High Availability under Eventual Consistency

Latency Magnitudes

- λ, up to 50ms (local region DC)
- Λ, between 100ms and 300ms (inter-continental)

No inter-DC replication
Client writes observe λ latency

Planet-wide geo-replication

Replication Techniques	Latency Ranges	Description
Consensus/Paxos	$[\Lambda, 2\Lambda]$	with no divergence
Primary-Backup	$[\lambda, \Lambda]$	asynchronous/lazy
Multi-Master	λ	allowing divergence

EC and CAP for Geo-Replication

Eventually Consistent

- In an ideal world there would be only one consistency model: when an update is made all observers would see that update.
- Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.

High Availability

- Special case of weak consistency
- After an update, if no new updates are made to the object, eventually all reads will return the same value, that reflects the last update. E.g: DNS.

This can later be reformulated to avoid quiescence, by adapting a session guarantee.

CAP Theorem

Of three properties of shared-data systems – **data consistency, system availability, and tolerance to network partition** – only two can be achieved at any given time.

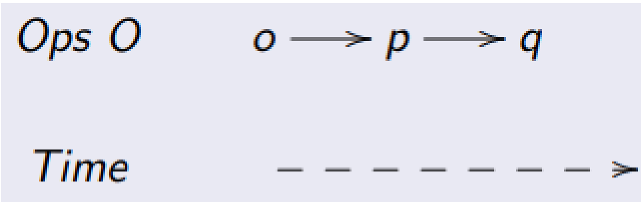
Session Guarantees

- **Read Your Writes** – read operations reflect previous writes
- **Monotonic Reads** – successive reads reflect a non-decreasing set of writes
- **Writes Follow Reads** – writes are propagated after reads on which they depend.
- **Monotonic Writes** – writes are propagated after writes that logically precede them.

From Sequential to Concurrent Executions

- Consensus provides illusion of a single replica
- This also preserves (slow) sequential behaviour
- EC Multi-master (or active-active) can expose concurrency

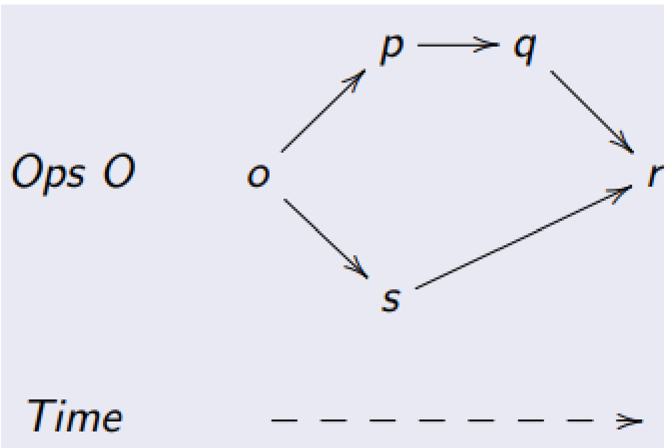
Sequential Execution



Ordered set (O, <).

- O = {o, p, q} and o < p < q

Concurrent Execution



Partially ordered set (O, <). o < p < q < r and o < s < r
Some ops in O are concurrent: p k s and q k s

Conflict-Free Replicated Data Types (CRDTs)

- Convergence after concurrent updates → favor AP under CAP
    - Examples include counters, sets, mv-registers, maps, graphs
- Operation based CRDTs → operation effects must commute

- State based CRDTs are rooted on join semi-lattices

Operation-based CRDTs, Effect Commutativity

- In some datatypes, all operations are commutative

State-based CRDTs, Join semi-lattices

S → (partial ordered set)
U → join, deriving least upper bounds
⊥ → initial state (usually the least element)

- Join properties in a semilattice (S, ≤, U):
    - **Idempotence**, a u a = a,
    - **Commutativity**, a u b = b u a,
    - **Associative**, (a u b) u c = a u (b u c).

Eventual Consistency, non stop

upds(a) ⊆ upds(b) ⇒ a ≤ b.

- This is slightly **weaker** than the previous definition and implies it: upds(a) = upds(b) ⇒ a = b.

