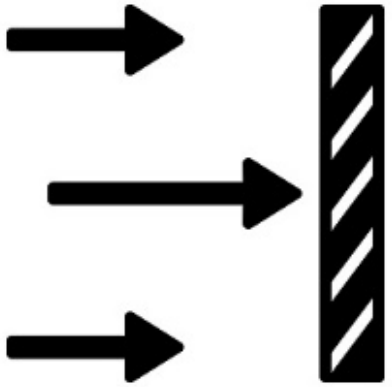
Pixabay.com

CRDT – CONFLICT-FREE REPLICATED DATA TYPES

CRDTs Units

- Eventual consistency, informally
- State-based objects
- Eventual consistency, more formally
- Conflict-free replicated data types



Pixabay.com

EVENTUAL CONSISTENCY, INFORMALLY

Eventual Consistency

- Eventual consistency is desirable for **large-scale distributed systems** where **high availability** is important

- Tends to be cheap to implement (e.g., via gossip) but may serve stale data

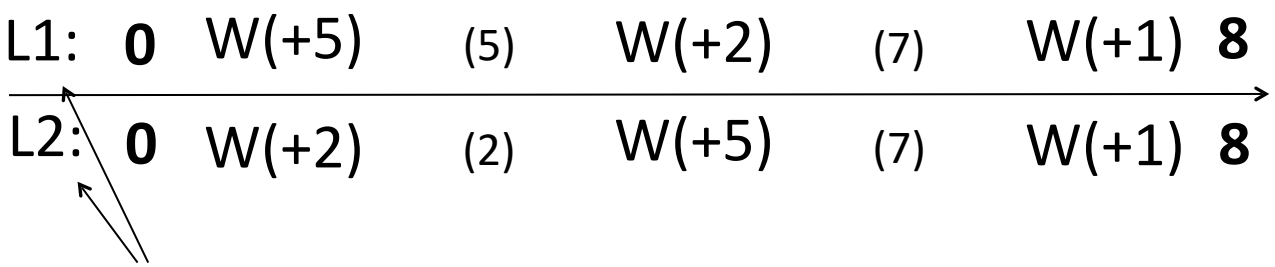
- Constitutes a **challenge** for environments where **stronger consistency is important**

Handling Concurrent Writes

- Premise for eventual consistency **were scenarios with few** (no) **concurrent writes** to the same key (cf. client-centric consistency)
- However, we do need a mechanism to **handle concurrent writes** should they so happen
- **If there were** a way to **handle concurrent writes**, we could support **eventual consistency** more broadly
- Would “only” need to guarantee that **after processing all writes for a key**, **all replicas converge**, **no matter what order** the writes are processed (e.g., assuming gossip)

Examples

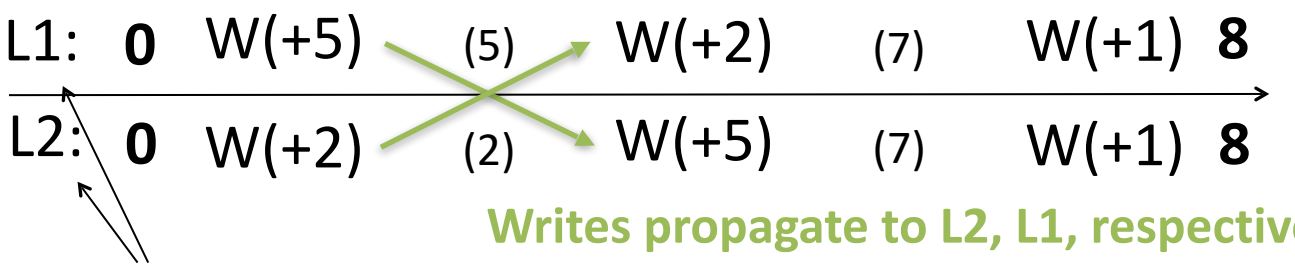
Growth-only counter (G-counter)



Different locations (replicas)

Examples

Growth-only counter (G-counter)

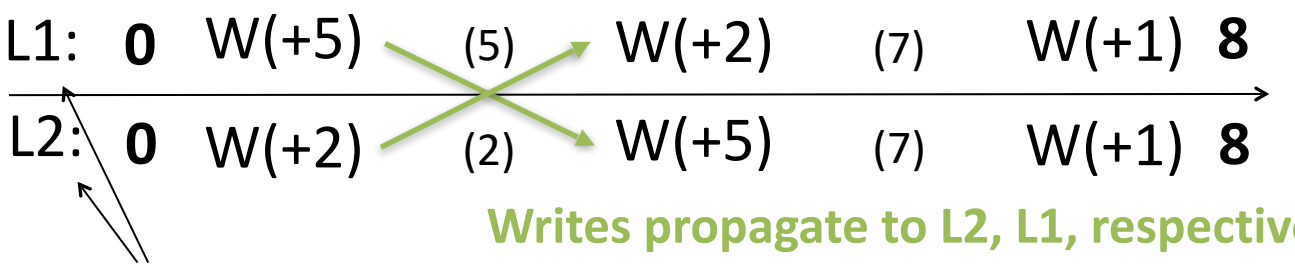


Writes propagate to L2, L1, respectively ✔

Different locations (replicas)

Examples

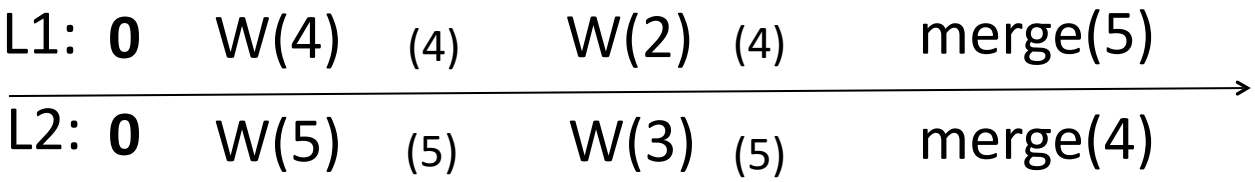
Growth-only counter (G-counter)



Writes propagate to L2, L1, respectively ✔

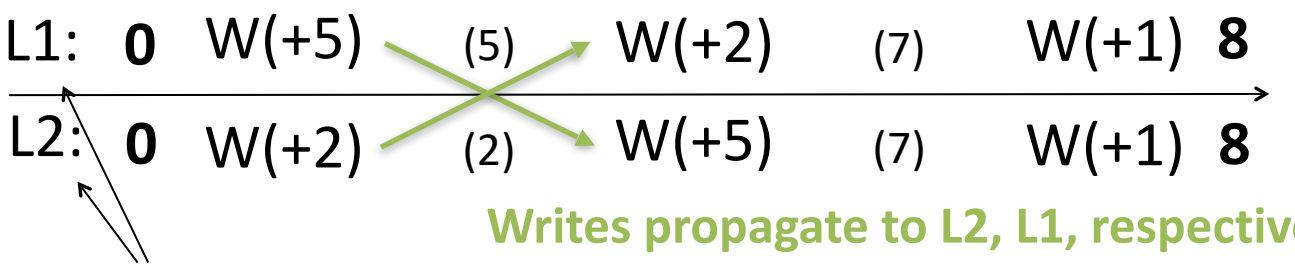
Different locations (replicas)

Max register



Examples

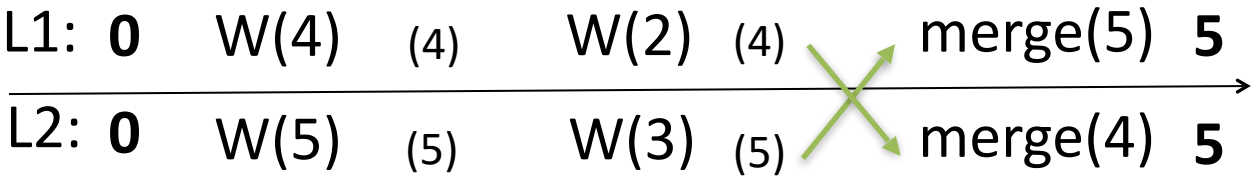
Growth-only counter (G-counter)



Writes propagate to L2, L1, respectively ✔

Different locations (replicas)

Max register



State propagate to L2, L1 via periodic merging ✔

Self-study Questions

- Think of a few basic data structures, like lists, sets, counters, binary trees, heaps, maps, etc., and visualize for yourself what happens if replicated instances of these structures are updated via gossip.
- Does their state converge, no matter the update sequence?
- What happens if update operations are lost or duplicated?
- What mechanisms we know other than gossip could be used to keep these replicated structures updated without violating their convergence.
- What are pros and cons of these mechanisms?

Pixabay.com

CRDT – FROM STATE-BASED OBJECTS TO REPLICATED STATE-BASED OBJECTS

State-based objects

Mostly plain old objects

- Offer update and query requests to clients
- Maintain internal state
- Process client requests
- Perform merge requests amongst each other
- Periodically merge (support infrastructure)

State-based Object

- What we commonly know as object

- Comprised of
  - Internal state
  - One or more `query` methods
  - One or more `update` methods
  - A `merge` method

Class Average

Running Example

```
class Avg(object):
def __init__(self):
  self.sum = 0
  self.cnt = 0

def query(self):
  if self.cnt != 0:
      return
        self.sum /
          self.cnt
  else:
    return 0
```

```
def update(self, x):
  self.sum += x
  self.cnt += 1

def merge(self, avg):
  self.sum += avg.sum
  self.cnt += avg.cnt
```

Average

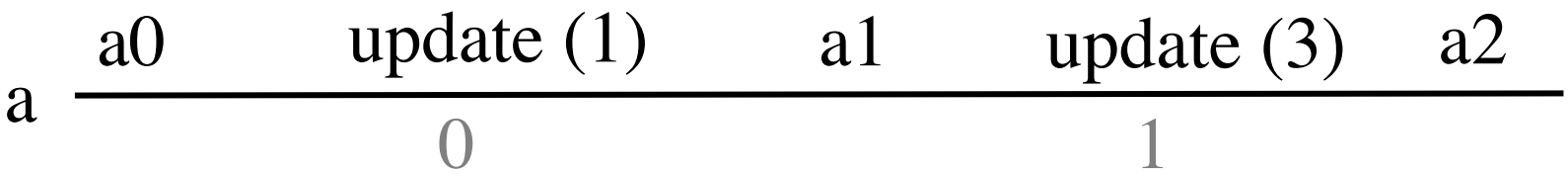
State-based object representing a running average

- Internal state
  - `self.sum` and `self.cnt`
- `Query` returns average
- `Update` updates average with a new value *x*
- `Merge` merges one `Avg` instance into another one

Replicated State-based Object

- State-based object replicated across multiple nodes
- E.g., replicate `Avg` across two nodes
- Both nodes have a copy of state-based object
- Clients send query and update to a single node
- Nodes periodically send their copy of state-based object to other nodes for merging

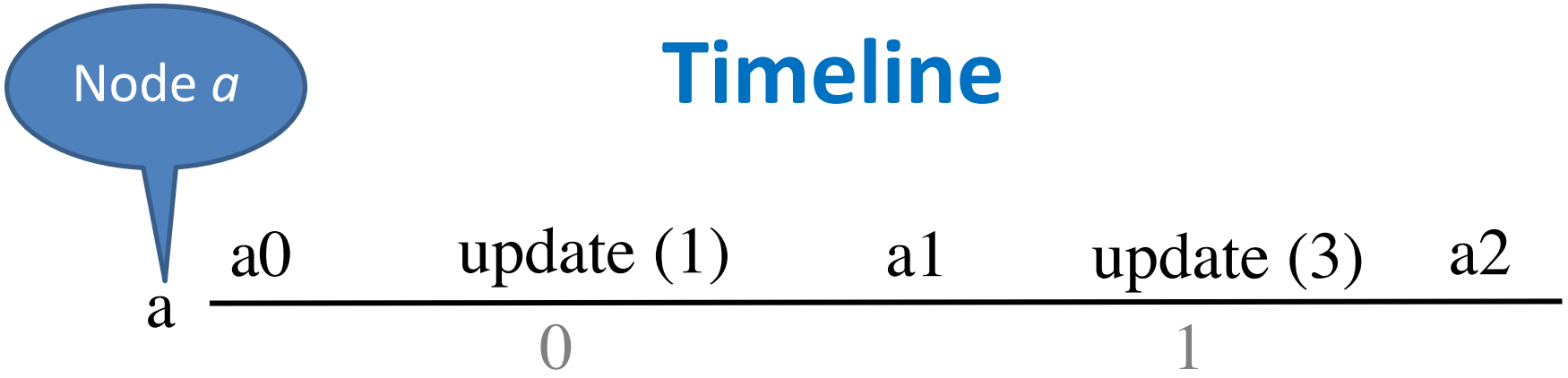
Timeline



Each state represents a snapshot of object in time that results from updates applied

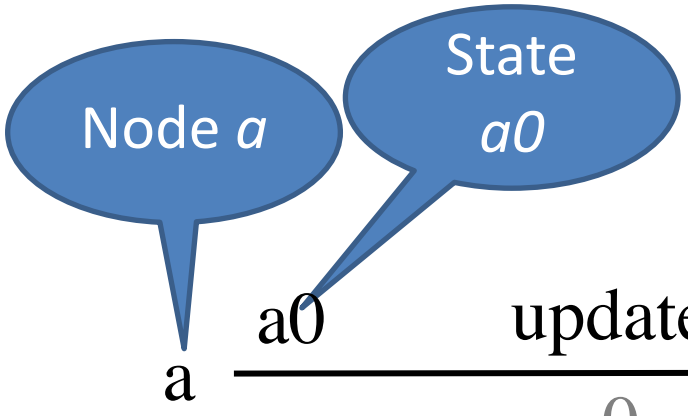
	state	query ()	history
a0	sum:0, cnt:0	0	{ }
a1	sum:1, cnt:1	1	{0}
a2	sum:4, cnt:2	2	{0,1}

Timeline



Each state represents a snapshot of object in time that results from updates applied

	state	query ()	history
a0	sum:0, cnt:0	0	{ }
a1	sum:1, cnt:1	1	{0}
a2	sum:4, cnt:2	2	{0,1}

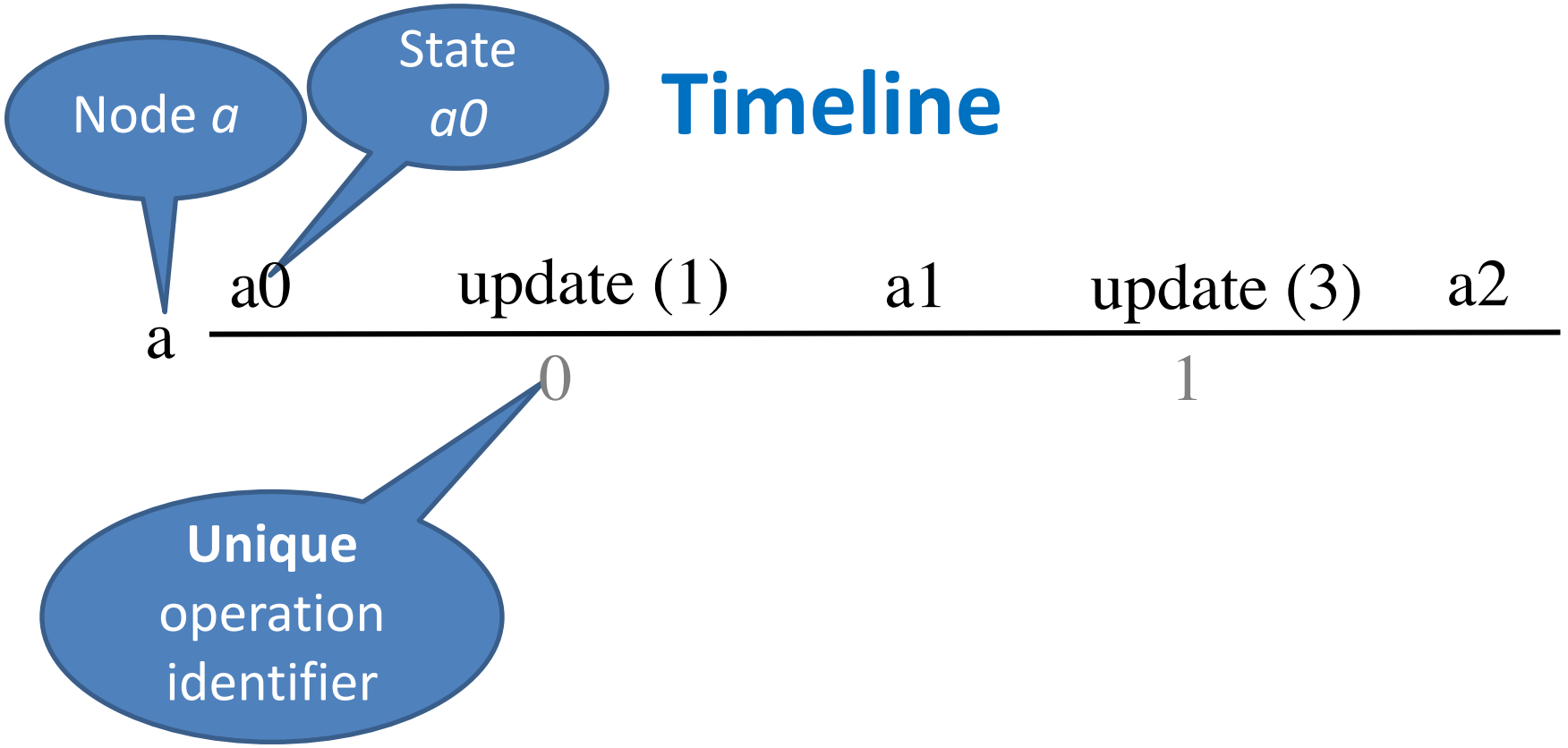


Timeline

Each state represents a snapshot of object in time that results from updates applied

	state	query ()	history
a0	sum:0, cnt:0	0	{ }
a1	sum:1, cnt:1	1	{0}
a2	sum:4, cnt:2	2	{0,1}

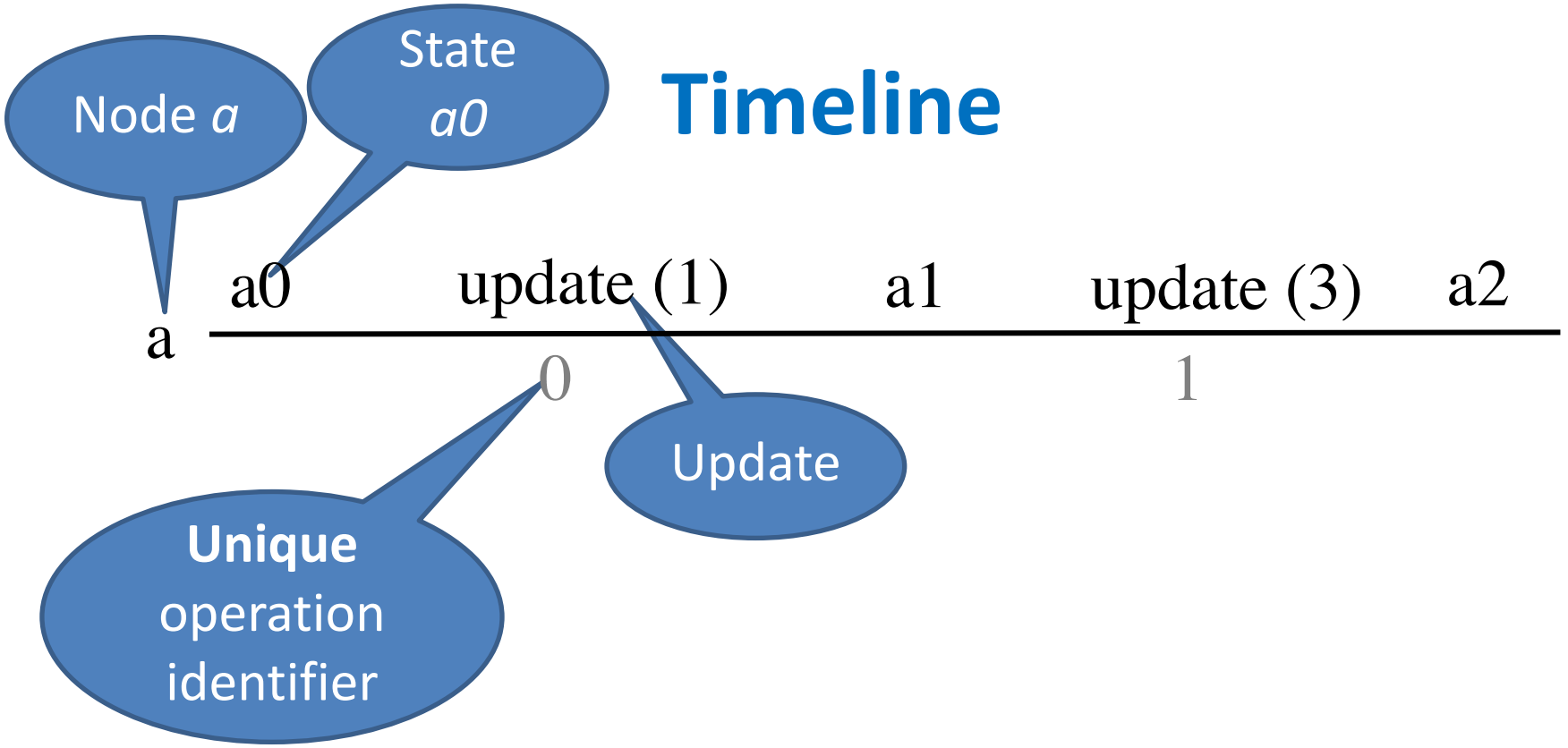
Timeline



Each state represents a snapshot of object in time that results from updates applied

	state	query ()	history
a0	sum:0, cnt:0	0	{ }
a1	sum:1, cnt:1	1	{0}
a2	sum:4, cnt:2	2	{0,1}

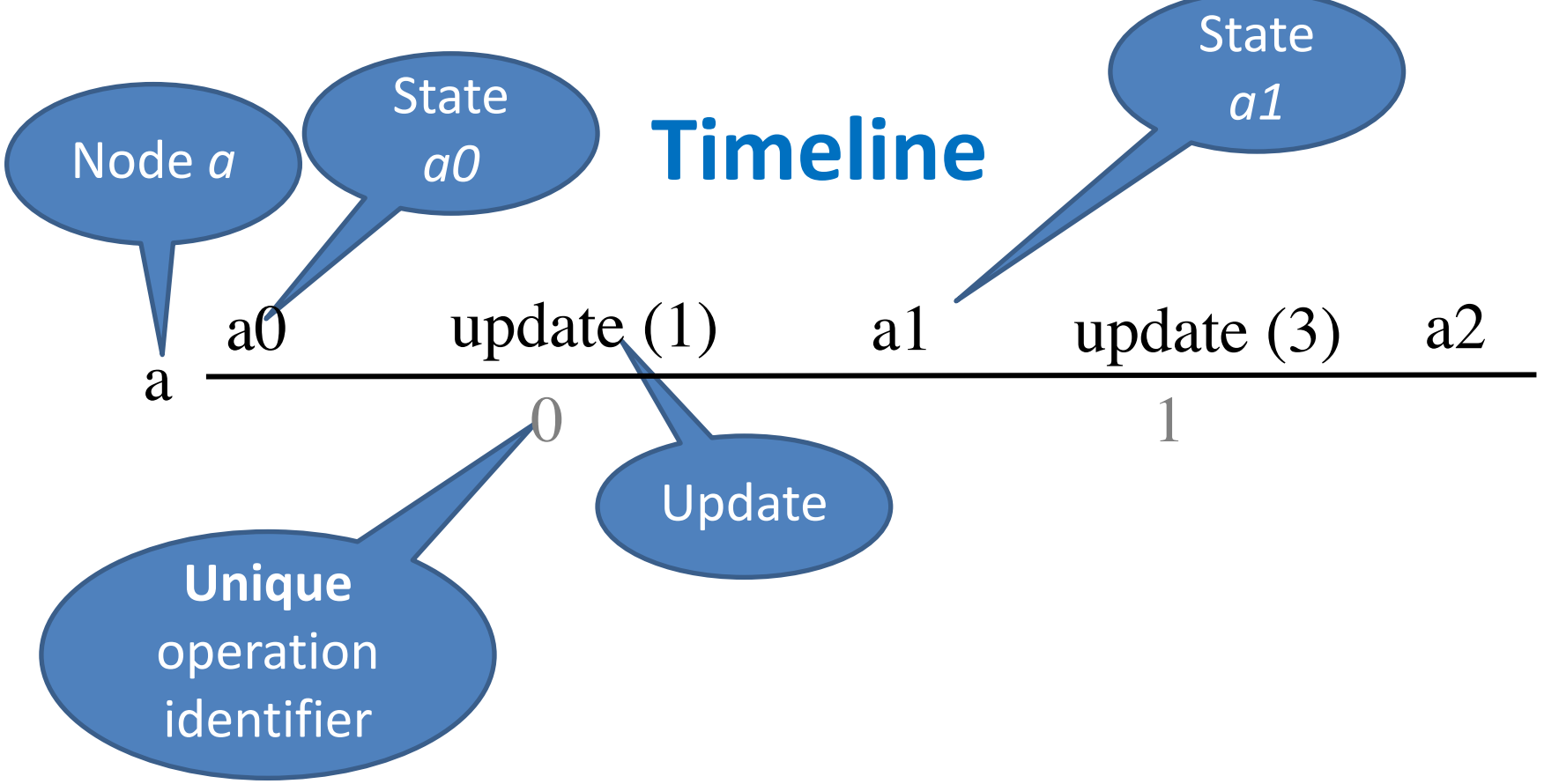
Timeline



Each state represents a snapshot of object in time that results from updates applied

	state	query ()	history
a0	sum:0, cnt:0	0	{ }
a1	sum:1, cnt:1	1	{0}
a2	sum:4, cnt:2	2	{0,1}

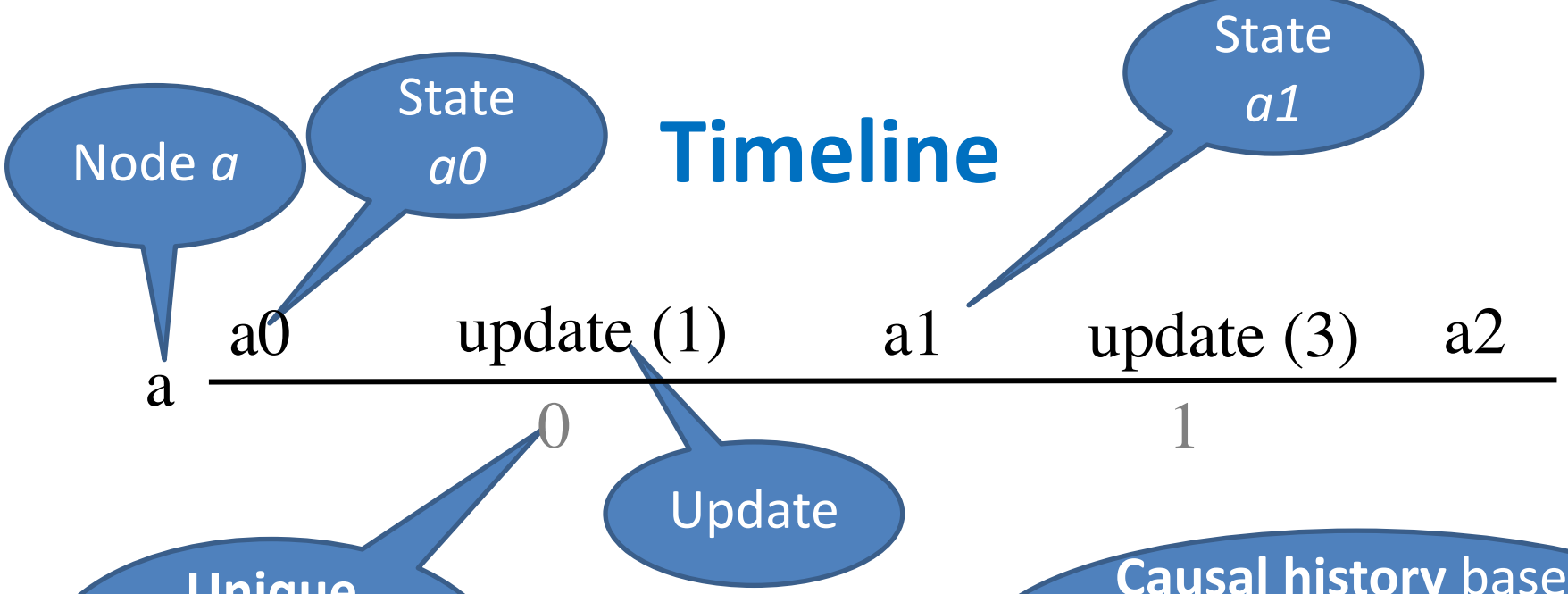
Timeline



Each state represents a snapshot of object in time that results from updates applied

	state	query ()	history
a0	sum:0, cnt:0	0	{ }
a1	sum:1, cnt:1	1	{0}
a2	sum:4, cnt:2	2	{0,1}

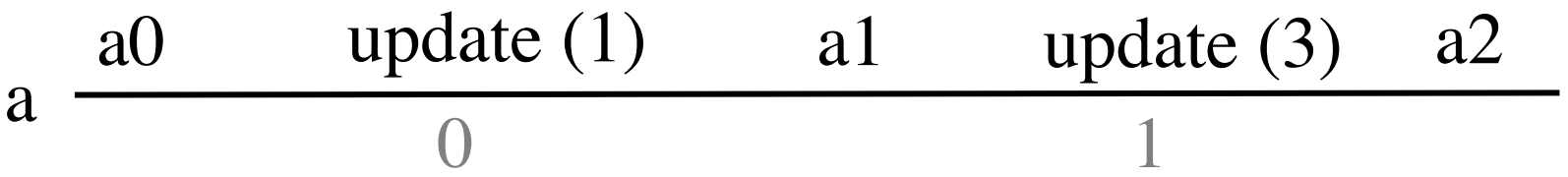
Timeline



Each state represents a snapshot of object in time that results from updates applied

	state	query ()	history
a0	sum:0, cnt:0	0	{ }
a1	sum:1, cnt:1	1	{0}
a2	sum:4, cnt:2	2	{0,1}

Timeline

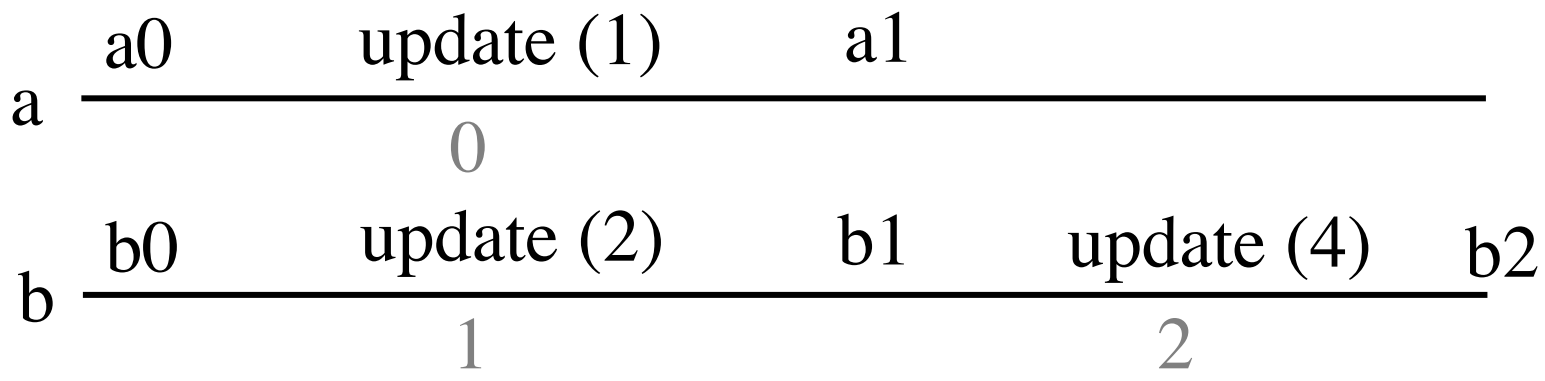


Operation identifier is unique across replicas

Each state represents a snapshot of object in time that results from updates applied

	state	query ()	history
a0	sum:0, cnt:0	0	{ }
a1	sum:1, cnt:1	1	{0}
a2	sum:4, cnt:2	2	{0,1}

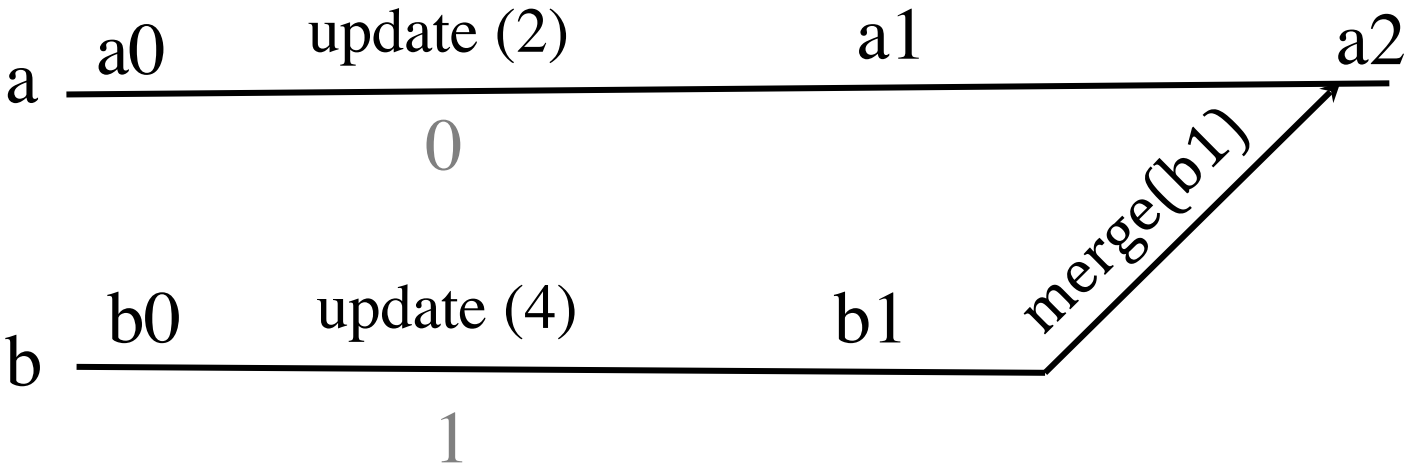
States and Causal Histories



If `y = x.update(…)` where the update has identifier *i*, then the causal history of *y* is the causal history of *x* union { *i* }.

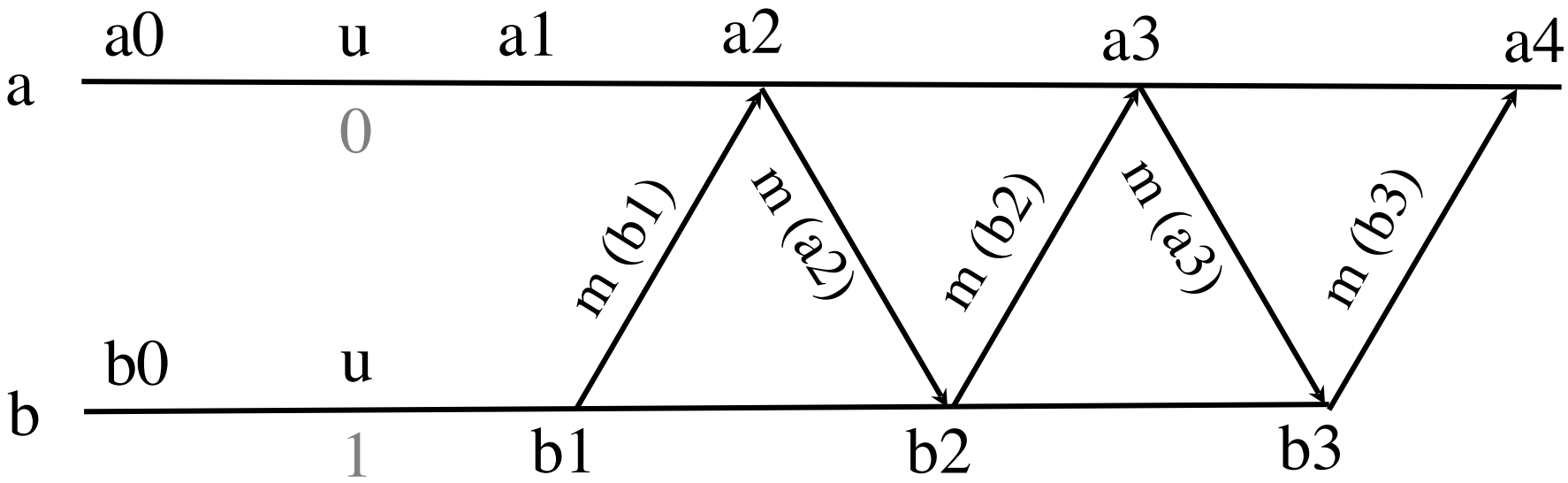
	state	query ()	history
a0	sum:0, cnt:0	0	{ }
a1	sum:1, cnt:1	1	{0}
b0	sum:0, cnt:0	0	{}
b1	sum:2, cnt:1	2	{1}
b2	sum:6, cnt:2	3	{1,2}

Merge



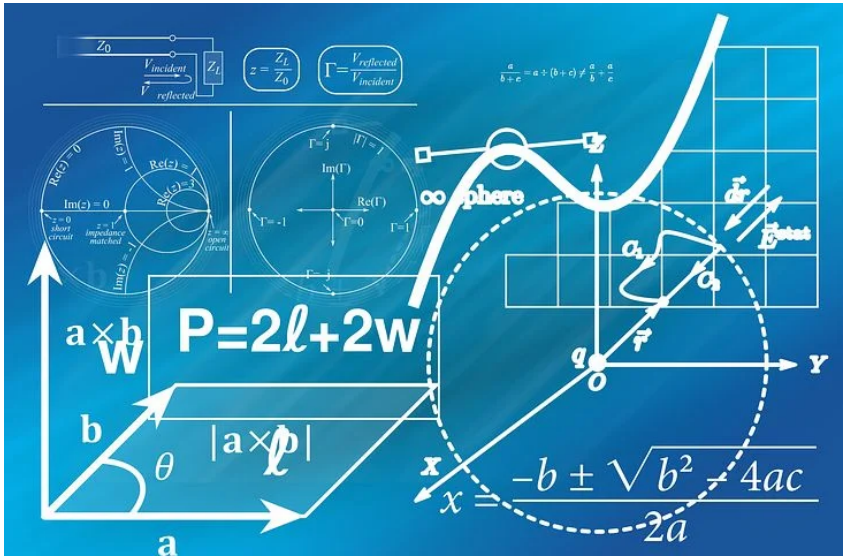
	state	query ()	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
a2	sum:6, cnt:2	3	{0,1}

Nodes Periodically Propagate Their State



Self-study Questions

- Think of a few basic data structures, like lists, sets, counters, binary trees, heaps, maps, etc., and visualize for yourself what happens if replicated instances of these structures are updated via gossip.
- For the above data structures, specify merge operations that merge the state of two instances of a given structure.
- Assume merge happens periodically, does your replicated structures' state converge?



Pixabay.com

CRDT –
EVENTUAL CONSISTENCY, MORE FORMALLY

Eventual Consistency

- A replicated state-based object is
  - **eventually consistent** if whenever two replicas of the state-based object have the same causal history, **they eventually** (not necessarily immediately) converge to the same internal state

Strong Eventual Consistency

- A replicated state-based object is
  - **strongly eventually consistent** if whenever two replicas of the state-based object have the same causal history, **they** (immediately) **have the same internal state**

- Strong eventual consistency implies eventual consistency

EC or SEC

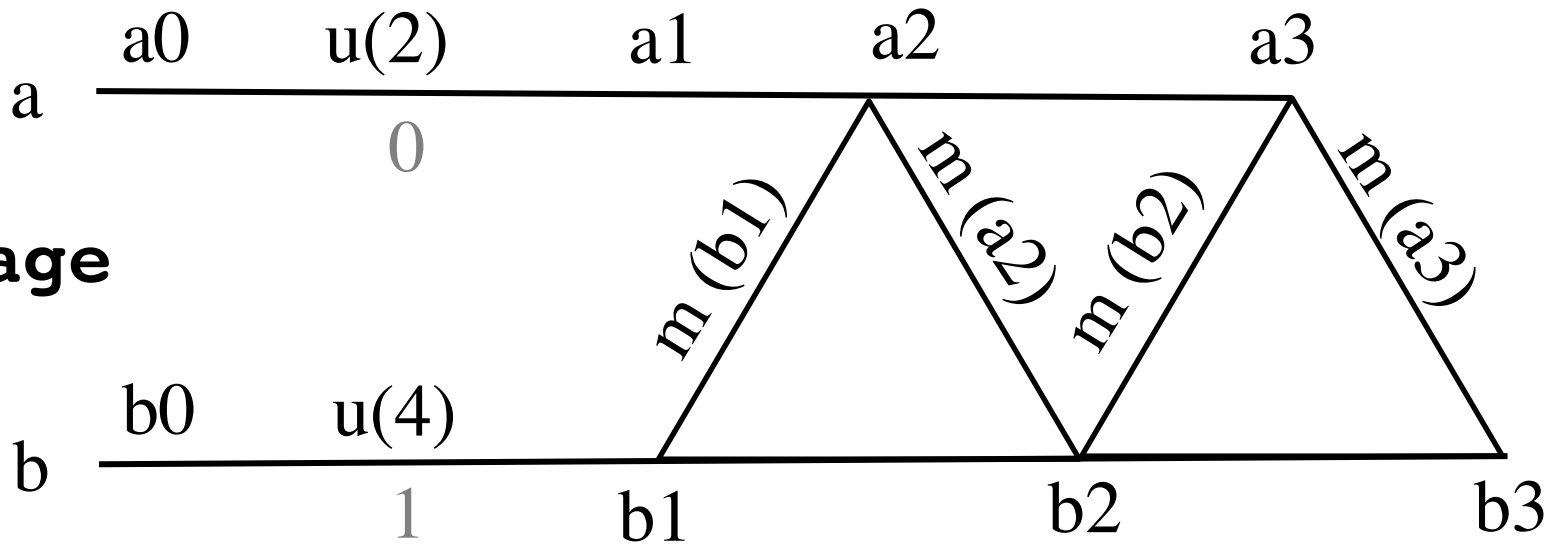
That is the question?

- Variants of our Average object, defined next
  - Average
  - NoMergeAverage
  - BMergeAverage
  - MaxAverage

- Note that some of these objects do not represent realistic functionality (i.e., needed functionality)

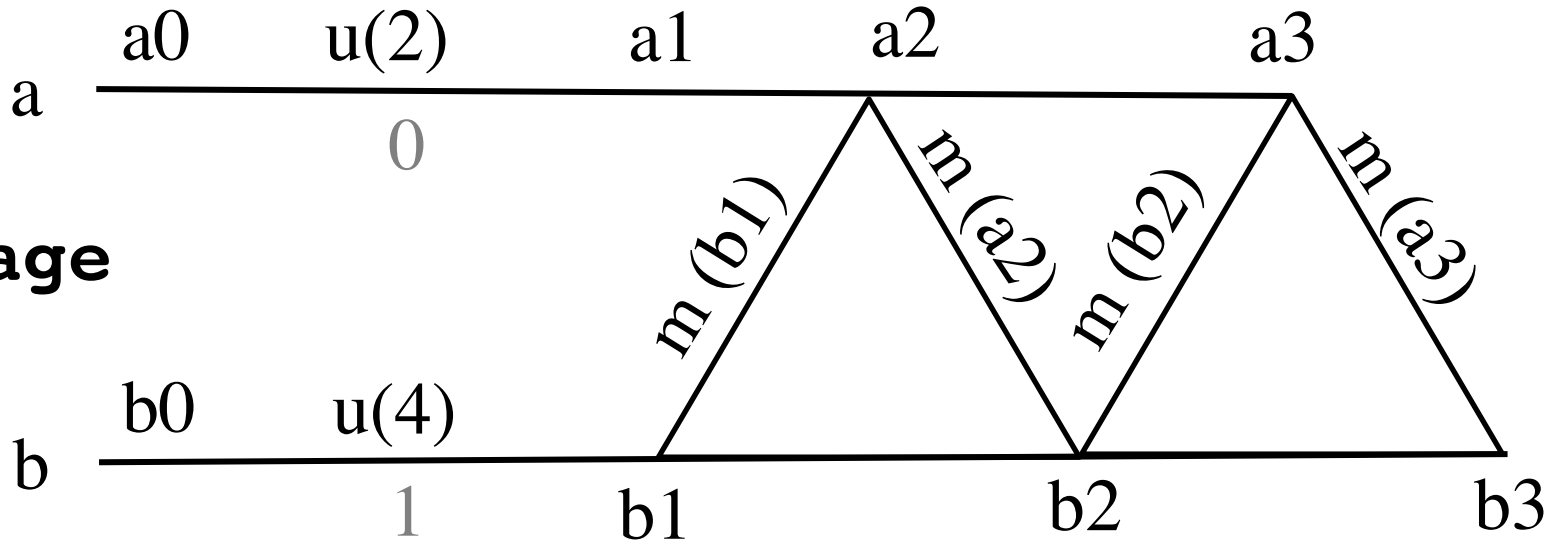
- These objects are meant to illustrate convergence concepts only

Average



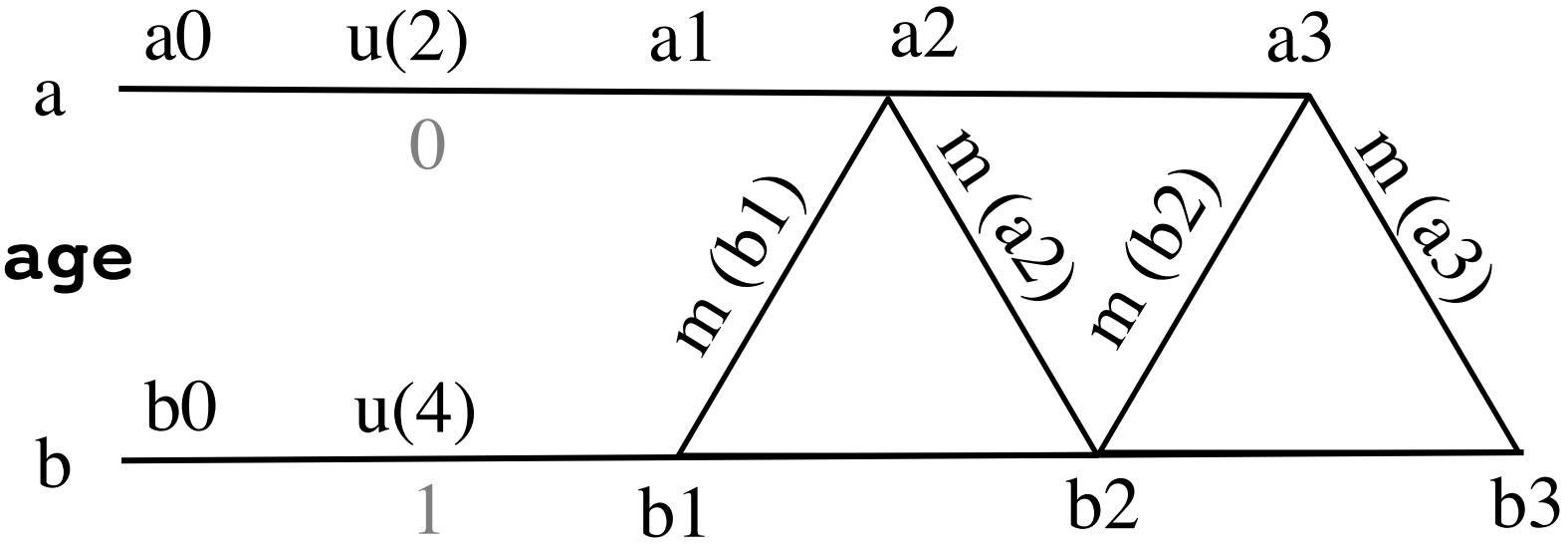
	state	query ( )	history
a0			
a1			
a2			
a3			
b0			
b1			
b2			
b3			

Average



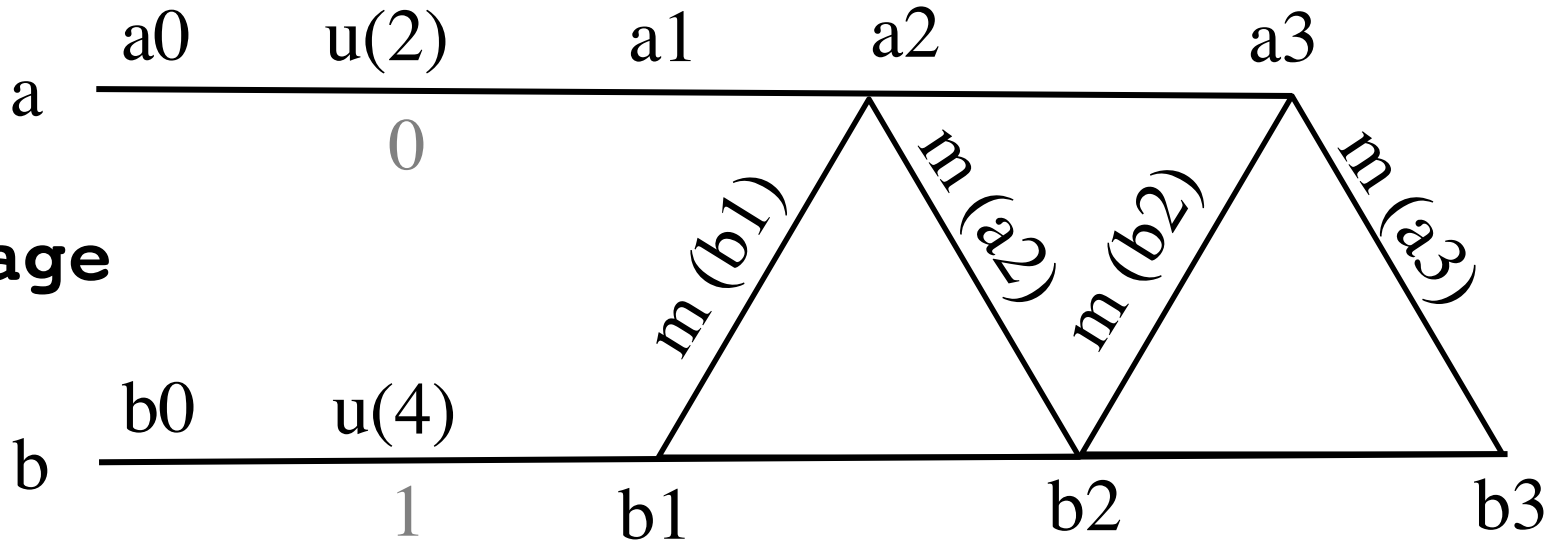
	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1			
a2			
a3			
b0			
b1			
b2			
b3			

Average



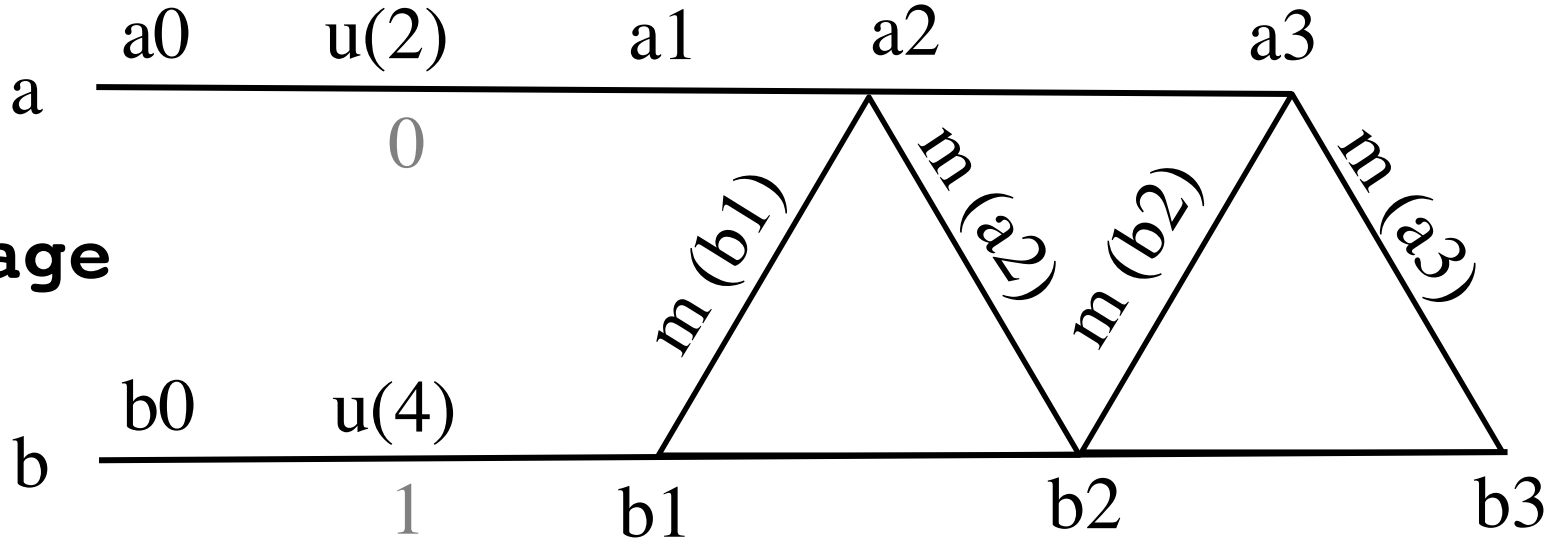
	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2			
a3			
b0			
b1			
b2			
b3			

Average



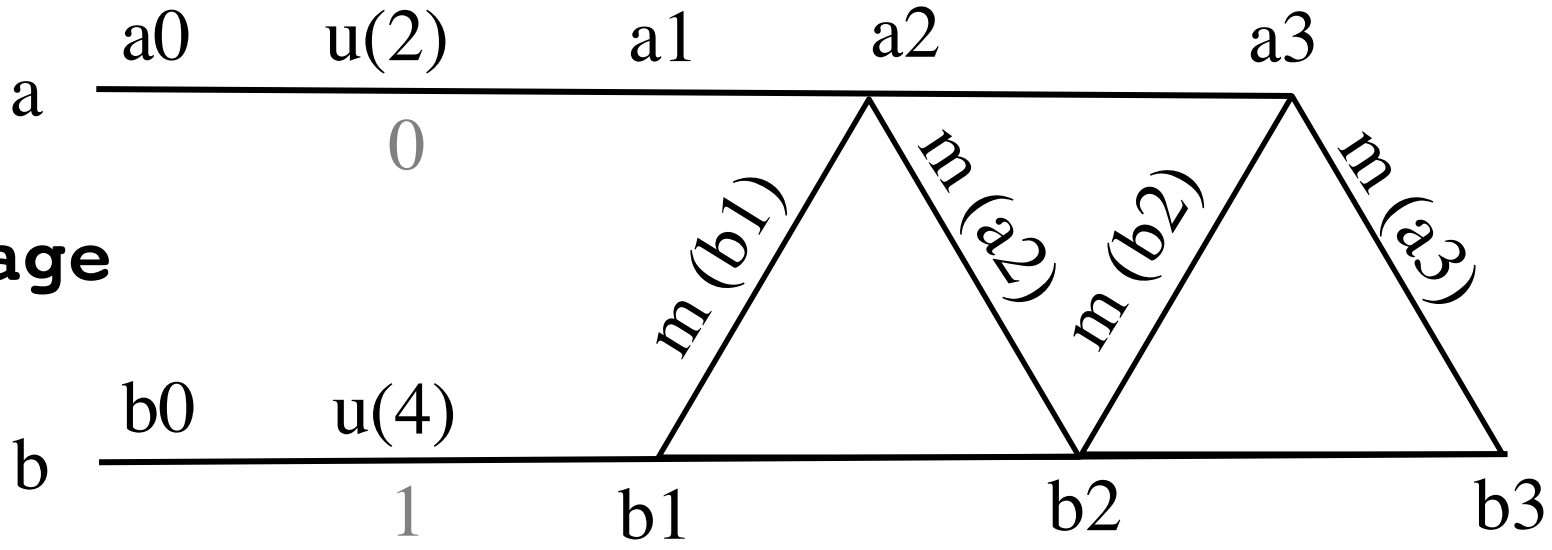
	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2			
a3			
b0	sum:0, cnt:0	0	{}
b1			
b2			
b3			

