Replication: Quorum based approaches + CRDTs

Philippas Tsigas

Distributed Systems, Advanced

TDA297(CTH), DIT290(GU)

What did we discuss during the previous lecture?

- ❑ **Replication for availiability: Lazy Updates**
  - ❑ **Gossiping**
- ❑ **Quorum Consenus**

“Pray with a quorum of 10
Debate with assembly of 9
Scottish dance with a collective of 8
Party with a gathering of 7
Play volleyball with a group of 6
Rank on a scale to 5
Practice music with a band of 4
Perceive in a dimension of 3
Make love with the intimacy of 2
Write poetry for an audience of 1”
— Beryl´Dov

Conflict-free Replicated Data Types

Philippas Tsigas

Distributed Systems, Advanced

TDA297(CTH), DIT290(GU)

Motivation

- Replication and Consistency - essential features of large distributed systems such as www, p2p, and cloud computing
- Lots of replicas
  - ✓ Great for fault-tolerance and read latency
  - × Problematic when updates occur
    - Slow synchronization
    - Conflicts in case of no synchronization

Motivation

- We look for an approach that:
  - supports Replication
  - guarantees Eventual Consistency
  - is Fast and Simple
- *Conflict-free* objects = no synchronization whatsoever
- Is this practical?

Theory

Strong Eventual Consistency (SEC)

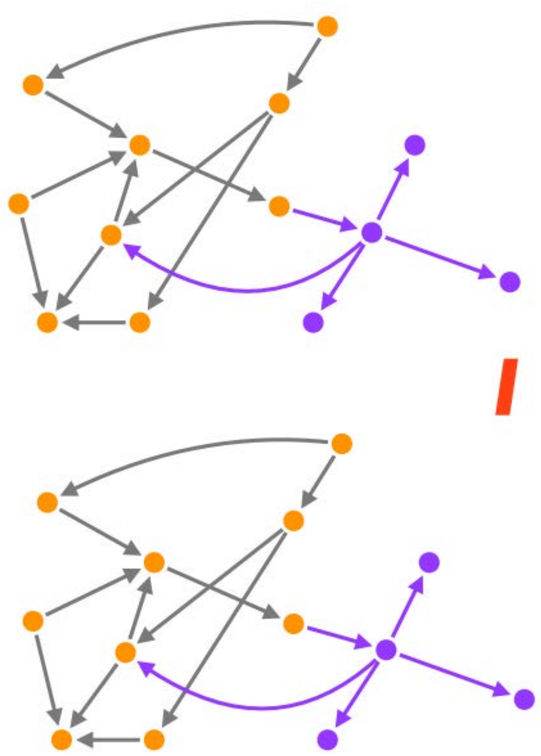
- A solution to the CAP problem
- Formal definitions
- Two sufficient conditions
- Strong equivalence between the two
- Incomparable to sequential consistency

Practice

CRDTs = Convergent or Commutative Replicated Data Types

- Counters
- Set
- Directed graph

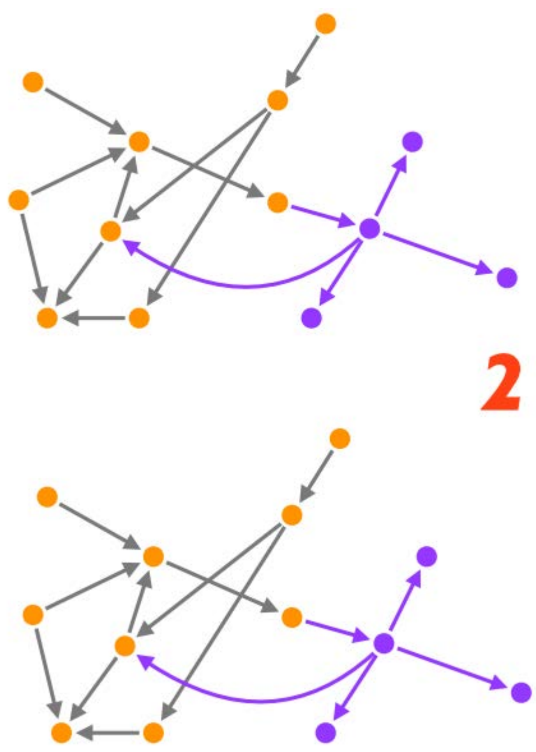
Strong Consistency



Ideal consistency: all replicas know about the update immediately after it executes

- Preclude conflicts
  - Replicas update in the same total order
  - Any deterministic object
- Consensus
  - Serialization bottleneck
  - Tolerates < n/2 faults
  - Correct, but doesn't scale

