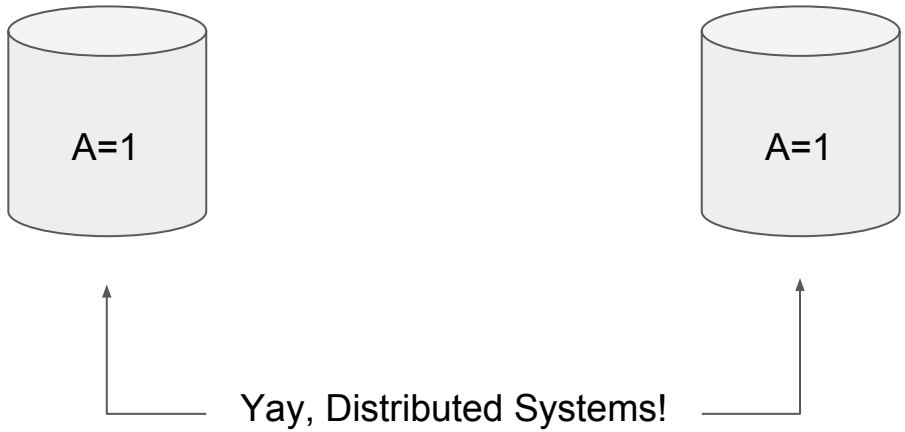
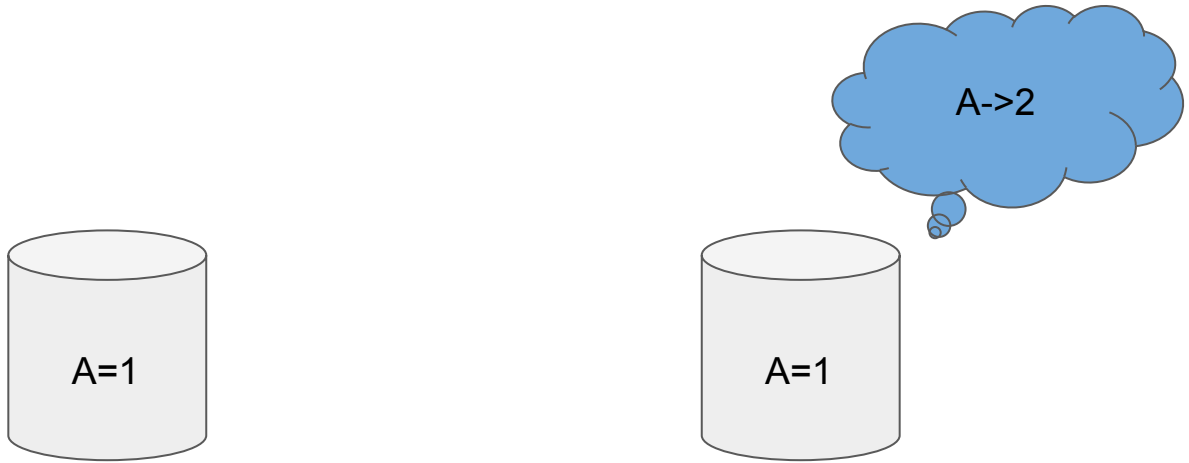
A comprehensive study of Convergent and Commutative Replicated Data Types

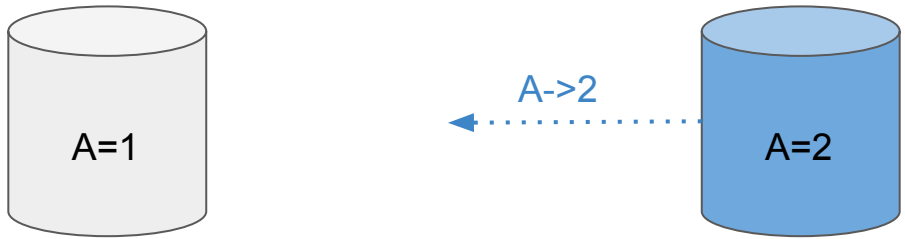
Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski

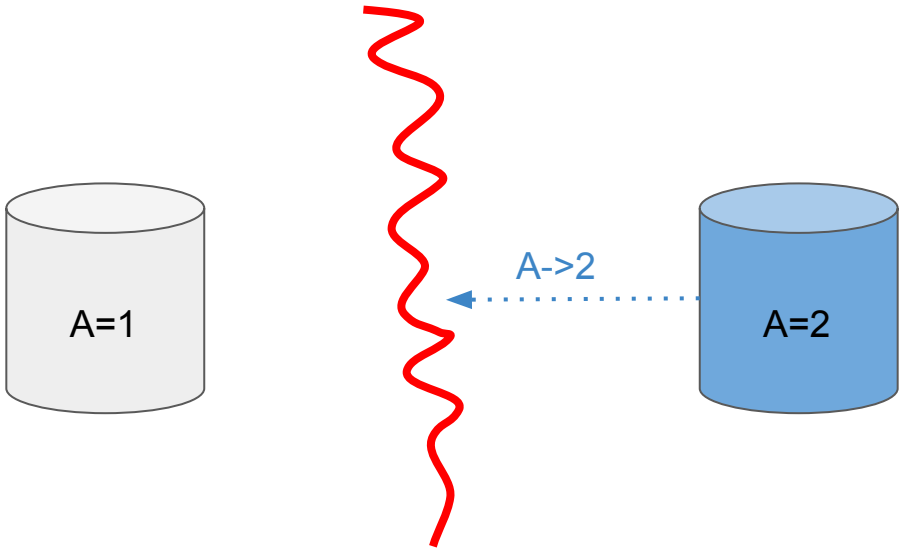
http://hal.upmc.fr/inria-00555588

Eventual Consistency

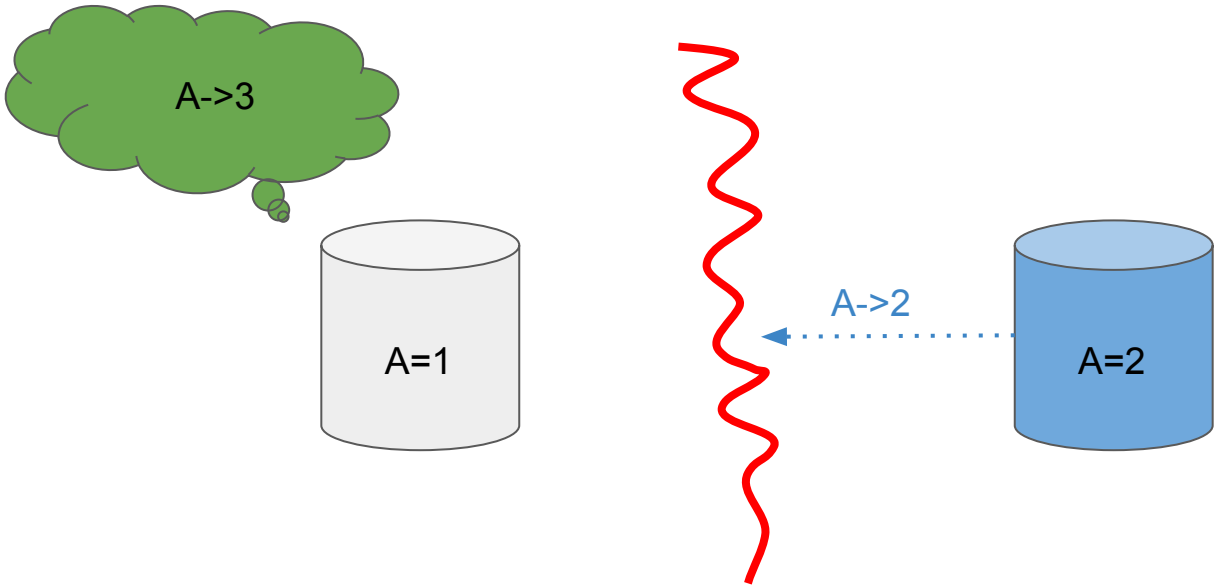

A=1
A=1
Yay, Distributed Systems!

