

Consistency & Transactions

Agenda

- Quick recap eventual consistency
- Replicated state-based object
- Conflict-free replicated data types

Eventual Consistency

- Eventual consistency states all replicas **eventually converge** when write operations stop
 - e.g., lazy replication using gossiping (cf. replication)
- Weak form of consistency with no time bound, but highly-available (i.e., always return a value, but value could be stale)

Eventual Consistency

- Works fine if a client always reads from the **same** replica...
- ...but gives “weird” results when the client reads from **multiple replicas** which may happen due to
 - Client mobility,
 - Replica failure
- Client-centric consistency models describe what must happen when a **single** client reads from **multiple replicas**

Eventual Consistency

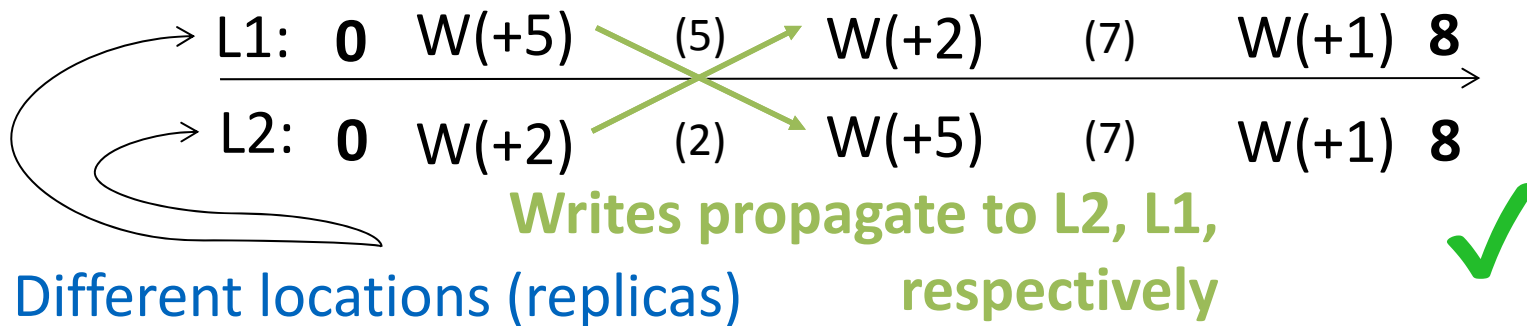
- Eventual consistency is desirable for **large-scale distributed systems** where **high availability** is important
- Tends to be cheap to implement (e.g., gossip)
- Constitutes a **challenge** for environments where **stronger consistency is important**

Handling Concurrent Writes

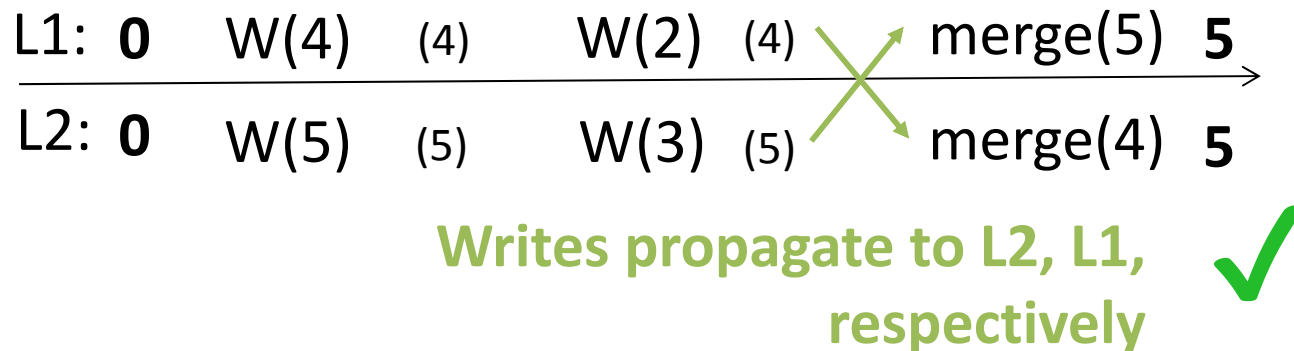
- Do need a mechanism to **handle concurrent writes**
- **If there were** a nice way to **handle concurrent writes**, we could support **eventual consistency** more broadly
- “Only” need to guarantee that after processing all writes for an item, all replicas converge, **no matter what order** the writes are processed