Design of Conflict-Free Replicated Data Types

- A partially ordered log (polog) of operations implements any CRDT
- Replicas keep increasing local views of an evolving distributed polog
- Any query, at replica i, can be expressed from local polog Oi
- CRDTs are efficient representations that follow some general rules

Principle of permutation equivalence

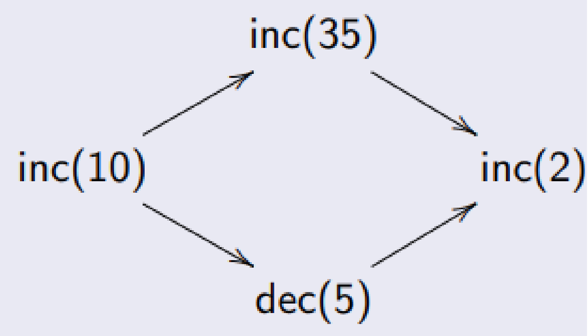
If operations in sequence can commute, preserving a given result, then under concurrency they should preserve the same result

Sequential

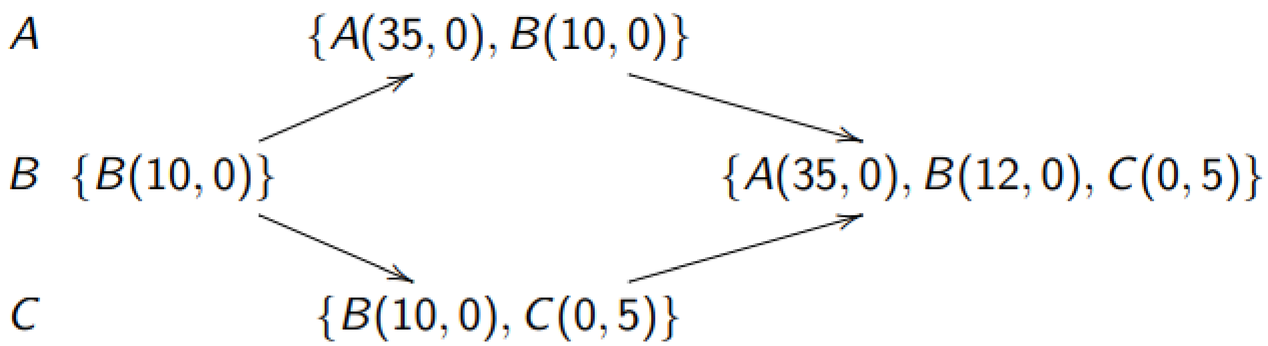
$\mathsf{inc}(10) \longrightarrow \mathsf{inc}(35) \longrightarrow \mathsf{dec}(5) \longrightarrow \mathsf{inc}(2)$

$\mathsf{dec}(5) \longrightarrow \mathsf{inc}(2) \longrightarrow \mathsf{inc}(10) \longrightarrow \mathsf{inc}(35)$