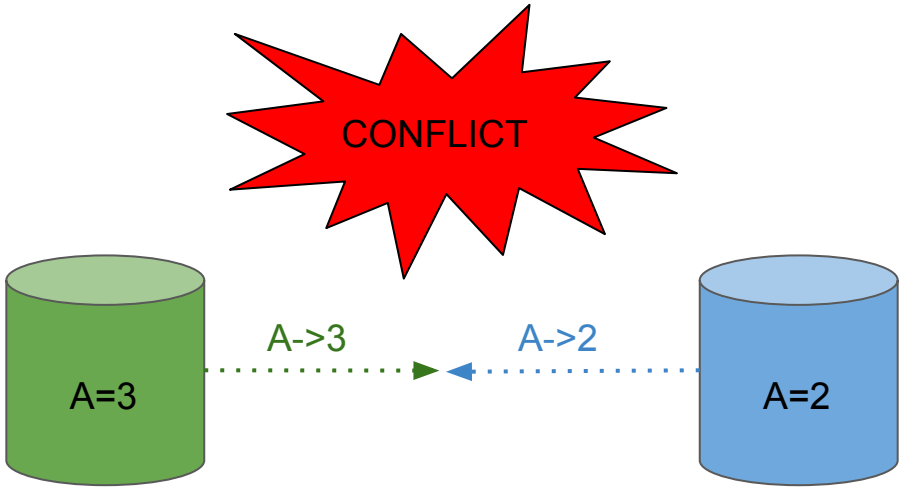

A->2
A=1
A=1


A=1
A->2
A=2



Shit Happens


A->3
A=1
A->2
A=2


CONFLICT
A=3
A->3
A->2
A=2

Shopping Carts circa 2007

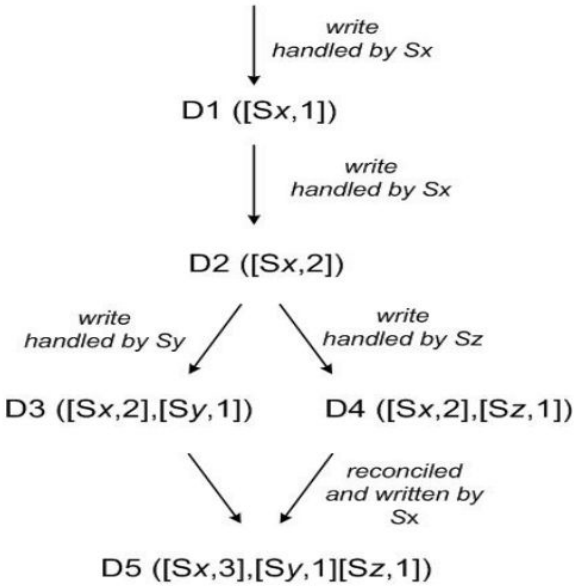
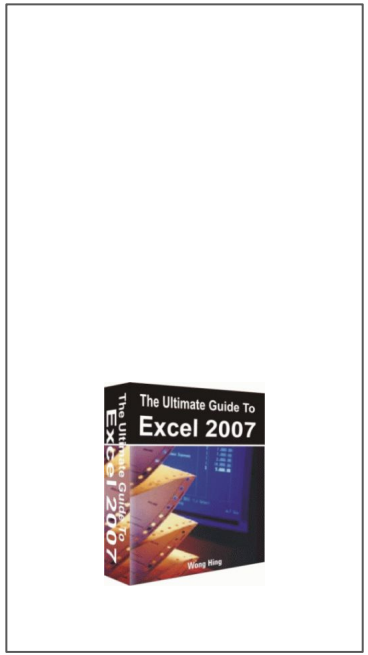


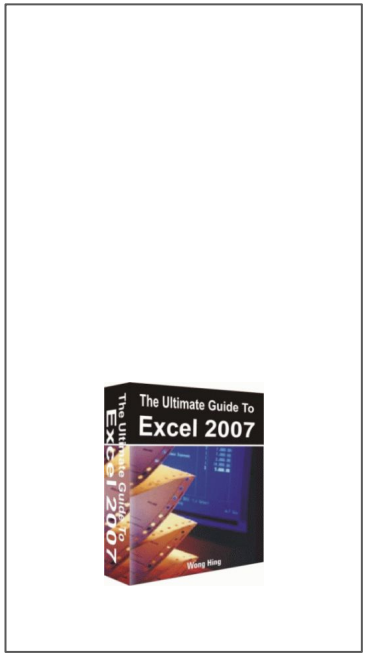
Figure 3: Version evolution of an object over time.

However, version branching may happen, in the presence of failures combined with concurrent updates, resulting in conflicting versions of an object. **In these cases, the system cannot reconcile the multiple versions of the same object and the client must perform the reconciliation in order to collapse multiple branches of data evolution back into one (semantic reconciliation).** A typical example of a collapse operation is “merging” different versions of a customer’s shopping cart. **Using this reconciliation mechanism, an “add to cart” operation is never lost. However, deleted items can resurface.**

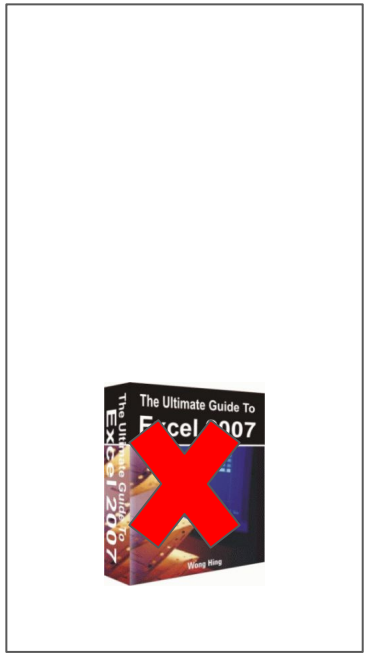
http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf


The Ultimate Guide To
Excel 2007
Wong Hing

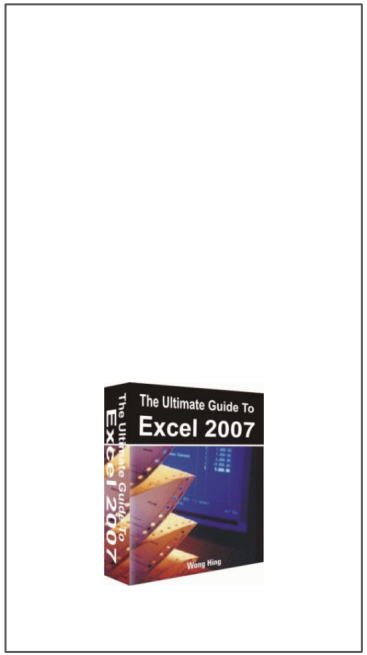
[a=1, b=1]


The Ultimate Guide To
Excel 2007
Wong Hing

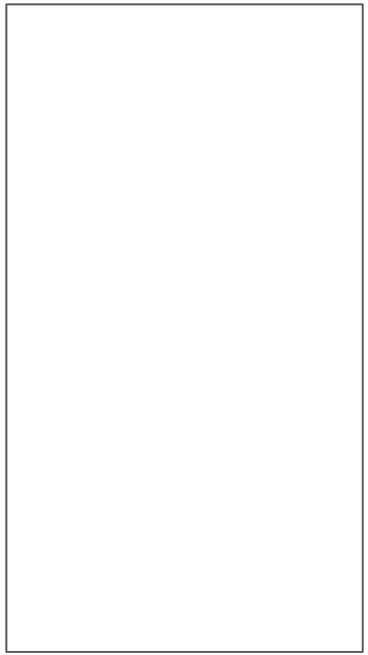
[a=1, b=1]