Example

Growth-only counter (G-counter)



Max register



State-based objects

Mostly plain objects

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

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

```
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
```

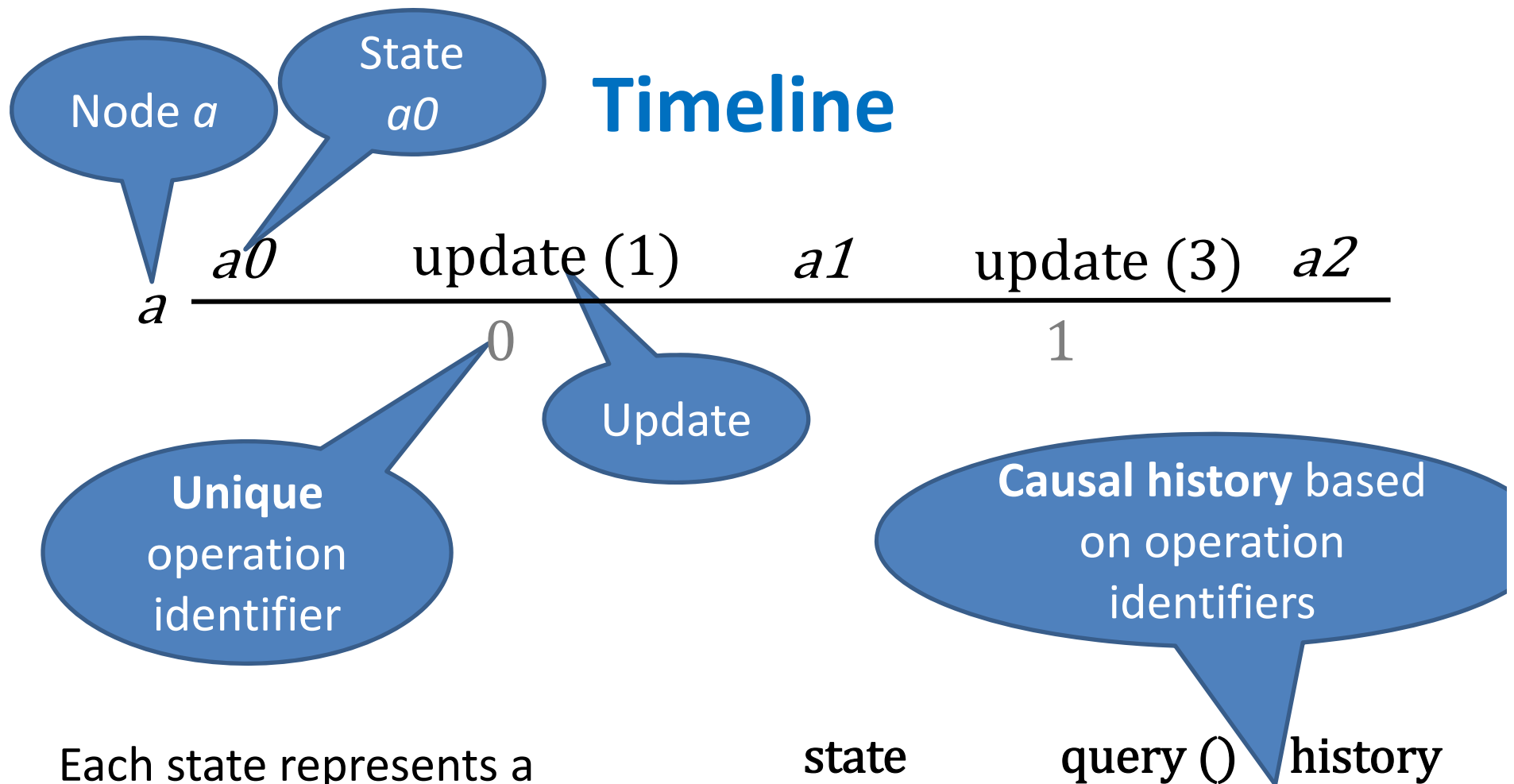
Average

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

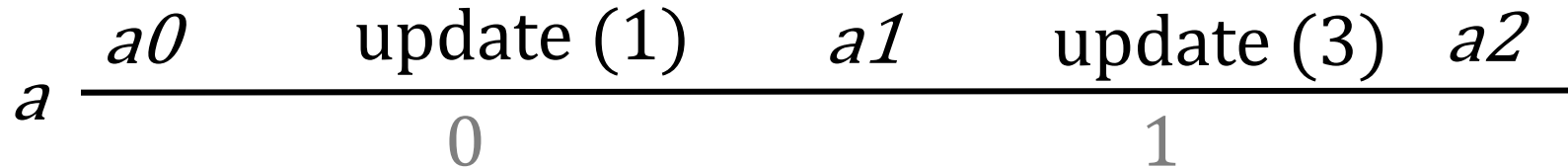
- State-based object replicated across multiple nodes
- E.g., replicate `Avg` across two nodes
- Both nodes have a copy of state-based object
- Clients send query and update to a single node
- Nodes periodically send their copy of state-based object to other nodes for merging