Average



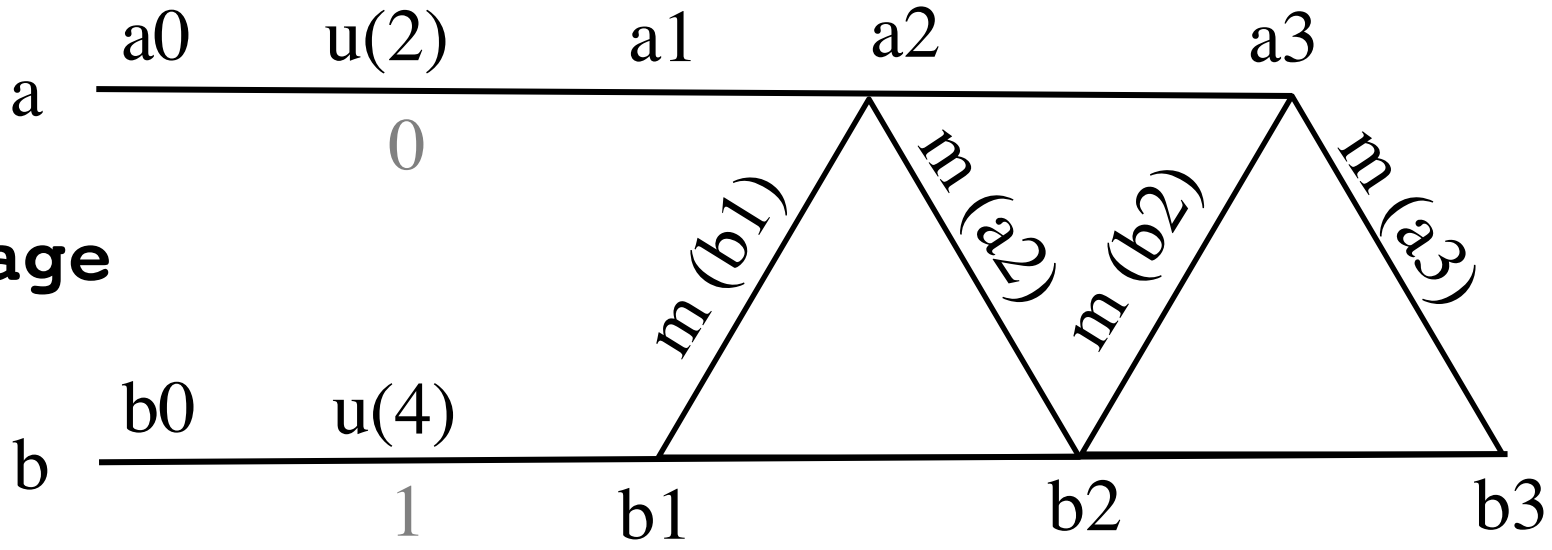
	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2			
a3			
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2			
b3			

Average



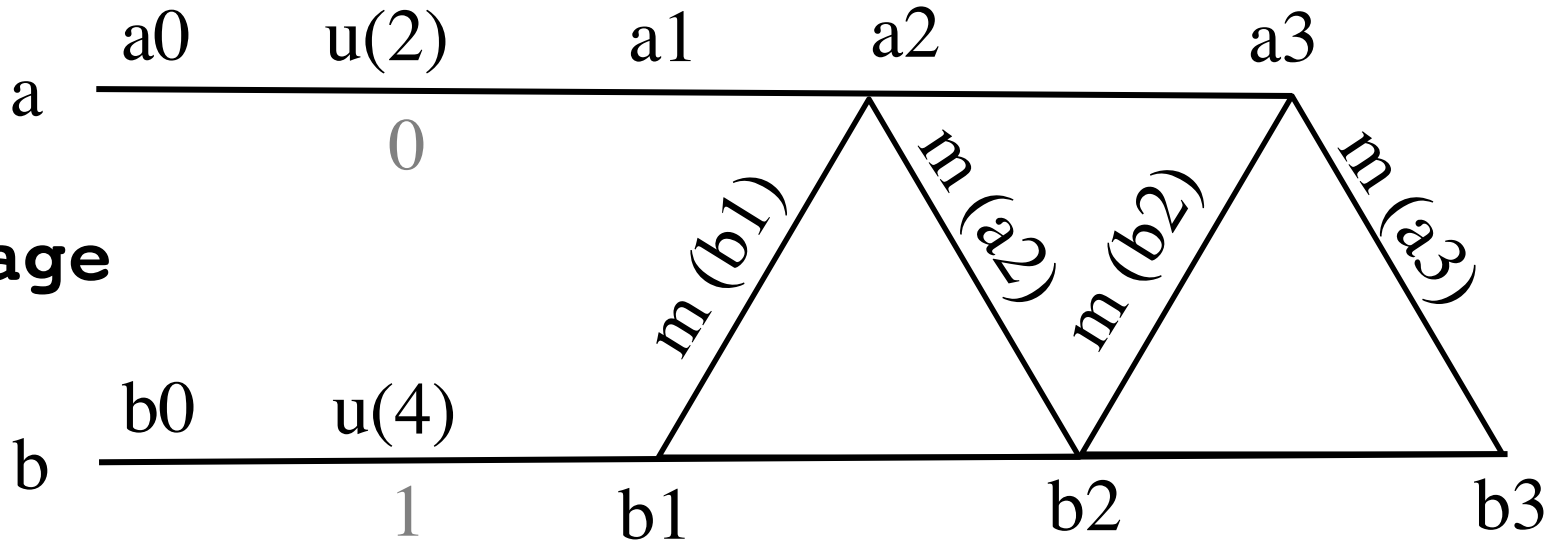
	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:6, cnt:2	3	{0,1}
a3			
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2			
b3			

Average



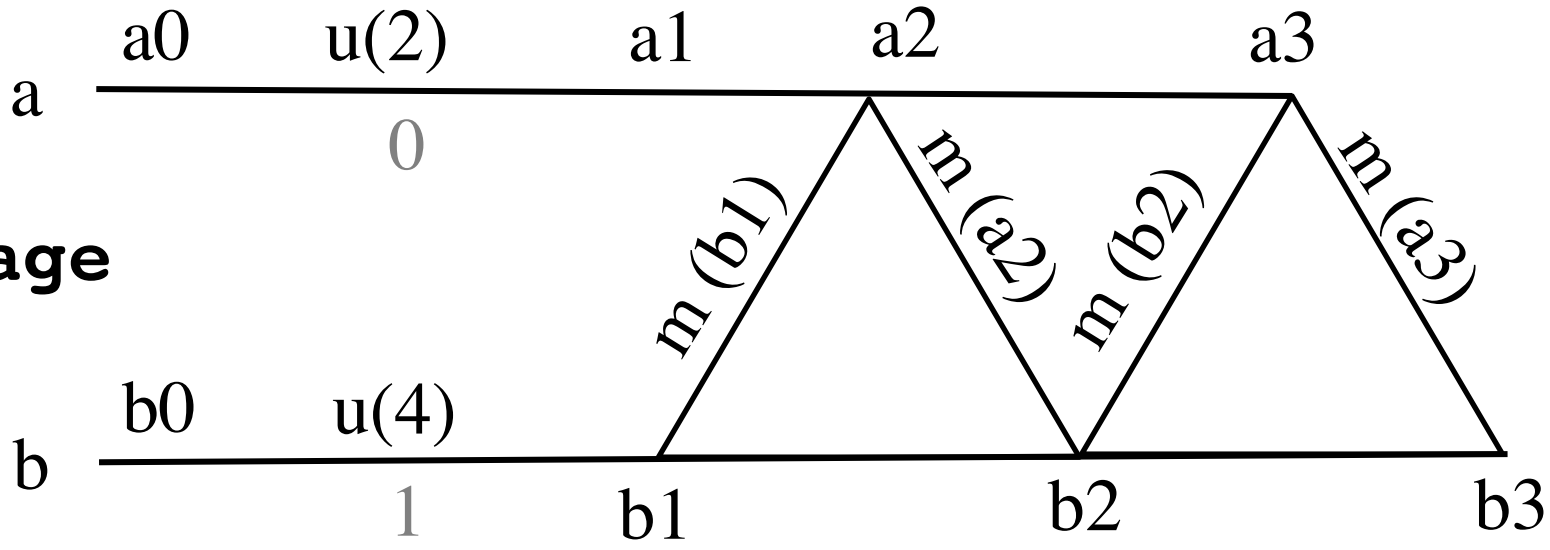
	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:6, cnt:2	3	{0,1}
a3			
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:10, cnt:3	3.3	{0,1}
b3			

Average



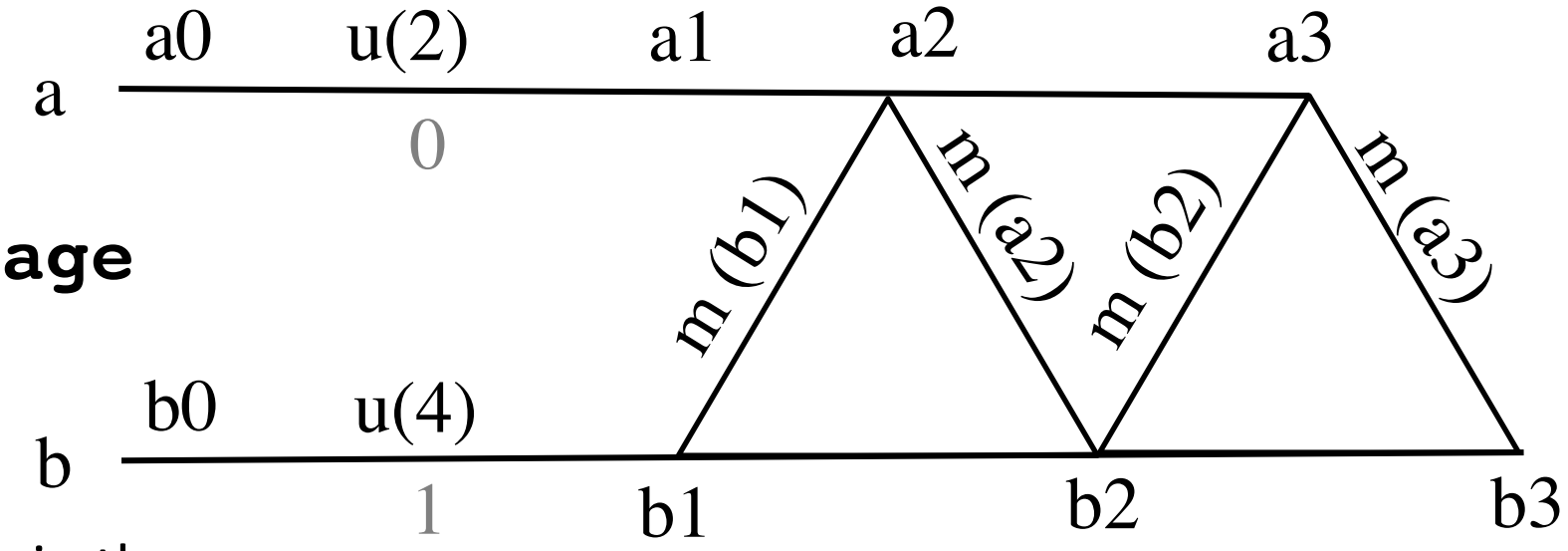
	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:6, cnt:2	3	{0,1}
a3	sum:16, cnt:5	3.2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:10, cnt:3	3.3	{0,1}
b3			

Average



	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:6, cnt:2	3	{0,1}
a3	sum:16, cnt:5	3.2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:10, cnt:3	3.3	{0,1}
b3	sum:26, cnt:8	3.25	{0,1}

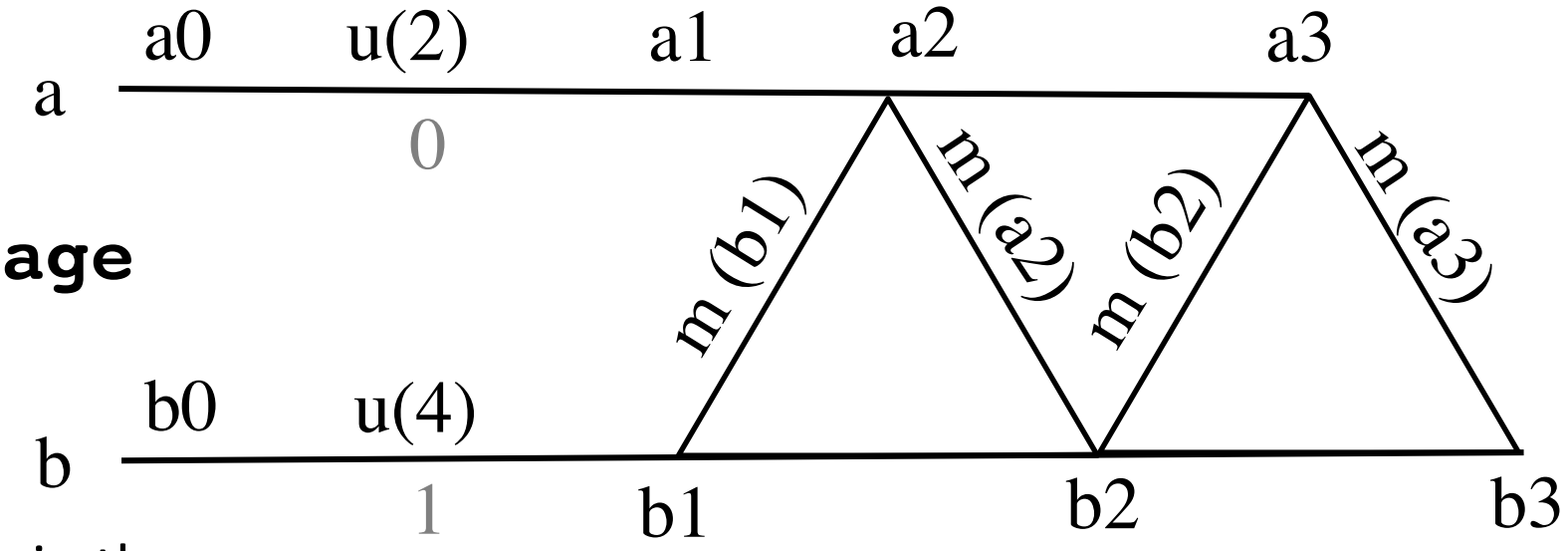
Average



a, b attain the **same causal history** but **do not converge** to the **same internal state** – they **do not converge** at all**!**

	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:6, cnt:2	3	{0,1}
a3	sum:16, cnt:5	3.2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:10, cnt:3	3.3	{0,1}
b3	sum:26, cnt:8	3.25	{0,1}

Average



a, b attain the **same causal history** but **do not converge** to the **same internal state** – they **do not converge** at all**!**

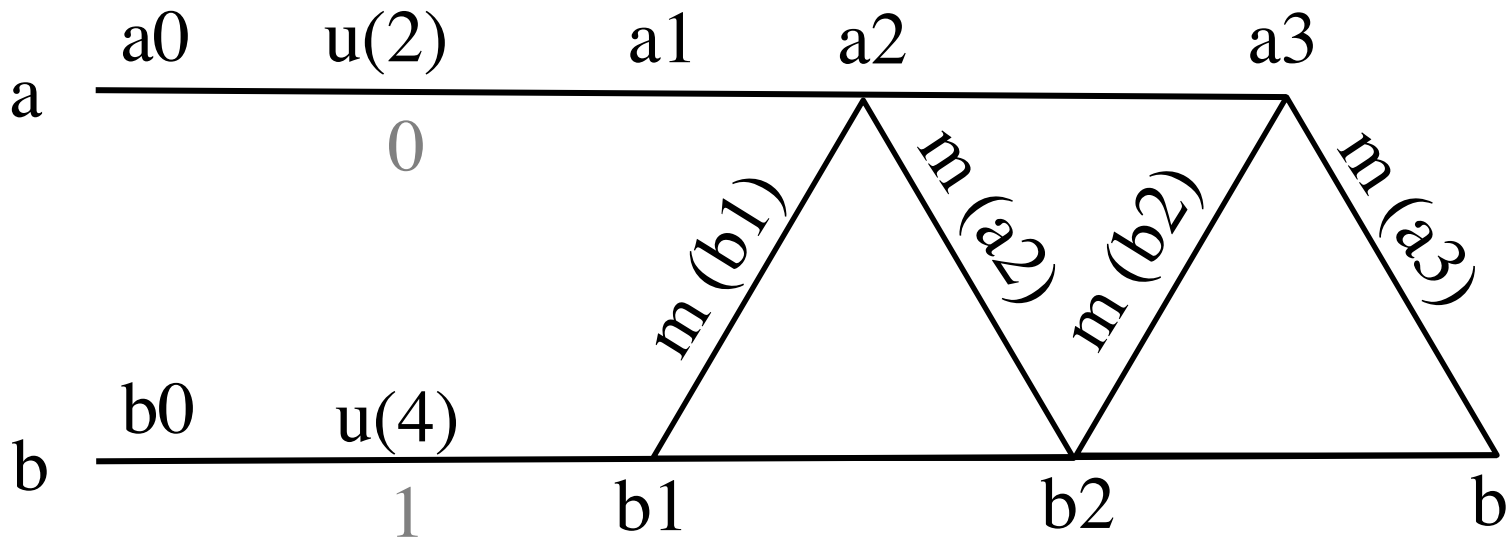
Neither eventually consistent, nor strongly eventually consistent

	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:6, cnt:2	3	{0,1}
a3	sum:16, cnt:5	3.2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:10, cnt:3	3.3	{0,1}
b3	sum:26, cnt:8	3.25	{0,1}

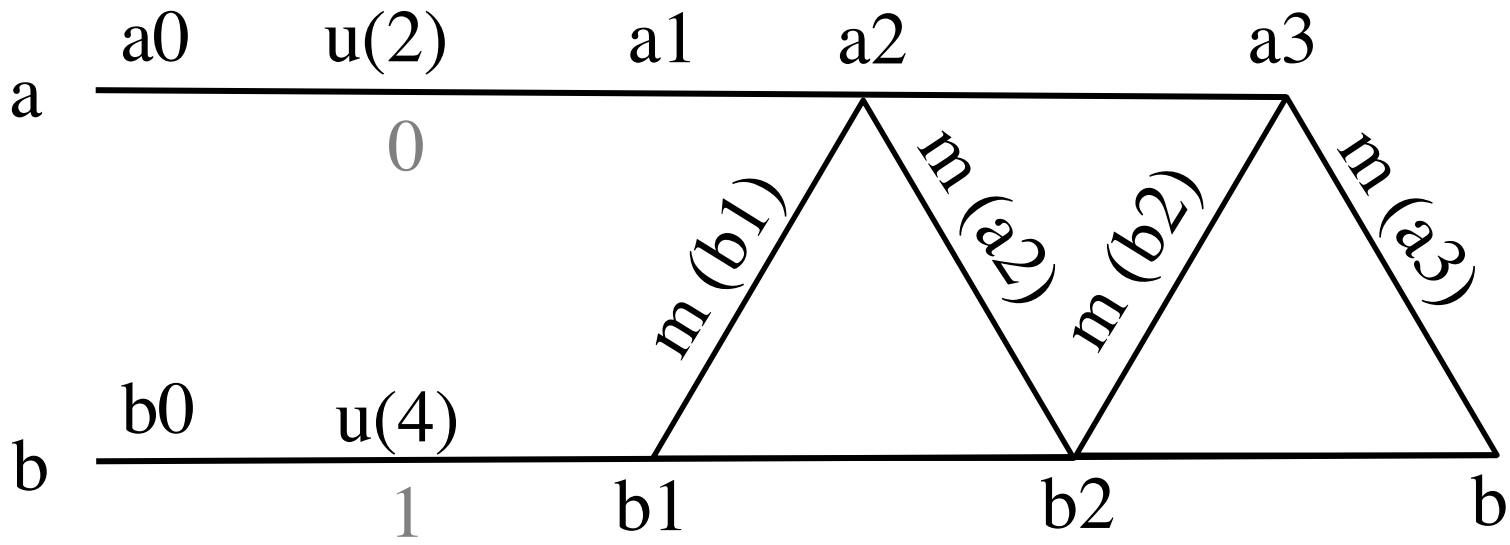
`NoMergeAverage`

- **Object's merge does nothing**

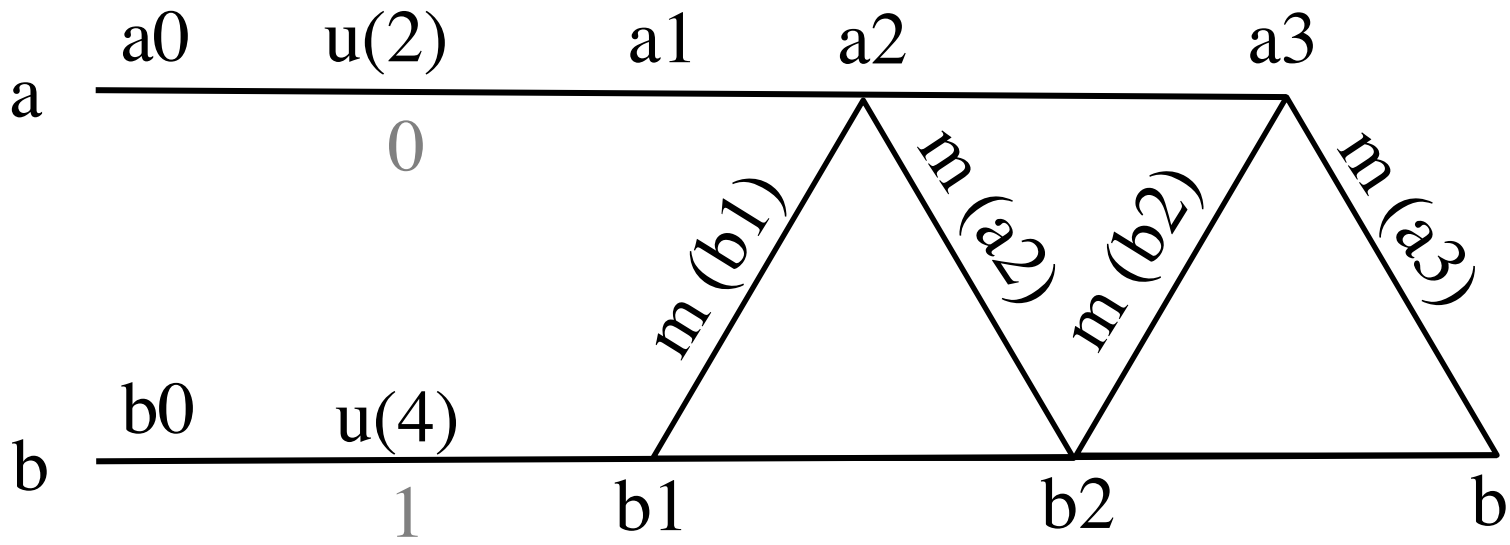
- All else is the same as for `Average`



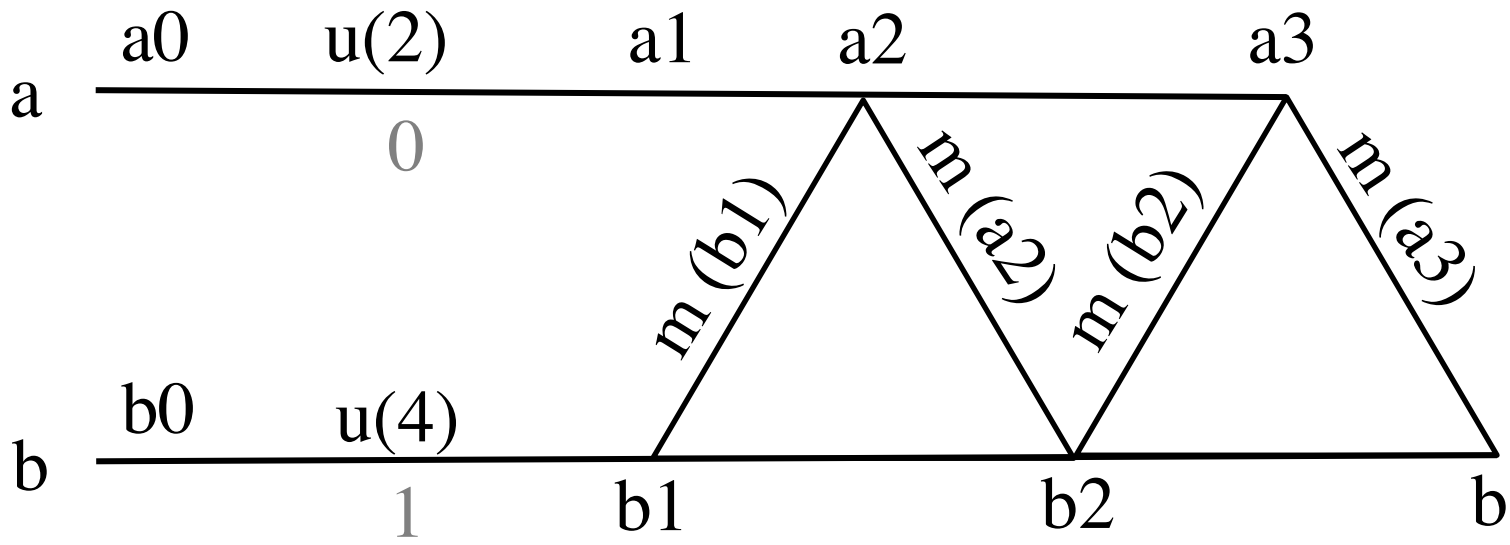
	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:2, cnt:1	2	{0,1}
a3	sum:2, cnt:1	2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}



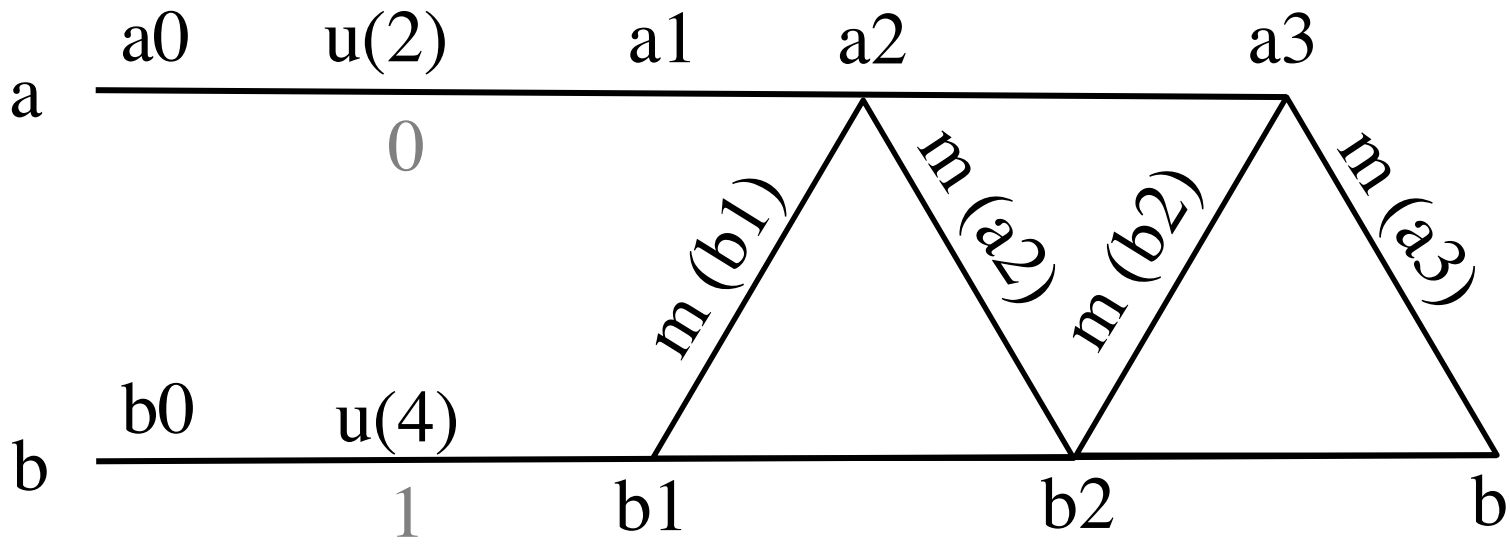
	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
→ a1	sum:2, cnt:1	2	{0}
a2	sum:2, cnt:1	2	{0,1}
a3	sum:2, cnt:1	2	{0,1}
b0	sum:0, cnt:0	0	{}
→ b1	sum:4, cnt:1	4	{1}
b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}



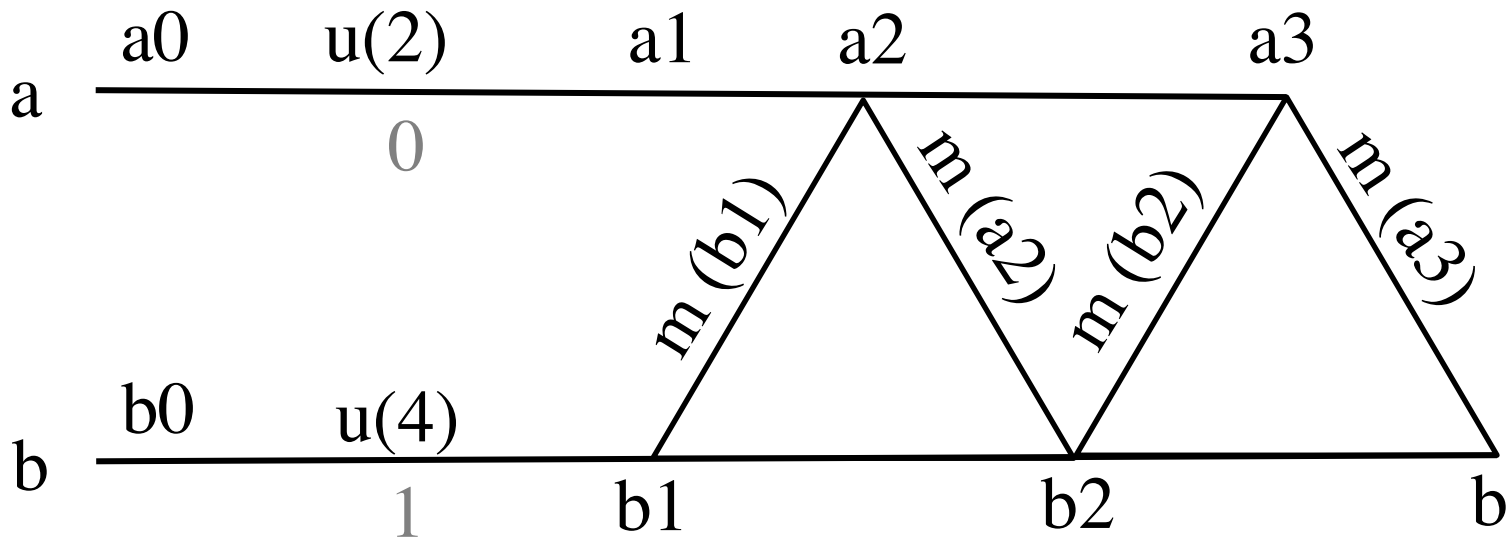
	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:2, cnt:1	2	{0,1}
a3	sum:2, cnt:1	2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}



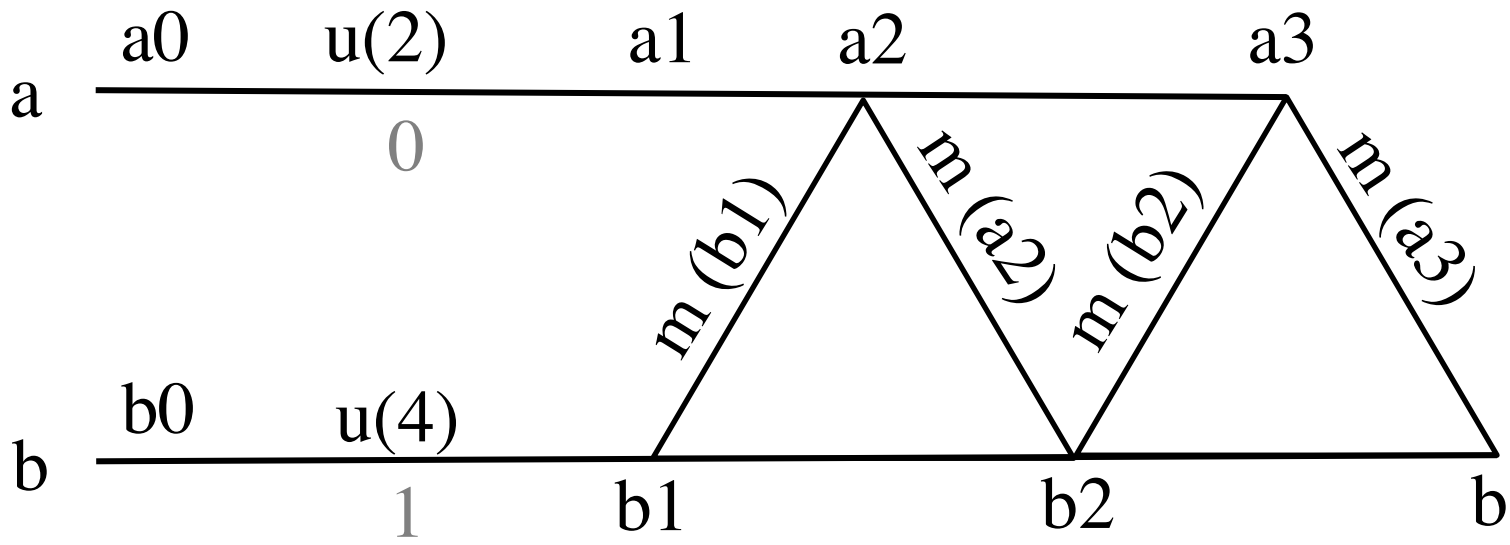
	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
→ a2	sum:2, cnt:1	2	{0,1}
a3	sum:2, cnt:1	2	{0,1}
b0	sum:0, cnt:0	0	{}
→ b1	sum:4, cnt:1	4	{1}
b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}



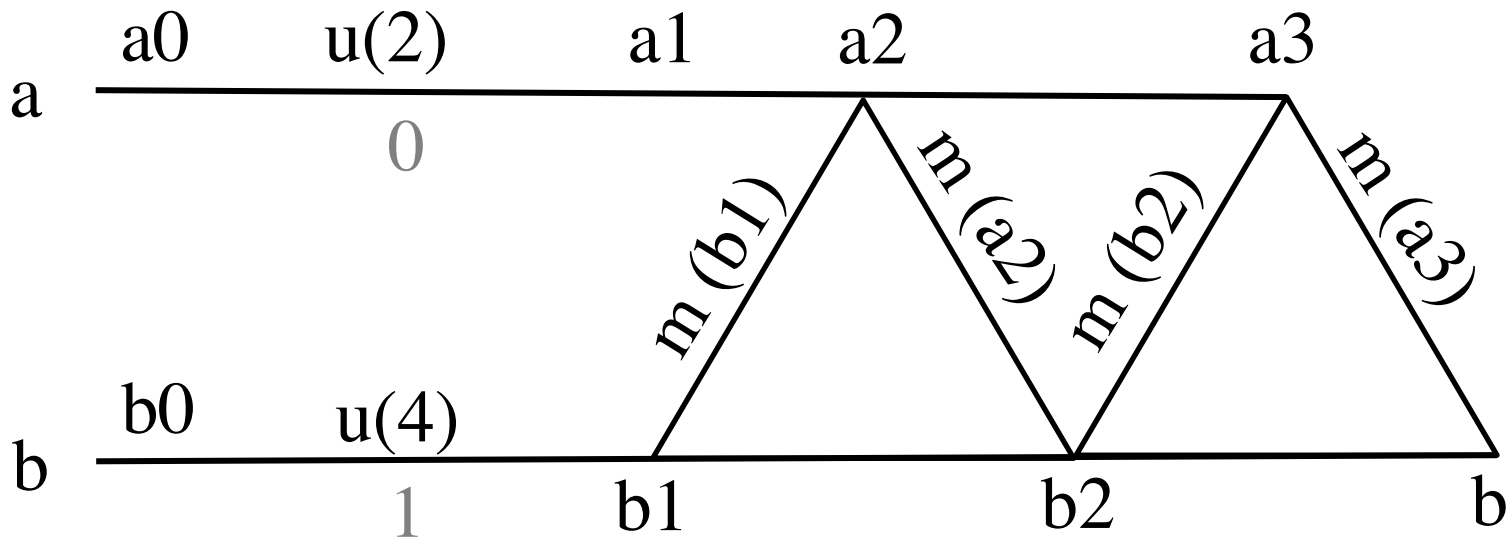
	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:2, cnt:1	2	{0,1}
a3	sum:2, cnt:1	2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}



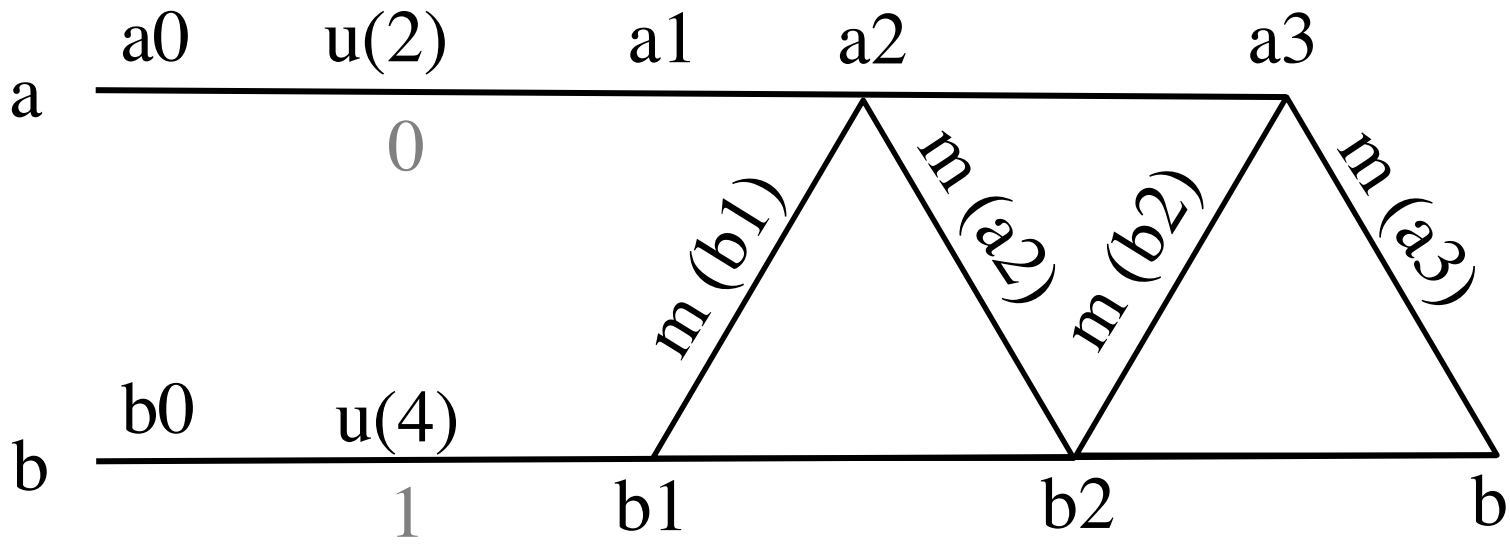
	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
→ a2	sum:2, cnt:1	2	{0,1}
a3	sum:2, cnt:1	2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
→ b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}



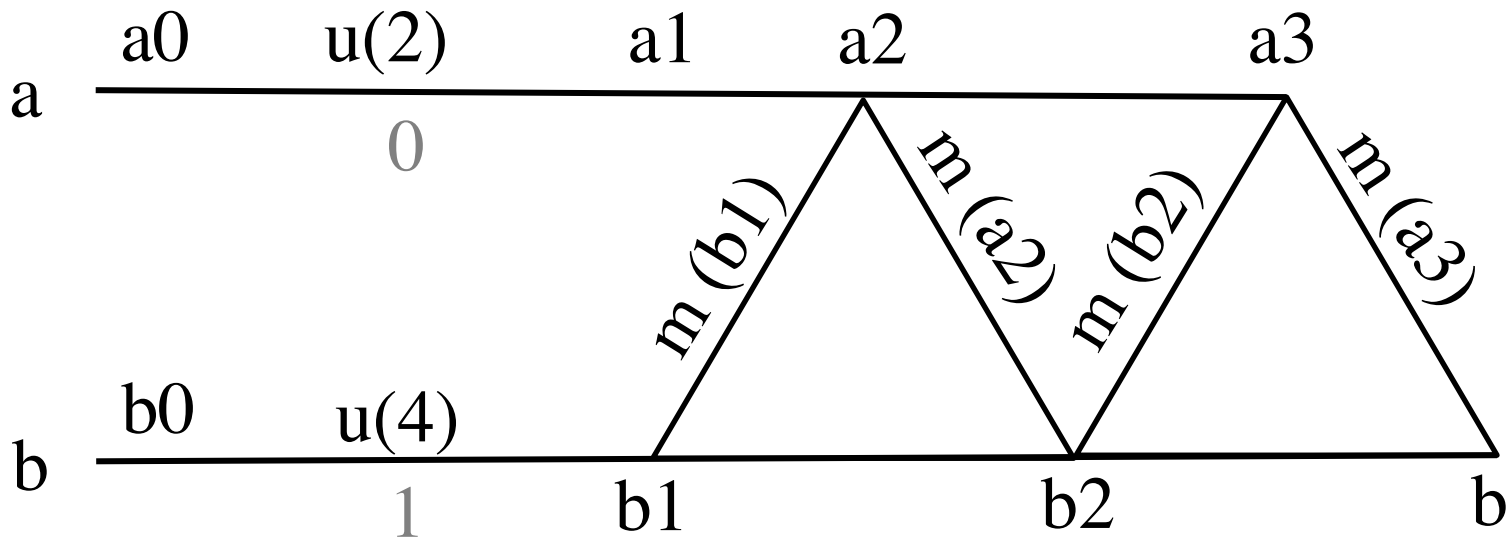
	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:2, cnt:1	2	{0,1}
a3	sum:2, cnt:1	2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}



	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:2, cnt:1	2	{0,1}
→ a3	sum:2, cnt:1	2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
→ b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}

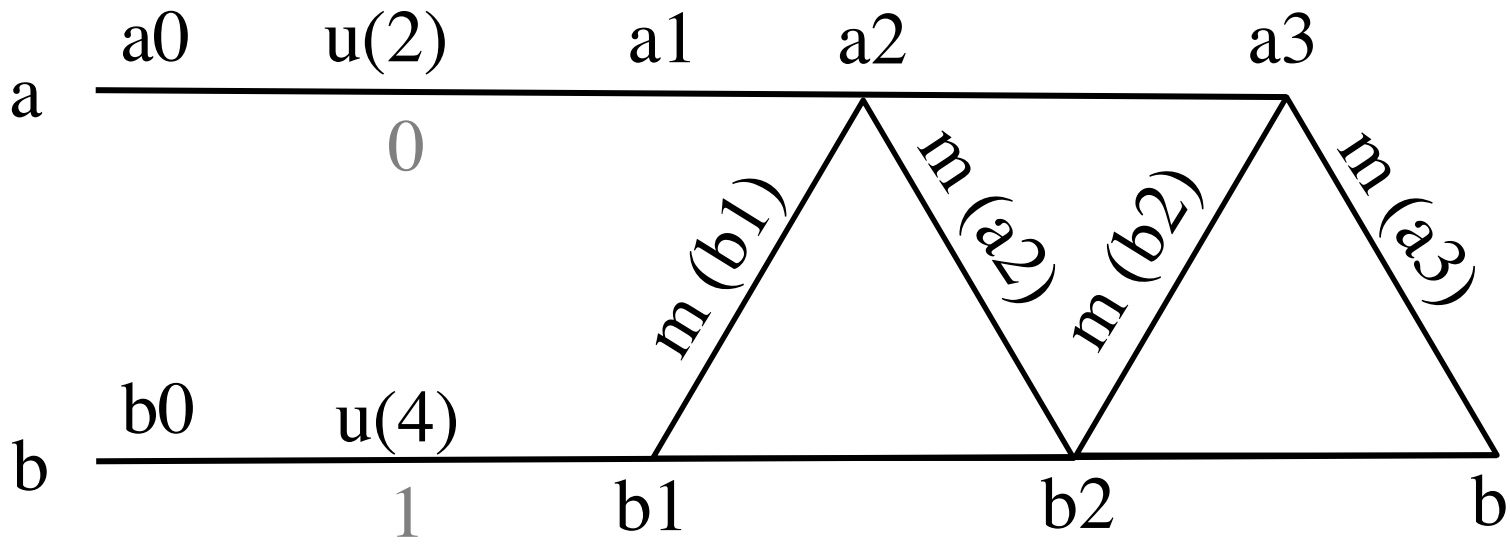


	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:2, cnt:1	2	{0,1}
a3	sum:2, cnt:1	2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}



a, b have **same causal history,** both **converge** to a stable but *different* **internal state**.

	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:2, cnt:1	2	{0,1}
a3	sum:2, cnt:1	2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}



a, b have **same causal history,** both **converge** to a stable but *different* **internal state**.

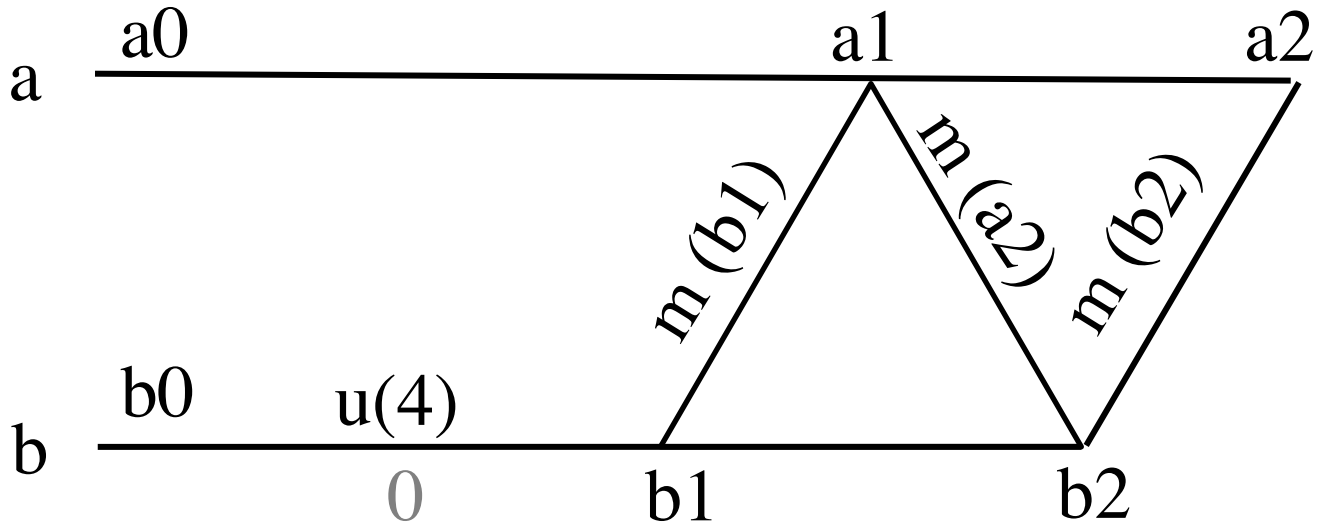
Neither eventually consistent, nor strongly eventually consistent.

	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:2, cnt:1	2	{0,1}
a3	sum:2, cnt:1	2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}

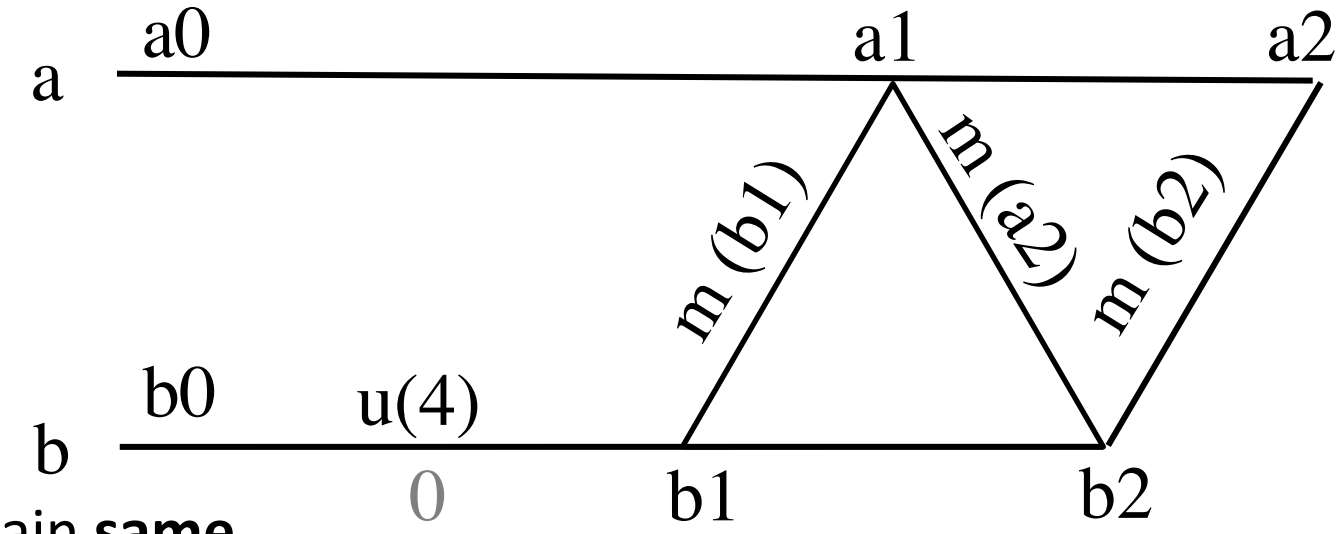
BMergeAverage

- Object's merge
  - **At *b* – overwrite state with state at *a***
  - **At *a* – do nothing**

- All else is the same as for `Average`

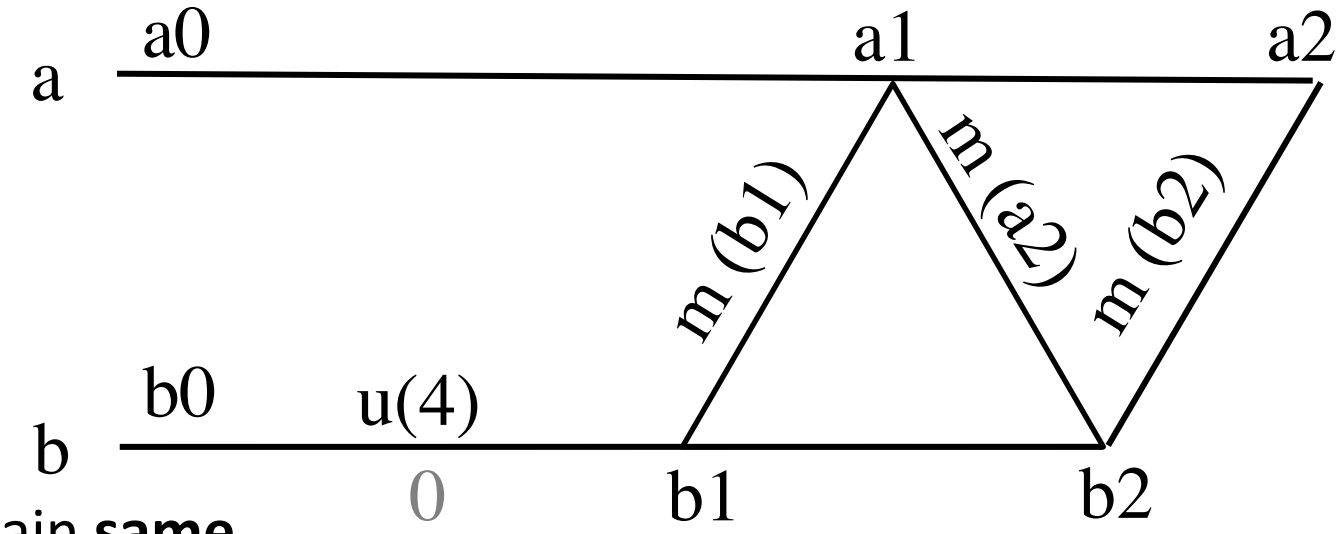


	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:0, cnt:0	0	{0}
a2	sum:0, cnt:0	0	{0}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{0}
b2	sum:0, cnt:0	0	{0}



a, b attain **same causal history,** both eventually **converge** to the same **internal state – eventual consistent**.

	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:0, cnt:0	0	{0}
a2	sum:0, cnt:0	0	{0}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{0}
b2	sum:0, cnt:0	0	{0}



a, b attain **same causal history,** both eventually **converge** to the same **internal state – eventual consistent**.

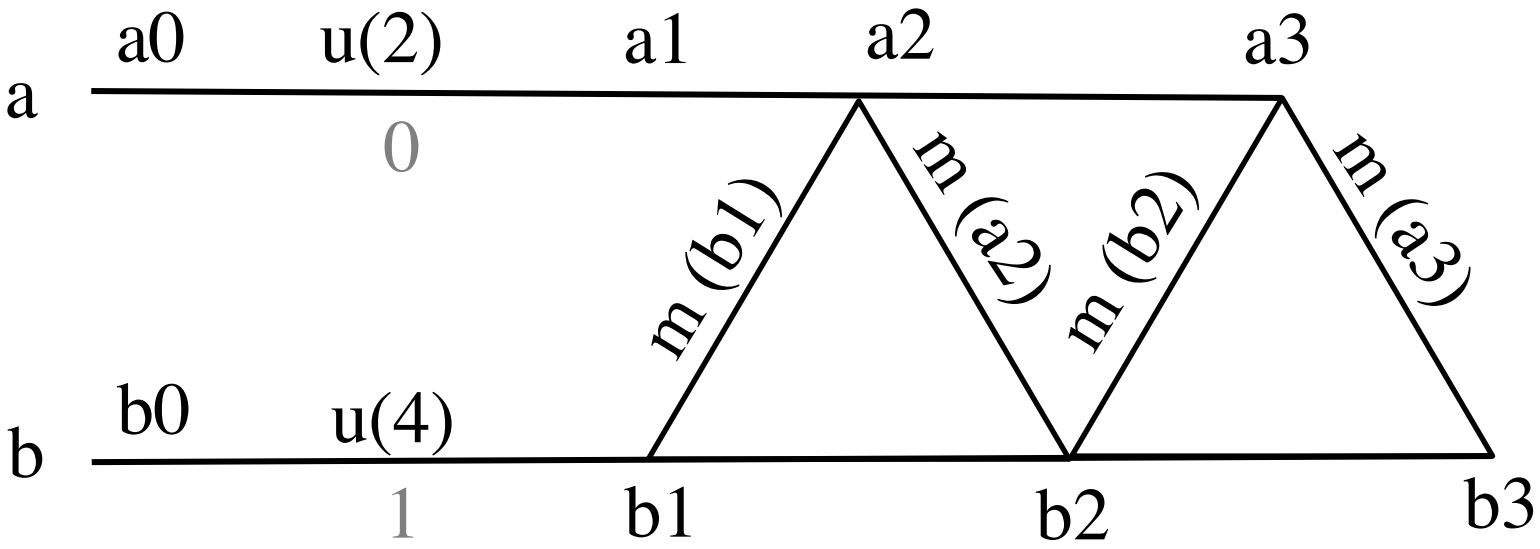
a1, b1 have same causal history but different internal state **– not strongly eventually consistent**

	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:0, cnt:0	0	{0}
a2	sum:0, cnt:0	0	{0}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{0}
b2	sum:0, cnt:0	0	{0}

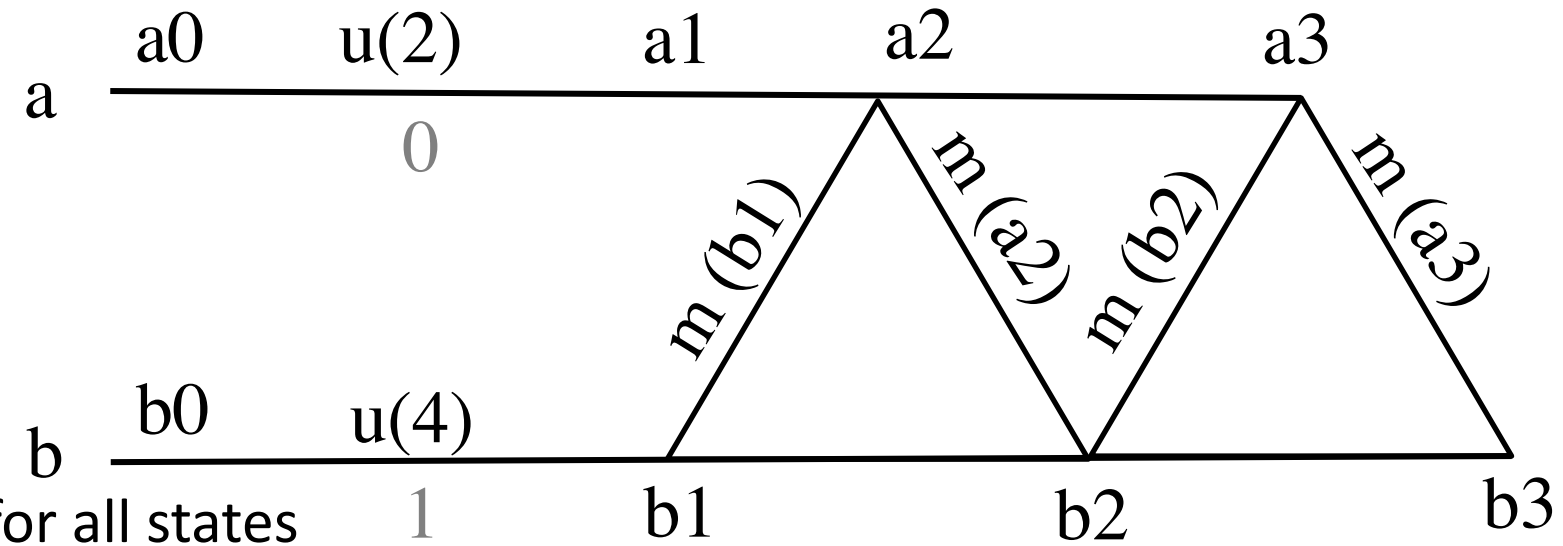
MaxAverage

- Object's merge
  - **Pair-wise max of `sum` and `cnt`**

- All else is the same as for `Average`

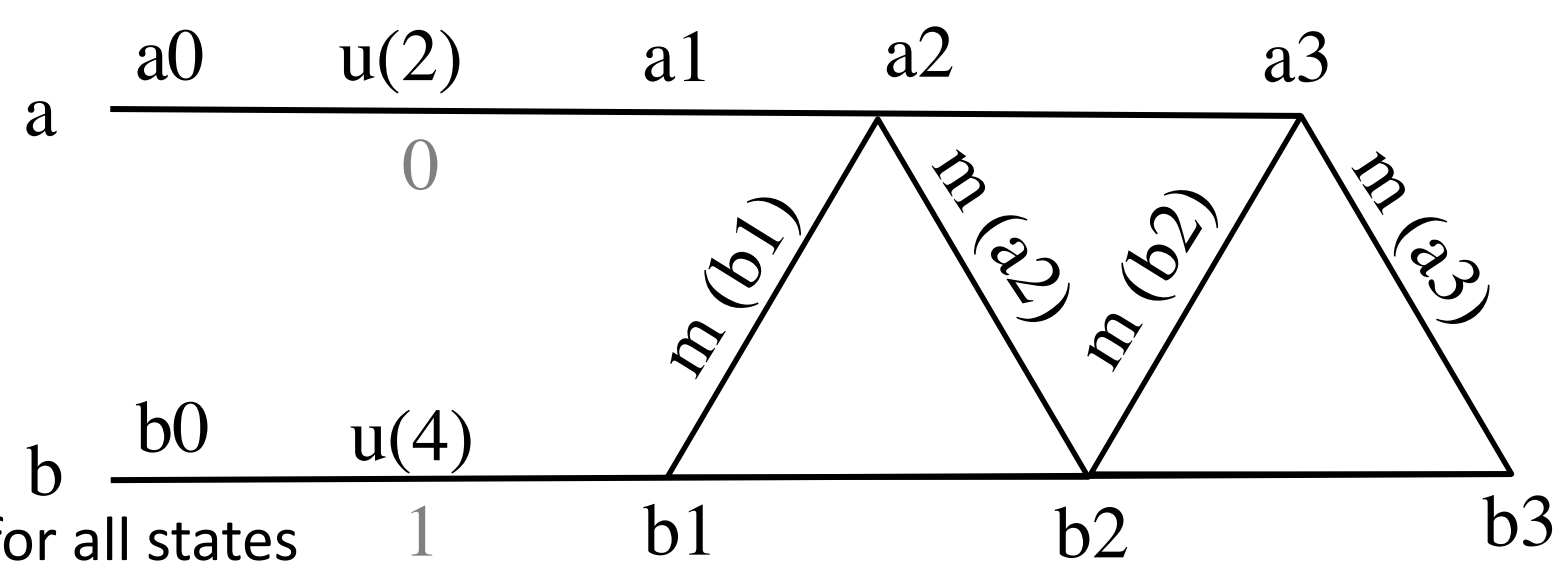


	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:4, cnt:1	4	{0,1}
a3	sum:4, cnt:1	4	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}



At a, b for all states with the **same causal history,** they have the **same internal state – strongly eventually consistent**.

