Time, Part 2

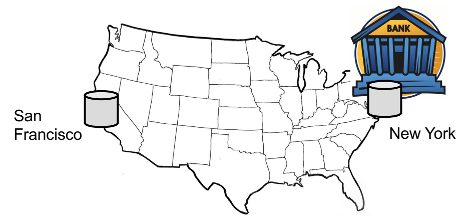
COS 418: Distributed Systems
Lecture 6

Mike Freedman

1

Motivation: Multi-site database replication

- A New York-based bank wants to make its transaction ledger database resilient to whole-site failures
- Replicate the database, keep one copy in sf, one in nyc



2

2

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

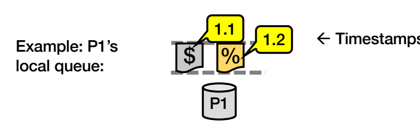
3

3

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



4

4

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 ack'ed from head of queue

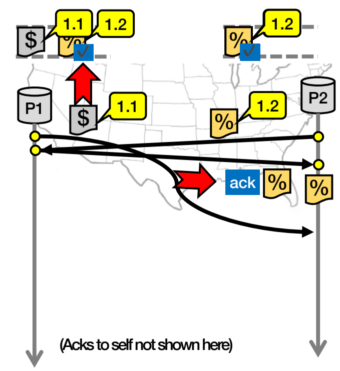
5

5

Totally-Ordered Multicast (Almost correct)

- P1 queues $, P2 queues %
- P1 queues and ack's %
  - P1 marks % fully ack'ed
- P2 marks % fully ack'ed

✗ P2 processes %



6

6

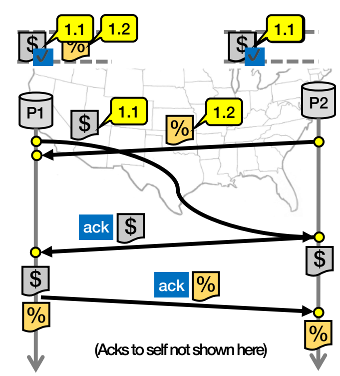
Totally-Ordered Multicast (Correct Version)

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 ack'ed from head of queue

7

7

Totally-Ordered Multicast (Correct Version)



8

8

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?)

9

9

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 → b implies *C*(a) < *C*(b),
  - The converse is not necessarily true:
    - *C*(a) < *C*(b) does not imply a → b (possibly, a || b)

Can't use Lamport timestamps to infer causal relationships between events

10

10

Lamport Clocks Review

Q: a → b => LC(a) < LC(b)

Q: LC(a) < LC(b) => b -/-> a (a → b or a || b)

Q: a || b => nothing

11

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 → b → ... → y → z

12

12

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) = $[c_1, c_2 \ldots, c_n]$
  - Each entry c_k is a count of events in process k that causally precede e

13

13

Vector clock: Update rules

- Initially, all vectors are [0, 0, …, 0]
- Two update rules:

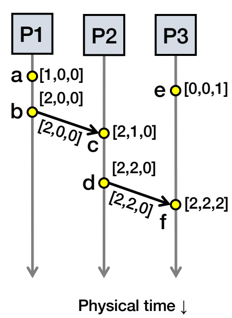
1. For each local event on process i, increment local entry c_i
2. If process j receives message with vector $[d_1, d_2, \ldots, d_n]$:
   - Set each local entry $c_k = \max\{c_k, d_k\}$
   - Increment local entry c_j

14

14

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



15

15

Comparing vector timestamps

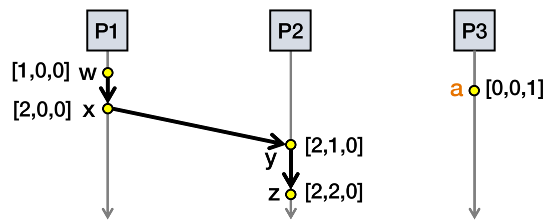
- Rule for comparing vector timestamps:
  - V(a) = V(b) when $a_k = b_k$ for all k
  - V(a) < V(b) when $a_k \leq b_k$ for all k and V(a) ≠ V(b)
- Concurrency:
  - V(a) || V(b) if $a_i < b_i$ and $a_j > b_j$, some i, j

16

16

Vector clocks capture causality

- V(w) < V(z) then there is a chain of events linked by Happens-Before (→) between a and z
- V(a) || V(w) then there is no such chain of events between a and w



17

17

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 $a_k \le b_k$ for all k and V(a) ≠ V(b)
    - a → b
- Concurrency:
  - V(a) || V(b) if $a_i < b_i$ and $a_j > b_j$, some i, j
    - a || b

18

18

Two events a, z

Lamport clocks: C(a) < C(z)

Conclusion: z -/-> a, i.e., either a → z or a || z

Vector clocks: V(a) < V(z)

Conclusion: a → z

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

19

19

20