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

	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

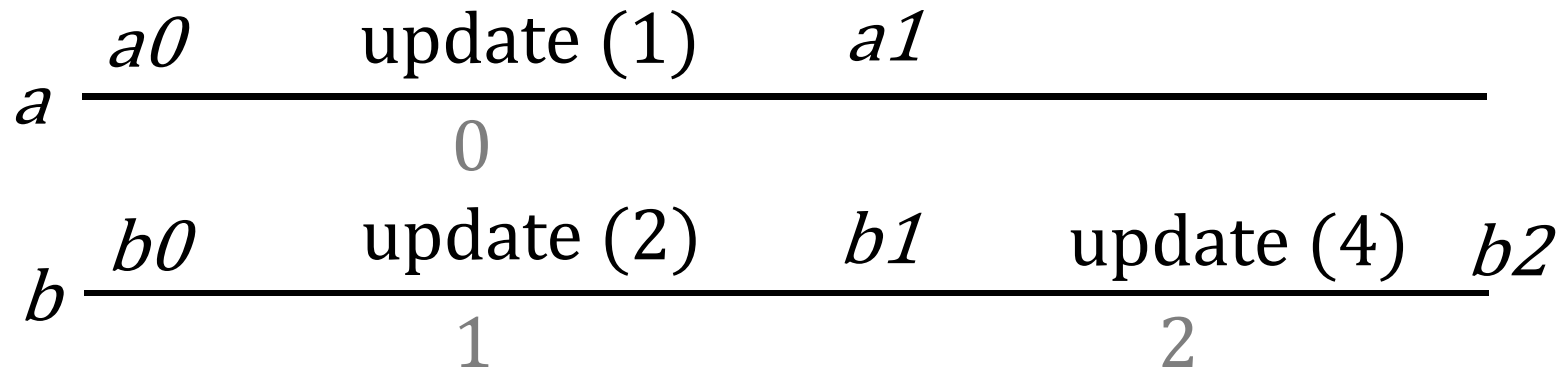


Operation identifier is unique across replicas

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

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

