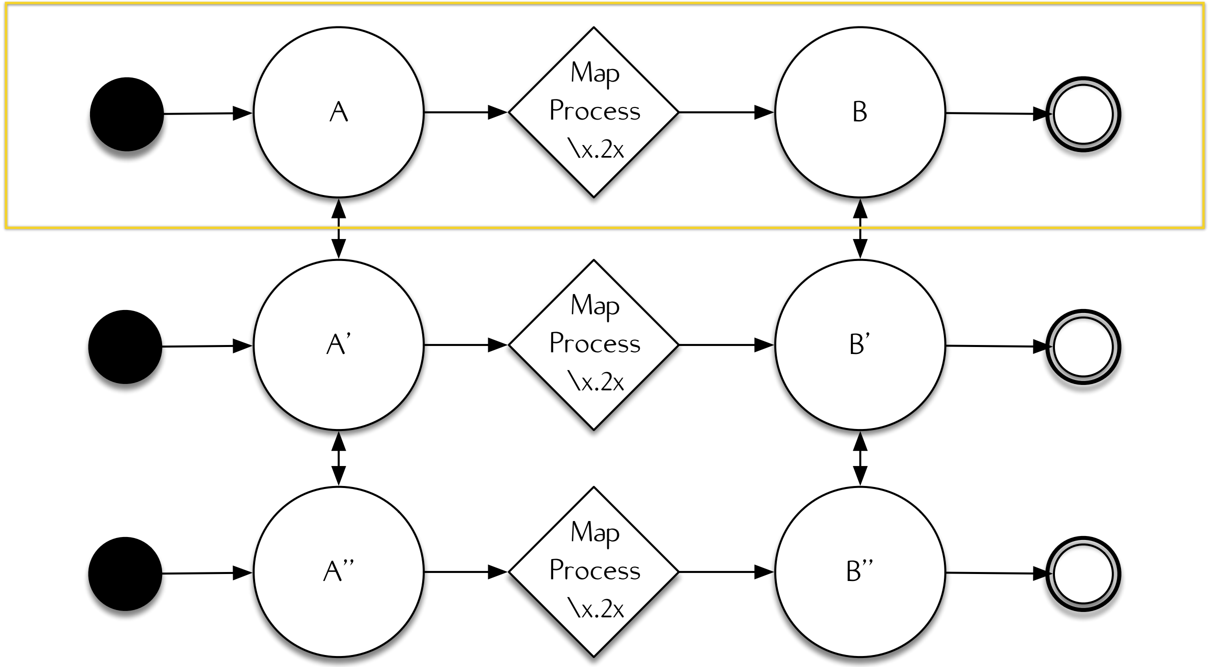
- **Asynchronous dataflow with streams**
  Combine and transform streams of inputs into streams of outputs
- **Convergent data structures**
  Data abstraction (inputs/outputs) is the CRDT
- **Confluence**
  Provides composition that **preserves the SEC property**

Lattice Processing
Confluence

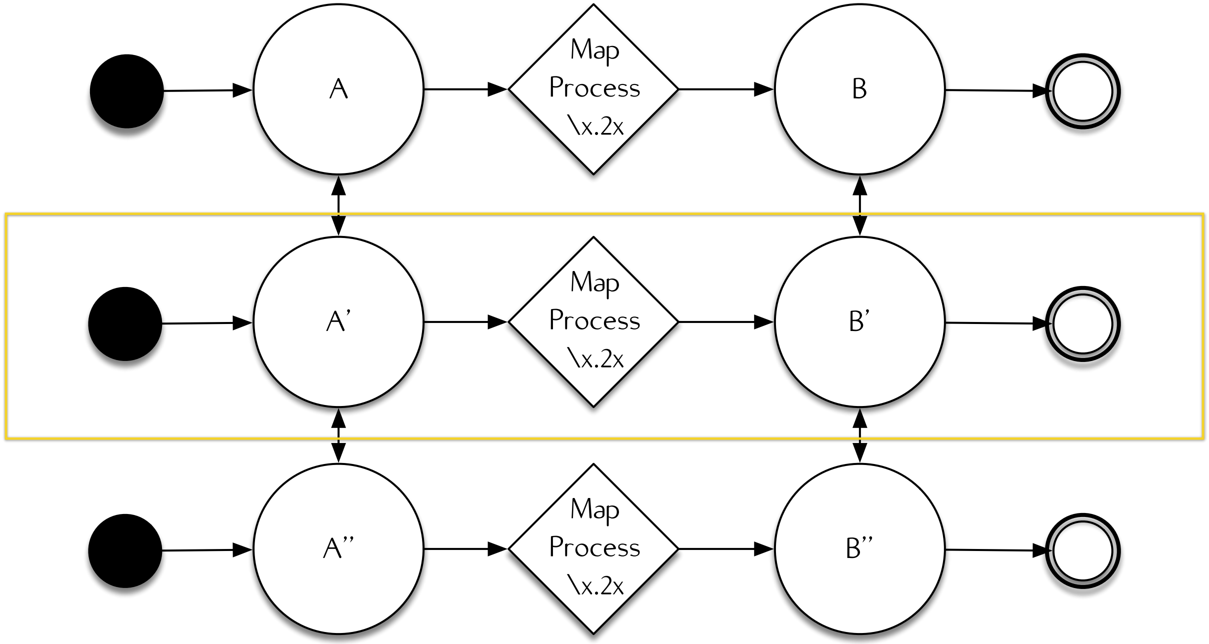


Sequential
specification.

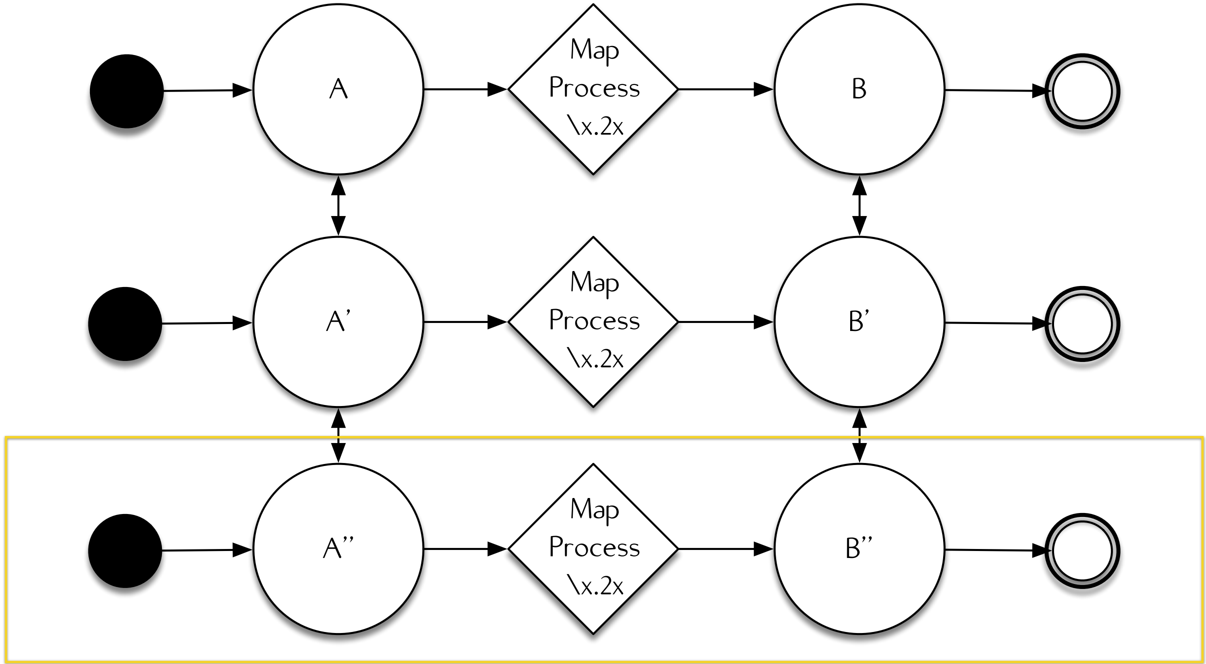
Replication
per node.



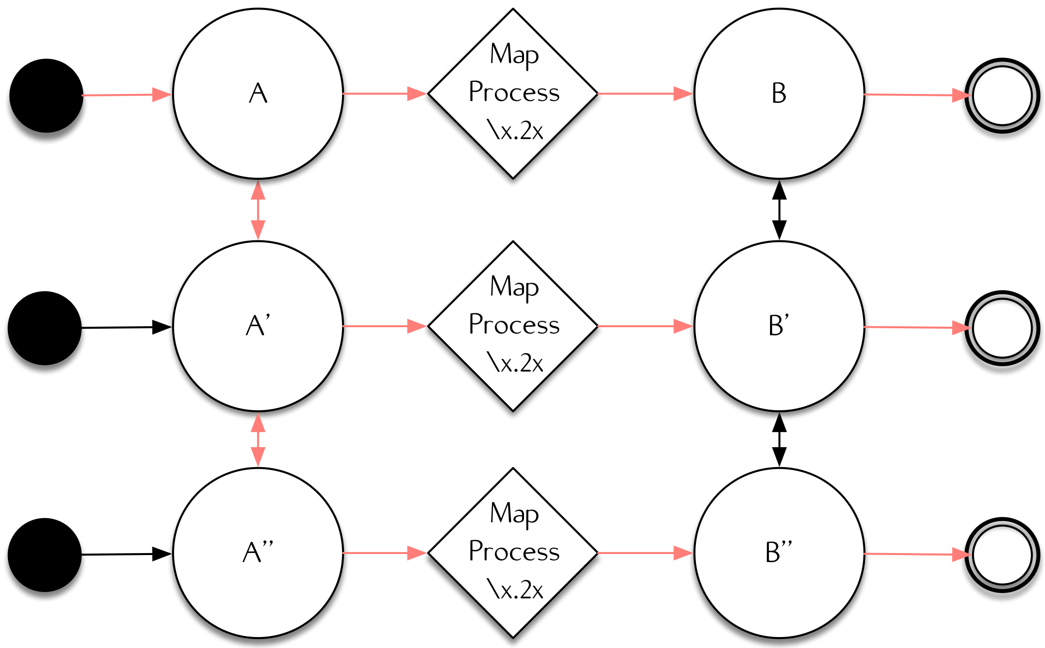
Replication
per node.

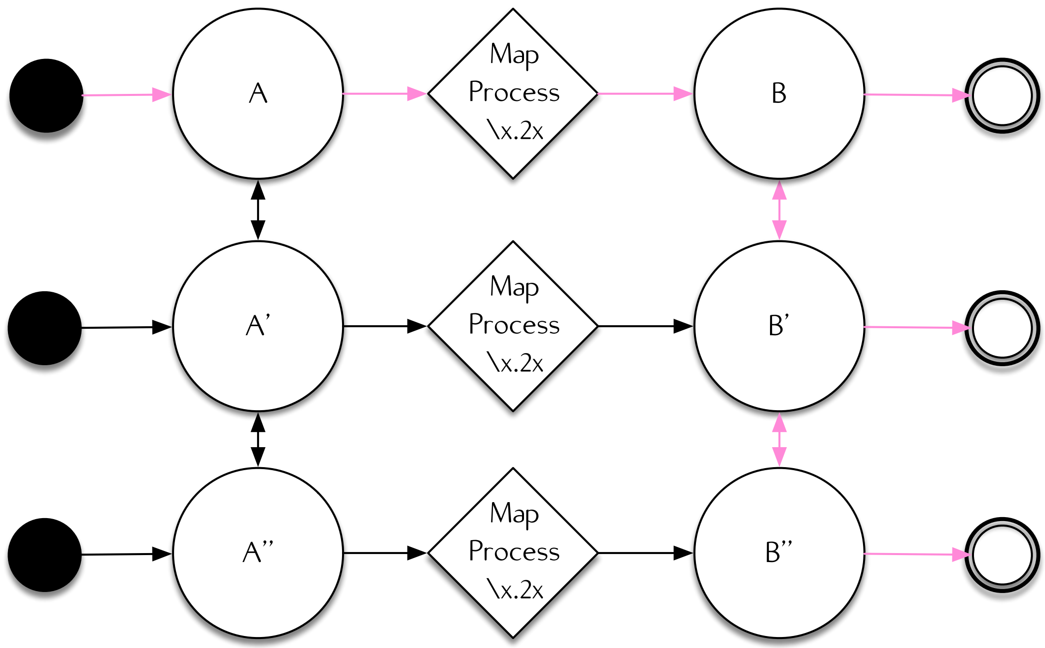


Replication
per node.

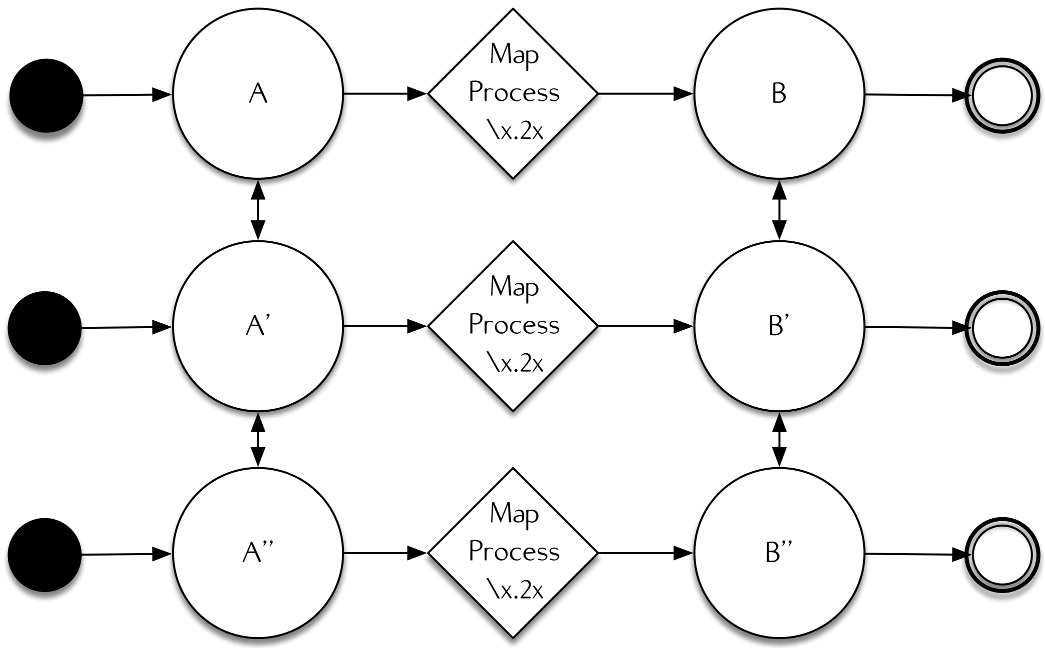


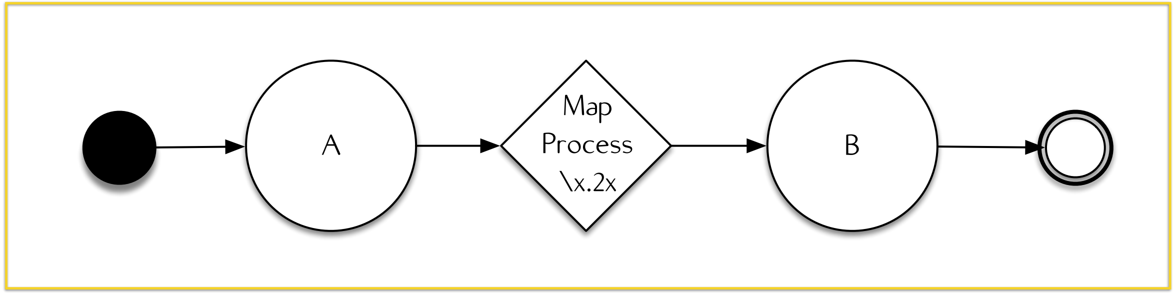
One **possible** schedule….



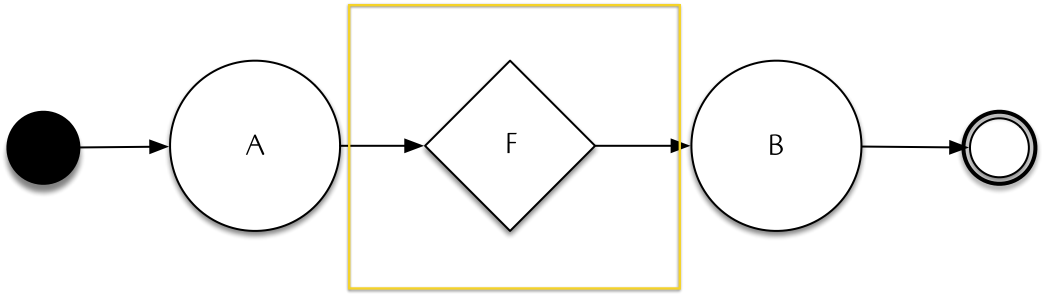


…another **possible** schedule.


A
Map
Process
\x.2x
B
A’
Map
Process
\x.2x
B’
A”
Map
Process
\x.2x
B”



All schedules **equivalent** to sequential schedule.



Arbitrary
application.

Lattice Processing
Example

```erlang
%% Create initial set.
S1 = declare(set),

%% Add elements to initial set and update.
update(S1, {add, [1,2,3]}),

%% Create second set.
S2 = declare(set),

%% Apply map operation between S1 and S2.
map(S1, fun(X) -> X * 2 end, S2).
```

```
%% Create initial set.
S1 = declare(set),

%% Add elements to initial set and update.
update(S1, {add, [1,2,3]}),

%% Create second set.
S2 = declare(set),

%% Apply map operation between S1 and S2.
map(S1, fun(X) -> X * 2 end, S2).
```

```erlang
%% Create initial set.
S1 = declare(set),

%% Add elements to initial set and update.
update(S1, {add, [1,2,3]}),

%% Create second set.
S2 = declare(set),

%% Apply map operation between S1 and S2.
map(S1, fun(X) -> X * 2 end, S2).
```

```
%% Create initial set.
S1 = declare(set),

%% Add elements to initial set and update.
update(S1, {add, [1,2,3]}),

%% Create second set.
S2 = declare(set),

%% Apply map operation between S1 and S2.
map(S1, fun(X) -> X * 2 end, S2).
```

```erlang
%% Create initial set.
S1 = declare(set),

%% Add elements to initial set and update.
update(S1, {add, [1,2,3]}),

%% Create second set.
S2 = declare(set),

%% Apply map operation between S1 and S2.
map(S1, fun(X) -> X * 2 end, S2).
```

Programming **Weak Synchronization** Models

Christopher S. Meiklejohn
Université catholique de Louvain, Belgium
Instituto Superior Técnico, Portugal

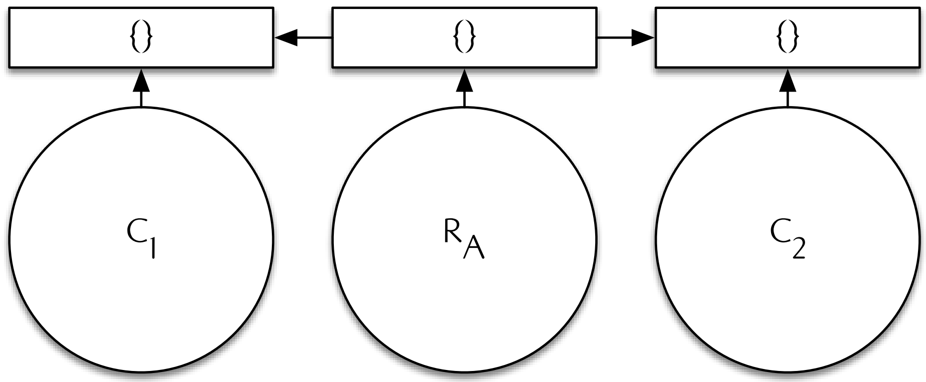
Processes

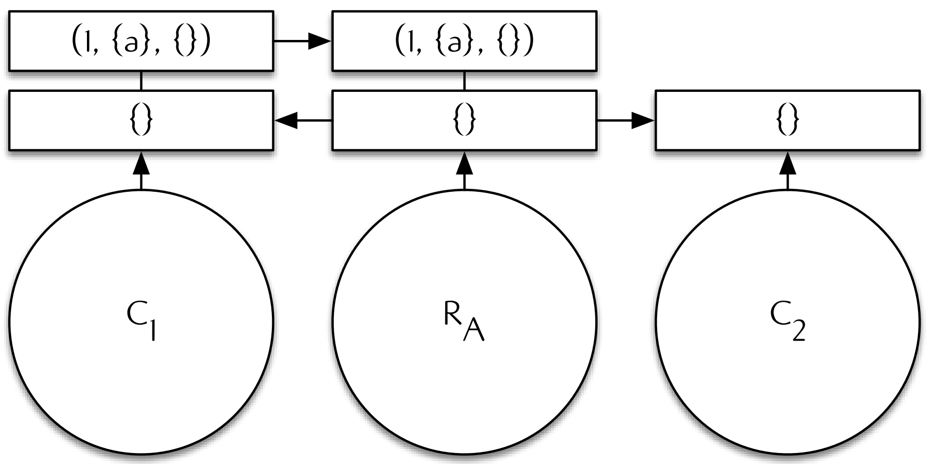
- **Replicas as monotonic streams**
  Each replica of a CRDT produces a **monotonic stream of states**

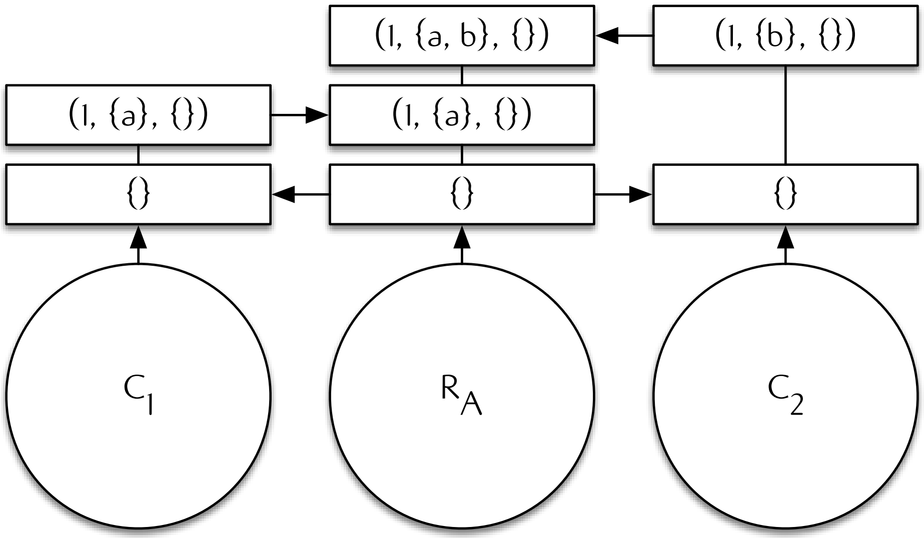
- **Monotonic processes**
  Read from one or more input replica streams and produce a single output replica stream

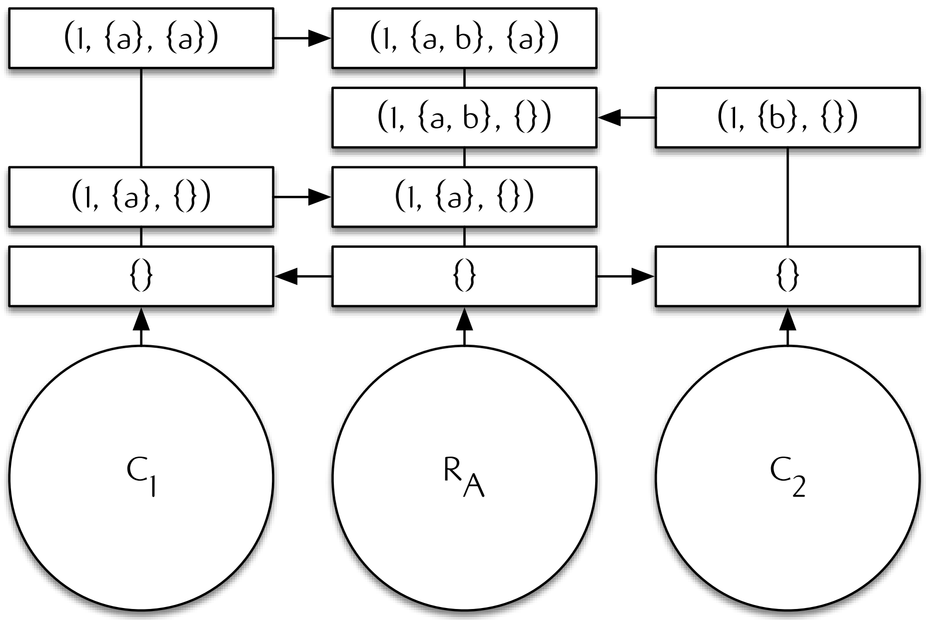
- **Inflationary reads**
  Read operation ensures that we only read **inflationary** updates to replicas

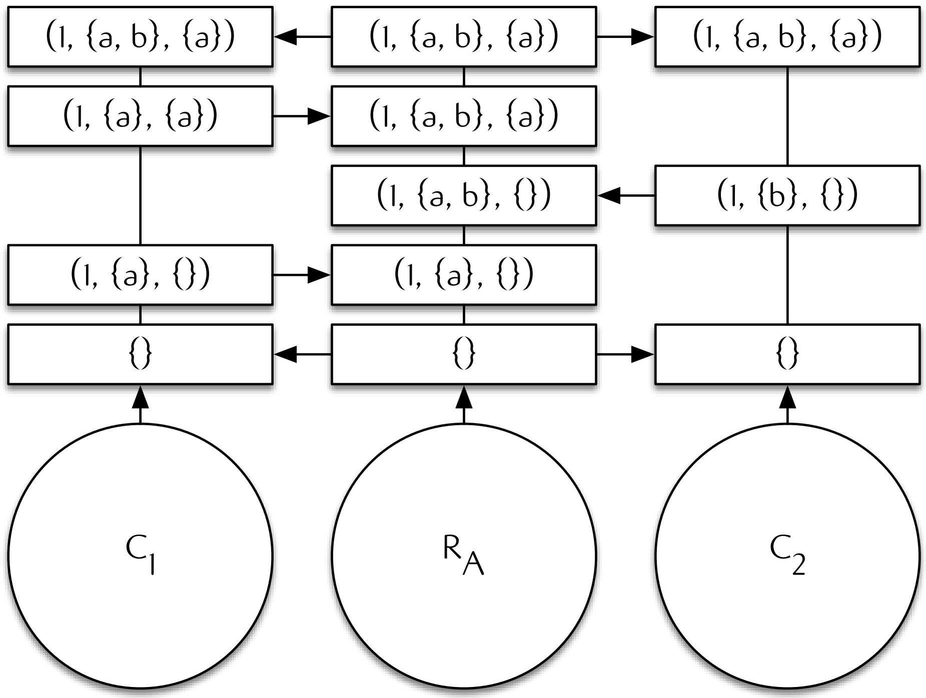
Lattice Processing
Monotonic Streams


{}
{}
{}
C1
RA
C2

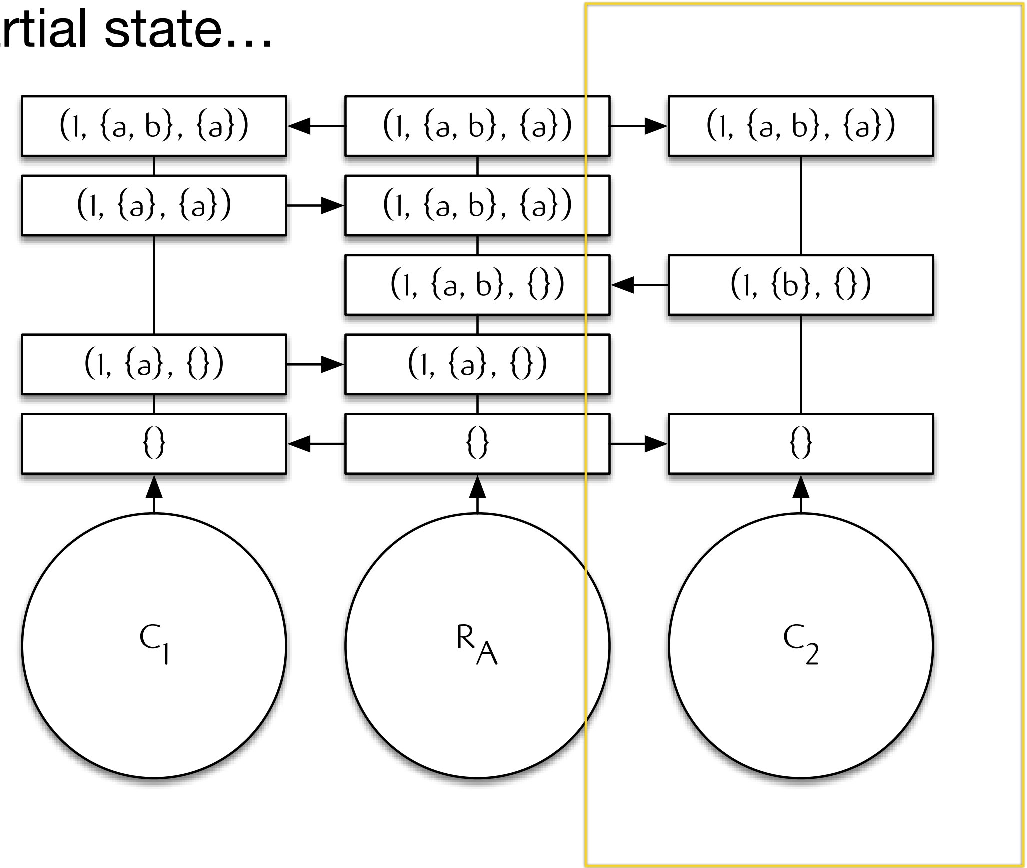

(1, {a}, {})
(1, {a}, {})
{}
{}
{}
C1
RA
C2


(1, {a, b}, {})
(1, {b}, {})
(1, {a}, {})
(1, {a}, {})
{}
{}
{}
C_1
R_A
C_2

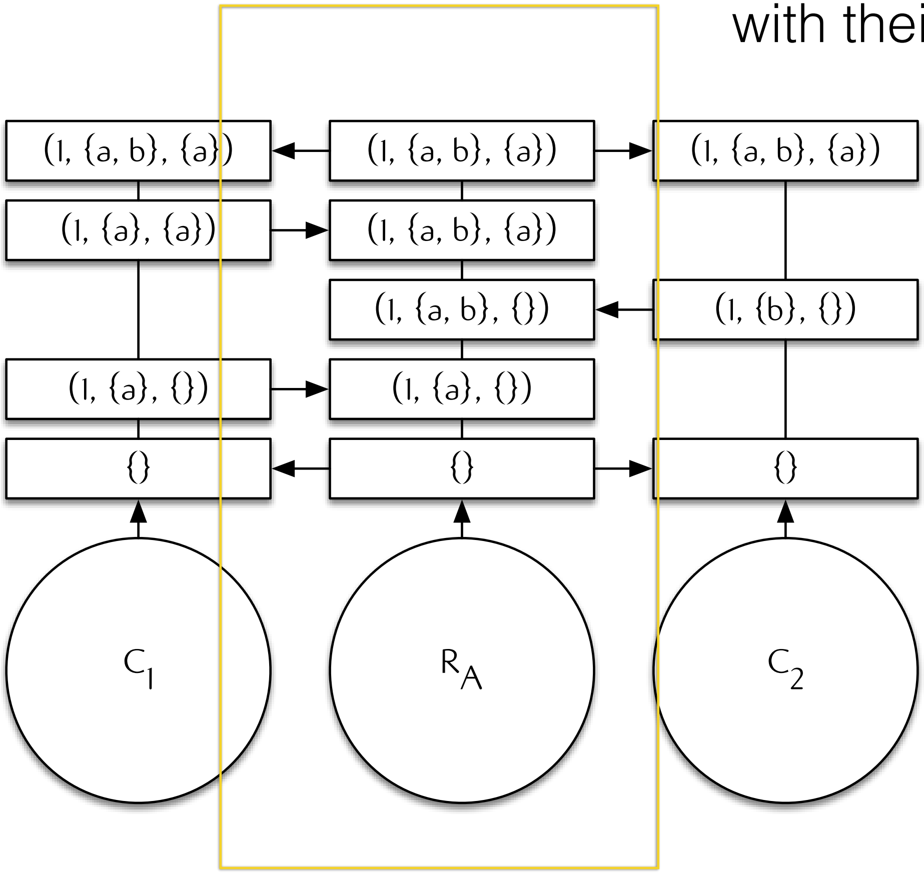

(1, {a}, {a})
(1, {a, b}, {a})
(1, {a, b}, {})
(1, {b}, {})
(1, {a}, {})
(1, {a}, {})
{}
{}
{}
C1
RA
C2