Concurrent



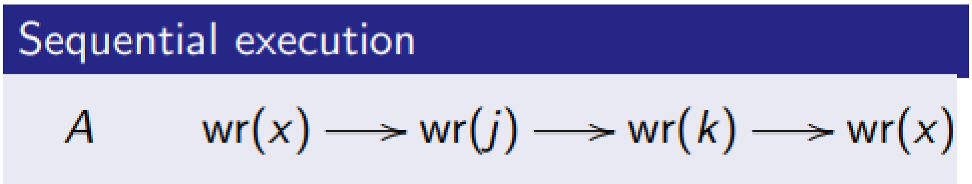
Implementing Counters

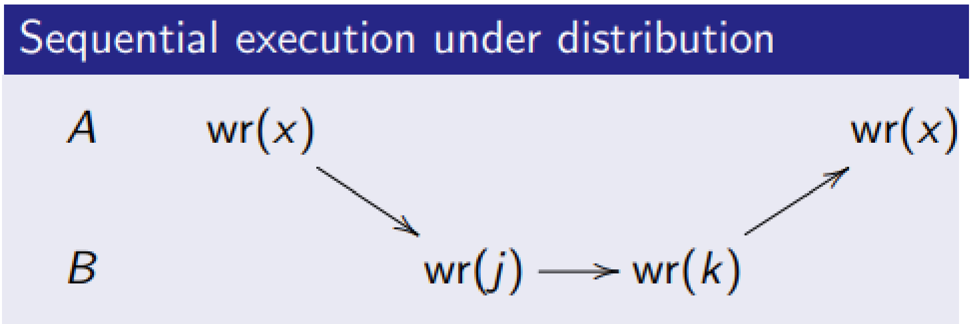


At any time, counter value is sum of incs minus sum of decs

Registers

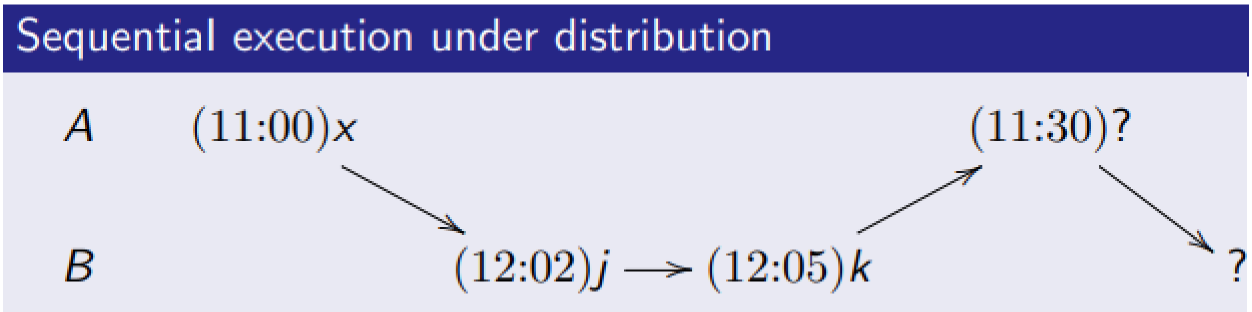
Ordered set of write operations
Register value is x, the last written value





Implementing Registers

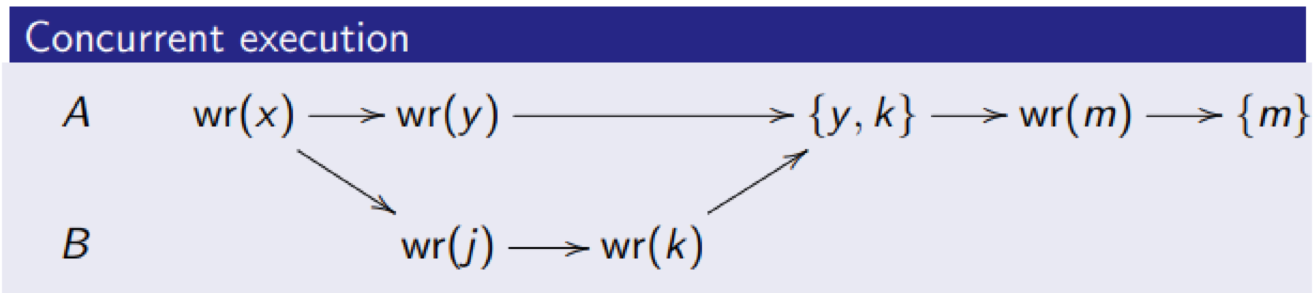
CRDT register implemented by attaching local wall-clock times



Problem: Wall-clock on B is one hour ahead of A
Value x might not be writeable again at A since 12:05 > 11:30

- Concurrent semantics should preserve the sequential semantics
    - This also ensures correct sequential execution under distribution

Multi-value Registers



Implementation

Concurrency can be precisely tracked with version vectors
Metadata can be compressed with a common causal context and a single scalar per

value (dotted version vectors)

Concurrent execution (version vectors)

$$\begin{tikzcd}
A & {[1,0]x} \arrow[r] \arrow[dr] & {[2,0]y} \arrow[rr] & & {[2,0]y, [1,2]k} \arrow[r] & {[3,2]m} \\
B & & {[1,1]j} \arrow[r] & {[1,2]k} \arrow[ur] & &
\end{tikzcd}$$

Registers in Redis

- Multi-value registers allows executions leading to concurrent values
- Presenting concurrent values is at odds with the sequential API
- Redis both tracks causality and registers wall-clock times
- Querying uses Last-Writer-Wins selection among concurrent values
- This preserves correctness of sequential semantics
- A value with clock 12:05 can still be causally overwritten at 11:30