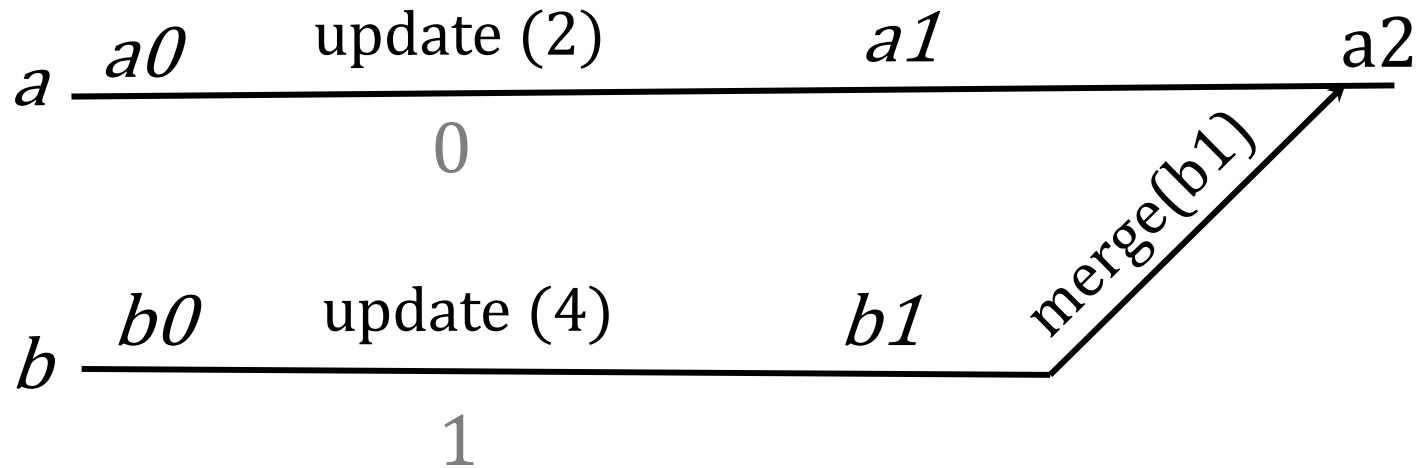
States and Causal Histories



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

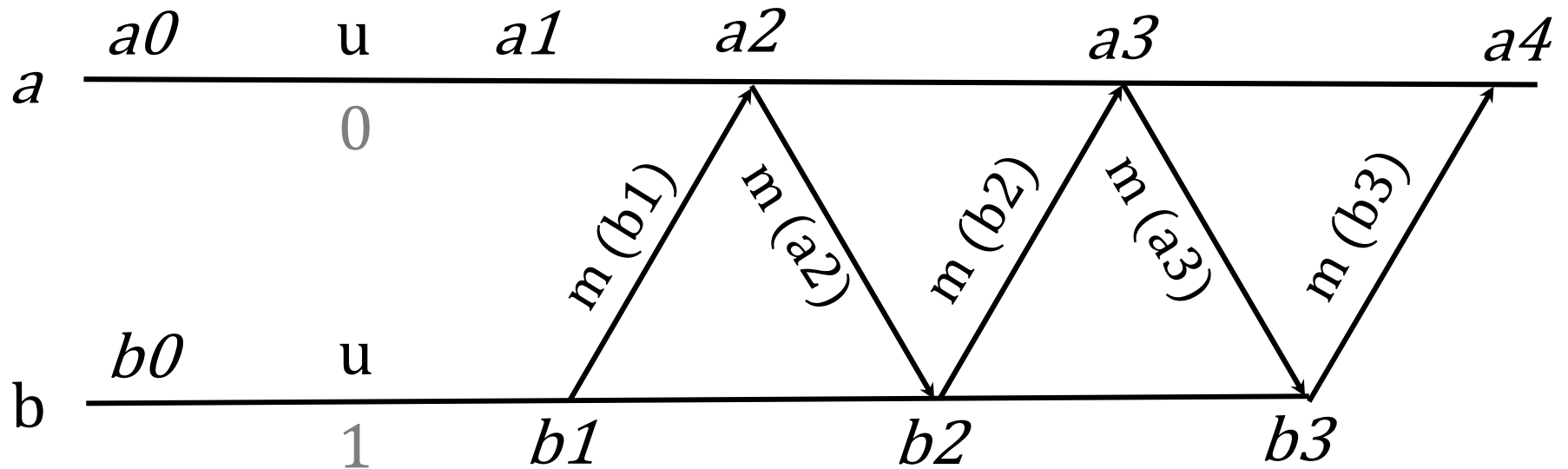
	state	query ()	history
a0	sum:0, cnt:0	0	{ }
a1	sum:1, cnt:1	1	{0}
b0	sum:0, cnt:0	0	{ }
b1	sum:2, cnt:1	2	{1}
b2	sum:6, cnt:2	3	{1,2}

