Conflict resolution in eventual consistency

COS 418: *Distributed Systems*
Lecture 9

Michael Freedman

[Selected content adapted from M. Shapiro and I. Stoica]

Eventual consistency

- **Eventual consistency:** If no new updates to the object, *eventually* all accesses will return the last updated value
- Common: git, iPhone sync, Dropbox, Amazon Dynamo
- Why do people like eventual consistency?
  - Fast read/write of local copy of data
  - Disconnected operation

Concurrent writes can conflict

- Encountered in many different settings:
  - Peer-to-peer (Bayou)
  - Multi-master clusters (Dynamo)
- Potential solutions
  - "Last writer wins"
    - Thomas Write Rule for DBs with timestamp-based concurrency control: Ignore outdated writes
  - Application-specific merge/update: Bayou, Dynamo

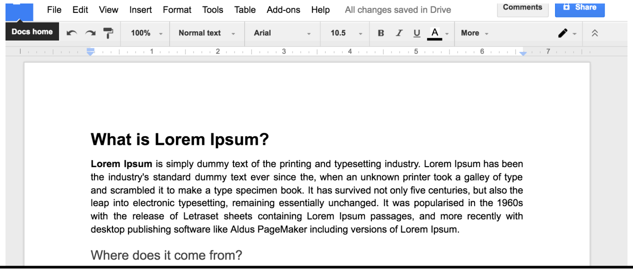
Towards generality?

General approach:
Encode ops as incremental update

- Consider banking (double-entry bookkeeping):
  - Initial: Alice = $50, Bob = $20
  - Alice pays Bob $10
    - Option 1: set Alice to $40, set Bob to $30
    - Option 2: decrement Alice -$10, incremental Bob +$10
      - #2 better, but can't always ensure Alice >= $0
- Works because common mathematical ops are
  - Commutative: $A \circ B == B \circ A$
  - Invertible: $A \circ A^{-1} == 1$

Consider shared word processing

- How do I insert a new word?
  - Send entire doc to server? Not efficient
  - Send update operation!



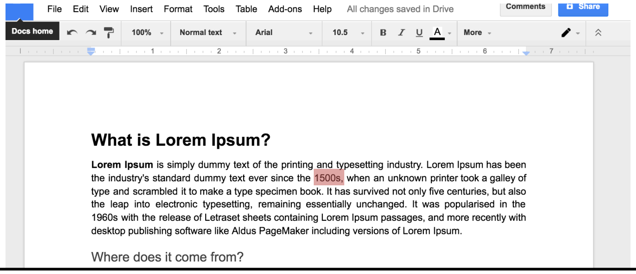
What is Lorem Ipsum?

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Where does it come from?

Consider shared word processing

- How do I insert a new word?
  - Send entire doc to server? Not efficient
  - Send update operation! insert (string, position) = insert("1500s", 166)
  - Warning: Insert (rather than replace) shifted position of all following text

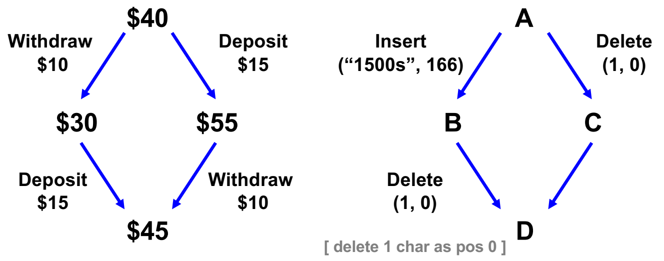


What is Lorem Ipsum?

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Where does it come from?

Operations must be commutative



$40

Withdraw $10

Deposit $15

$30

$55

Deposit $15

Withdraw $10

$45

A

Insert ("1500s", 166)

Delete (1, 0)

B

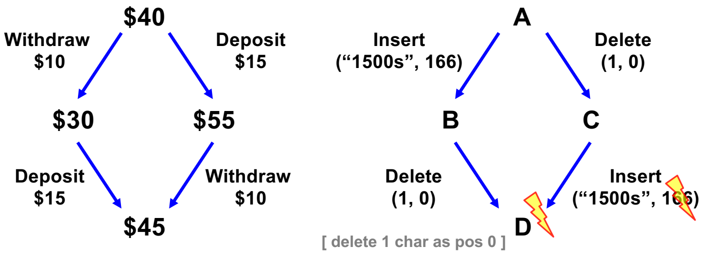
C

Delete (1, 0)