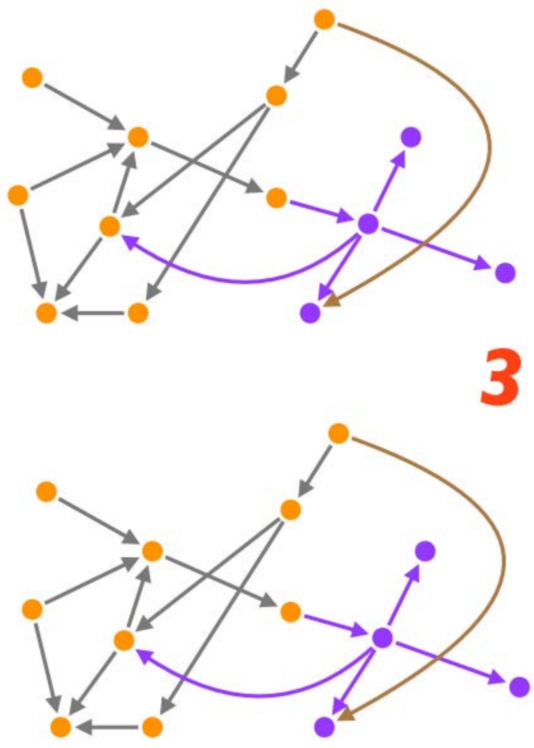
Strong Consistency



Ideal consistency: all replicas know about the update immediately after it executes

- Preclude conflicts
  - Replicas update in the same total order
  - Any deterministic object
- Consensus
  - Serialization bottleneck
  - Tolerates < n/2 faults
  - Correct, but doesn't scale

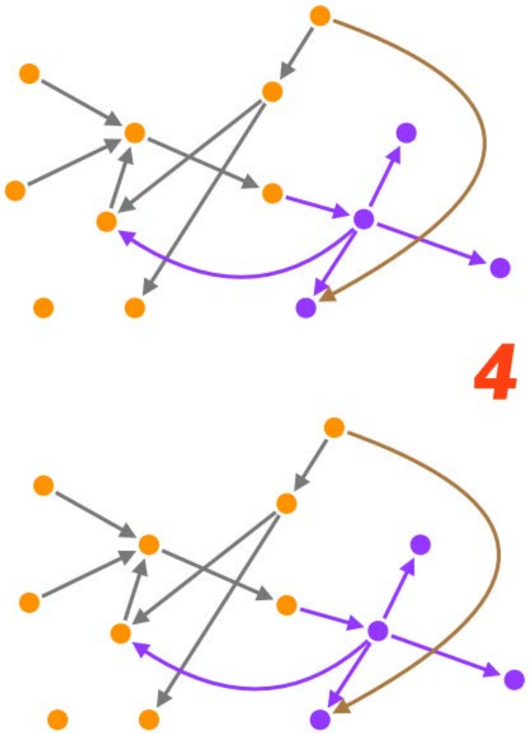
Strong Consistency



Ideal consistency: all replicas know about the update immediately after it executes

- Preclude conflicts
  - Replicas update in the same total order
  - Any deterministic object
- Consensus
  - Serialization bottleneck
  - Tolerates < n/2 faults
  - Correct, but doesn't scale

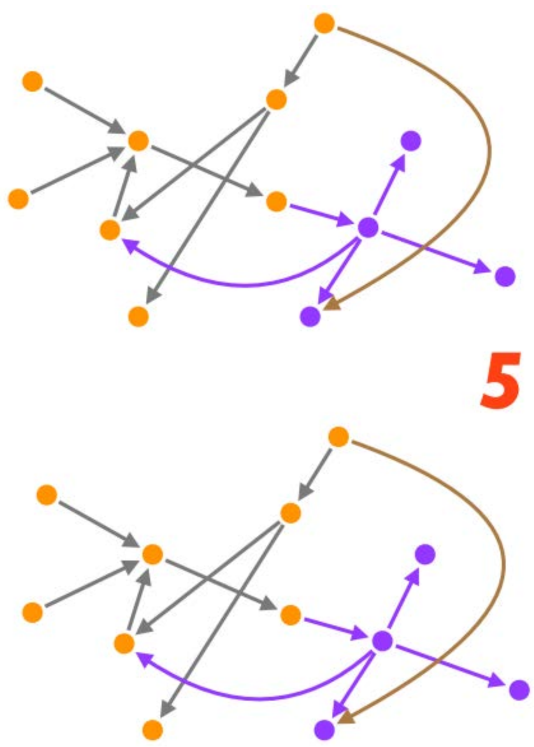
Strong Consistency



Ideal consistency: all replicas know about the update immediately after it executes

- Preclude conflicts
  - Replicas update in the same total order
  - Any deterministic object
- Consensus
  - Serialization bottleneck
  - Tolerates < n/2 faults
  - Correct, but doesn't scale

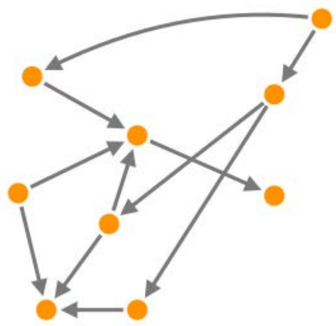
Strong Consistency

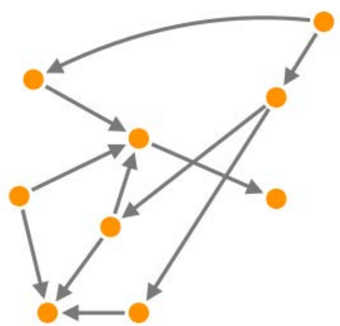


Ideal consistency: all replicas know about the update immediately after it executes