(1, {a, b}, {a})
(1, {a, b}, {a})
(1, {a, b}, {a})
(1, {a}, {a})
(1, {a, b}, {a})
(1, {a, b}, {})
(1, {b}, {})
(1, {a}, {})
(1, {a}, {})
{}
{}
{}
C_1
R_A
C_2

Clients can operate with **partial state…**

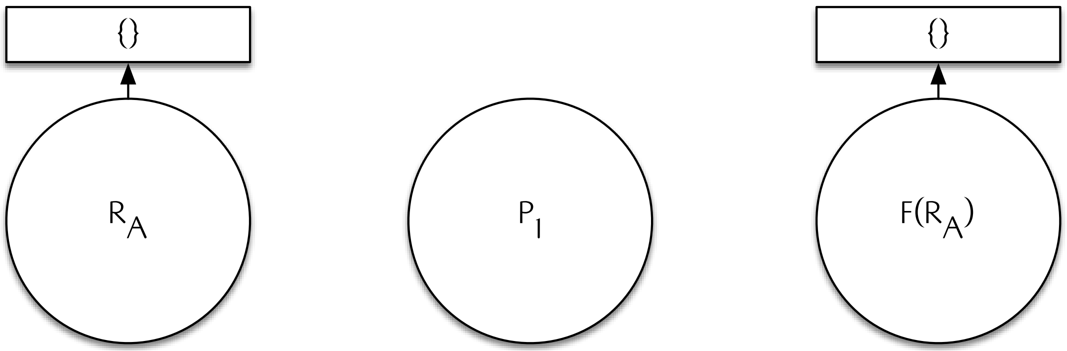


… and synchronize
with their **local replica**.

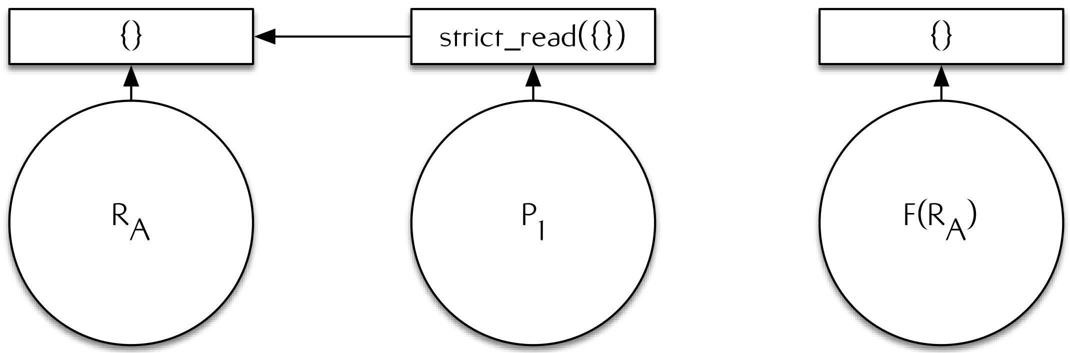


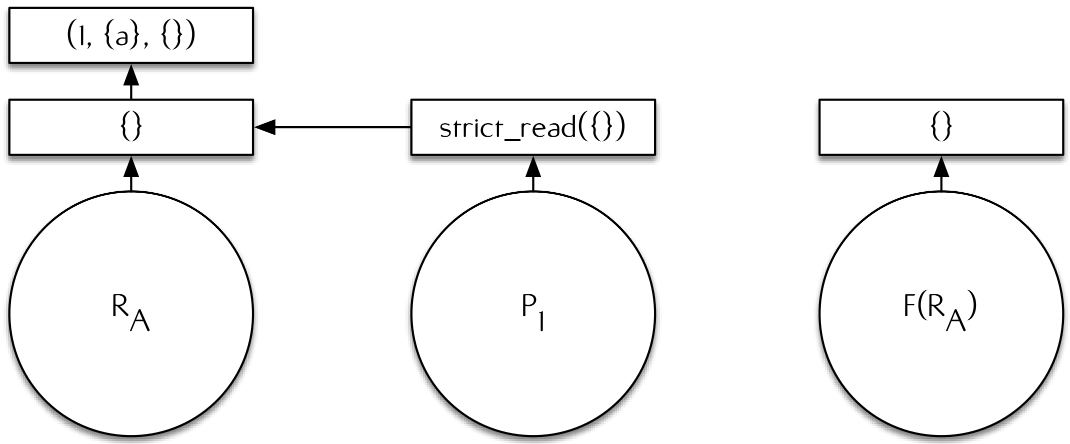
Lattice Processing

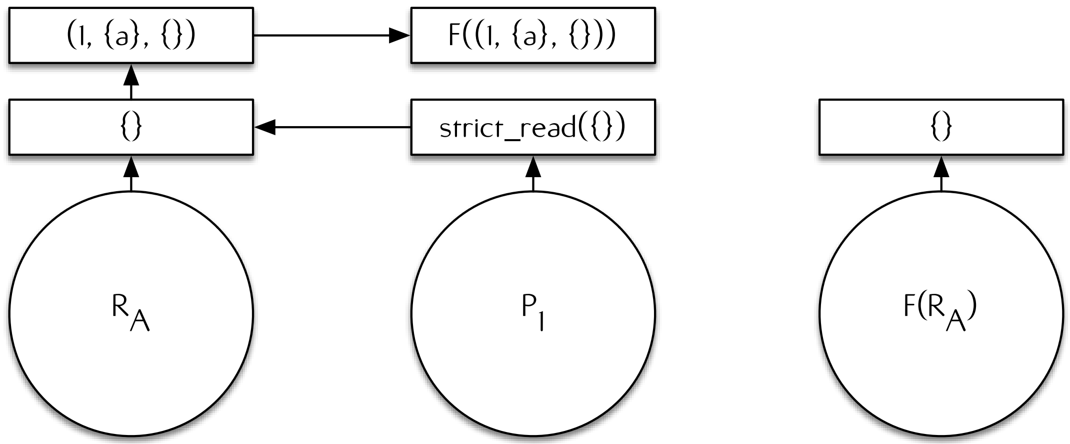
Monotonic Processes



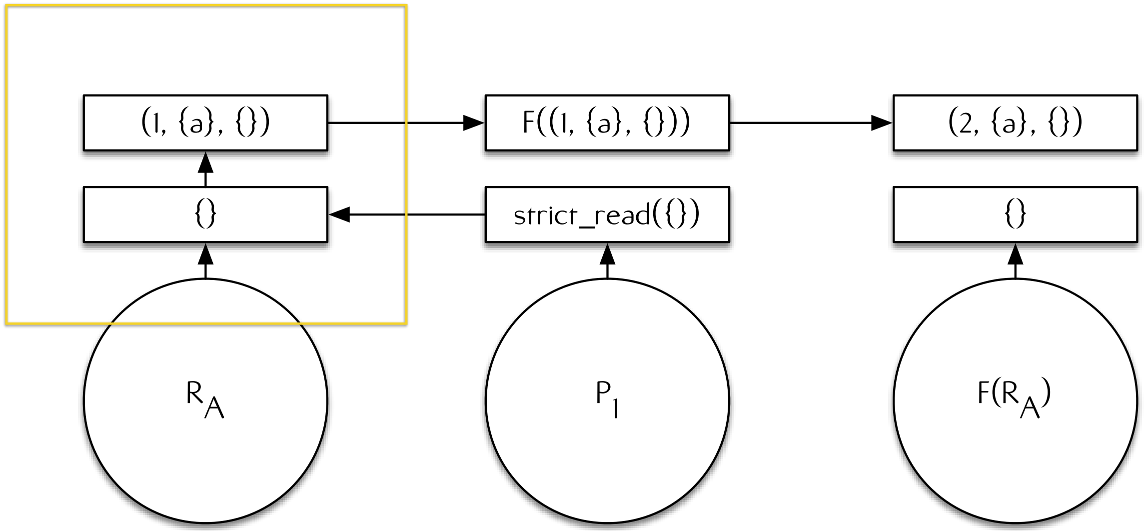
{}
{}
R_A
P_1
F(R_A)


{}
strict_read({})
{}
R_A
P_1
F(R_A)

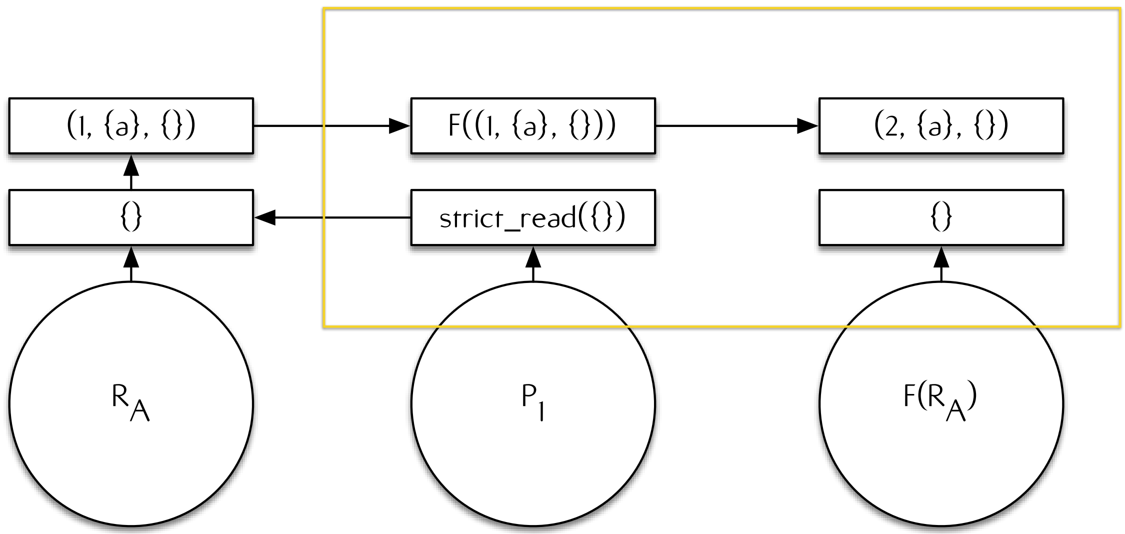

(1, {a}, {})
{}
strict_read({})
{}
R_A
P_1
$F(R_A)$

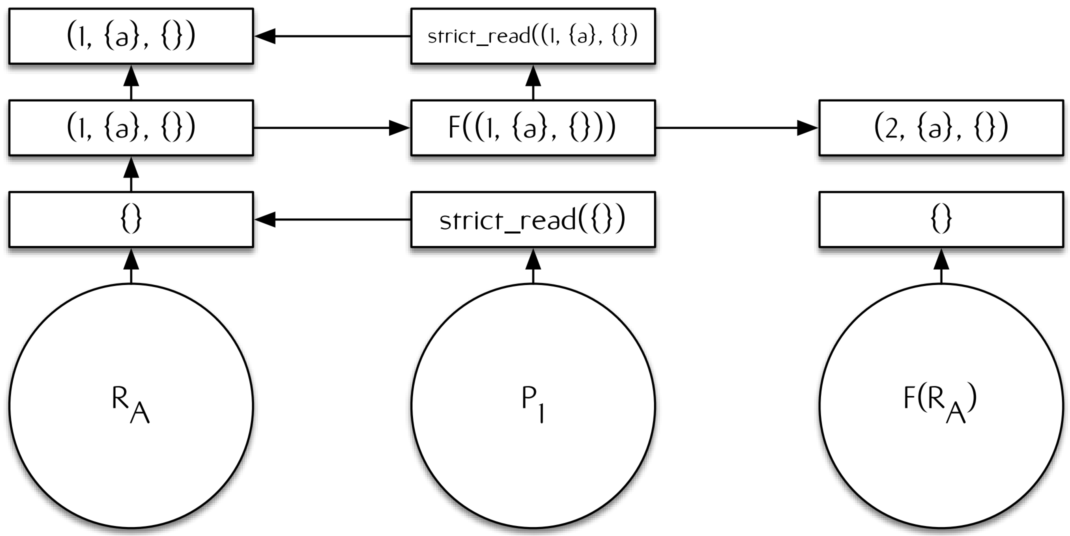

(1, {a}, {})
F((1, {a}, {}))
{}
strict_read({})
{}
R_A
P_1
$F(R_A)$

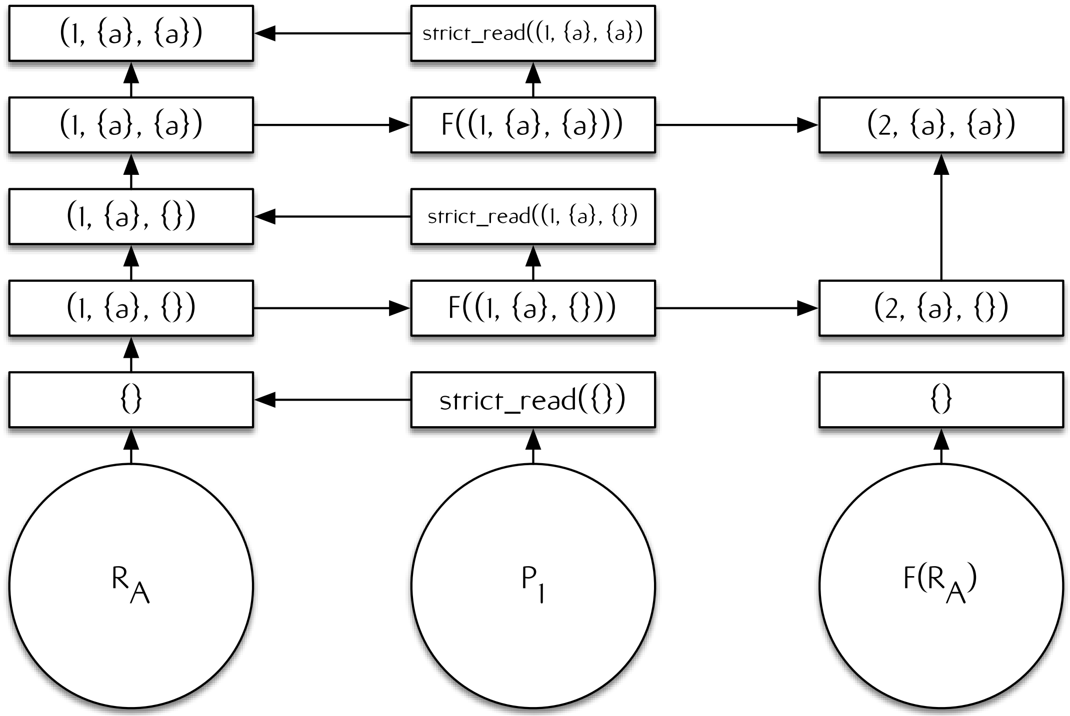
Every time **replica** changes…



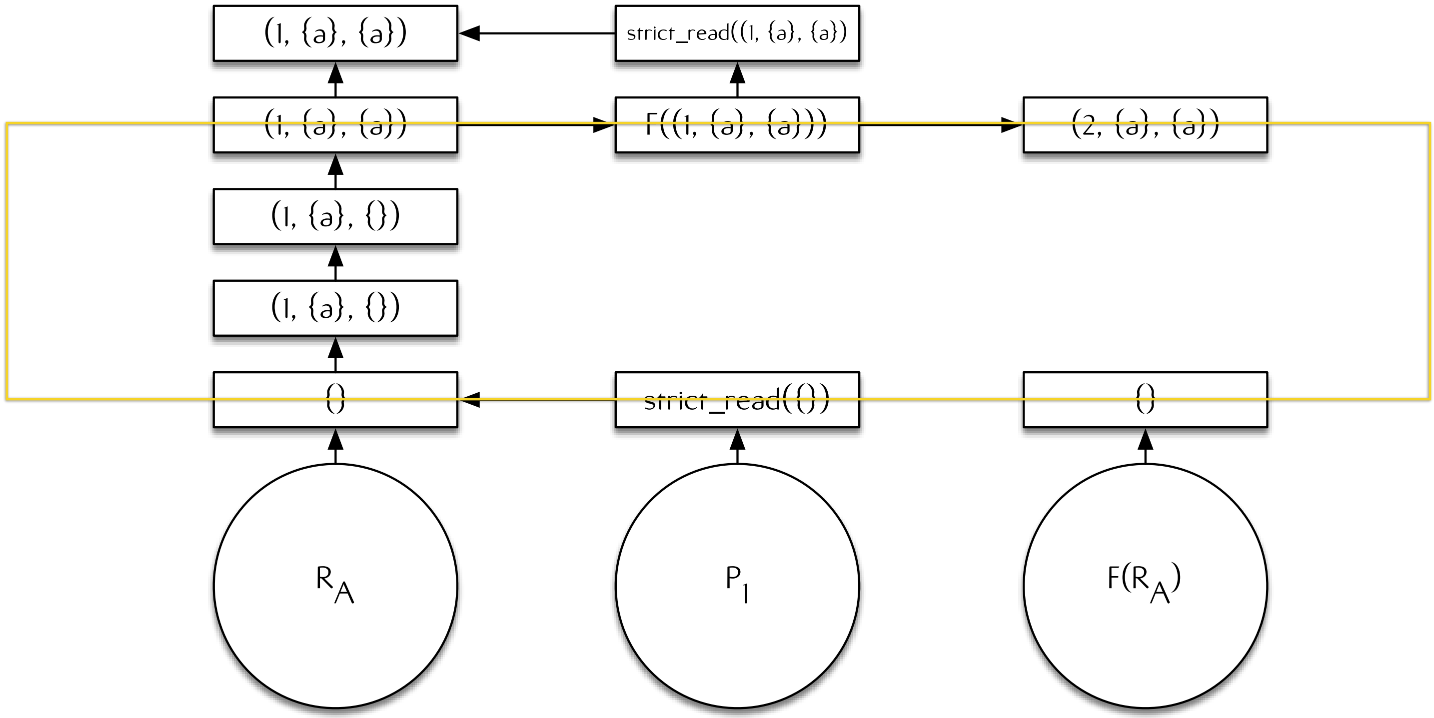
….the **process** will compute a new result.




(1, {a}, {})
strict_read((1, {a}, {})
(1, {a}, {})
F((1, {a}, {}))
(2, {a}, {})
{}
strict_read({})
{}
R_A
P_1
F(R_A)


(1, {a}, {a})
strict_read((1, {a}, {a})
(1, {a}, {a})
F((1, {a}, {a}))
(2, {a}, {a})
(1, {a}, {})
strict_read((1, {a}, {})
(1, {a}, {})
F((1, {a}, {}))
(2, {a}, {})
{}
strict_read({})
{}
R_A
P_1
$F(R_A)$

Omitted interleaving
does not sacrifice **correctness.**



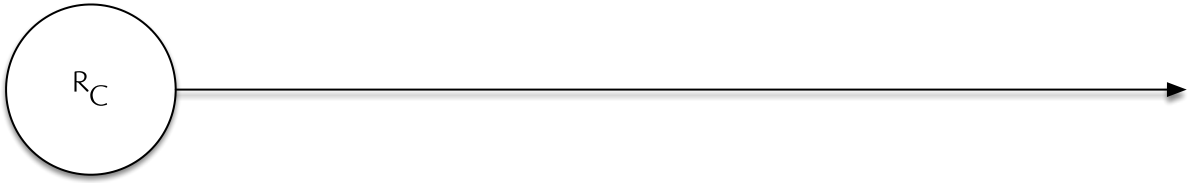
Programming **Weak Synchronization** Models

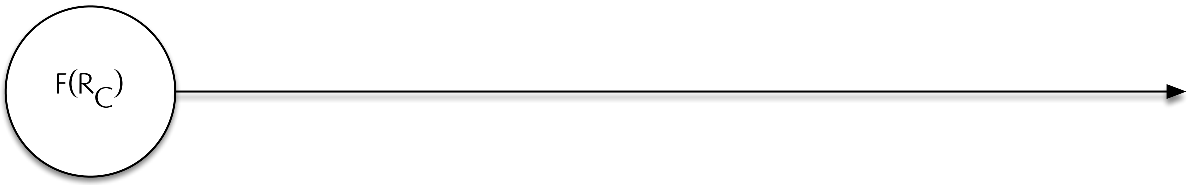
Christopher S. Meiklejohn
Université catholique de Louvain, Belgium
Instituto Superior Técnico, Portugal

Convergent Programs

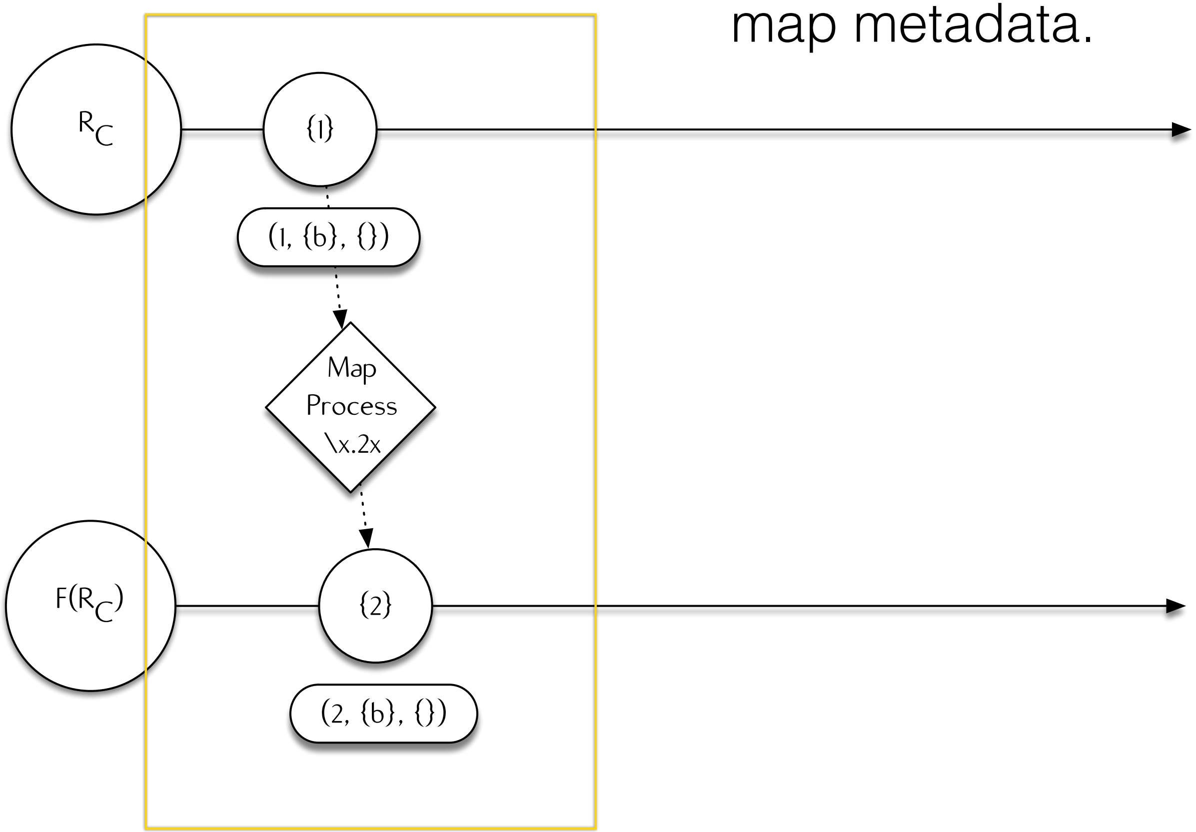
Lattice Processing

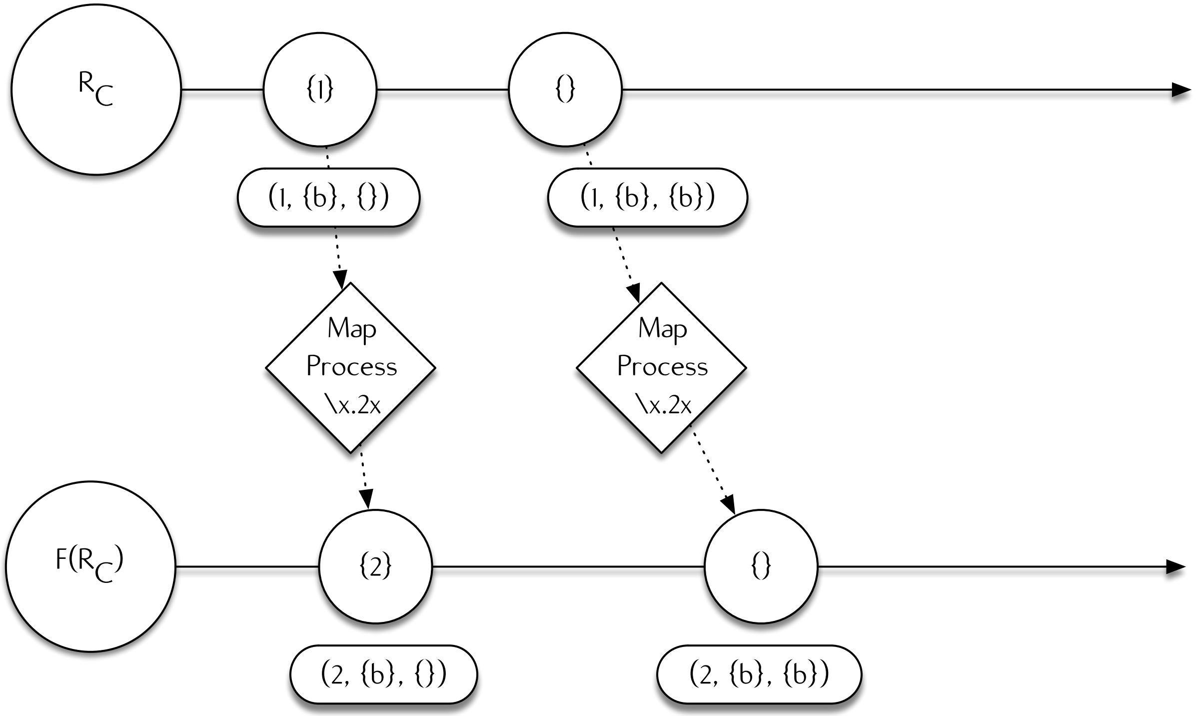
Lattice Processing
Map Example


R_C

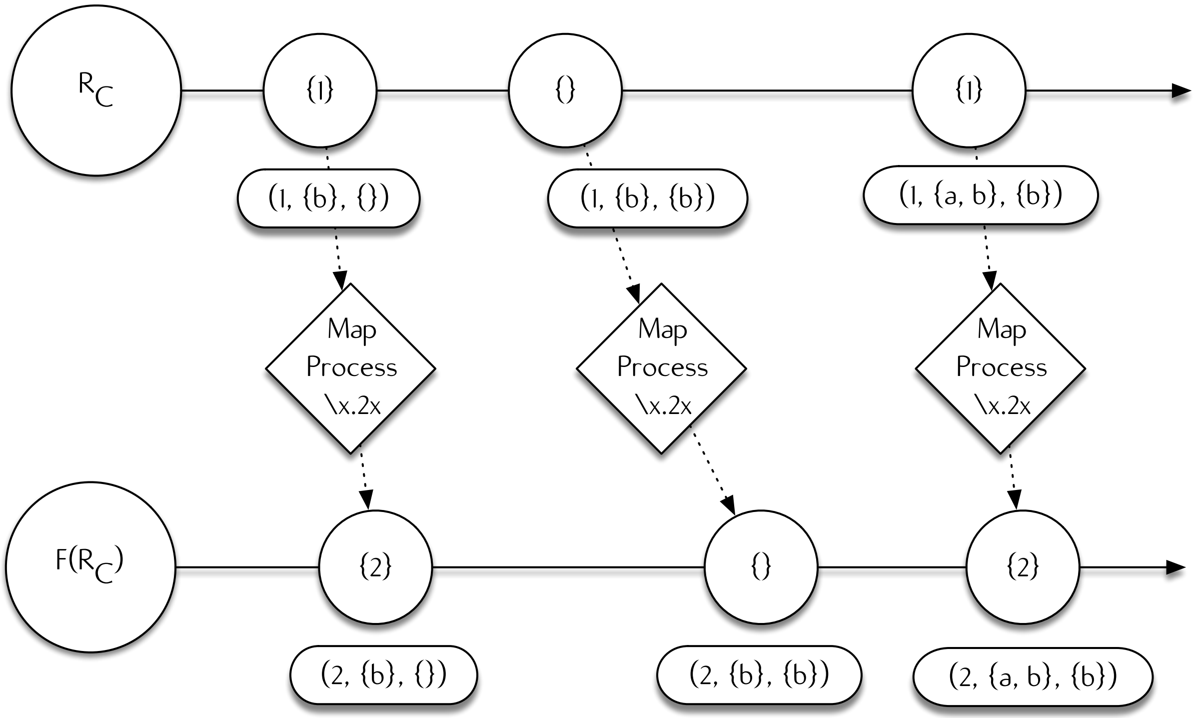

$F(R_C)$

Transform element,
map metadata.