D

[delete 1 char as pos 0]

Operations must be commutative

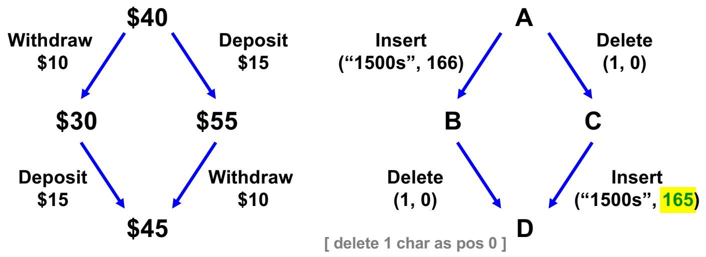


$40
Withdraw $10
Deposit $15
$30
$55
Deposit $15
Withdraw $10
$45

A
Insert (“1500s”, 166)
Delete (1, 0)
B
C
Delete (1, 0)
Insert (“1500s”, 166)
D
[delete 1 char as pos 0]

PROBLEM!

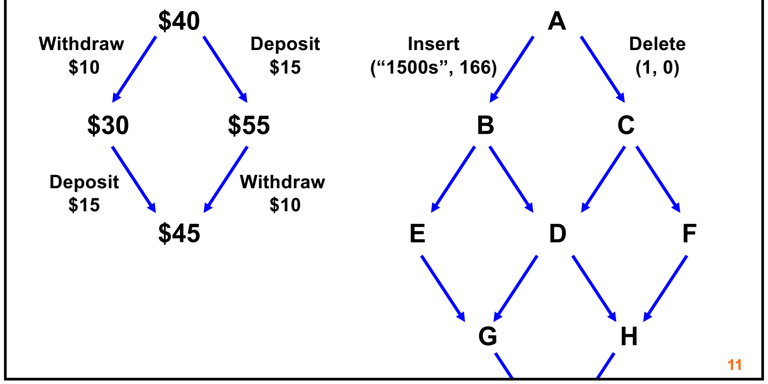
Operations must be commutative



$40
Withdraw $10
Deposit $15
$30
$55
Deposit $15
Withdraw $10
$45

A
Insert (“1500s”, 166)
Delete (1, 0)
B
C
Delete (1, 0)
Insert (“1500s”, 165)
D
[delete 1 char as pos 0]

Operations must be commutative



$40
Withdraw $10
Deposit $15
$30
$55
Deposit $15
Withdraw $10
$45

A
Insert (“1500s”, 166)
Delete (1, 0)
B
C
E
D
F
G
H

Operational Transformation

Pioneered in GROVE (GRoup Outline Viewing Edit)
C. Ellis and S. Gibbs, 1989

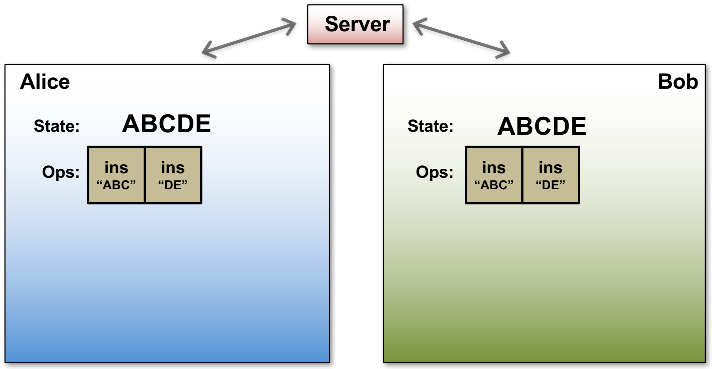
Now found in Apache Wave & Google Docs

Operational Transformation (OT)