- Preclude conflicts
  - Replicas update in the same total order
  - Any deterministic object
- Consensus
  - Serialization bottleneck
  - Tolerates < n/2 faults
  - Correct, but doesn't scale

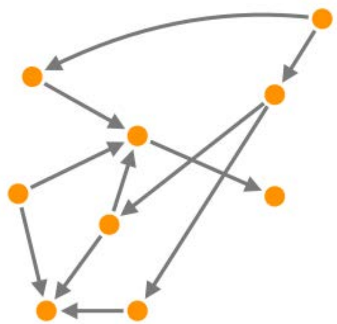
Lazy/Causal Consistency

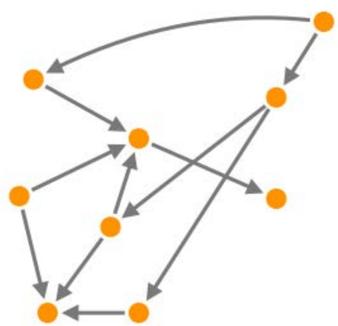




- Update local if no causal conflict and propagate
- On conflict
  - Request
- Consensus replaced with vector clocks
  - ✓ Better performance
  - × Still complex
  - × Causal consistency

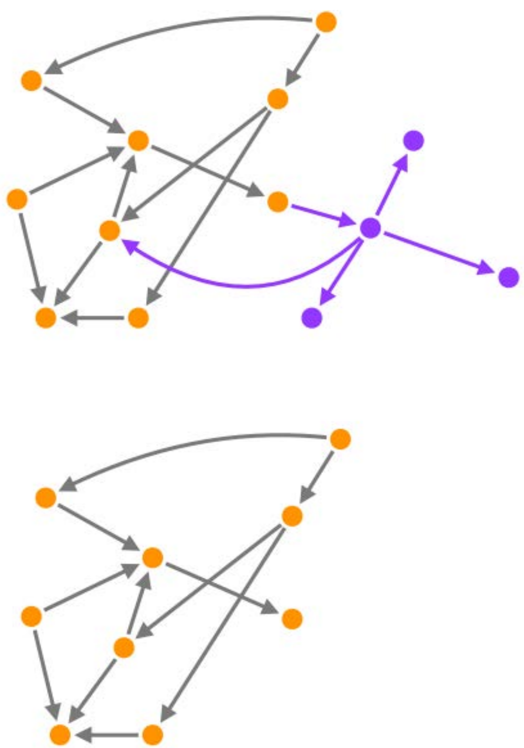
Eventual Consistency





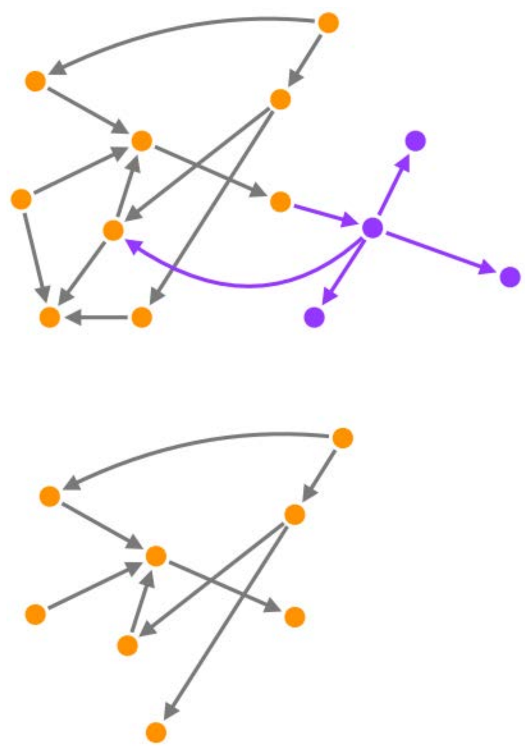
- ➢ Update local and propagate
  - ▪ No foreground synch
  - ▪ Eventual, reliable delivery
- ➢ On conflict
  - ▪ Arbitrate
  - ▪ Roll back
- ➢ Consensus moved to background
  - ✓ Better performance
  - × Still complex

Eventual Consistency



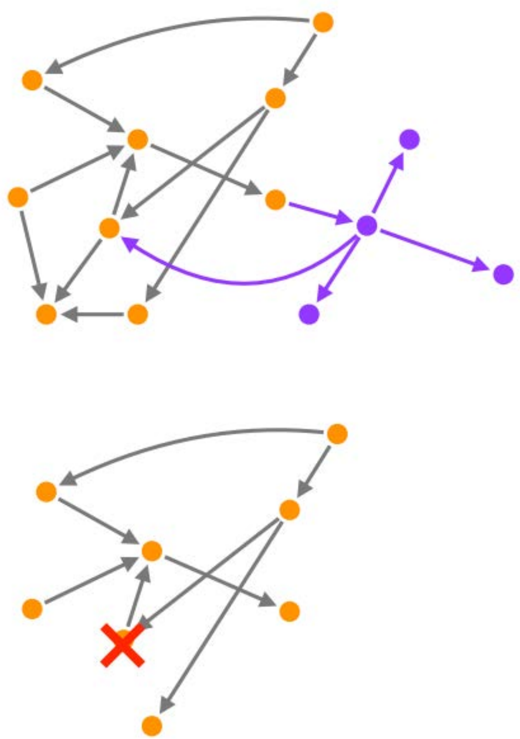
- Update local and propagate
  - No foreground synch
  - Eventual, reliable delivery
- On conflict
  - Arbitrate
  - Roll back
- Consensus moved to background
  - ✓ Better performance
  - × Still complex