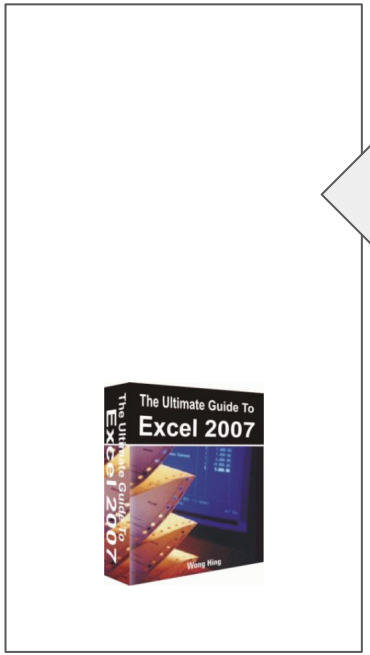

The Ultimate Guide To
Excel 2007
Wong Hing

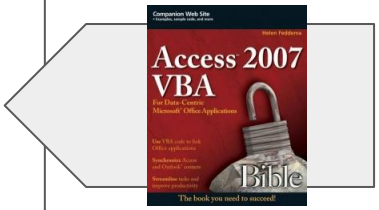
[a=2, b=1]


The Ultimate Guide To
Excel 2007
Wong Hing

[a=1, b=1]






Companion Web Site
Helen Feddema
Access 2007
VBA
For Data-Centric
Microsoft® Office Applications
Bible
The book you need to succeed!

The Ultimate Guide To
Excel 2007
Wong Hing

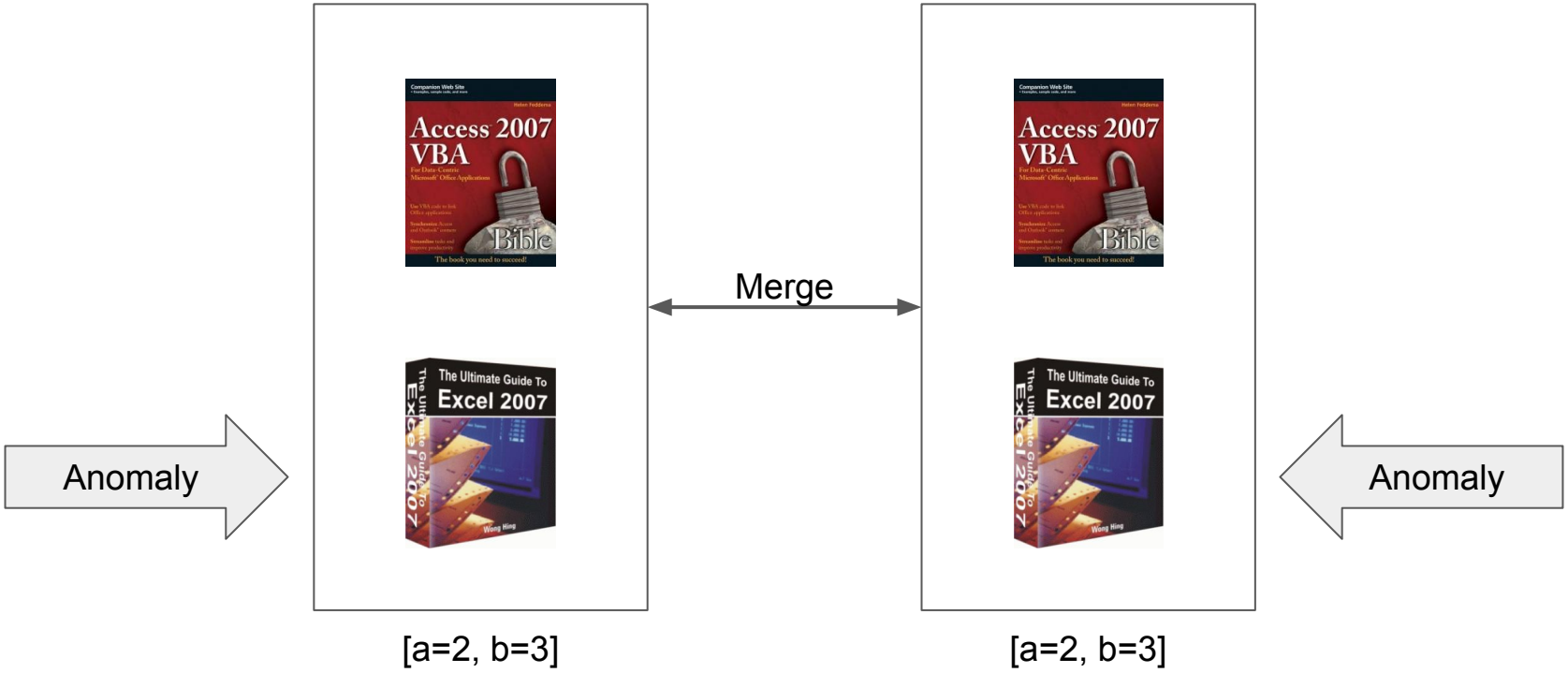
[a=2, b=1]