- State of system is *S*, ops *a* and *b* performed by concurrently on state *S*
- Different servers can apply concurrent ops in different sequential order
  - Server 1:
    - Receives *a*, applies *a* to state *S*: *S* ◎ *a*
    - Receives b (which is dependent on *S*, not *S* ◎ *a*)
    - Transforms b across all ops applied since S (namely a): *b'* = OT(*b*, { *a* })
    - Applies b' to state: *S* ◎ *a* ◎ *b'*
  - Server 2
    - Receives b, applies b to state: *S* ◎ *b*
    - Receives *a*, performs transformation *a'* = OT(*a*, { *b* }),
    - Applies *a'* to state: *S* ◎ *b* ◎ *a'*
- Servers 1 and 2 have identical final states: *S* ◎ *a* ◎ *b'* == *S* ◎ *b* ◎ *a'*

Operational Transformation (OT)

(Used in Google Docs, EtherPad, etc.)



Server

Alice

State: ABCDE

Ops: ins "ABC" | ins "DE"

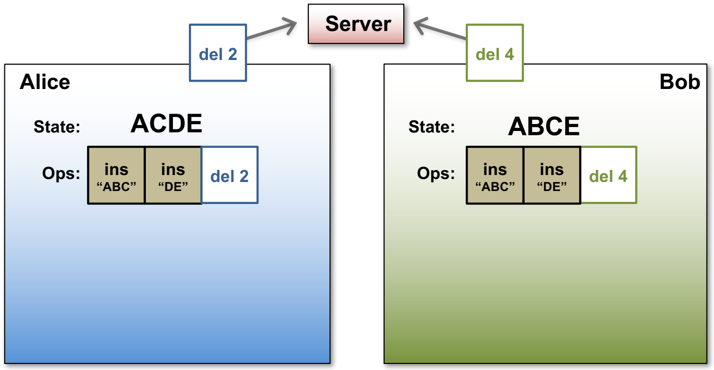
Bob

State: ABCDE

Ops: ins "ABC" | ins "DE"

Operational Transformation (OT)

(Used in Google Docs, EtherPad, etc.)



Server

del 2

del 4

Alice

State: ACDE

Ops: ins "ABC" | ins "DE" | del 2

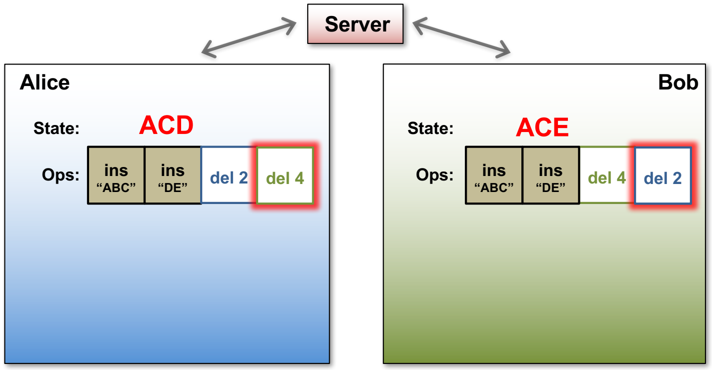
Bob

State: ABCE

Ops: ins "ABC" | ins "DE" | del 4

Operational Transformation (OT)

(Used in Google Docs, EtherPad, etc.)