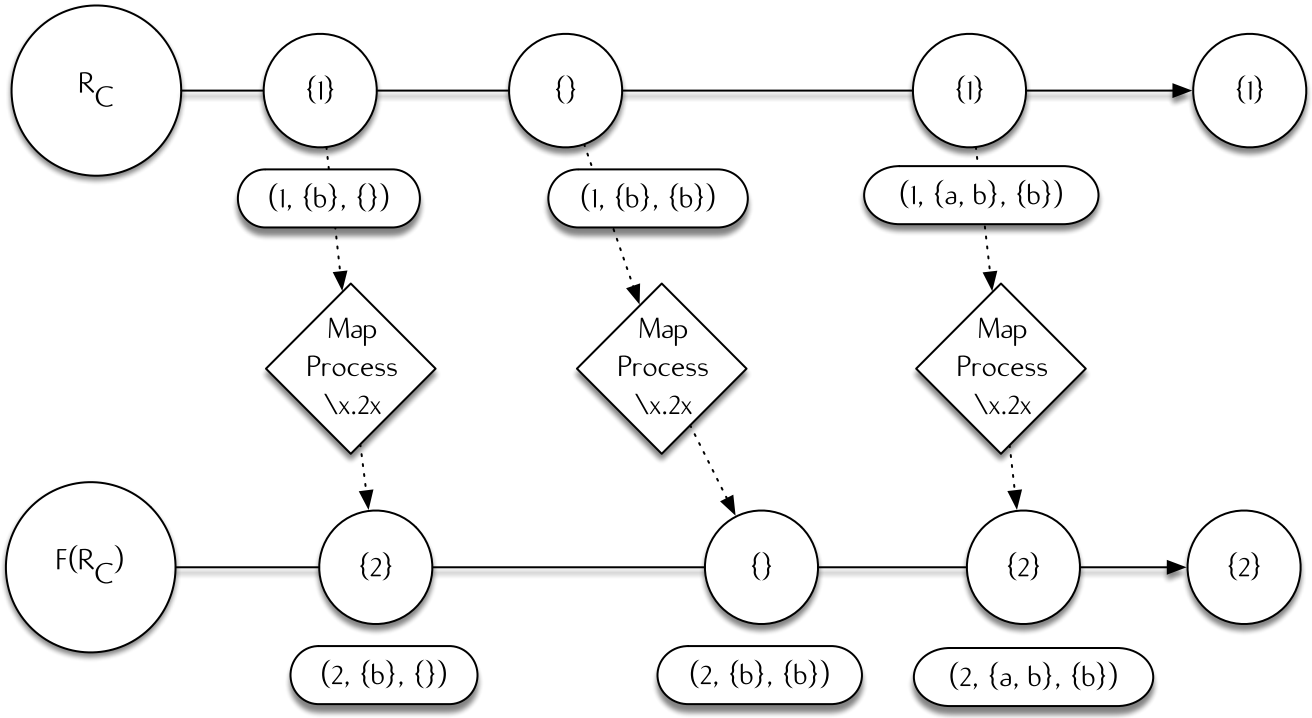




R_C
{1}
{}
(1, {b}, {})
(1, {b}, {b})
Map
Process
\x.2x
Map
Process
\x.2x
$F(R_C)$
{2}
{}
(2, {b}, {})
(2, {b}, {b})

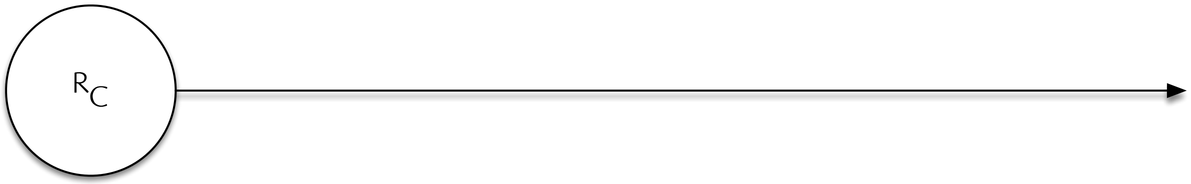


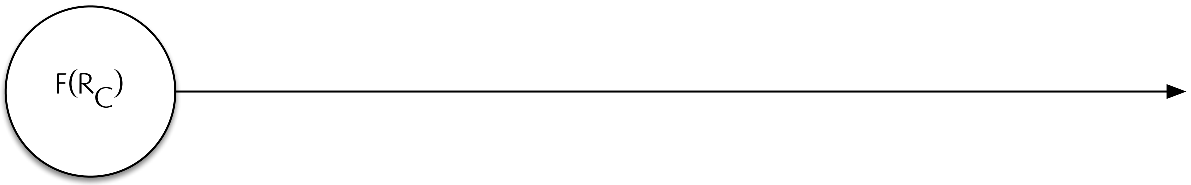
RC
{1}
{}
{1}
(1, {b}, {})
(1, {b}, {b})
(1, {a, b}, {b})
Map Process \x.2x
Map Process \x.2x
Map Process \x.2x
F(RC)
{2}
{}
{2}
(2, {b}, {})
(2, {b}, {b})
(2, {a, b}, {b})



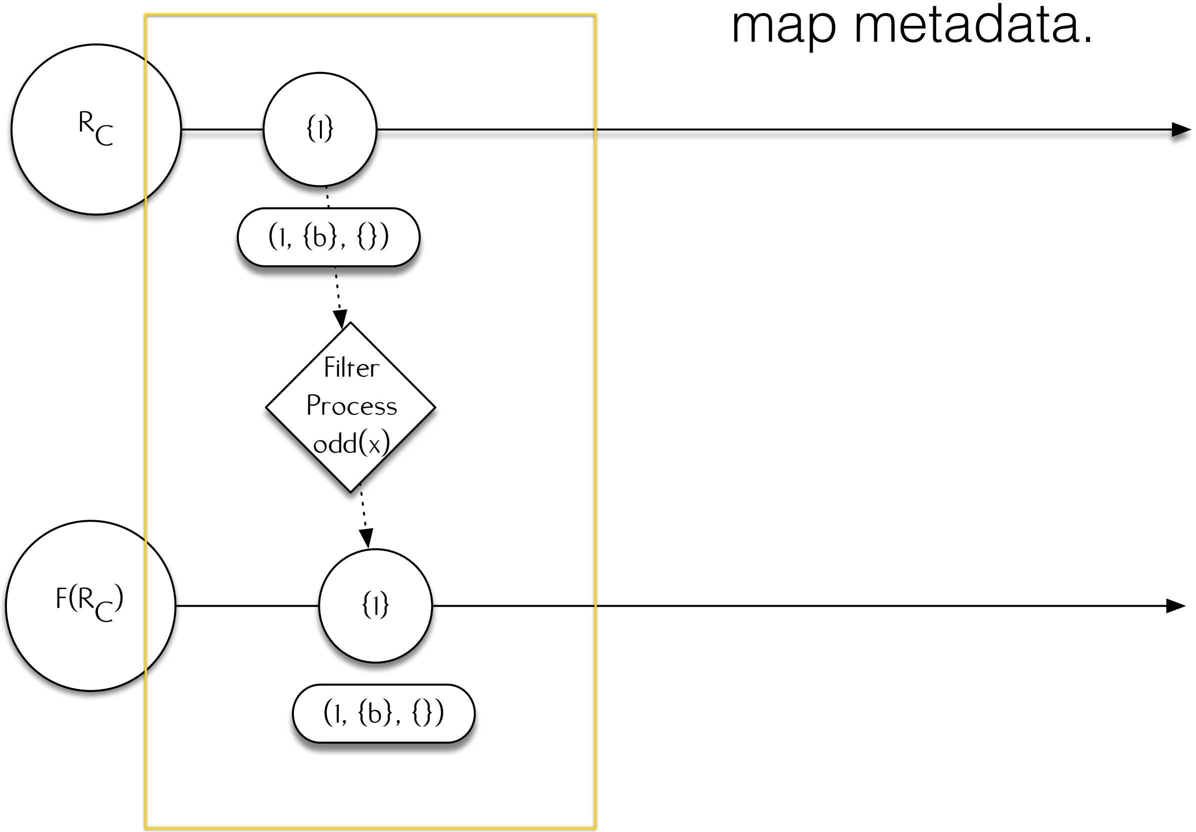
R_C
{1}
{}
{1}
{1}
(1, {b}, {})
(1, {b}, {b})
(1, {a, b}, {b})
Map
Process
\x.2x
Map
Process
\x.2x
Map
Process
\x.2x
$F(R_C)$
{2}
{}
{2}
{2}
(2, {b}, {})
(2, {b}, {b})
(2, {a, b}, {b})

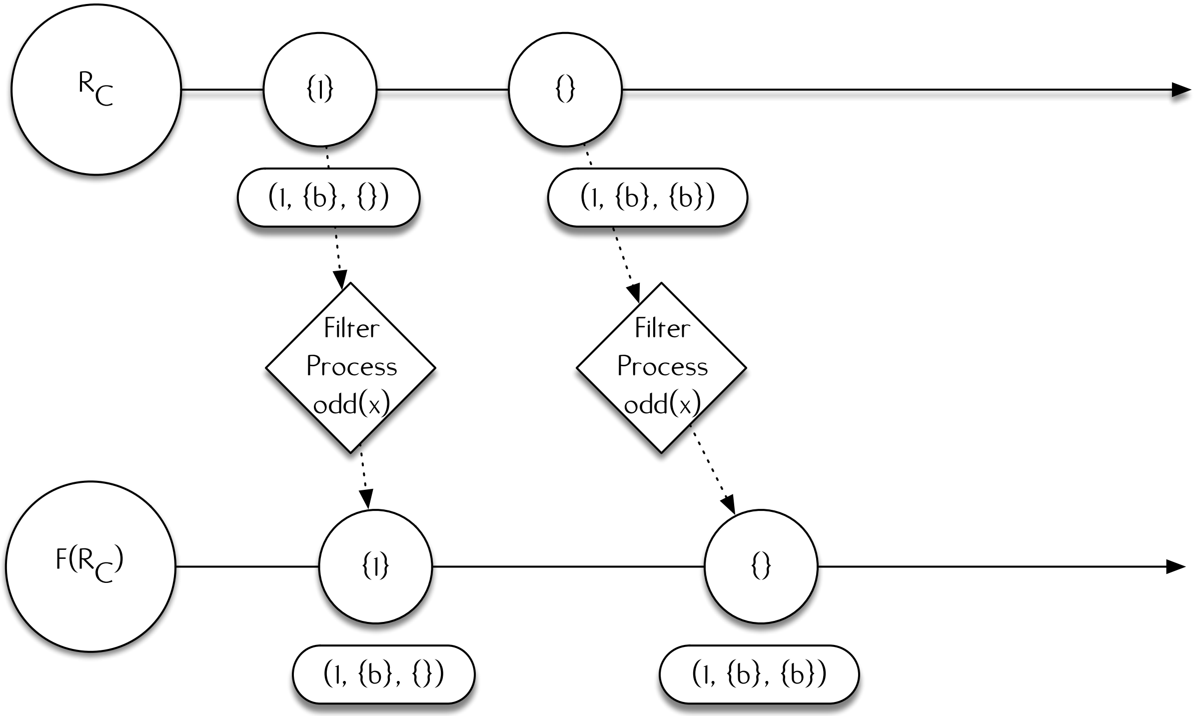
Lattice Processing
Filter Example

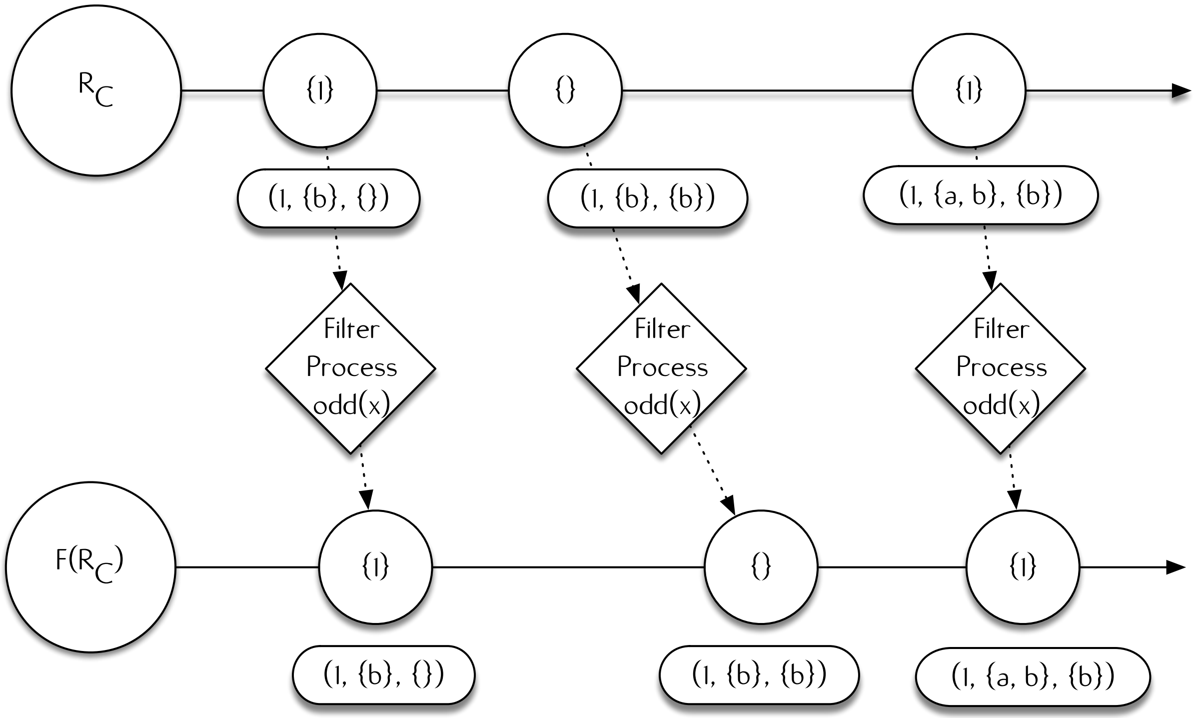

R_C


$F(R_C)$

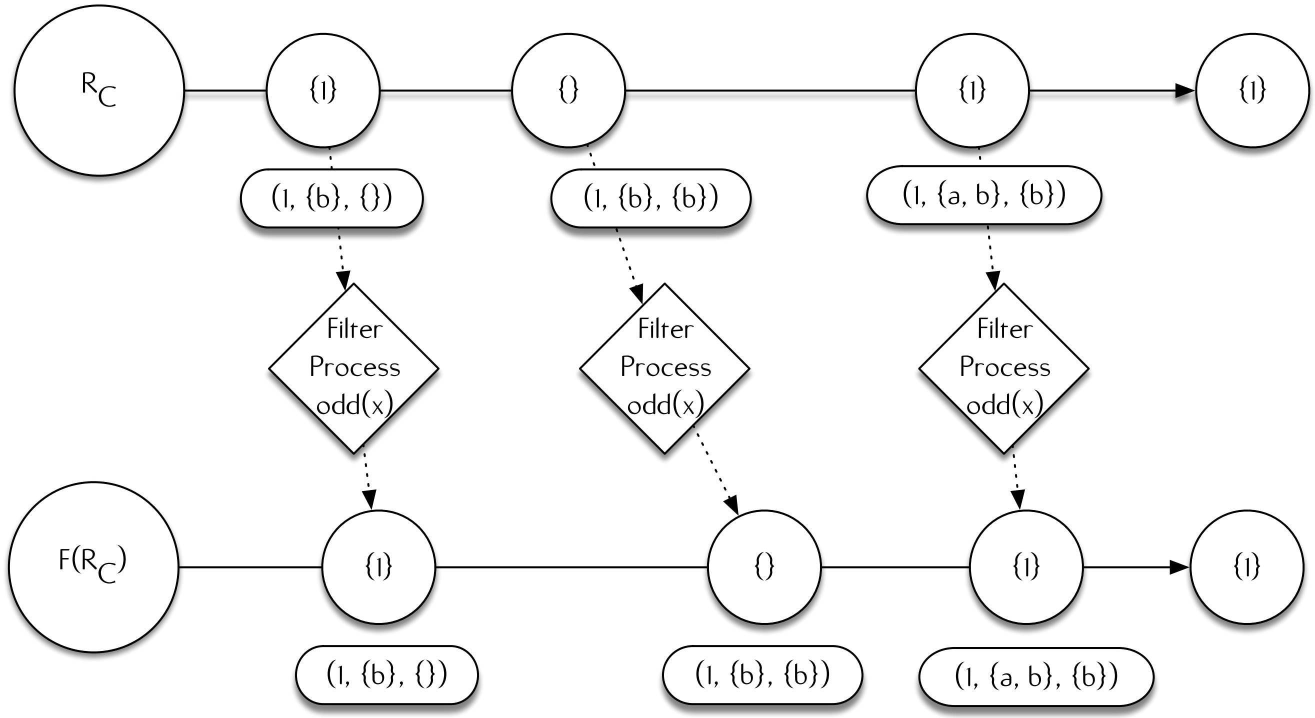
Possible omit element,
map metadata.




R_C
{1}
{}
(1, {b}, {})
(1, {b}, {b})
Filter Process odd(x)
Filter Process odd(x)
F(R_C)
{1}
{}
(1, {b}, {})
(1, {b}, {b})



R_C
{1}
{}
{1}
(1, {b}, {})
(1, {b}, {b})
(1, {a, b}, {b})
Filter
Process
odd(x)
Filter
Process
odd(x)
Filter
Process
odd(x)
$F(R_C)$
{1}
{}
{1}
(1, {b}, {})
(1, {b}, {b})
(1, {a, b}, {b})



R_C
{1}
{}
{1}
{1}
(1, {b}, {})
(1, {b}, {b})
(1, {a, b}, {b})
Filter
Process
odd(x)
Filter
Process
odd(x)
Filter
Process
odd(x)
$F(R_C)$
{1}
{}
{1}
{1}
(1, {b}, {})
(1, {b}, {b})
(1, {a, b}, {b})

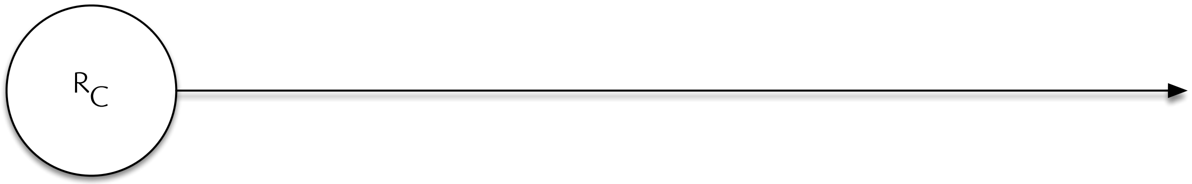
Lattice Processing
Fold Example

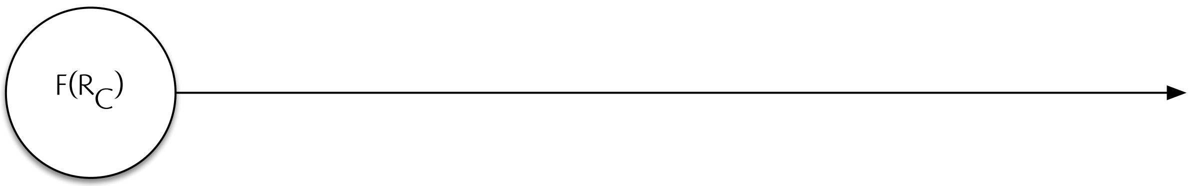
Fold Operation

- Morphism between CRDTs
  For example, from a CRDT set to a CRDT counter

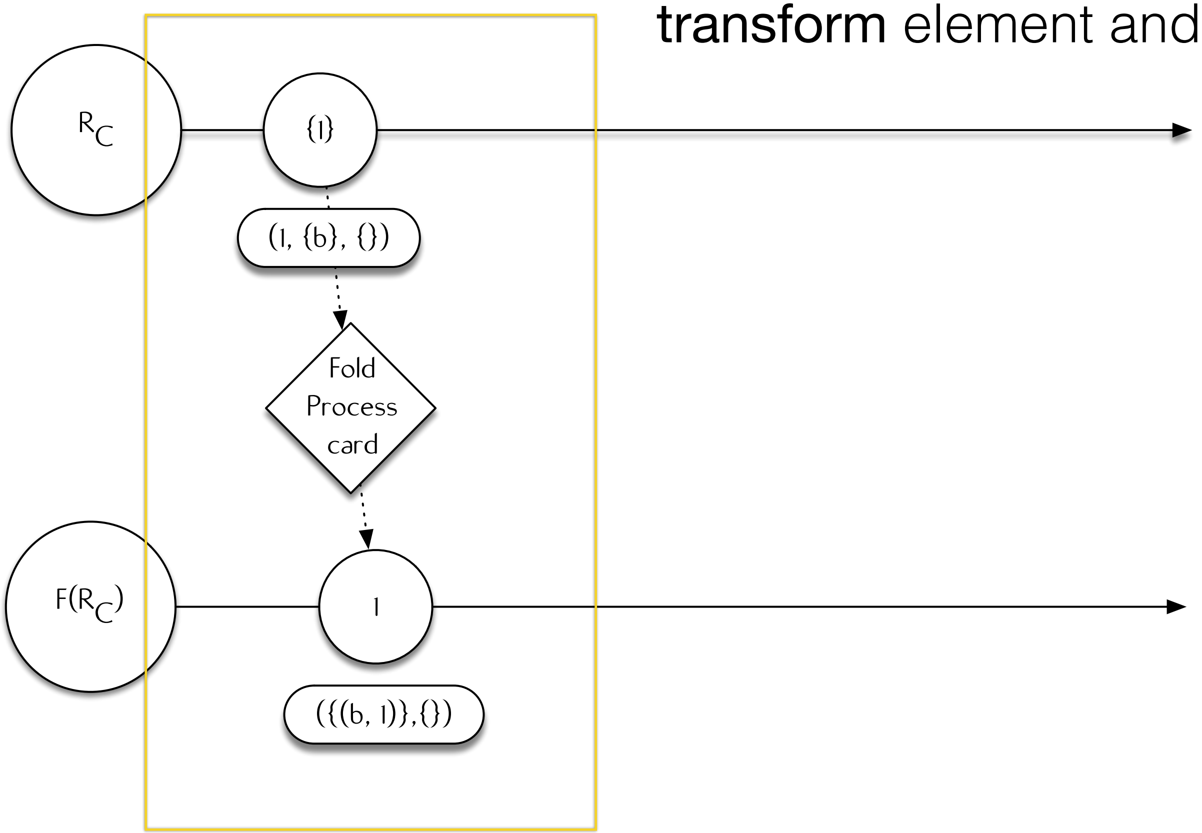
- Restricted in expressiveness
  "Unordered" sets imply combiner must be associative, commutative, idempotent

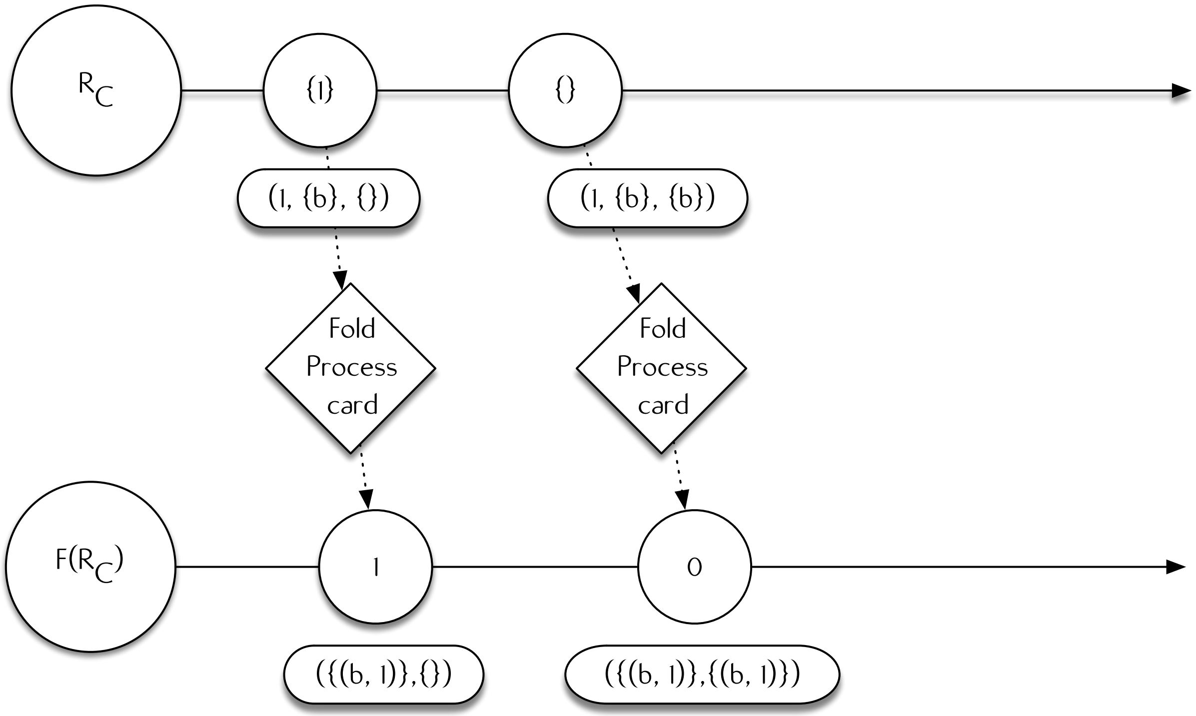
- Invertable
  Operations must have an inverse for operating on "tombstone" values


R_C

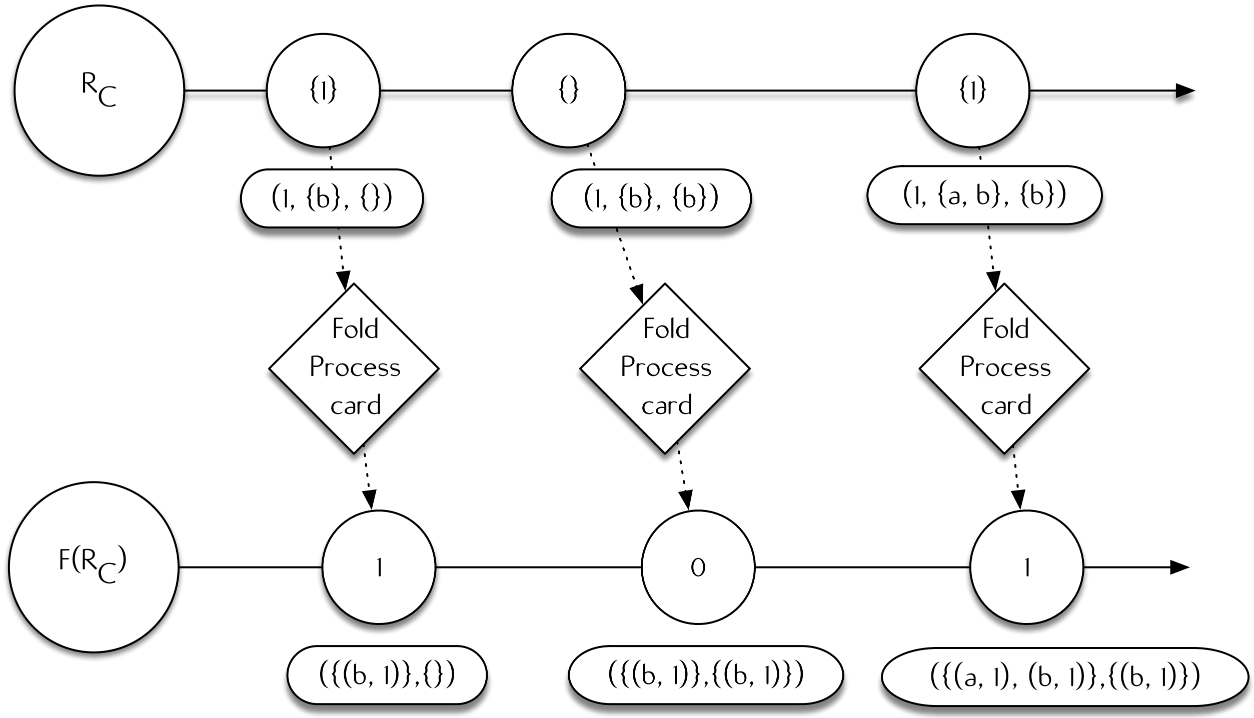

$F(R_C)$

Apply morphism and **transform** element and metadata.

