The consequences of concurrent updates

- Replicate the database, keep one copy in sf, one in nyc
  - Client sends reads to the nearest copy
  - Client sends update to both copies

**Inconsistent replicas!**

Updates should have been performed in the same order at each copy

- Deposit $100
- Pay 1% interest
- $1,000
- $1,100
- $1,110
- $1,010
- $1,110

Inconsistent replicas!

Totally-Ordered Multicast

**Goal:** All sites apply updates in (same) Lamport clock order

- Client sends update to one replica site \( j \)
  - Replica assigns it Lamport timestamp \( C_{j, j} \)

**Key idea:** Place events into a sorted local queue

- Sorted by increasing Lamport timestamps

Example: \( P_1 \)'s local queue:
**Totally-Ordered Multicast** *(Almost correct)*

1. On receiving an update from client, broadcast to others (including self)

2. On receiving an update from replica:
   a) Add it to your local queue
   b) Broadcast an acknowledgement message to every replica (including yourself)

3. On receiving an acknowledgement:
   • Mark corresponding update acknowledged in your queue

4. Remove and process updates everyone has acknowledged from head of queue

---

**Totally-Ordered Multicast** *(Correct Version)*

1. On receiving an update from client, broadcast to others (including self)

2. On receiving or processing an update:
   a) Add it to your local queue, if received update
   b) Broadcast an acknowledgement message to every replica (including yourself) only from head of queue

3. On receiving an acknowledgement:
   • Mark corresponding update acknowledged in your queue

4. Remove and process updates everyone has acknowledged from head of queue

---

**Totally-Ordered Multicast** *(Almost correct)*

- P1 queues $S$, P2 queues $\%$

- P1 queues and ack's $\%

- P1 marks $\%$ fully acknowledged

- P2 marks $\%$ fully acknowledged

\(X\) P2 processes %

(Acks to self not shown here)
So, are we done?

- Does totally-ordered multicast solve the problem of multi-site replication in general?
- Not by a long shot!

1. Our protocol assumed:
   - No node failures
   - No message loss
   - No message corruption
2. All to all communication does not scale
3. Waits forever for message delays (performance?)

Take-away points: Lamport clocks

- Can totally-order events in a distributed system: that’s useful!
- We saw an application of Lamport clocks for totally-ordered multicast
- But: while by construction, \( a \rightarrow b \) implies \( C(a) < C(b) \),
  - The converse is not necessarily true:
    - \( C(a) < C(b) \) does not imply \( a \rightarrow b \) (possibly, \( a \parallel b \))

Lamport Clocks Review

Q: \( a \rightarrow b \) => \( LC(a) < LC(b) \)

Q: \( LC(a) < LC(b) \) => \( b -/\rightarrow a \quad (a \rightarrow b \text{ or } a \parallel b) \)

Q: \( a \parallel b \) => nothing

Lamport Clocks and Causality

- Lamport clock timestamps do not capture causality
- Given two timestamps \( C(a) \) and \( C(z) \), want to know whether there’s a chain of events linking them:

\[ a \rightarrow b \rightarrow ... \rightarrow y \rightarrow z \]
Vector clock: Introduction

• One integer can't order events in more than one process
• So, a Vector Clock (VC) is a vector of integers, one entry for each process in the entire distributed system
• Label event e with VC(e) = [c₁, c₂, ..., cₙ]
  • Each entry cₖ is a count of events in process k that causally precede e

Vector clock: Update rules

• Initially, all vectors are [0, 0, ..., 0]
• Two update rules:
  1. For each local event on process i, increment local entry cᵢ
  2. If process j receives message with vector [d₁, d₂, ..., dₙ]:
     • Set each local entry cₖ = max{cₖ, dₖ}
     • Increment local entry cᵢ

Vector clock: Example

• All processes’ VCs start at [0, 0, 0]
• Applying local update rule
• Applying message rule
  • Local vector clock piggybacks on inter-process messages

Comparing vector timestamps

• Rule for comparing vector timestamps:
  • V(a) = V(b) when aₖ = bₖ for all k
  • V(a) < V(b) when aₖ ≤ bₖ for all k and V(a) ≠ V(b)
• Concurrency:
  • V(a) || V(b) if aᵢ < bᵢ and aⱼ > bⱼ, some i, j
• $V(w) < V(z)$ then there is a chain of events linked by Happens-Before ($\rightarrow$) between $a$ and $z$

• $V(a) \parallel V(w)$ then there is no such chain of events between $a$ and $w$

Vector clocks capture causality

Comparing vector timestamps

• Rule for comparing vector timestamps:
  • $V(a) = V(b)$ when $a_k = b_k$ for all $k$
    • They are the same event
  • $V(a) < V(b)$ when $a_k \leq b_k$ for all $k$ and $V(a) \neq V(b)$
    • $a \rightarrow b$
  • Concurrency:
    • $V(a) \parallel V(b)$ if $a_i < b_i$ and $a_j > b_j$, some $i, j$
      • $a \parallel b$

Two events $a, z$

Lamport clocks: $C(a) < C(z)$
  Conclusion: $z \land \rightarrow a$, i.e., either $a \rightarrow z$ or $a \parallel z$

Vector clocks: $V(a) < V(z)$
  Conclusion: $a \rightarrow z$

Vector clock timestamps precisely capture happens-before relation (potential causality)