Merge



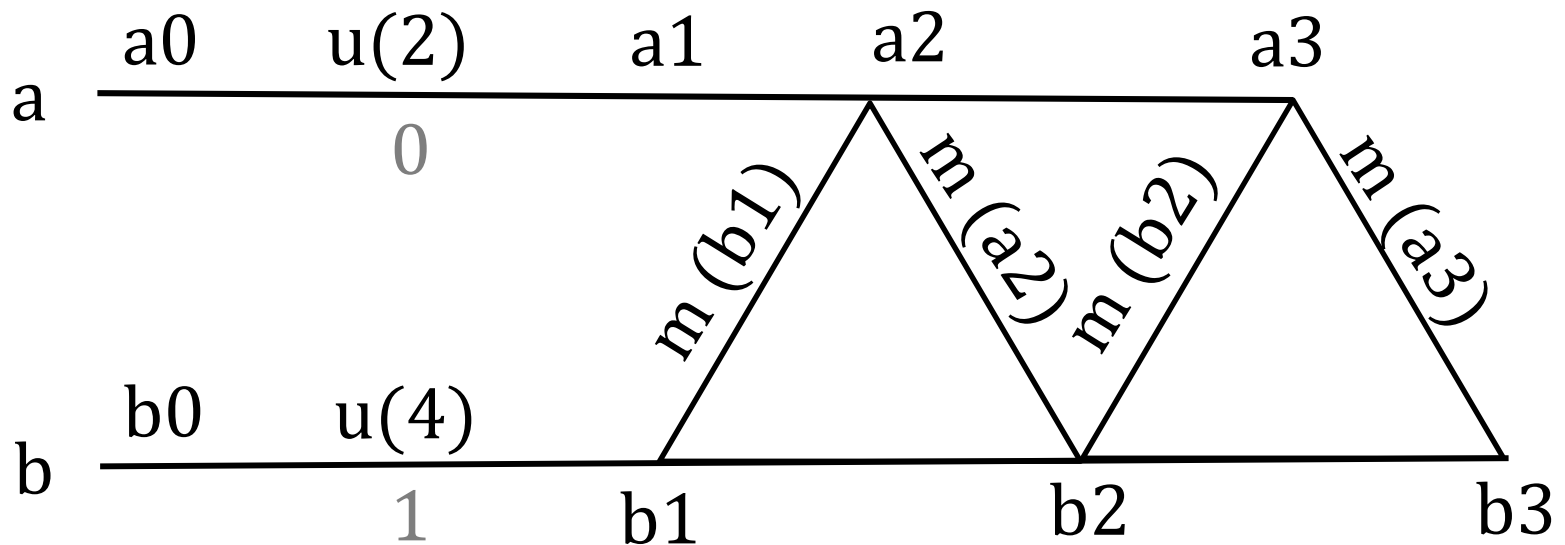
	state	query ()	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
b0	sum:0, cnt:0	0	{ }
b1	sum:4, cnt:1	4	{1}
a2	sum:6, cnt:2	3	{0,1}

Nodes Periodically Propagate Their State



Strong Eventual Consistency & 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
- 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.