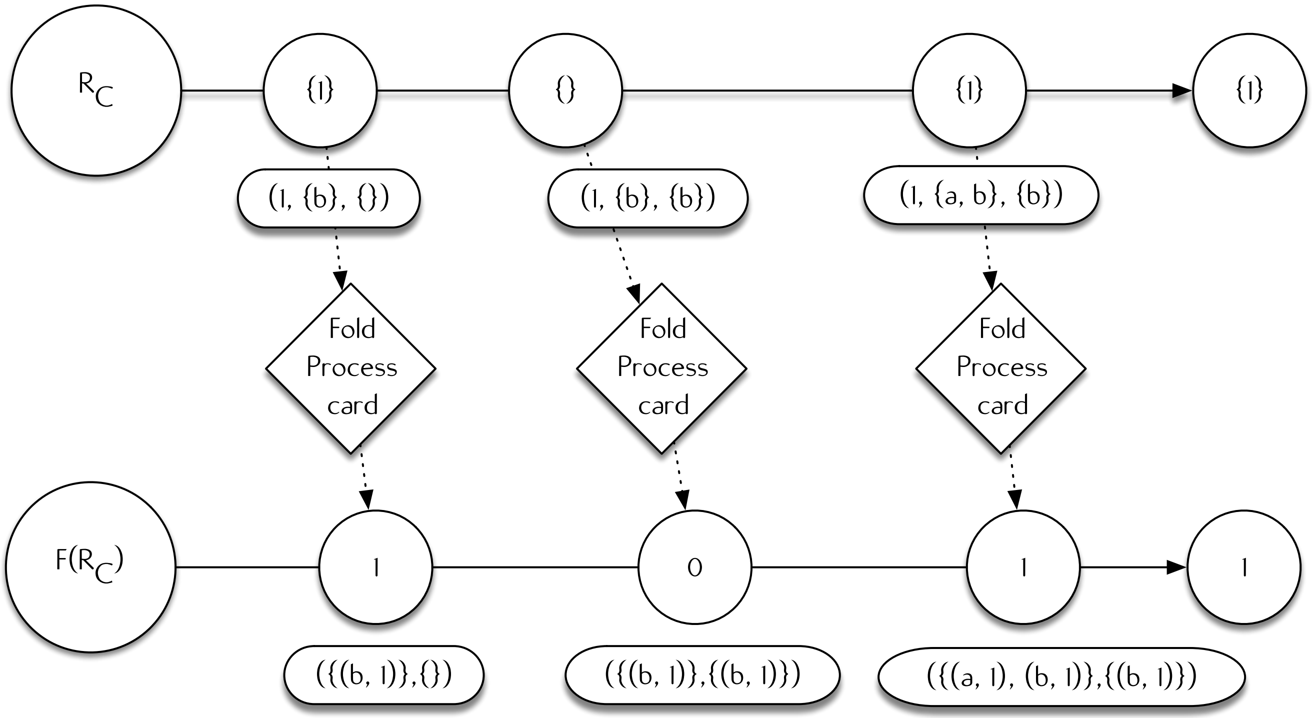




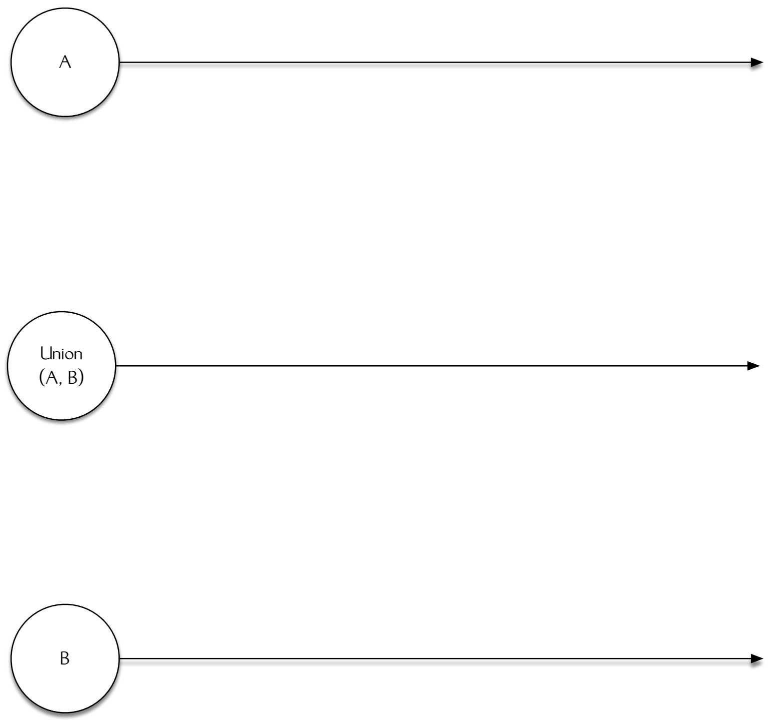
R_C
{1}
{}
(1, {b}, {})
(1, {b}, {b})
Fold Process card
Fold Process card
$F(R_C)$
1
0
({(b, 1)},{})
({(b, 1)},{(b, 1)})

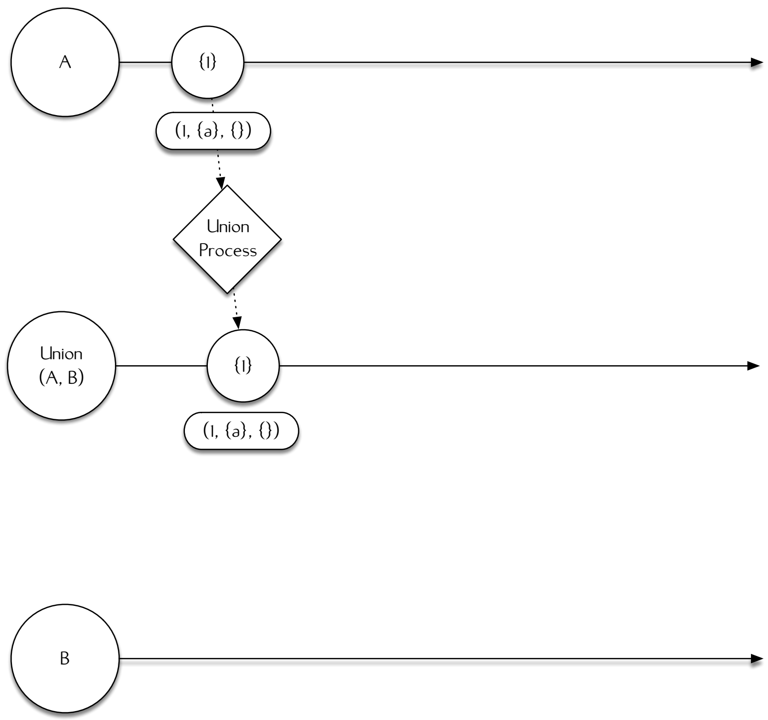


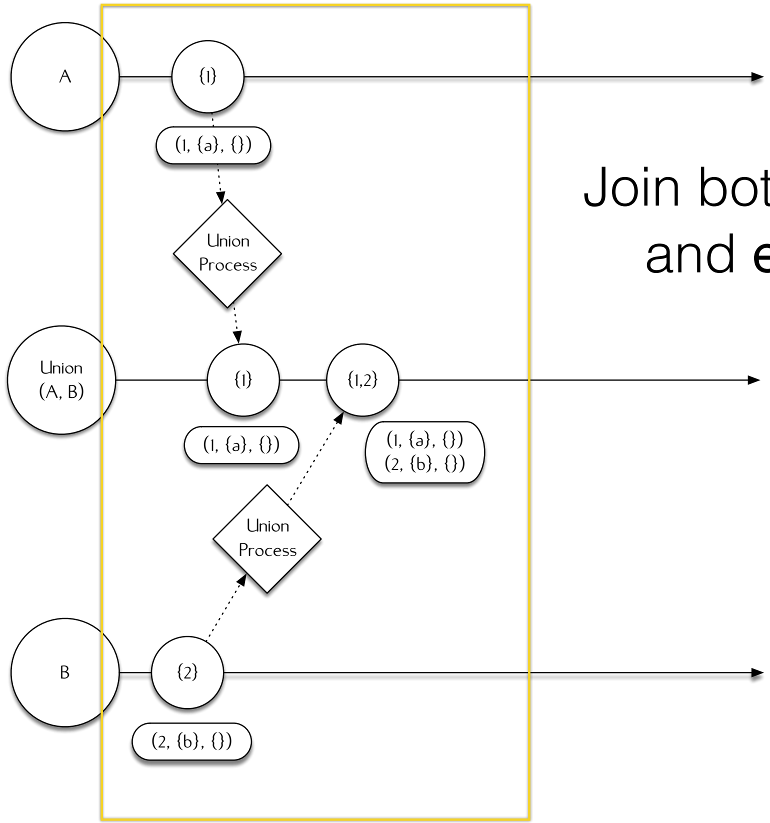
R C
{1}
{}
{1}
(1, {b}, {})
(1, {b}, {b})
(1, {a, b}, {b})
Fold Process card
Fold Process card
Fold Process card
F(R C)
1
0
1
({(b, 1)},{})
({(b, 1)},{(b, 1)})
({(a, 1), (b, 1)},{(b, 1)})


R_C
{1}
{}
{1}
{1}
(1, {b}, {})
(1, {b}, {b})
(1, {a, b}, {b})
Fold Process card
Fold Process card
Fold Process card
F(R_C)
1
0
1
1
({(b, 1)},{})
({(b, 1)},{(b, 1)})
({(a, 1), (b, 1)},{(b, 1)})

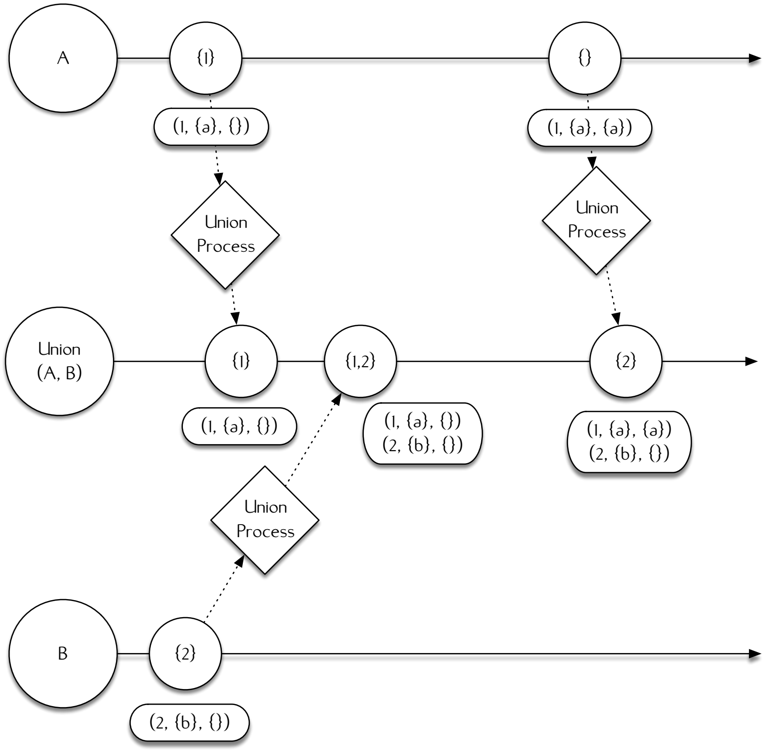
Lattice Processing
Union Example

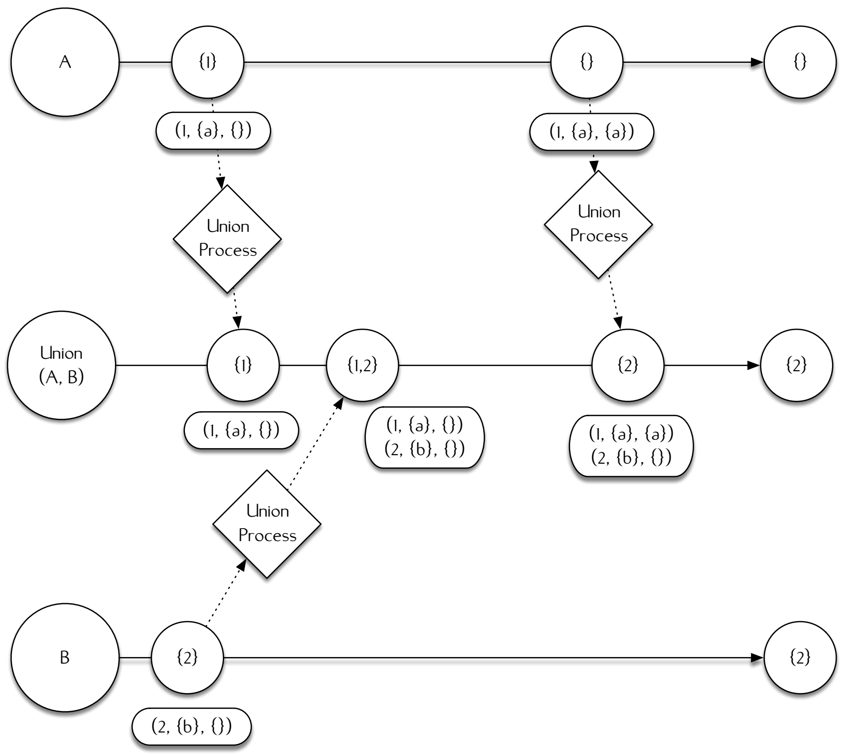

A
Union
(A, B)
B


A
{1}
(1, {a}, {})
Union
Process
Union
(A, B)
{1}
(1, {a}, {})
B

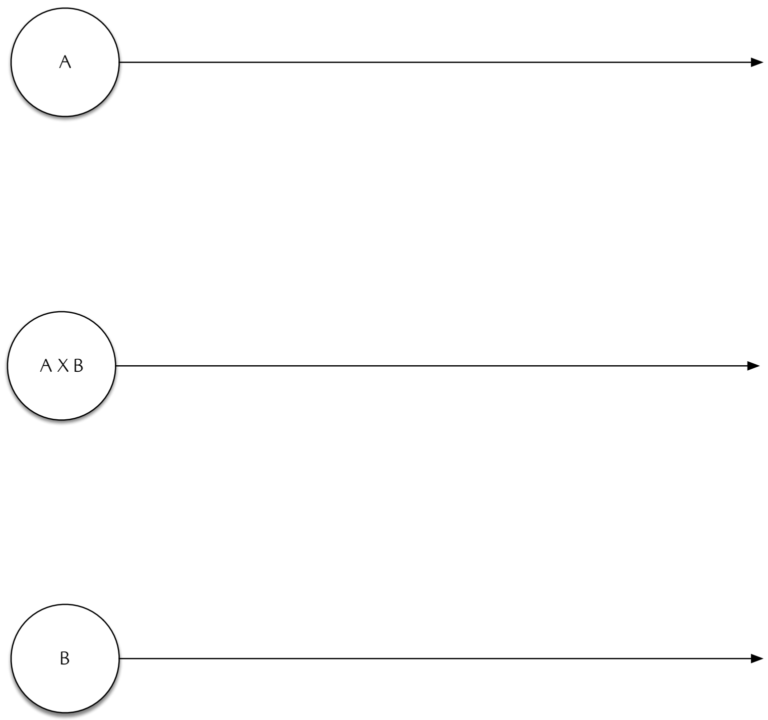


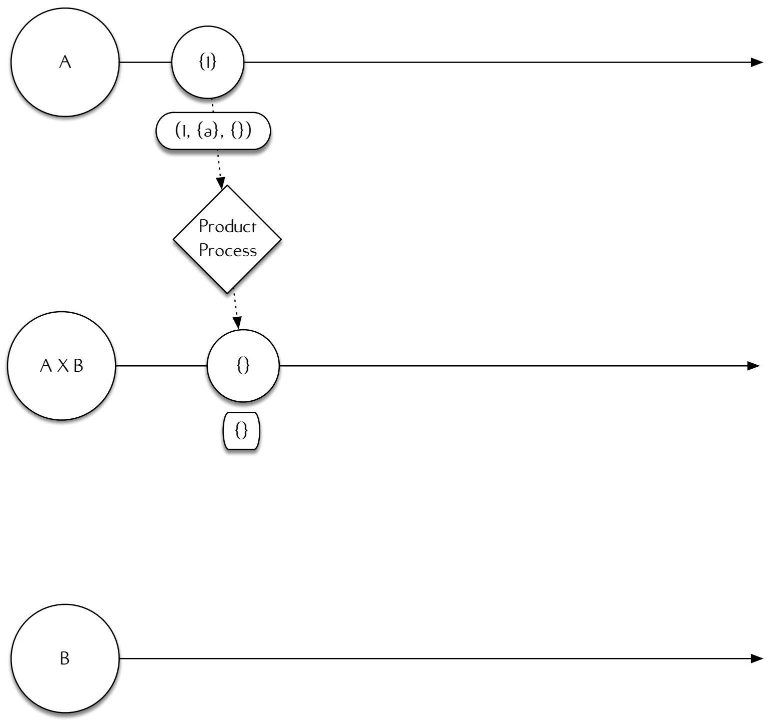
Join both **metadata** and **elements**.

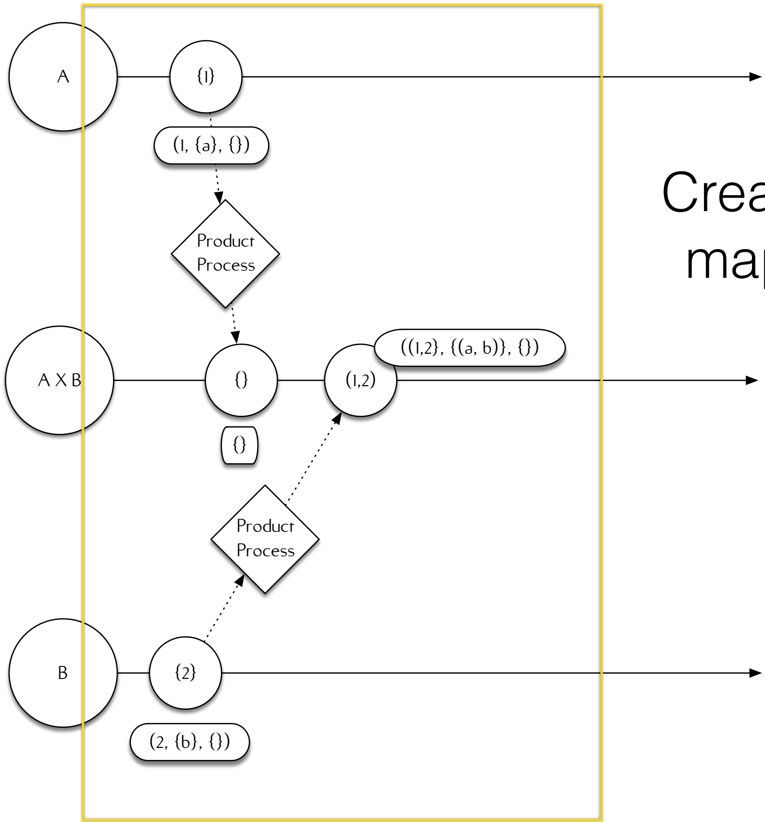

A
{1}
(1, {a}, {})
{}
(1, {a}, {a})
Union
Process
Union
Process
Union
(A, B)
{1}
{1,2}
{2}
(1, {a}, {})
(1, {a}, {})
(2, {b}, {})
(1, {a}, {a})
(2, {b}, {})
Union
Process
B
{2}
(2, {b}, {})


A
{1}
{}
{}
(1, {a}, {})
(1, {a}, {a})
Union Process
Union Process
Union (A, B)
{1}
{1,2}
{2}
{2}
(1, {a}, {})
(1, {a}, {})
(2, {b}, {})
(1, {a}, {a})
(2, {b}, {})
Union Process
B
{2}
{2}
(2, {b}, {})

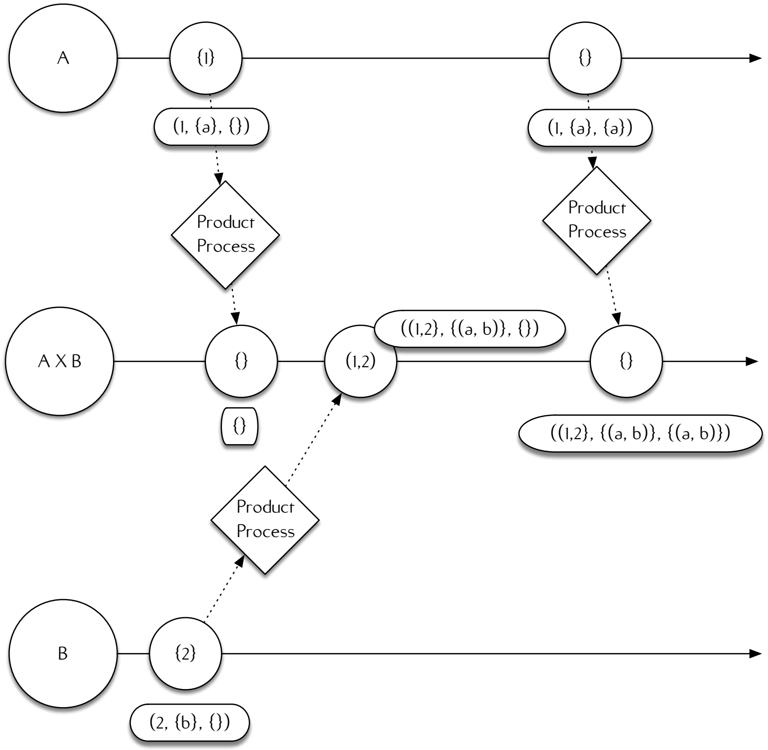
Lattice Processing
Product Example


A
A X B
B

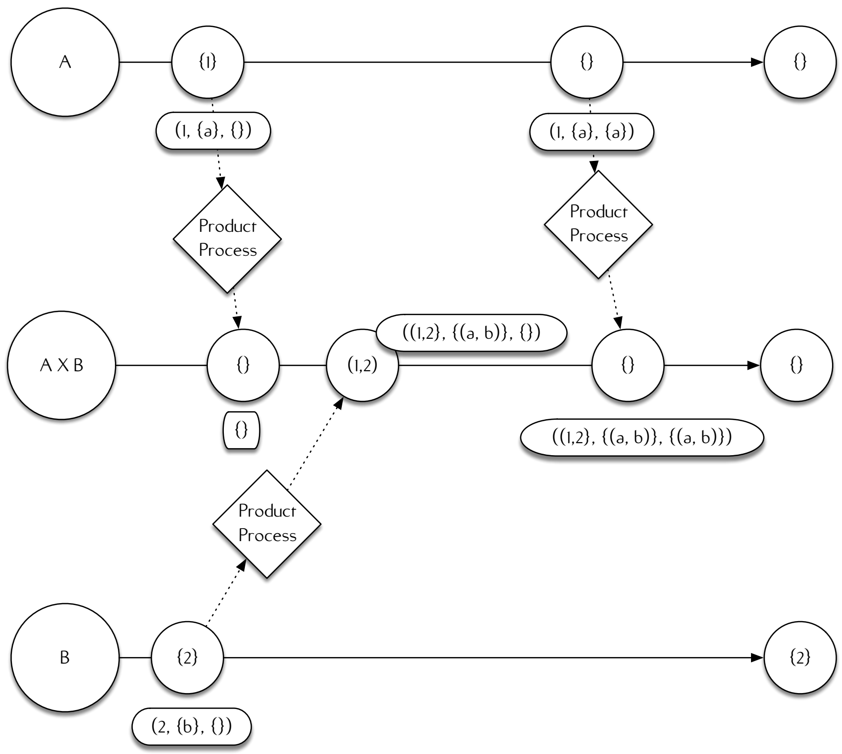

A
{1}
(1, {a}, {})
Product
Process
A X B
{}
{}
B



Create new **elements** and map **metadata** through.