Server

Alice

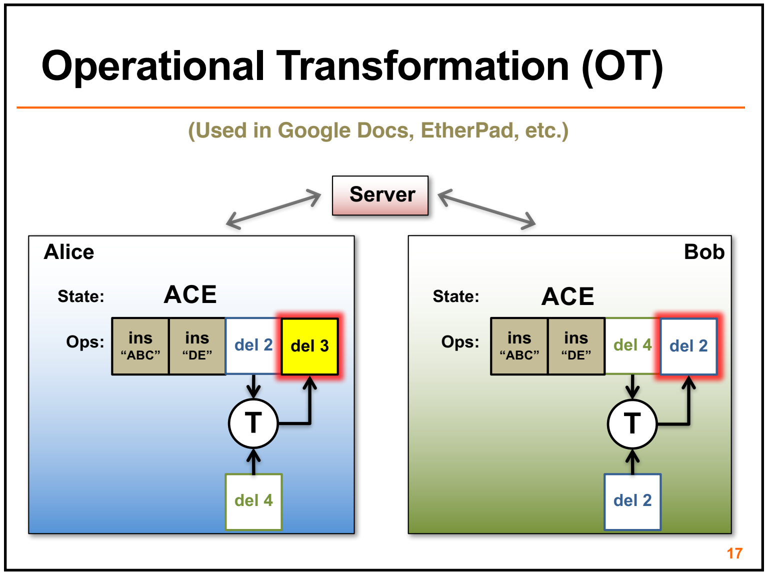
State: ACD

Ops: ins "ABC" | ins "DE" | del 2 | del 4

Bob

State: ACE

Ops: ins "ABC" | ins "DE" | del 4 | del 2



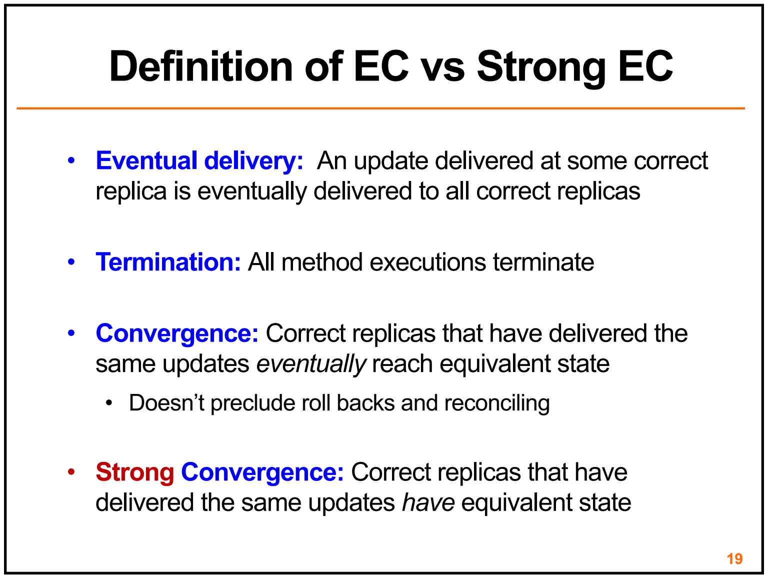
Operational Transformation (OT)

(Used in Google Docs, EtherPad, etc.)

More rigorous approach:

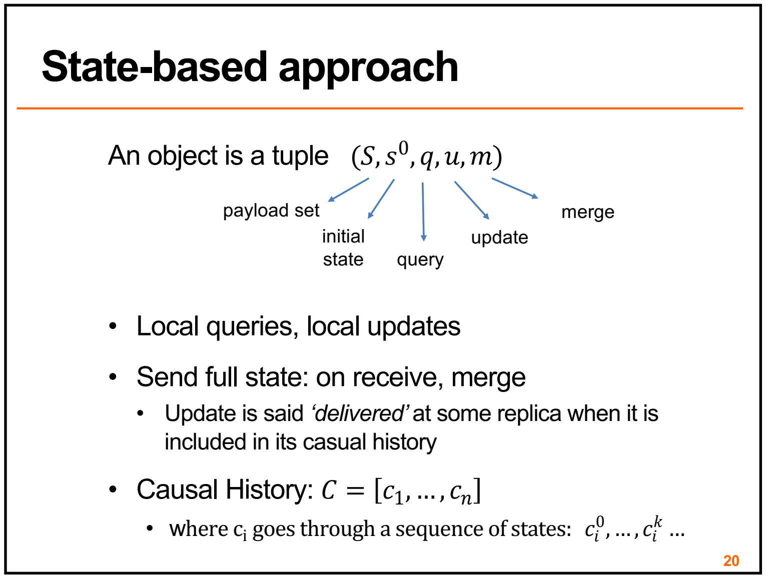
Conflict-free replicated data type

Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski
2011



Definition of EC vs Strong EC

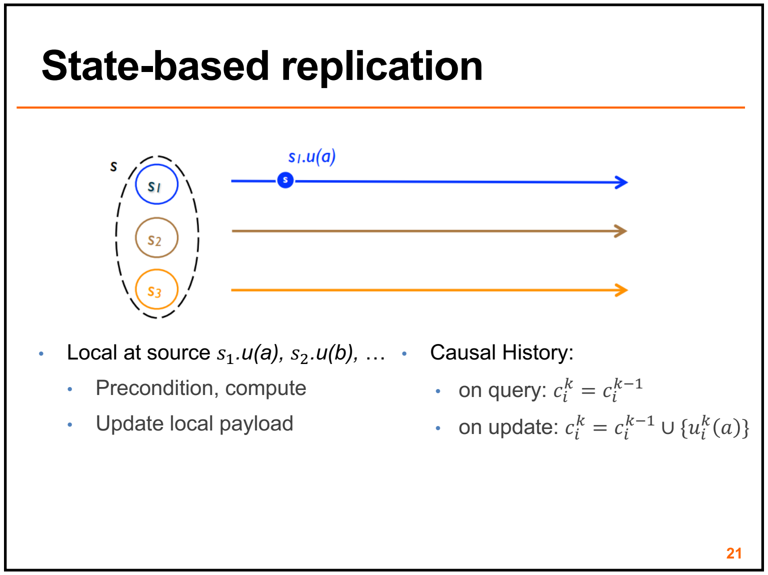
- **Eventual delivery:** An update delivered at some correct replica is eventually delivered to all correct replicas
- **Termination:** All method executions terminate
- **Convergence:** Correct replicas that have delivered the same updates *eventually* reach equivalent state
  - Doesn't preclude roll backs and reconciling
- **Strong Convergence:** Correct replicas that have delivered the same updates *have* equivalent state



State-based approach

An object is a tuple (S, s^0, q, u, m)

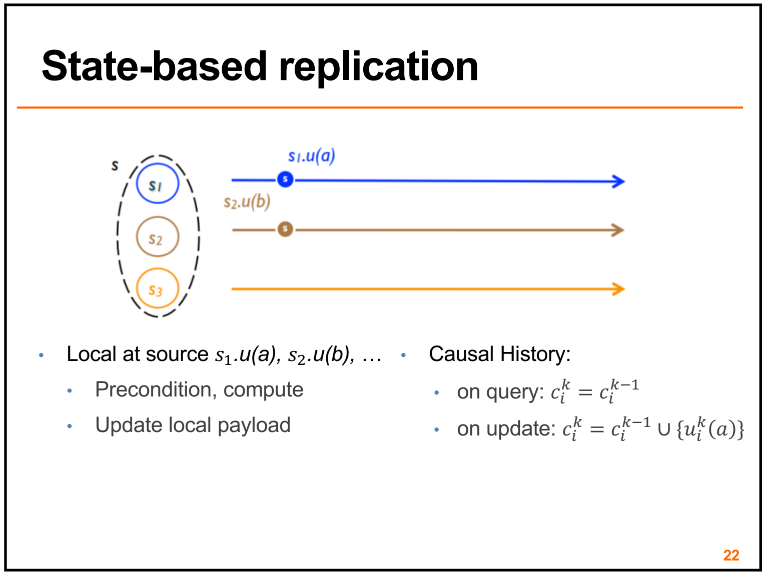
- Local queries, local updates
- Send full state: on receive, merge
  - Update is said *'delivered'* at some replica when it is included in its casual history
- Causal History: $C = [c_1, \ldots, c_n]$
  - where c_i goes through a sequence of states: $c_i^0, \ldots, c_i^k \ldots$



State-based replication

- Local at source s_1.*u(a)*, s_2.*u(b)*, …
  - Precondition, compute
  - Update local payload
- Causal History:
  - on query: $c_i^k = c_i^{k-1}$
  - on update: $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$