State-based CRDTs: G-Set

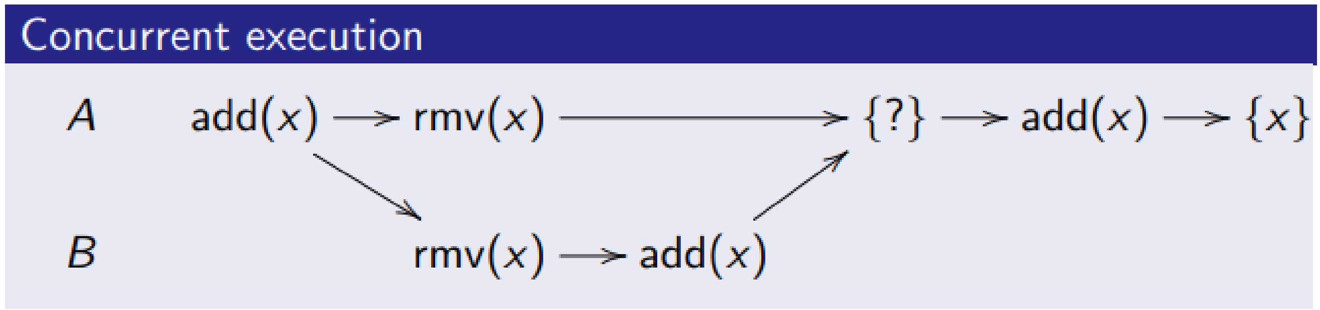
$$\begin{aligned}
\Sigma &= \mathcal{P}(V) \\
\sigma_i^0 &= \{\} \\
\mathsf{apply}_i((\mathsf{add}, v), s) &= s \cup \{v\} \\
\mathsf{eval}_i(\mathsf{rd}, s) &= s \\
\mathsf{merge}_i(s, s') &= s \cup s'
\end{aligned}$$

State-based CRDTs: 2P-Set

$$\begin{aligned}
\Sigma &= \mathcal{P}(V) \times \mathcal{P}(V) \\
\sigma_i^0 &= \{\}, \{\} \\
\mathsf{apply}_i((\mathsf{add}, v), (s, t)) &= s \cup \{v\}, t \\
\mathsf{apply}_i((\mathsf{rmv}, v), (s, t)) &= s, t \cup \{v\} \\
\mathsf{eval}_i(\mathsf{rd}, s) &= s \setminus t \\
\mathsf{merge}_i((s, t), (s', t')) &= s \cup s', t \cup t'
\end{aligned}$$