[a=1, b=2]


CONFLICT
Access 2007 VBA
Bible
The Ultimate Guide To Excel 2007
Wong Hing

[a=2, b=1]

[a=1, b=2]


Access 2007 VBA
Bible
The Ultimate Guide To Excel 2007
Wong Hing
Merge
Anomaly
Anomaly
[a=2, b=3]
[a=2, b=3]

{C} Replicated Data Types

{C} Replicated Data Types

- Specification of Convergent Abstract Data Types (baquero/moura; 1997)
- Designing a commutative replicated data type (shapiro/preguica; 2007)
- Convergent and Commutative Replicated Data Types (s/p/b/zawirski; 2011)
- Conflict-free Replicated Data Types (s/p/b/z; 2011)

State Based Replication

- Every replica has a local state
- Define query operations as functions of the local state
- Define update operations (preconditions, side effects on local payload)
- Continually broadcast your local state to your peers

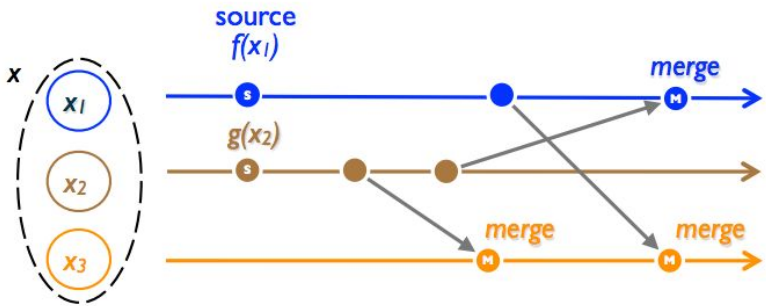


Figure 4: State-based replication

State Based Replication

- Define merge(v1, v2) -> vM to merge peer's state w/ local state
- If merges **converge** replicas will end up consistent
- It's a **Convergent** Replicated Data Type (CvRDT)

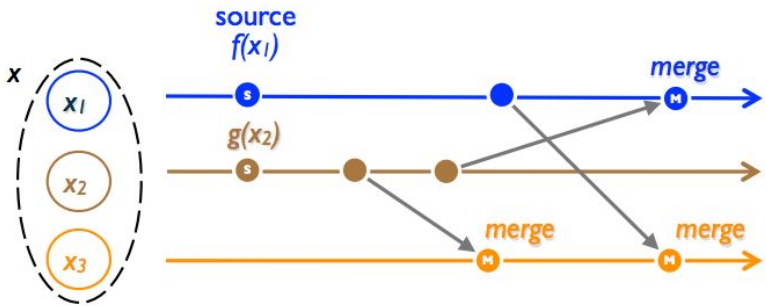


Figure 4: State-based replication

State Based Replication - MATH TIME!

- Define a partial order (<=) on possible states
- Updates may only move to “bigger” states
- Make sure it has a least upper bound (LUB) function
- Define merge(v1, v2) == LUB(v1, v2)
- Congrats, replicas will eventually be consistent!

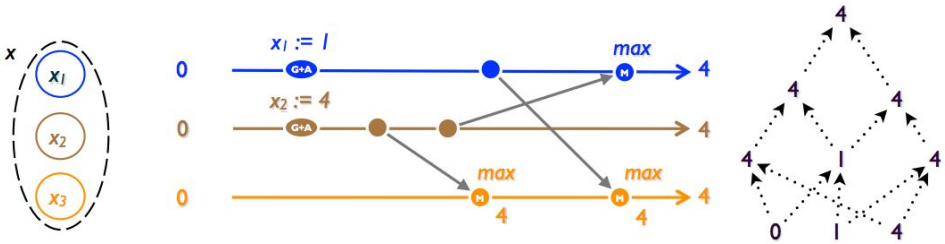


Figure 5: Example CvRDT: integer + max

Operation Based Replication

- Every replica has a local state, queries on local state (as above)
- Define update operations in two parts:
  - Local-only enforcement of preconditions (atSource)
  - Local + remote application of update to state (downstream)
- Replicate log of operations to your peers so they can apply them too

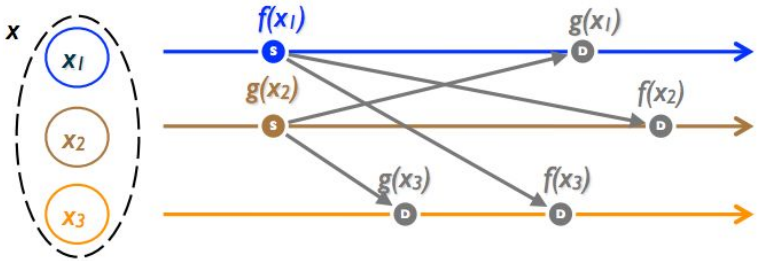


Figure 6: Operation-Based Replication

Operation Based Replication

- Peers update their state by doing the same thing you did
- If operations **commute** replicas will end up consistent
- It's a **Commutative** Replicated Data Type (CmRDT)