Eventual Consistency



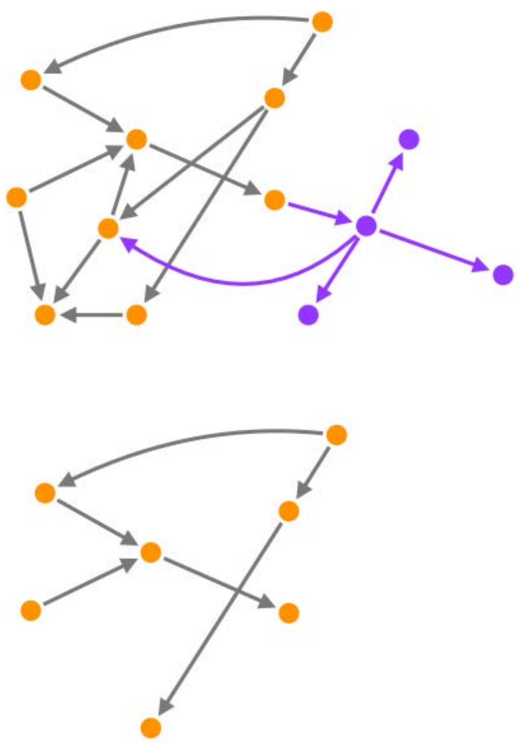
- Update local and propagate
  - No foreground synch
  - Eventual, reliable delivery
- On conflict
  - Arbitrate
  - Roll back
- Consensus moved to background
  - ✓ Better performance
  - × Still complex

Eventual Consistency



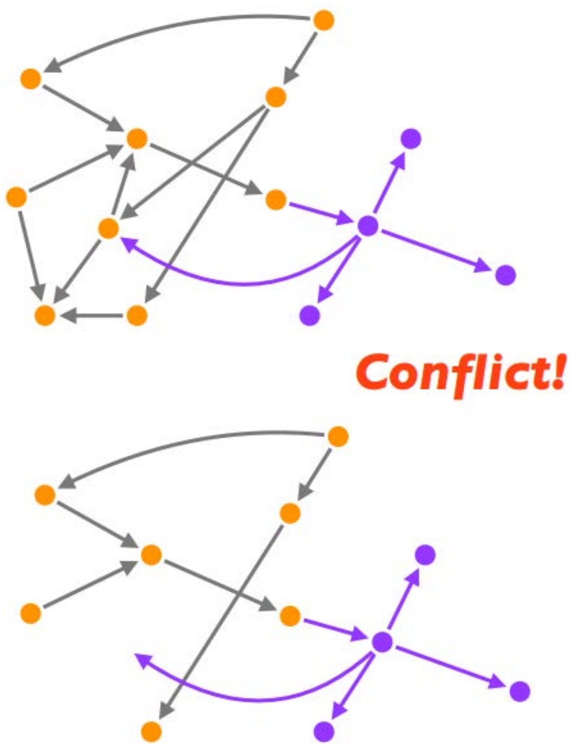
- Update local and propagate
  - No foreground synch
  - Eventual, reliable delivery
- On conflict
  - Arbitrate
  - Roll back
- Consensus moved to background
  - ✓ Better performance
  - × Still complex

Eventual Consistency



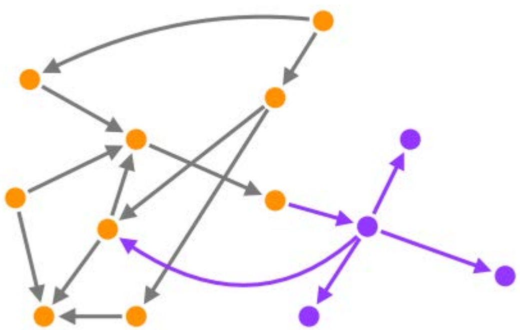
- Update local and propagate
  - No foreground synch
  - Eventual, reliable delivery
- On conflict
  - Arbitrate
  - Roll back
- Consensus moved to background
  - ✓ Better performance
  - × Still complex

Eventual Consistency

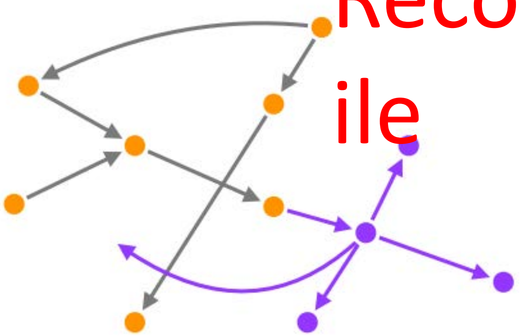


- Update local and propagate
  - No foreground synch
  - Eventual, reliable delivery
- On conflict
  - Arbitrate
  - Roll back
- Consensus moved to background
  - ✓ Better performance
  - × Still complex

