Time 2: Totally Ordered Multicast & Vector Clocks

COS 418/518: Distributed Systems
Lecture 6

Wyatt Lloyd
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
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
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 $C_j \cdot j$

- Key idea: Place events into a sorted local queue
  - Sorted by increasing Lamport timestamps

Example: P1’s local queue:

Telegram: P1

Example $P1$’s local queue:

\[ 1.1 \quad 1.2 \quad \leftarrow \text{Timestamps} \]
Q1) What is bad about using order a,b,d,c?

Q2) What are all the valid lamport timestamp total orders of a—f?
1. On receiving an update from client, broadcast to others (including yourself)

2. On receiving an update from replica:
   a) Add it to your local queue
   b) Broadcast an acknowledgment 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
Totally-Ordered Multicast  *(Almost correct)*

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

X  P2 processes %

(Acks to self not shown here)
1. On receiving an update from client, broadcast to others (including yourself)

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

Why is this correct?
Totally-Ordered Multicast (Correct version)

(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?)
Lamport Clocks Review

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

Q: $LC(a) < LC(b)$ => $b \rightarrow/\rightarrow a$ ( $a \rightarrow b$ 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 $\text{VC}(e) = [c_1, c_2, \ldots, c_n]$
    • Each entry $c_k$ is a count of events in process $k$ that causally precede $e$
Vector clock: Update rules

• Initially, all vectors are \([0, 0, \ldots, 0]\)

• 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 \(c_j\)
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 $a_k = b_k$ for all $k$
  • $V(a) < V(b)$ when $a_k \leq b_k$ for all $k$ and $V(a) \neq V(b)$

• Concurrency:
  • $V(a) \parallel V(b)$ if $a_i < b_i$ and $a_j > b_j$, some $i, j$
Vector clocks capture causality

- $V(w) < V(z)$ 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 $w$
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:
  • $V(a) || V(b)$ if $a_i < b_i$ and $a_j > b_j$, some $i, j$
    • $a || b$
Two events $a, z$

Lamport clocks: $C(a) < C(z)$
Conclusion: $z \not\rightarrow a$, 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)