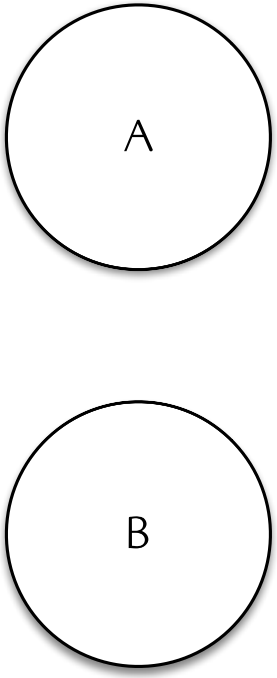


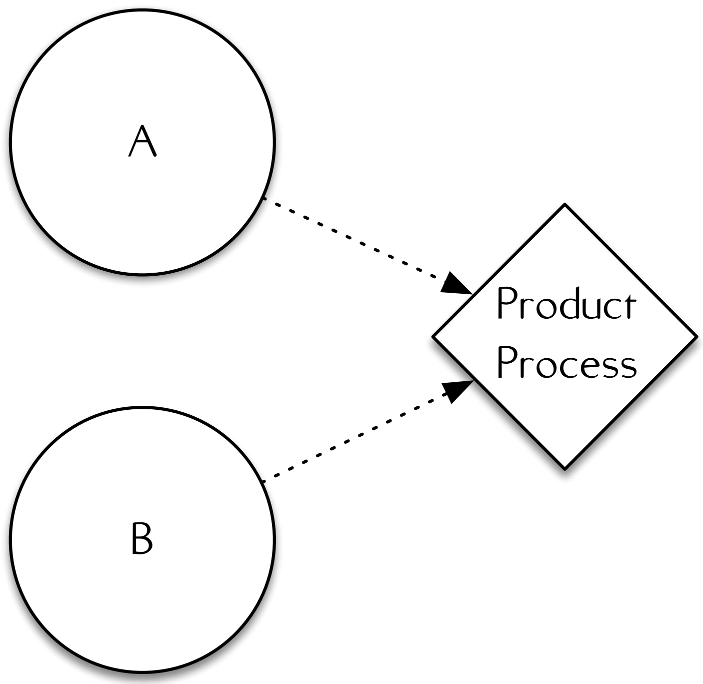
A
{1}
{}
(1, {a}, {})
(1, {a}, {a})
Product Process
Product Process
A X B
{}
(1,2)
((1,2}, {(a, b)}, {})
{}
{}
((1,2}, {(a, b)}, {(a, b)})
Product Process
B
{2}
(2, {b}, {})

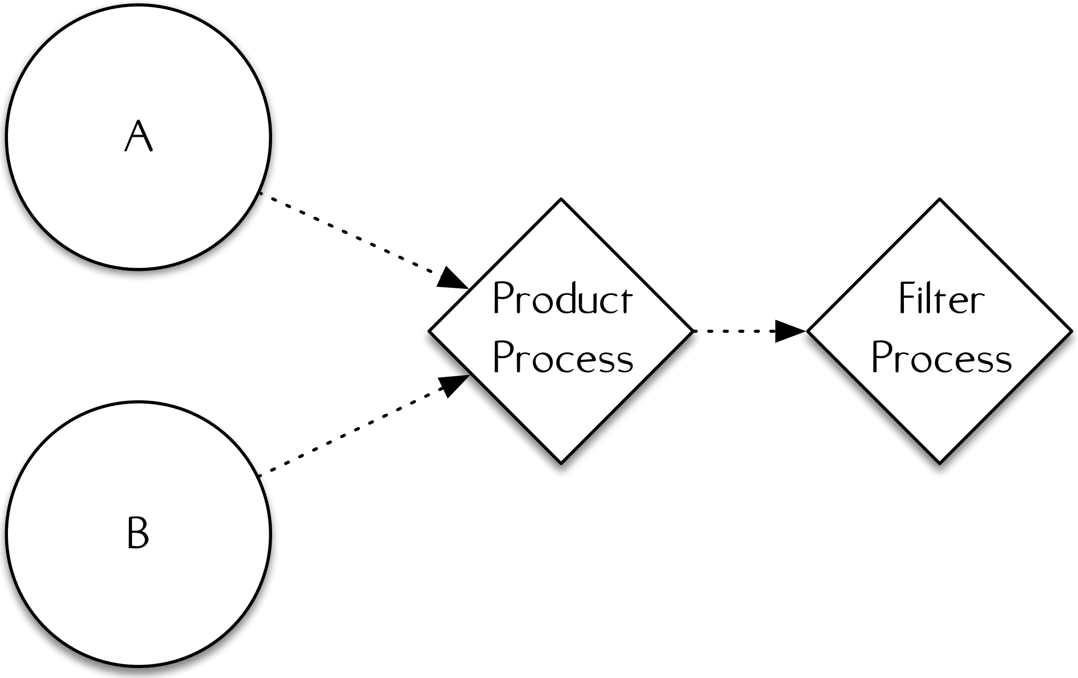


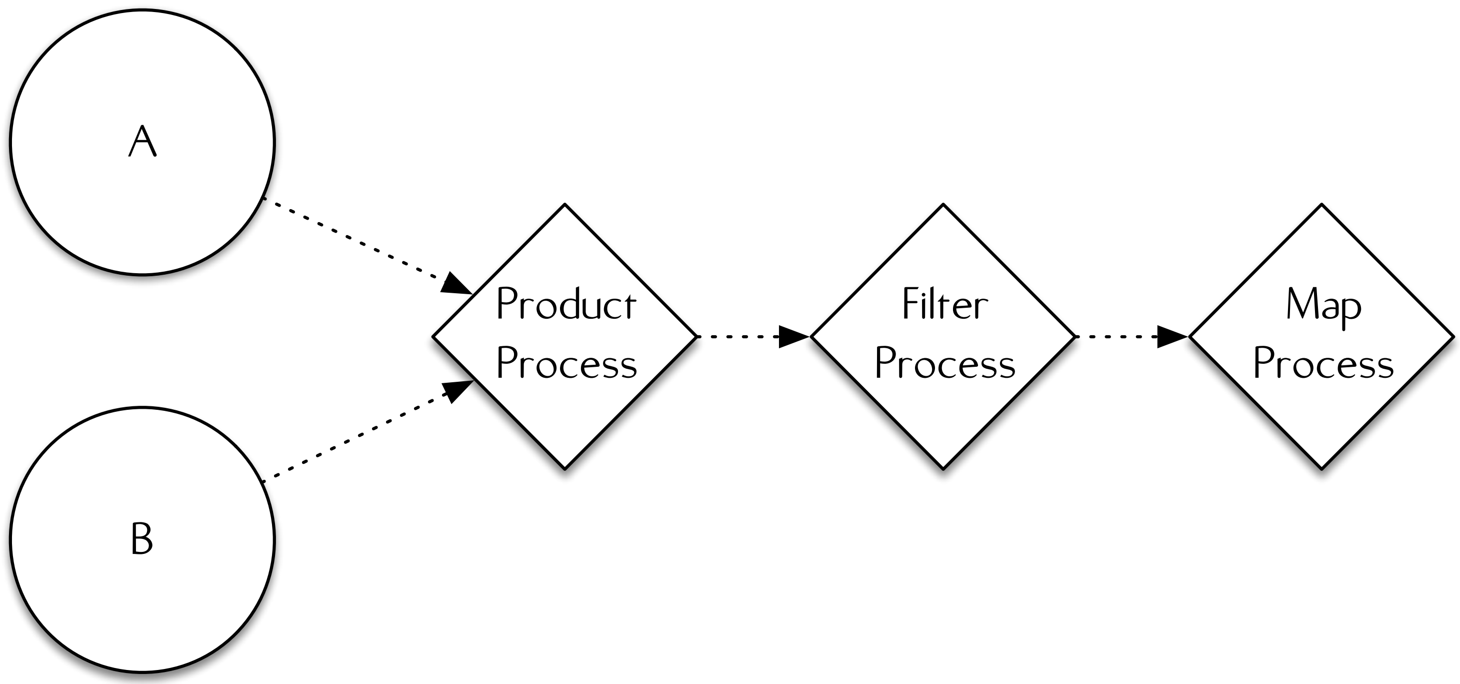
A
{1}
{}
{}
(1, {a}, {})
(1, {a}, {a})
Product Process
Product Process
A X B
{}
(1,2)
{}
{}
((1,2}, {(a, b)}, {})
{}
((1,2}, {(a, b)}, {(a, b)})
Product Process
B
{2}
{2}
(2, {b}, {})

Lattice Processing
Intersection Example


A
B


A
B
Product
Process


A
B
Product
Process
Filter
Process


A
B
Product Process
Filter Process
Map Process

Programming **Weak Synchronization** Models

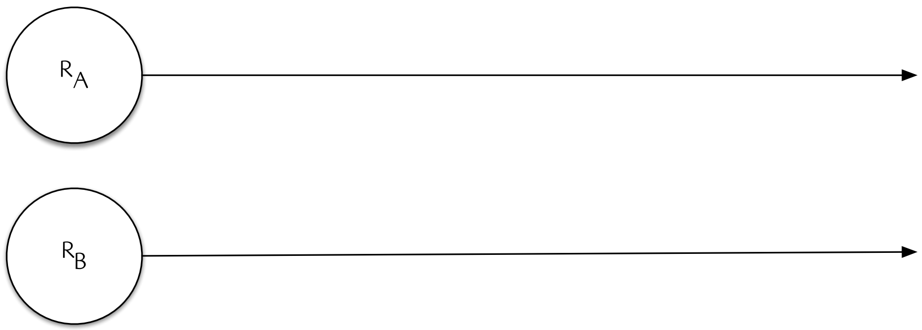
Christopher S. Meiklejohn
Université catholique de Louvain, Belgium
Instituto Superior Técnico, Portugal

Example Application
Advertisement Counter

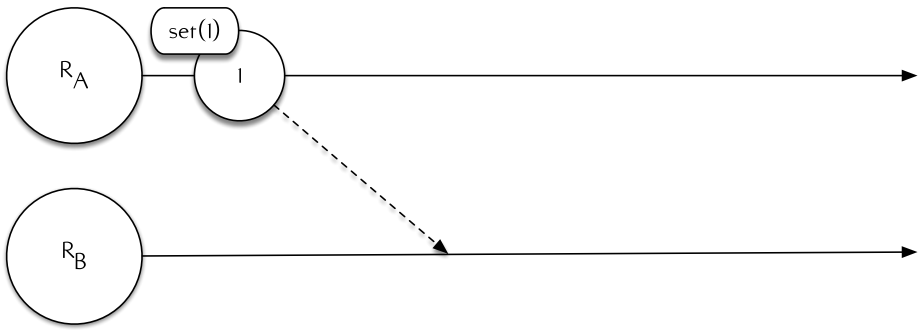
Advertisement Counter

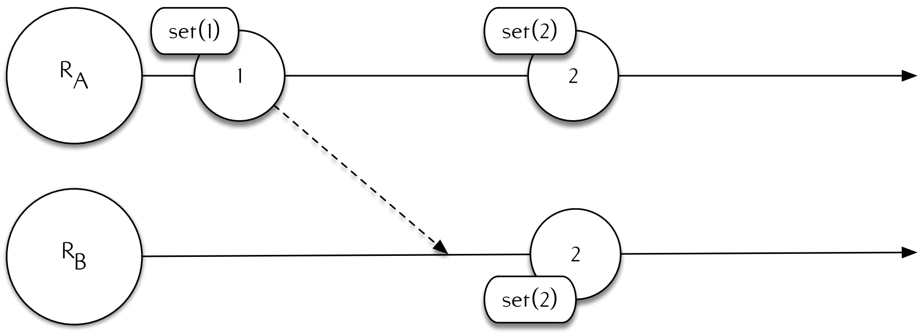
- Lower-bound invariant
  Advertisements are paid according to a minimum number of impressions
- Clients will go offline
  Clients have limited connectivity and the system still needs to make progress while clients are offline
- No lost updates
  All displayed advertisements should be accounted for, with no lost updates

Advertisement Counter
Losing Updates

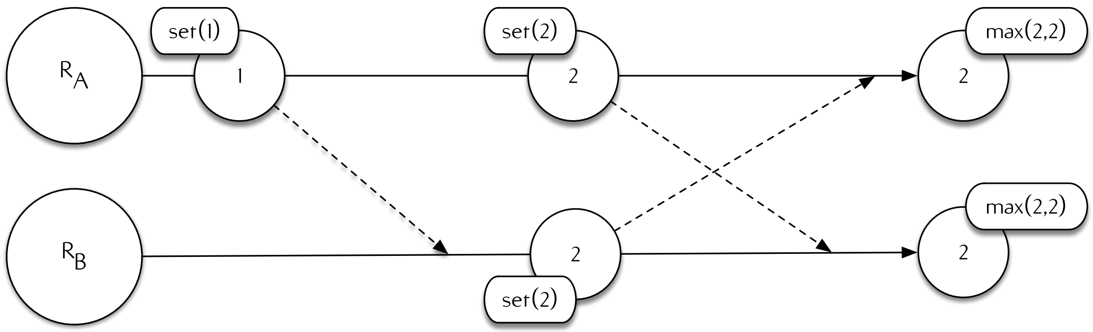


R_A
R_B

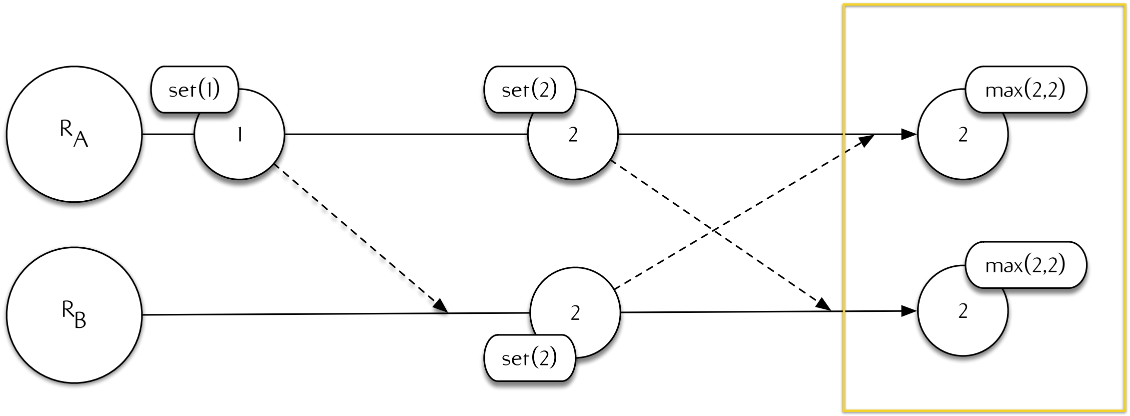

set(1)
R_A
1
R_B



R_A
set(1)
1
set(2)
2
R_B
2
set(2)



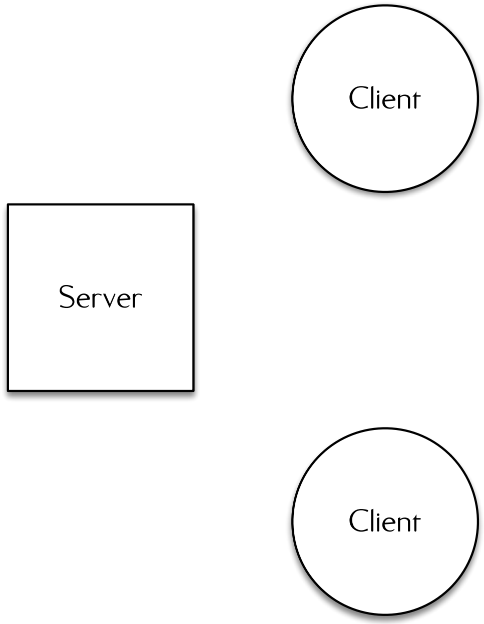
set(1)
set(2)
max(2,2)
R_A
1
2
2
R_B
2
set(2)
max(2,2)
2

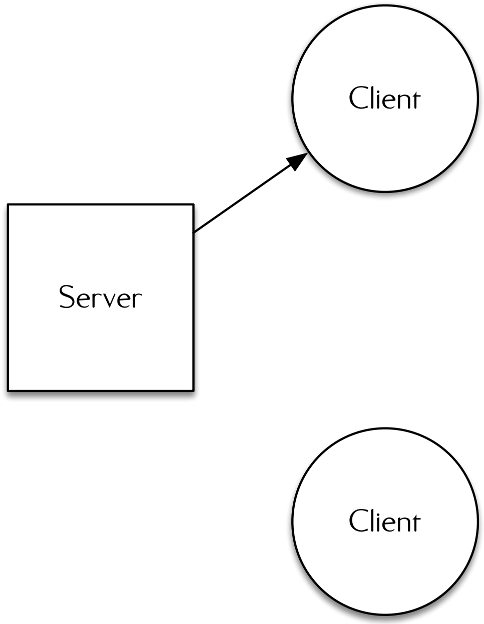


Incorrect value is computed because of incompatible lattice.

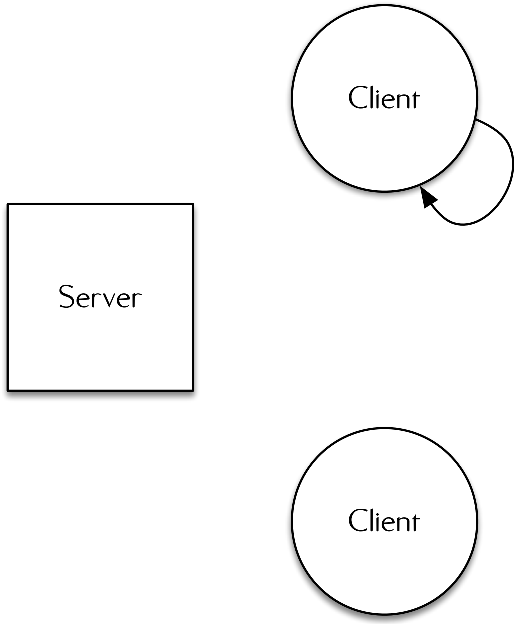
Advertisement Counter

Application Flow

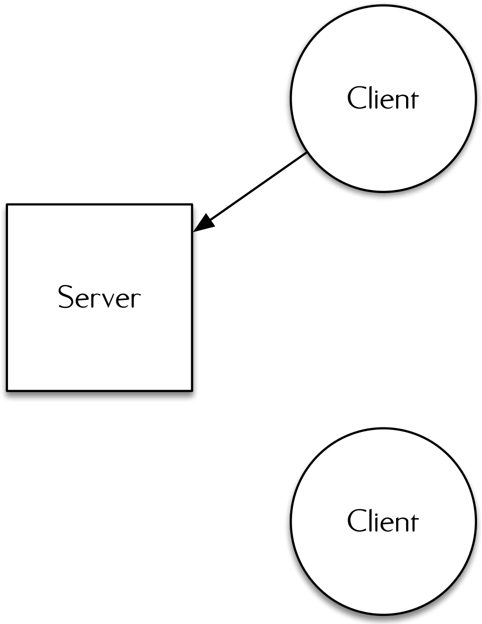

Client
Server
Client



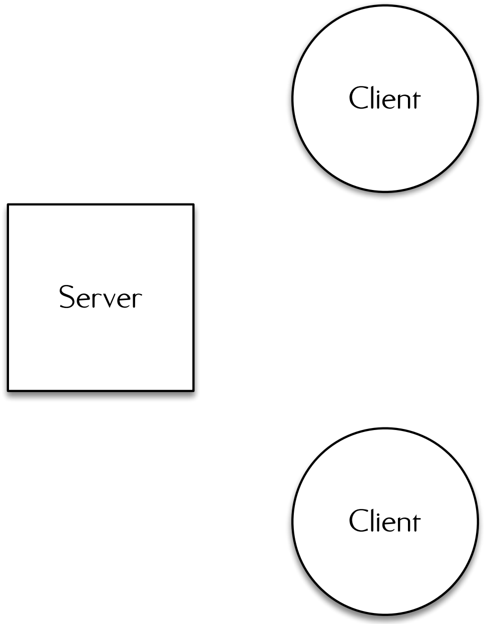
Client
reads state
from the server.

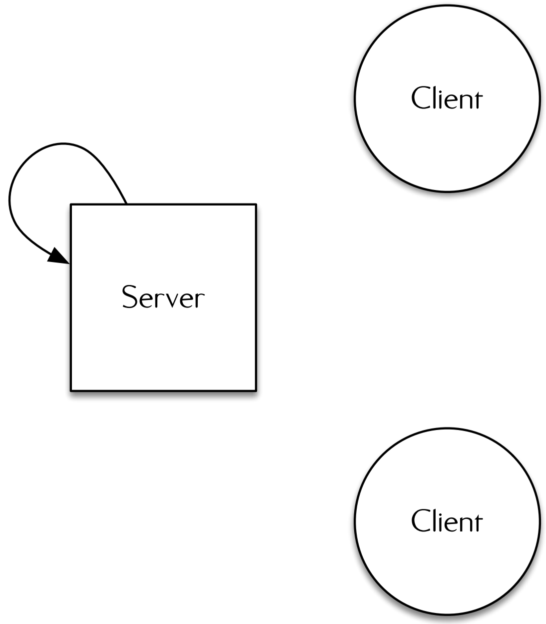


Client
locally mutates
state.

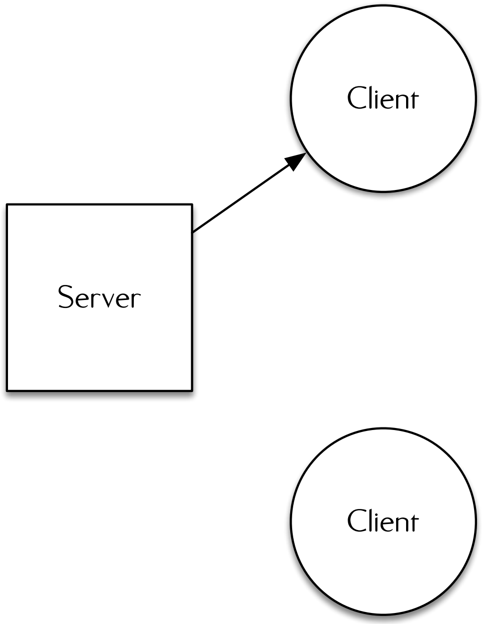


Client
pushes changes
back to
the server.

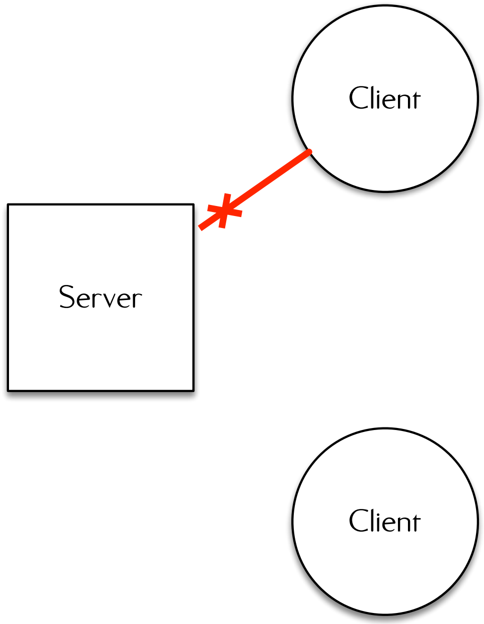

Client
Server
Client



Server
enforces invariants
over state.

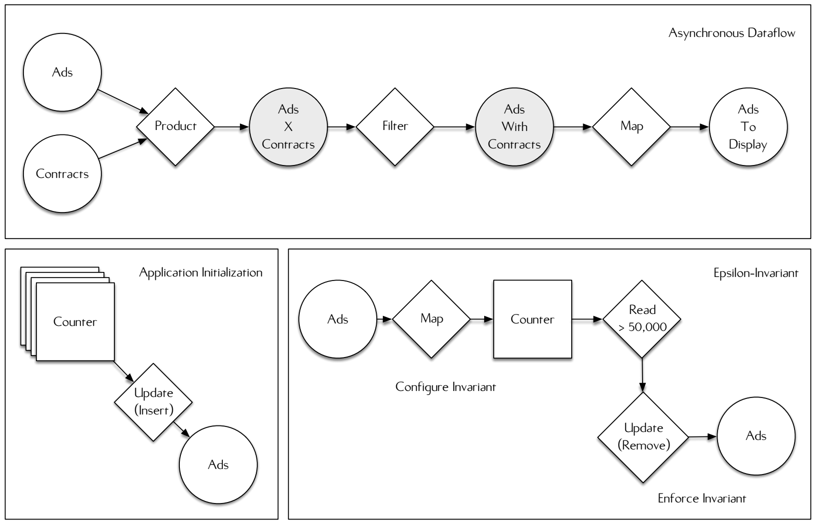


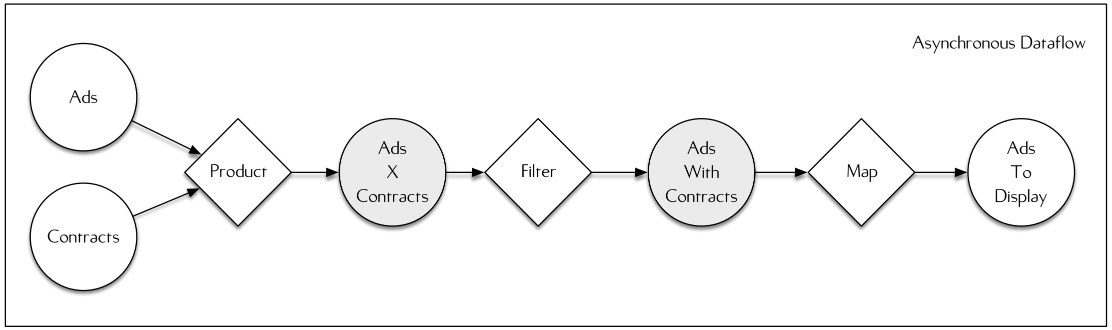
Client retrieves updated state **periodically.**

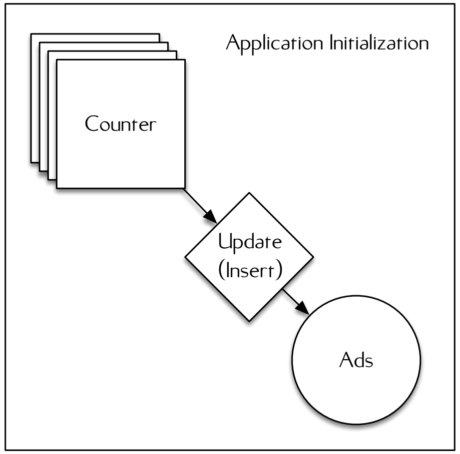


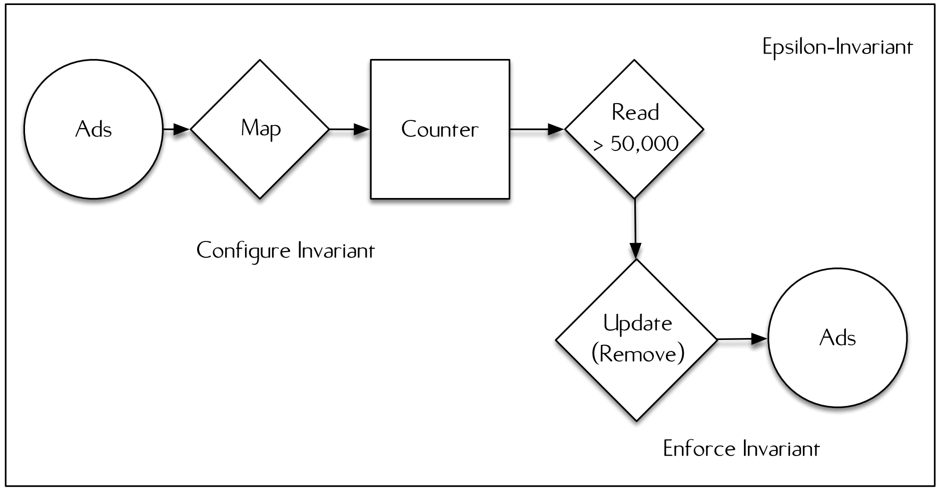
Clients
unable to communicate
may
violate invariant.

Advertisement Counter
Application Design

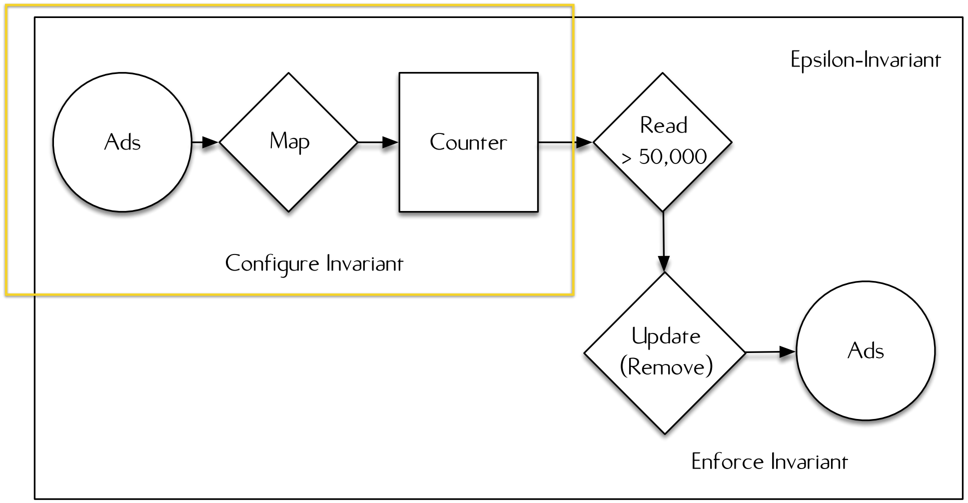

Asynchronous Dataflow
Ads
Contracts
Product
Ads
X
Contracts
Filter
Ads
With
Contracts
Map
Ads
To
Display
Application Initialization
Counter
Update
(Insert)
Ads
Epsilon-Invariant
Ads
Map
Counter
Read
> 50,000
Configure Invariant
Update
(Remove)
Ads
Enforce Invariant

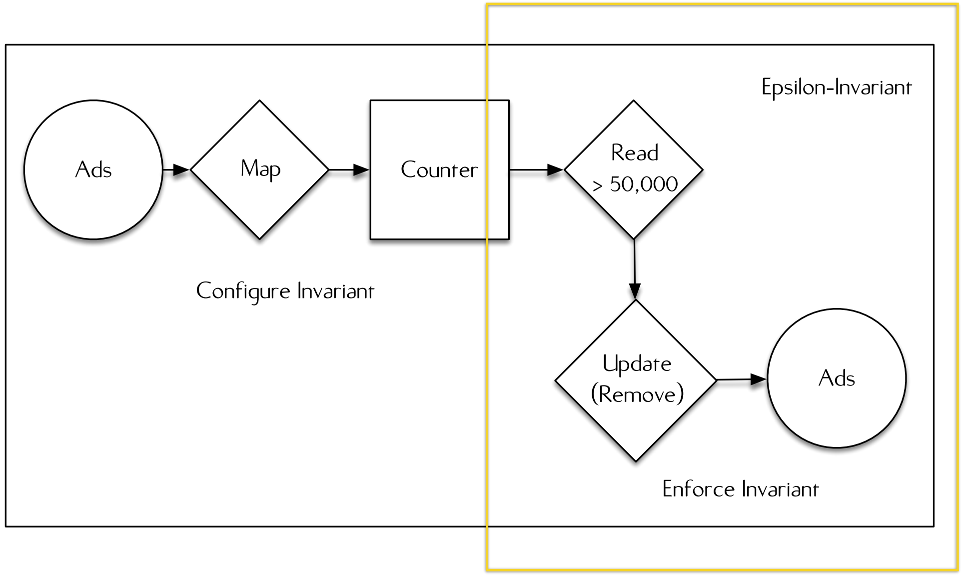

Asynchronous Dataflow
Ads
Contracts
Product
Ads
X
Contracts
Filter
Ads
With
Contracts
Map
Ads
To
Display


Application Initialization
Counter
Update
(Insert)
Ads


Epsilon-Invariant
Ads
Map
Counter
Read
> 50,000
Configure Invariant
Update
(Remove)
Ads
Enforce Invariant

Configure
invariants
for all of the
advertisements.





Remove the advertisement from the list.