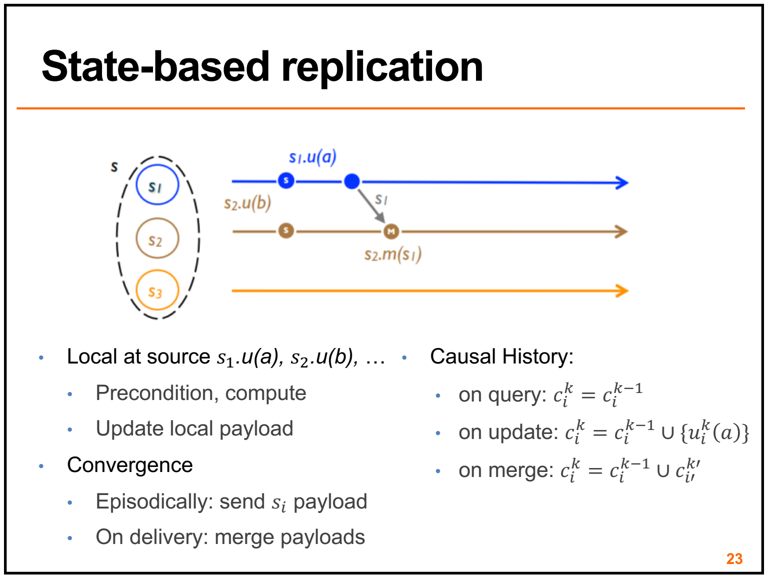
21



State-based replication

- Local at source s_1.*u(a)*, s_2.*u(b)*, …
  - Precondition, compute
  - Update local payload
- Causal History:
  - on query: $c_i^k = c_i^{k-1}$
  - on update: $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$

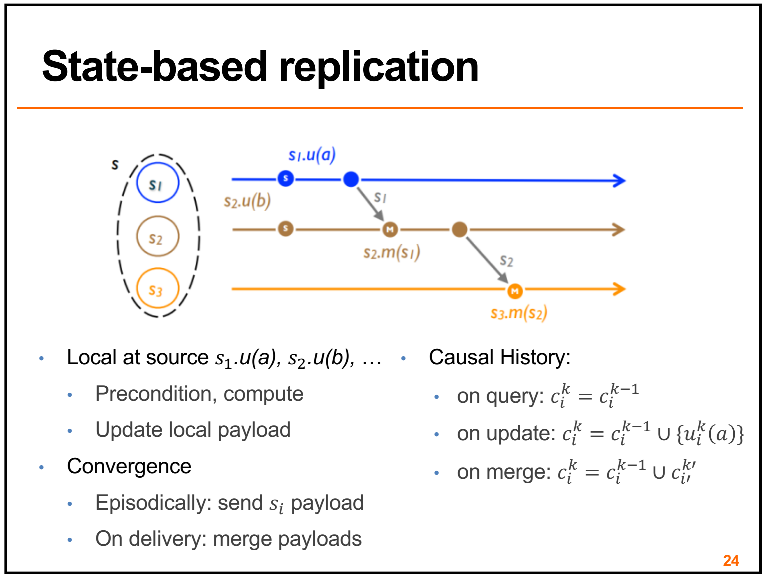
22



State-based replication

- Local at source s_1.*u(a)*, s_2.*u(b)*, …
  - Precondition, compute
  - Update local payload
- Convergence
  - Episodically: send s_i payload
  - On delivery: merge payloads
- Causal History:
  - on query: $c_i^k = c_i^{k-1}$
  - on update: $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$
  - on merge: $c_i^k = c_i^{k-1} \cup c_{i'}^{k'}$

23

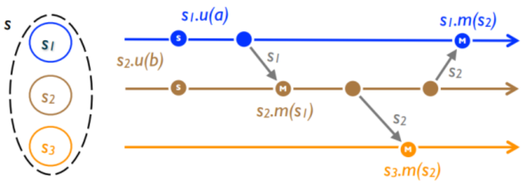


State-based replication

- Local at source s_1.*u(a)*, s_2.*u(b)*, …
  - Precondition, compute
  - Update local payload
- Convergence
  - Episodically: send s_i payload
  - On delivery: merge payloads
- Causal History:
  - on query: $c_i^k = c_i^{k-1}$
  - on update: $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$
  - on merge: $c_i^k = c_i^{k-1} \cup c_{i'}^{k'}$

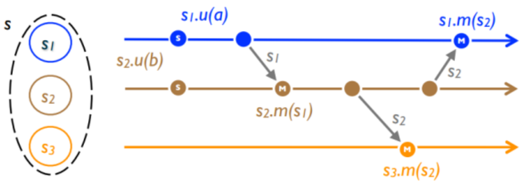
24

State-based replication



- Local at source s_1.*u(a)*, s_2.*u(b)*, …
  - Precondition, compute
  - Update local payload
- Convergence
  - Episodically: send s_i payload
  - On delivery: merge payloads
- Causal History:
  - on query: $c_i^k = c_i^{k-1}$
  - on update: $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$
  - on merge: $c_i^k = c_i^{k-1} \cup c_{i'}^{k'}$