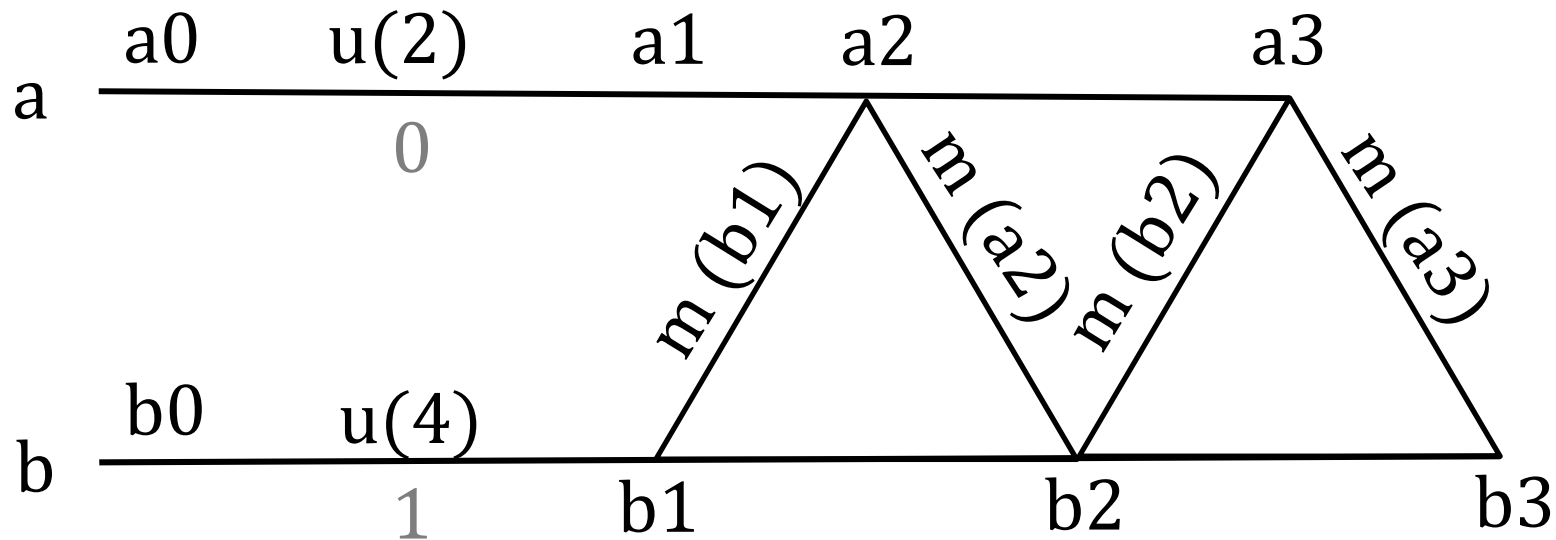
a, *b* attain the **same causal history** but **do not converge** to the **same internal state** – they **do not converge** at all!

Neither eventually consistent, nor strongly eventually consistent

	state	query ()	history
<i>a</i> 0	sum:0, cnt:0	0	{ }
<i>a</i> 1	sum:2, cnt:1	2	{0}
<i>a</i> 2	sum:6, cnt:2	3	{0,1}
<i>a</i> 3	sum:16, cnt:5	3.2	{0,1}
<i>b</i> 0	sum:0, cnt:0	0	{ }
<i>b</i> 1	sum:4, cnt:1	4	{1}
<i>b</i> 2	sum:10, cnt:3	3.3	{0,1}
<i>b</i> 3	sum:26, cnt:8	3.25	{0,1}