Eventual Consistency

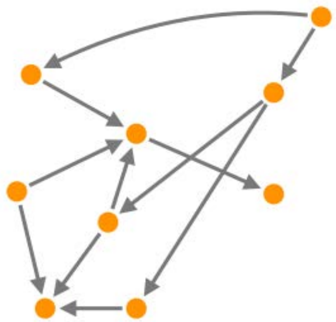


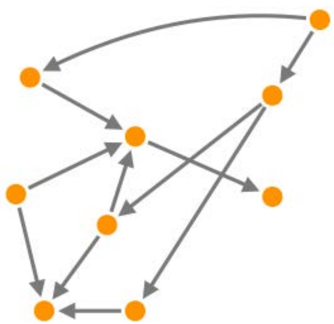
Reconc
ile



- Update local and propagate
  - No foreground synch
  - Eventual, reliable delivery
- On conflict
  - Arbitrate
  - Roll back
- Consensus moved to background
  - ✓ Better performance
  - × Still complex

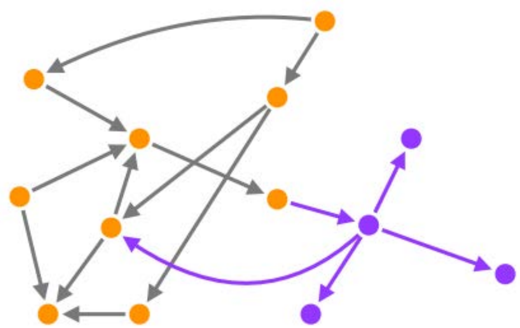
Strong Eventual Consistency

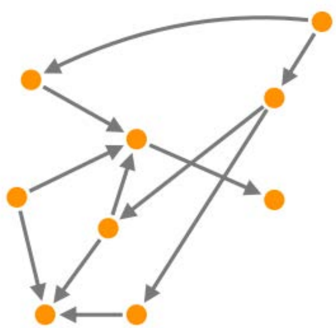




- Update local and propagate
  - No synch
  - Eventual, reliable delivery
- No conflict
  - deterministic outcome of concurrent updates
- No consensus: ≤ n-1 faults
- Solves the CAP problem

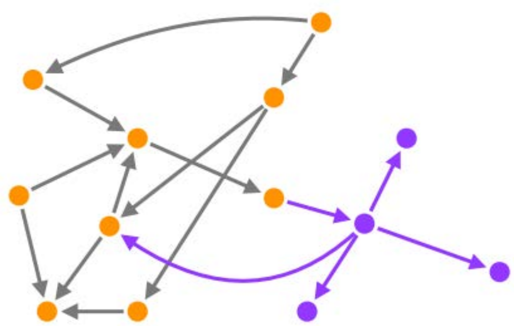
Strong Eventual Consistency

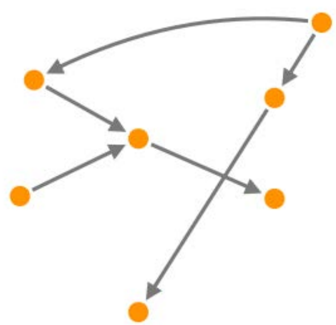




- ➢ Update local and propagate
  - No synch
  - Eventual, reliable delivery
- ➢ No conflict
  - deterministic outcome of concurrent updates
- ➢ No consensus: ≤ n-1 faults
- ➢ Solves the CAP problem