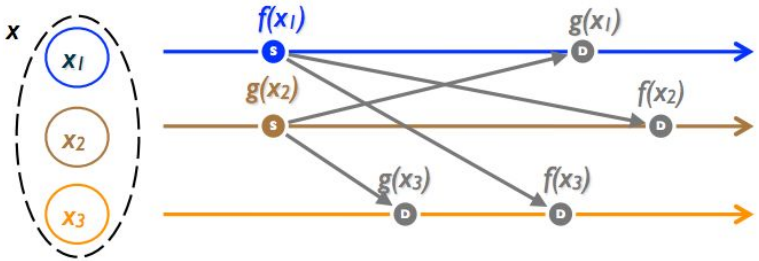


Figure 6: Operation-Based Replication

Operation Based Replication - MATH TIME!

- Define a partial order on operations (generally causal order, possibly weaker)
- Make sure operations are delivered to peers exactly once, in order*
- Prove that operations **not** related by the partial order commute
  - ie f(g(x)) == g(f(x))
- Congrats, replicas will eventually be consistent!

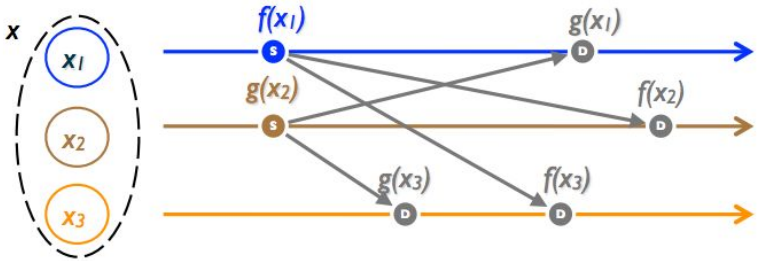


Figure 6: Operation-Based Replication

* some assembly required

TL;DR: They’re Isomorphic

Specification 3 Operation-based emulation of state-based object

```
1: payload State-based S                                  ▷ S: Emulated state-based object
2:     initial  Initial payload
3: update State-based-update (operation f, args a) : state s
4:     atSource (f, a) : s
5:         pre S.f.precondition(a)
6:         let s = S.f(a)                                 ▷ Compute state applying f to S
7:     downstream (s)
8:         S := merge(S, s)
```

Specification 4 State-based emulation of operation-based object

```
1:  payload Operation-based P, set M, set D    ▷ Payload of emulated object, messages, delivered
2:      initial Initial state of payload, ∅, ∅
3:  update op-based-update (update f, args a) : returns
4:      pre P.f.atSource.pre(a)                                  ▷ Check at-source precondition
5:      let returns = P.f.atSource(a)                            ▷ Perform at-source computation
6:      let u = unique()
7:      M := M ∪ {(f, a, u)}                                     ▷ Send unique operation
8:      deliver()                                                ▷ Deliver to local op-based object
9:  update deliver ()
10:     for (f, a, u) ∈ (M \ D) : f.downstream.pre(a) do
11:         P := P.f.downstream(a)                               ▷ Apply downstream update to replica
12:         D := D ∪ {(f, a, u)}                                 ▷ Remember delivery
13: compare (R, R′) : boolean b
14:     let b = R.M ≤ R′.M ∨ R.D ≤ R′.D
15: merge (R, R′) : payload R″
16:     let R″.M = R.M ∪ R′.M
17:     R″.deliver()                                             ▷ Deliver pending enabled updates
```

Some “Basic” CRDTs

Counters

- A replicated integer
- Two operations, increment() and decrement()
- Should converge to the global # of increments minus # of decrements
- About as simple as you can get but still be interesting

Operation-Based Counter

- Stupid easy: all the complexity is in your reliable broadcast channel
- No preconditions: delivery order is empty
- inc(dec(x)) == dec(inc(x)) for all x
- … that’s it

State-Based **increment-only** Counter (G-Counter)

- More work - basically implementing reliable delivery channel
- State is a vector clock {a=1, b=2, c=3, …}
- Query returns the sum of all entries
- x1 <= x2 <-> x1[i] <= x2[i] for all i
- Increment increases the local node’s entry (strictly “larger”)
- Merge takes the pairwise max for each entry (LUB)
- Useful for counting events (events can’t un-happen)

State-Based PN-Counter

- Two G-Counters, one for increments one for decrements
- Query returns the difference between the two counters
- Partial order by &&'ing the two G-Counter partial orders
- Merge by merging the two G-Counters
- Yay, composition!
- Useful for things like the number of current users
- Non-negative counters are hard (global invariant vs local invariant)

Registers

- A box with a value in it (type of the value doesn’t matter)
- Query returns the value
- Assign sets it
- Preserve causal order of assigns
- Do **something** with concurrent assigns

“Last” Writer Wins (LWW) Register

- Payload is (value, timestamp)
- Biggest timestamp wins during merge/downstream apply
- Hopefully clock drift isn’t a thing?
- Veeeeeery common approach in the wild