NoMergeAverage

- Object's merge does nothing
- All else is the same as for Avg



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

Neither eventually consistent, nor strongly eventually consistent

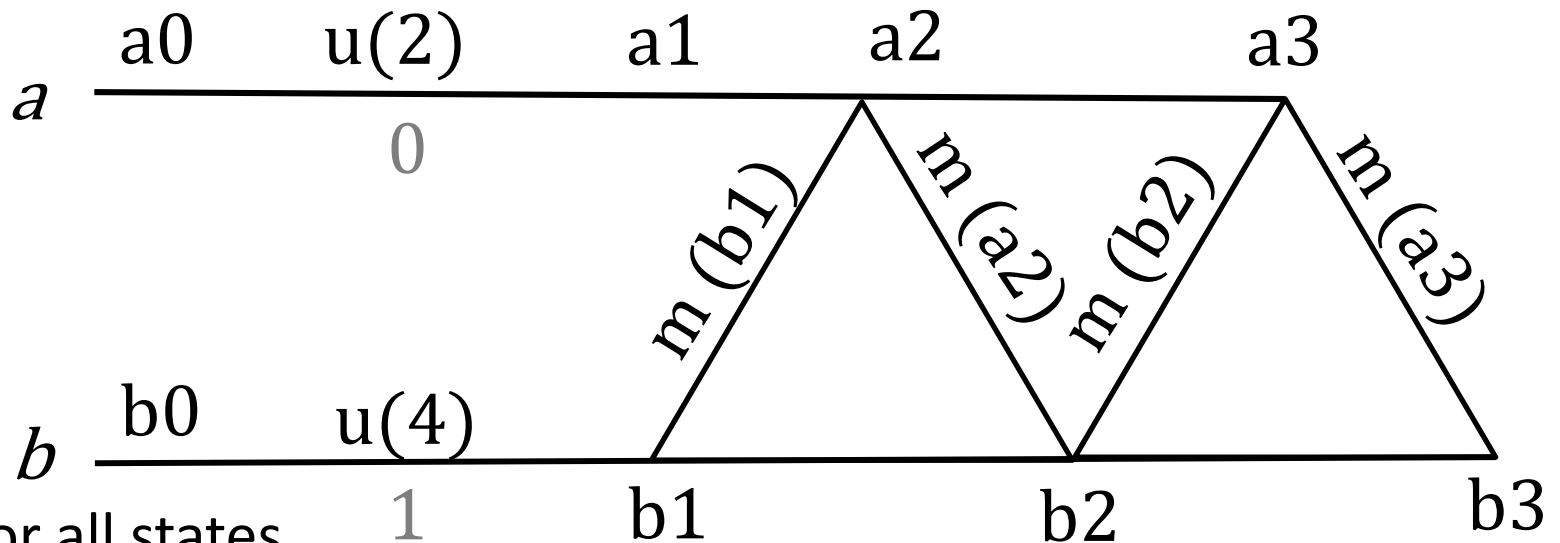
	state	query ()	history
<i>a</i> 0	sum:0, cnt:0	0	{ }
<i>a</i> 1	sum:2, cnt:1	2	{0}
<i>a</i> 2	sum:2, cnt:1	2	{0,1}
<i>a</i> 3	sum:2, cnt:1	2	{0,1}
<i>b</i> 0	sum:0, cnt:0	0	{ }
<i>b</i> 1	sum:4, cnt:1	4	{1}
<i>b</i> 2	sum:4, cnt:1	4	{0,1}
<i>b</i> 3	sum:4, cnt:1	4	{0,1}

BMergeAverage

- Object's merge
 - At b – overwrite state with state at a
 - At a – does nothing
- All else is the same as for Avg

MaxAverage

- Object's merge
 - Pair-wise max of sum and cnt
- All else is the same as for Avg



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

	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

Conflict-Free Replicated Data Types

- CRDT is a conflict-free replicated state-based object
- CRDT can handle concurrent writes
- **Solution:** 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 ,
 $\text{merge}(\text{merge}(x, y), z)$ is equal to $\text{merge}(x, \text{merge}(y, z))$
- Merge is **commutative** (e.g., $A + B = B + A$)
 - For any two state-based objects, x and y , $\text{merge}(x, y)$ is equal to $\text{merge}(y, x)$
- Merge is **idempotent**
 - For any state-based object x , $\text{merge}(x, x)$ is equal to x
- Every **update is increasing**
 - Let x be an arbitrary state-based object and let $y = \text{update}(x, \dots)$ be the result of applying an arbitrary update to x
 - Then, update is increasing if $\text{merge}(x, y)$ 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))
```

- Commutativity

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

- Impotence

```
merge(a, a)
= max(a.x, a.x)
= a.x
= 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)` – first of two updates – invokes `a.add(x)`.
- `sub(x)` – second of two updates – invokes `r.add(x)`
- `merge` performs a pairwise merge of a and r

CRDTs

- Formalized and introduced in 2014
- CmCRDTs and CvCRDTs are equivalent
- Nice solution if they apply
- Challenge is to design new CRDTs