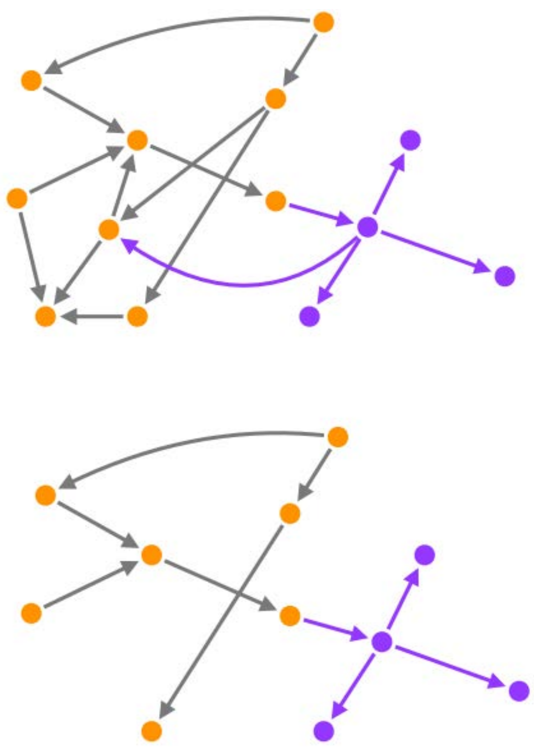
Strong Eventual Consistency





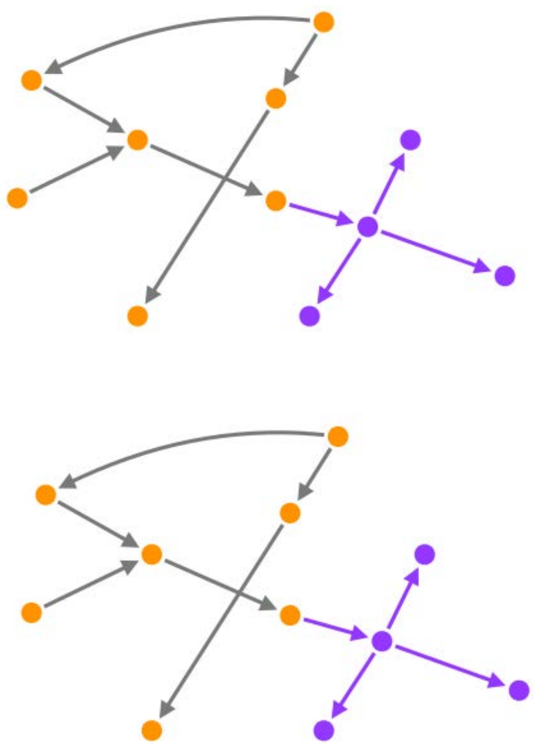
- Update local and propagate
  - No synch
  - Eventual, reliable delivery
- No conflict
  - deterministic outcome of concurrent updates
- No consensus: ≤ n-1 faults
- Solves the CAP problem

Strong Eventual Consistency



- Update local and propagate
  - No synch
  - Eventual, reliable delivery
- No conflict
  - deterministic outcome of concurrent updates
- No consensus: ≤ n-1 faults
- Solves the CAP problem

Strong Eventual Consistency



- ➢ Update local and propagate
  - No synch
  - Eventual, reliable delivery
- ➢ No conflict
  - deterministic outcome of concurrent updates
- ➢ No consensus: ≤ n-1 faults
- ➢ Solves the CAP problem

Definition of 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

Definition of SEC

- Eventual delivery: An update delivered at some correct replica is eventually delivered to all correct replicas
- Termination: All method executions terminate
- Strong Convergence: Correct replicas that have delivered the same updates *have* equivalent state

CRDT design concepts

Backward-compatible with sequential datatype

If operations commute, they can be concurrent

- *add(e); rm(f)* ≡ *rm(f); add(e)* ≡ *add(e) || rm(f)*

Otherwise, deterministic semantics

- Close to sequential *rm(e);add(e)* or *add(e); rm(e)*
- Don’t lose updates
- Result doesn't depend on order received
- Stable preconditions

CRDT Set design space

Many Set operations commute: *add(e) / add(f), add(e) / rm(f)*, etc.

Non-commuting pair: *add(e) / rm(e)*

- ~~sequential~~ consistency
- last writer wins? $\{ add(e)<rmv(e) \Rightarrow e \notin S$
  $\wedge rmv(e)<add(e) \Rightarrow e \in S \}$
- error state? $\{\perp_e \in S\}$
- add wins? $\{e \in S\}$
- remove wins? $\{e \notin S\}$

All deterministic, satisfy conditions

SEC is incomparable to sequential consistency

- There is a SEC object that is not sequentially-consistent:

Consider a Set CRDT S with operations *add(e)* and *remove(e)*

- *remove(e)* → *add(e)* ⇒ e ∈ S
- *add(e)* ‖ *remove(e')* ⇒ e ∈ S ∧ e' ∉ S
- *add(e)* ‖ *remove(e)* ⇒ e ∈ S (suppose *add* wins)

Consider the following scenario with replicas p_0, p_1, p_2:

1. p_0[*add(e)*; *remove(e')*] ‖ p_1[*add(e')*; *remove(e)*]
2. p_2 merges the states from p_0 and p_1 ⇒ p_2: e ∈ S ∧ e' ∈ S

The state of replica p_2 will never occur in a sequentially-consistent execution (either *remove(e)* or *remove(e')* must be last)

Numeric Invariants

Many applications need to enforce conditions like:

$$\text{counter} \geq K$$

E.g.:

- Number of impressions left ≥ 0
- Virtual money in a game ≥ 0

Numeric invariants

$$X \geq 0$$

Given $X = n$, there are n rights to execute *dec()*

Distribute rights among replicas

- Consume rights for *dec()*
- Create rights on *inc()*

CRDT-ish

Execute operations locally without coordination

Peer-to-peer synchronisation

Fail if not enough rights exist

Bounded Counter: API

Create(type, value);

Increment(value);
Decrement(value);
Value();

Transfer(to, qty);

Bounded Counter: increment

R_1

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

Increment(10);

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

R_2

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

Increment(15);

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	15	0	0
r_3	0	0	0	0

R_3

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

Increment(8);