Advertisement Counter

- Completely monotonic
  Disabling advertisements and contracts are all modeled through monotonic state growth

- Arbitrary distribution
  Use of convergent data structures allows computational graph to be arbitrarily distributed

- Divergence
  Divergence is a factor of synchronization period, concurrency, and throughput rate

Programming **Weak Synchronization** Models

Christopher S. Meiklejohn
Université catholique de Louvain, Belgium
Instituto Superior Técnico, Portugal

Programming **Weak Synchronization** Models

Christopher S. Meiklejohn
Université catholique de Louvain, Belgium
Instituto Superior Técnico, Portugal

UCL Université catholique de Louvain

TÉCNICO LISBOA

LIGHTKONE

SYNCFREE

Distributed Runtime
Anabranch

work-in-progress

Anabranch

- Layered approach
  Cluster membership and state dissemination for large clusters

- Delta-state synchronization
  Efficient incremental state dissemination and anti-entropy mechanism [Almeida et al. 2016]

- Epsilon-invariants
  Lower-bound invariants, configurable at runtime

- Scalable
  Demonstrated high scalability in production Cloud environments

Anabranch

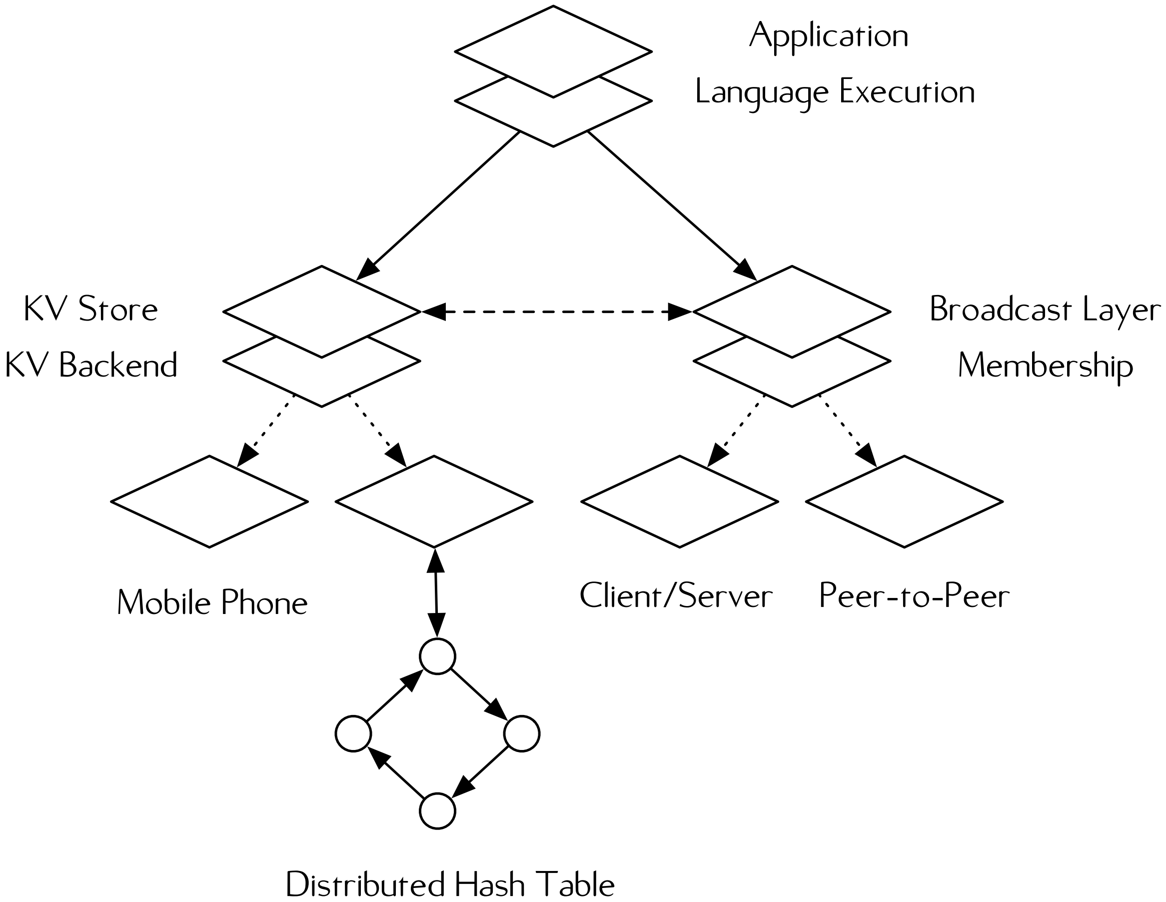
Layered Approach

Layered Approach

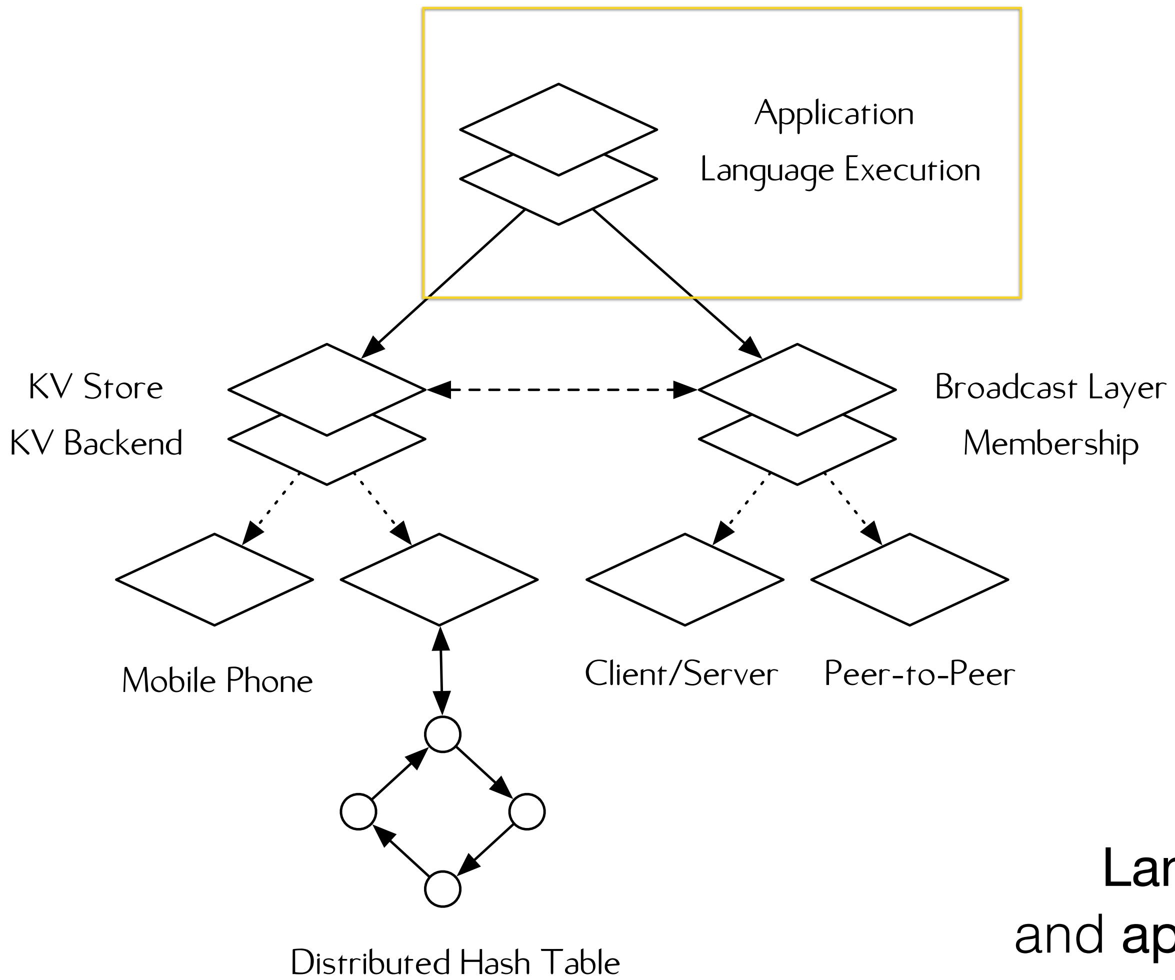
- Backend
  Configurable persistence layer depending on application.

- Membership
  Configurable membership protocol which can operate in a client-server or peer-to-peer mode [Leitao et al. 2007]

- Broadcast (via Gossip, Tree, etc.)
  Efficient dissemination of both program state and application state via gossip, broadcast tree, or hybrid mode [Leitao et al. 2007]

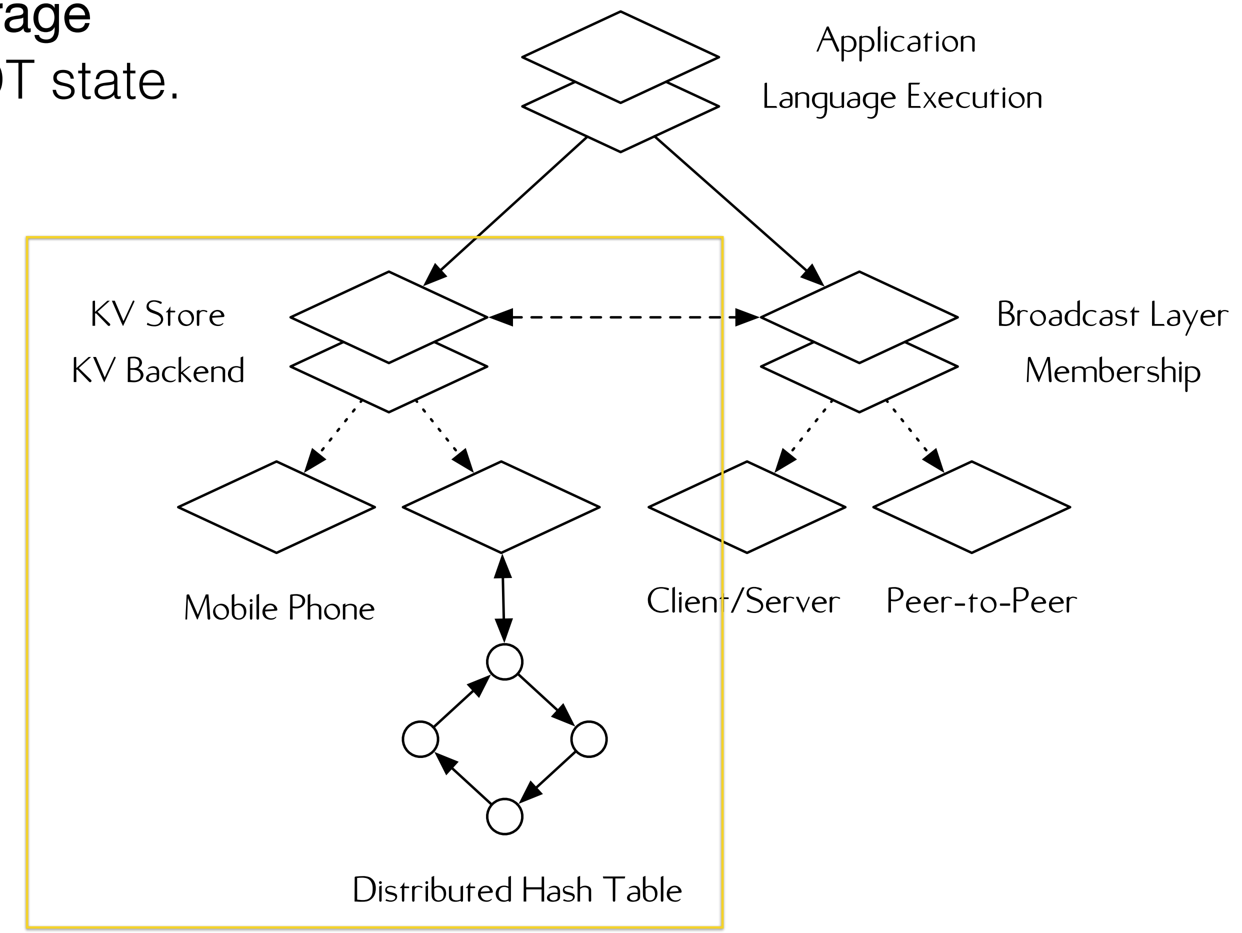


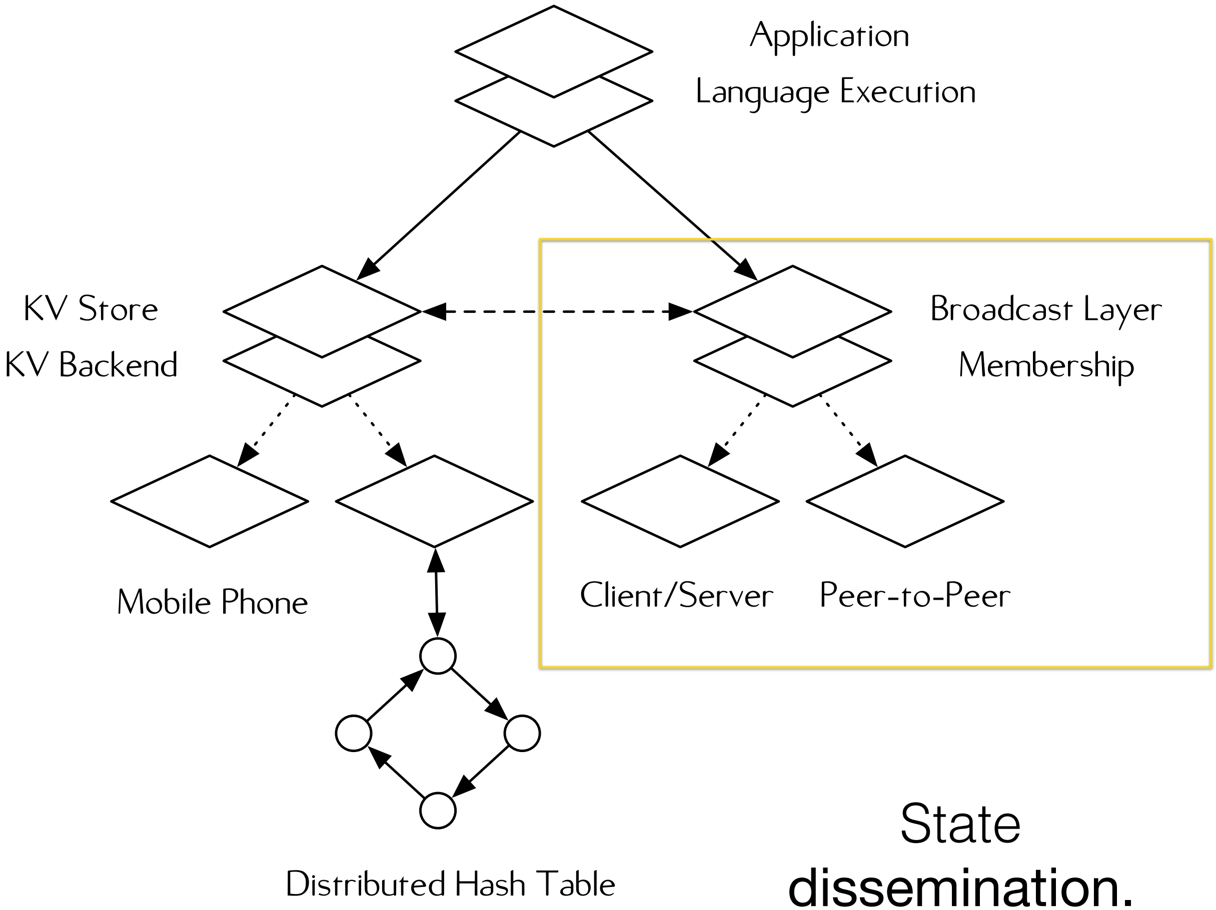
Application
Language Execution
KV Store
KV Backend
Broadcast Layer
Membership
Mobile Phone
Client/Server
Peer-to-Peer
Distributed Hash Table



Language
and applications.

Storage
for CRDT state.





State
dissemination.

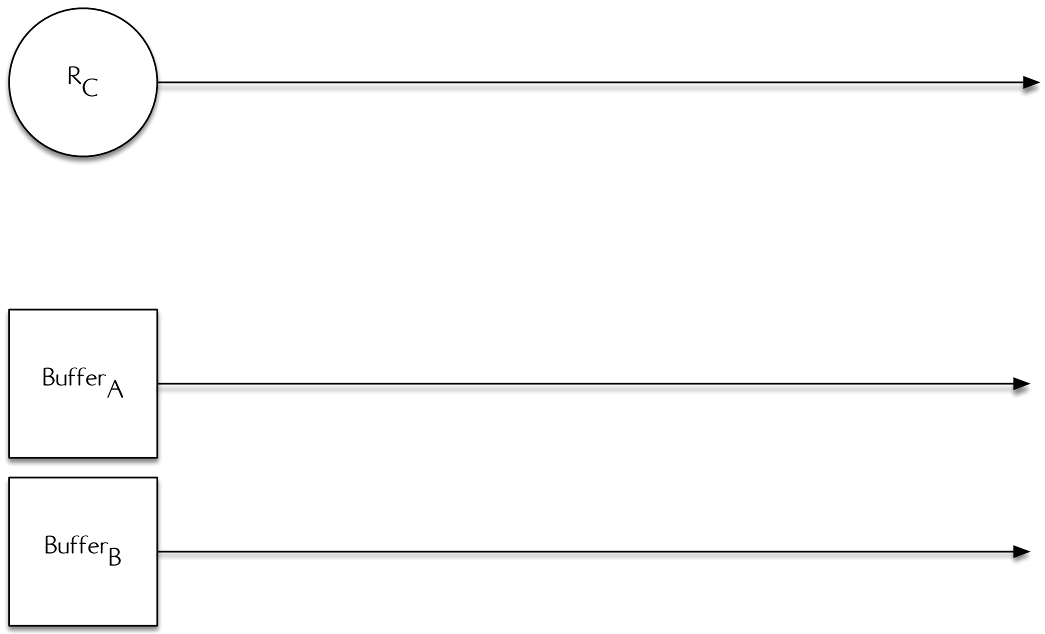
Anabranch
Delta-state CRDTs

Delta-based Dissemination

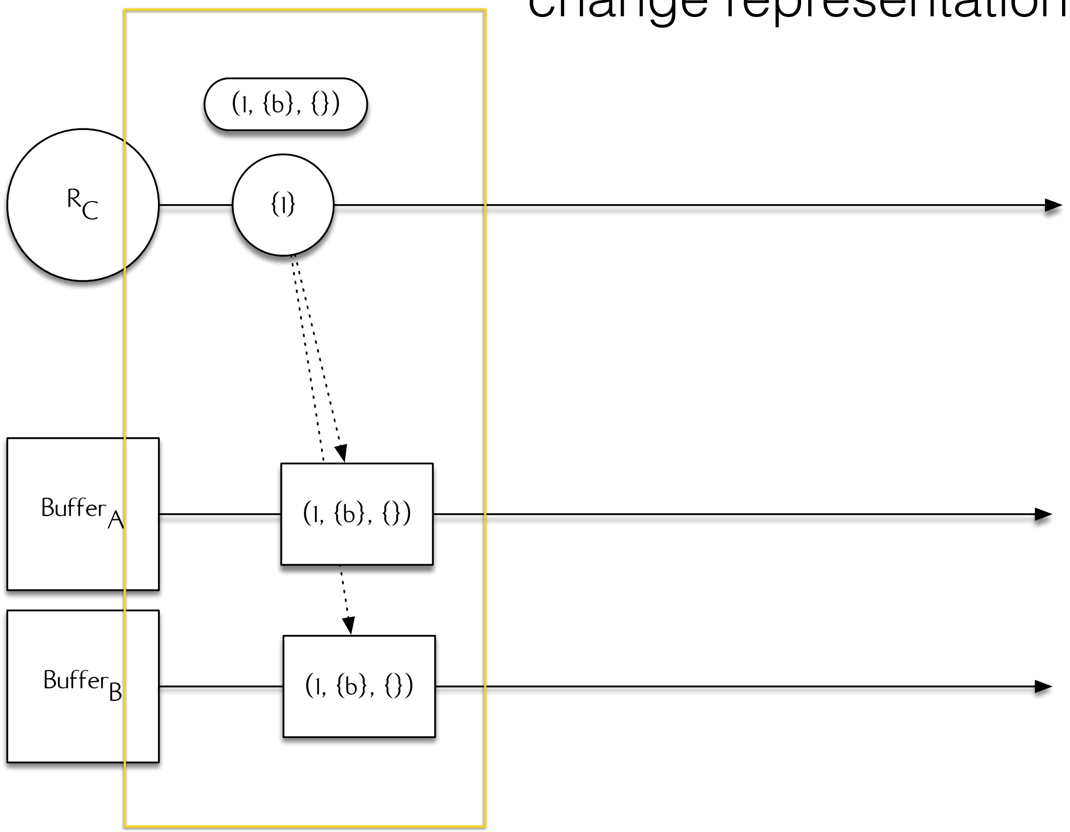
- Delta-state based CRDTs
  Reduces state transmission for clients

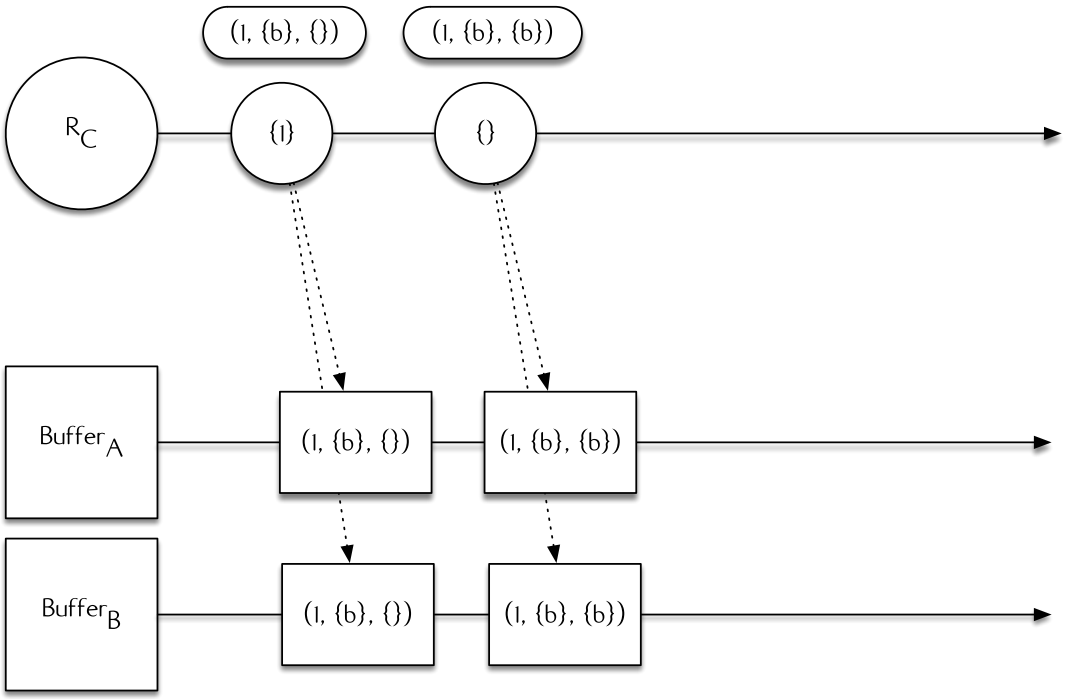
- Operate locally
  Objects are mutated locally; delta's buffered locally and periodically gossiped

- Only fixed number of clients
  Clients resort to full state synchronization when they've been partitioned too long

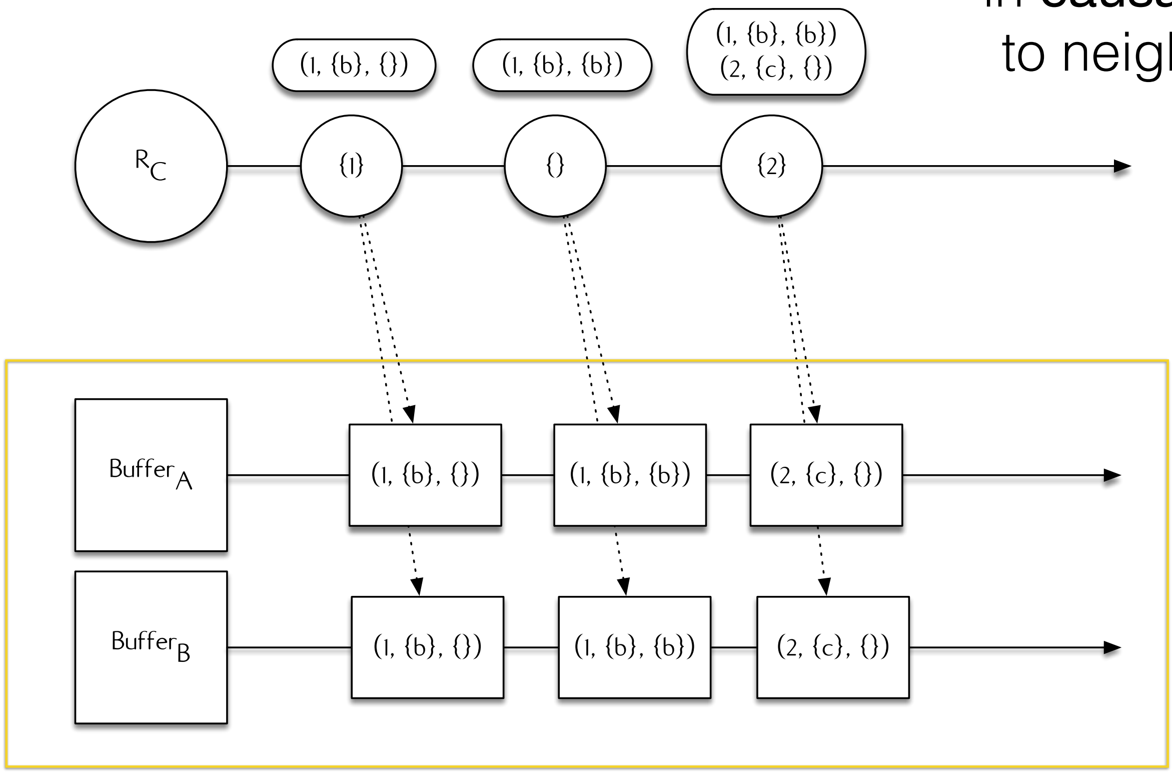

R_C
$Buffer_A$
$Buffer_B$

Buffer **minimal**
change representation…

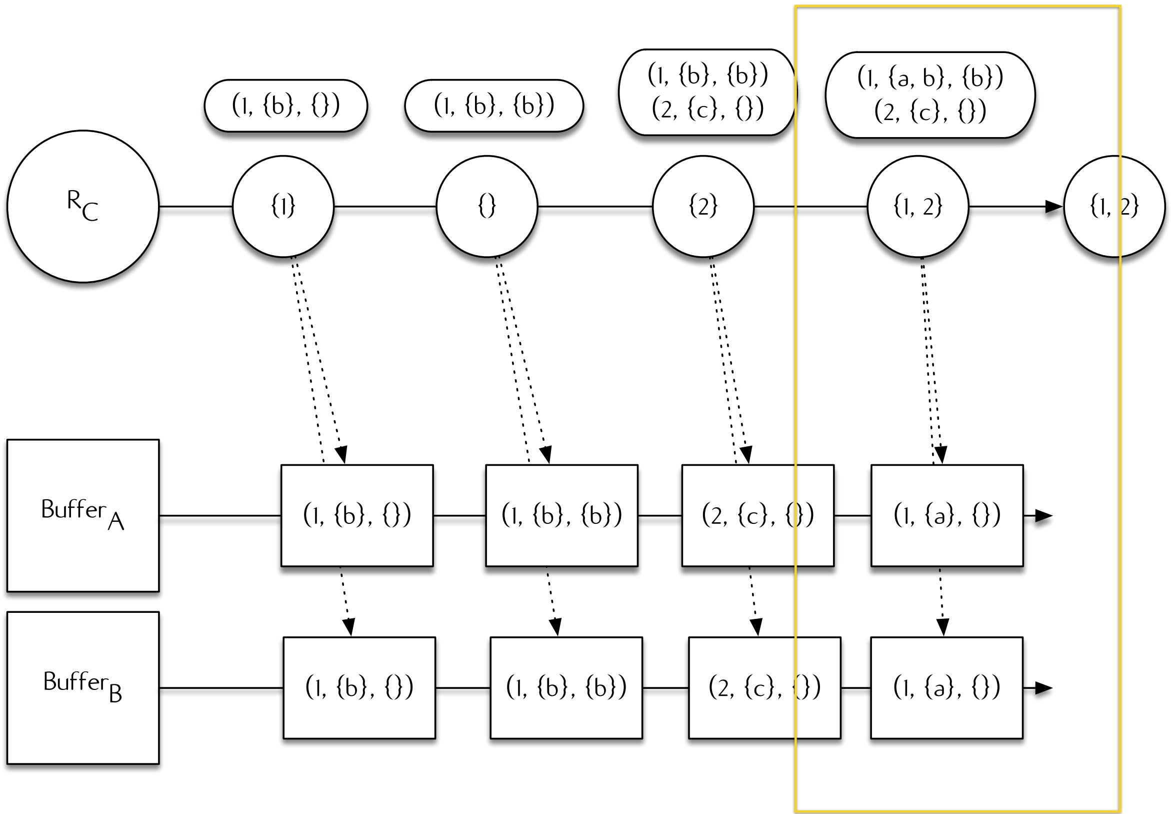



(1, {b}, {})
(1, {b}, {b})
R_C
{1}
{}
Buffer_A
(1, {b}, {})
(1, {b}, {b})
Buffer_B
(1, {b}, {})
(1, {b}, {b})

…then, disseminate state in **causal order** to neighbors.