	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:4, cnt:1	4	{0,1}
a3	sum:4, cnt:1	4	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}



At a, b for all states with the **same causal history,** they have the **same internal state – strongly eventually consistent**.

Great!!! But, what does it actually compute? Here, update(2) overwritten by update(4)! ☹

	state	query ( )	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:4, cnt:1	4	{0,1}
a3	sum:4, cnt:1	4	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}

Lessons Learned I

- Same causal history, different internal state
- Same causal history, same stale state but different internal state
- Same causal history, eventually same internal state
- Same causal history, always same internal state – SEC

	C?	EC?	SEC?
Average	no	no	no
NoMergeAverage	yes	no	no
BMergeAverage	yes	yes	no
MaxAverage	yes	yes	yes

Designing a **strongly eventually consistent state-based object** with intuitive semantics is challenging!

Lessons Learned II

- Replicated state-based object
- No convergence
- Convergence
- Eventual consistency in this model
- Strong eventual consistency in this model

Self-study Questions

- Can you design Average such that it becomes EC or SEC as well as offers correct averaging semantics?
- Think of other data structures and design update, query, and merge operations with reasonable semantics.
- Always draw timelines and state diagrams for your designs and proof EC or SEC, if possible.
- Think of data structures that support multiple update operations and one or more query operations.

Pixabay.comv

CRDT –
CONFLICT-FREE REPLICATED DATA TYPES

Conflict-Free Replicated Data Types

- CRDT is a conflict-free replicated state-based object
- CRDT handles concurrent writes
- **Intuition**:
  - Do not allow writes with arbitrary values, limit to write operations which are **guaranteed not to conflict**
  - CRDTs are data structures with **special** write operations; they guarantee **strong eventual consistency** and are monotonic (no rollbacks)
- CRDTs are no panacea but a great solution when they apply!

Conflict-Free Replicated Data Types

- CRDTs can be **commutative / op-based (CmRDT)**:
  - **Example**: A growth-only counter, which can only process *increment* operations
  - Propagate operations among replicas (**duplicate-free, no-loss messaging**)
- CRDTs can be **convergent / state-based (CvRDT)**:
  - **Example**: A max register, which stores the maximum value written
  - Propagate and merge states (idempotent)
- Therefore, the value of a CRDT depends on **multiple write operations or states**, not just the latest one

State-based CRDTs

- A CRDT is a replicated state-based object

- Supports
  - Query
  - Update
  - Merge

CRDT Properties

A CRDT is a replicated state-based object that satisfies

- Merge is **associative** (e.g., (A + (B + C)) = ((A + B) + C))
  - For any three state-based objects x, y, and z, merge(merge(x, y), z) is equal to merge(x, merge(y, z))
- Merge is **commutative** (e.g., A + B = B + A)
  - For any two state-based objects, x and y, merge(x, y) is equal to merge(y, x)
- Merge is **idempotent**
  - For any state-based object x, merge(x, x) is equal to x
- Every **update is increasing**
  - Let *x* be a state-based object and let *y* = update(*x*, ...) be the result of applying an update to *x*
  - Then, update is increasing if merge(*x*, *y*) is equal to y

Max Register is a CRDT

The state-based object IntMax is a CRDT

- `IntMax` wraps an integer
- `Merge(a, b)` is the max of a, b
- `Update(x)` adds x to the wrapped integer
- Prove that `IntMax` is associative, commutative, idempotent, increasing

```
class IntMax(object):
  def __init__(self):
    self.x = 0
  def query(self):
    return self.x
  def update(self, x):
    assert x >= 0
    self.x += x
  def merge(self,
      other):
    self.x =
      max(self.x,
        other.x)
```

Establish Four Properties of CRDT

- Associativity

```
merge(merge(a, b), c)
= max(max(a.x, b.x), c.x)
= max(a.x, max(b.x, c.x))
= merge(a, merge(b, c))
```

- Impotence

```
merge(a, a)
= max(a.x, a.x)
= a.x
= a
```

- Commutativity

```
merge(a, b)
= max(a.x, b.x)
= max(b.x, a.x)
= merge(b, a)
```

- Update is increasing

```
merge(a, update(a, x))
= max(a.x, a.x + x)
= a.x + x
= update(a, x)
```

G-Counter CRDT

Replicated growth-only counter

- Internal state of a G-Counter replicated on *n* nodes is an *n*-length array of non-negative integers

- `query` returns sum of every element in *n*-length array
- `add(x)` when invoked on the i-th server, increments the i-th entry of the *n*-length array by x
  - E.g., Server 0 increments 0th entry, Server 1 increments 1st entry of array, and so on
- `merge` performs a pairwise maximum of the two arrays

PN-Counter CRDT

Replicated counter supporting addition & subtraction

- Internal state of a PN-Counter
  - pair of two G-Counters named *p* and *n*.
    - *p* represents total value added to PN-Counter
    - *n* represents total value subtracted from PN-Counter.

- query method returns difference `p.query() - n.query()`
- `add(x)` –first of two updates– invokes `p.add(x)`
- `sub(x)` –second of two updates– invokes `n.add(x)`
- `merge` performs a pairwise merge of *p* and *n*

G-Set CRDT

Replicated growth-only set

A G-Set CRDT represents a replicated set which can be added to but not removed from

- Internal state of a G-Set is just a set
- `query` returns the set
- `add(x)` adds *x* to the set
- `merge` performs a set union

2P-Set CRDT

Replicated set supporting addition and subtraction

- Internal state of a 2P-Set is a
  - pair of two G-Sets named *a* and *r*
    - *a* represents set of values added to the 2P-Set
    - *r* represents set of values removed from the 2P-Set
- `query` method returns the set difference `a.query() - r.query()`
- `add(x)` is the first of two updates
  - invokes a.add(x).
- `sub(x)` is the second of two updates
  - invokes `r.add(x)`
- `merge` performs a pairwise merge of *a* and *r*

Summary on CRDTs

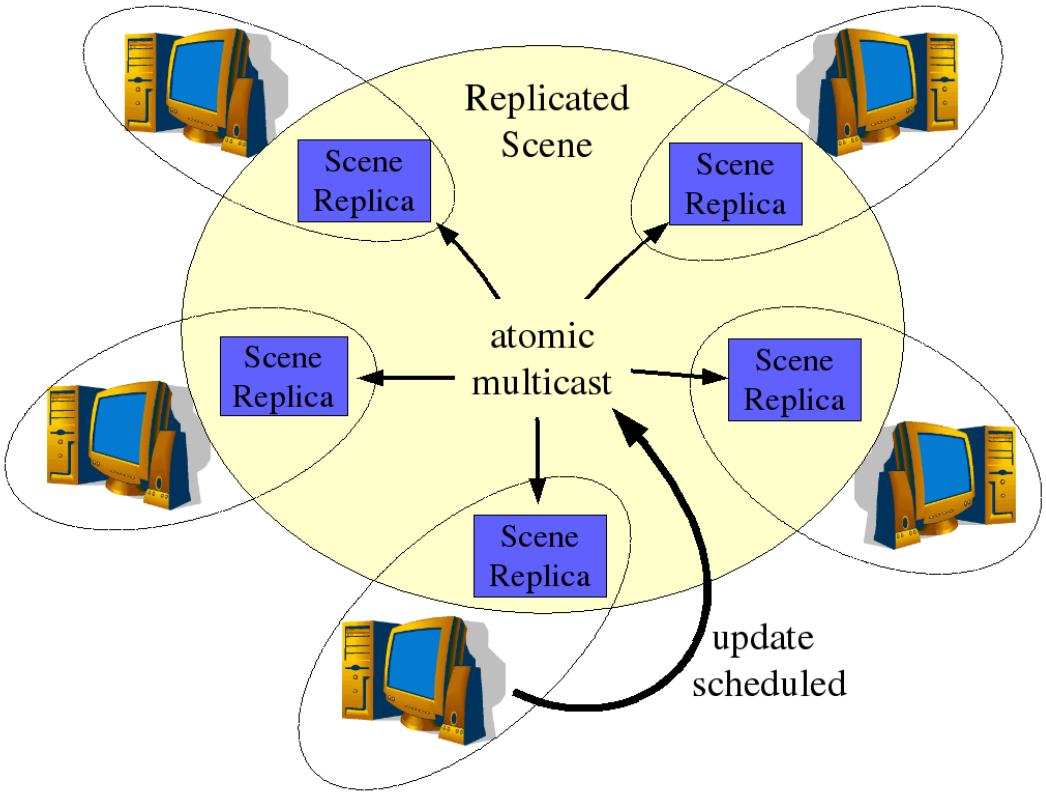
- Formalized and introduced in 2014
- CmCRDTs and CvCRDTs are equivalent!
- Really neat solution if it applies
- Challenge is to design new CRDTs

Self-study Questions

- For the CRDTs introduced, establish its four properties.
- Create example executions for each CRDT and complete a timeline and a state table.
- Fine use cases where the introduced CRDTs apply and show how they are used.
- Think of new CRDTs and repeat the above.

Pixabay.com

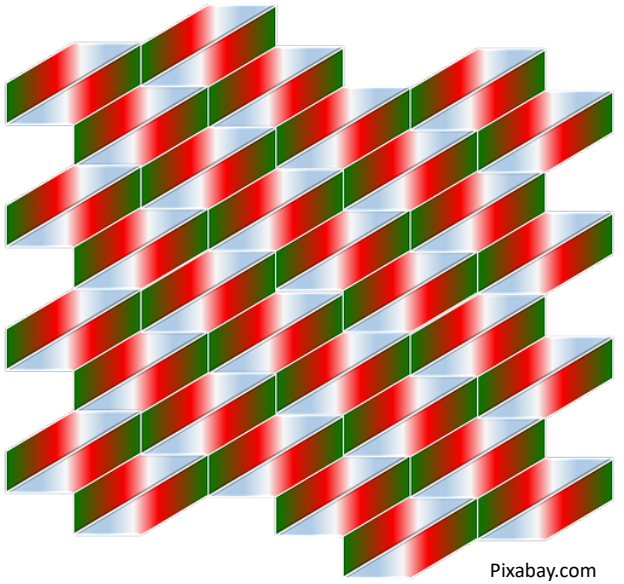
Replication



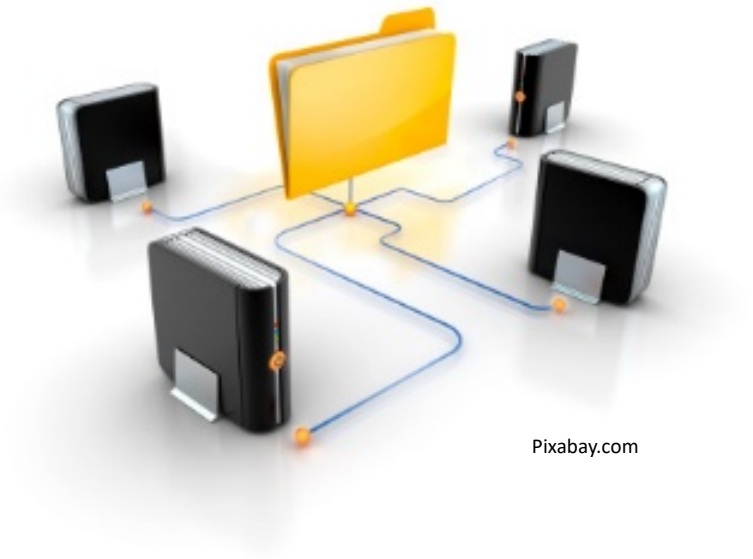
https://en.wikipedia.org/wiki/File:Active_Replication_Consistency_Model.png

Outline

- Data replication
- Replication patterns
- Chain replication
- Gossiping



Pixabay.com



Pixabay.com

DATA REPLICATION

Why Replicate?

- Performance

- High availability

    - Fault tolerance

Why Replicate?

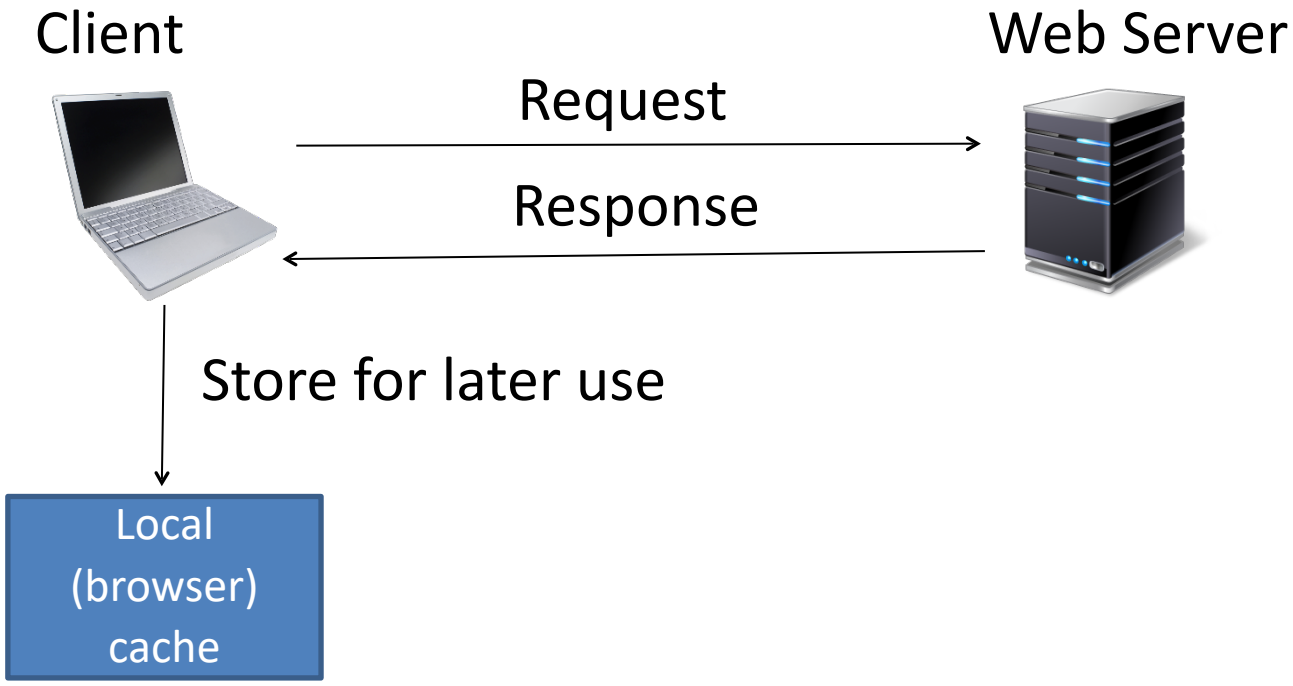
- Performance
- High availability
- Fault tolerance

Here, we look at **data replication**.

Computation replication is another concept (cf. MapReduce)

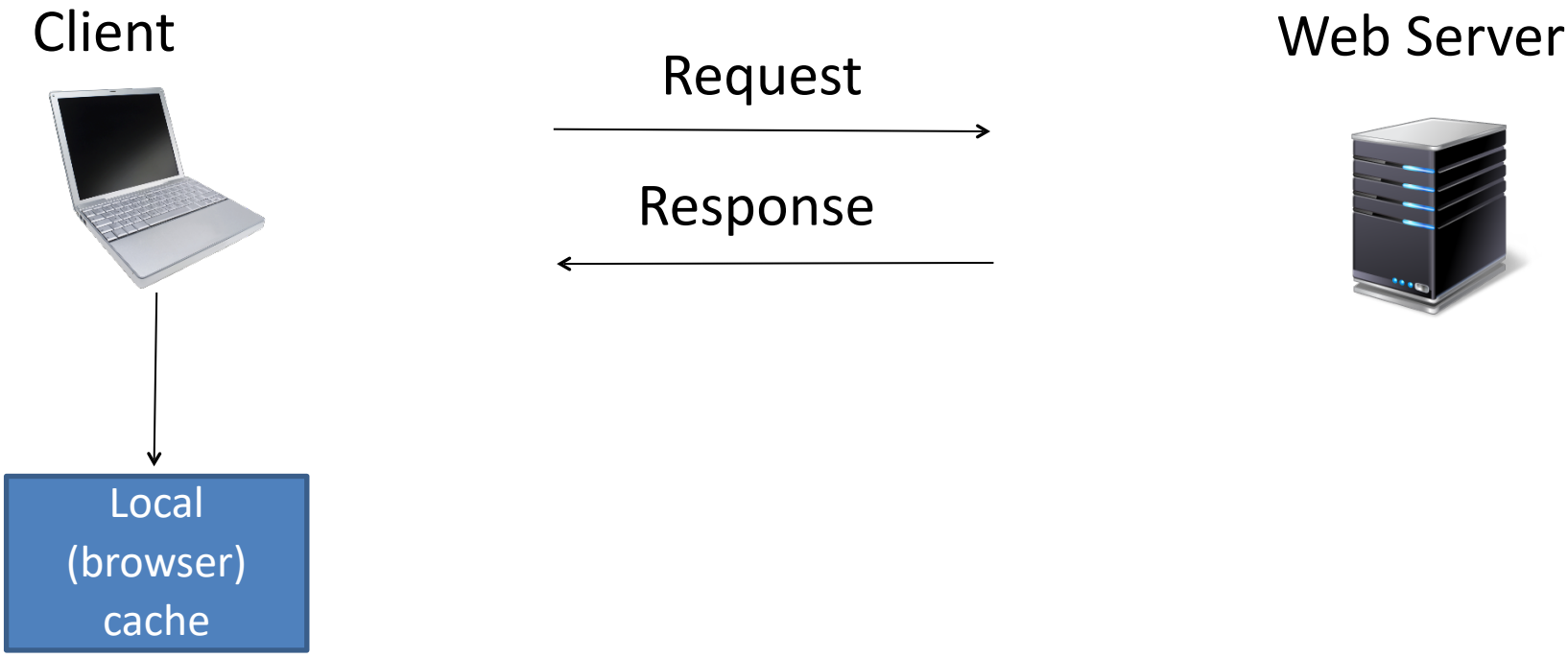
Performance

Caching data at **browsers** and **proxy servers**



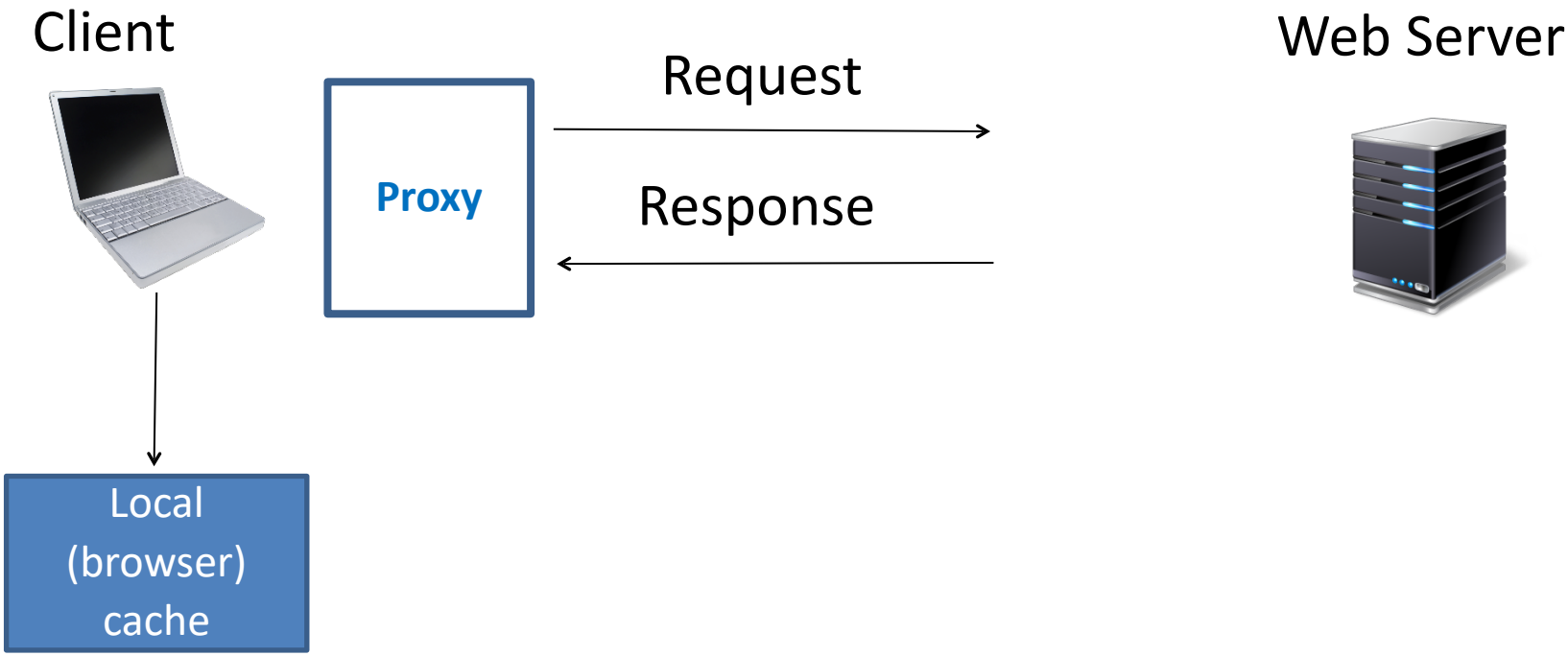
Performance

Caching data at **browsers** and **proxy servers**



Performance

Caching data at **browsers** and **proxy servers**



Caching data a

Client

Proxy

Local (browser) cache

Configure Proxy Access to the Internet

- No proxy
- Auto-detect proxy settings for this network
- Use system proxy settings
- Manual proxy configuration

HTTP Proxy — Port 0

Also use this proxy for FTP and HTTPS

HTTPS Proxy — Port 0

FTP Proxy — Port 0

SOCKS Host — Port 0

SOCKS v4 — SOCKS v5

- Automatic proxy configuration URL

Reload

No proxy for

Example: .mozilla.org, .net.nz, 192.168.1.0/24

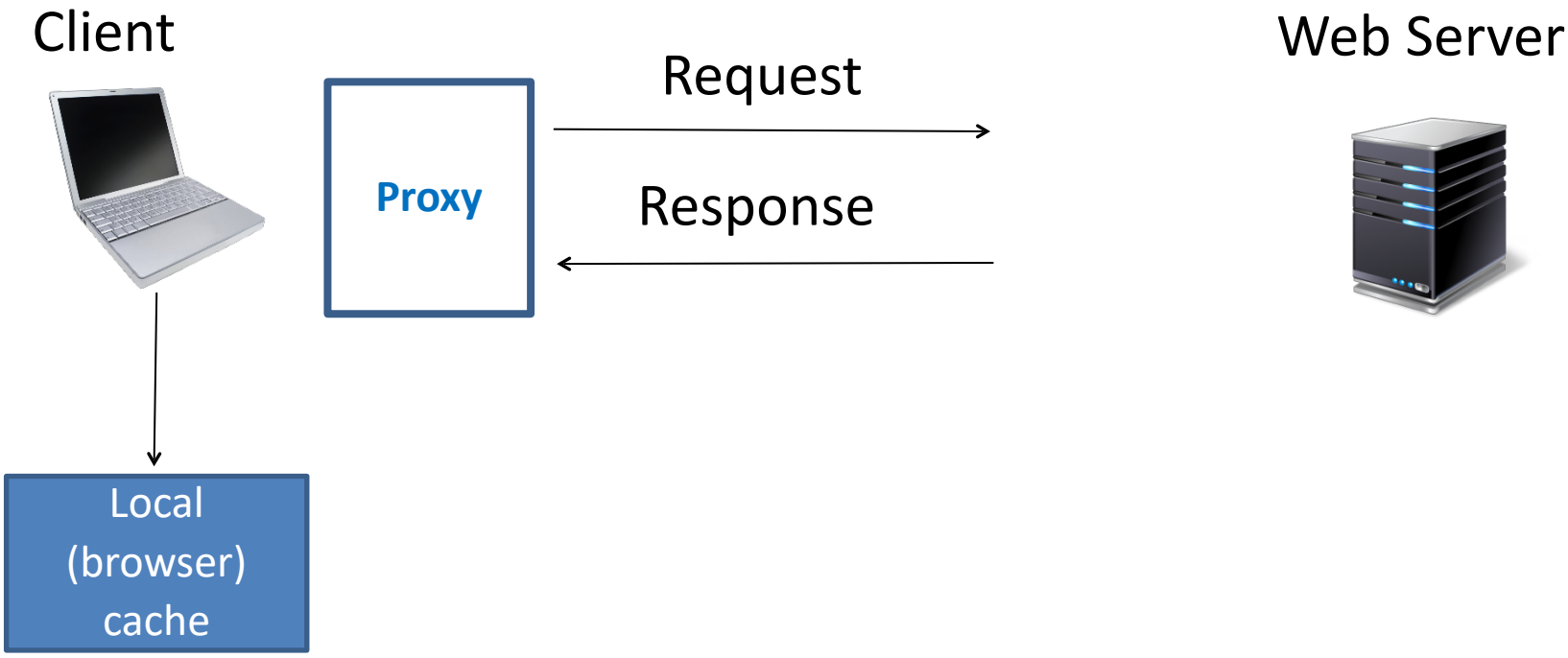
Connections to localhost, 127.0.0.1, and ::1 are never proxied.

- [ ] Do not prompt for authentication if password is saved
- [ ] Proxy DNS when using SOCKS v5
- [x] Enable DNS over HTTPS

Use Provider: Cloudflare (Default)

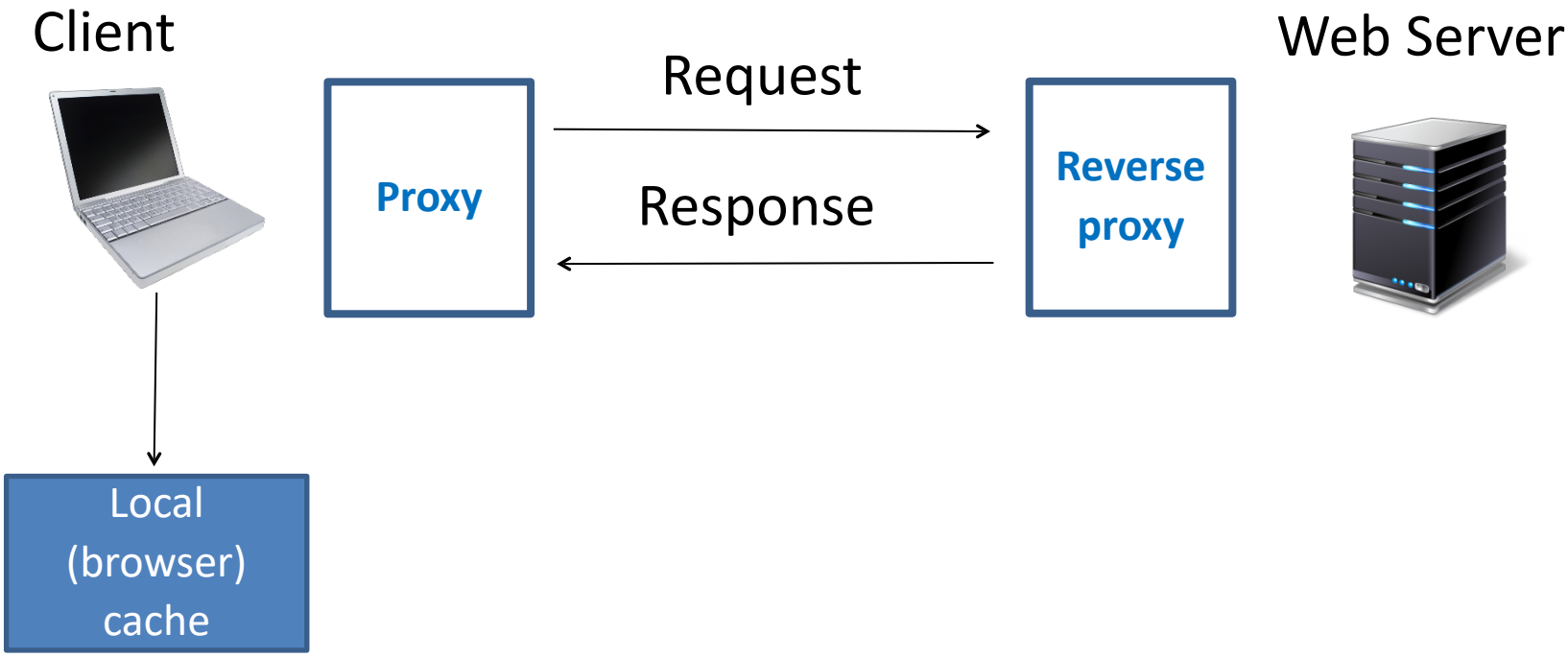
Performance

Caching data at **browsers** and **proxy servers**



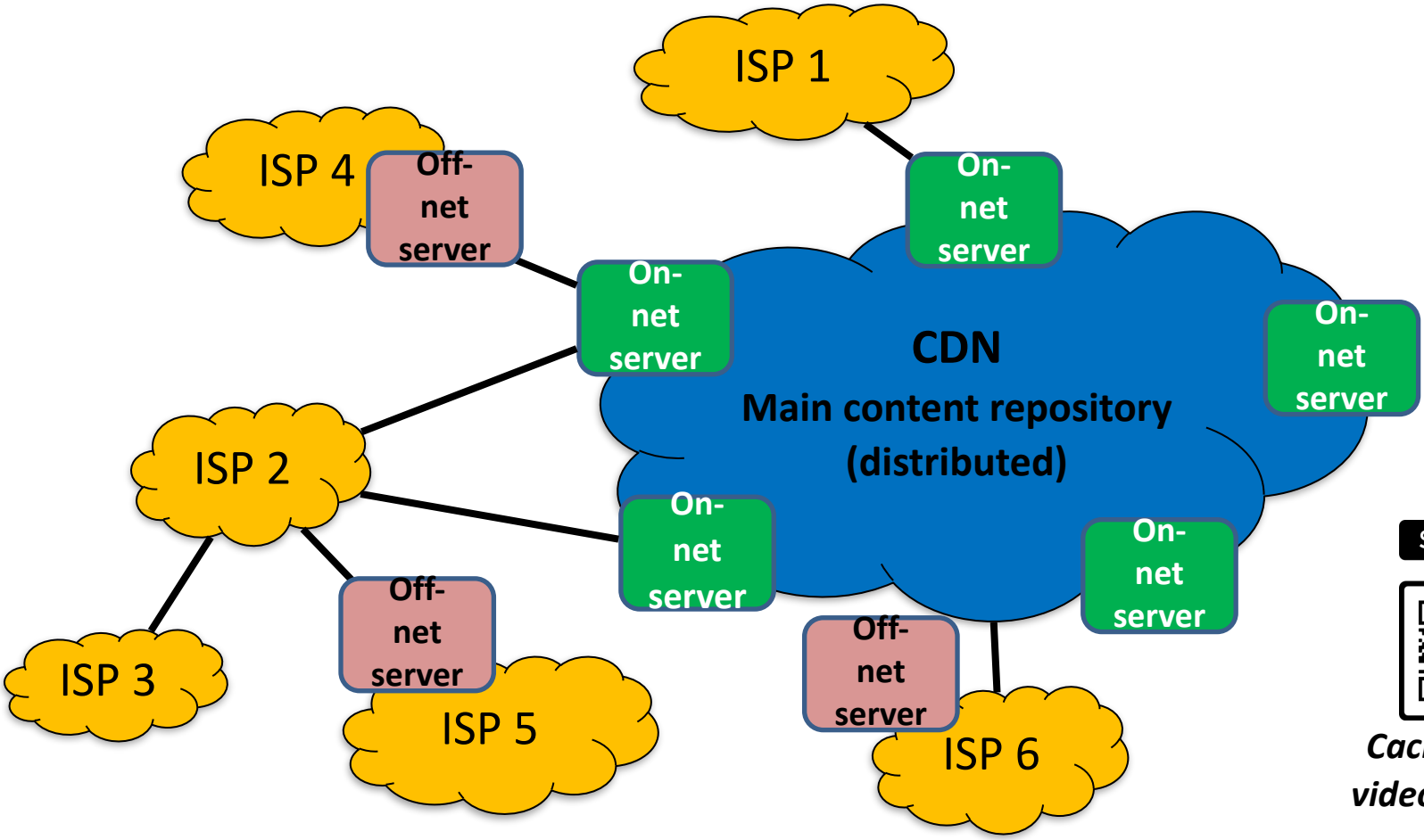
Performance

Caching data at **browsers** and **proxy servers**



Typical (video) CDN

(Content Delivery Network)

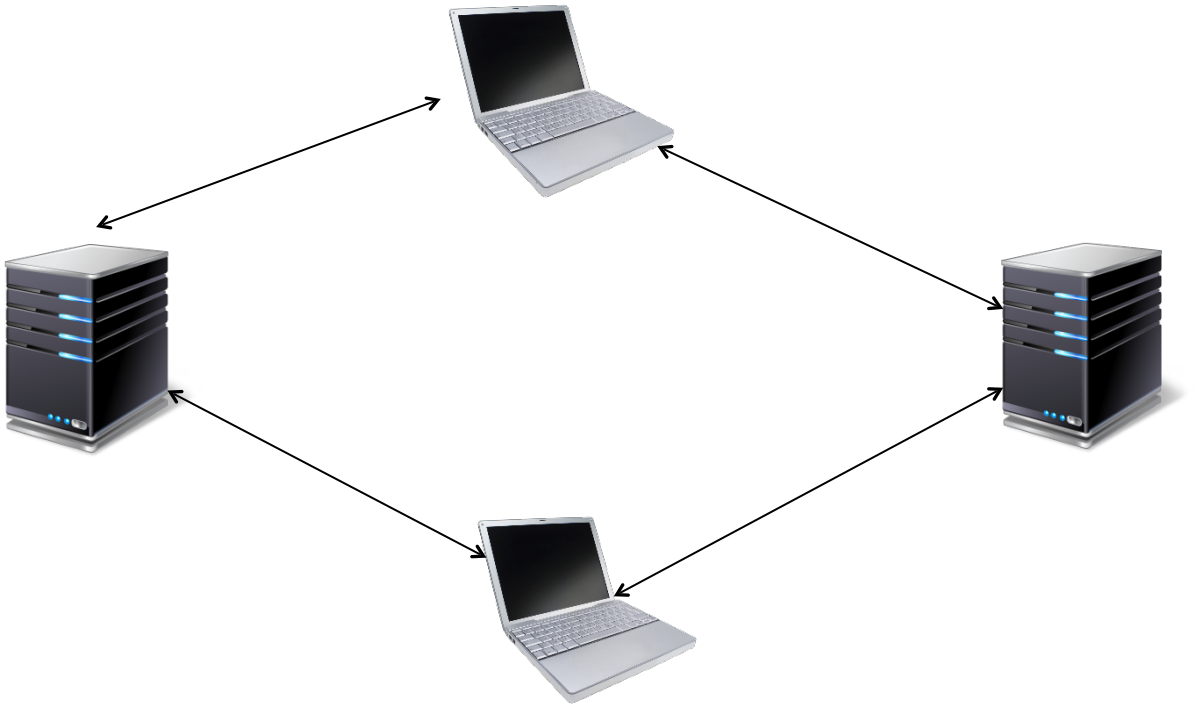


SCAN ME

Caching in video CDNs

High Availability

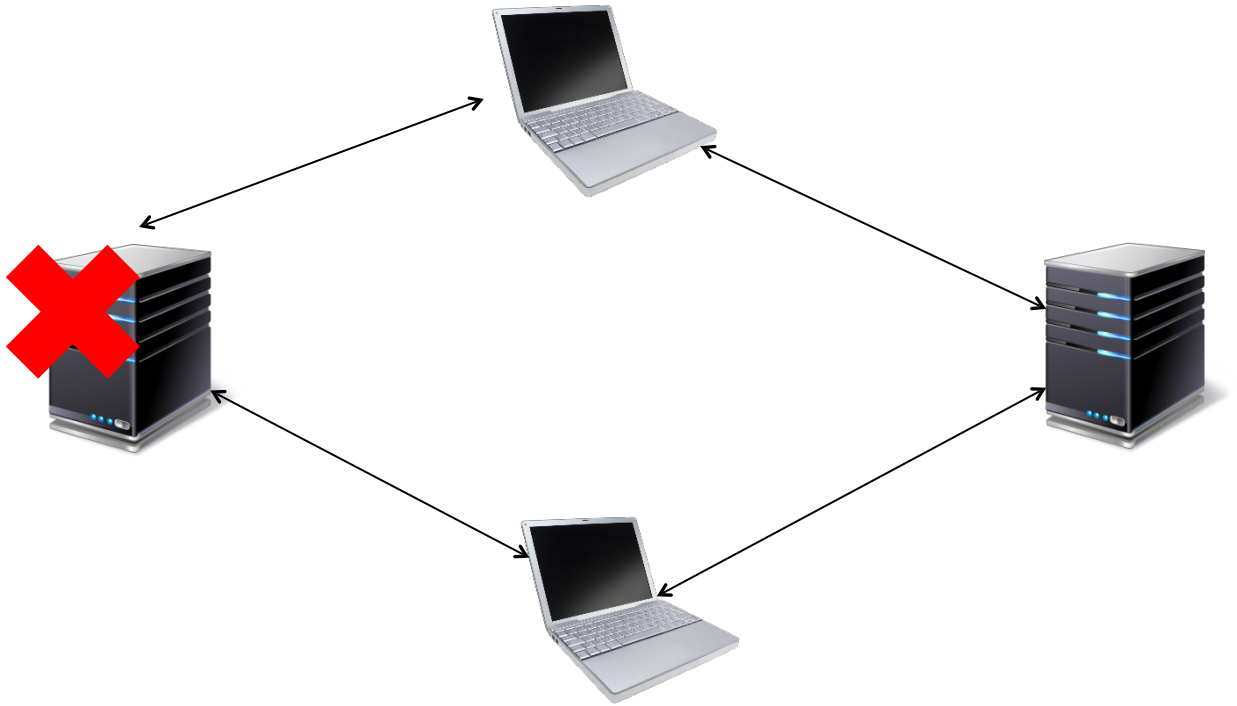
Upon crashes, data offered/retrieved by/from replica



[1] The availability of a service by replicating its nodes grows.

High Availability

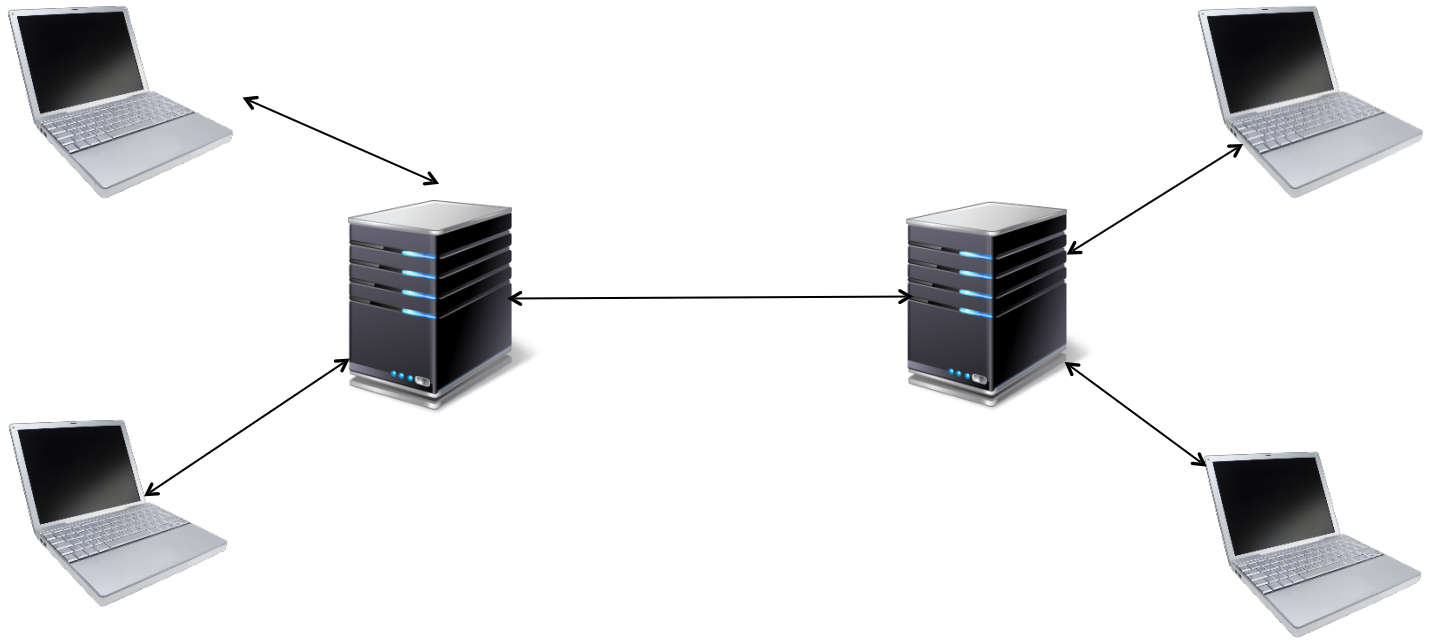
Upon crashes, data offered/retrieved by/from replica



[1] The availability of a service by replicating its nodes grows.

High Availability: Network Partitioning

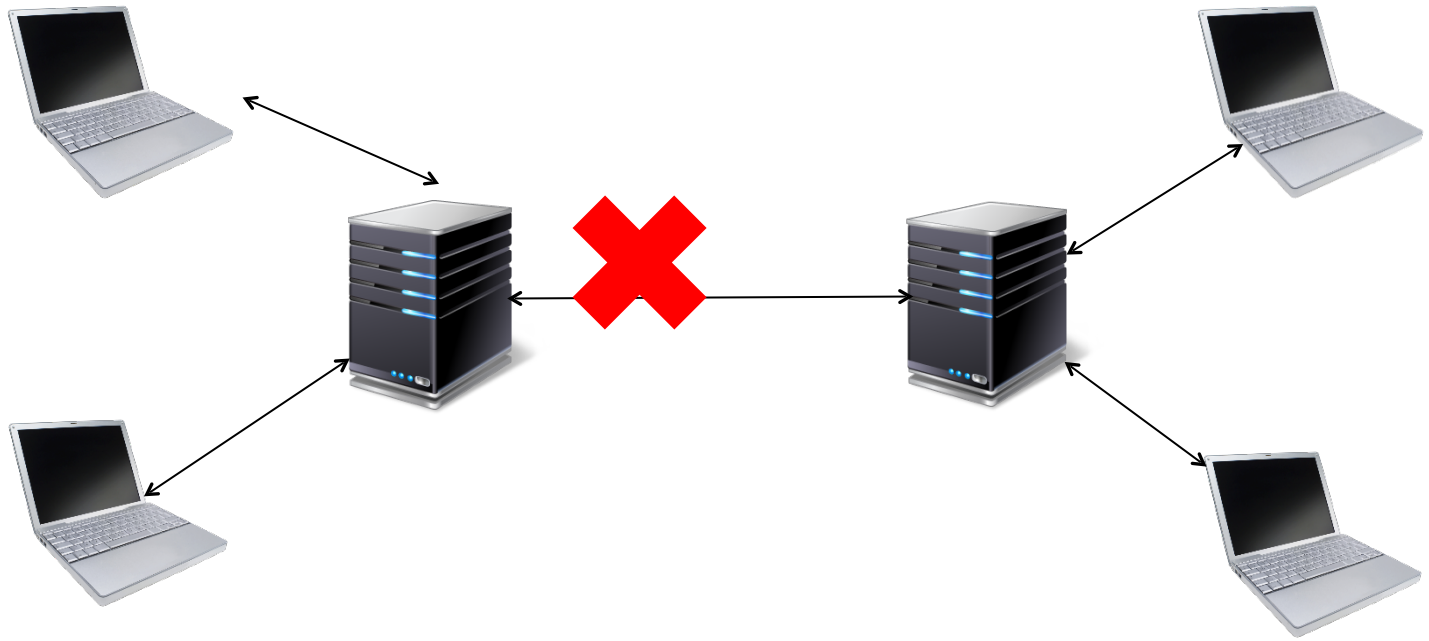
Upon network partitioning, data available to clients from within partition



Partition tolerance: A system that continues to operate in face of network partitions

High Availability: Network Partitioning

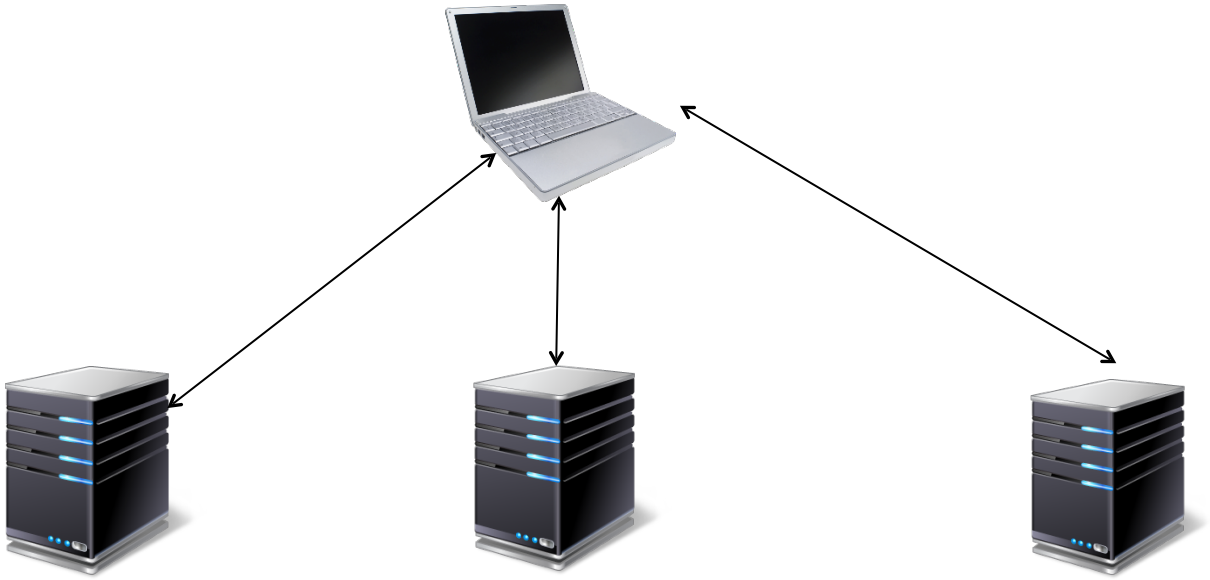
Upon network partitioning, data available to clients from within partition



Partition tolerance: A system that continues to operate in face of network partitions

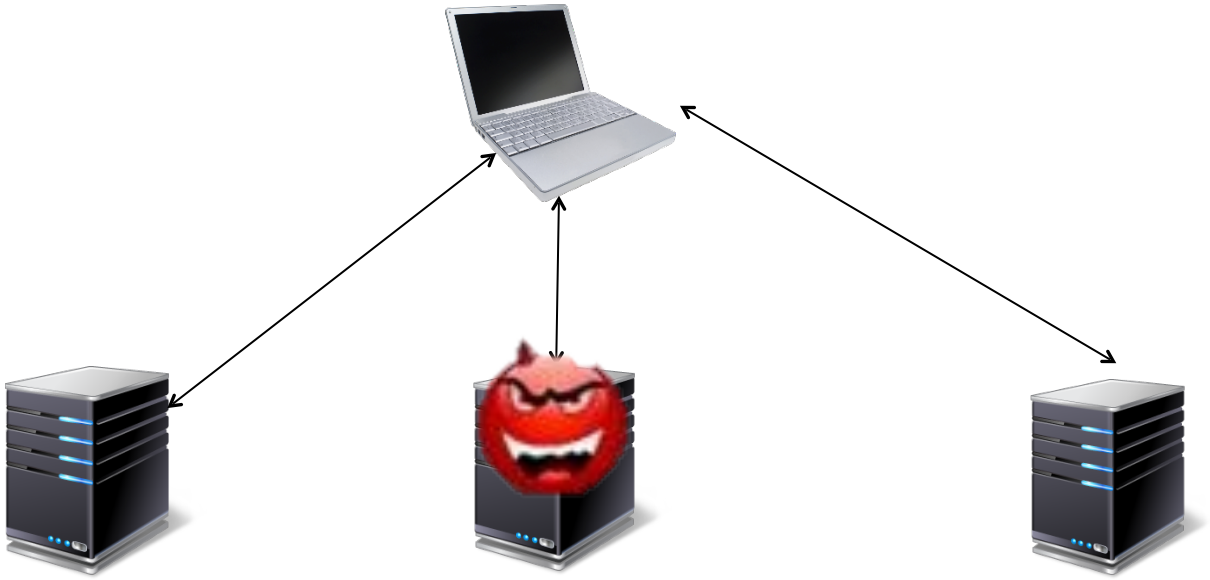
High Availability: Fault Tolerance

Providing reliable service in face of faulty nodes
(not just crashes, but also arbitrary failures!)



High Availability: Fault Tolerance

Providing reliable service in face of faulty nodes
(not just crashes, but also arbitrary failures!)



“Cost” of Replication

- Not just cost of **storing** additional copies of data

- Cost to **keep replicas up to date** in face of updates

- *How to deal with **stale** (out-of-date) **data at replicas?***

Self-study Questions

- What is the “cache hierarchy” for web data? Think of web pages with varying complexity of content; understand where each content element is cached.
- Is this cache hierarchy the same for any distributed data processing system?
- Find some popular examples of open-source caching technology and understand their position in the cache hierarchy.
- Think of alternatives for recovering a crashed replica – how does it catch up to the current state?
- Determine conflicts when issuing reads and writes concurrently to replicas.