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	0	0	8	0

Bounded Counter: increment

R	r_1	r_2	r_3	***U***
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

R₁

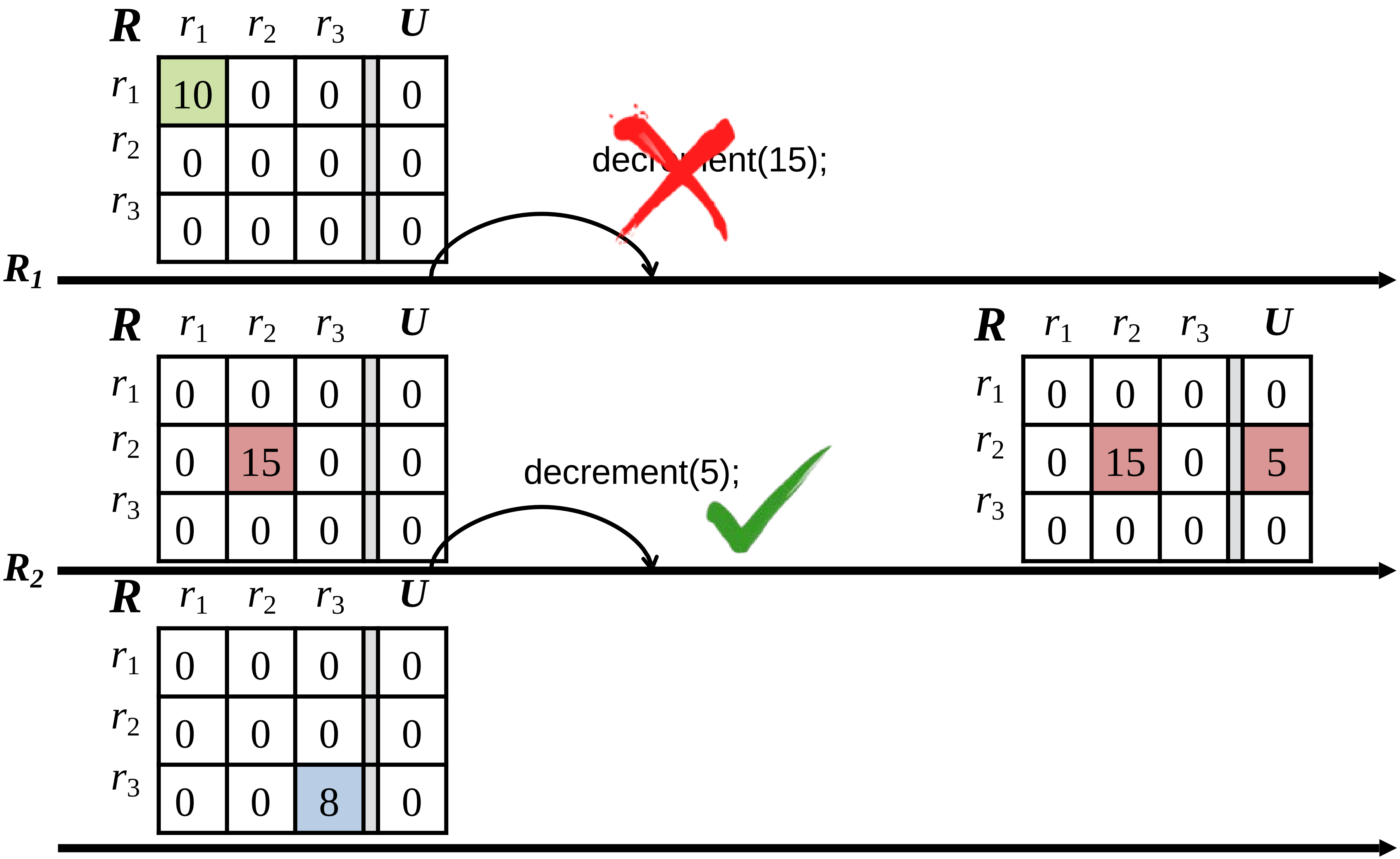
R	r_1	r_2	r_3	***U***
r_1	0	0	0	0
r_2	0	15	0	0
r_3	0	0	0	0