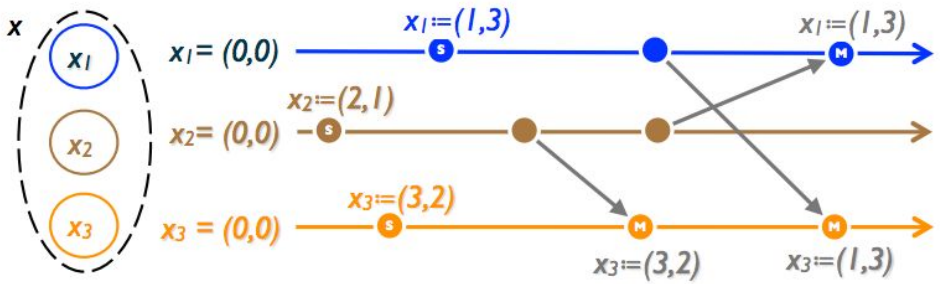


Figure 7: Integer LWW Register (state-based). Payload is a pair (value, timestamp)

Multi-Value (MV) Register

- Payload is a set of (value, vector-clock) pairs
- Non-concurrent assigns (via vector clock) overwrite
- Concurrent assigns get unioned together
- AKA Dynamo

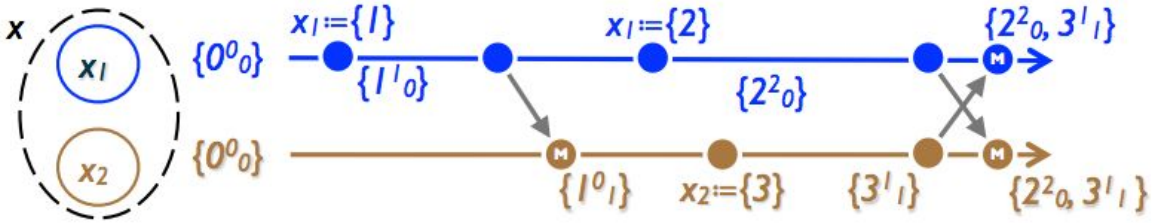


Figure 9: MV-Register (state-based)

Sets

- Unordered set of values (type of value doesn’t matter)
- add(x): S := S U {x}
- remove(x): S := S ^ ~{x}
- add(x) and remove(x) don’t commute, so not *strictly* possible
- Need to decide what “wins” in case of concurrent add/remove

Grow-Only (G) Set

- Remove is hard, so just don’t do it
- Concurrent adds commute so op-based is easy
- For state-based, partial order is set-contains, LUB is union
- Useful mainly as a building block for other sets

Two-Phase (2P) Set

- Two G-Sets - one for add, one for remove
- Value is { added ^ ~(removed) }
- Remove ALWAYS wins; once you remove you can NEVER re-add
- Reject removes of things that haven’t been added
- Grows unbounded :(

Unique (U) Set

- Like a 2P-Set, but assume ahead of time that every element is unique
- If op-based, partial order ensures remove(x) is seen after add(x)
  - Don’t need to explicitly track removed set, yay!
- If state-based, garbage collect (same as ^’s reliable broadcast impl)
- Mainly interesting as the basis for OR-Set

LWW-element Set

- 2P-Set of (element, timestamp) pairs
- Element is in the set if it’s in the added set and NOT in the removed set with a bigger timestamp
- Again, hopefully clock drift isn’t a thing...

PN Set

- Set of (element, counter) pairs
- Add increments the counter, remove decrements
- Value is all elements with counter > 0
- “Anomalies” if counter goes negative or extra-positive
- ???

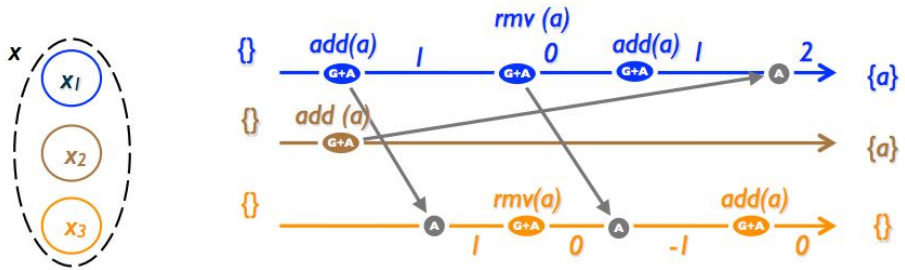


Figure 13: PN-Set (op-based)

Observed-Remove (OR) Set

- Builds on U-Set
- Set of (element, GUID) pairs to force uniqueness
- Value is “collapsed” set of unique elements (ignore GUID(s))
- Add generates a new GUID and adds (element, GUID)
- Remove removes all locally-known (element, GUID) pairs

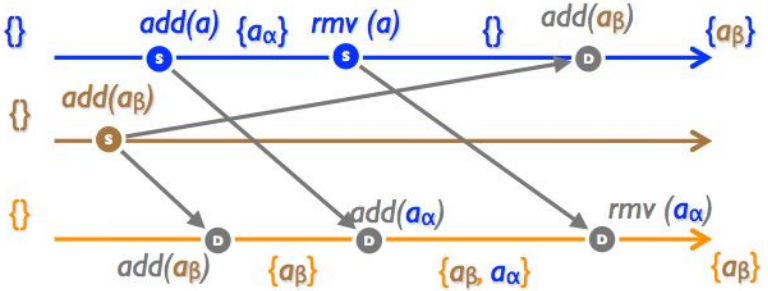


Figure 14: Observed-Remove Set (op-based)