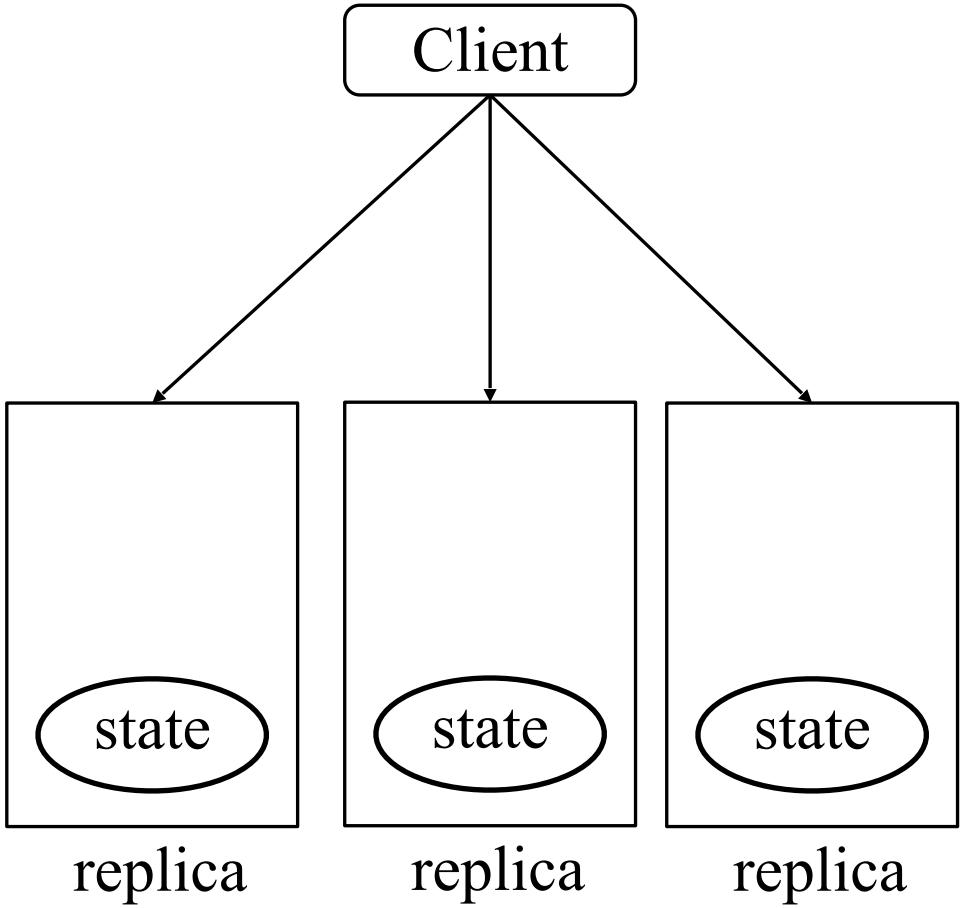
Pixabay.com

REPLICATION PATTERNS

WITH UPDATE PROCESSING

Replication Pattern

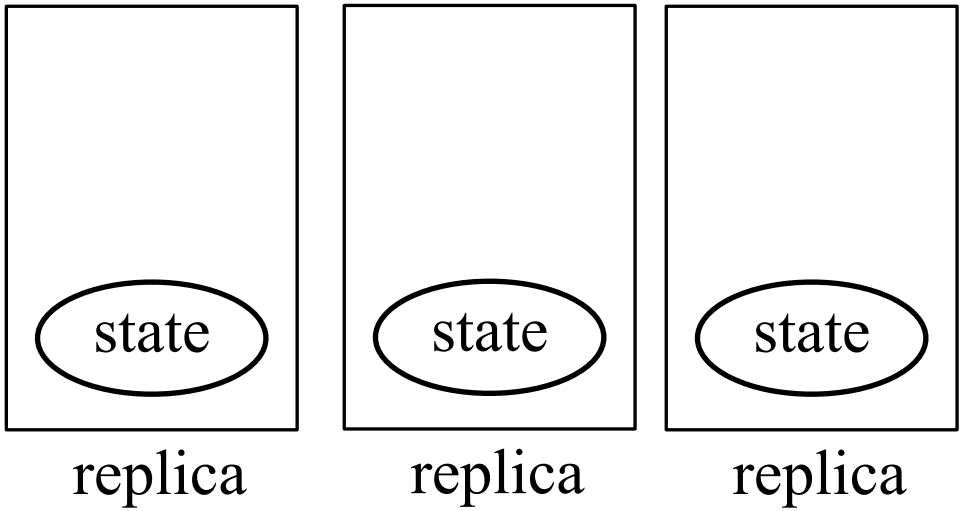
- **Active replication**
- Requests can be reads or writes
- Replicas can crash



Replication Pattern

- **Active replication**

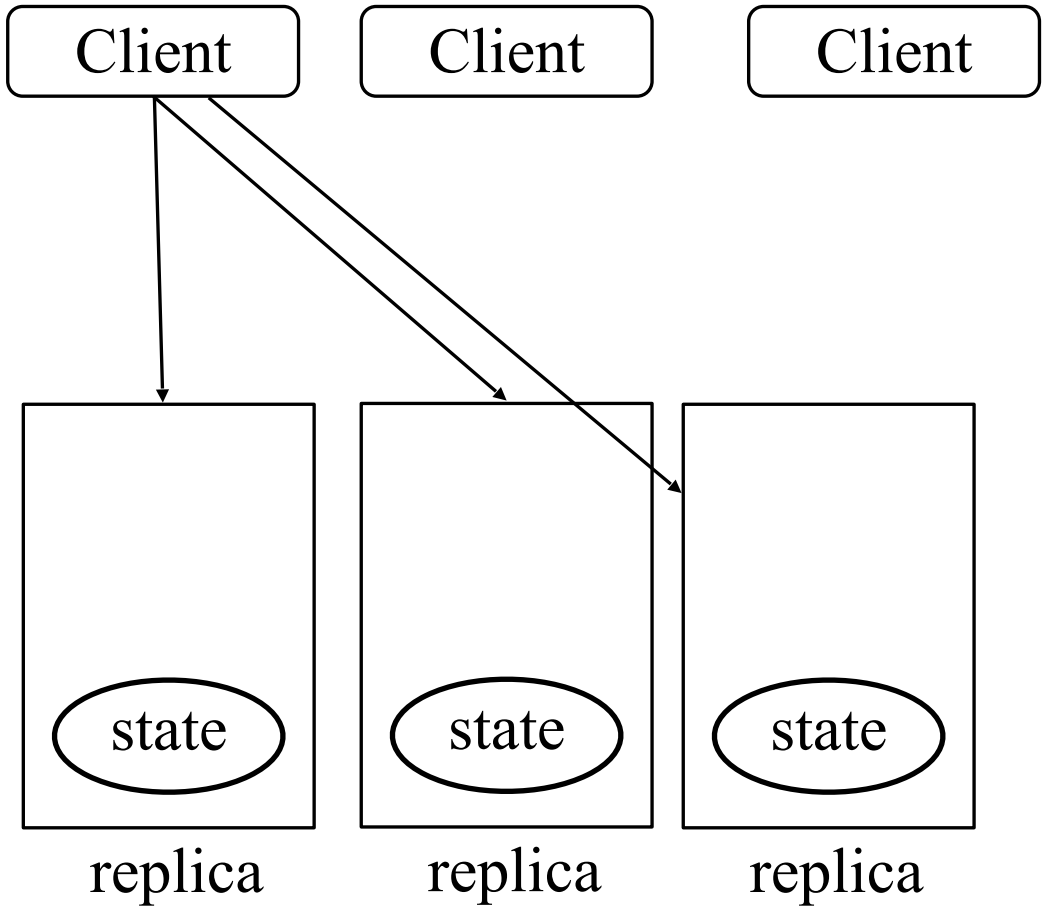
- Clients send requests to every replica



Replication Pattern

- **Active replication**

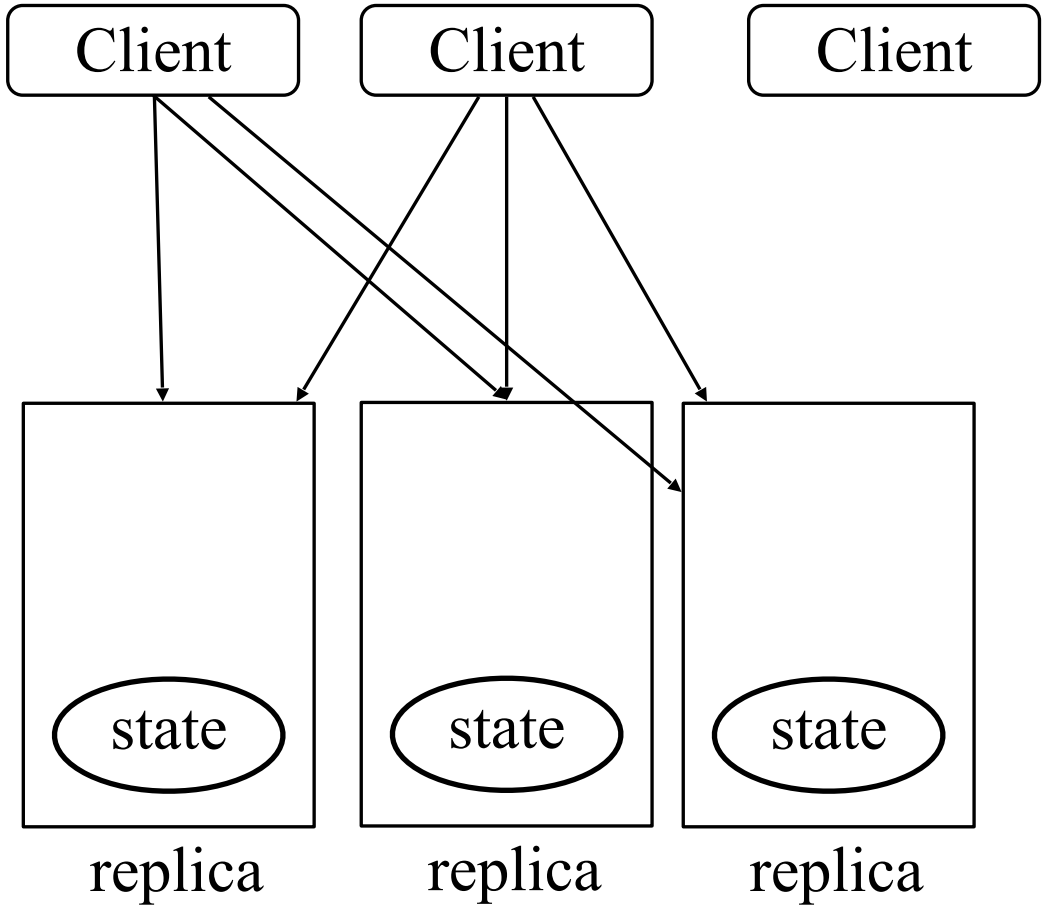
- Clients send requests to every replica



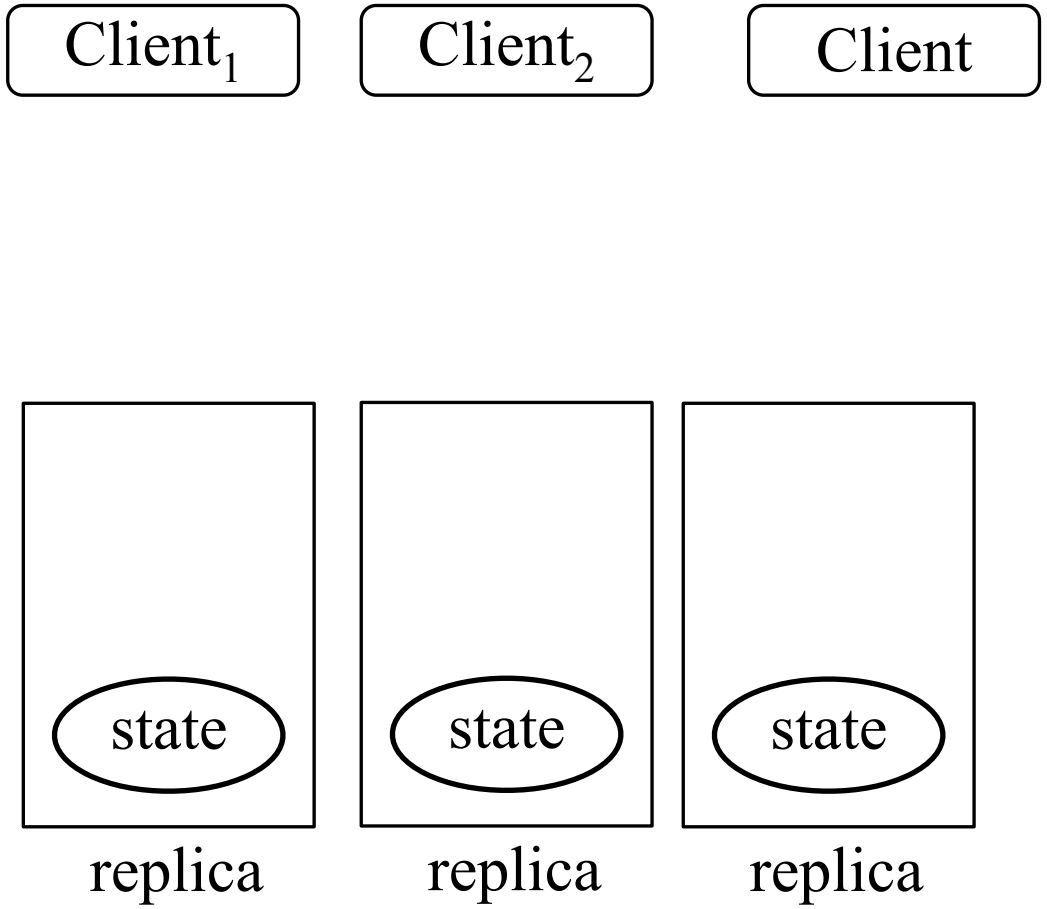
Replication Pattern

- **Active replication**

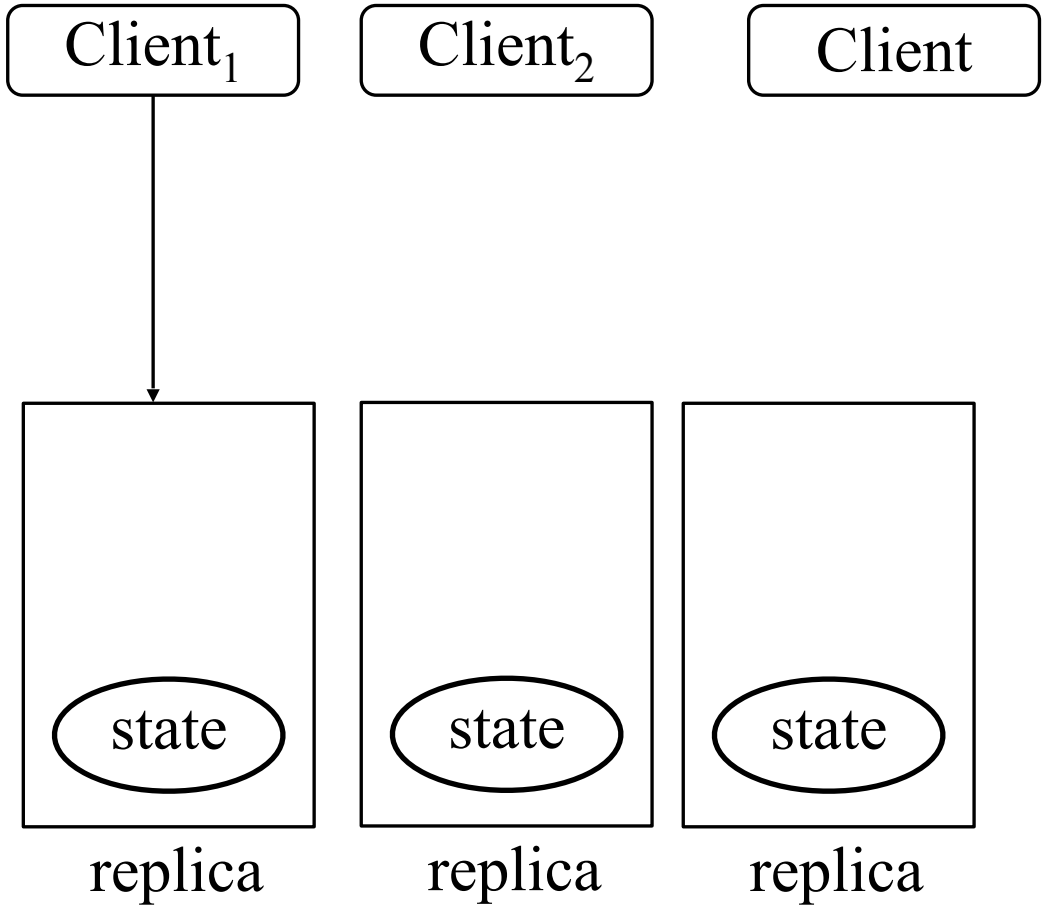
- Clients send requests to every replica



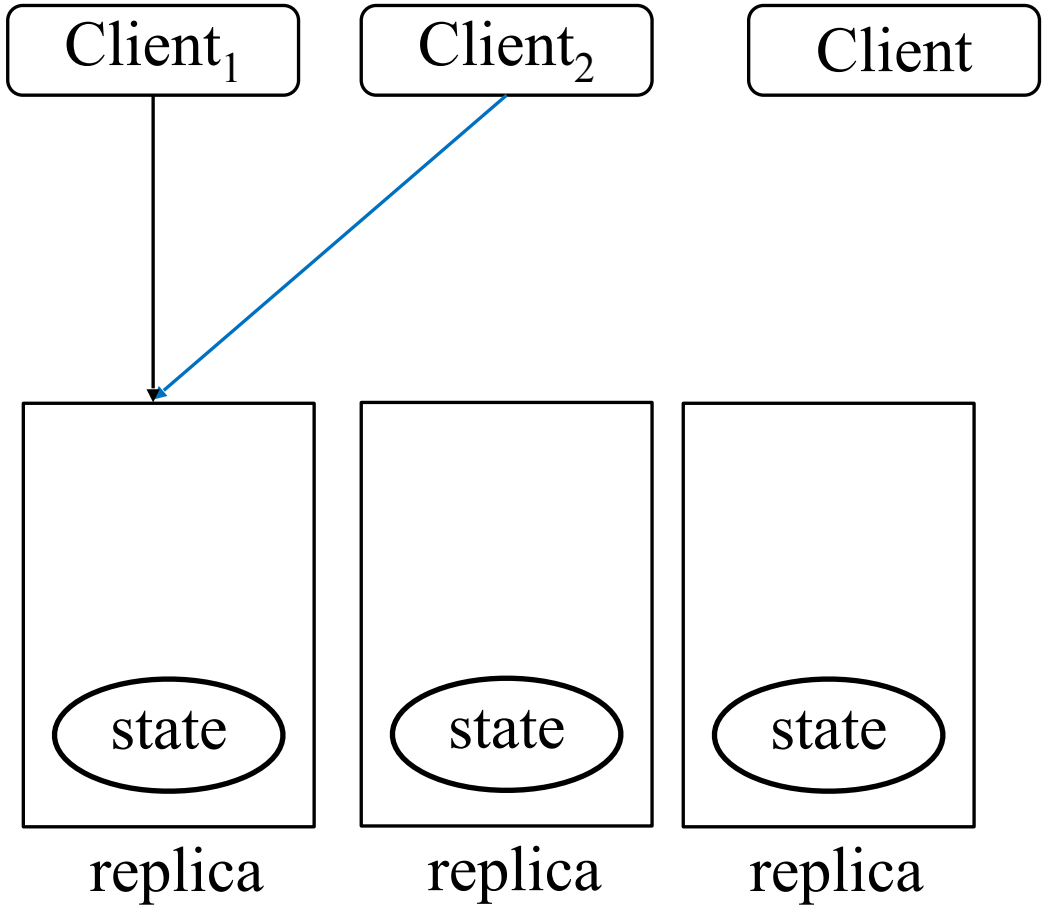
Replication Pattern



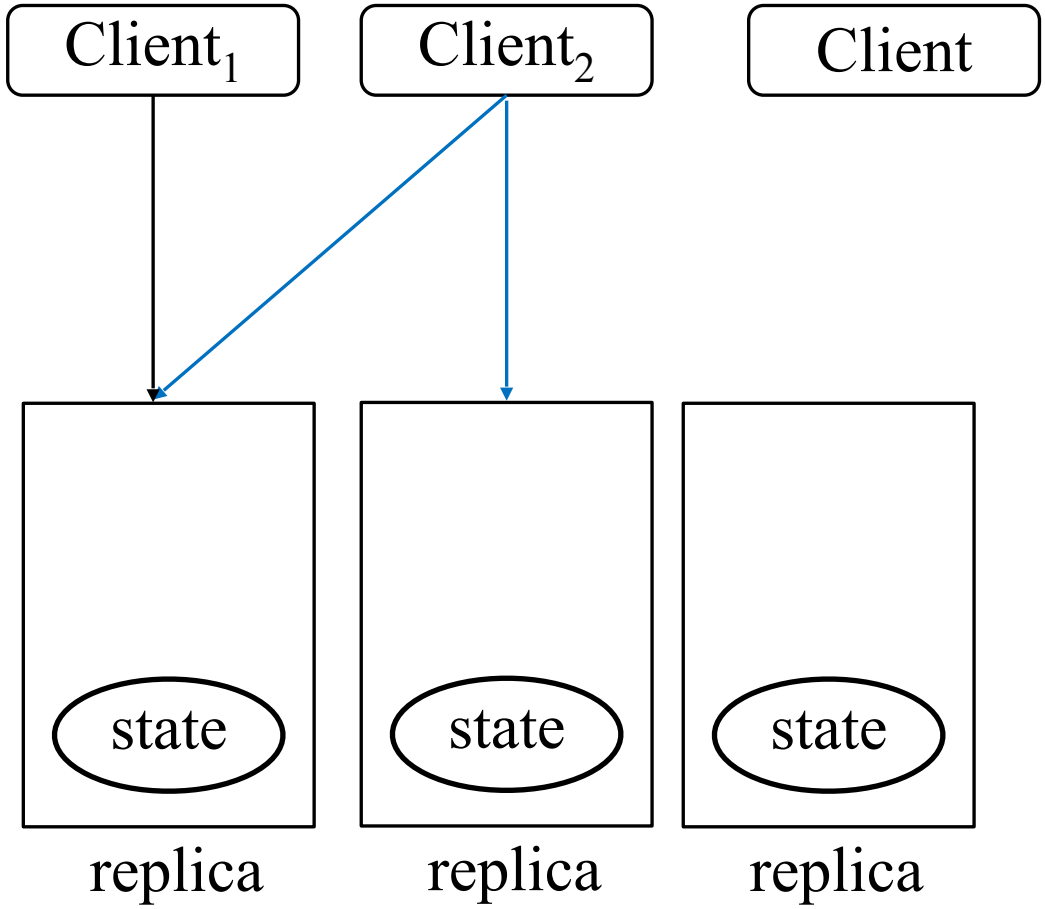
Replication Pattern



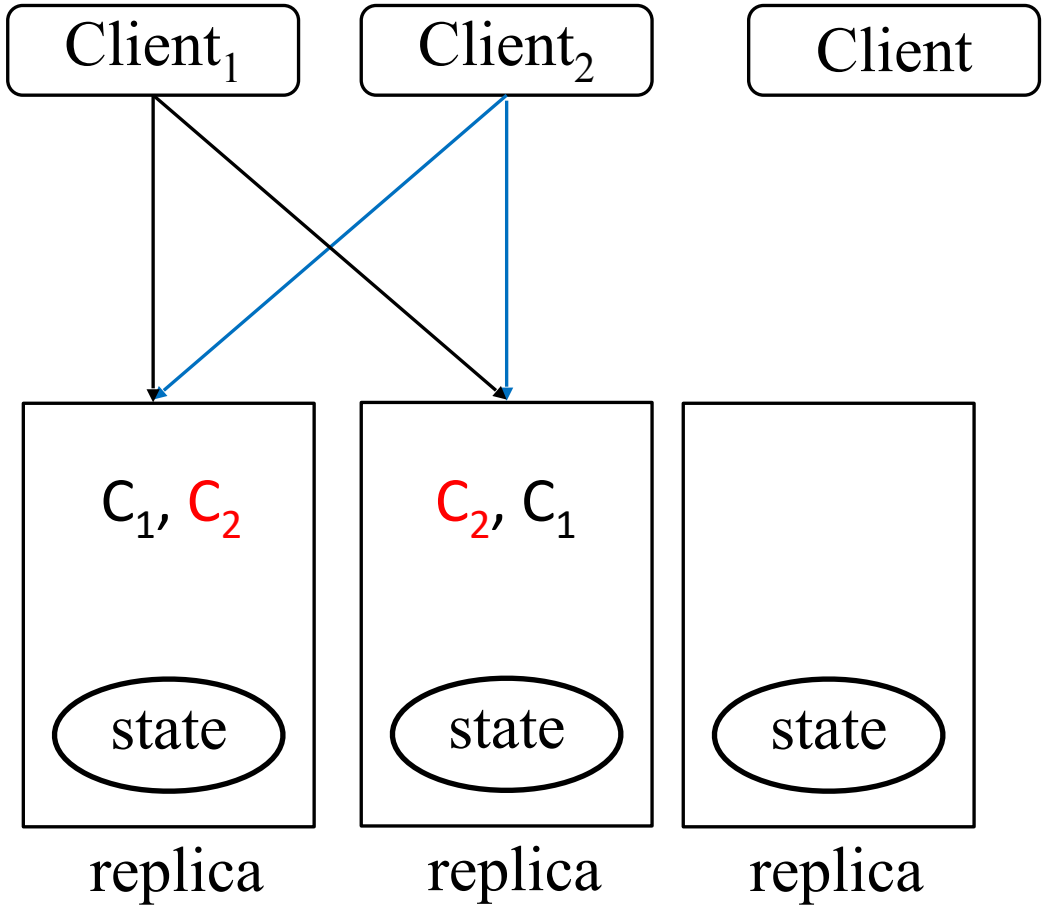
Replication Pattern



Replication Pattern

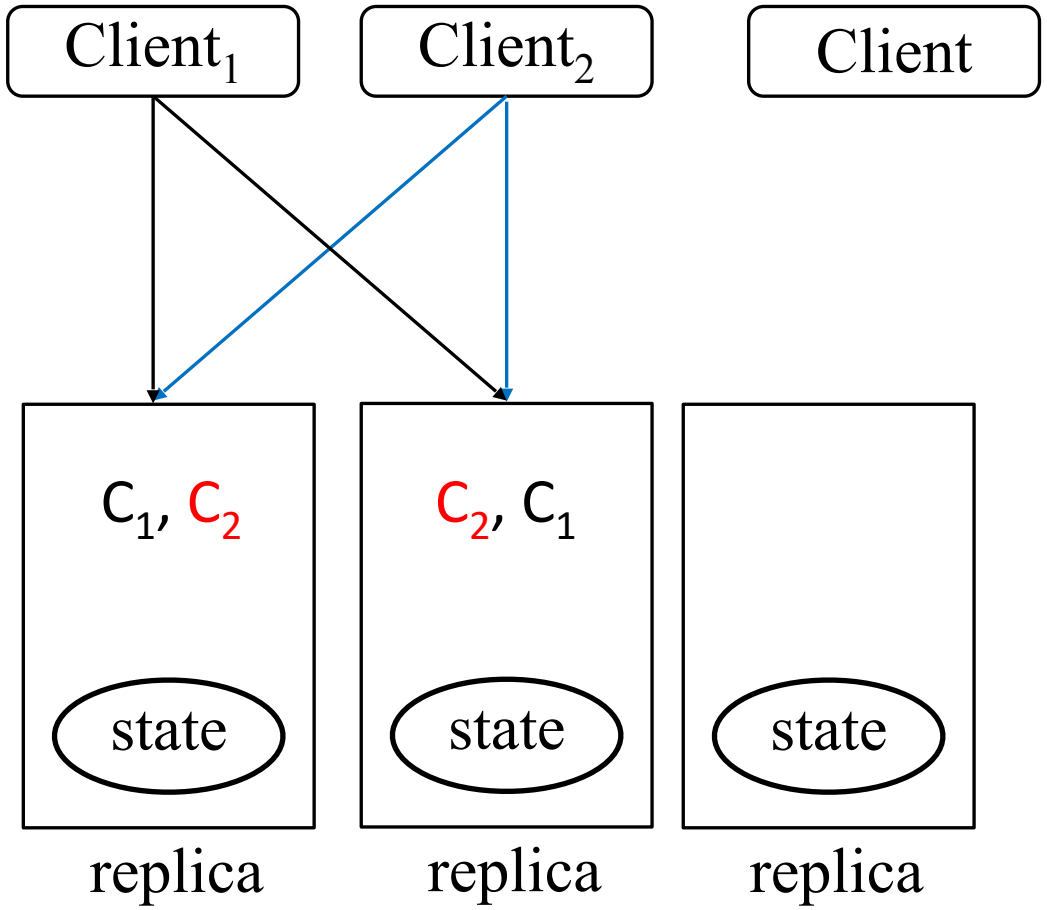


Replication Pattern



Replication Pattern

- **Active replication**
- Replicas may diverge



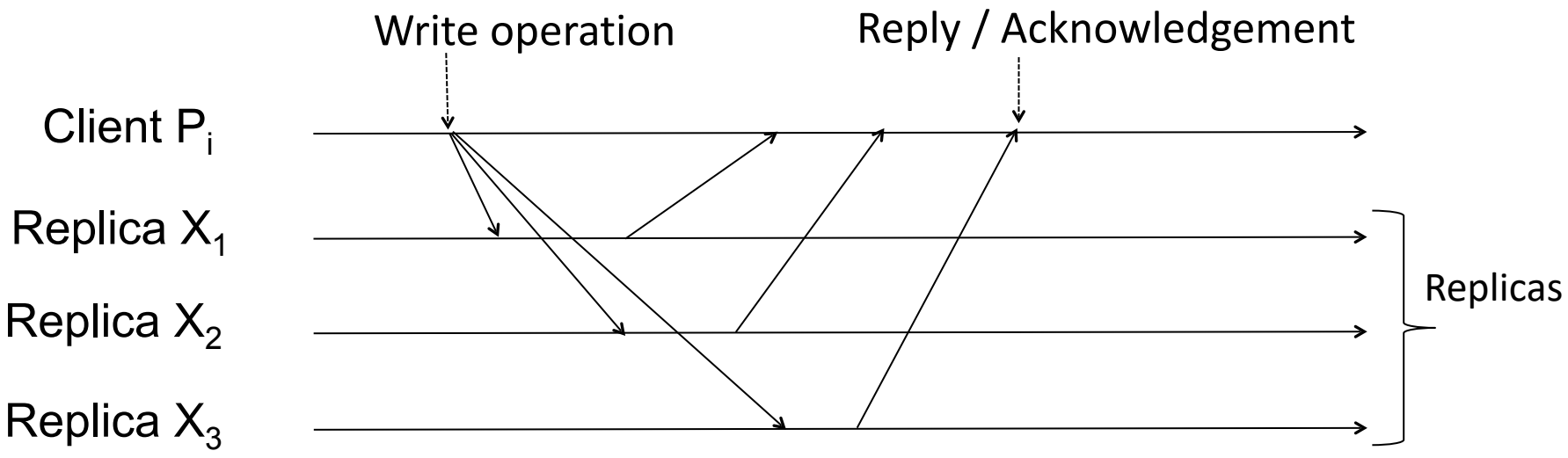
Active Replication

- Requires **total order broadcast** to guarantee each replica receives
  - **all** requests (from **all clients**)
  - **in the same order**

- Like client-server, “natural” to think about

- Fast to get a response, first result received, unless Byzantine failure assumptions (majority result)

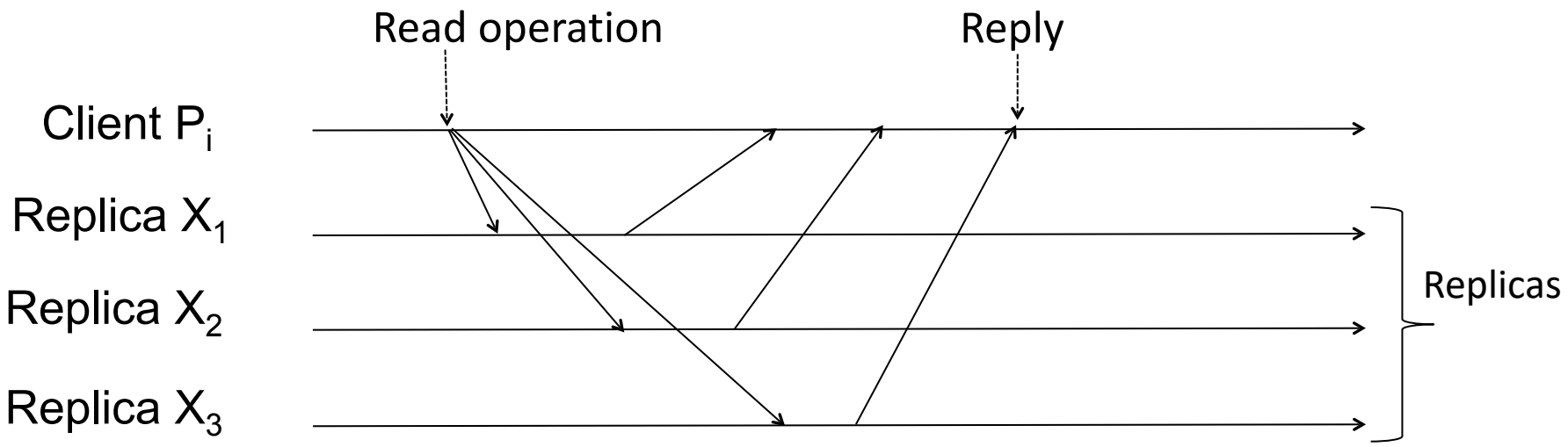
Active Replication



Configuration service:

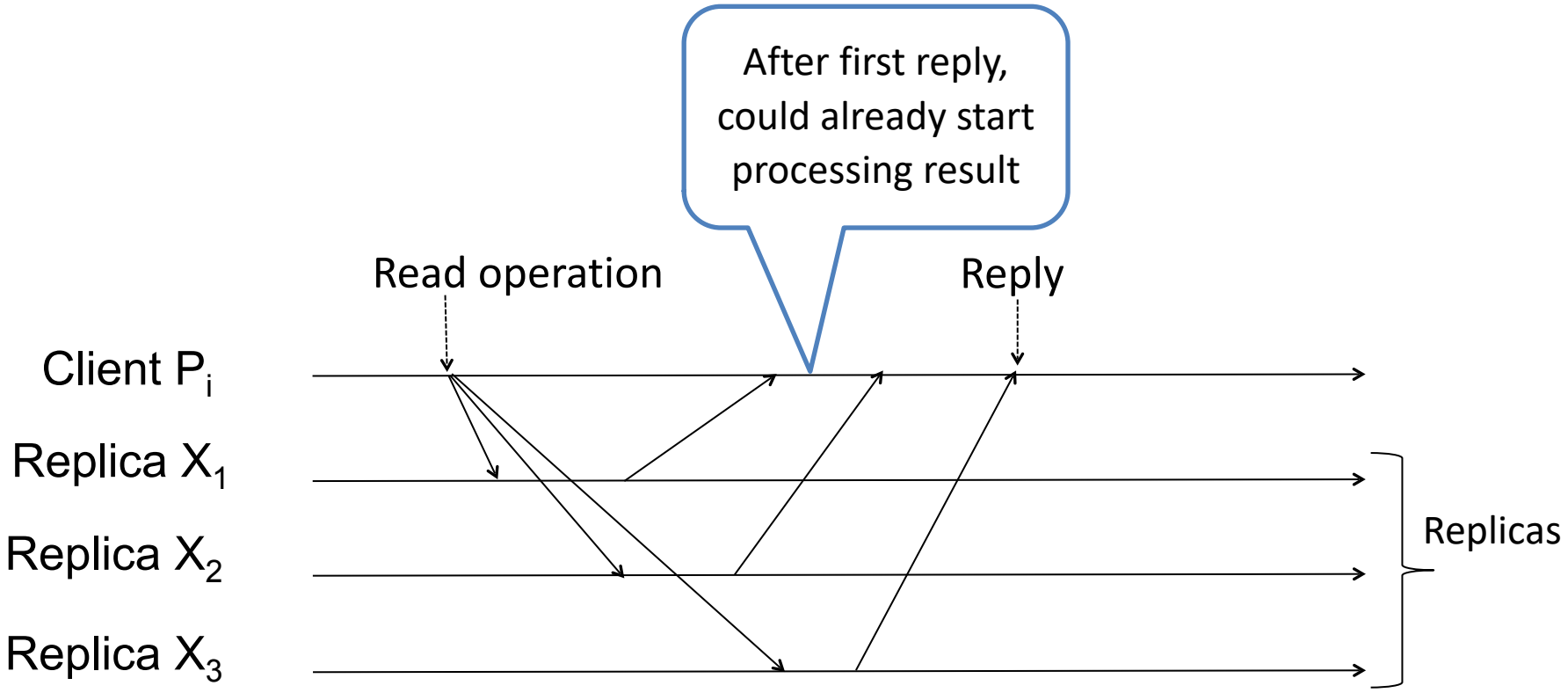
- Failure detection
- Configuration management

Active Replication



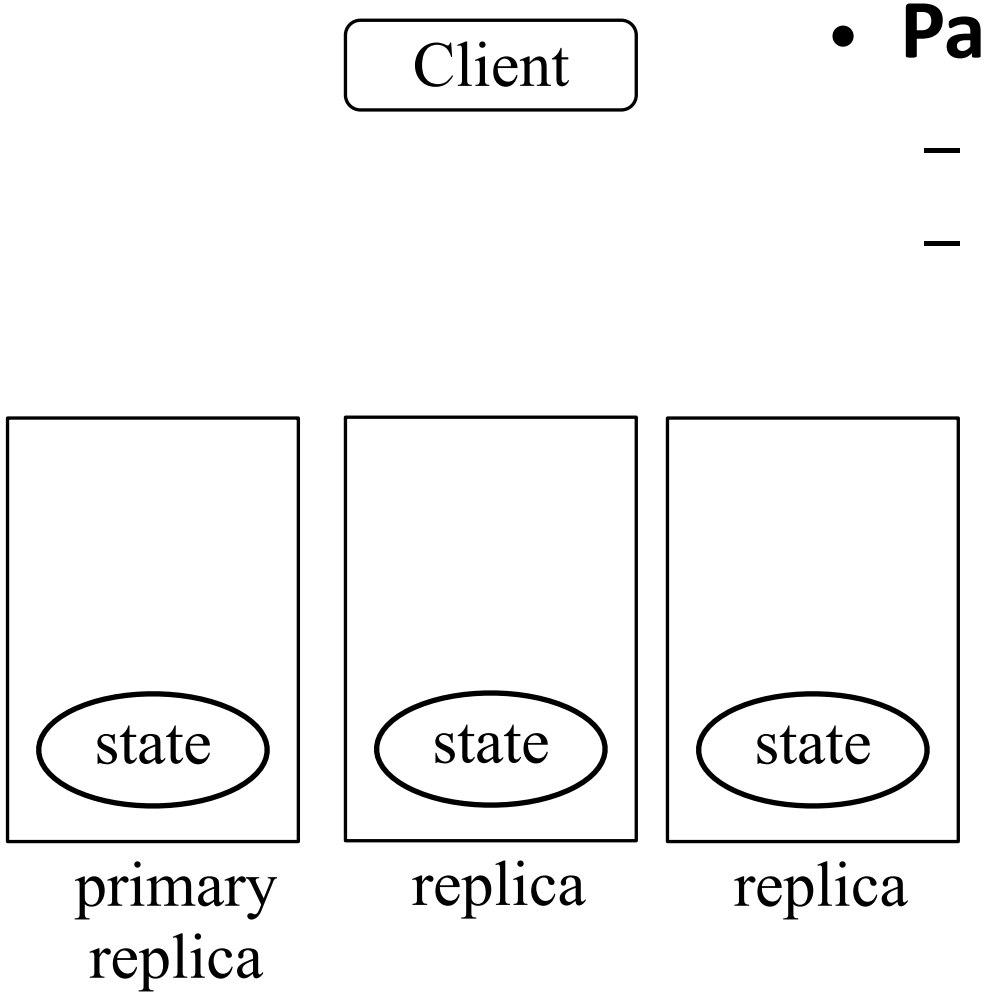
Alternative: Read from a quorum

Active Replication



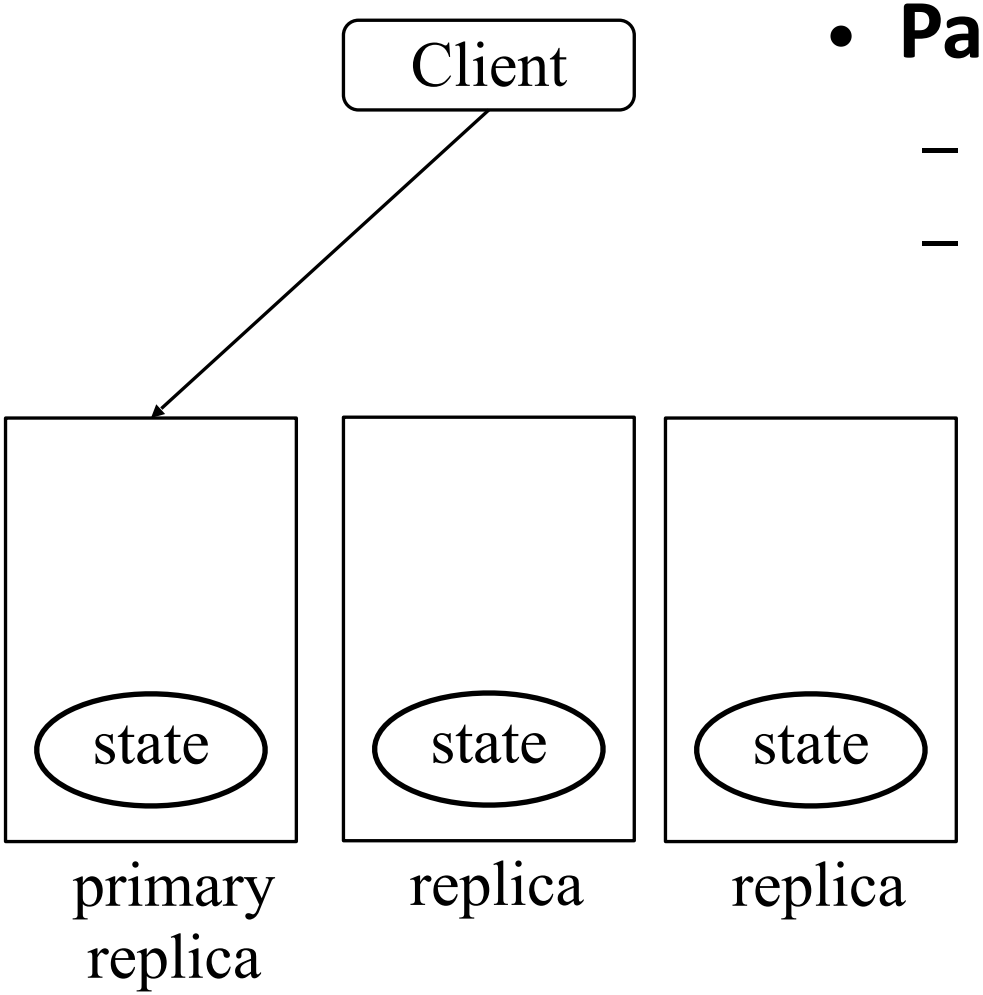
Alternative: Read from a quorum

Replication Pattern



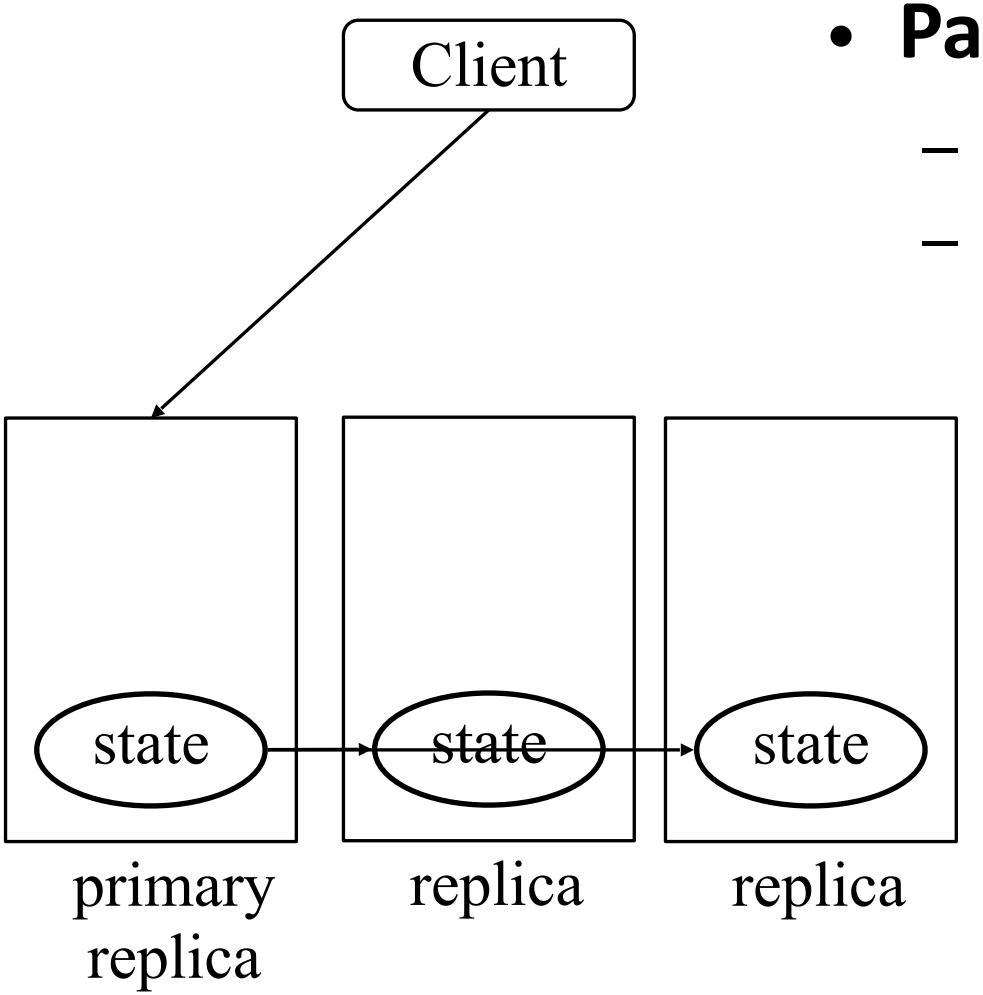
- **Passive replication**
  - **Primary-backup**
  - Multi-primary

Replication Pattern



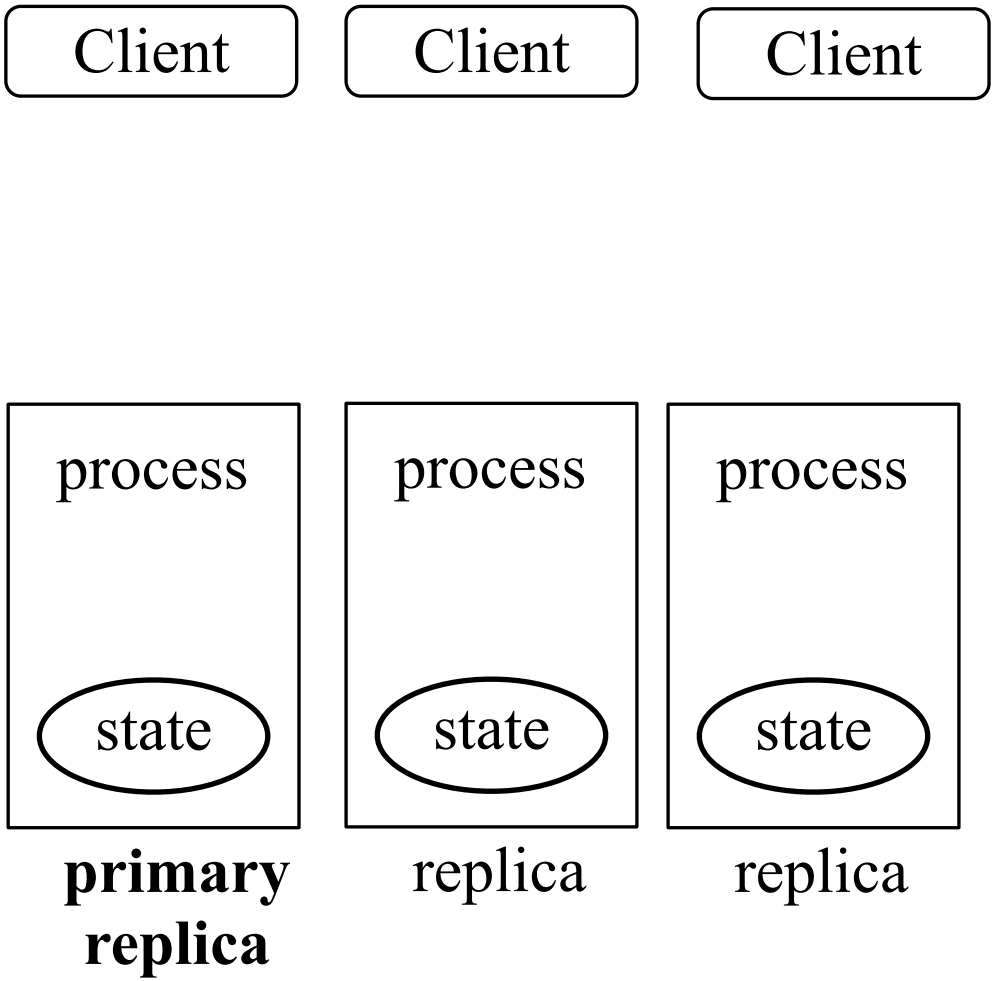
- **Passive replication**
  - **Primary-backup**
  - Multi-primary