R₂

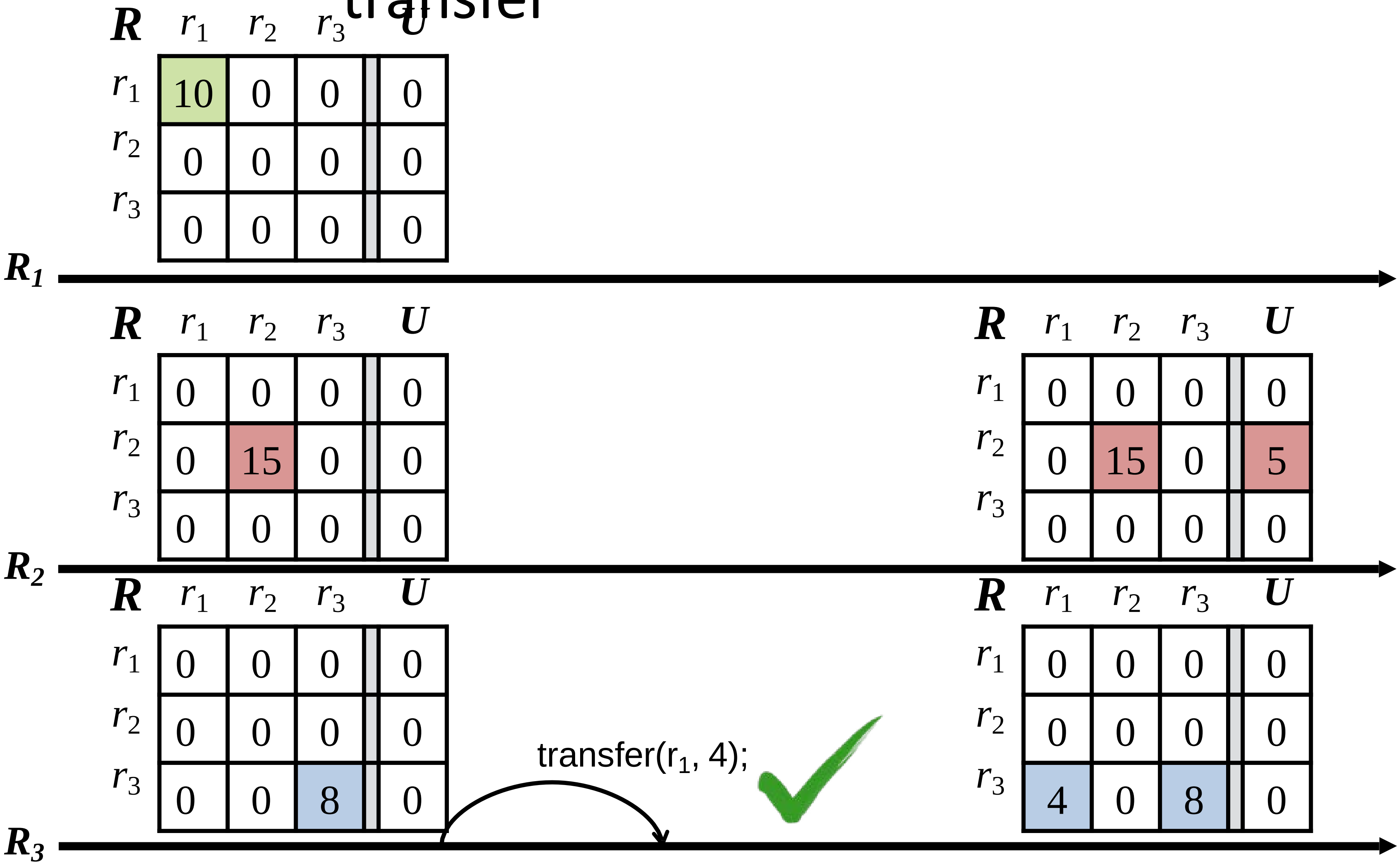
R	r_1	r_2	r_3	***U***
r_1	0	0	0	0
r_2	0	0	0	0
r_3	0	0	8	0

R₃

Bounded Counter: decrement



Bounded Counter: transfer



Bounded Counter: transfer

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

R_1

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

R_2

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

R_3

Bounded Counter: merge

R_1

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

Each replica only touches his line. Merge by taking max of each cell.

R_2

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

merge(r_1,r_2);

R_3

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

Bounded Counter: merge

Each replica only touches its line. Merge by taking max of each cell.

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

R_1

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

merge(r_1,r_2);

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

R_2

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

Bounded Counter: merge

R_1

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

Each replica only touches his line. Merge by taking max of each cell.

R_2

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

R_3

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

merge(r_3,r_2);

Bounded Counter: merge

R	r_1	r_2	r_3	*U*
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

Each replica only touches his line. Merge by taking max of each cell.

R₁

R	r_1	r_2	r_3	*U*
r_1	0	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

R	r_1	r_2	r_3	*U*
r_1	10	0	0	0
r_2	0	15	0	5
r_3	4	0	8	0

R₂

R	r_1	r_2	r_3	*U*
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

merge(r_3,r_2);

Bounded Counter: decrement

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	15	0	5
r_3	4	0	8	0

decrement(12);

R_1

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	15	0	5
r_3	4	0	8	0

Check local rights ≥ 12

R_2

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

Bounded Counter: decrement

R_1

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	15	0	5
r_3	4	0	8	0

decrement(12);

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	15	0	5
r_3	4	0	8	0

Check local rights ≥ 12

local = R[1][1] + ΣR[i][1] - ΣR[1][j] - U[1]

14 = 10 + 4 - 0 - 0

R_2

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

Using Bounded Counter

Operation execute locally; fail if no rights available

Redistribute rights

- On-demand when needed
- Proactive

Peer-to-peer synchronization

Prototype implemented on top of Riak

Some numbers from bet365

Largest European on-line betting operator

- Bursty load: 2.5 million simultaneous users
- 1 Tb working set
- 1000s servers
- Slow users: transient inconsistency OK
- Availability, read my writes, monotonic reads
- Transparency

Before: SQLserver, doesn't scale, hours to converge

mid 2013: noSQL riak: available, siblings; ad-hoc merge (hard!)

Summary

Applications requires multiple CRDTs

- Composition (e.g. Rick Map)

Need to lower expectations…

… but still possible to enforce some invariants

- Multi-key updates: HATs
- Causality
- Numeric invariants
- General invariants: red-blue, just-right consistency

Questions?

"The problem with eventual consistency jokes is that you can't tell who doesn't get it from who hasn't gotten it."

Next……

Based on slides from INRIA.