ATTRIBUTION

• These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license

• These slides incorporate material from:
  • Kyle Jamieson, Princeton University (also under a CC BY-NC-SA 3.0 Creative Commons license)
ANNOUNCEMENTS
Outline

1. Lamport clocks
2. Vector clocks
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.
MOTIVATION: MULTI-SITE DATABASE REPLICATION

• **Replicate** the database, keep one copy in sf, one in nyc
  
  • Client sends **query** to the **nearest** copy
  
  • Client sends **update to both** copies

Inconsistent replicas!

Updates should have been performed in the same order at each copy
IDEA: LOGICAL CLOCKS

- Landmark 1978 paper by Leslie Lamport
- Insight: only the events themselves matter

Idea: Disregard the precise clock time
Instead, capture just a “happens before” relationship between a pair of events
DEFINING “HAPPENS-BEFORE”

• Consider three processes: P1, P2, and P3

• **Notation:** Event a *happens before* event b (a \(\rightarrow\) b)
DEFINING “HAPPENS-BEFORE”

1. Can observe event order at a single process
DEFINING “HAPPENS-BEFORE”

1. If same process and a occurs before b, then a → b
DEFINING “HAPPENS-BEFORE”

1. If same process and a occurs before b, then a → b

2. Can observe ordering when processes communicate

Physical time ↓
DEFINING “HAPPENS-BEFORE”

1. If same process and a occurs before b, then a \rightarrow b

2. If c is a message receipt of b, then b \rightarrow c

![Diagram showing the order of events](image)
DEFINING “HAPPENS-BEFORE”

1. If same process and \( a \) occurs before \( b \), then \( a \rightarrow b \)
2. If \( c \) is a message receipt of \( b \), then \( b \rightarrow c \)
3. Can observe ordering transitively
1. If *same process* and $a$ occurs before $b$, then $a \rightarrow b$
2. If $c$ is a message receipt of $b$, then $b \rightarrow c$
3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$
We seek a clock time $C(a)$ for every event $a$.

Plan: Tag events with clock times; use clock times to make distributed system correct.

Clock condition: If $a \rightarrow b$, then $C(a) < C(b)$.
THE LAMPORT CLOCK ALGORITHM

• Each process $P_i$ maintains a local clock $C_i$

1. Before executing an event, $C_i \leftarrow C_i + 1$
1. Before executing an event $a$, $C_i \leftarrow C_i + 1$:

- Set event time $C(a) \leftarrow C_i$
1. Before executing an event \( b \), \( C_i \leftarrow C_i + 1 \):

- Set event time \( C(b) \leftarrow C_i \)

![Diagram showing Lamport clock algorithm with states for P1, P2, and P3, and event times for a, b, and c. Physical time decreases.]
1. Before executing an event $b$, $C_i \leftarrow C_i + 1$

2. Send the local clock in the message $m$
3. On process $P_j$ receiving a message $m$:

- Set $C_j$ and receive event time $C(c) \leftarrow 1 + \max\{ C_j, C(m) \}$
ORDERING ALL EVENTS

• **Break ties** by appending the process number to each event:

1. Process $P_i$ timestamps event $e$ with $C_i(e).i$

2. $C(a).i < C(b).j$ when:
   - $C(a) < C(b)$, or $C(a) = C(b)$ and $i < j$

• Now, for any two events $a$ and $b$, $C(a) < C(b)$ or $C(b) < C(a)$

• This is called a **total ordering** of events
MAKING CONCURRENT UPDATES CONSISTENT

- Recall multi-site database replication:
  - San Francisco (P1) deposited $100:
  - New York (P2) paid 1% interest:

We reached an inconsistent state.

Could we design a system that uses Lamport Clock total order to make multi-site updates consistent?
TOTALLY-ORDERED MULTICAST

- Client sends update to **one replica** \rightarrow Lamport timestamp \( C(x) \)

- **Key idea:** Place events into a **local queue**
  - Sorted by increasing \( C(x) \)

Goal: All sites apply the updates in (the same) Lamport clock order
1. On **receiving** an event from **client**, broadcast to others (including yourself)

1. On **receiving** an event from **replica**:
   a) Add it to your local queue
   b) Broadcast an **acknowledgement message** to every process (including yourself)

2. On **receiving** an **acknowledgement**:
   • Mark corresponding event **acknowledged** in your queue

3. **Remove and process** events **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

P2 processes %
1. On receiving an event from client, broadcast to others (including yourself)

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

3. When you receive an acknowledgement:
   • Mark corresponding event acknowledged in your queue

4. Remove and process events everyone has ack’ed from head of queue
TOTALLY-ORDERED MULTICAST (CORRECT VERSION)

1.1

1.2

$%

(P1)

$1.1

$1.2

(2)

(Ack's 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!
- 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, \( a \parallel b \))

Can’t use Lamport clock timestamps to infer causal relationships between events
Outline

1. Lamport clocks
2. Vector clocks
VECTOR CLOCK (VC)

• Label each event $e$ with a vector $V(e) = [c_1, c_2, ..., c_n]$
  • $c_i$ is a count of events in process $i$ that causally precede $e$

• 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 $[d_1, d_2, ..., d_n]$:
    • Set each local entry $c_k = \max\{c_k, d_k\}$
    • Increment local entry $c_j$
VECTOR CLOCK: EXAMPLE

- All counters start at \([0, 0, 0]\)
- Applying local update rule
- Applying message rule
  - Local vector clock piggybacks on inter-process messages

\([2,2,2]\): Remember we have event e at P3 with timestamp \([0,0,1]\). D’s message gets timestamp \([2,2,0]\), we take max to get \([2,2,1]\) then increment the local entry to get \([2,2,2]\).
VECTOR CLOCKS CAN ESTABLISH CAUSALITY

• Rule for comparing vector clocks:
  • \( V(a) = V(b) \) 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: \( a \parallel b \) if \( a_i < b_i \) and \( a_j > b_j \), some \( i, j \)

• \( V(a) < V(z) \) when there is a chain of events linked by \( \rightarrow \) between \( a \) and \( z \)
Two events a, z

Lamport clocks: $C(a) < C(z)$
**Conclusion:** None

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

Vector clock timestamps tell us about causal event relationships.
VC APPLICATION: CAUSALLY-ORDERED BULLETIN BOARD SYSTEM

• Distributed bulletin board application
  • Each post \( \rightarrow \) multicast of the post to all other users

• **Want:** No user to see a reply before the corresponding original message post

• Deliver message only **after** all messages that **causally precede** it have been delivered
  • Otherwise, the user would see a reply to a message they **could not find**
VC APPLICATION: CAUSALLY-ORDERED BULLETIN BOARD SYSTEM

• User 0 posts, user 1 replies to 0’s post; user 2 observes
UC San Diego