Replication Pattern

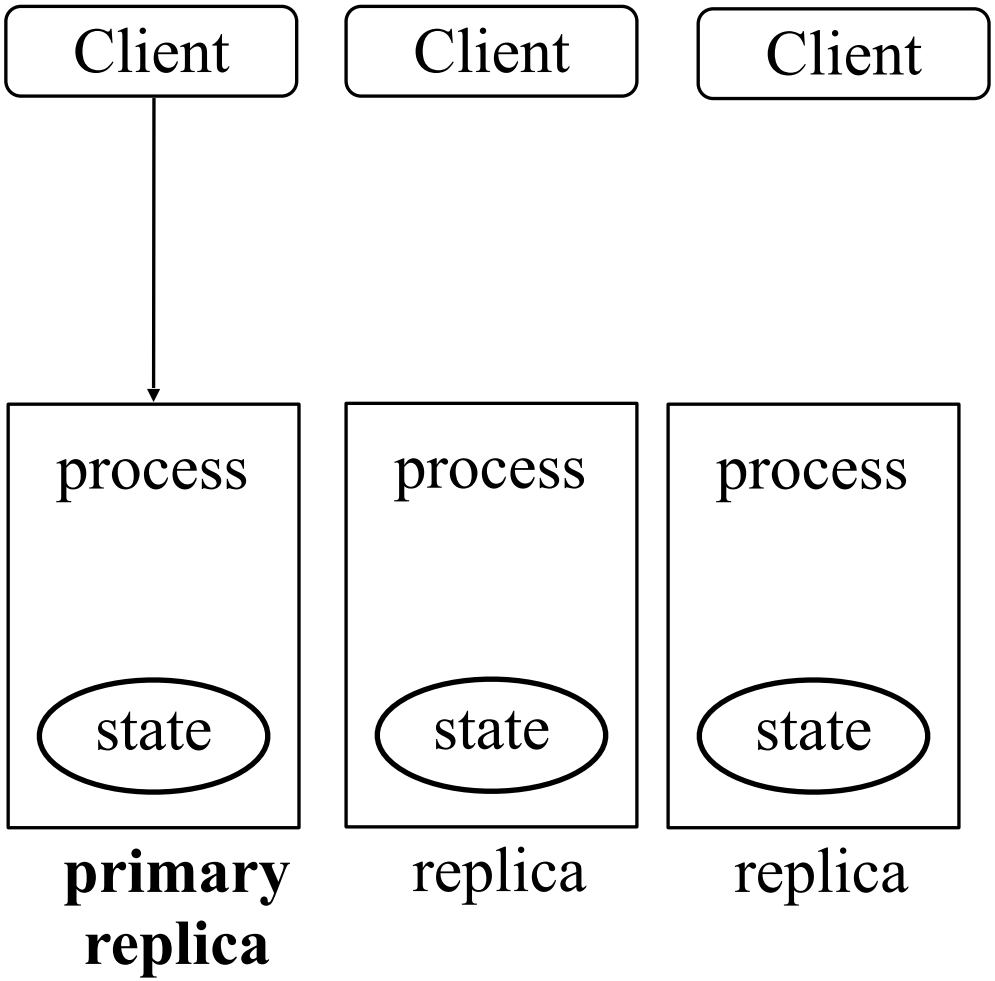


- **Passive replication**
  - **Primary-backup**
  - Multi-primary

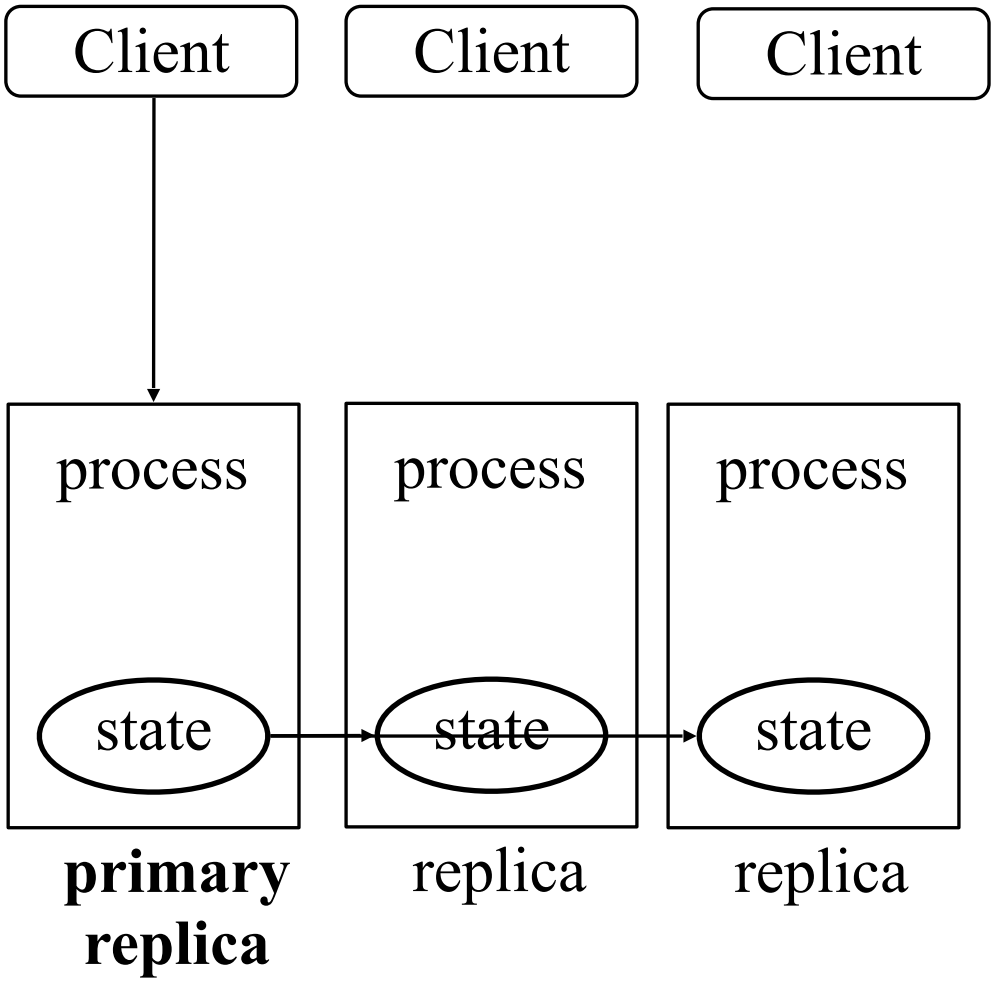
Replication Pattern



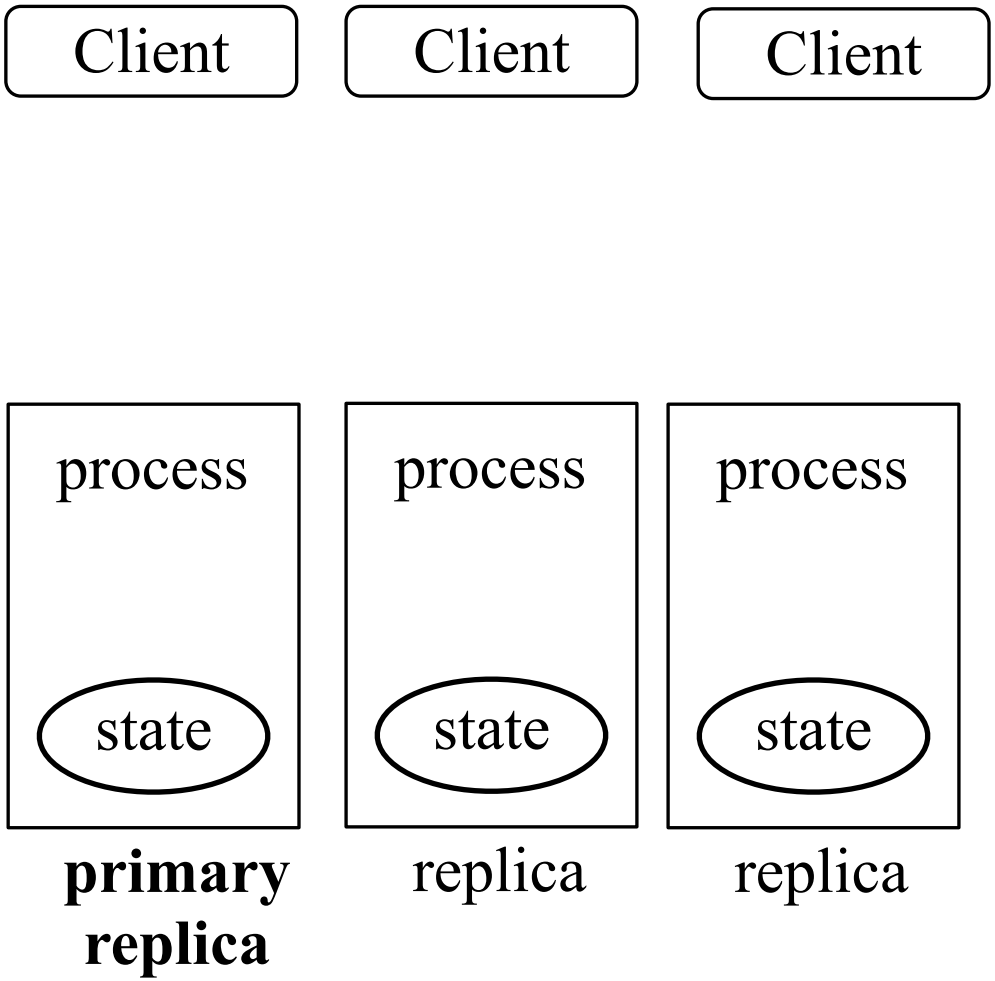
Replication Pattern



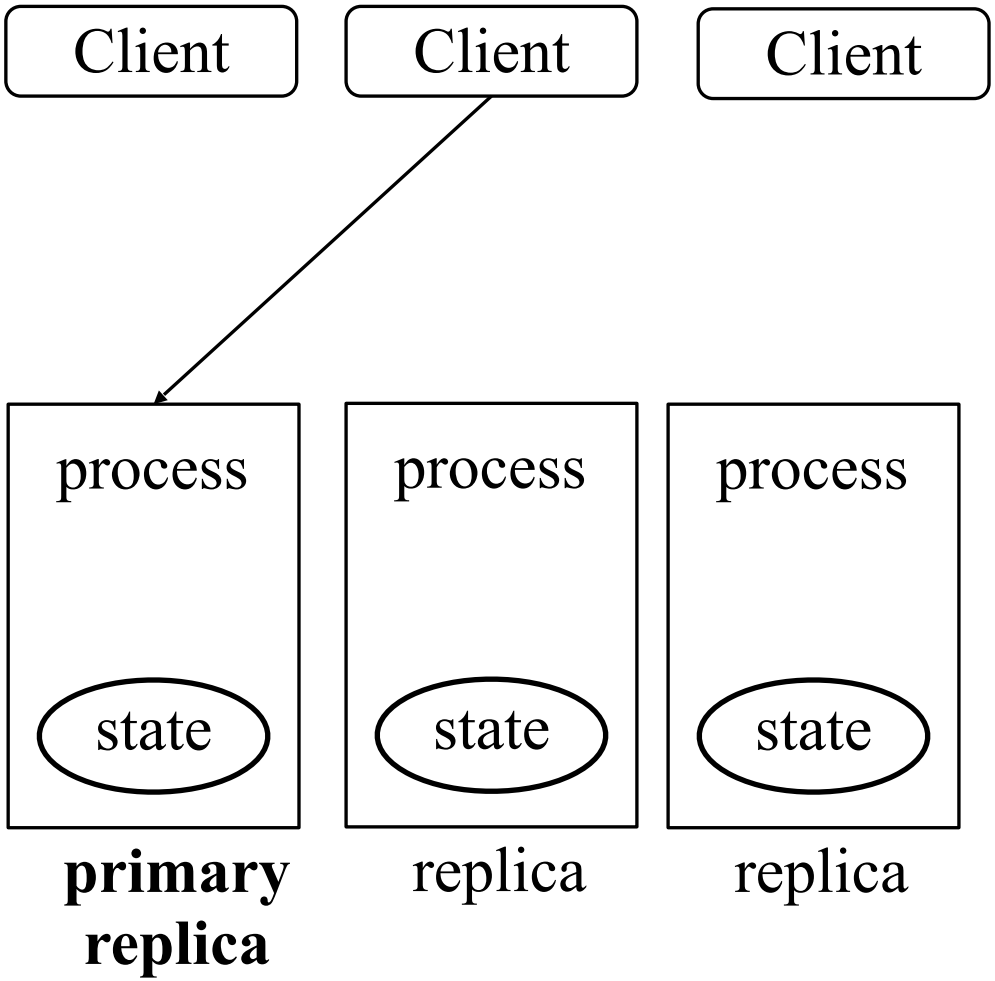
Replication Pattern



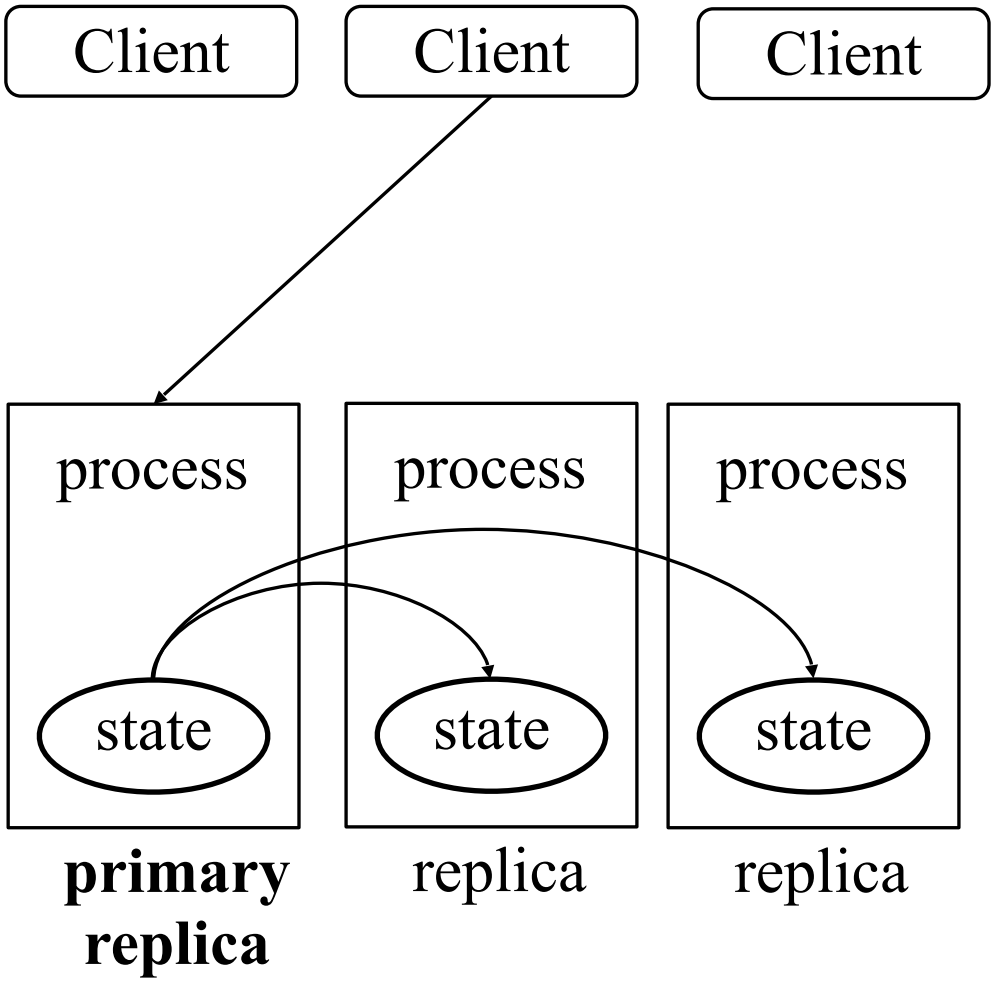
Replication Pattern



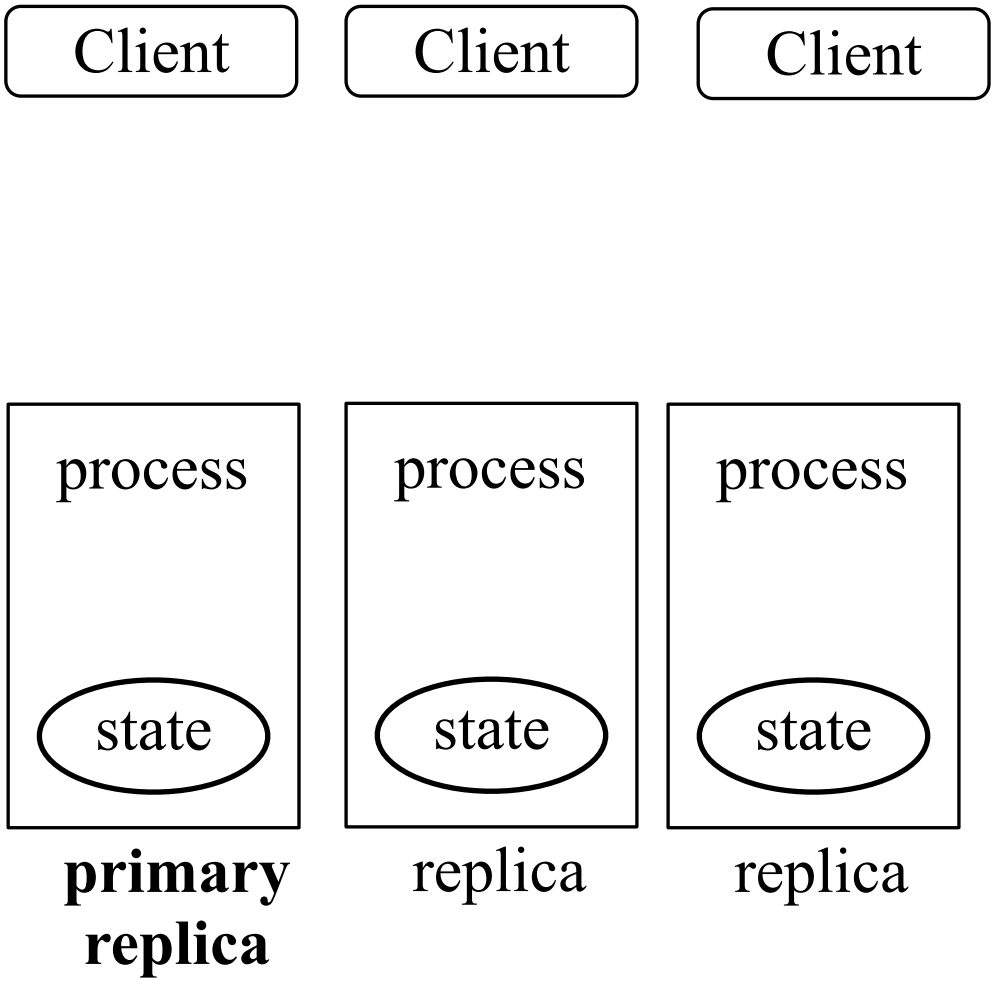
Replication Pattern



Replication Pattern



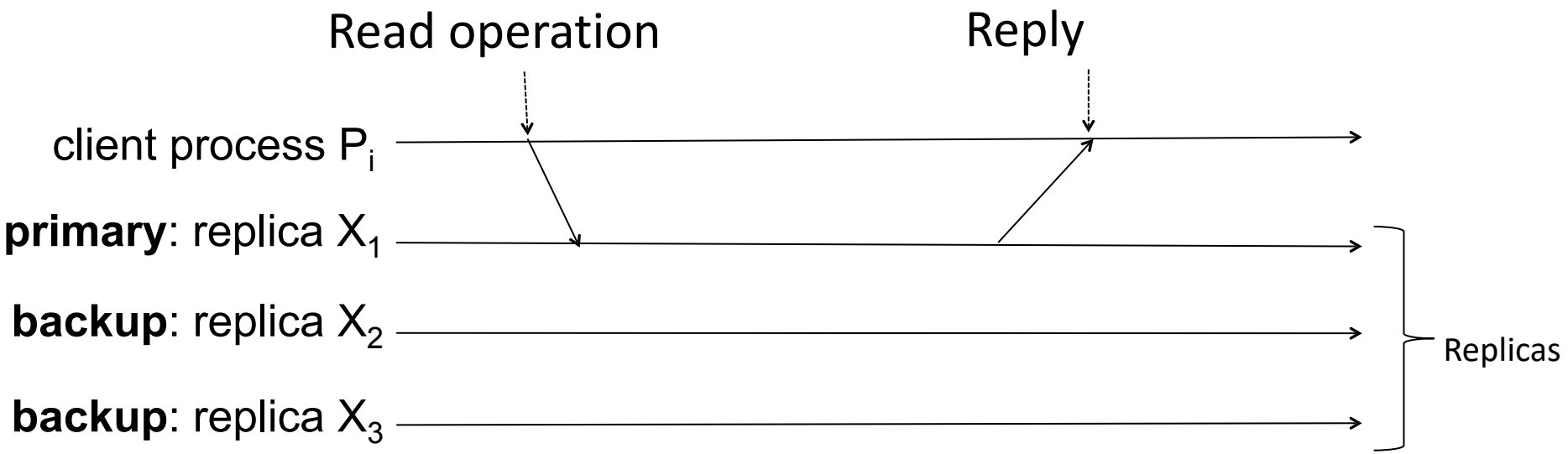
Replication Pattern



Passive Replication: Primary-Backup

- A replica is chosen to be the **primary** (leader election)
- **Primary**
  - Receives invocations from clients
  - Executes requests and sends back replies
  - Replicates the state to other replicas
- **Backup**
  - Interacts with primary only
  - Used to replace primary when it crashes (leader election)
- Called **eager replication** if replication is performed **within request boundary** (e.g., before the reply is sent)

Primary-Backup Scenario

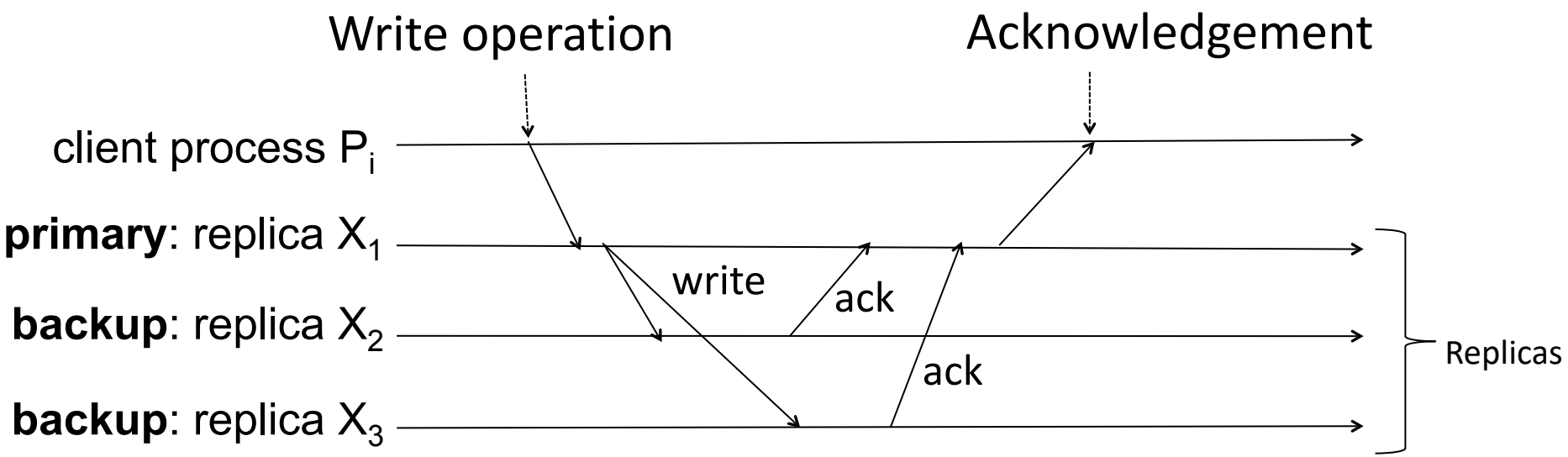


Configuration service:

- Failure detection
- Configuration management

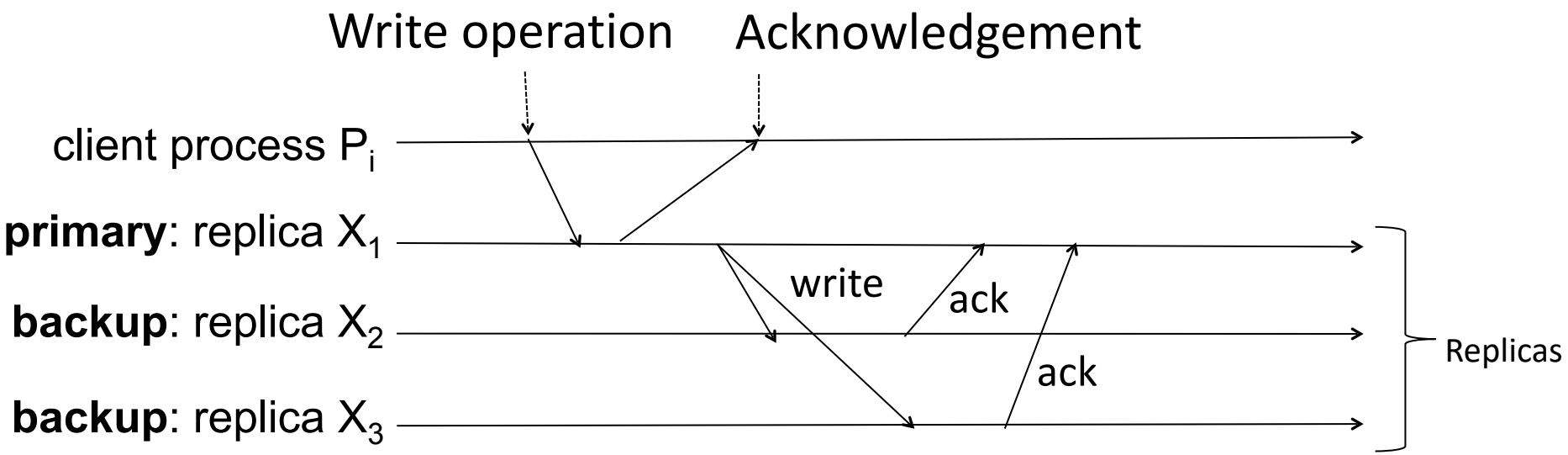
Eager replication

Primary-Backup Scenario



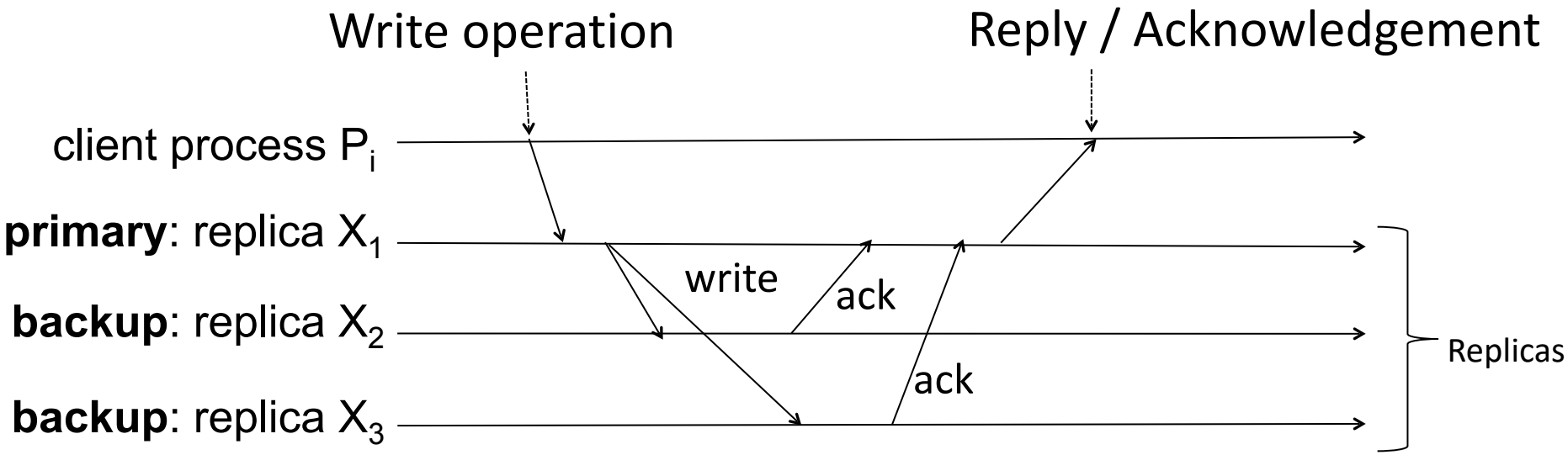
Lazy replication

Primary-Backup Scenario



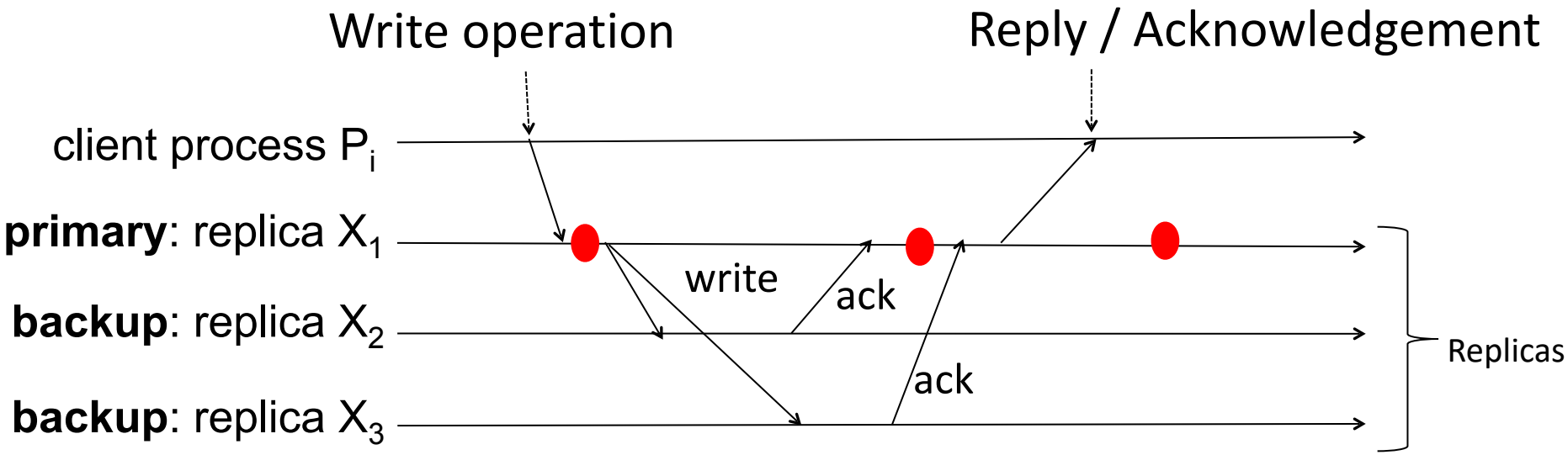
- Writes are propagated asynchronously after primary acknowledges update to client
- Replicas may diverge
- Requires additional mechanism to deal with primary crashing

Primary-Backup: Presence of Failures



- In all cases, a new **primary is elected** from among the backups

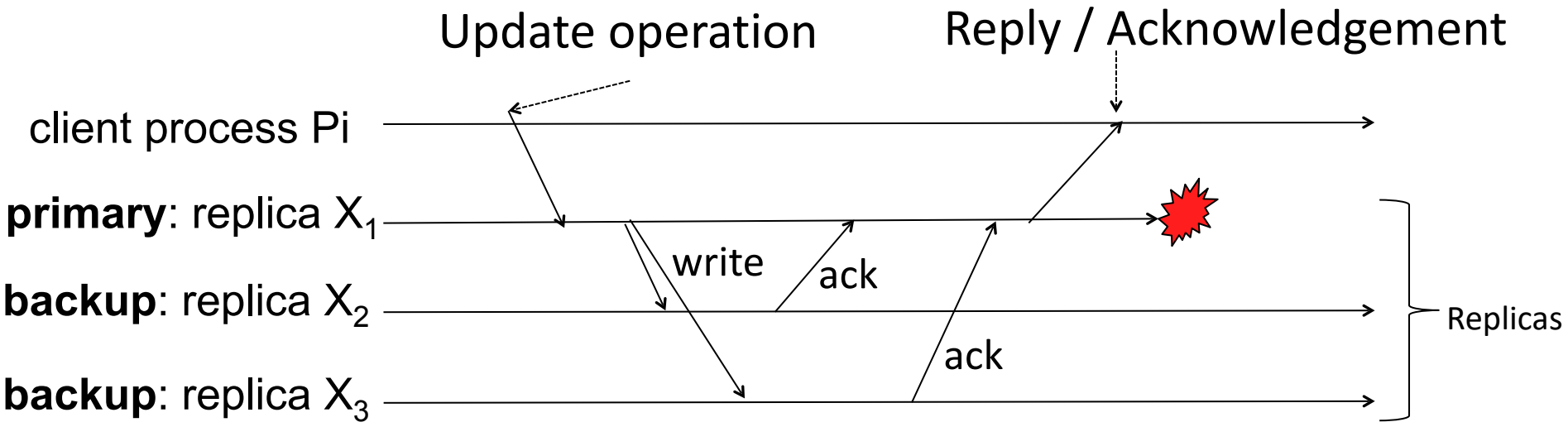
Primary-Backup: Presence of Failures



- In all cases, a new **primary is elected** from among the backups

Scenario 1

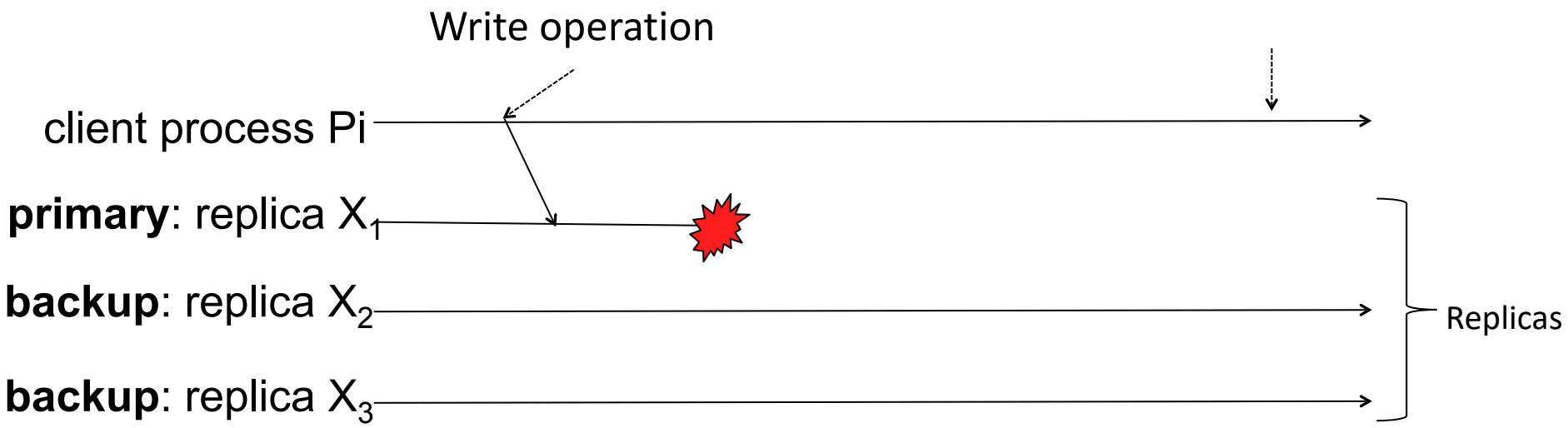
Primary fails after client receives reply



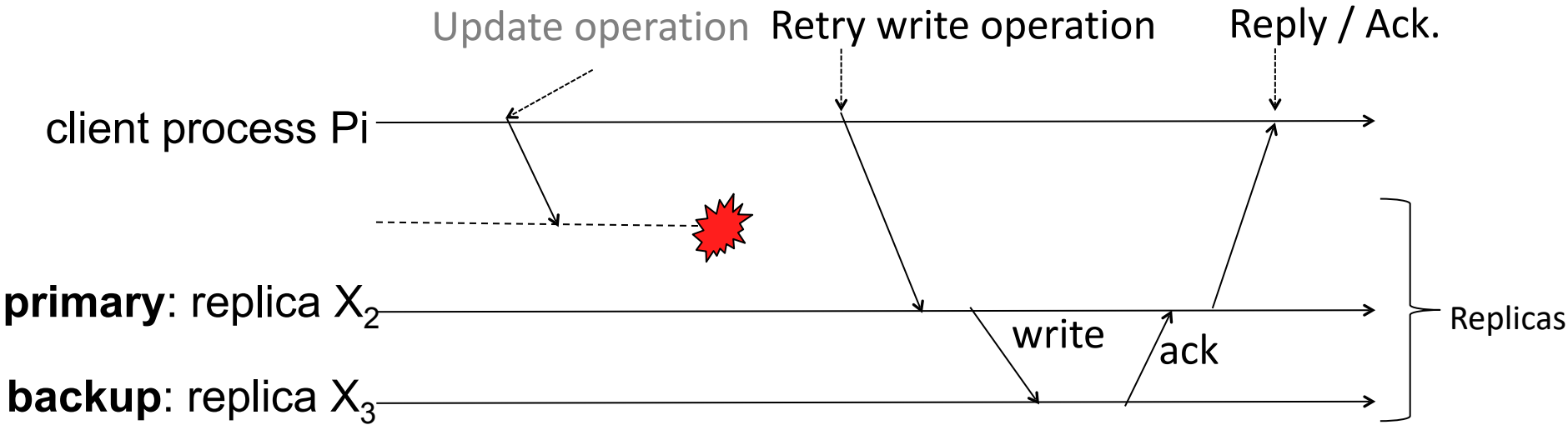
- New primary is elected

Scenario 2

Primary fails before propagating updates



Scenario 2

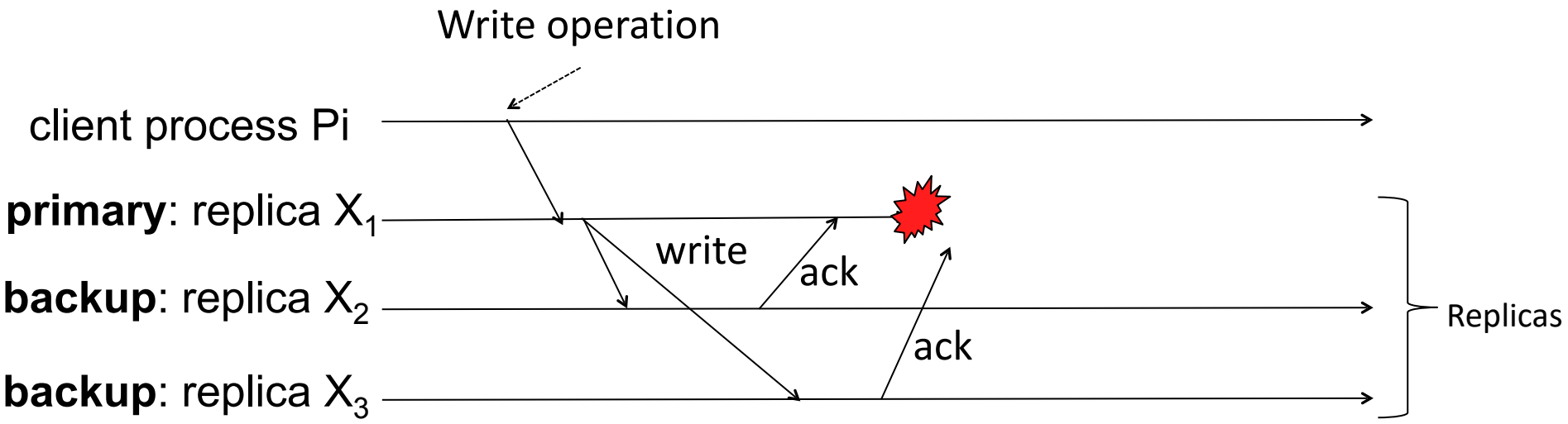


Configuration service:
- Failure detection
- Leader election
- Configuration management

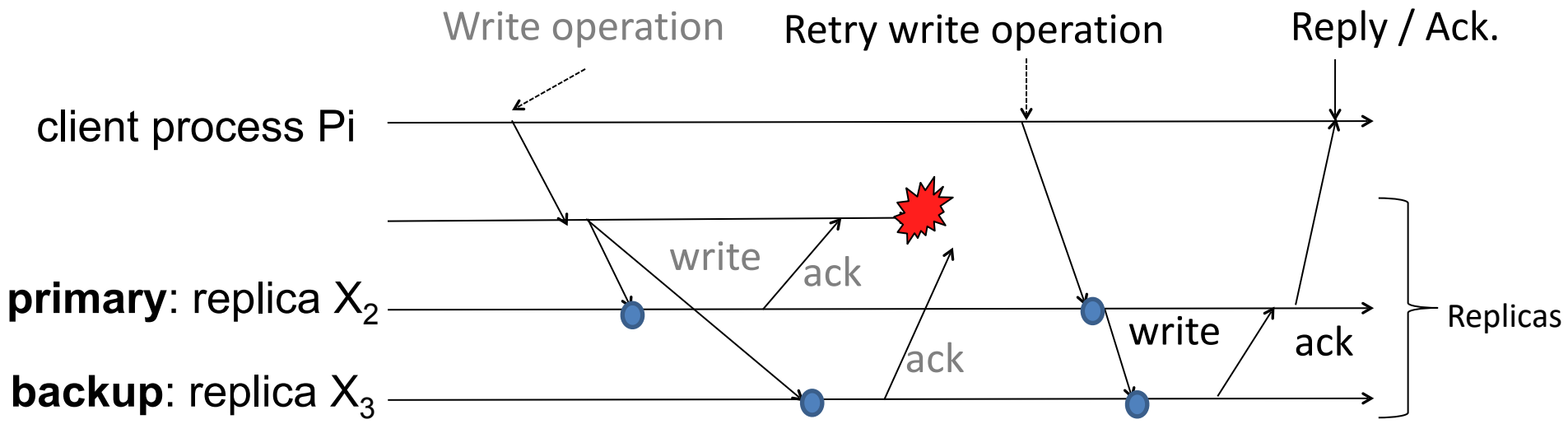
- Timeout mechanism at client triggers retry
- Retry could fail, if against old primary
- Check configuration service for new leader, retry

Scenario 3

Primary fails before receiving all write acknowledgements



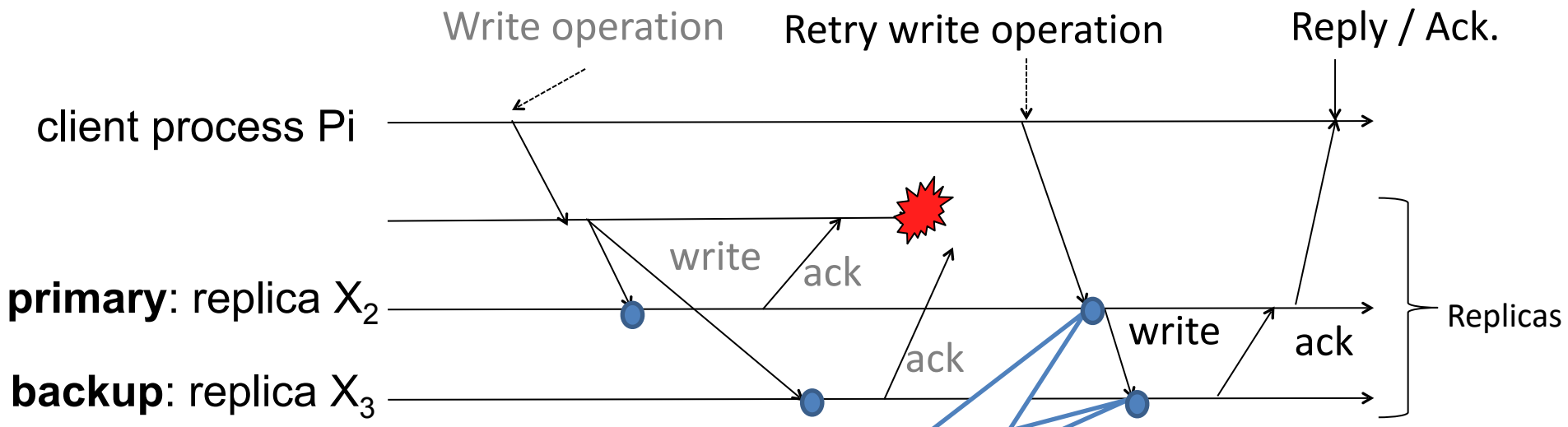
Scenario 3



Configuration service:
- Failure detection
- Leader election
- Configuration management

- Timeout and retry write
- Persistent log at replicas
- Configuration service

Scenario 3

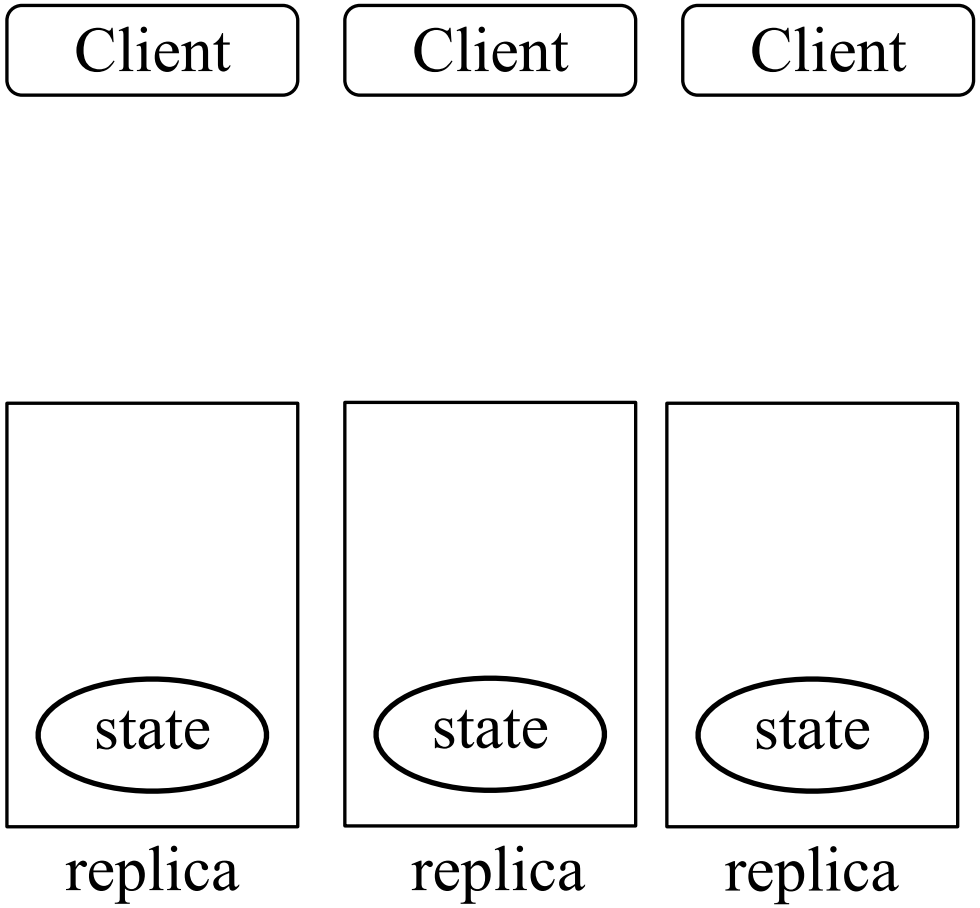


Configuration service:
- Failure detection
- Leader election
- Configuration management

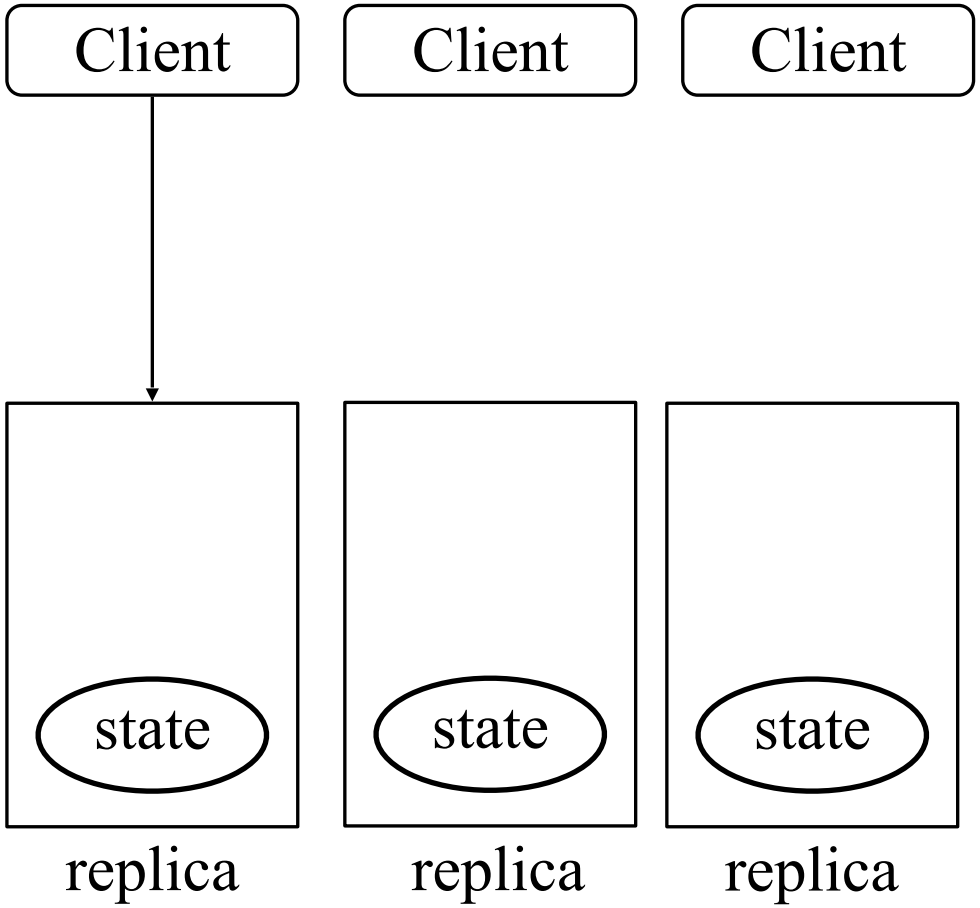
Use WAL at replicas to differentiate between Scenario 2 and 3 (**avoid updating twice**)

- Timeout and retry write
- Persistent log at replicas
- Configuration service

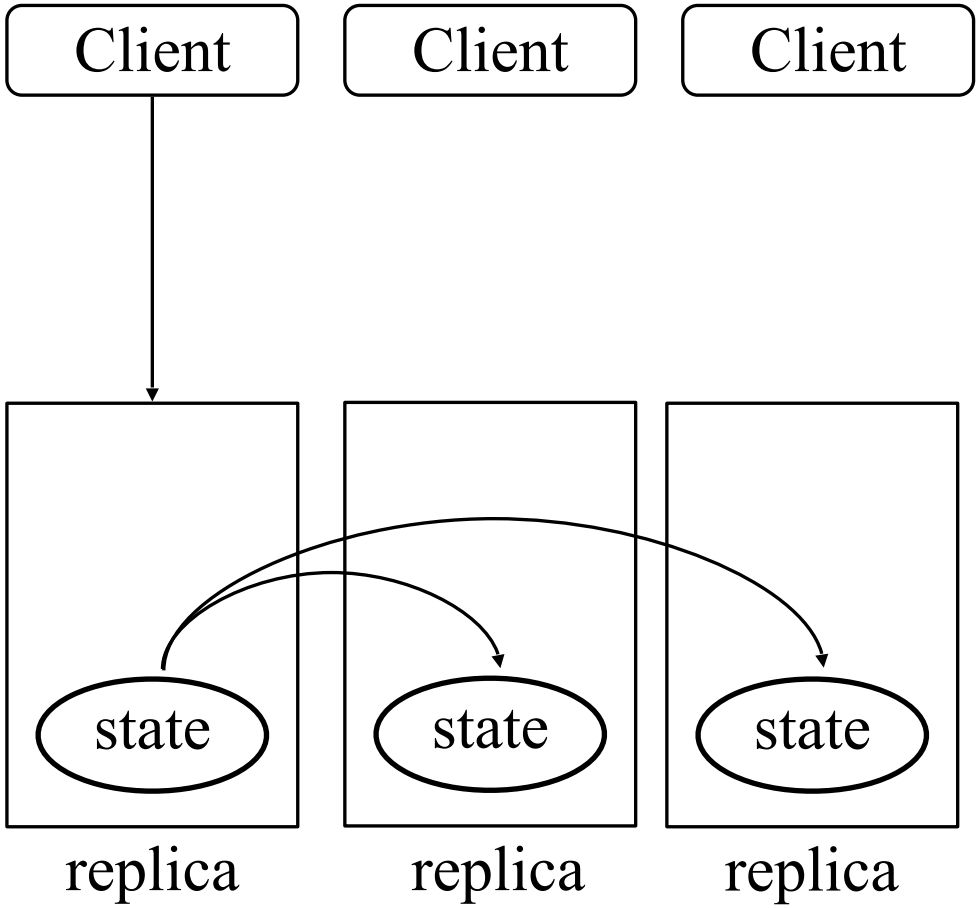
Multi-primary Replication (MPR)



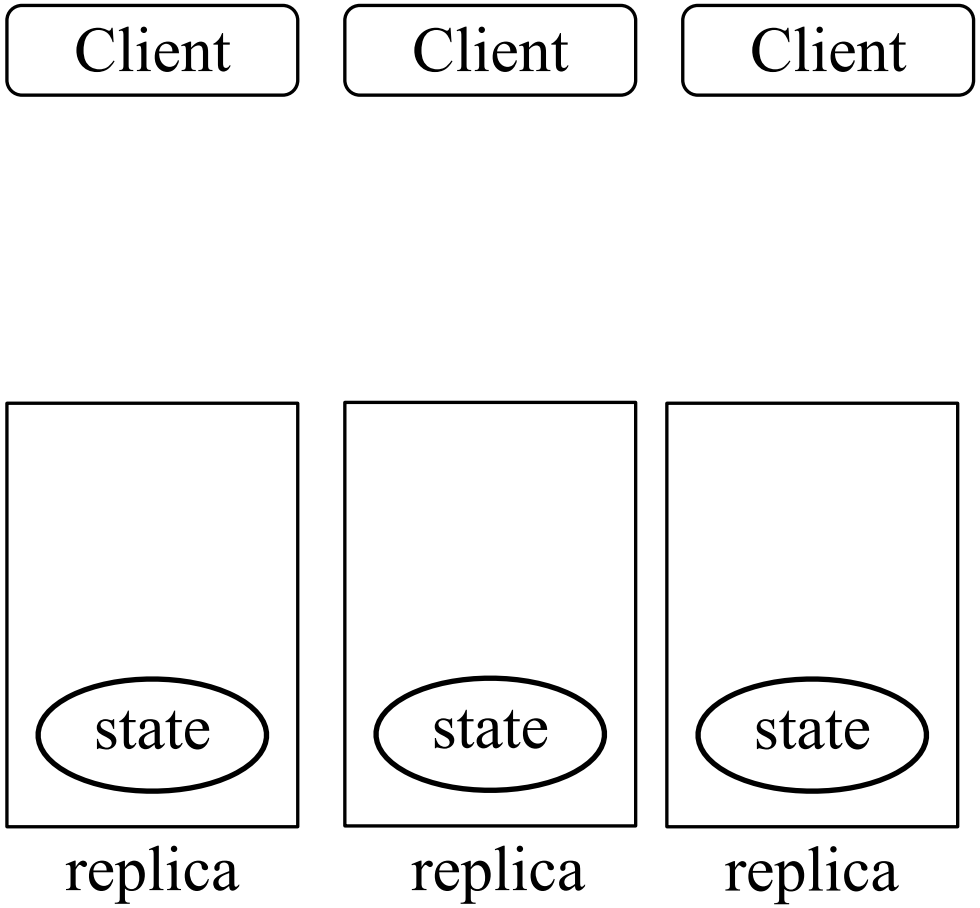
Multi-primary Replication (MPR)



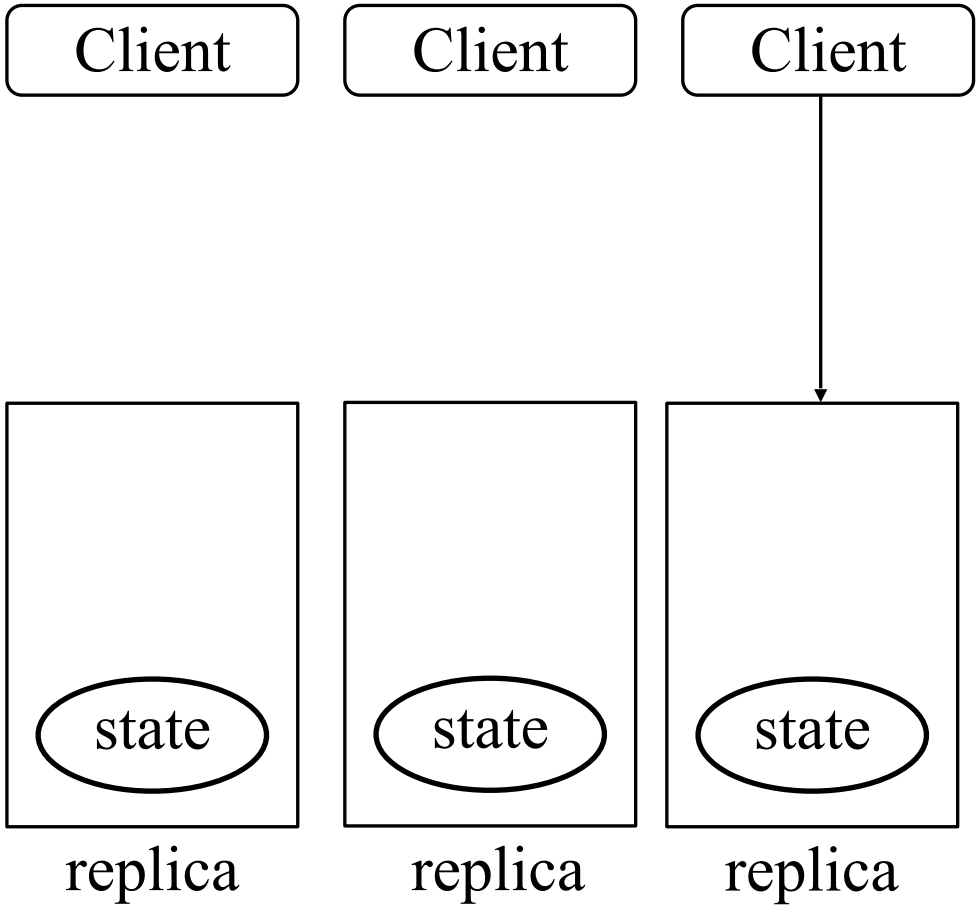
Multi-primary Replication (MPR)



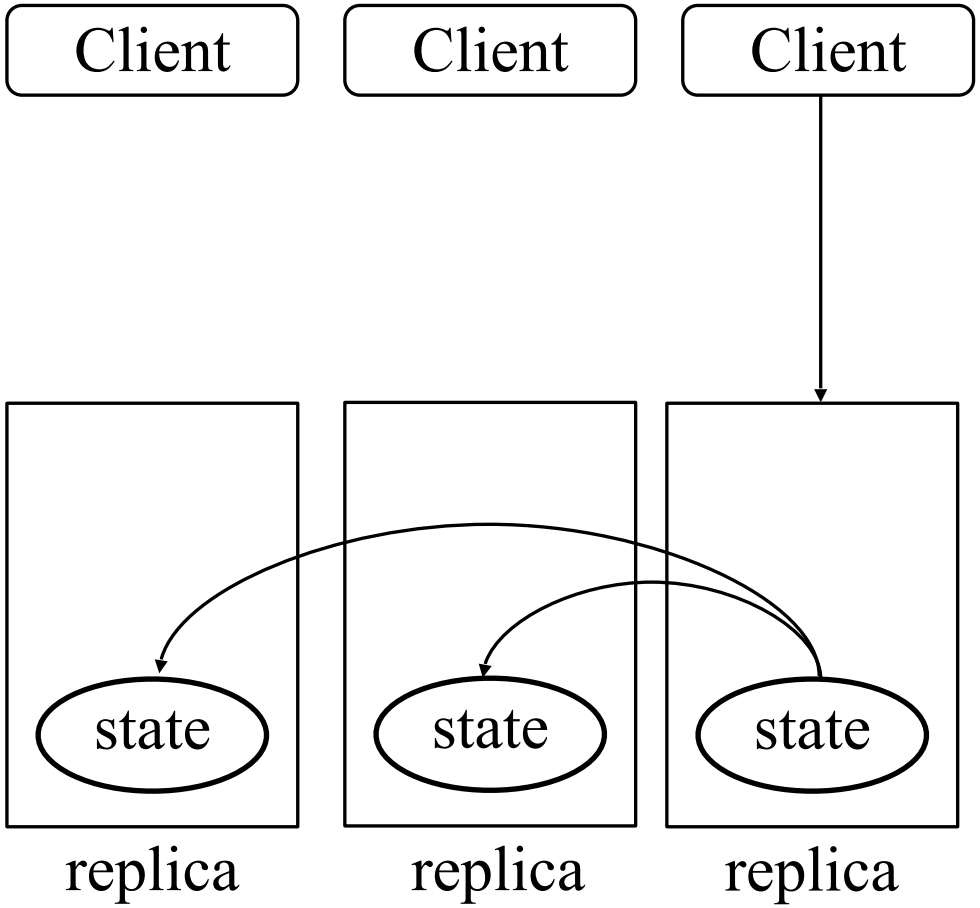
Multi-primary Replication (MPR)



Multi-primary Replication (MPR)



Multi-primary Replication (MPR)



Multi-primary Replication (MPR)

- Primary-backup approach is not scalable since a single replica handles client requests
  - Inefficient use of replica resources
- Multi-primary approach solves this issue by allowing every replica to handle client requests
  - Replicas have to figure out how to order requests (e.g., **using consensus**)
- If replication is **eager**, replicas have to **agree on order of operations** before they execute any command and respond to clients
  - Can be slow since replica must be locked

Optimistic Lazy MPR

- To improve response times, **replication** is often **done lazily**
  - Replica first executes locally and returns a response to client right away
  - Replicas asynchronously propagate updates they made

- Also called **optimistic replication**
  - Replicas may diverge, which can introduce inconsistencies, aborts, and rollbacks

Self-study Questions

- Specify as pseudo code how a configuration service is used in each replication failure scenario.
- Further, specify as pseudo code each the client and replica, including the additional mechanisms needed in each failure scenario.
- Analytically compare all replication patterns in terms of number of messages exchanged for reading and writing, given *n* replicas.
- How do the replication patterns compare in terms of potential for concurrency of operations?

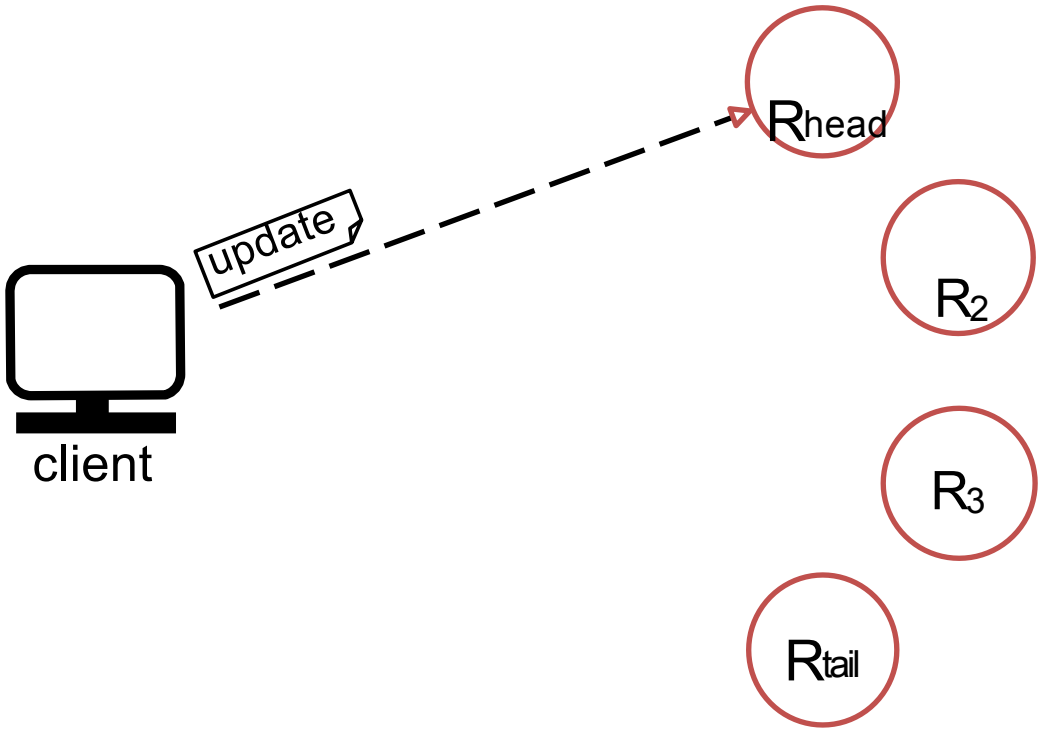
Pixabay.com

CHAIN REPLICATION

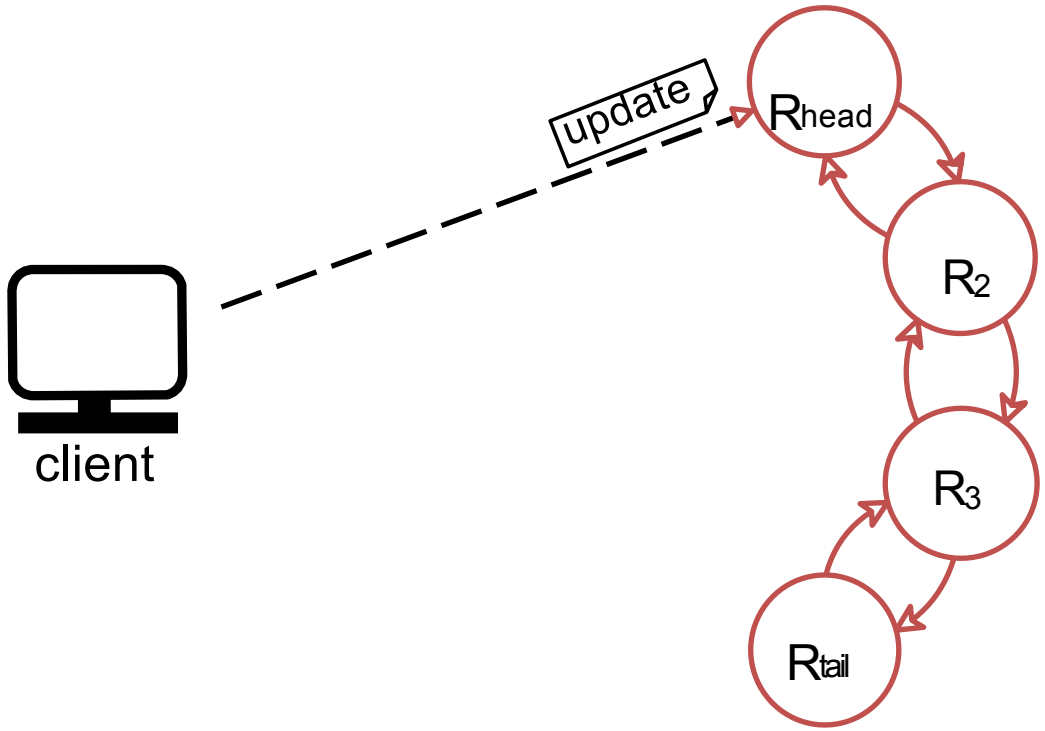
OVERVIEW

Inspiration for this lecture taken from a talk given by Deniz Altınbüken.

