Logical Clocks

• If two machines do not interact, there is no need to synchronize them
• What usually matters is that processes agree on the order in which events occur rather than the time at which they occurred
  – Absolute time is not important
  – Use logical clocks

Event Ordering

• **Problem:** define a global ordering of all events that occur in a system
• Events in a single processor machine are locally ordered
• In a distributed system:
  – No global clock, local clocks may be unsynchronized
  – Can not order events on different machines using local times
• Key idea [Lamport]
  – Processes exchange messages
  – Message must be sent before received
  – Send/receive used to order events and synchronize logical clocks
Happens-Before (HB) Relation

• If A and B are events in the same process and A occurs before B, then A → B
• If A represents sending of a message and B is the receipt of this message, then A → B
• Relation is transitive:
  – A → B and B → C implies A → C
• Unordered events are *concurrent*
  – A !→ B and B !→ A implies A || B

Lamport’s Logical Clocks

• Goal: assign timestamps to events such that
  If A → B then timestamp(A) < timestamp(B)
• Lamport’s Algorithm
  – Each process i maintains a logical clock Li
  – Whenever an event occurs locally at i, Li = Li+1
  – When i sends message to j, piggyback Li
  – When j receives message from i, Lj = max (Li, Lj) + 1
• In this algorithm, A → B implies L(A) < L(B), but
  L(A) < L(B) does not necessarily imply A → B
Lamport’s Logical Clocks: An Example

Totally Ordered Multicast

- We need to guarantee that concurrent updates on a replicated database are seen in the same order everywhere. This requires a **totally-ordered multicast**
  - Update 1: add $100 to an account (initial value = $1000)
  - Update 2: add 1% interest to account
  - In absence of proper synchronization: replica 1 = $1111, replica 2 = $1110
- Lamport’s logical clocks can be used to implement totally-ordered multicast in a completely distributed fashion
Implementing Totally Ordered Multicast

- Assumptions:
  - No messages are lost
  - Messages from the same sender are received in the order they were sent
- Process $P_i$ sends timestamped message $m_i$ to all others. The message itself is put in a local queue.
- Any incoming message at $P_j$ is queued in queue, according to its timestamp, and ACKed to every other process.
- $P_j$ passes a message $m_i$ to its application if
  - $m_i$ is at the head of queue,
  - $m_i$ has been ACKed by each other process
- Observation: all processes will eventually have the same copy of the local queue, therefore, all messages are delivered in the same order everywhere

Causality

- Lamport’s logical clocks:
  - If $A \rightarrow B$ then $L(A) < L(B)$
  - Reverse is not true!!
    - Nothing can be said about events by comparing timestamps!
- Need to capture causality
  - If $A \rightarrow B$ then $A$ causally precedes $B$
  - Need a timestamping mechanism such that:
    - $T(A) < T(B)$ iff $A$ causally precedes $B$

Events:

- Event A: $m_1$ is received at $t=16$
- Event B: $m_2$ is sent at $t=20$
- $L(A) < L(B)$, but $A$ does not causally precede $B$. 
Vector Clocks

- Each process \( i \) maintains a vector clock \( V_i \) of size \( N \), where \( N \) is the number of processes
  - \( V_i[j] = \) number of events that have occurred at process \( i \)
  - \( V_i[j] = \) number of events \( i \) knows have occurred at process \( j \)
  - Initially, \( V_i[j] = 0 \) for all \( i \) and \( j \)

- Update vector clocks as follows
  - Local event at \( p_i \): increment \( V_i[i] \) by 1
  - When a message is sent from \( p_i \) to \( p_j \): piggyback vector \( V_i \)
  - Receipt of a message from \( p_i \) by \( p_j \):
    \[
    V_j[k] = \max(V_j[k], V_i[k]), j \neq k
    \]
    \[
    V_j[j] = V_j[j] + 1
    \]
  - Receiver is told about how many events the sender knows occurred at another process \( k \)

- We have \( V(A) < V(B) \) iff \( A \) causally precedes \( B \! \)
  - \( V(A) < V(B) \) iff for all \( i \), \( V(A)[i] \leq V(B)[i] \) and there exists \( k \) such that \( V(A)[k] < V(B)[k] \)

- \( A \) and \( B \) are concurrent iff \( V(A) \nless V(B) \) and \( V(B) \nless V(A) \)

Example contrasting vector and scalar clocks
• **Causally-ordered multicasting**: a message is delivered only if all messages that causally precede it have also been delivered

• Implementing causally-ordered multicasting:
  1. if $P_j$ receives a message from $P_i$, delay delivery of the message until
  2. $ts(m)[k] <= VC_j[k]$ for all $k \neq i$ (if $P_i$ has seen all messages seen by $P_i$ when it sent $m$)