Only ship **inflation** from incoming state.



Anabranch

Scalability

Scalability

- 1024+ nodes
  Demonstrated scalability to 1024 nodes in Amazon cloud computing environment

- Modular approach
  Many of the components built and can be operated outside of Lasp to improve scalability of Erlang

- Automated and repeatable
  Fully automated deployment, scenario execution, log aggregation and archival of experimental results

Just-right consistency: Antidote

Guest lecture

Peter Van Roy
Christopher Meiklejohn
Annette Bieniusa

LIGHTKONE
Lightweight computation for networks at the edge

Outline

Part I: Consistency in geo-replicated data stores

Part II: Consistency and invariant preservation

Part III: Antidote

LIGHTKONE
Lightweight computation for networks at the edge

Part I

Consistency in geo-replicated data stores

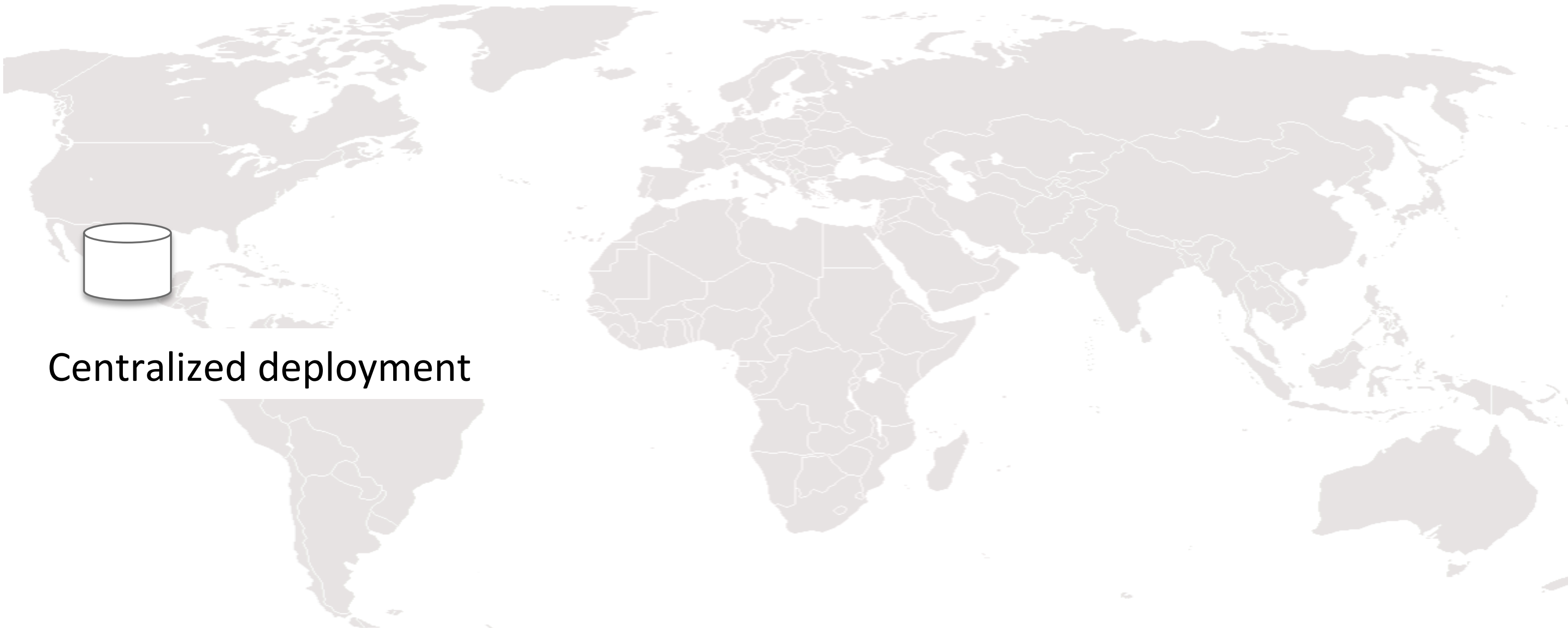
LIGHTKONE
Lightweight computation for networks at the edge

Interactive distributed applications

- Shared mutable data
- High-availability expected
- Low latency is business critical

LIGHTKONE
Lightweight computation for networks at the edge

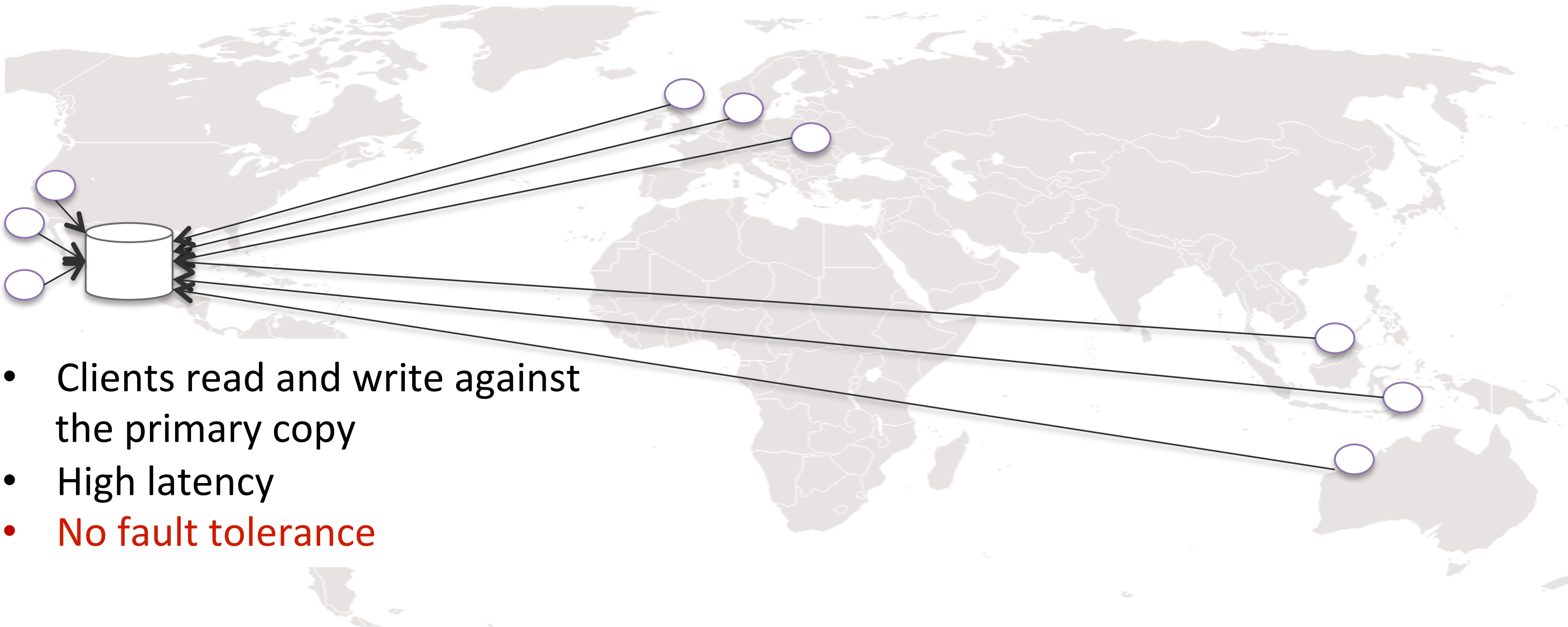
Cloud Databases



LIGHTKONE

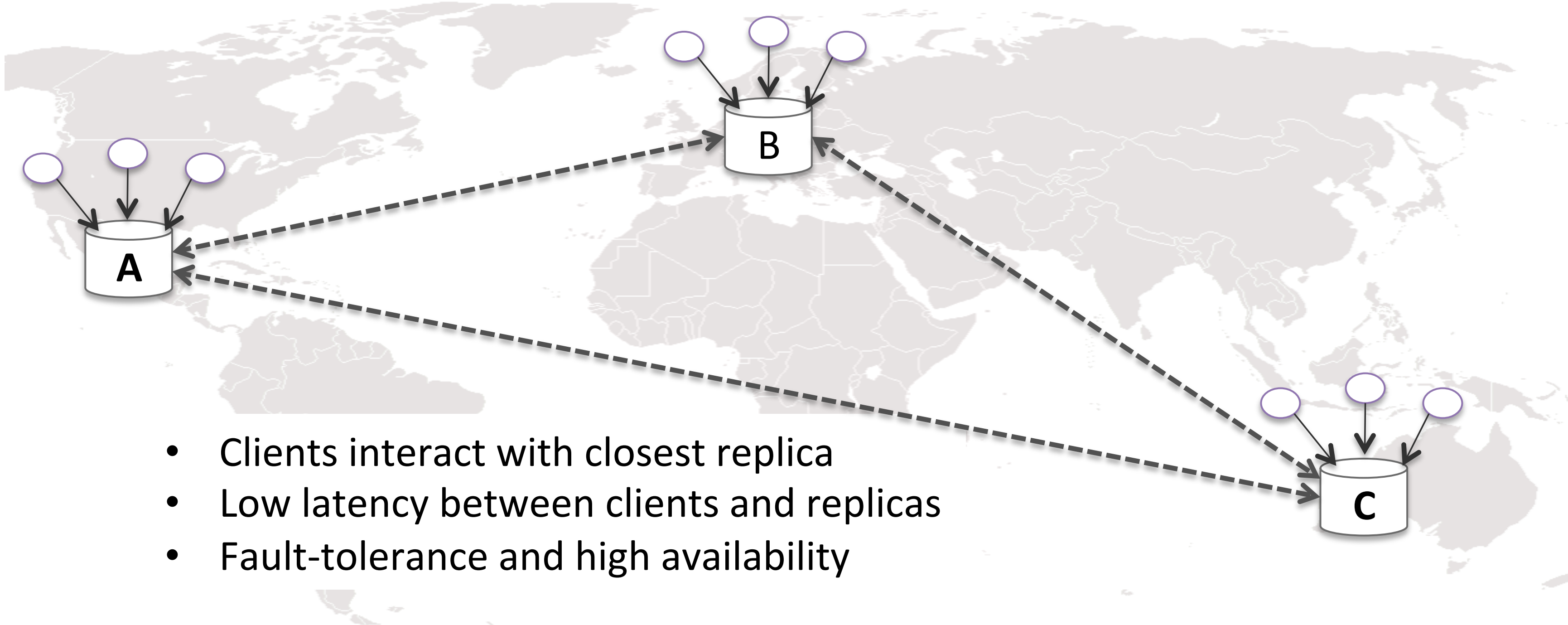
Lightweight computation for networks at the edge

Cloud Databases: Centralized deployment



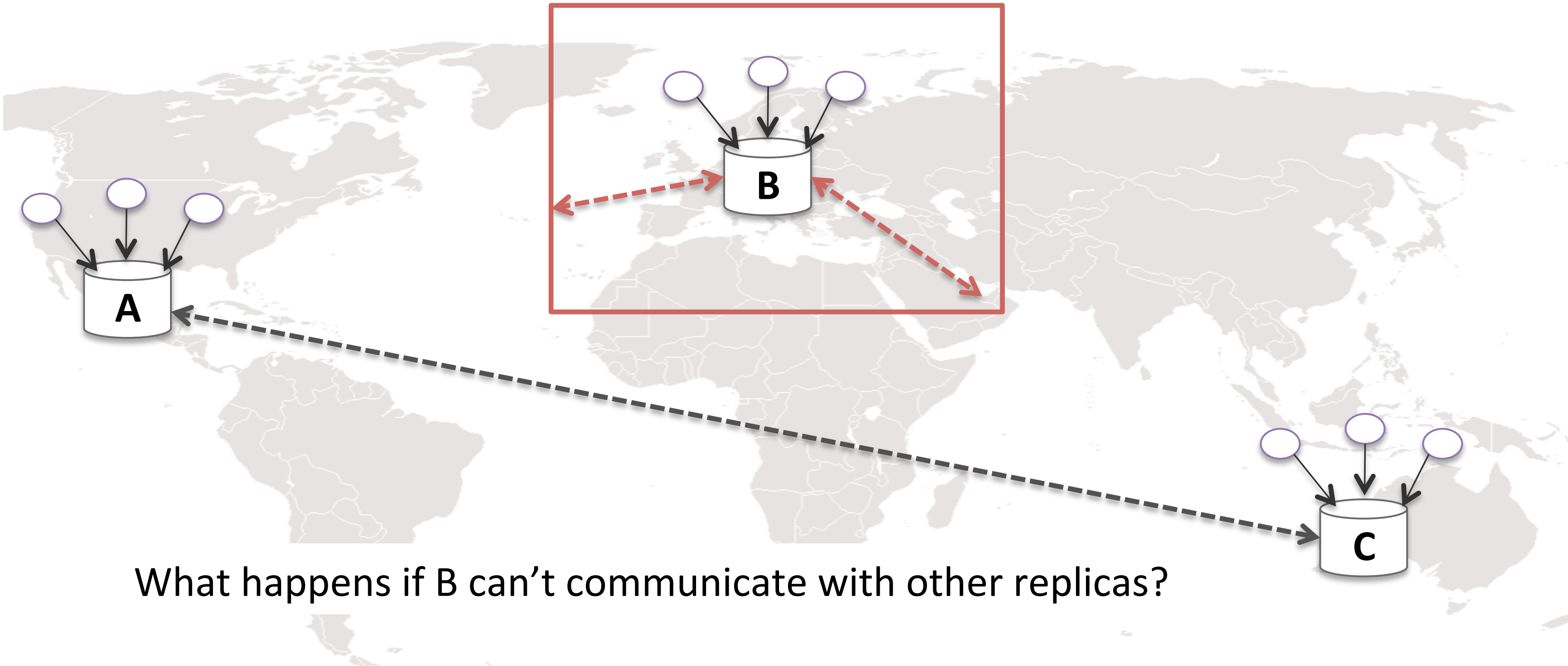
- Clients read and write against the primary copy
- High latency
- No fault tolerance

Cloud Databases: Geo-replication



- Clients interact with closest replica
- Low latency between clients and replicas
- Fault-tolerance and high availability

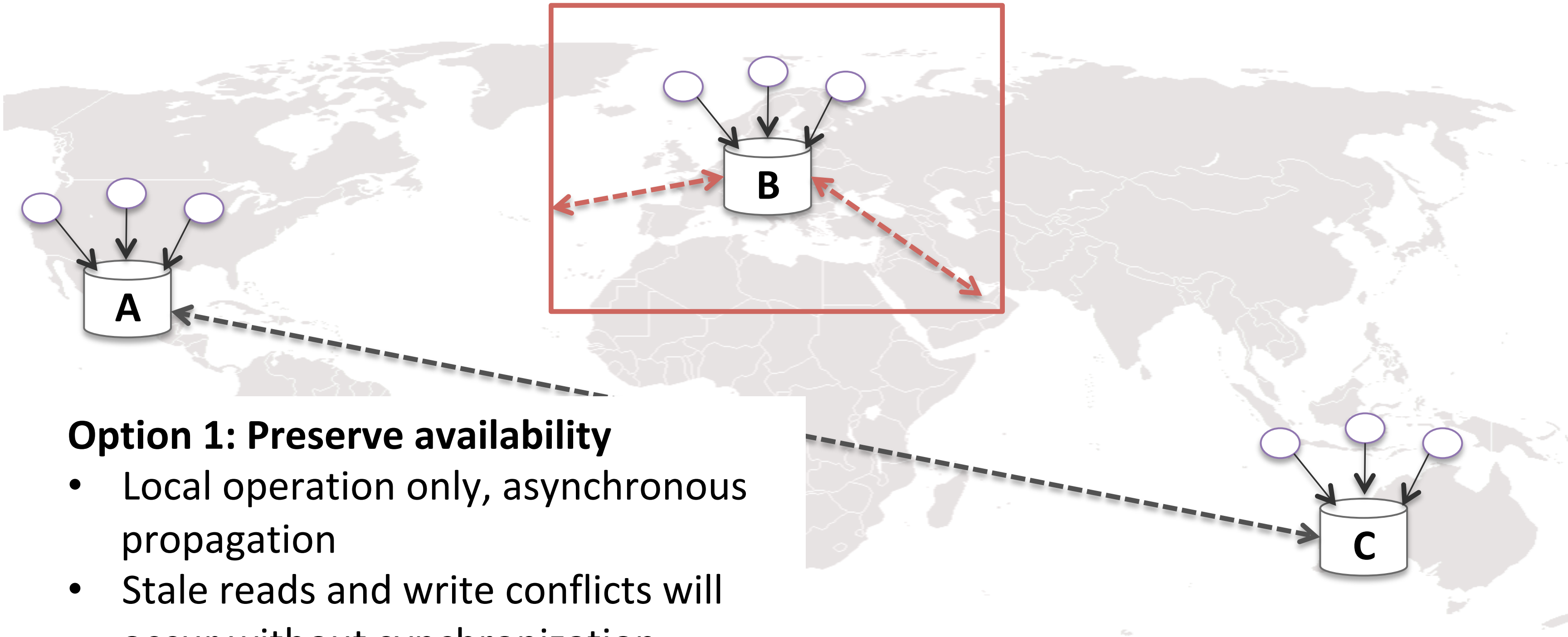
Cloud Databases



What happens if B can't communicate with other replicas?

LIGHTKONE
Lightweight computation for networks at the edge

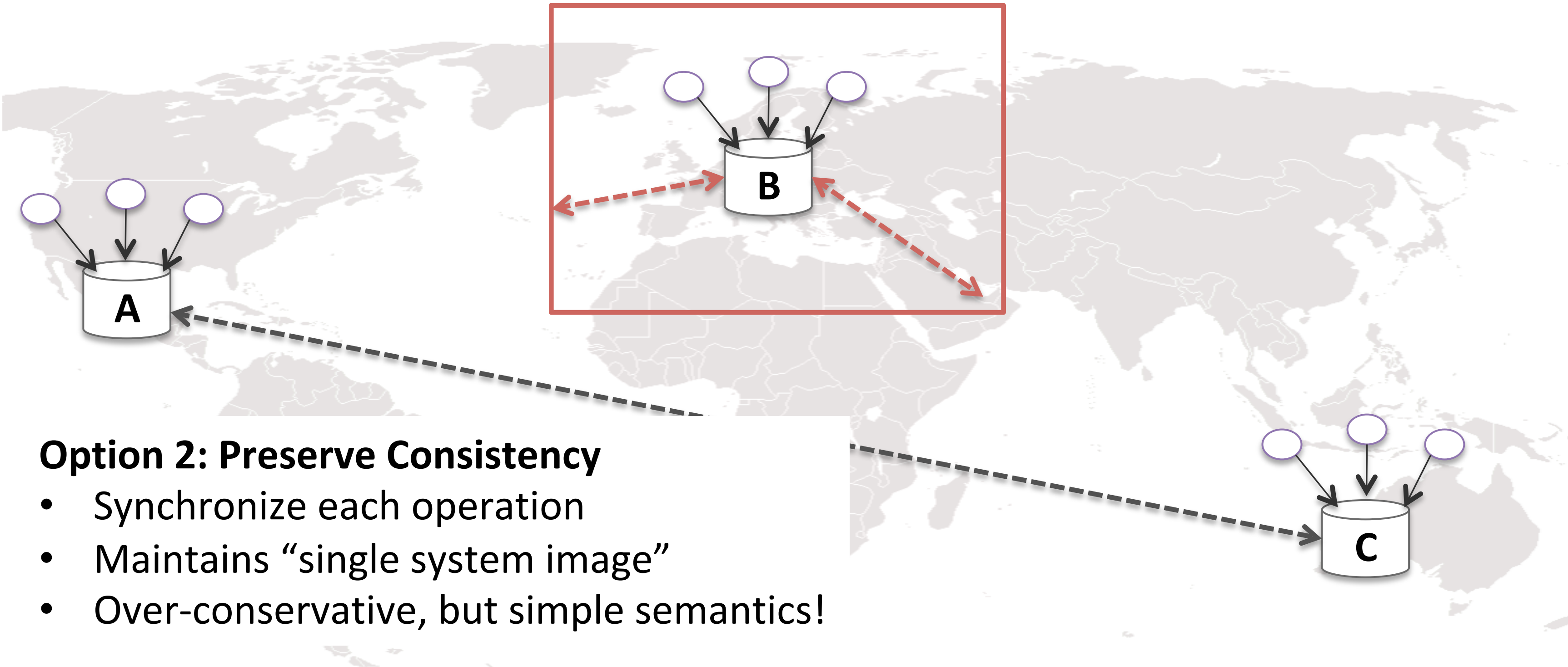
Cloud Databases



Option 1: Preserve availability

- Local operation only, asynchronous propagation
- Stale reads and write conflicts will occur without synchronization

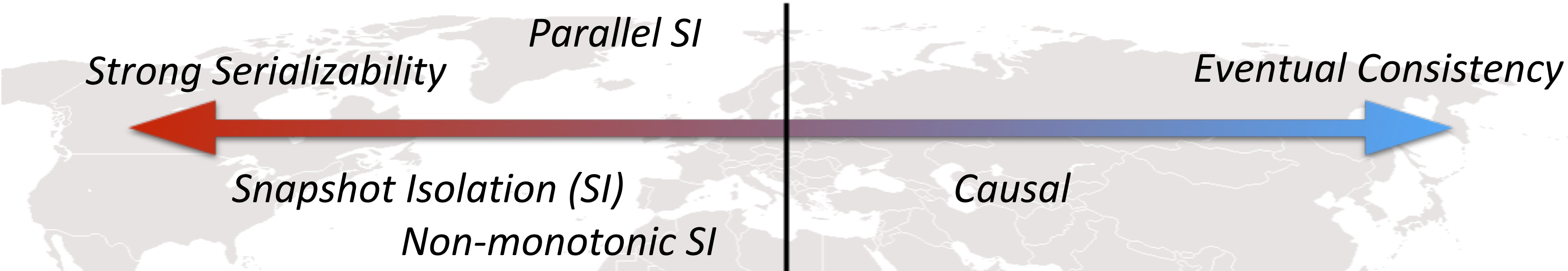
Cloud Databases



Option 2: Preserve Consistency

- Synchronize each operation
- Maintains “single system image”
- Over-conservative, but simple semantics!

Consistency-Availability trade-off



CP: Consistent-under-Partition

- Cross-replica consistency
- High latency and less availability, especially under faults

AP: Available-under-Partition

- Allows (temporary) state divergence
- Low latency, high availability
- Weaker guarantees

CAP

LIGHTKONE
Lightweight computation for networks at the edge

Cloud storage: AP systems

- To achieve low latency, high availability and throughput, systems have to forego strong consistency
  - Complex semantics
  - Low-level programming interface
    - Key-value map
  - No transactional support
  - No relational mappings

Cloud storage: CP Systems

- Cloud provides rely on expensive infrastructure to provide more guarantees

Google Cloud Platform

Azure Cosmos DB

  - Strong consistency
  - Support for transactions and SQL queries
  - Coordination across sites
    - ... still high latency

Alternative: AntidoteDB

- **AP** data store
- Provides strongest form of consistency that is highly available
- Use coordination only if its unavoidable
  - Allows for Just-right-consistency
- Supports programmer with comprehensive interface
  - Abstract data-types (CRDTs) and transactions

Conclusion: Part I

- Choice of consistency has consequences for system availability
- **CP systems** provide strong consistency, but require expensive infrastructure to provide high availability and introduce higher latencies
- **AP systems** opt for high availability, but provide weaker consistency guarantees and increase complexity in data management

LIGHTKONE
Lightweight computation for networks at the edge

Part II

Consistency and
Invariant Preservation

LIGHTKONE
Lightweight computation for networks at the edge

Which consistency does my application need?

- Many applications have constraints defined on the data that might not hold when operations execute concurrently.
- No “one-size-fits-all” consistency model
  - Choosing either model will either be over-conservative or risk anomalies
- Idea: Tailor consistency choices based on application-level invariants for each operation
- AP-compatible invariants
  - Invariants that only require “one way” communications
- CAP-sensitive invariants
  - Involve operations that require coordination
  - “two way” communication invariants

AP-compatible invariants

- No synchronization
  – Updates occur locally without blocking
- Asynchronous operation
  – Updates are fast, available, and exploit concurrency
- CRDTs are AP-compatible data model
- Compatible invariants
  – Relative order and joint update invariants can be preserved

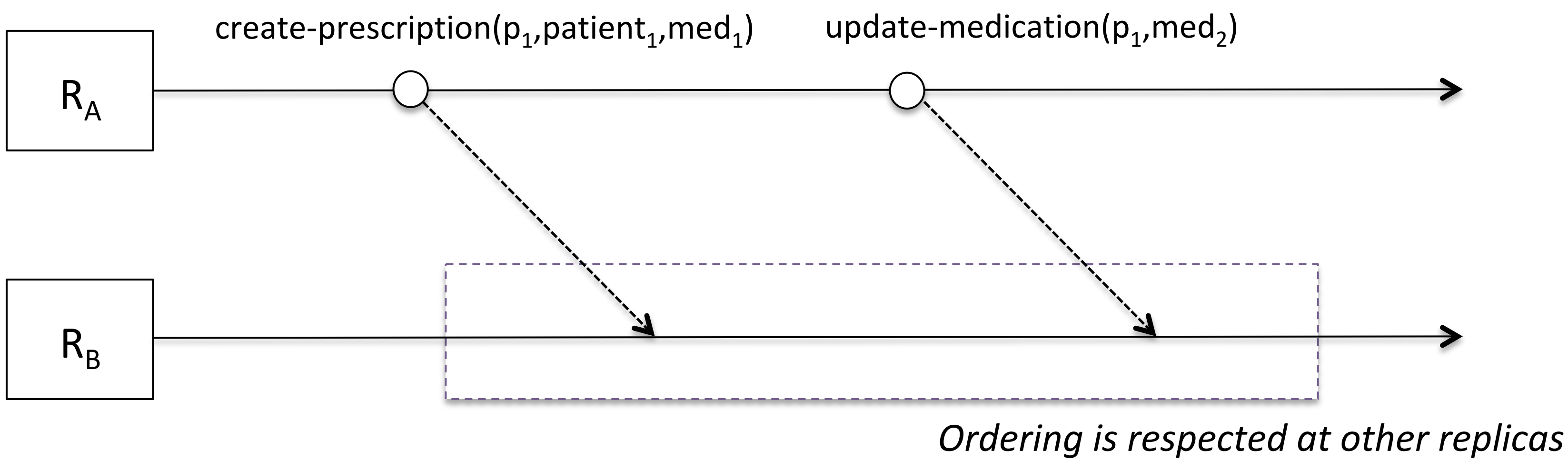
Use Case: FMK

- Fælles Medicinkort: Prescription management in the Danish national healthcare system

  - **create-prescription**
    Create prescription for patient, doctor, pharmacy
  - **update-medication**
    Add or increase medication to prescription
  - **process-prescription**
    Deliver a medication by a pharmacy
  - **get-*-prescriptions**
    Query functions to return information about prescriptions

LIGHTKONE
Lightweight computation for networks at the edge

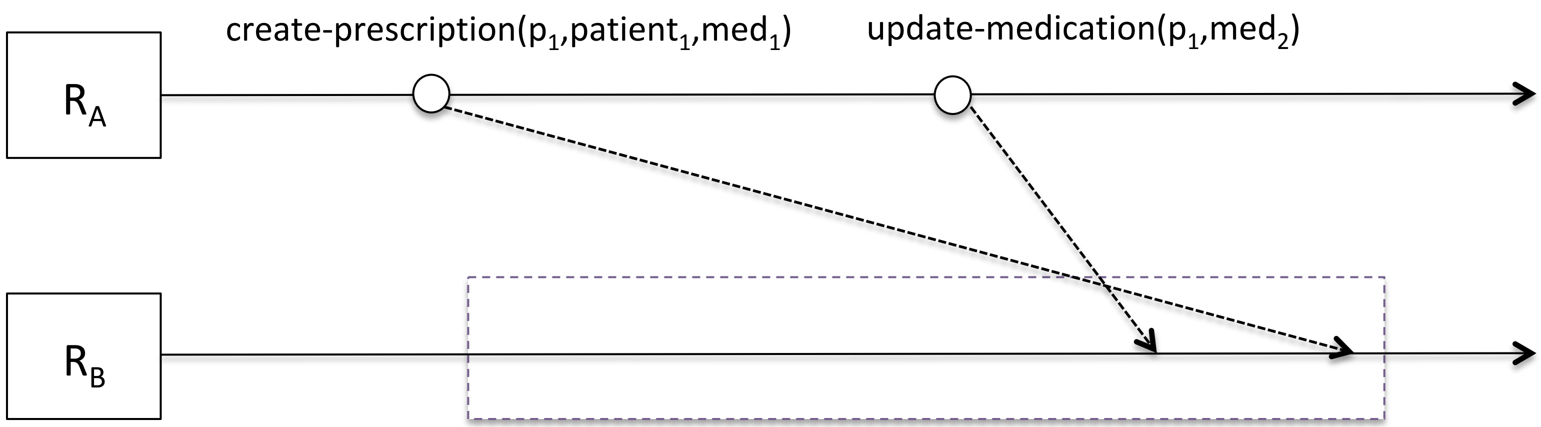
AP-compatible: Relative order



- Maintain program order implication invariant:
  “Only if prescription exists, medication can be adapted”
- Transmit in the right order!