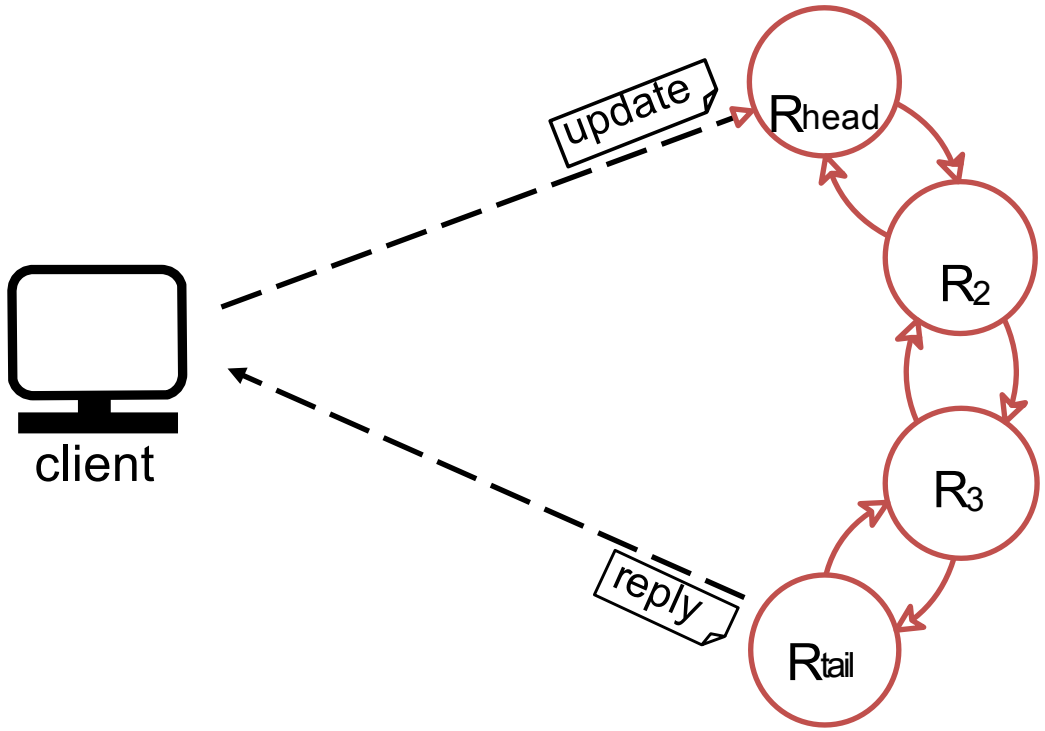
Chain Replication



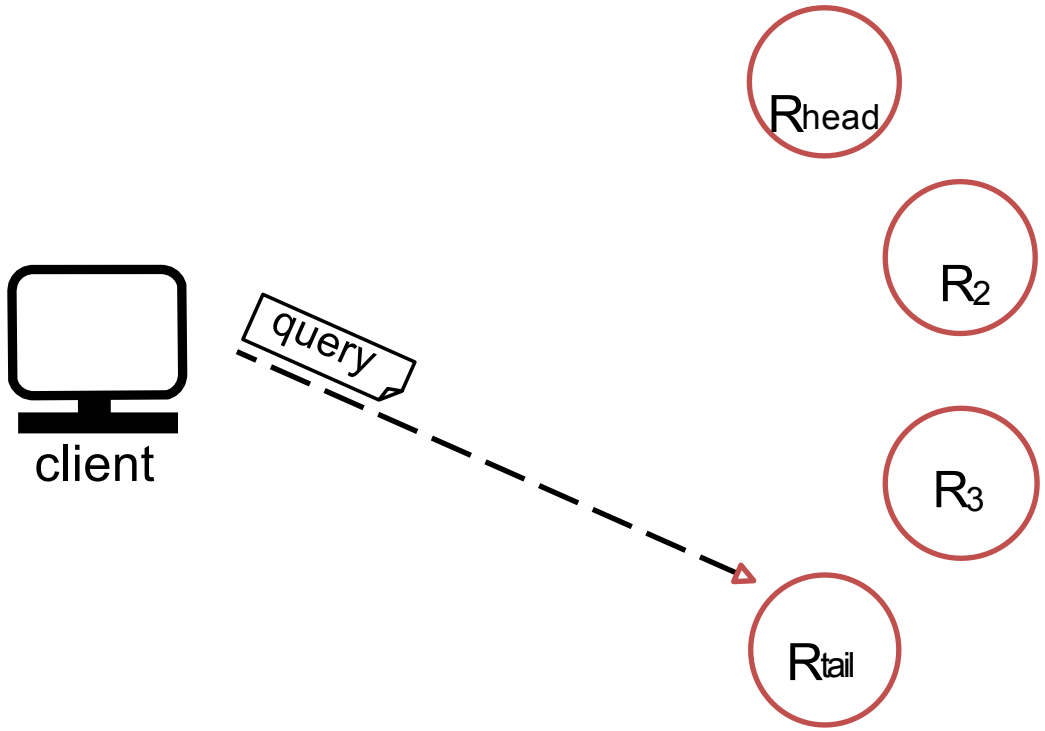
Chain Replication



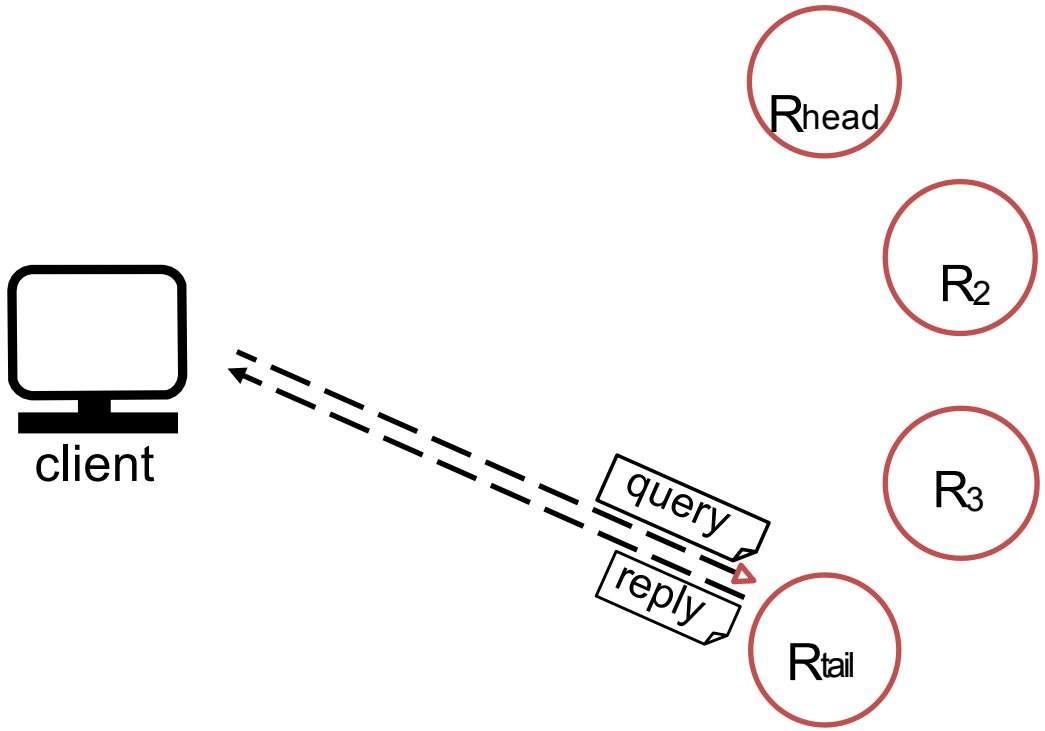
Chain Replication



Chain Replication



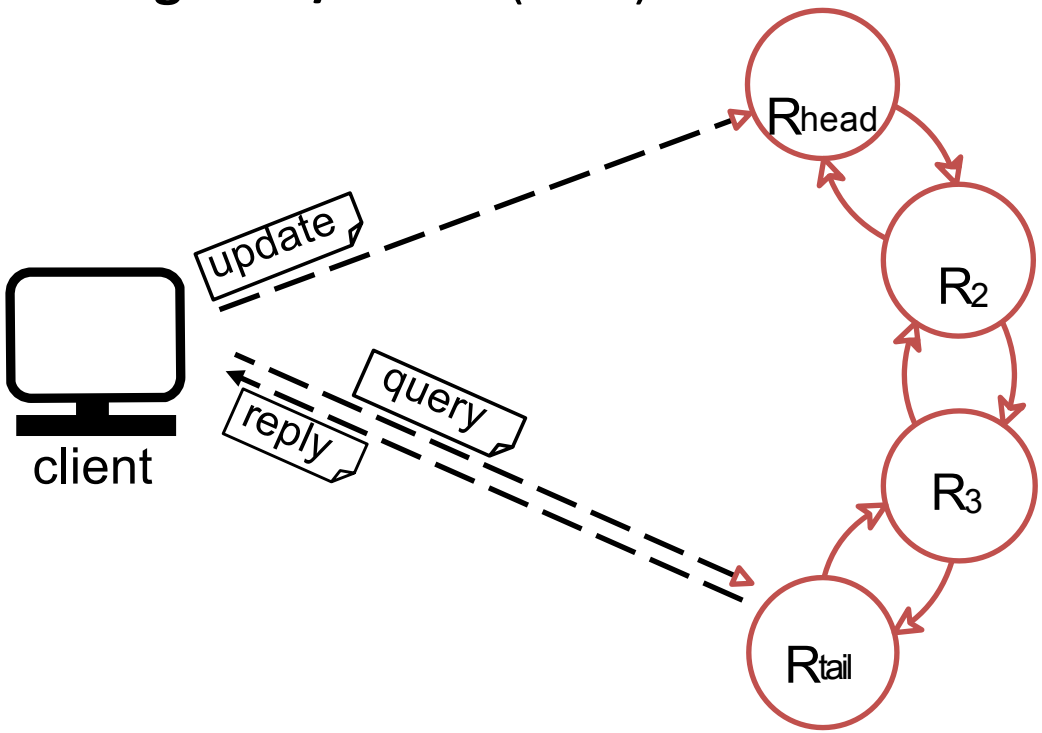
Chain Replication



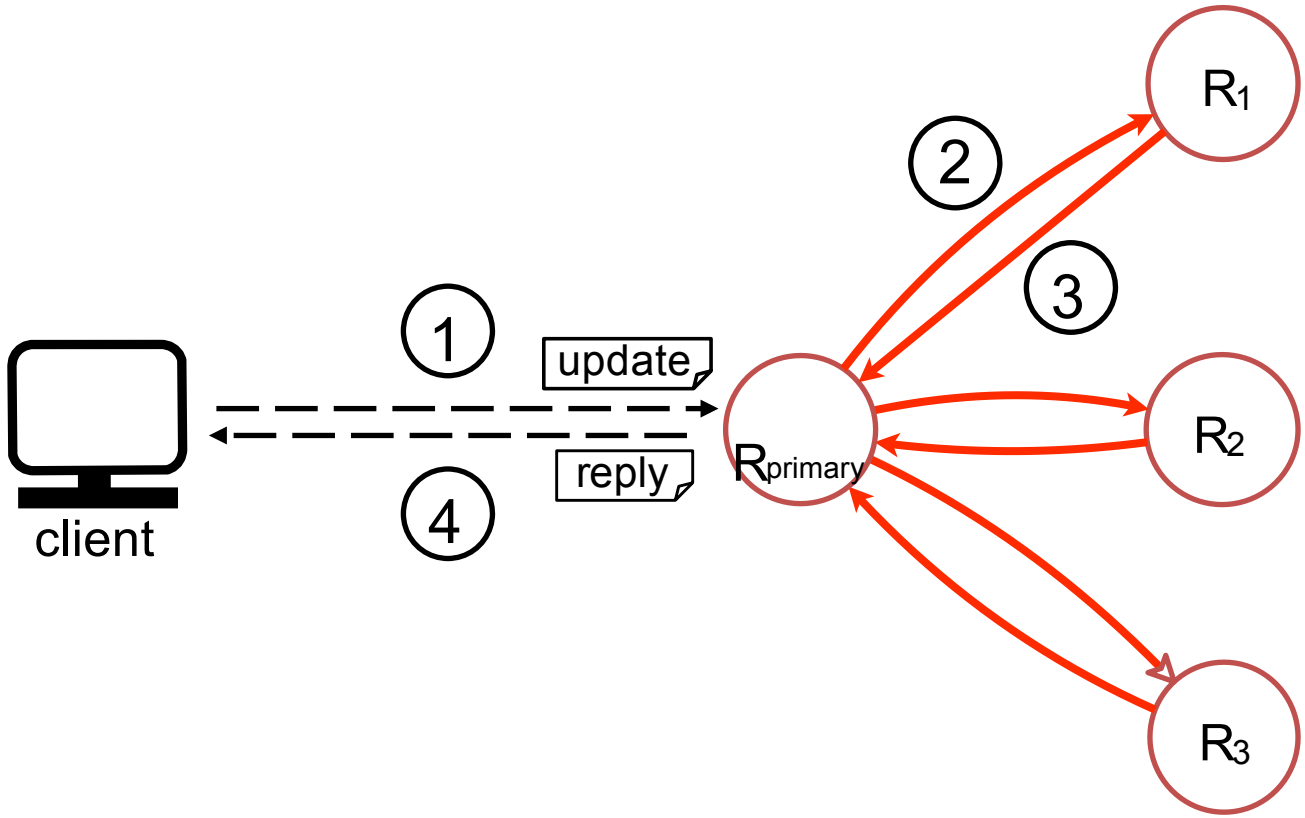
Chain Replication

Separate concerns across replicas:

- **Update processing request to head** (write)
- **Update processing reply from tail** (write)
- **Query processing from/to tail** (read)



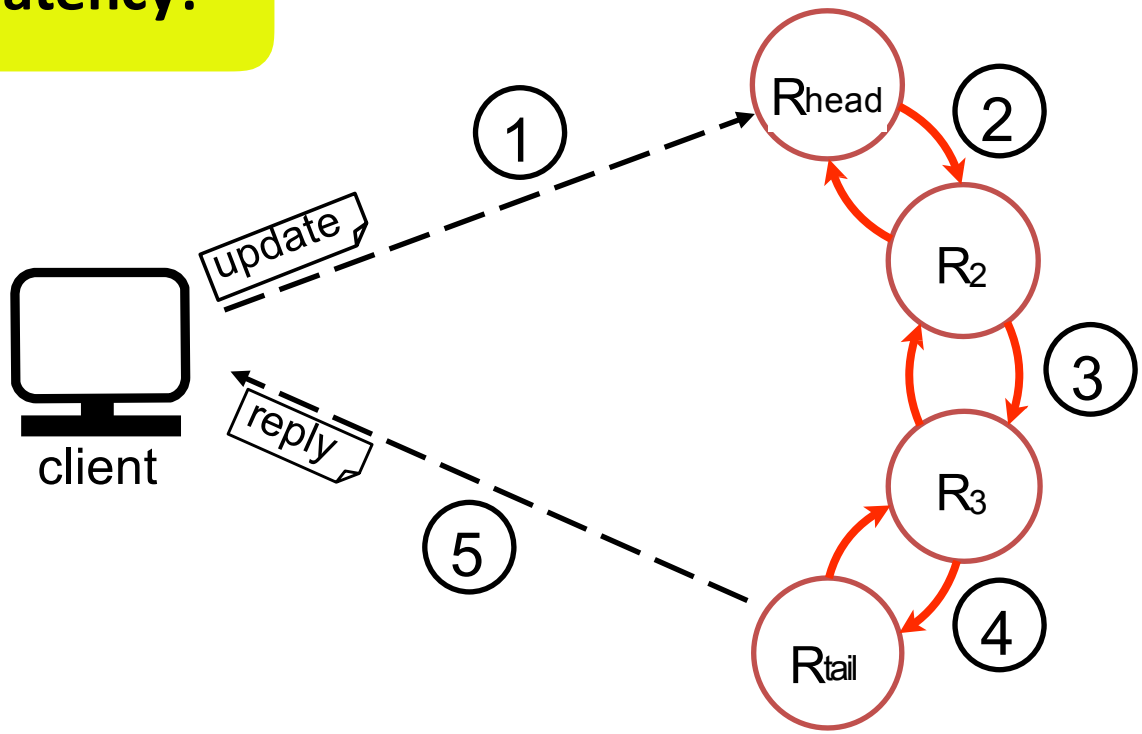
Primary-Backup Replication



Four replicas – latency of **four** messages

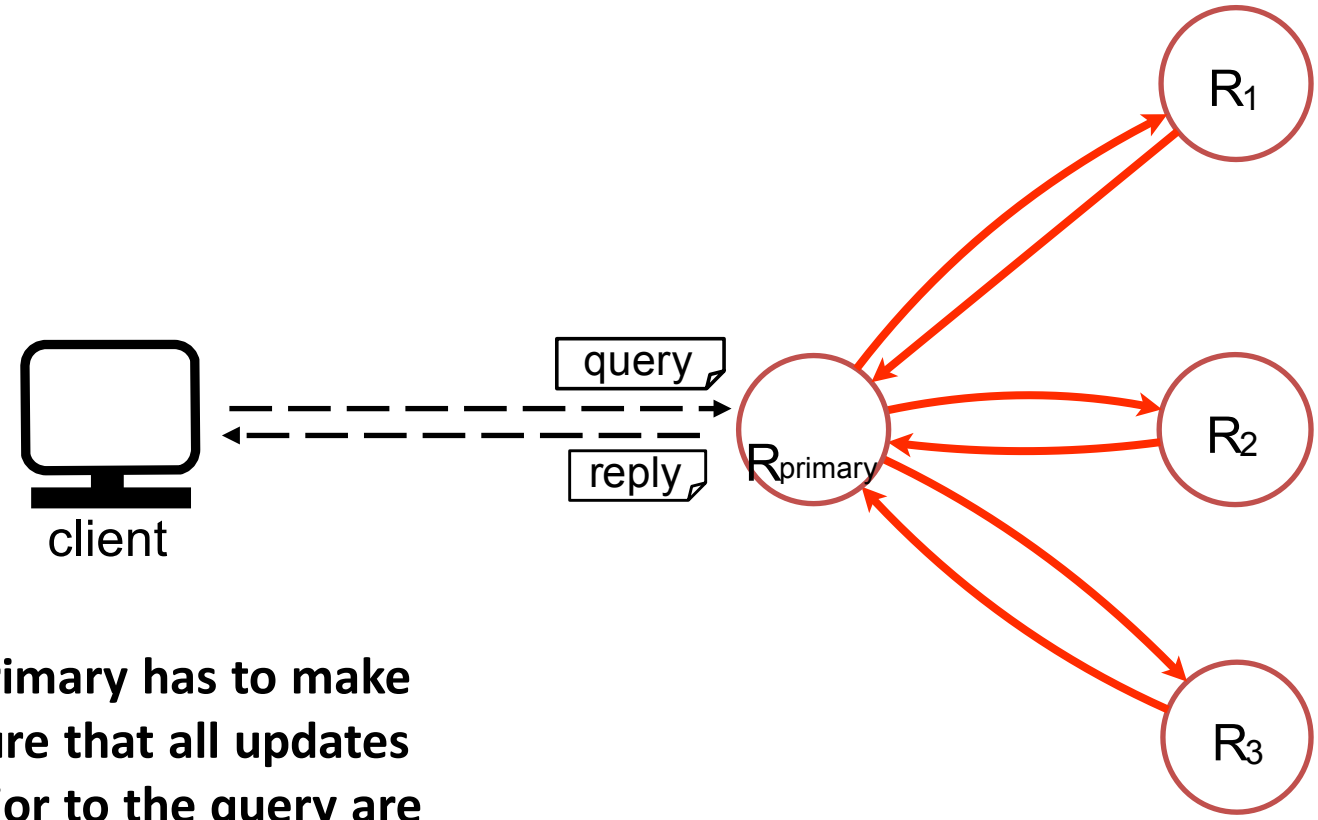
Chain Replication

Higher latency!



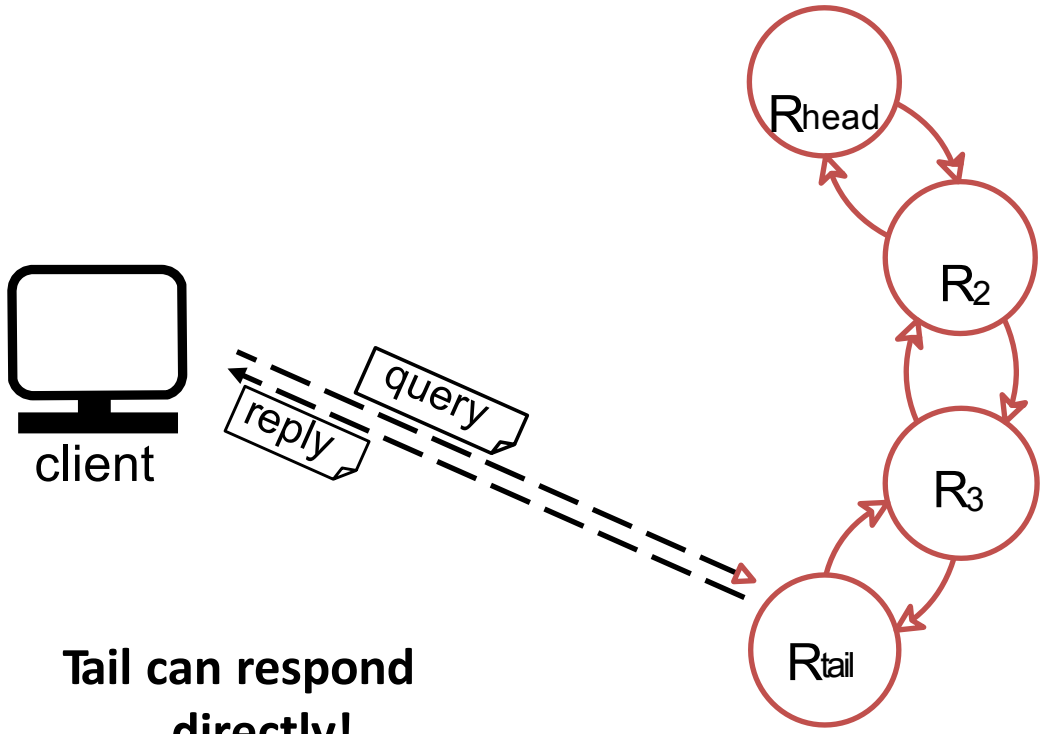
Four replicas – latency of **five** messages

Primary-Backup Replication



Primary has to make sure that all updates prior to the query are completed!

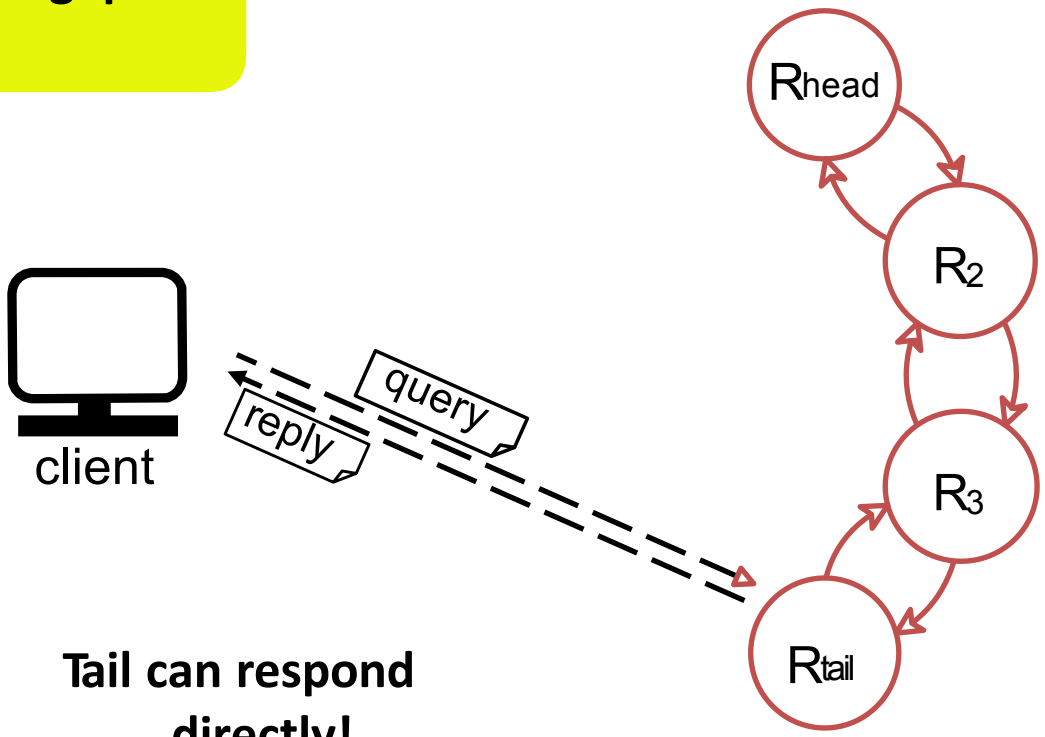
Chain Replication



Tail can respond directly!

Chain Replication

Higher throughput!



Tail can respond directly!

Fault-tolerance in Chain Replication

- Need f + 1 nodes to tolerate f failures

Pixabay.com

Self-study Questions

- How many node failures can the primary-backup approach sustain without disrupting service?
- Discuss pros and cons of chain, primary-backup, active, and multi-primary replication?
- Quantify the number of messages exchanged by drawing on *n* replicas for each of the replication schemes for *m* updates.
- Discuss the replica balance across all approaches?

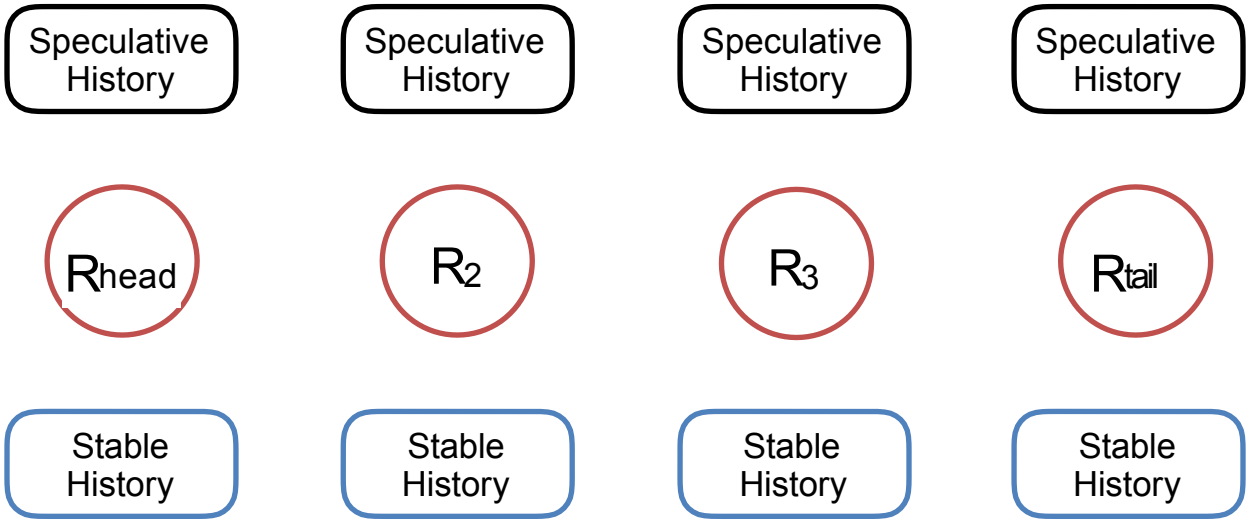
Pixabay.com

CHAIN REPLICATION

UPDATE & QUERY OPERATIONS

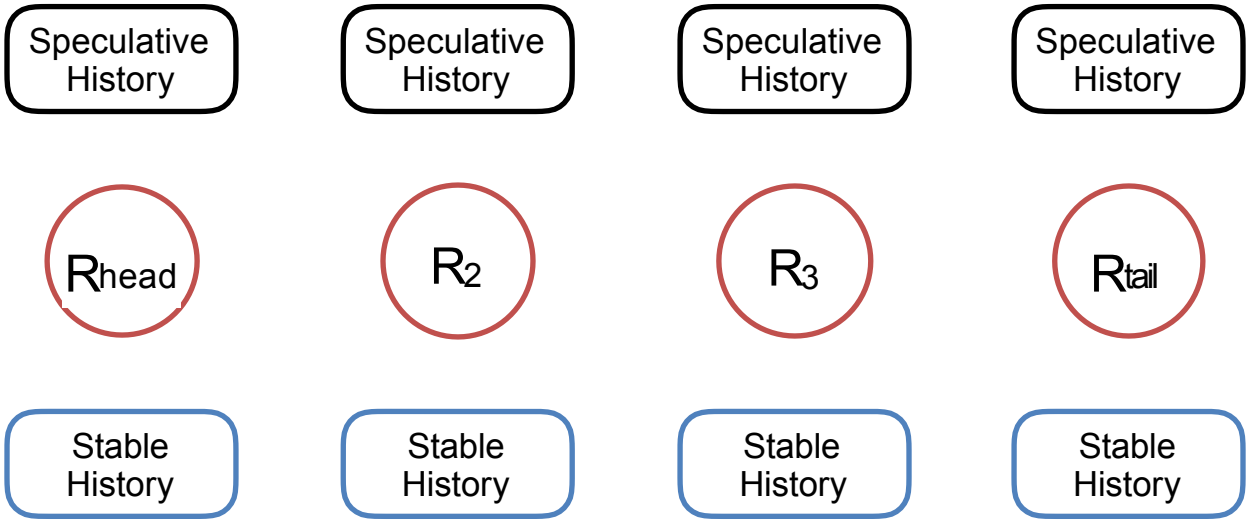
Inspiration for this lecture taken from a talk given by Deniz Altınbüken.

Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

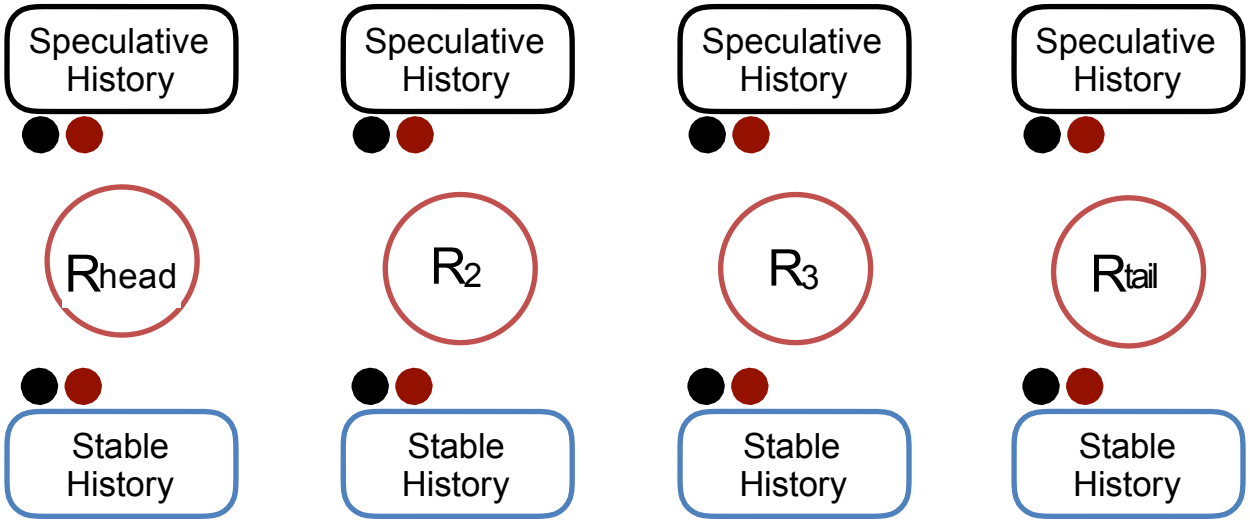
Updates

R_2 is the **predecessor** of R_3

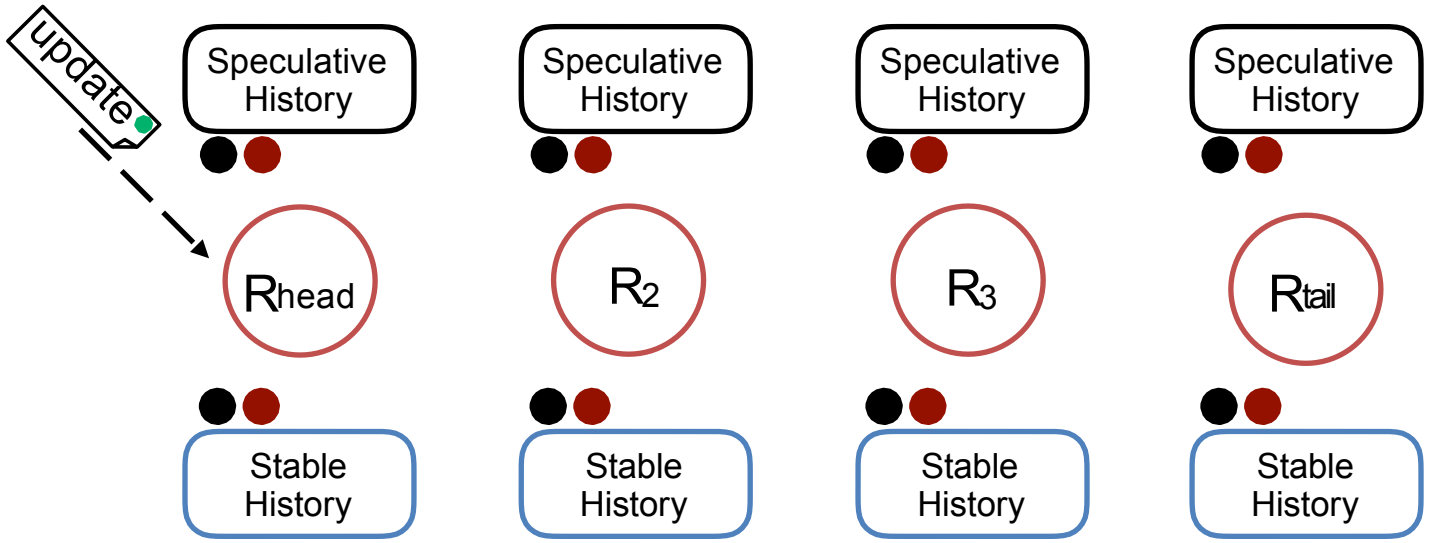


R_3 is the **successor** of R_2

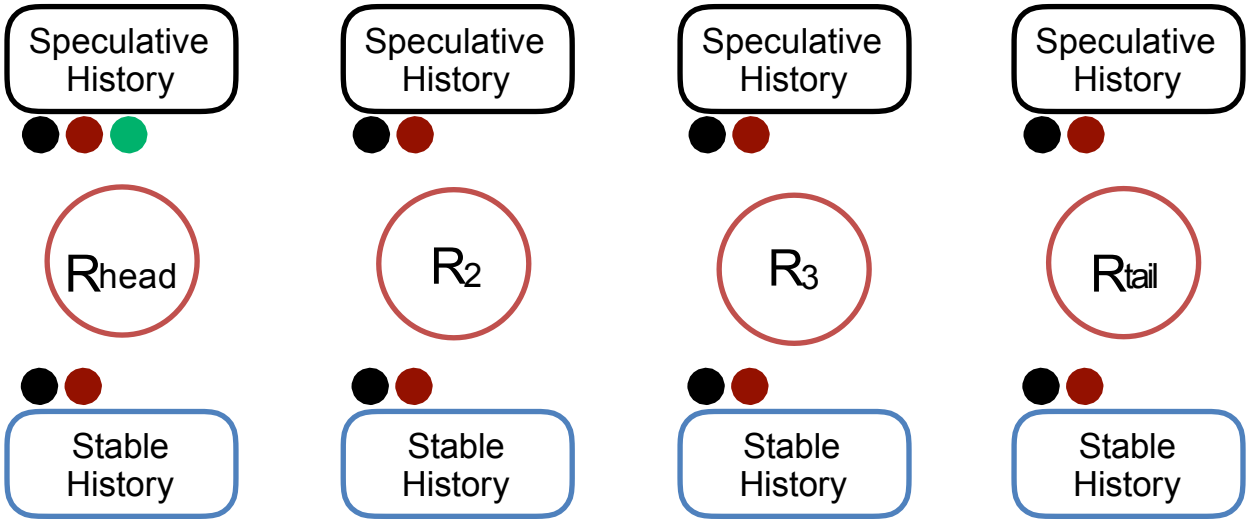
Updates


Speculative History
Rhead
Stable History
Speculative History
R2
Stable History
Speculative History
R3
Stable History
Speculative History
Rtail
Stable History

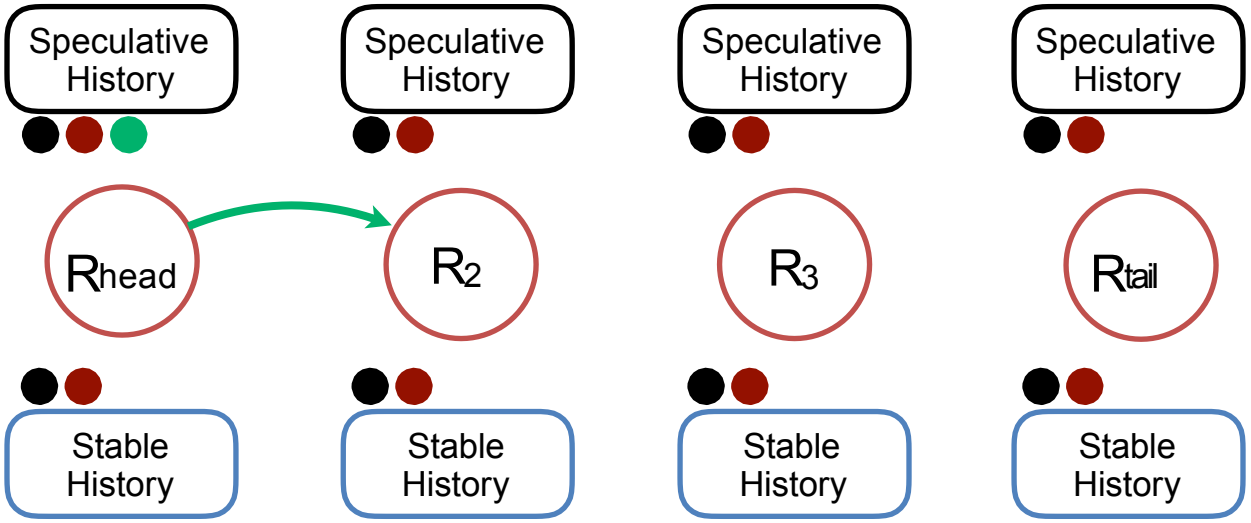
Updates


update
Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

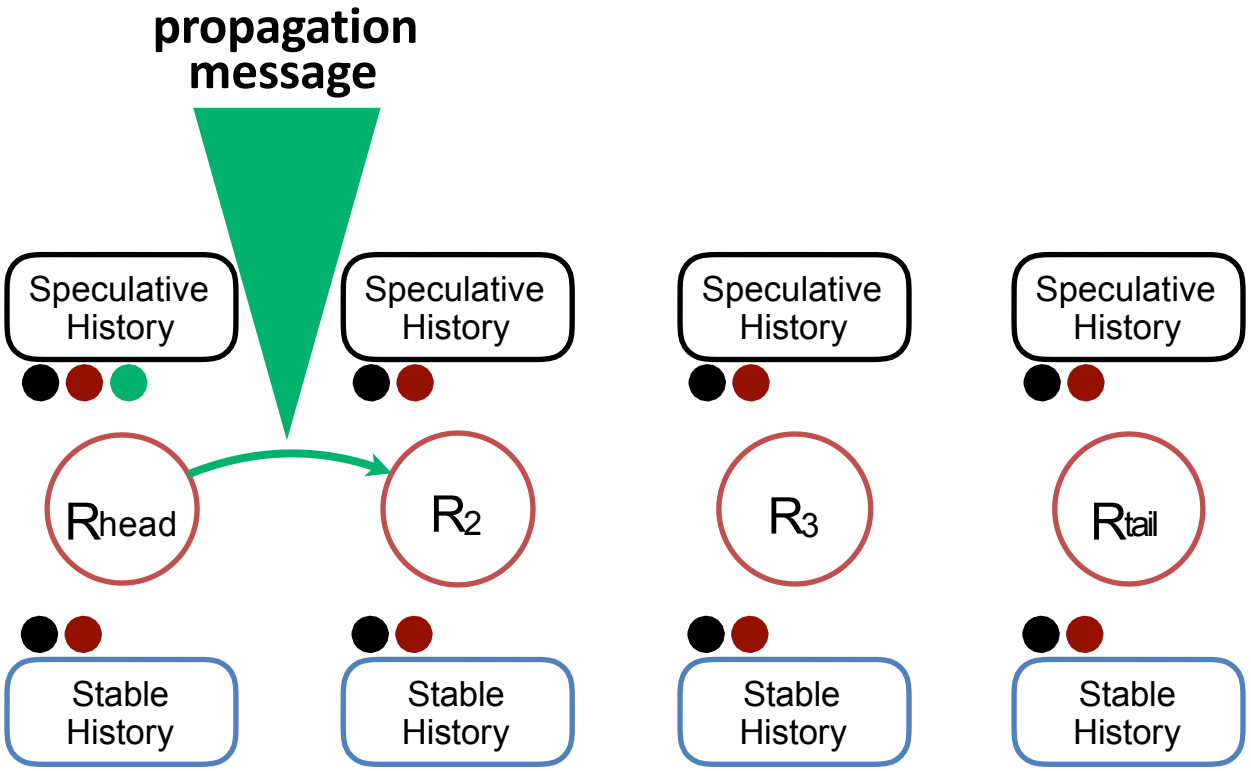
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

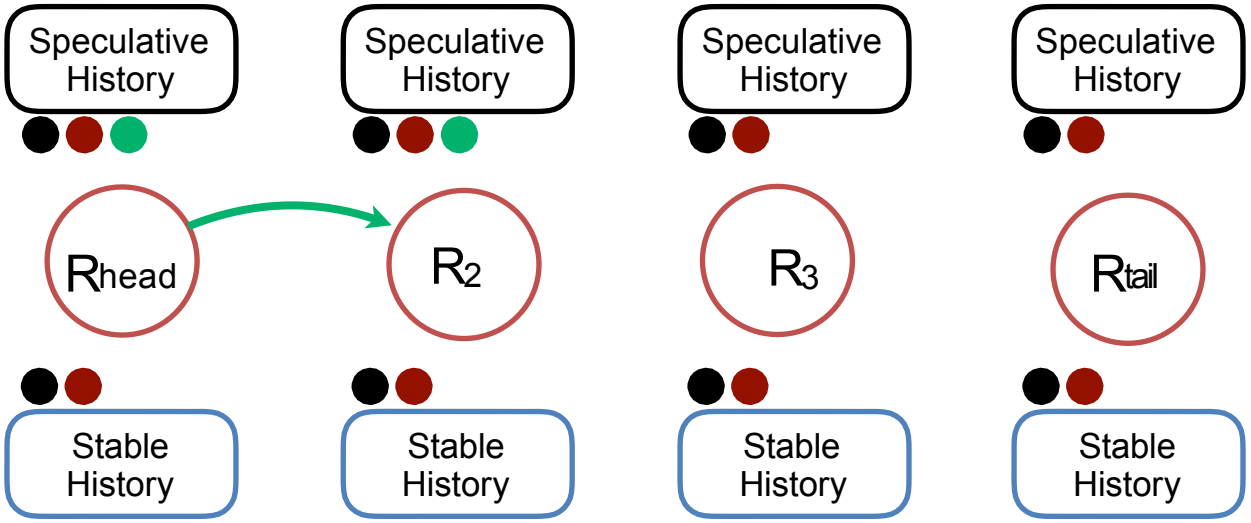
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

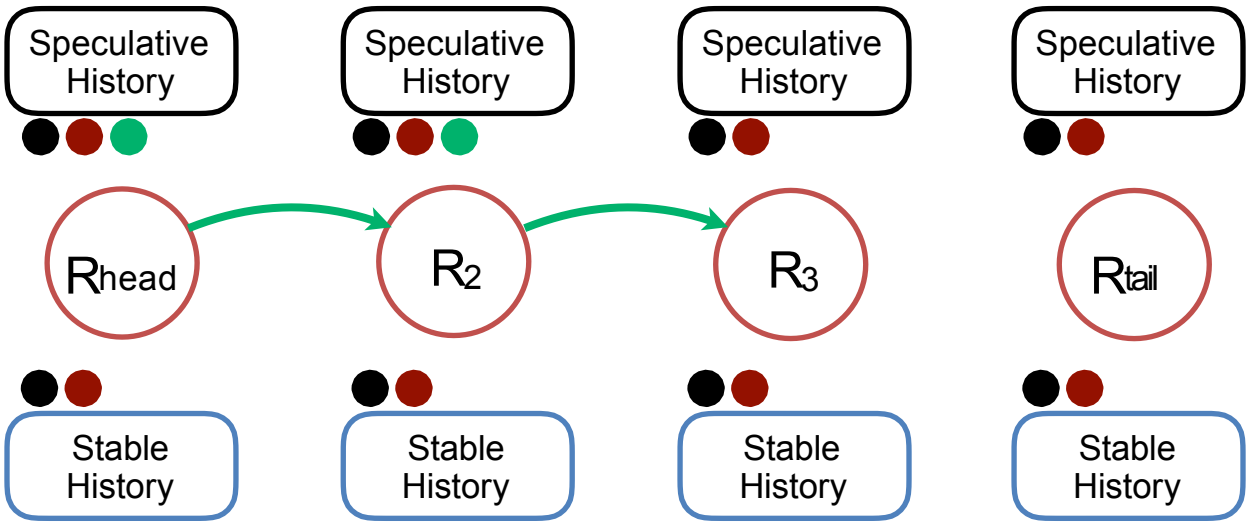
Updates


propagation message
Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

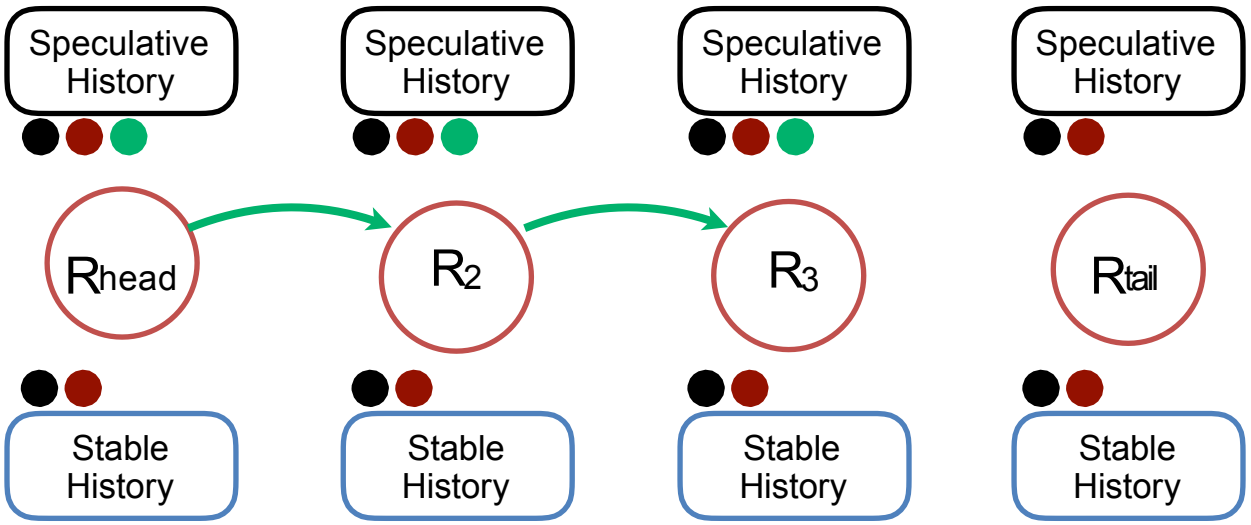
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

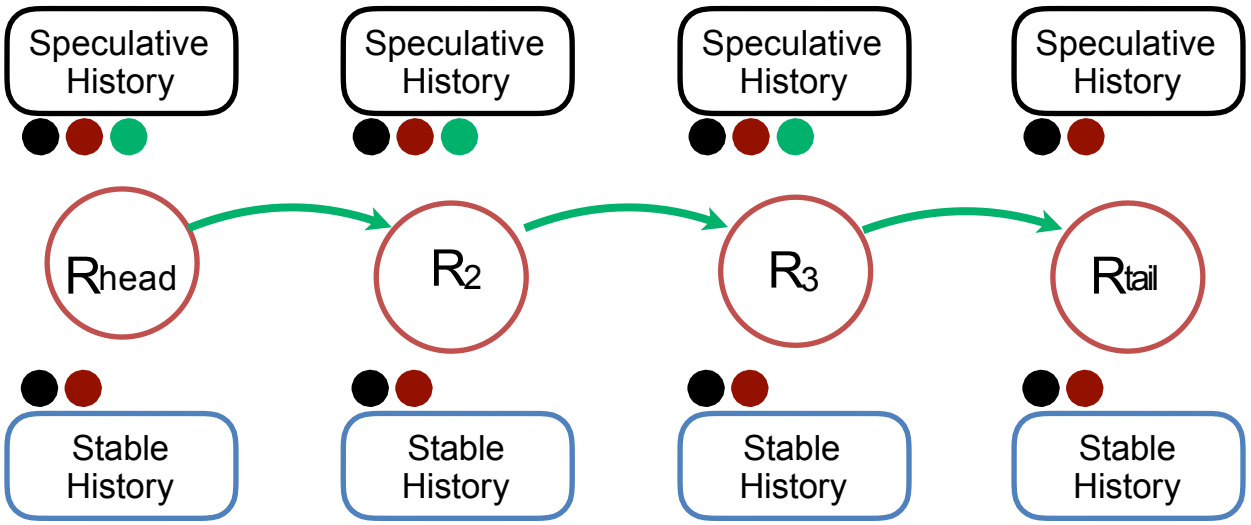
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

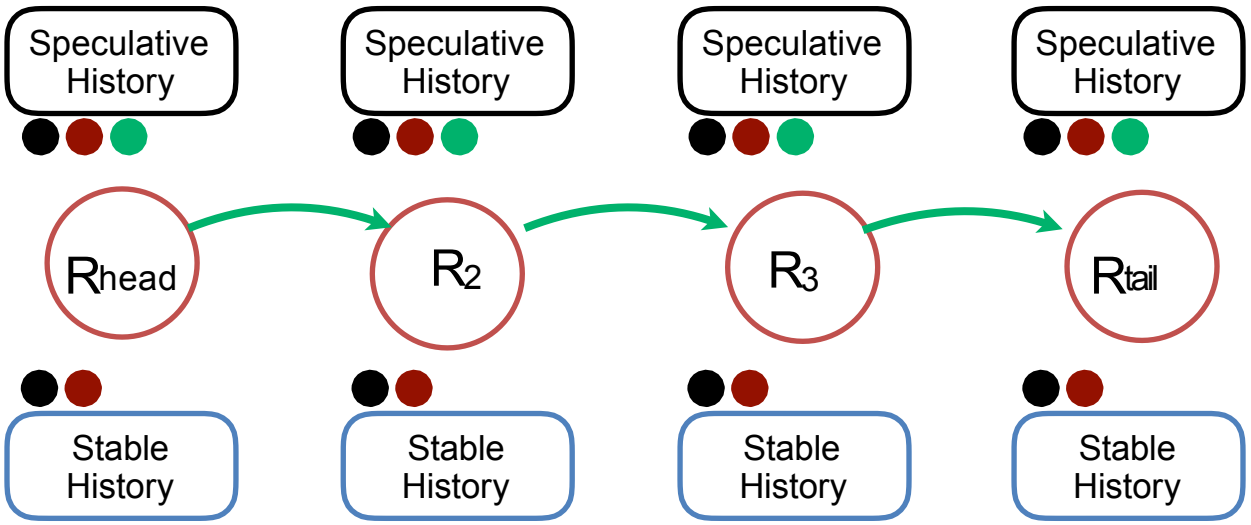
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

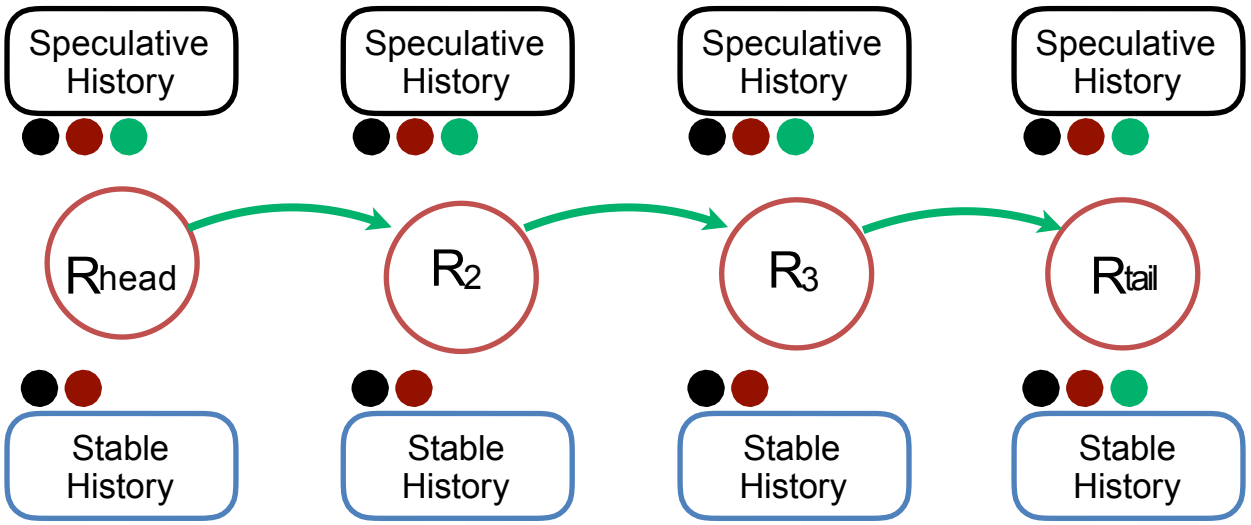
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

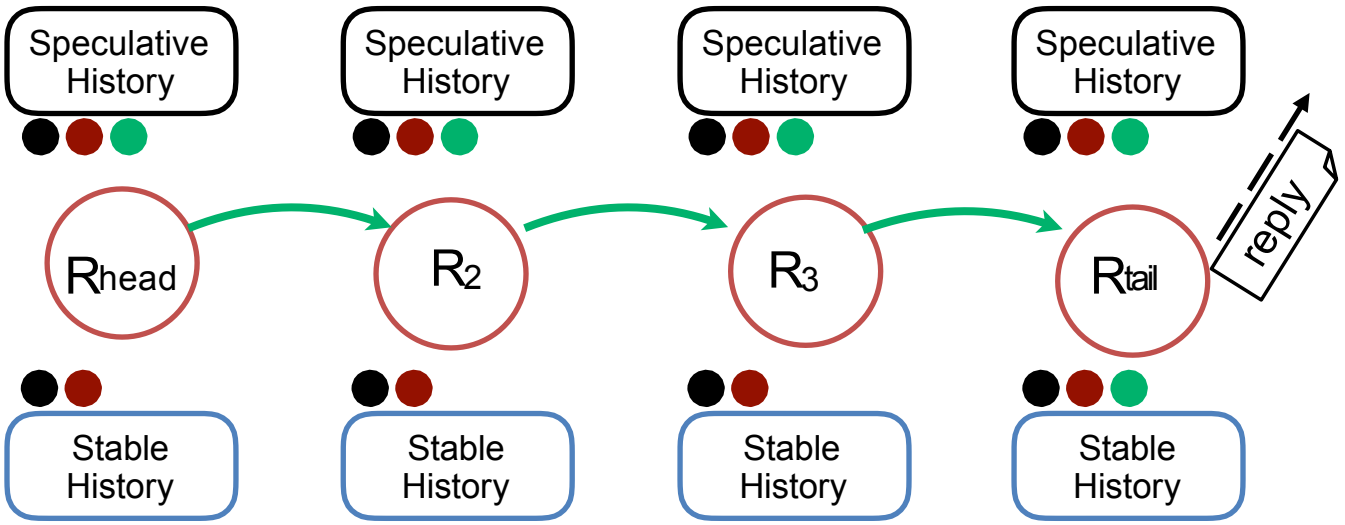
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