State-based replication

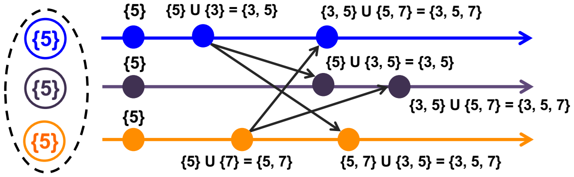


- Desired property:
  - After receiving all updates (irrespective of order), each replica will have same state

---

Example: Union Set

- u: add new element to local replica
- q: return entire set
- merge: union between remote set and local replica



Example

- Partial order ⊆ on sets
- ⊔ : U (set union)
- Then, we have:
  - – commutative: A U B = B U A
  - – idempotent: A U A = A
  - – associative: (A U B) U C = A U (B U C)

Example

- Partial order ≤ on set of integers
- $\sqcup$: max( )
- Then, we have:
  - commutative: max(x, y) = max(y, x)
  - idempotent: max(x, x) = x
  - associative: max(max(x, y), z) = max(x, max(y, z))

Example: Grow-Only Counter

```
payload integer[n] P
    initial [0,0,...,0]
update increment()
    let g = myId()
    P[g] := P[g] + 1
query value() : integer v
    let v = $\sum_i$ P[i]
compare (X, Y) : boolean b
    let b = ($\forall i \in$ [0, n - 1] : X.P[i] $\leq$ Y.P[i])
merge (X, Y) : payload Z
    let $\forall i \in$ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
```

Example: Positive-Negative Counter

```
payload integer[n] P, integer[n] N
    initial [0,0,...,0], [0,0,...,0]
update increment()
    let g = myId()
    P[g] := P[g] + 1
update decrement()
    let g = myId()
    N[g] := N[g] + 1
query value() : integer v
    let v = $\sum_i$ P[i] - $\sum_i$ N[i]
compare (X, Y) : boolean b
    let b = ($\forall i \in$ [0, n - 1] : X.P[i] $\leq$ Y.P[i] $\wedge$ $\forall i \in$
[0, n - 1] : X.N[i] $\leq$ Y.N[i])
merge (X, Y) : payload Z
    let $\forall i \in$ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
    let $\forall i \in$ [0, n - 1] : Z.N[i] = max(X.N[i], Y.N[i])
```

Semi-lattice

- Partial order ≤ set S with a least upper bound (LUB), denoted $\sqcup$
  - $m = x \sqcup y$ is a LUB of $\{ x, y \}$ under ≤ iff
    $\forall m', \; x \leq m' \wedge y \leq m'$
    $\Rightarrow x \leq m \wedge y \leq m \wedge m \leq m'$
- It follows that $\sqcup$ is:
  - commutative: $x \sqcup y = y \sqcup x$
  - idempotent: $x \sqcup x = x$
  - associative: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$

Monotonic Semi-lattice Object

- A state-based object with partial order ≤ and the following properties, is a *monotonic semi-lattice*:
  1. Set S of values forms a semi-lattice ordered by ≤
  2. Merging state s with remote state s′ computes the LUB of the two states, i.e., $s \bullet m(s') = s \sqcup s'$
  3. State is monotonically non-decreasing across updates, i.e., $s \leq s \bullet u$

Convergent Replicated Data Type (CvRDT)

- Theorem: Assuming eventual delivery and termination, any state-based object that satisfies the monotonic semi-lattice property is SEC
- Why?
  - Don’t care about order:
    - Merge is both commutative and associative
  - Don’t care about delivering more than once
    - Merge is idempotent

Commutative Replicated Data Type (CmRDT)

- Update-based CRDTs:
  - Sends update operations, not state like CvRDT
- Operations are commutative, but not idempotent
  - System must ensure all ops are delivered to other replicas, without duplication, but in any order
  - Often used in more complex settings for concurrent editing

Industry Use of CRDTs:

Databases: Redis, Riak, Facebook Apollo

Other: League of Legends Chat
Soundcloud user stream
TomTom device sync

New Module on Monday:

Replicated State Machines