Graphs

- Pair of sets (V, E) - vertexes and edges
- E from V x V - vertexes must exist
- What happens on concurrent addEdge(u, v) and removeVertex(u)?
  - removeVertex wins, all edges implicitly deleted
  - addEdge wins, vertex is implicitly recreated
  - Synchronization :(
- Authors think option (1) is less weird
- 2P2P Graph: Pair of 2P Sets, one for vertexes, one for edges
- removeVertex wins over concurrent addEdge, edge doesn’t count

Graphs

- In general, impossible to enforce global constraints like DAG
- Can do stronger constraints like *monotonic* DAG append-only
- Strengthening to partial order you can safely do add/remove

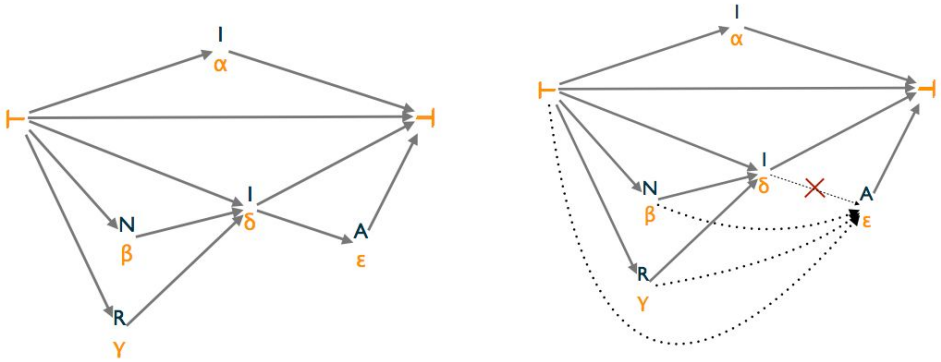


Figure 16: Monotonic DAG. Left: Greek letters indicate vertex identifiers; roman letters are characters in a text-editing application. Right: Remove OK only if paths are maintained. Dashed: removed; dotted: added.

Partial Orders

- Fully-transitive monotonic DAG (x<y && y<z -> x<z)
- Represented as a 2P-Set of vertices and a G-Set of edges
- addBetween(u, v, w) adds vertex ‘v’ between existing ‘u’ and ‘w’
  - Atomically adding all appropriate edges
- remove(v) “removes” v and all associated edges
  - Need to keep tombstones to deal with concurrent addBetweens

Collaborative Editing

- The major driving use-case (esp for op-based CRDTs)
- Document is a totally ordered set of “elements” (text, images, etc)
- Generally: partial order + some strategy for ordering “concurrent” adds

Replicated Growable Array (RGA)

- Conceptually a linked list of elements
- Elements are (value, timestamp) pairs
- addRight(u, a) adds a new element ‘a’ immediately to the right of ‘u’
- Implemented as partial order w/ concurrent adds ordered by decreasing time
- (timestamp here is local-clock.client-id to ensure uniqueness)

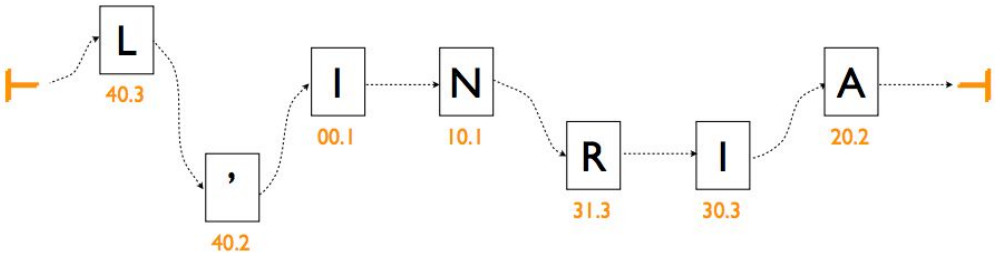


Figure 18: Replicated Growable Array (RGA)

Continuous Sequence

- U-Set of (value, position) pairs where position is a unique real number
- Add between two existing elements by picking an appropriate real number
- Do some magic to make sure positions are unique
- Expensive to actually do it with random real numbers
- Represent the continuum as a sparse tree (ala HAMTs!)

Garbage Collection

Garbage Collection

- You often need to leave tombstones to handle delete anomalies
  - Differentiating “This new thing got created” and “I don’t know this thing is deleted yet”
- Over time that can be a lot of tombstones…
- Important to garbage collect
- Stability problems
  - For example: cleaning up (x, ~x) pairs from a U-Set
  - Once everyone has (x, ~x) you can safely drop both w/out anomalies
  - Requires knowing who “everyone” is
- Commitment problems
  - For example: rebalancing tree for Continuous Sequence
  - Requires stronger coordination (but that may be okay since it’s offline)

Discuss!

TOO FAR! GO BACK!