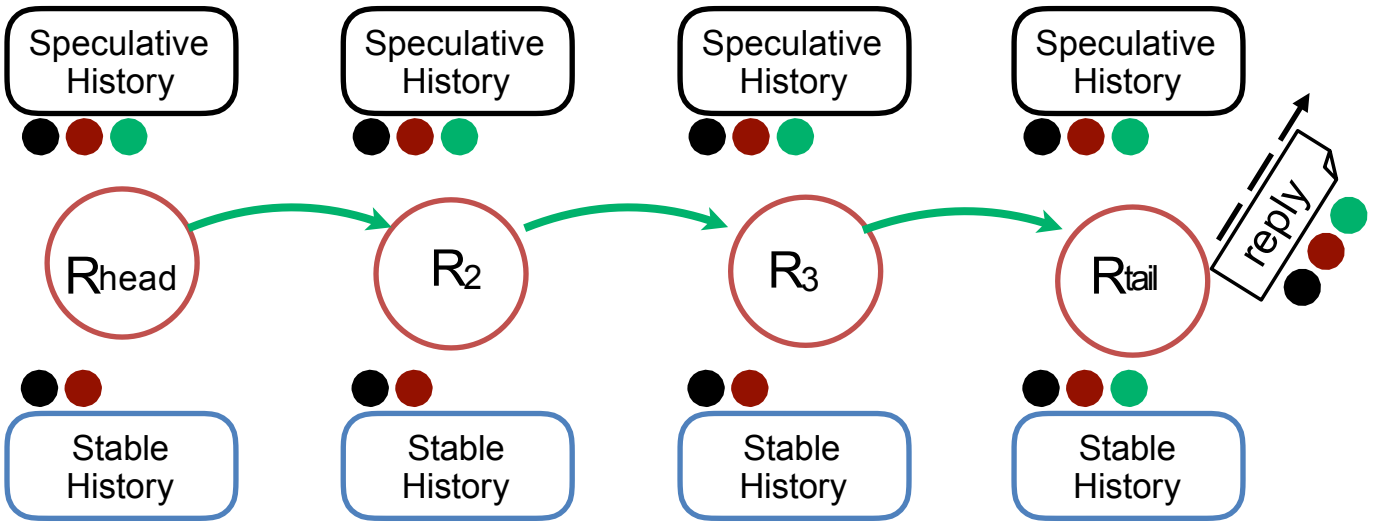
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

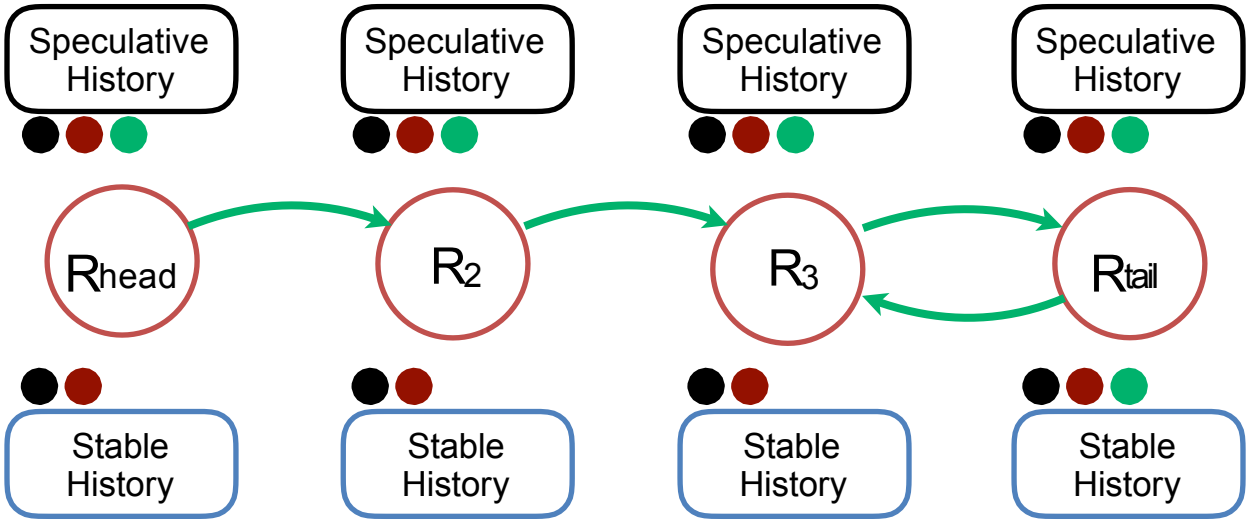
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
reply
Stable History
Stable History
Stable History
Stable History

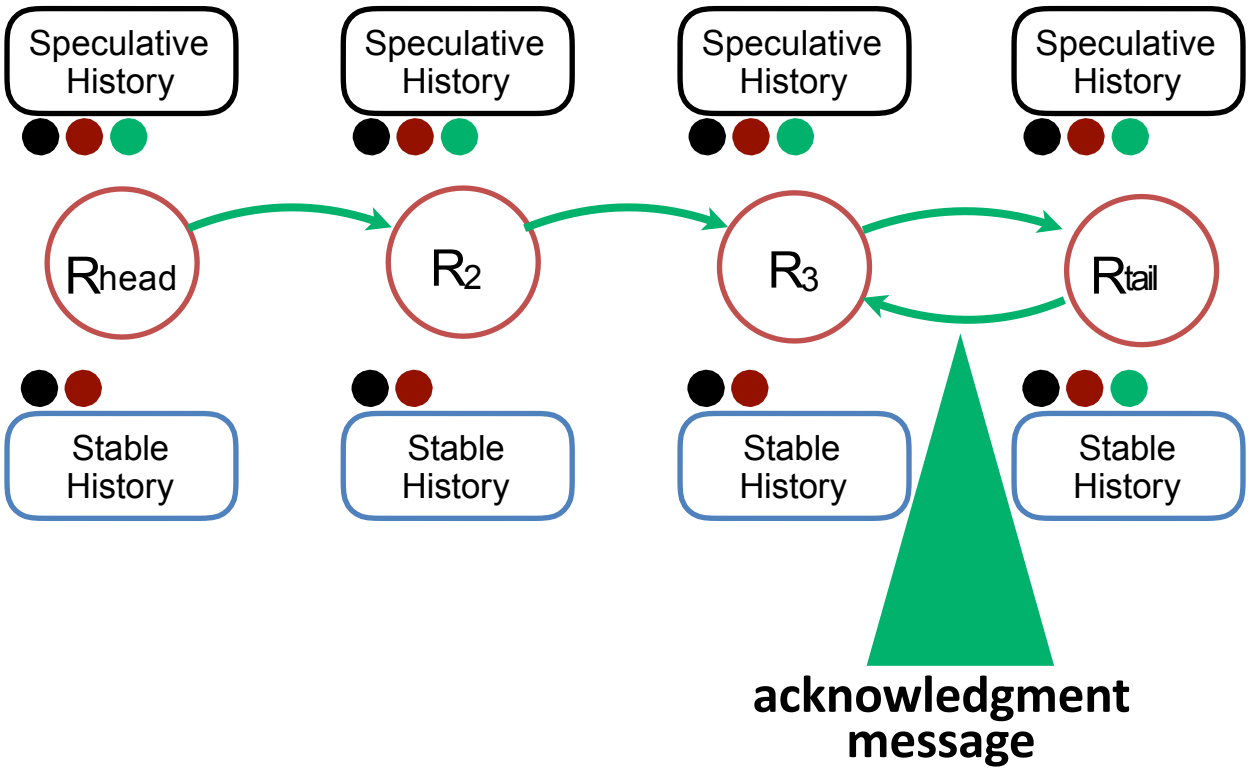
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
reply
Stable History
Stable History
Stable History
Stable History

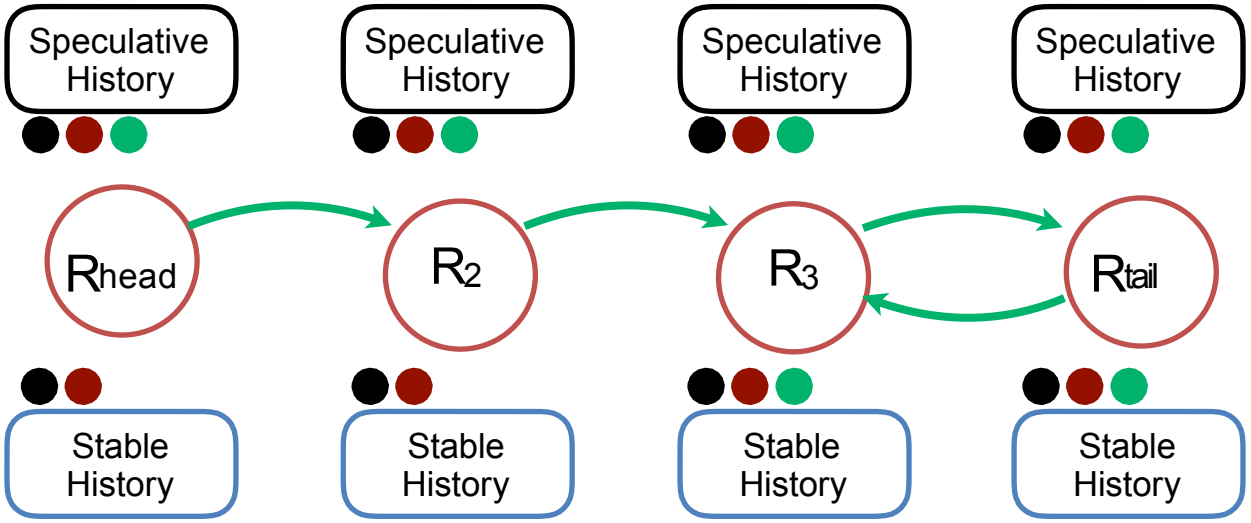
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

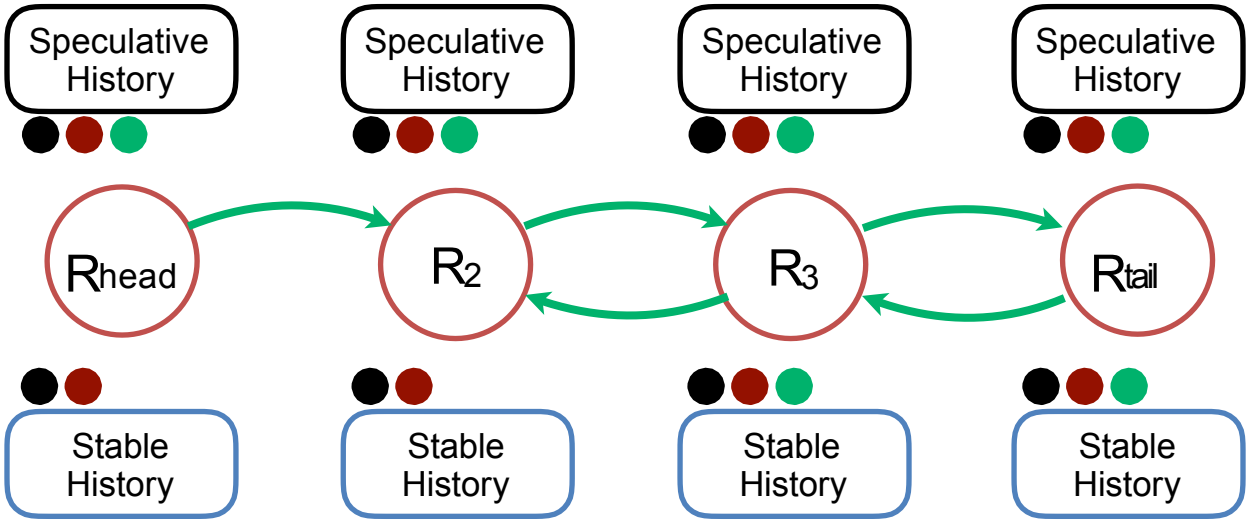
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History
acknowledgment message

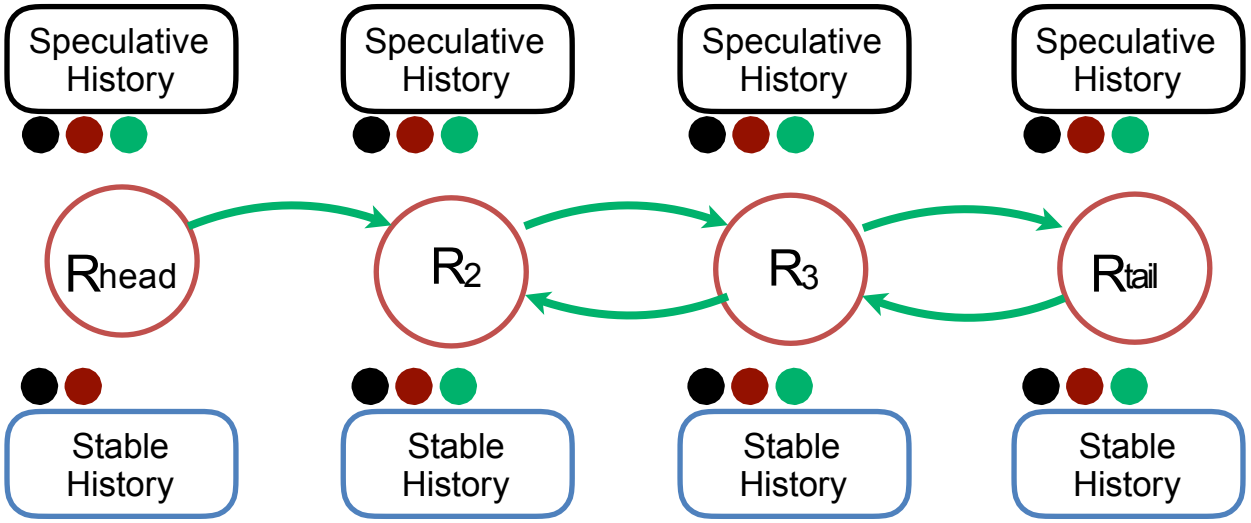
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

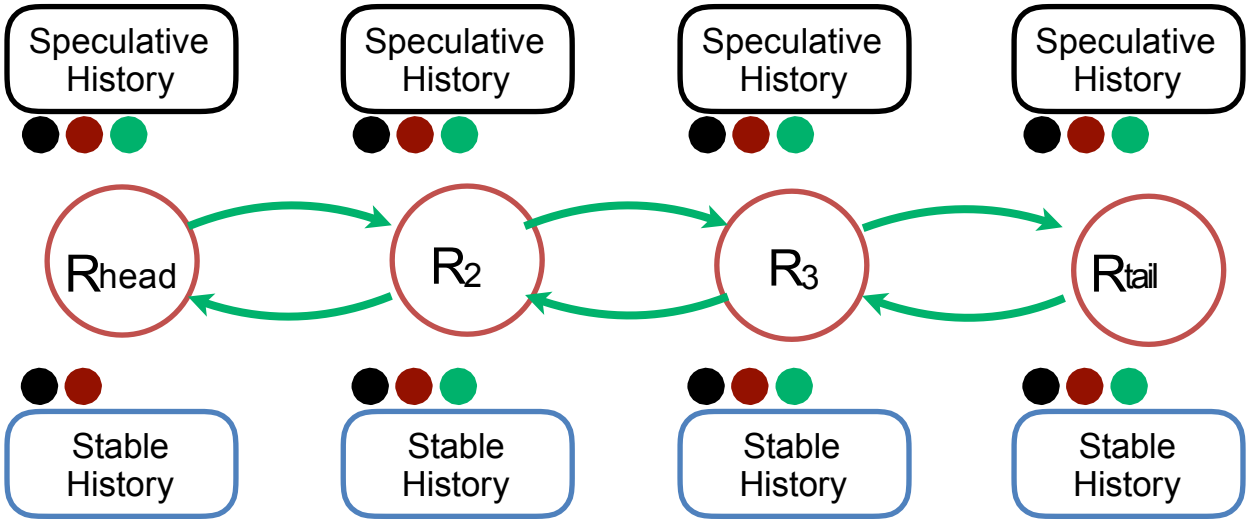
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

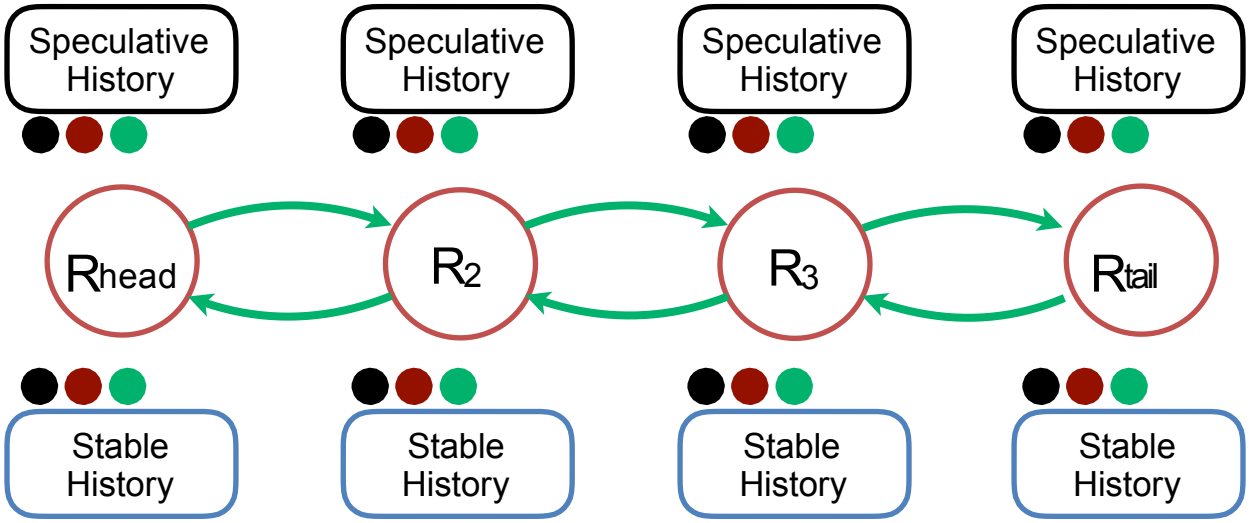
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

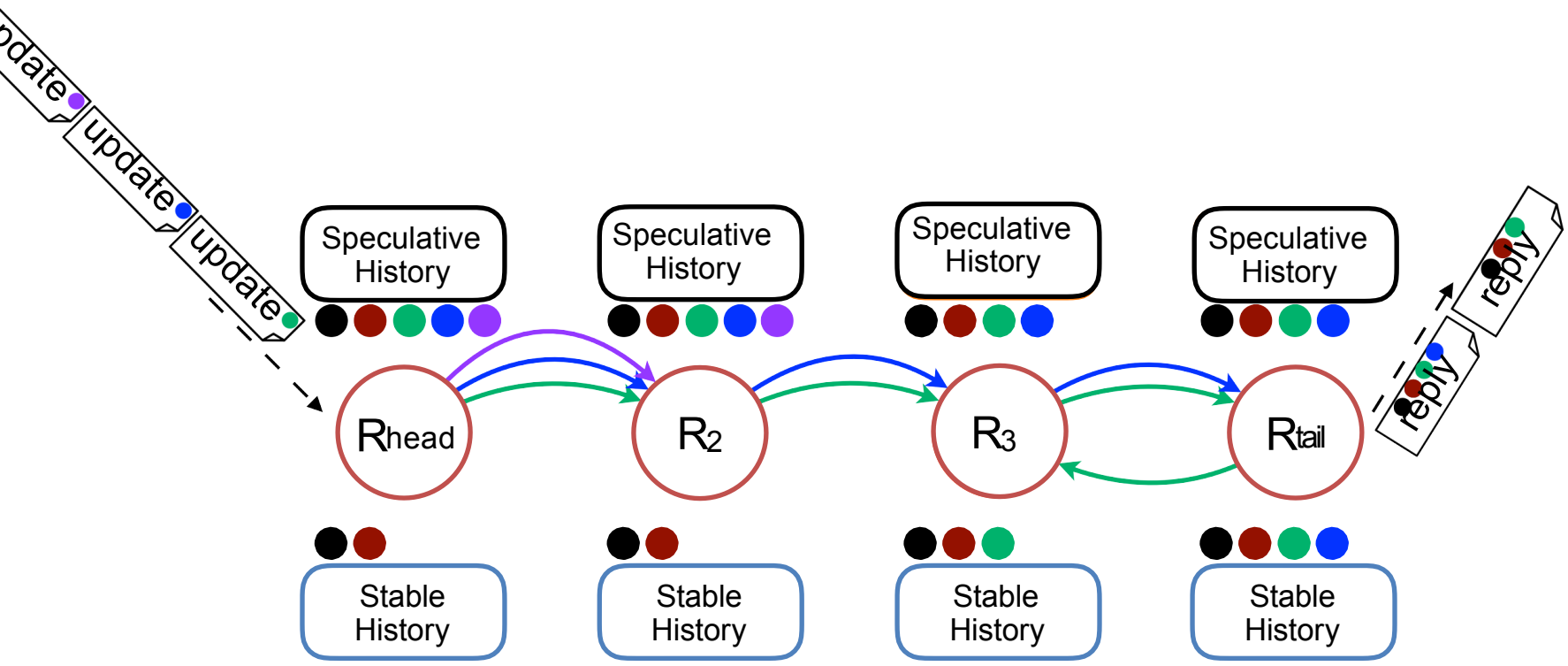
Updates


Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

Updates

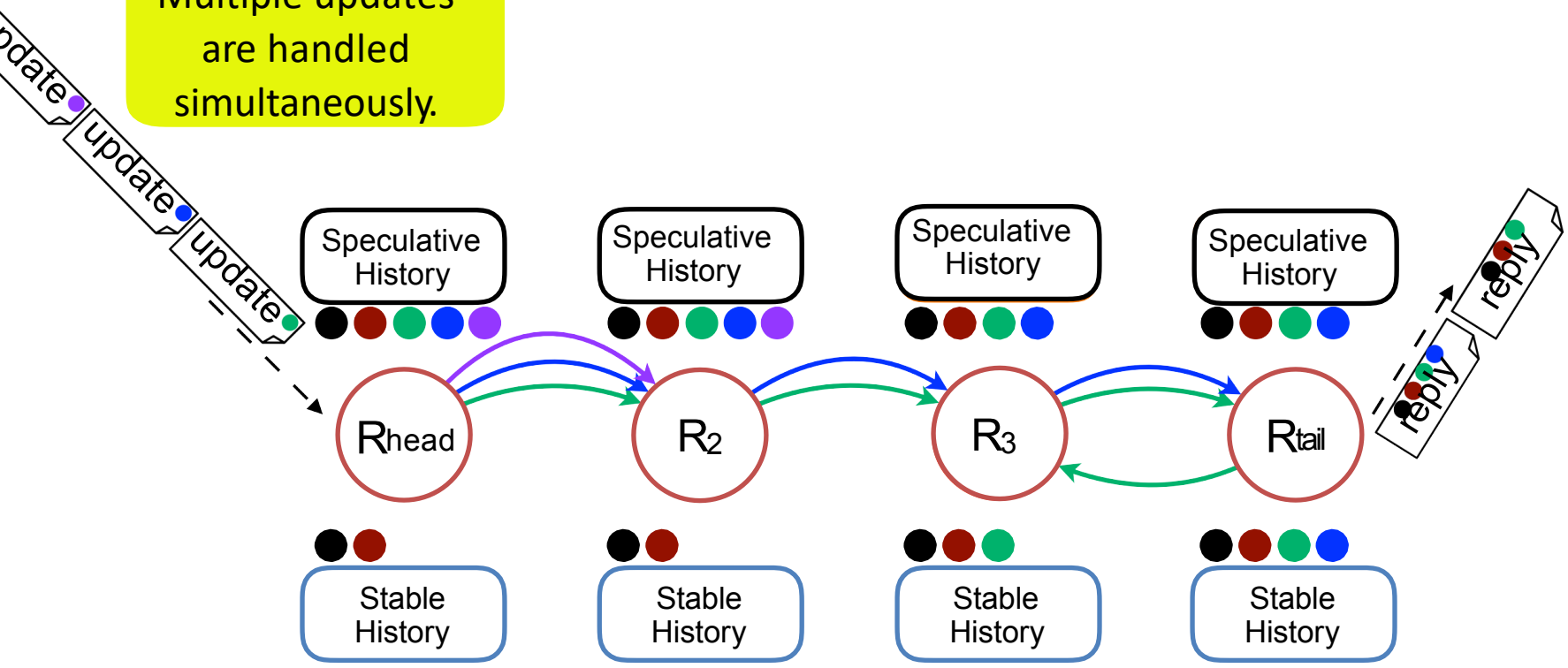

Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

Updates

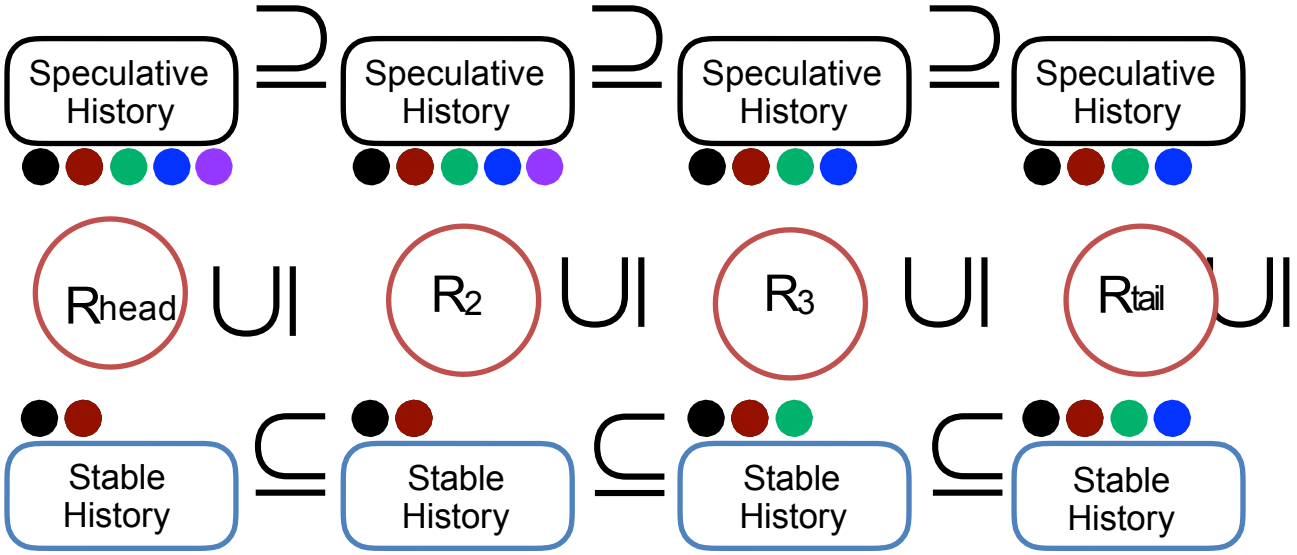

update
update
update
Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
reply
reply
Stable History
Stable History
Stable History
Stable History

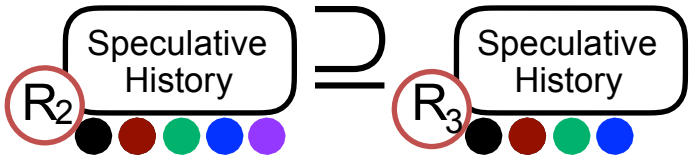
Updates

Multiple updates are handled simultaneously.

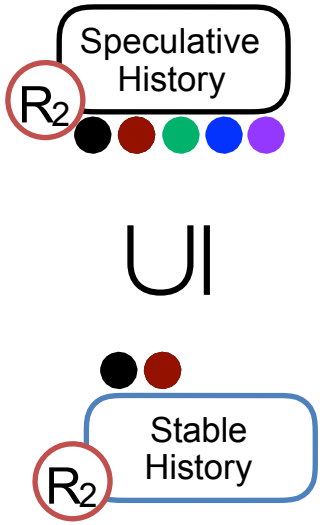


Updates

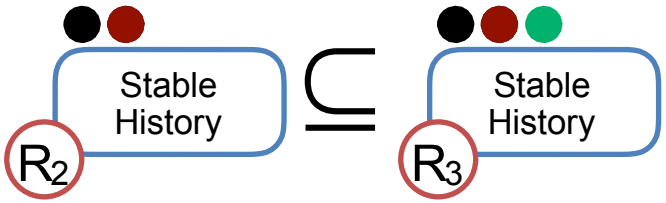

Speculative History
Speculative History
Speculative History
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History



The speculative history of a node's successor is a **subset** of that node's speculative history.

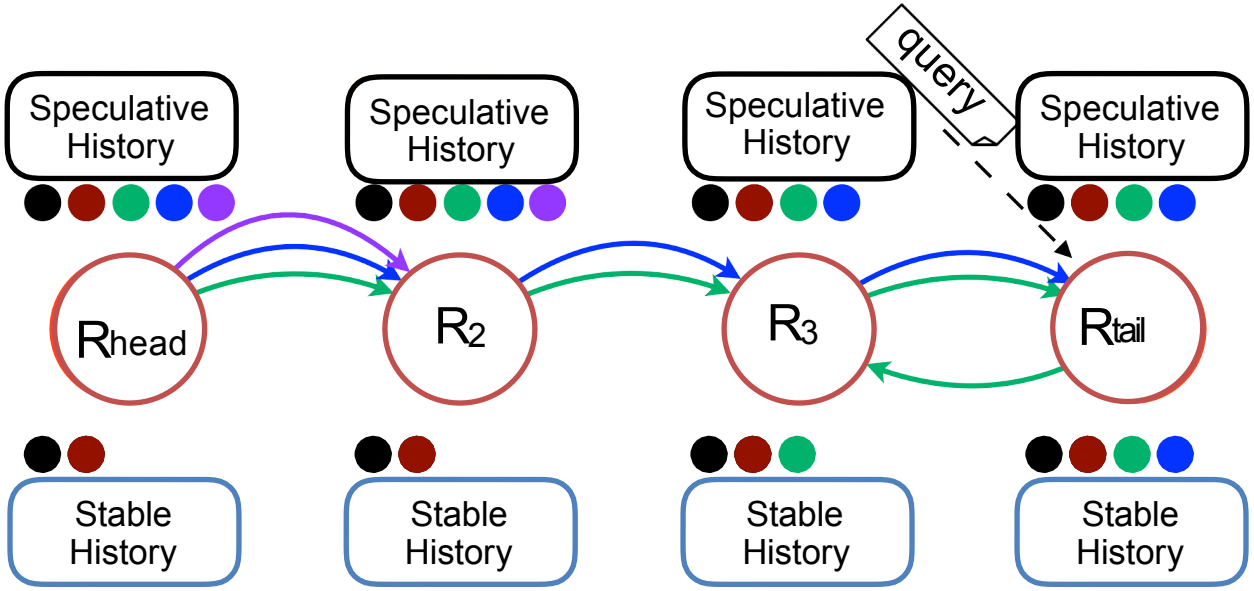


The speculative history of a node is a **superset** of its stable history.

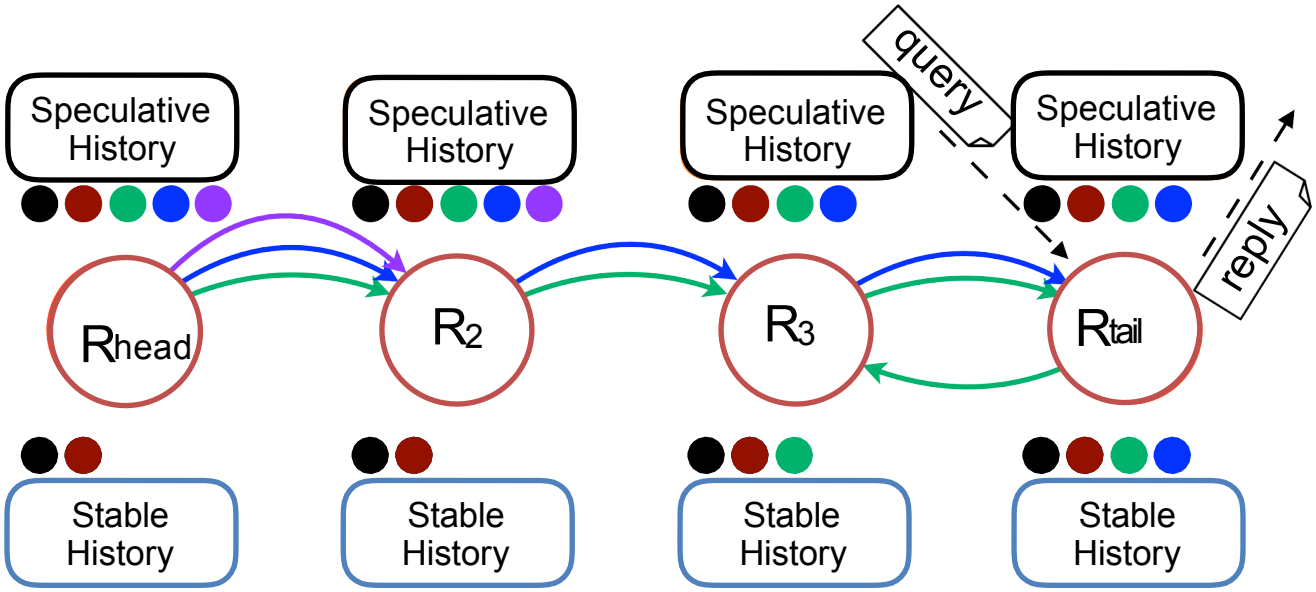


The stable history of a node's successor is a **superset** of that node's stable history.

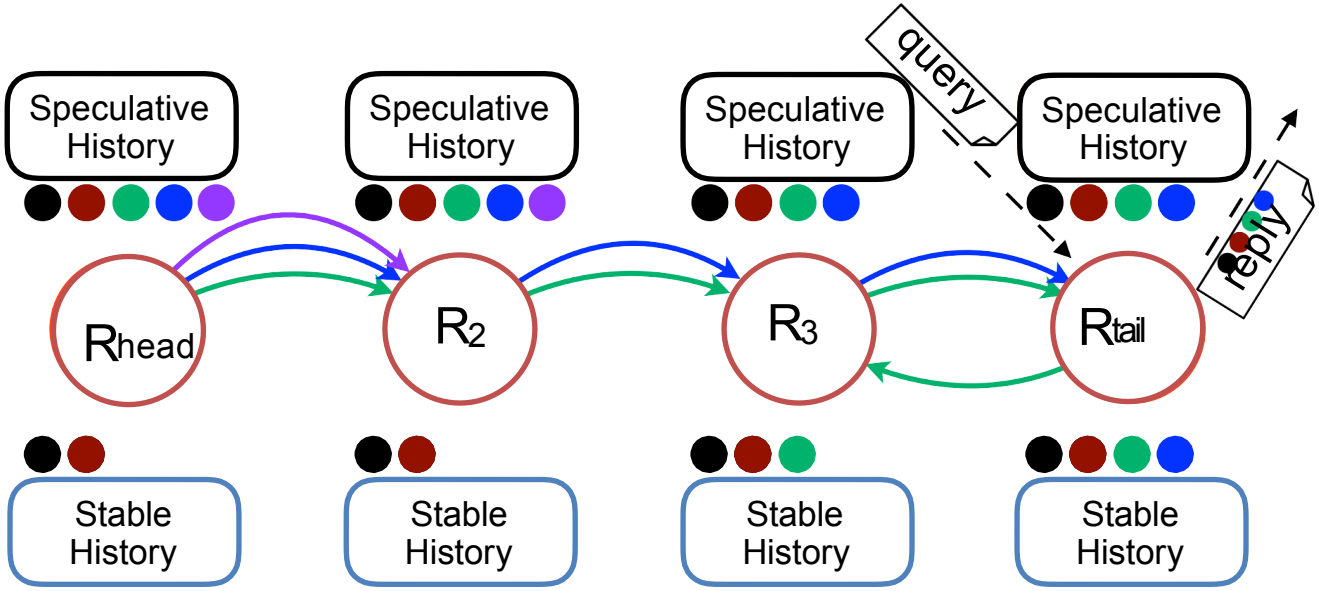
Queries


Speculative History
Speculative History
Speculative History
query
Speculative History
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

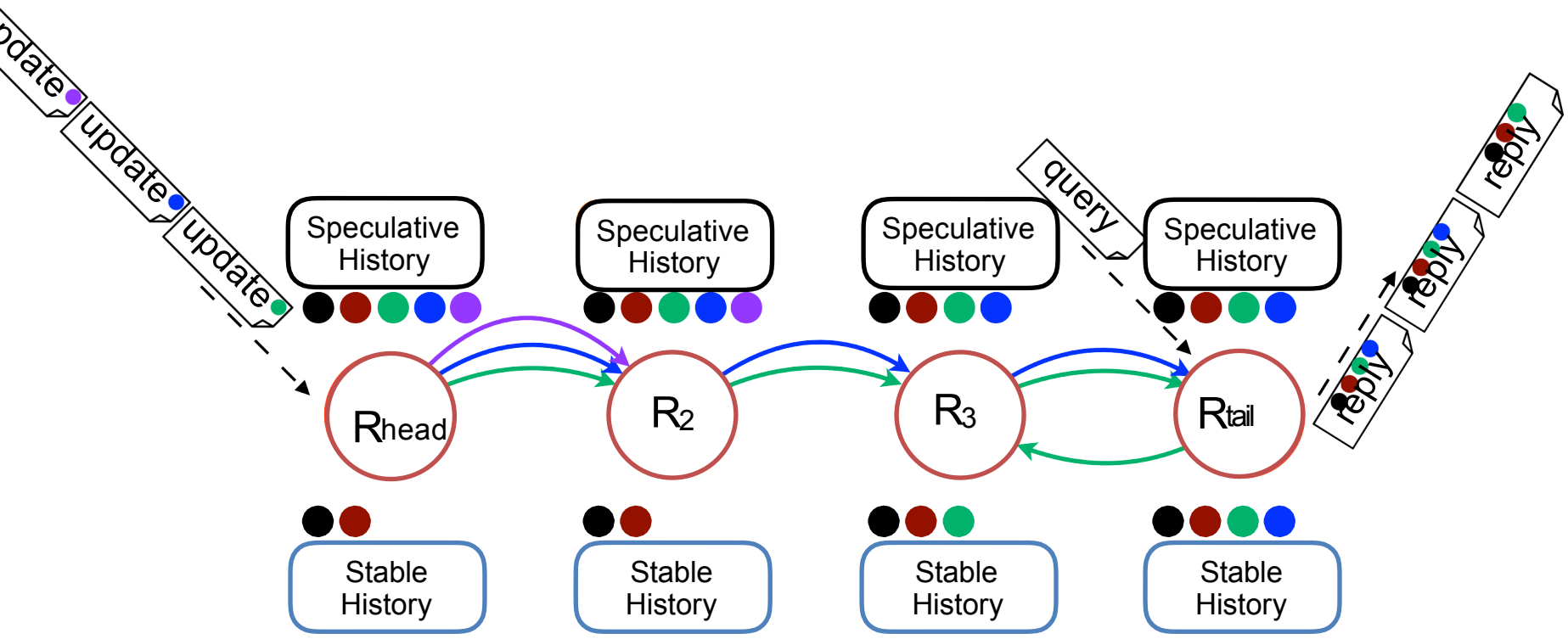
Queries


Speculative History
Speculative History
Speculative History
Speculative History
query
reply
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

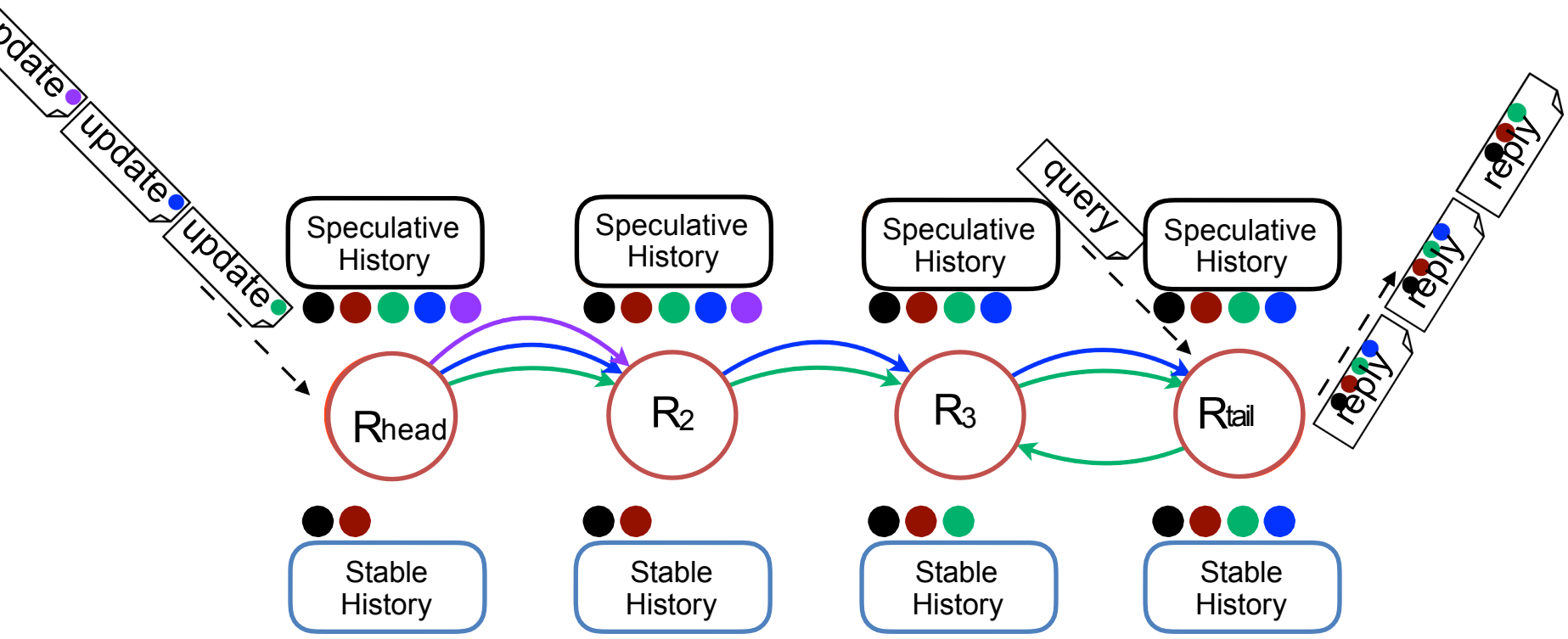
Queries


Speculative History
Speculative History
Speculative History
Speculative History
query
reply
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

Queries


update
update
update
Speculative History
Speculative History
Speculative History
query
Speculative History
reply
reply
reply
Rhead
R2
R3
Rtail
Stable History
Stable History
Stable History
Stable History

Queries



The tail is the point of linearization!

Self-study Questions

- Do speculative histories have to be persisted to disk?
- Do stable histories have to be persisted to disk?
- While the subset relationships are neat, do they serve an ulterior purpose?
- Could a reply to an update be sent before transitioning the update to the stable history of the tail?
- Can chain replication work effectively with a single replica?

Pixabay.com

CHAIN REPLICATION

FAILUR HANDLING & RECONFIGURATIONS

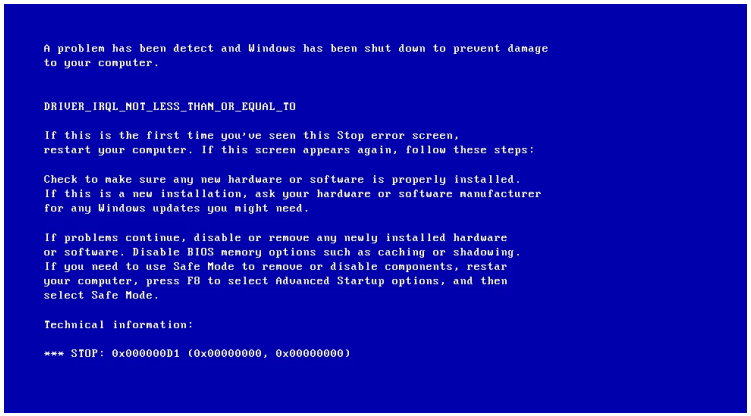
Inspiration for this lecture taken from a talk given by Deniz Altınbüken.

Failure Handling & Reconfigurations

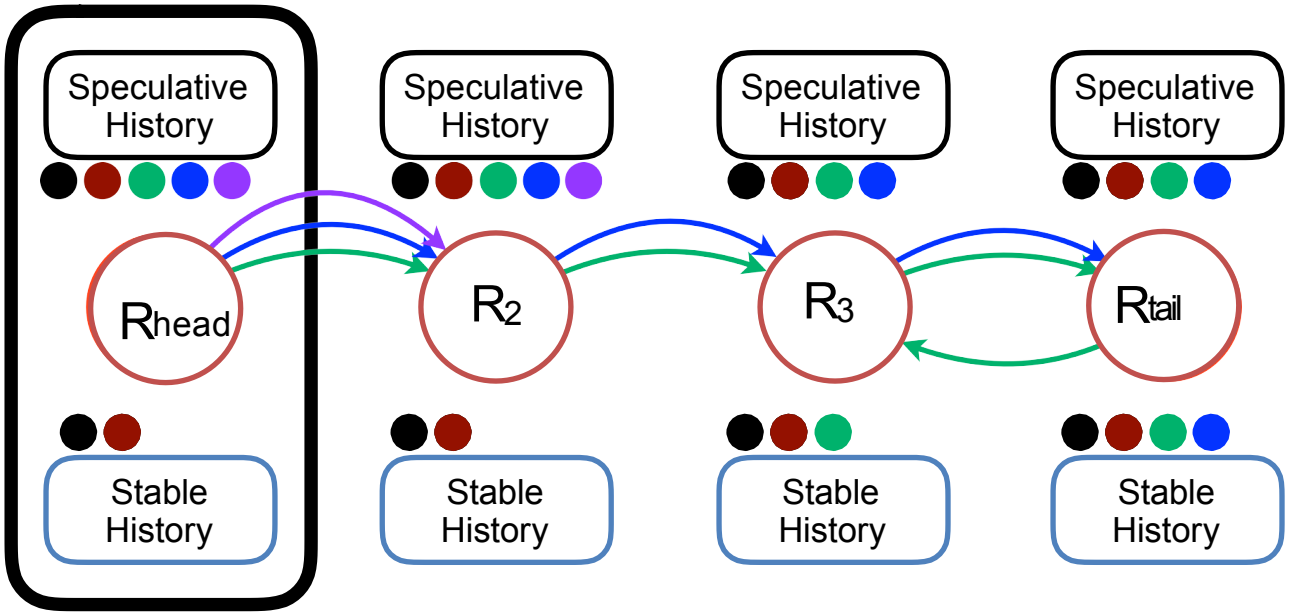
- Chain replication operations:
  - Updates
  - Queries
  - Failures
  - Reconfigurations