Sets

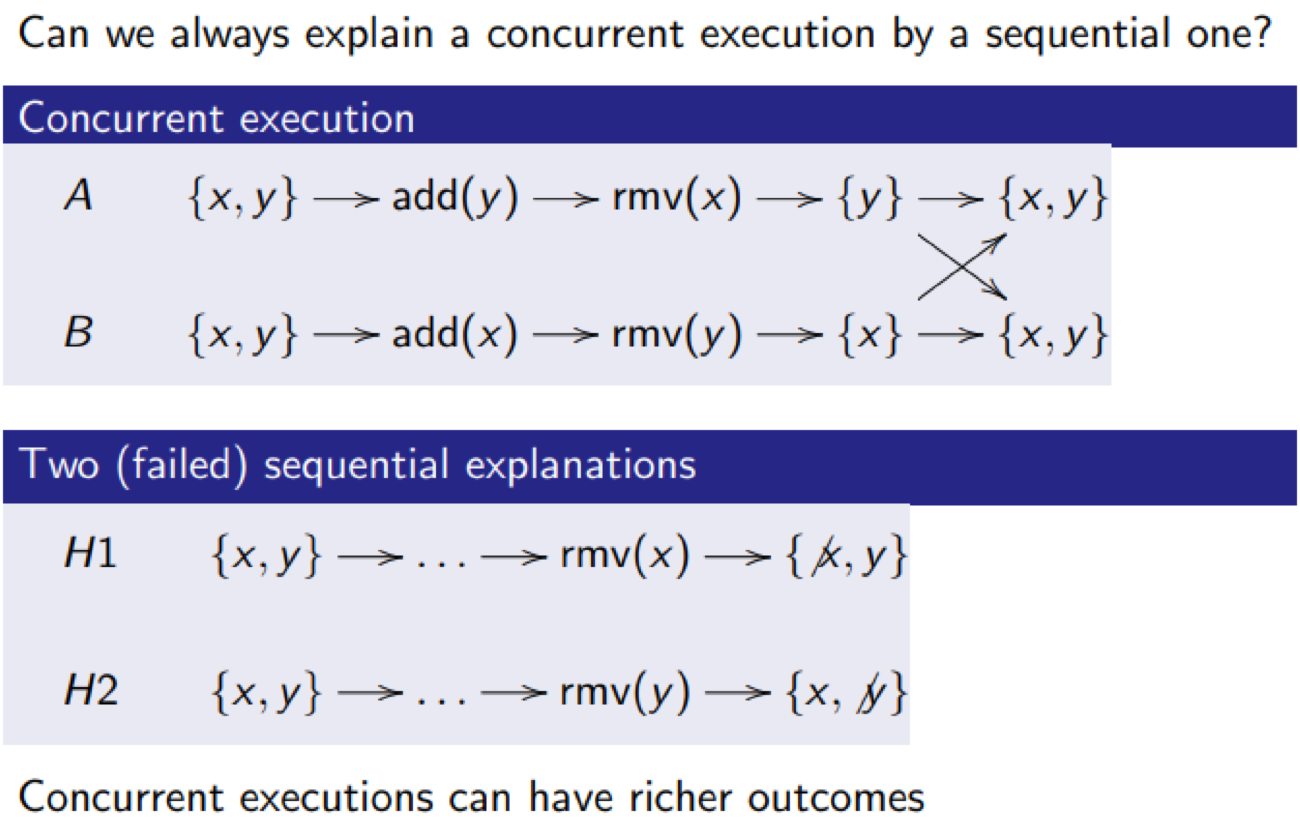
Problem: Concurrently adding and removing the same element



Add-Wins Sets

Consider a set of known operations Oi , at node i, that is ordered by an happens-before partial order <. Set has elements
{e | add(e) ∈ Oi ∧ ∄ rmv(e) ∈ Oi · add(e) < rmv(e)}

- Redis CRDT sets are Add-Wins Sets



Remove-Wins Sets

Xi .= {e | add(e) ∈ Oi ∧ ∀ rmv(e) ∈ Oi · rmv(e) < add(e)}

- Remove-Wins requires more metadata than Add-Wins
- Both Add and Remove-Wins have same semantics in a total order
- They are different but both preserve sequential semantics

Sequence/List

Element x is kept

$$
\begin{tikzcd}
 & & & \mathsf{rpush(b)} \arrow[r] & \langle xb \rangle \arrow[dr] & \\
\langle\rangle \arrow[r] & \mathsf{lpush(x)} \arrow[r] & \langle x \rangle \arrow[ur] \arrow[dr] & & & \langle axb \rangle \\
 & & & \mathsf{lpush(a)} \arrow[r] & \langle ax \rangle \arrow[ur] &
\end{tikzcd}
$$

Element x is removed (Redis enforces Strong Specification)

$$
\begin{tikzcd}
 & & & \mathsf{rpush(b)} \arrow[r] & \langle xb \rangle \arrow[dr] & & \\
\langle\rangle \arrow[r] & \mathsf{lpush(x)} \arrow[r] & \langle x \rangle \arrow[ur] \arrow[r] \arrow[dr] & \mathsf{rem(x)} \arrow[r] & \langle\rangle \arrow[r] & \langle ab \rangle & \neg\langle ba \rangle \\
 & & & \mathsf{lpush(a)} \arrow[r] & \langle ax \rangle \arrow[ur] & &
\end{tikzcd}
$$

Summary

- Concurrent executions are needed to deal with latency
- Behaviour changes when moving from sequential to concurrent

Road to accommodate transition:

- Permutation equivalence
- Preserving sequential semantics
- Concurrent executions lead to richer outcomes

CRDTs provide sound guidelines and encode policies