Logical clocks

Assign sequence numbers to messages
- All cooperating processes can agree on order of events
- vs. physical clocks: report time of day

Assume no central time source
- Each system maintains its own local clock
- No total ordering of events
  - No concept of happened-when

Happened-before

Lamport’s “happened-before” notation

\( a \rightarrow b \)  event \( a \) happened before event \( b \)

e.g.: \( a \): message being sent, \( b \): message receipt

Transitive:
  - if \( a \rightarrow b \) and \( b \rightarrow c \) then \( a \rightarrow c \)

Logical clocks & concurrency

Assign a “clock” value to each event
- If \( a \rightarrow b \) then \( \text{clock}(a) < \text{clock}(b) \)
- since time cannot run backwards

If \( a \) and \( b \) occur on different processes that do not exchange messages, then neither \( a \rightarrow b \) nor \( b \rightarrow a \) are true
  - These events are concurrent
  - Otherwise, they are causal

Event counting example

- Three systems: \( P_0, \ P_1, \ P_2 \)
- Events \( a, b, c, \ldots \)
- Local event counter on each system
- Systems occasionally communicate
Event counting example

Lamport's algorithm

• Each message carries a timestamp of the sender’s clock

• When a message arrives:
  - If receiver’s clock < message_timestamp
    - Set system clock to (message_timestamp + 1)
  - Else do nothing

• Clock must be advanced between any two events in the same process

Lamport's algorithm

Algorithm allows us to maintain time ordering among related events

- Partial ordering

Summary

• Algorithm needs monotonically increasing software counter

• Incremented at least when events that need to be timestamped occur

• Each event has a Lamport timestamp attached to it

• For any two events, where a \(\rightarrow\) b:
  \[ L(a) < L(b) \]

Problem: Identical timestamps

a \(\rightarrow\) b, b \(\rightarrow\) c, … : local events sequenced

i \(\rightarrow\) c, f \(\rightarrow\) d, d \(\rightarrow\) g, … : Lamport imposes a send \(\rightarrow\) receive relationship

Concurrent events (e.g., b & g; i & k) may have the same timestamp or not
Unique timestamps (total ordering)

We can force each timestamp to be unique
- Define global logical timestamp \((T_i, i)\)
  - \(T_i\) represents local Lamport timestamp
  - \(i\) represents process number (globally unique)
  - e.g., (host address, process ID)
- Compare timestamps:
  \((T_i, i)\) < \((T_j, j)\)
  if and only if
  \(T_i < T_j\) or \(T_i = T_j\) and \(i < j\)

Does not necessarily relate to actual event ordering

Problem: Detecting causal relations

If \(L(e) < L(e')\)
- We cannot conclude that \(e \rightarrow e'\)

By looking at Lamport timestamps
- We cannot conclude which events are causally related

Solution: use a vector clock
Vector clocks are a way to prove the sequence of events by keeping version history based on each process that made changes to an object

Example

- Group of processes: Alice, Bob, Cindy, David
- They concurrently modify one object: "what should we eat?"
- Each process keeps a local counter

Solution: use a vector clock

Vector clocks are a way to prove the sequence of events by keeping version history based on each process that made changes to an object.

Example
- Group of processes: Alice, Bob, Cindy, David
- They concurrently modify one object: "what should we eat?"
- Each process keeps a local counter

Rules:
1. Vector initialized to 0 at each process
   \[ V[i][j] = 0 \text{ for } i, j = 1, \ldots, N \]
2. Process increments its element of the vector in local vector before timestamping event:
   \[ V[i][j] = V[i][j] + 1 \]
3. Message is sent from process \(P_i\) with \(V_i\) attached to it
4. When \(P_j\) receives message, compares vectors element by element and sets local vector to higher of two values
   \[ V[j][i] = \max[V[i][j], V[j][i]] \text{ for } i = 1, \ldots, N \]

For example, received: \([0, 5, 12, 1]\), have: \([2, 8, 10, 1]\)
new timestamp: \([2, 8, 12, 1]\)
Comparing vector timestamps

Define
\[ V = V' \text{ if } V[i] = V'[i] \text{ for } i = 1 \ldots N \]
\[ V \leq V' \text{ if } V[i] \leq V'[i] \text{ for } i = 1 \ldots N \]

For any two events \( e, e' \)
- if \( e \rightarrow e' \) then \( V(e) < V(e') \)
- ... just like Lamport's algorithm
- if \( V(e) < V(e') \) then \( e \rightarrow e' \)

Two events are concurrent if neither \( V(e) \not< V(e') \) nor \( V(e') \not< V(e) \)

Vector timestamps

For any two events \( e, e' \) if \( e \rightarrow e' \) then \( V(e) < V(e') \)

Two events are concurrent if neither \( V(e) \not< V(e') \) nor \( V(e') \not< V(e) \)
Vector timestamps

Event | timestamp
--- | ---
a | (1,0,0)
b | (2,0,0)
c | (2,1,0)
d | (2,2,0)
e | (0,0,1)
f | (2,2,2)

Concurrent events:
- Events a and d have overlapping timestamps (2,0,0), indicating concurrent occurrence.
- Events c and d have overlapping timestamps (2,1,0), indicating concurrent occurrence.
- Events e and f have overlapping timestamps (0,0,1), indicating concurrent occurrence.
- Events e and f have overlapping timestamps (2,2,2), indicating concurrent occurrence.

Event | timestamp
--- | ---
a | (1,0,0)
b | (2,0,0)
c | (2,1,0)
d | (2,2,0)
e | (0,0,1)
f | (2,2,2)
Generalizing Vector Timestamps:

- A "vector" can be a list of tuples:
  - For processes \( P_1, P_2, P_3, \ldots \)
  - Each process has a globally unique Process ID, \( P_i \) (e.g., MAC_address:PID)
  - Each process maintains its own timestamp: \( T_{P1}, T_{P2}, \ldots \)
  - Vector: \( \{ <P_1, T_{P1}>, <P_2, T_{P2}>, <P_3, T_{P3}>, \ldots \} \)

- Any one process may have only partial knowledge of others
  - New timestamp for a received message:
    - Compare all matching sets of process IDs: set to highest of values
    - Any non-matched \( <P, T> \) sets get added to the timestamp
  - For a happened-before relation:
    - At least one set of process IDs must be common to both timestamps
    - Match all corresponding \( <P, T> \) sets: \( A: <P_i, T_a>, B: <P_i, T_b> \)
    - If \( T_a \leq T_b \) for all common processes \( P \), then \( A \rightarrow B \)

Summary: Logical Clocks & Partial Ordering:

- Causality
  - If \( a \rightarrow b \) then event \( a \) can affect event \( b \)

- Concurrency
  - If neither \( a \rightarrow b \) nor \( b \rightarrow a \) then one event cannot affect the other

- Partial Ordering
  - Causal events are sequenced

- Total Ordering
  - All events are sequenced