Failures

- Head failure
- Middle node failure
- Tail failure

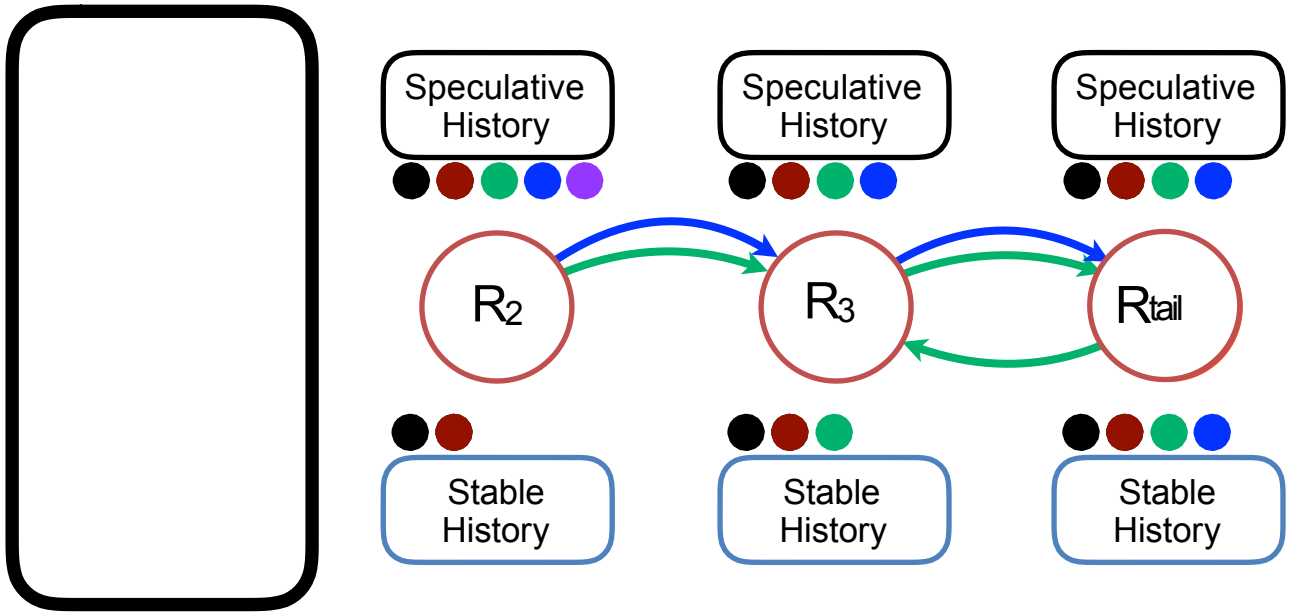


Head Failure I



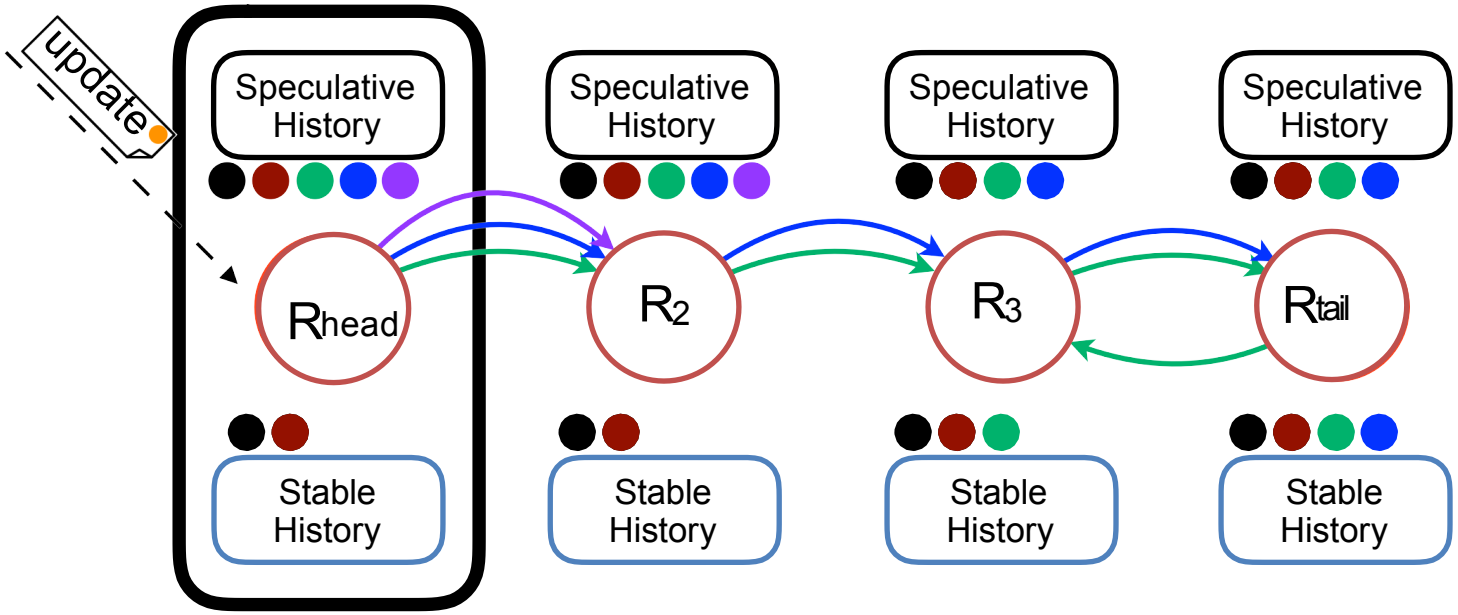
Head Failure I

R_2 becomes new head



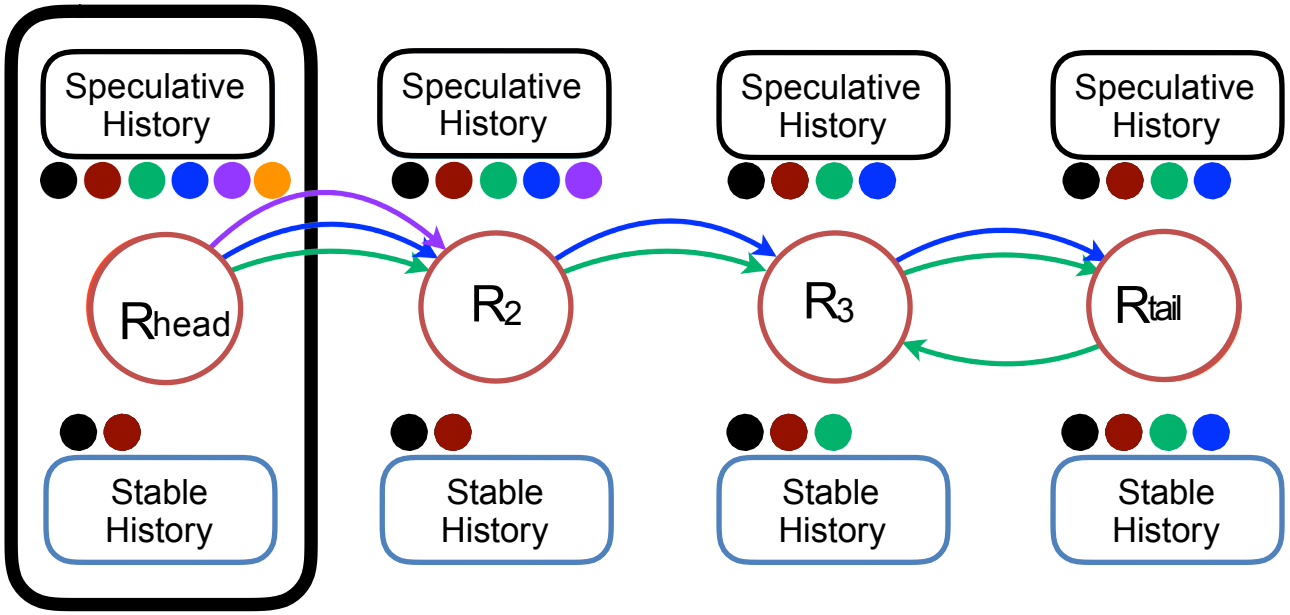
Head Failure I

In-flight, non-propagated updates



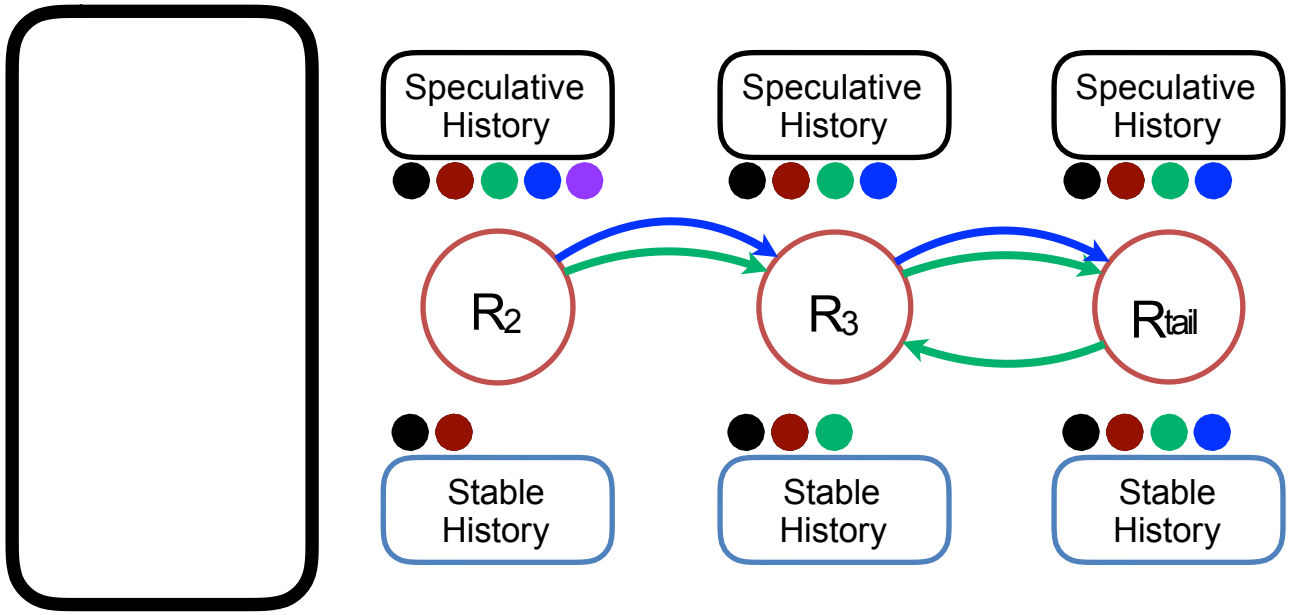
Head Failure I

In-flight, non-propagated updates



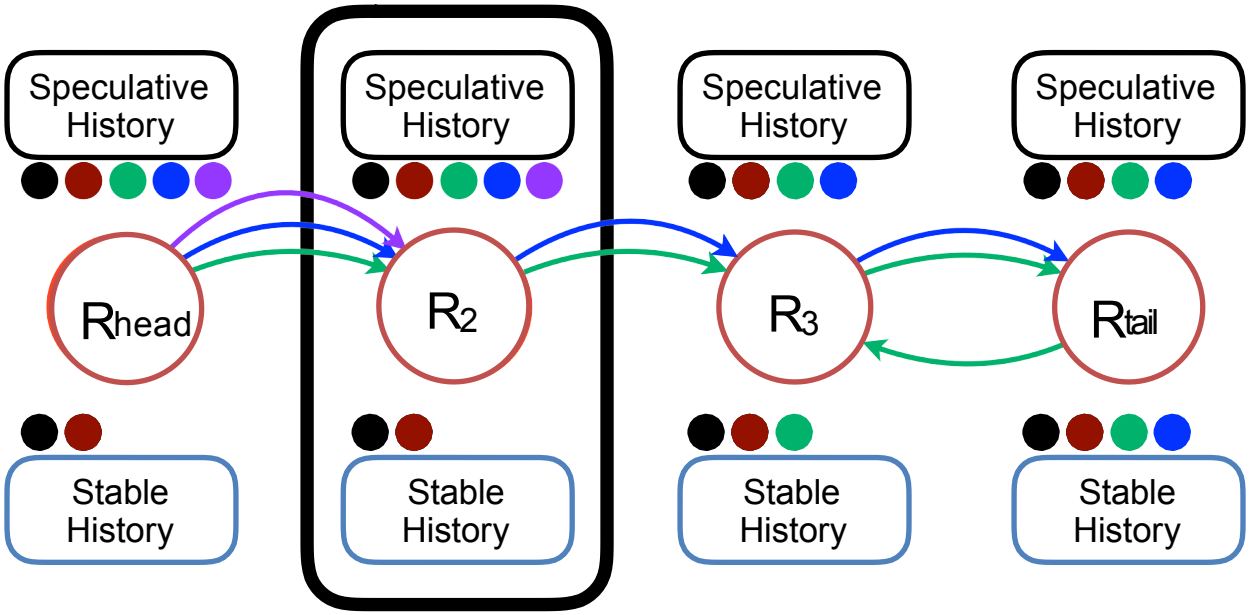
Head Failure I

In-flight, non-propagated updates

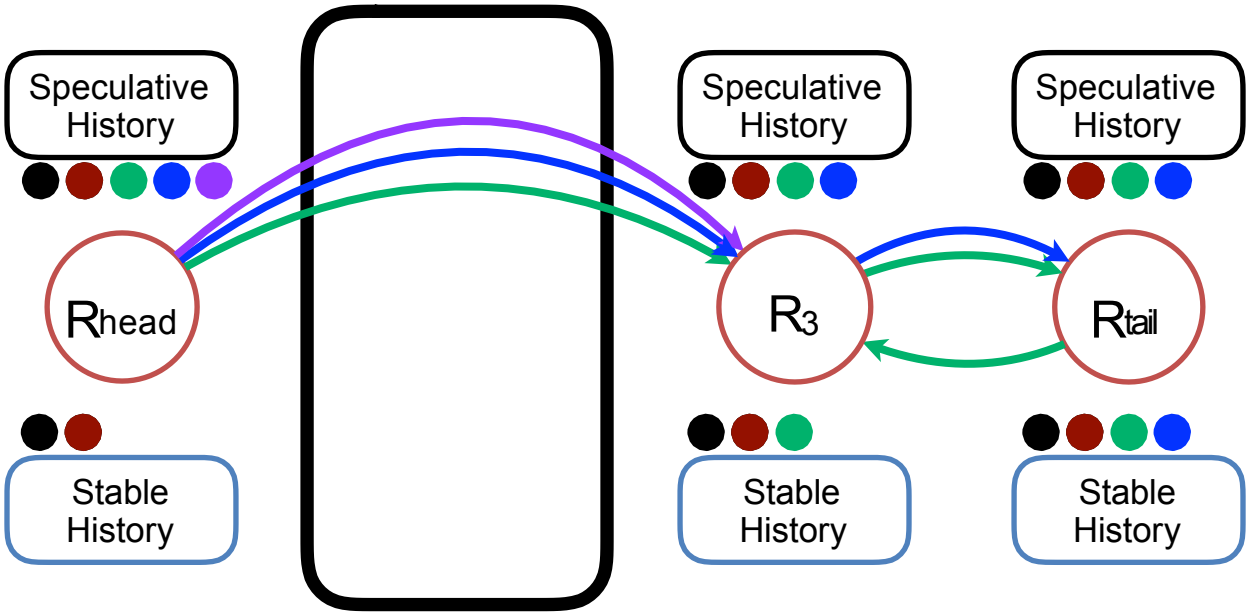


Client would not receive a reply, timeout, and retry

Middle Node Failure II

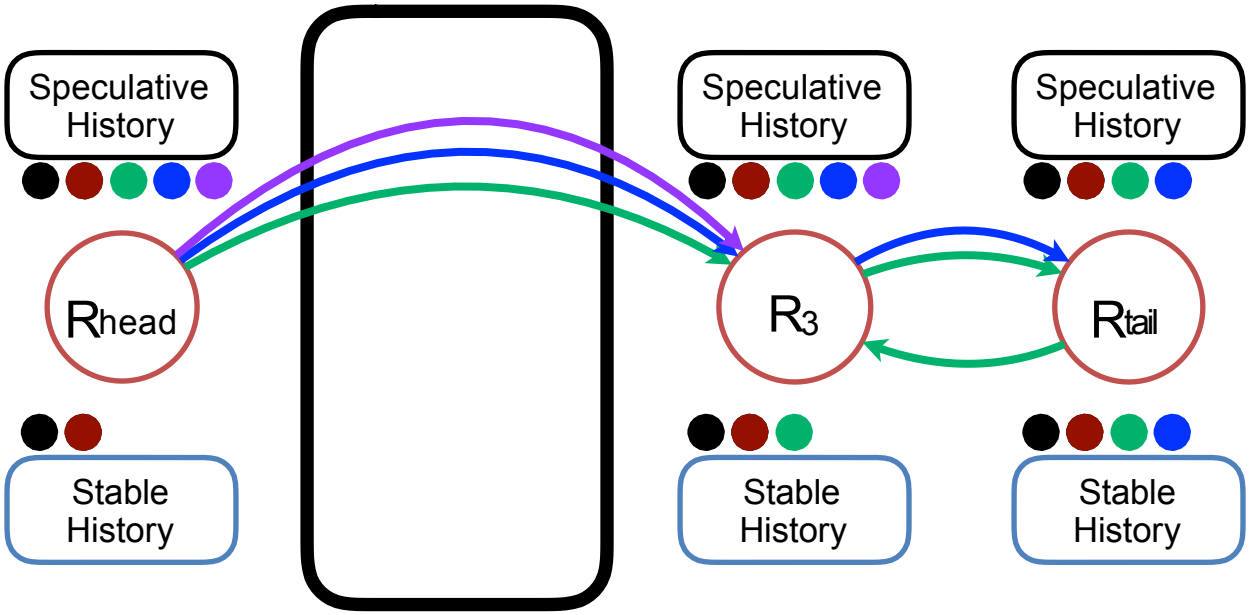


Middle Node Failure II



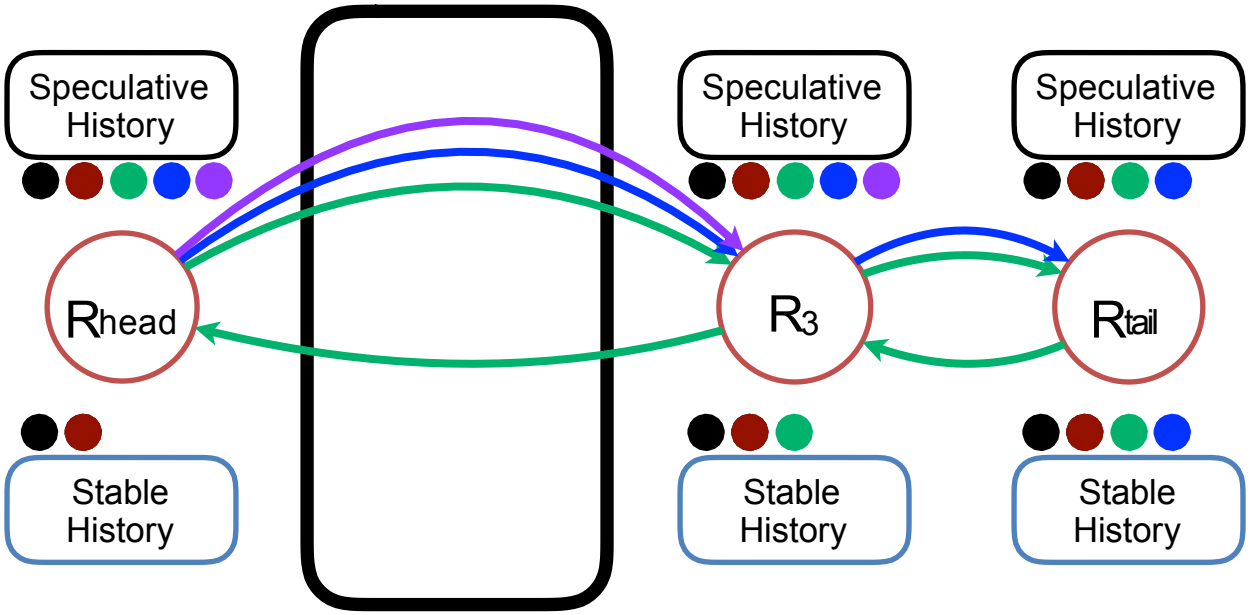
Predecessor needs to talk to failed node's successor

Middle Node Failure II

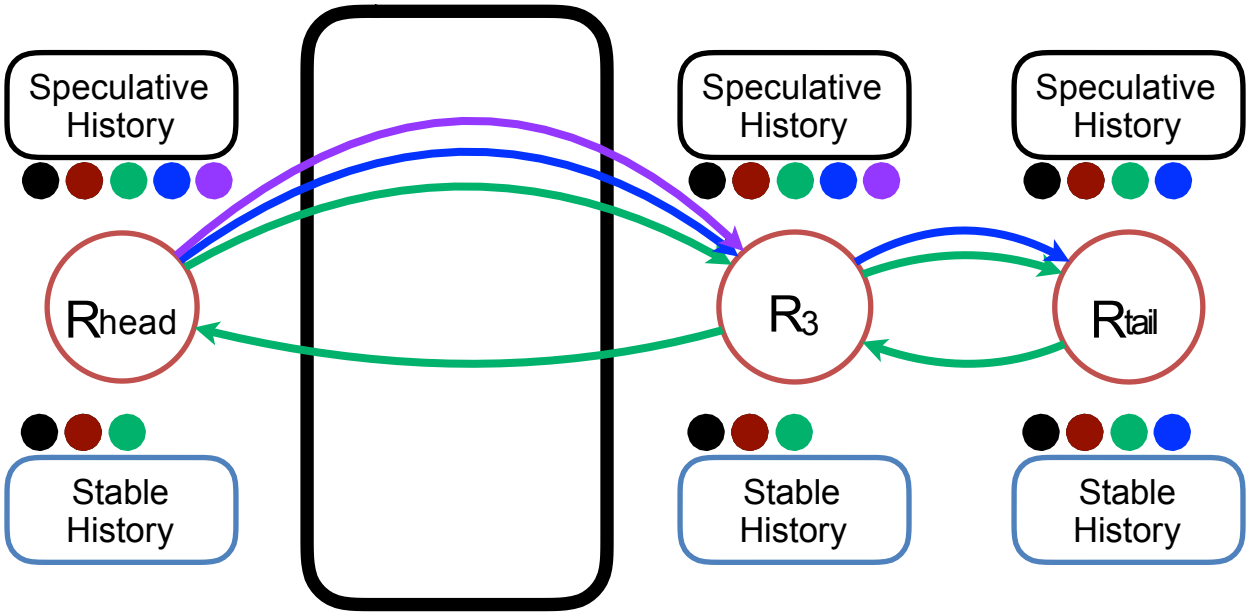


Predecessor propagates update to new successor

Middle Node Failure II

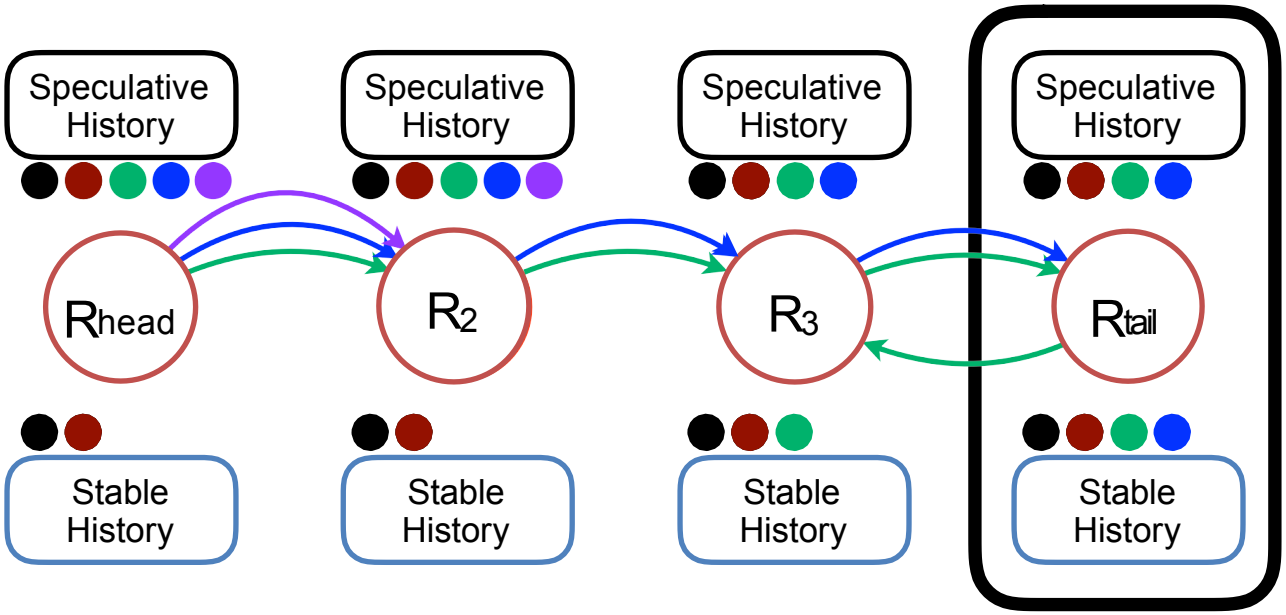


Middle Node Failure II

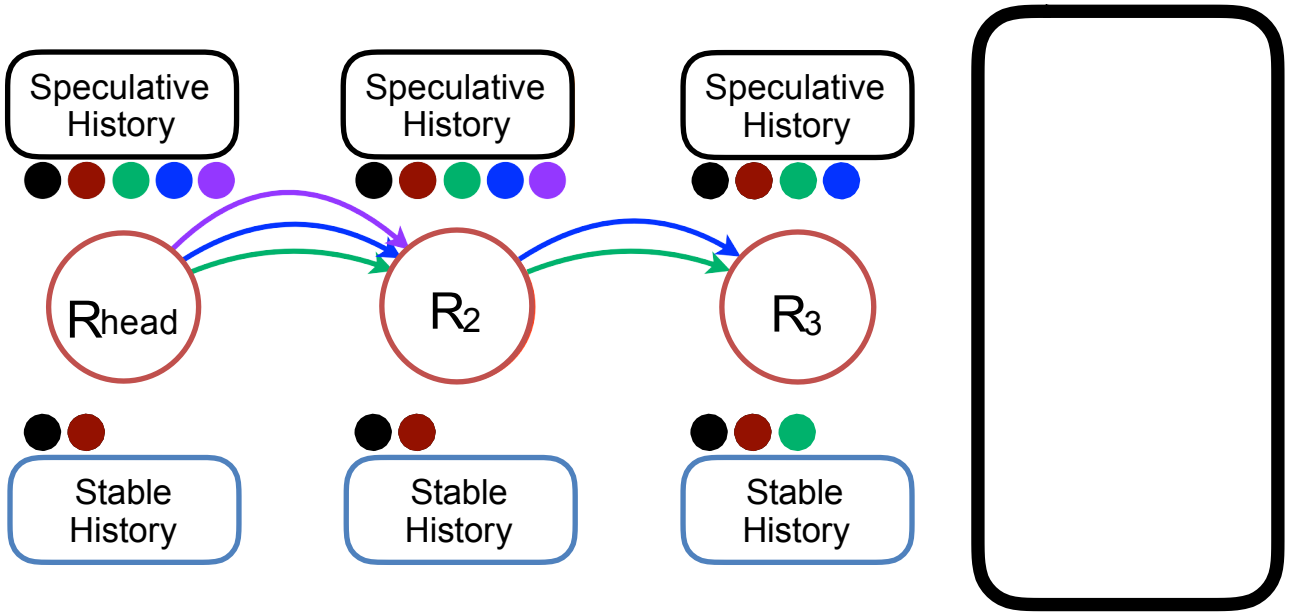


Successor propagates in-flight acknowledgements to new predecessor

Tail Failure III

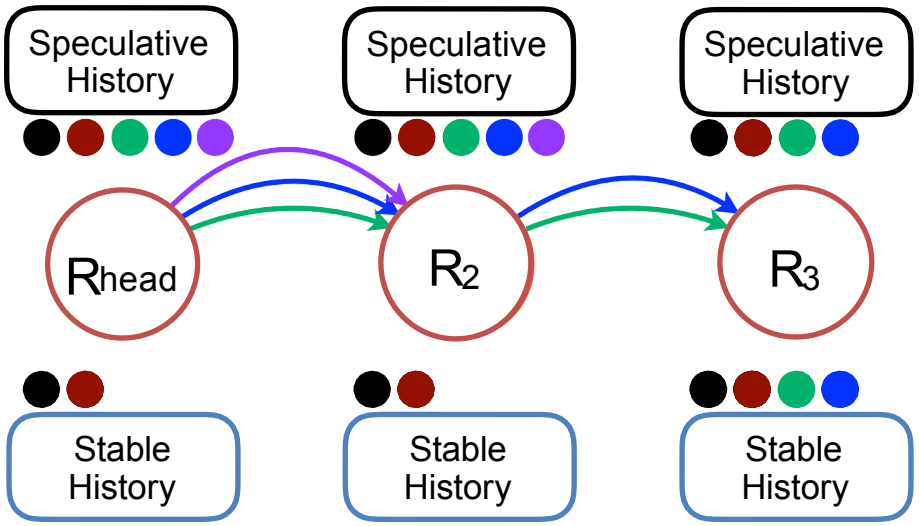


Tail Failure III



R_3 becomes new tail

Tail Failure III



R_3 flushes its speculative history s.t. stable equals speculative history again

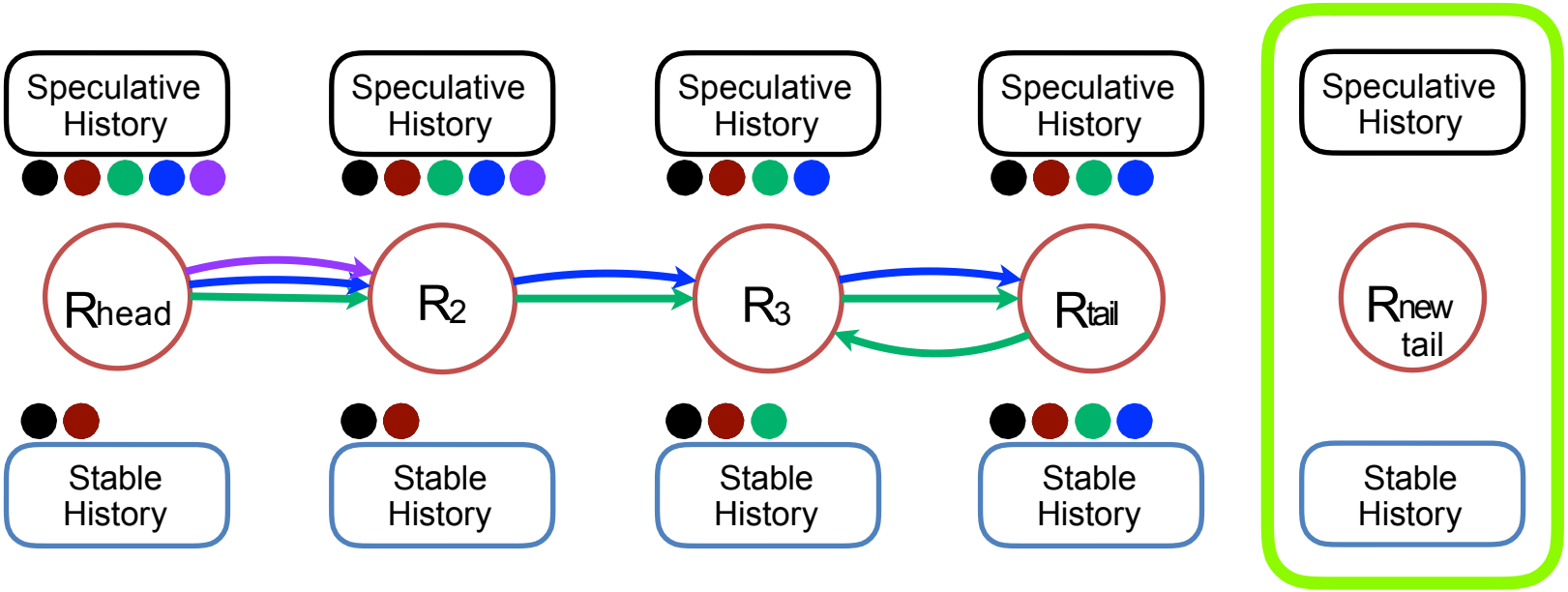
Reconfigurations

- Adding a new node for failure recovery
- Adding a new node to extend topology
- Setting up a new chain of replicas

Adding a New Node I

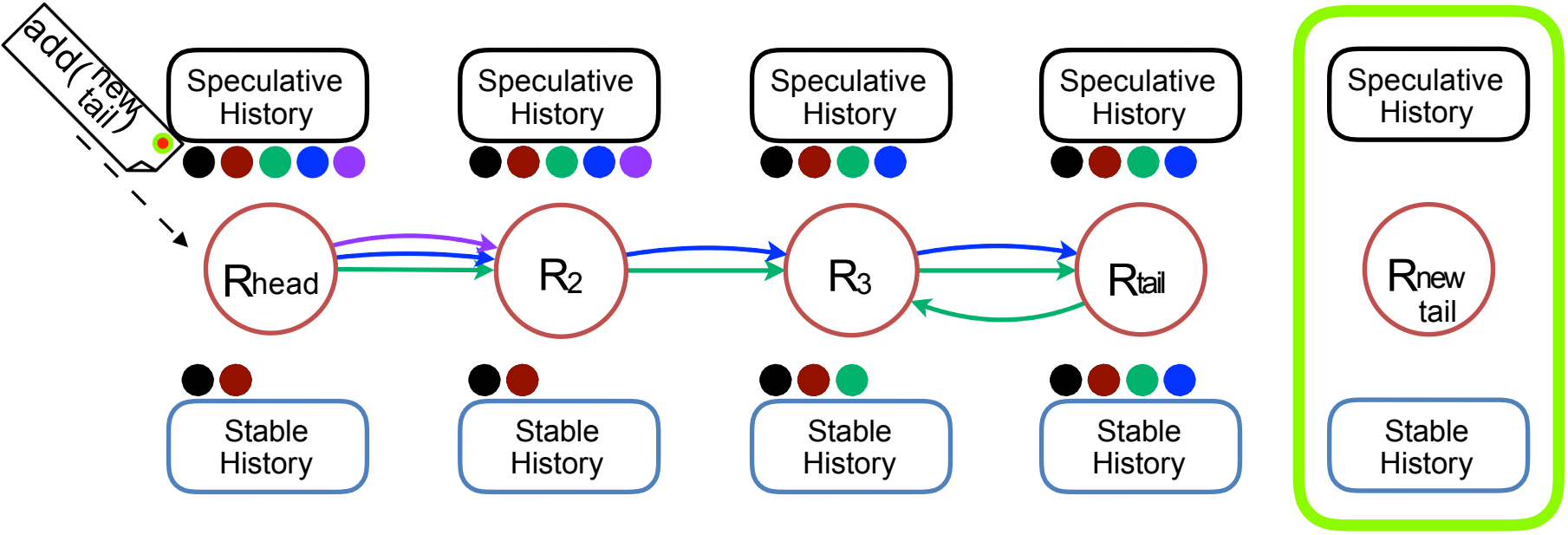
A Configuration Change

Adding an initially empty node



Adding a New Node II

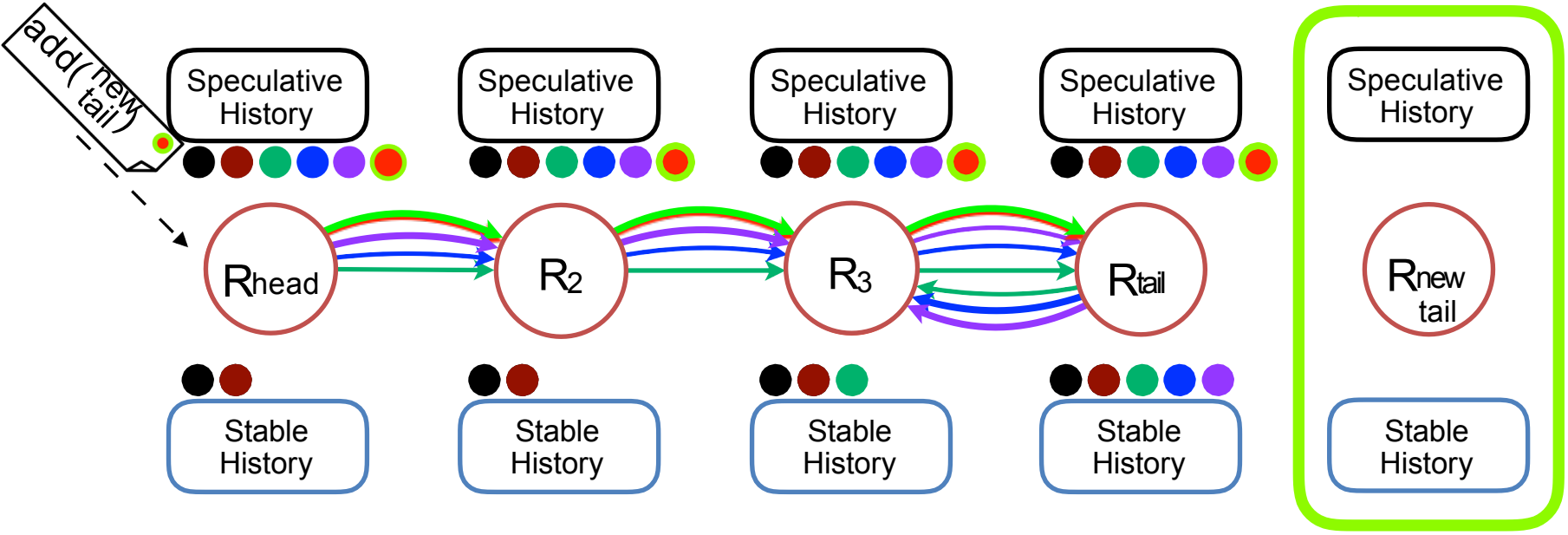
A Configuration Change



- New nodes are added to chain with special **configuration updates**, added to histories: **add(nodeid)**
- Entire chain is build in this manner

Adding a New Node

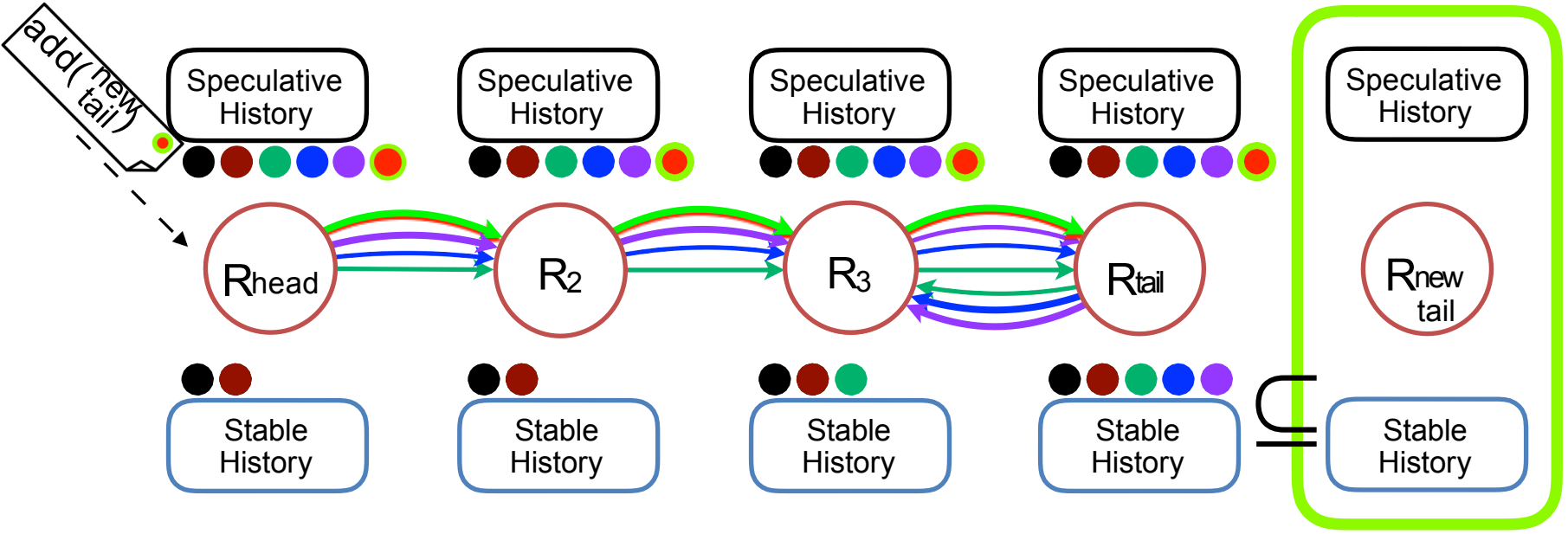
Inferring Configuration



- By looking at **order of these updates**, a node can **determine configuration of chain**
- Old tail discovers it no longer is the tail (via receipt of)

Adding a New Node I

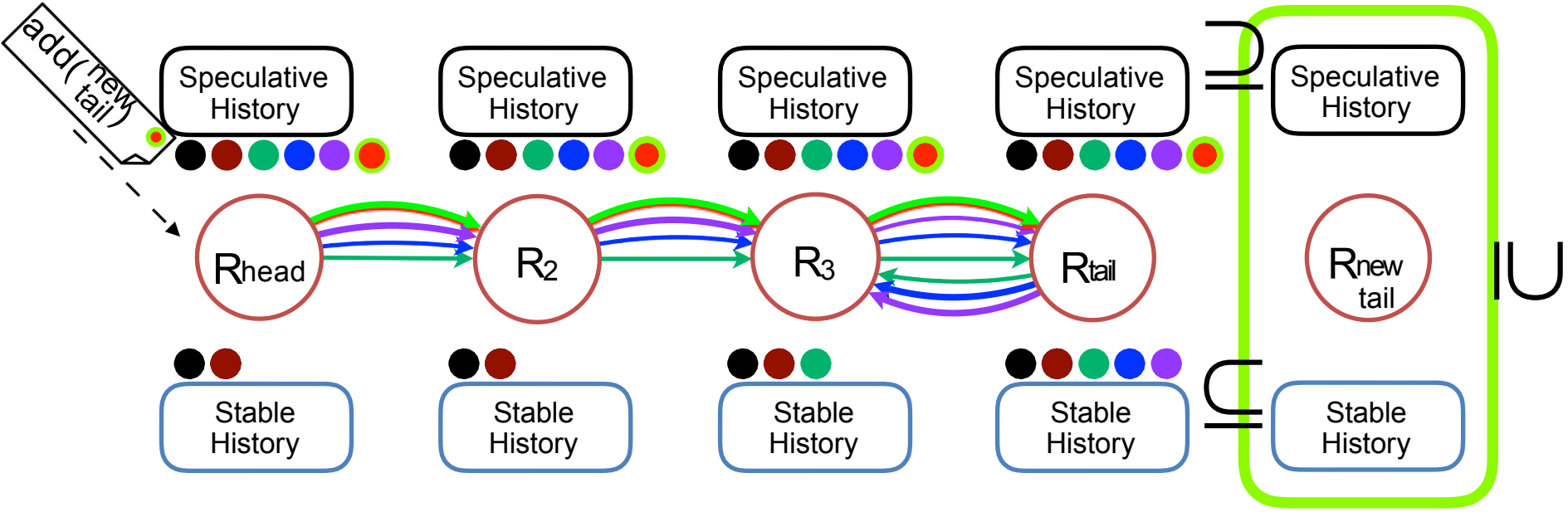
Relationship Among Histories



- **Stable history** of new tail should be **superset of stable history of old tail**

Adding a New Node II

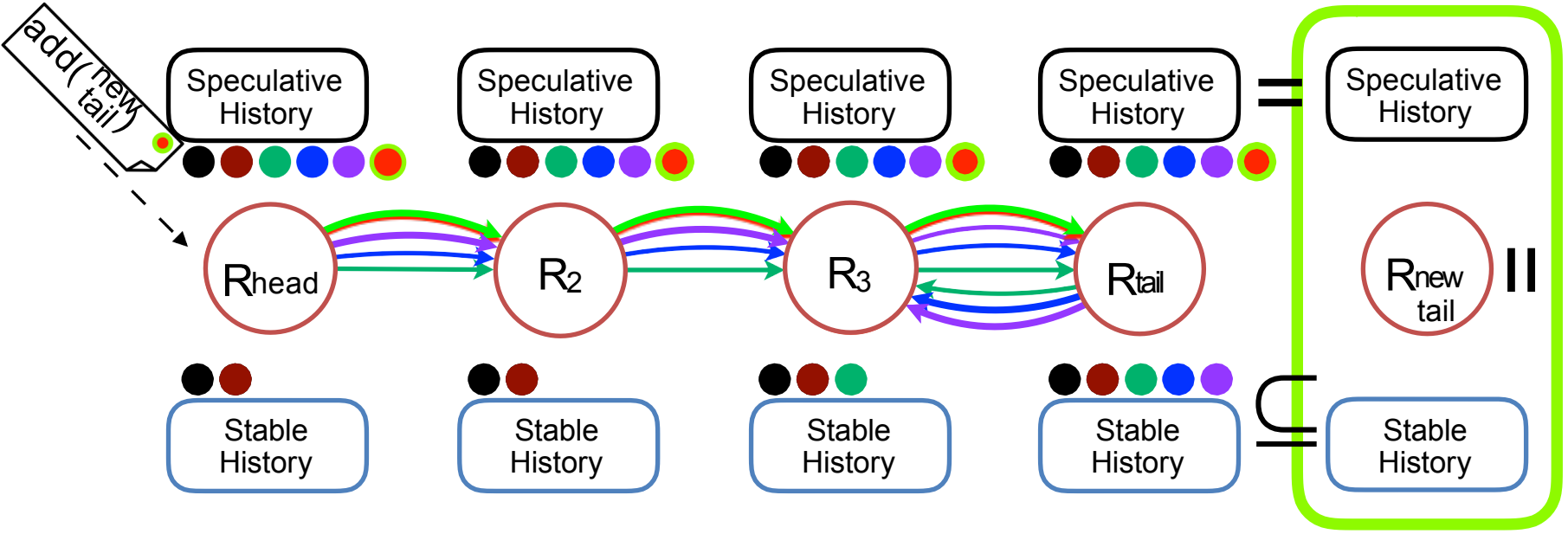
Relationship Among Histories



- **Speculative history** of new tail should be a **superset** of **its stable history**

Adding a New Node III

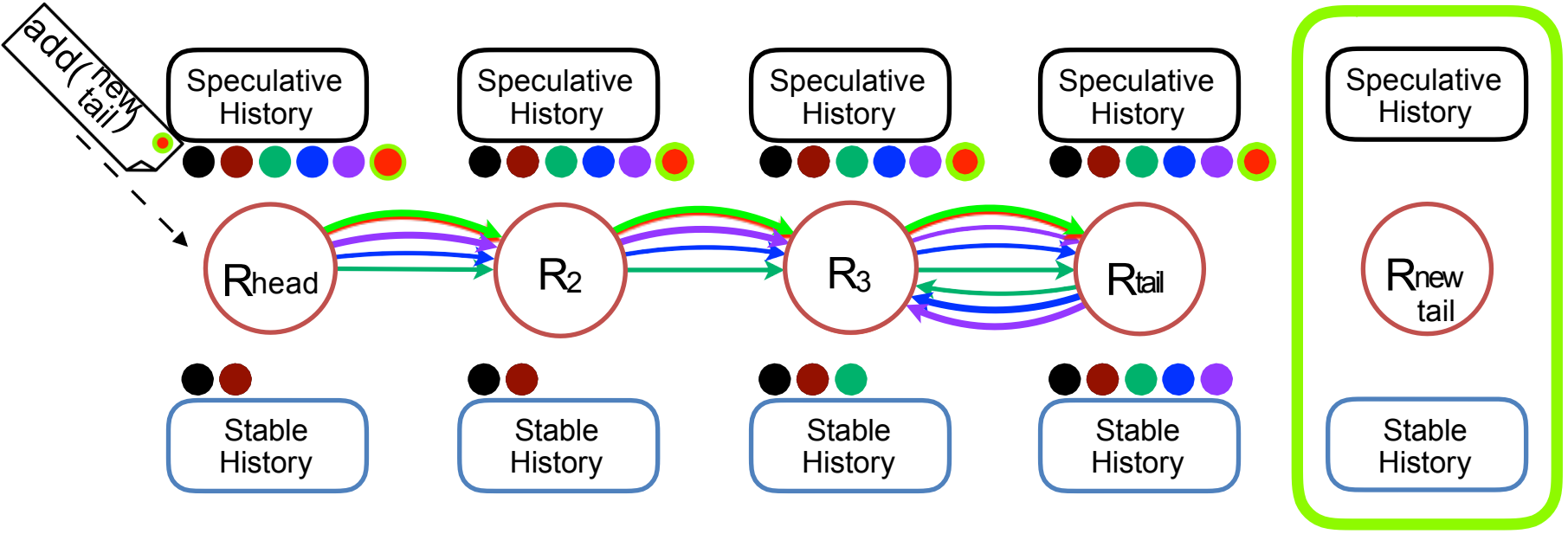
Relationship Among Histories



- **Speculative** and **stable histories** of new tail should become **equal** to the **speculative history of old tail**

Adding a New Node IV

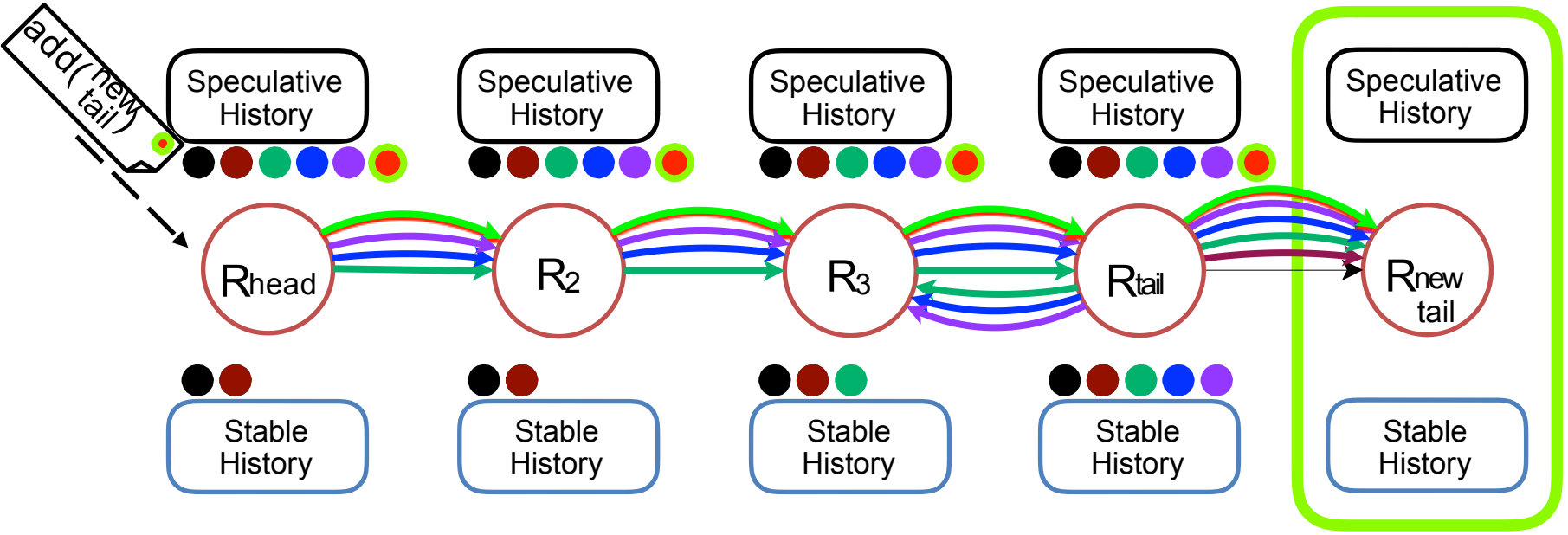
Relationship Among Histories



- **Old tail** should not answer to queries when the new tail does

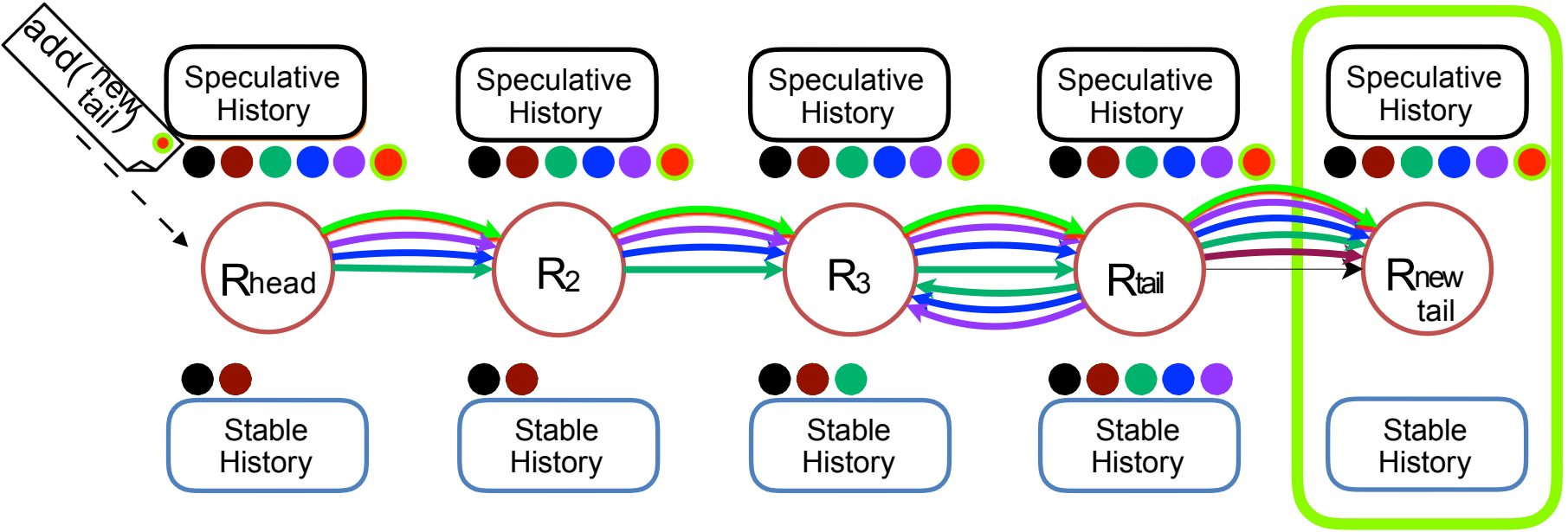
Adding a New Node

Flush History to New Tail



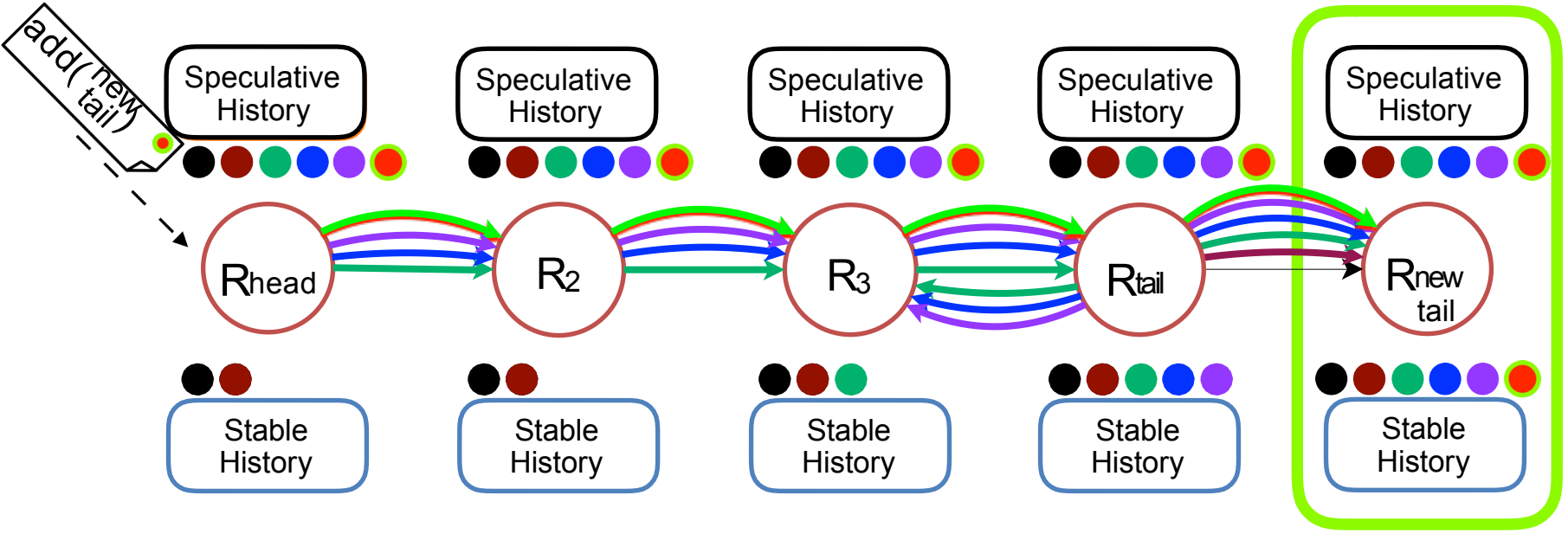
Adding a New Node

Flush History to New Tail



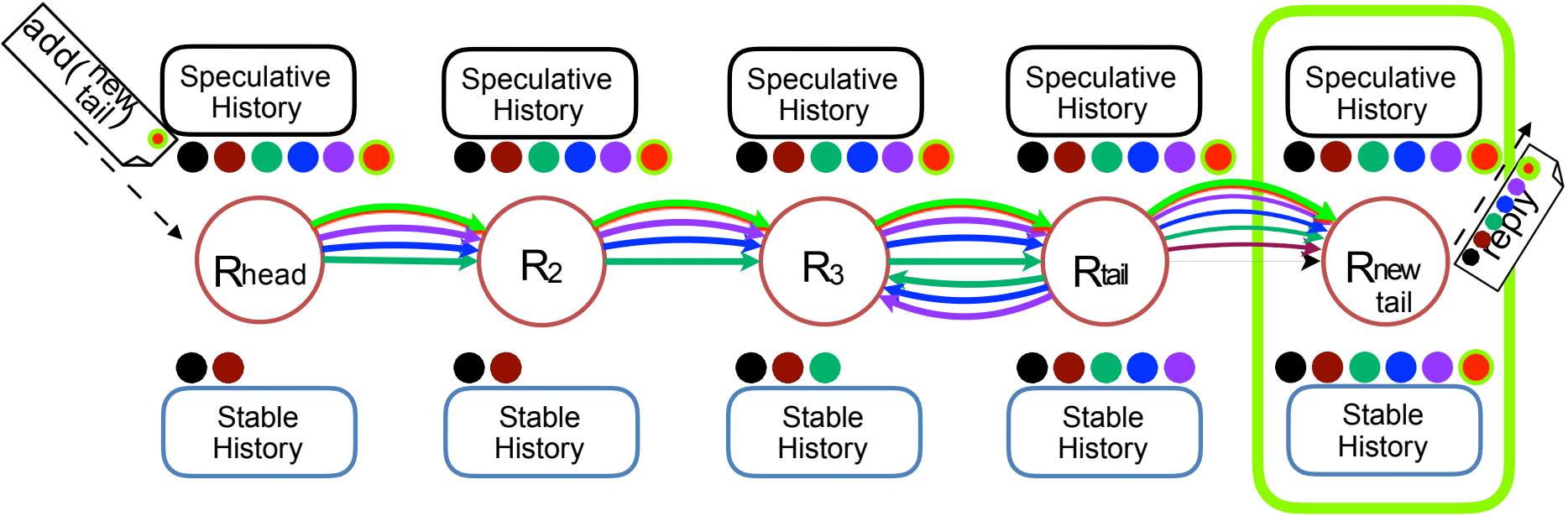
Adding a New Node

Copy Speculative onto Stable History



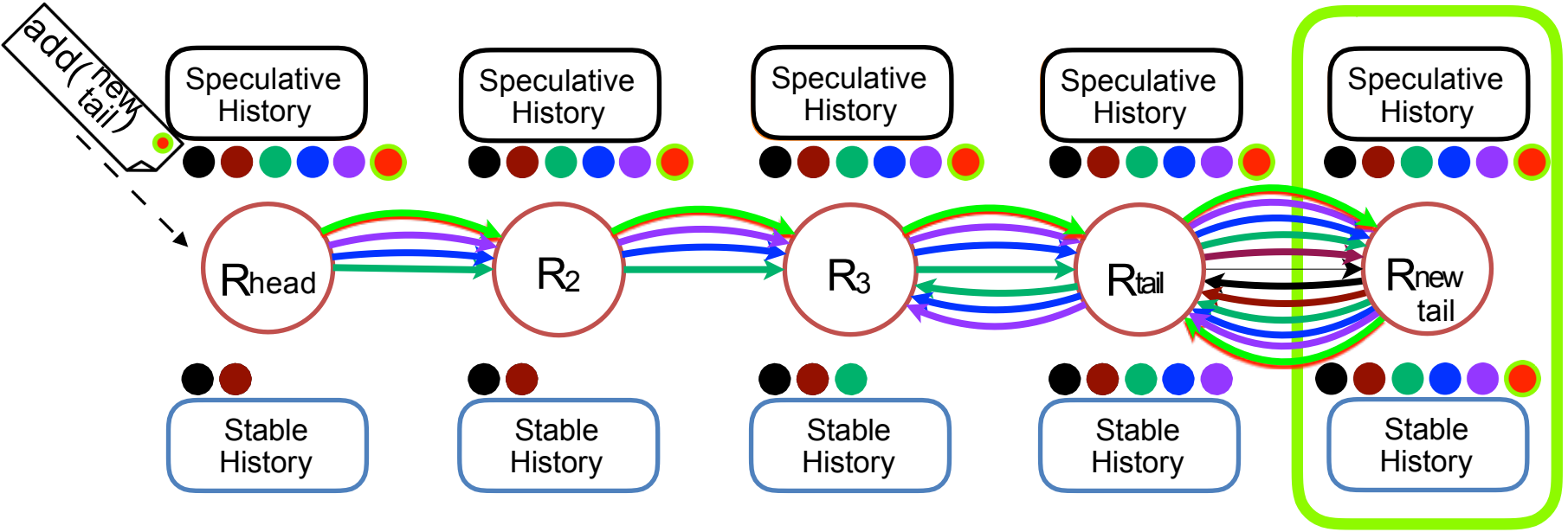
Adding a New Node

Respond to Queries and Acknowledge Updates



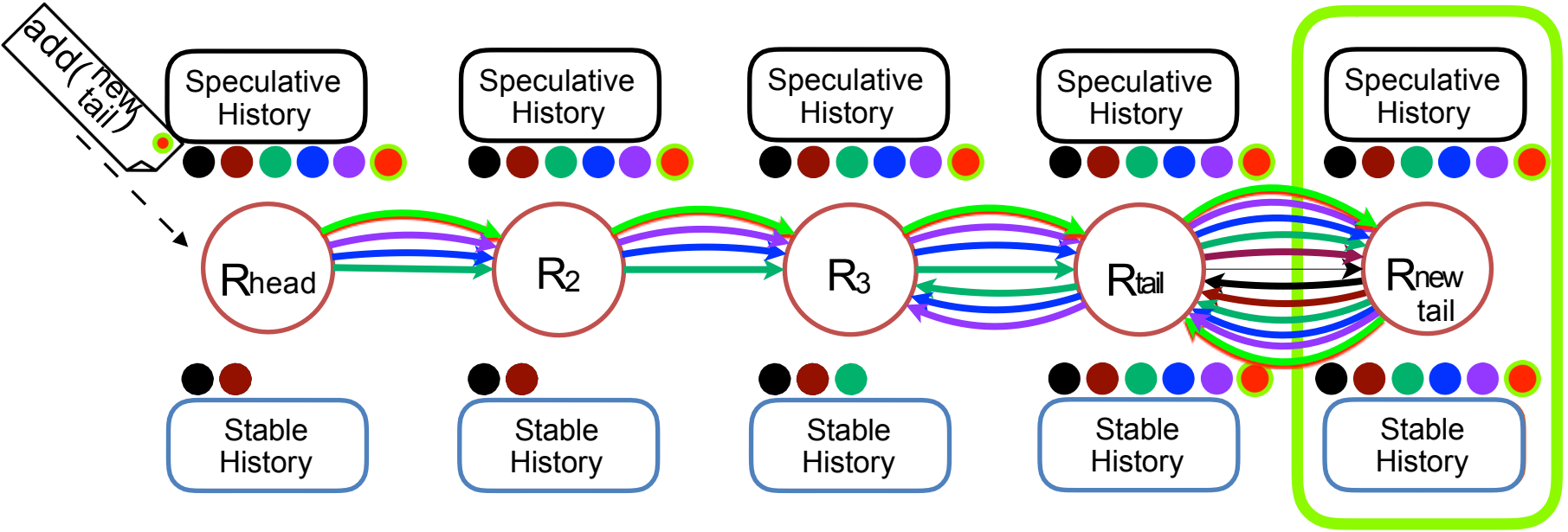
Adding a New Node I

Propagate Acknowledgements



Adding a New Node II

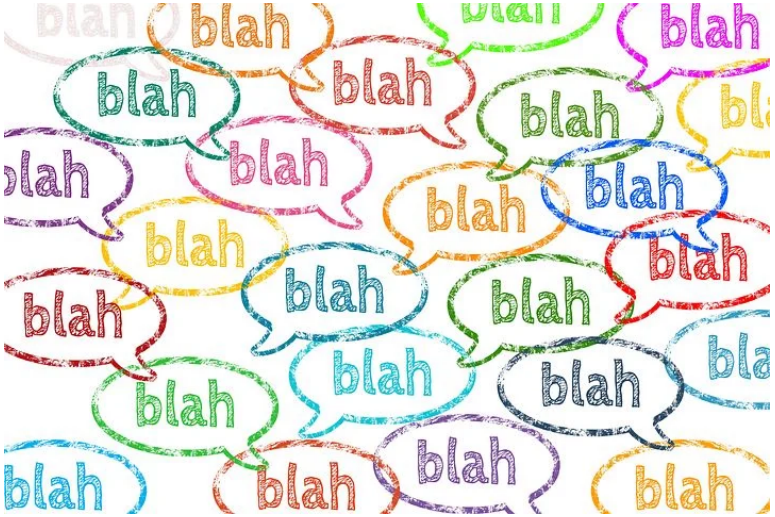
Propagate Acknowledgements



Self-study Questions

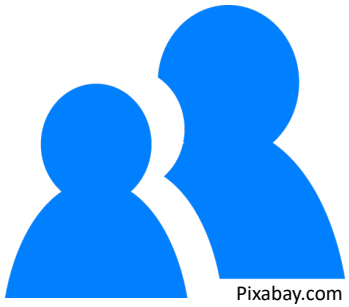
- Given a toplogy *n* replicas, *n>2*, how does replica *i* know about replica *i+2*, assuming replica *i+1* failed?
- Go through the motions of constructing a topology with `add(nodeId)` messages.
- When does a replica know for sure a node has been permanently added?

GOSSIPING



Pixabay.com

Gossiping protocols



- Disseminate information in **incremental manner**
  - Avoid overloading nodes with heavy broadcast messages
  - Drawback is longer propagation time for information
- Each node maintains a **partial view** of other nodes
- Each node chooses **random** nodes from its view to exchange information with
  - Application data (e.g., current state)
  - Its partial view (e.g., topology information)
- Nodes update their state and partial view based on the information received
- Gossiping happens **periodically** and **non-deterministically**
- Used in Cassandra for propagating status of each node, failure information and metadata

Lazy Replication Using Gossiping

- Replicas gossip about operations processed
- Replicas **reconcile** (compare) their operation logs and each apply any operations not yet seen
- Former step is highly application dependent
- Assumes updates can be applied in any order
- If system processes no more operations. then each replica **eventually** converges to the same state by gossiping enough times

Pixabay.com

Self-study Questions

- Draw out any topology of nodes and inject a message at a randomly chosen node, compare a broadcast (send to all neighbours versus a gossip (send to some neighbours):
  - How many messages are required?
  - How long does it take for all nodes to be up-to-date?

- Have each node in your topology maintain a data structure (e.g., counter, list, array, set, etc.), inject data structure updates at random nodes and propagate these updates via gossip:
  - What is the net result?
  - Does your replicated system converge at each replica to one and the same state (data structure)?