LIGHTKONE
Lightweight computation for networks at the edge

AP-compatible: Relative order



Out of order propagation violates invariant!

- Maintain program order implication invariant:
  "Only if prescription exists, medication can be adapted"
- Transmit in the right order!

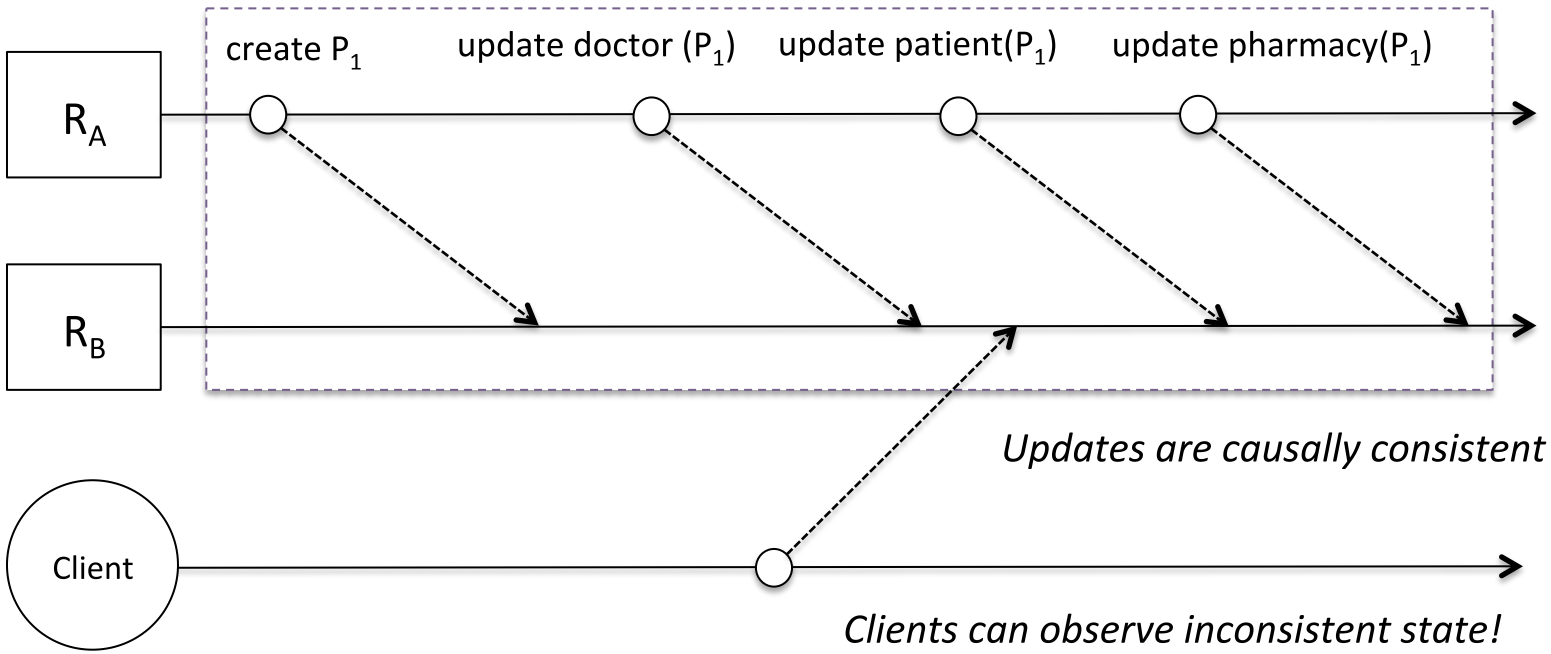
LIGHTKONE
Lightweight computation for networks at the edge

Causal consistency

- No ordering anomalies: $u \longrightarrow v \wedge visible(v) \Longrightarrow visible(u)$
- Respect causality
  - Ensure updates are delivered in the causal order [Lamport 78]
- Strongest available model
  - Always able to return some compatible version for an data object
- Referential integrity
  - Causal consistency is sufficient for providing referential integrity in an AP database

LIGHTKONE
Lightweight computation for networks at the edge

AP-compatible: Joint updates

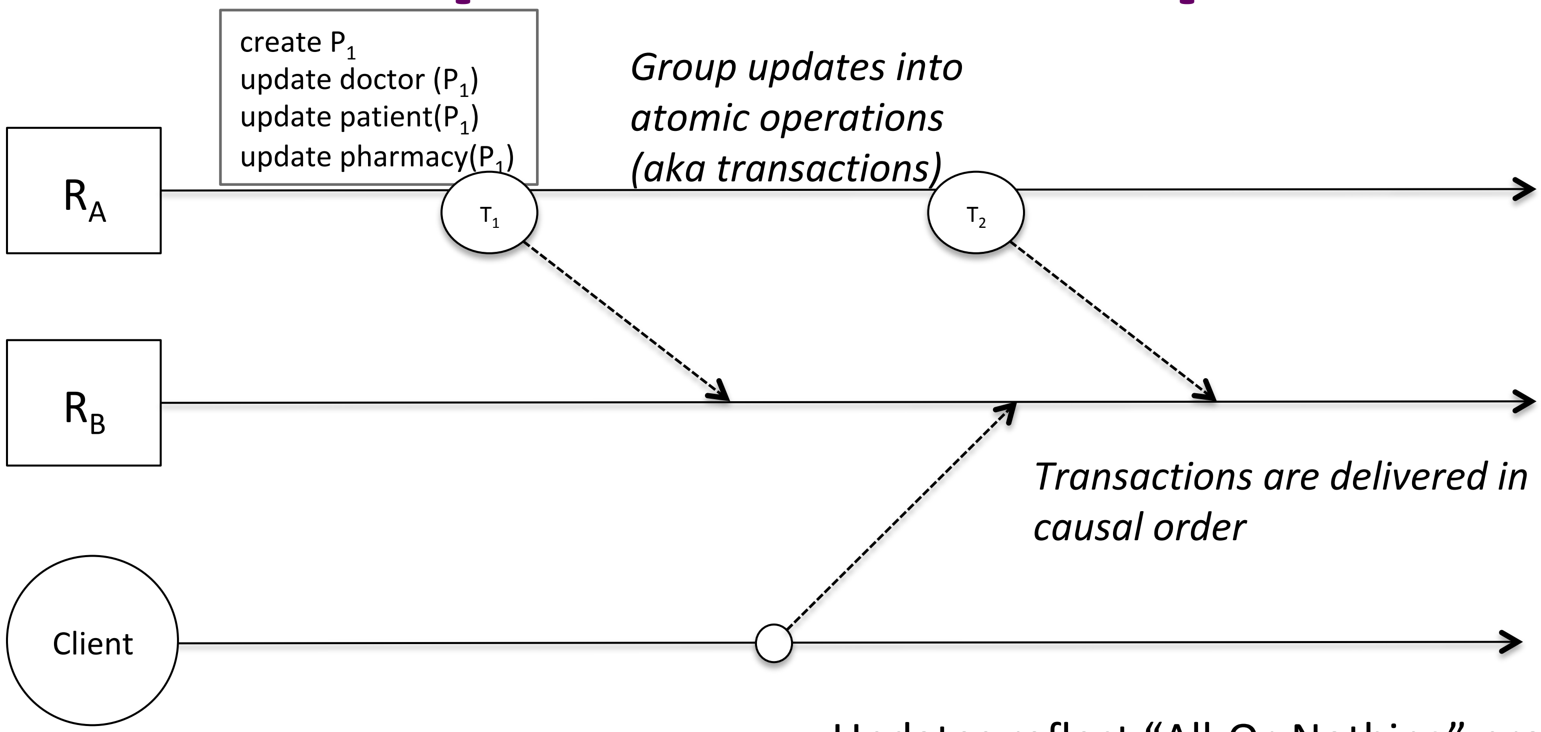


Updates are causally consistent

Clients can observe inconsistent state!
Missing update to patient and pharmacy!

LIGHTKONE
Lightweight computation for networks at the edge

AP-compatible: Joint updates



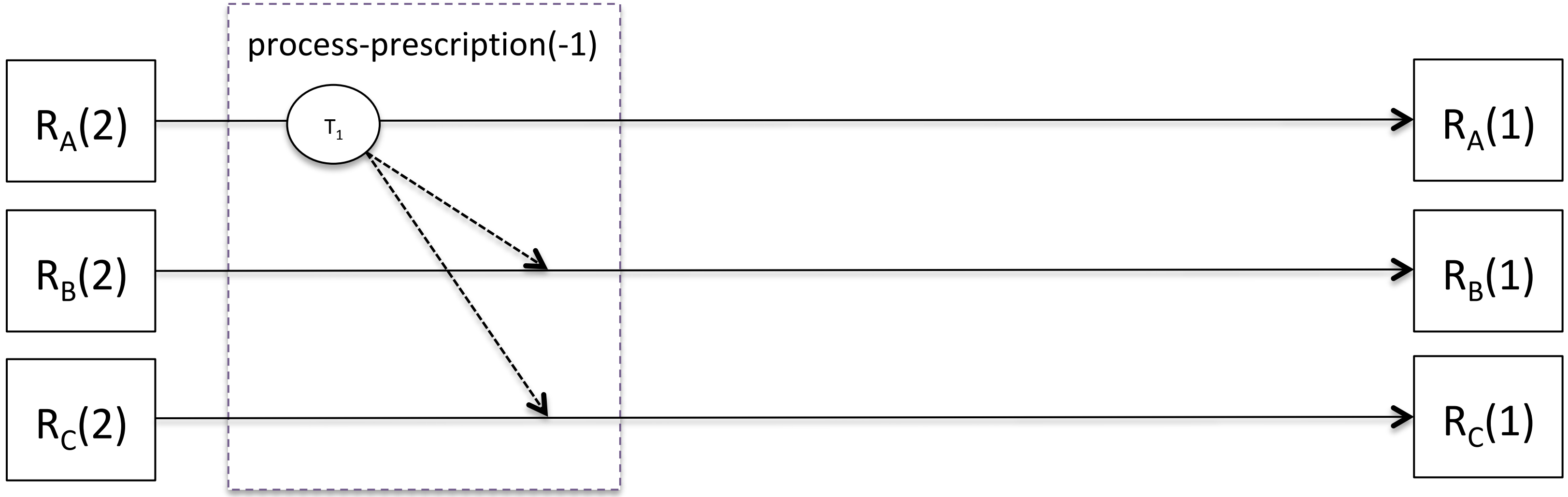
Updates reflect "All-Or-Nothing" property through snapshots.

Transactional Causal+ Consistency

- Causal consistency
- Transactional reads
  - Clients observe a consistent snapshot
- Transactional writes+
  - Updates become observable all-or-nothing
  - Concurrent updates converge to same value for all replicas

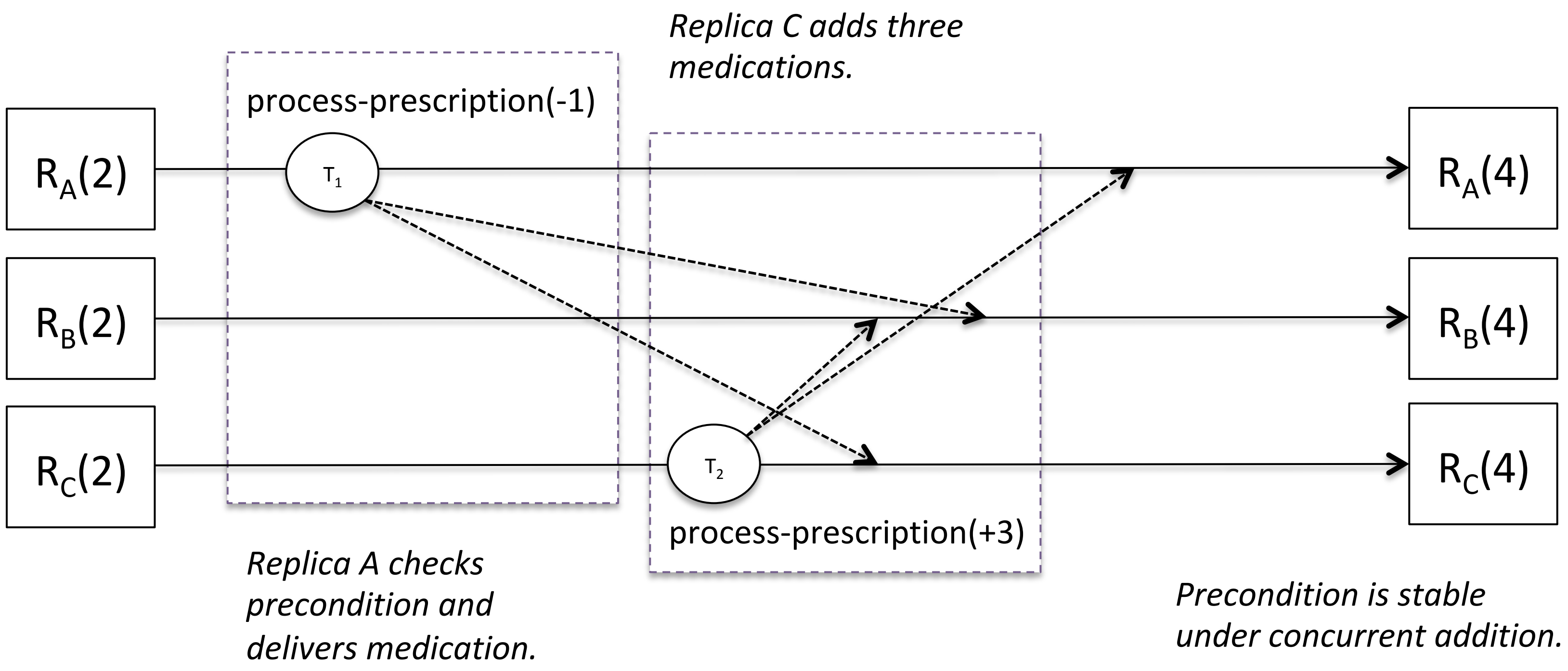
CAP-sensitive invariants

Do not over-deliver medication!



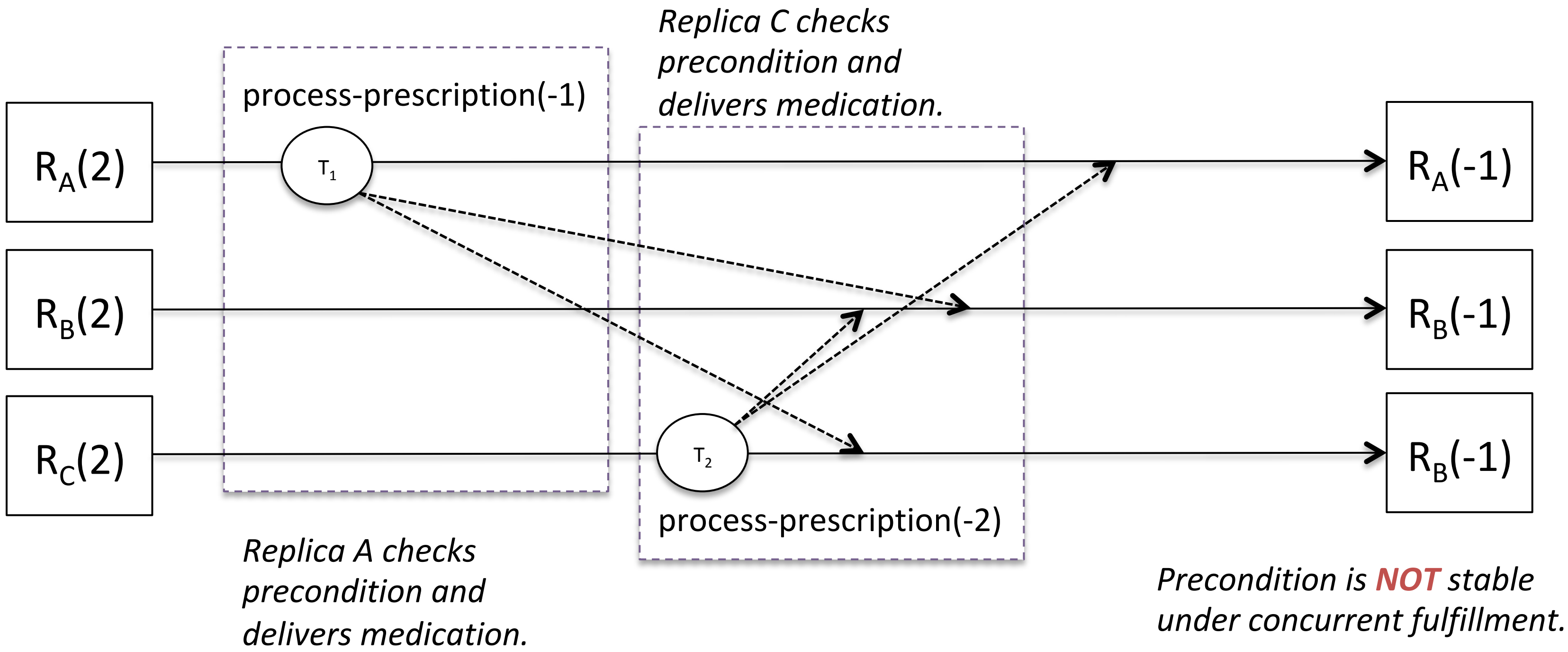
Replica A checks precondition and delivers medication.

CAP-sensitive invariants



LIGHTKONE
Lightweight computation for networks at the edge

CAP-sensitive invariants



LIGHTKONE
Lightweight computation for networks at the edge

Conclusion: Part II

- Strong consistency is often too conservative
  - Many operations are safe without cross-site coordination
- **Causal consistency** ensures that relative order invariants are preserved transparently
- **Transactional** causal consistency provides additionally atomicity and isolation
- What about operations that are really unsafe under weak consistency?
  - Require coordination (but only sporadically)

LIGHTKONE

Lightweight computation for networks at the edge

Part III

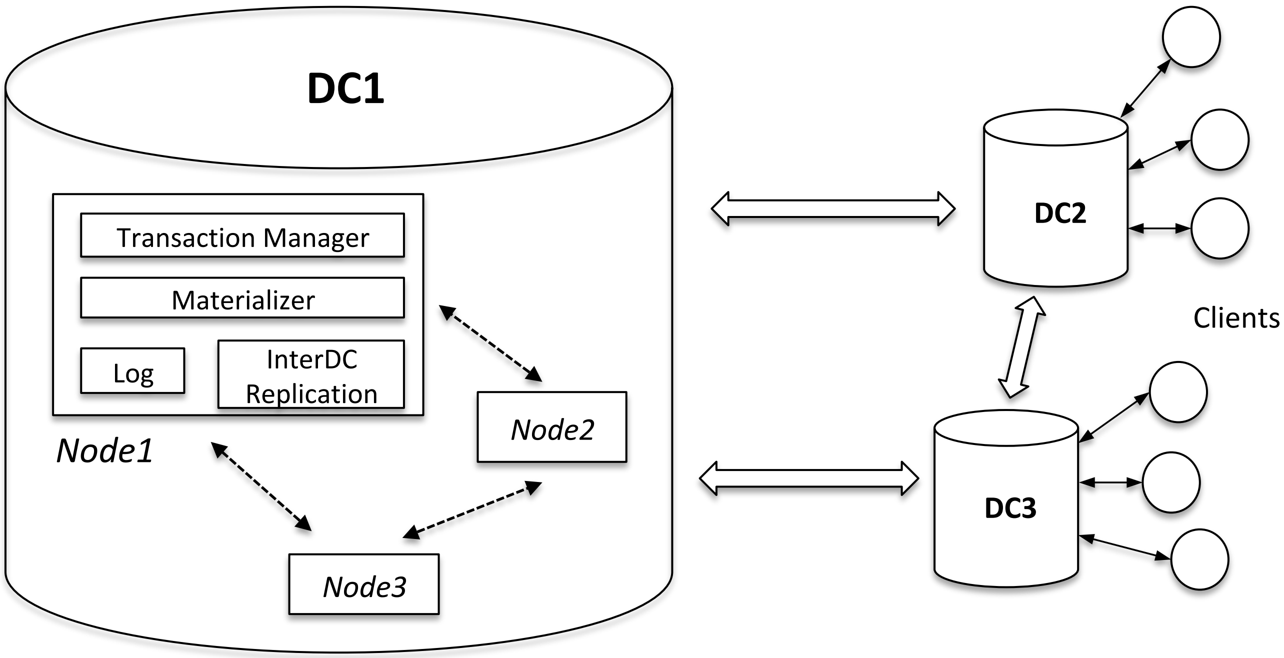
Antidote

Antidote

- **AP** data store for geo-replication in the cloud
- Provides strongest form of consistency that is highly available, namely transactional causal consistency (Cure protocol)
- Supports programmer with comprehensive interface
  - Abstract data-types (CRDTs), including maps, sets, sequences, counters
  - Transactions operate on a consistent snapshot
  - Atomic update (e.g., allows non-normalised data)
- Use coordination only if its unavoidable (bounded counters)
  - Allows for Just-right-consistency

LIGHTKONE
Lightweight computation for networks at the edge

Architecture



Object API

```
let connection = connect(8087, "localhost")
```

Establish connection

```
connection.defaultBucket = "bucket1"
```

Select bucket

```
let s1 = connection.set("programmingLanguages")
await connection.update(s1.addAll(["Java","Erlang"]))
```

Create new CRDT and perform update

```
let res = await s1.read()
```

Read current value

LIGHTKONE

Lightweight computation for networks at the edge

Transaction API

```
let set = connection.set(”programmingLanguages")
{
	let tx = await connection.startTransaction()
	await tx.update(set.remove("Java"))
	await tx.update(set.add("Kotlin"))
	await tx.commit()
}{
	await connection.update([
	set.remove("Java"),
	set.add("Kotlin")])
}
```

Variant 1:
dynamic transaction

Variant 2:
static transaction /
batch updates

Conclusion: Part III

- Antidote provides Just-right consistency
  - Transactional Causal Consistency for AP-compatible invariants
  - Bounded Counters for CAP-sensitive invariants
- Supports programmer with rich interface
  - Transactions with snapshot reads and atomic updates
  - CRDTs avoid conflicting updates
- Documentation
  - http://antidotedb.org
- Code repository
  - https://github.com/SyncFree/antidote

LIGHTKONE
Lightweight computation for networks at the edge

Conclusion of the lesson

- We have now arrived at the end of this lesson on how to program a distributed system with weak synchronization
  - Synchronization: eventual node-to-node communication
  - Consistency model: strong eventual consistency
- We have shown three important applications of this idea
  - CRDTs, Lasp, and Antidote
- We are convinced that the approach has a promising future
  1. Edge computing
  2. Synchronization-free services

LIGHTKONE

Different consistency models

- Strong consistency: the system obeys linearizability
  - Easy to program but can be very inefficient
- Eventual consistency: the system can support many concurrent operations « in flight »
  - Efficient execution but hard to program because of potential conflicts
- Convergent consistency: the system can support many concurrent operations, plus it obeys strong eventual consistency
  - Both efficient execution and easy to program
  - We cannot do CAP but we can do AP + ◊C = available, partition-tolerant, and convergent

LIGHTKONE

1. Edge computing

- Distributed systems « at the edge » are omnipresent
  - Internet of Things and mobile devices far outnumber data center nodes
  - Edge networks are highly dynamic for computation and communication
- Synchronization-free programming is well-matched to edge systems
  - Convergent computation layer with a hybrid gossip communication layer
- It is naturally tolerant to faults in edge systems
  - Partitions
  - Message loss and reordering
  - Nodes going offline and online
  - Node crashes

Partitions; Message loss and reordering: Slows down convergence

Nodes going offline and online; Node crashes: Tolerant as long as state exists on at least one node

LIGHTKONE

2. Synchronization-free services (1)

Today

- We are using CRDTs as the basis for a programming framework and a transactional database
  - Lasp and Antidote

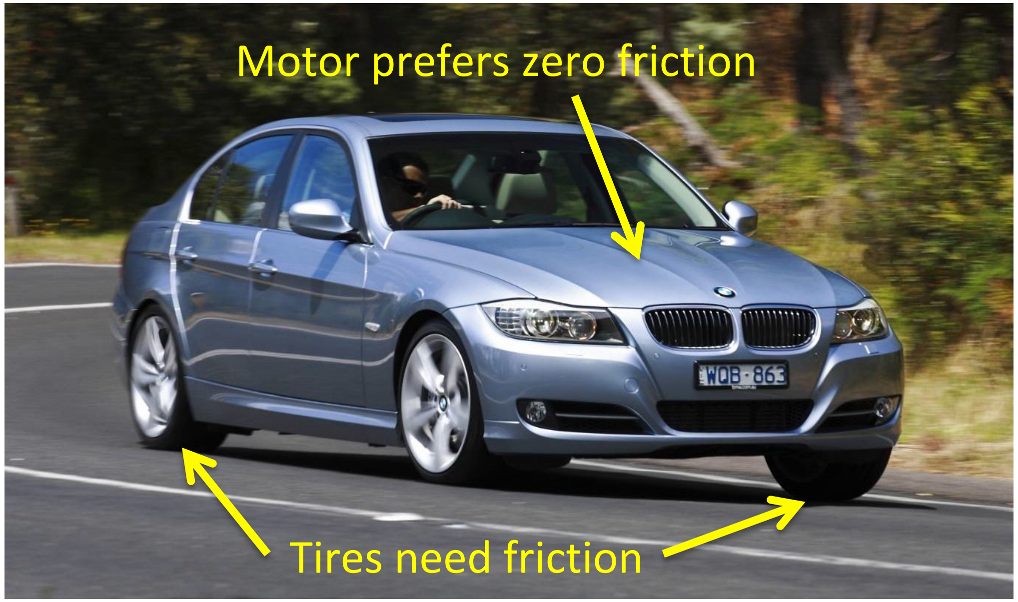
Future

- But the synchronization-free approach can be applied much more generally
  - Let me introduce this with a parable...

LIGHTKONE

Parable of the car (1)

Synchronization is like friction

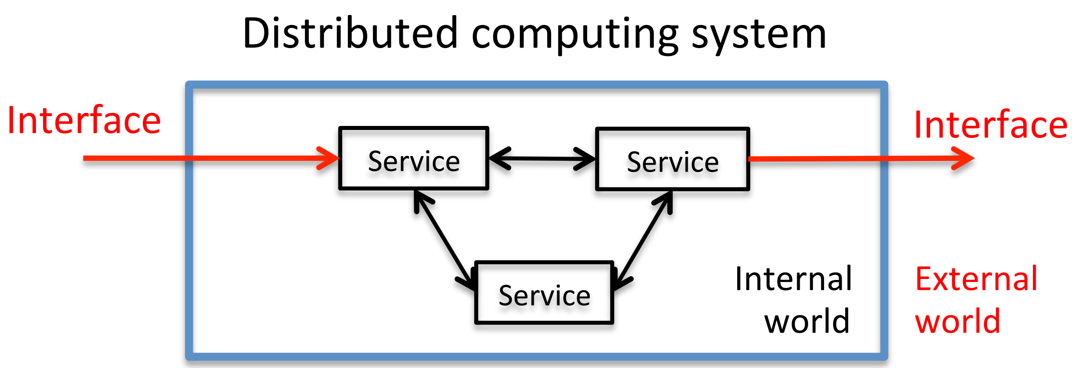


- Like friction, synchronization is both desirable and undesirable
- Consider a car on a highway
- The car needs friction: it moves because the tires grip the road
- But the car's motor avoids friction: the motor should be as frictionless as possible, otherwise it will heat up and wear out

LIGHTKONE

Parable of the car (2)

Consider a distributed computing system made of services connected together



- Synchronization is only needed at the system's interface with the external world
- Internally the services should avoid synchronization

Friction is only needed externally, so the tires can grip the road

Internally, the motor avoids friction

LIGHTKONE

Synchronization-free services (2)

- The system has a synchronization boundary
  - Inside this boundary, all services are synchronization-free
  - Synchronization is only needed at the boundary
- Services are inside this boundary
  - Internal state of each service obeys SEC
  - Service API has asynchronous streams, in and out

LIGHTKONE

Going forward!

- In this lesson have introduced the basic concepts of programming with weak synchronization
  - We presented data structures (CRDTs), a programming framework (Lasp), and a transactional database (Antidote)
- Our future work will focus on edge computing and synchronization-free services
  - LightKone H2020 project (lightkone.eu)
  - This project is working on both Lasp and Antidote

LIGHTKONE

Lasp and Antidote resources

- Documentation
  - https://lasp-lang.org
  - http://antidotedb.org
- Code repository
  - https://github.com/lasp-lang
  - https://github.com/SyncFree/antidote

Bibliography

- Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *Conflict-free replicated data types*. Technical Report RR-7687. INRIA (July 2011).
- Christopher Meiklejohn and Peter Van Roy. Lasp: A language for distributed, coordination-free programming. In *PPDP*. ACM, 184–195 (2015).
- SyncFree: Large-scale computation without synchronisation. European FP7 project, 2013–2016. syncfree.lip6.fr
- LightKone: Lightweight computation for networks at the edge. European H2020 project, 2017–2019. lightkone.eu
- Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. Cure: strong semantics meets high availability and low latency. In *ICDCS*. 405–414 (2016).
- Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: virtues and limitations. In *PVLDB 7(3)*. 181–192 (2013).

LIGHTKONE