CIS 505: Software Systems
Lecture Note on Logical Clocks

Insup Lee
Department of Computer and Information Science
University of Pennsylvania

CIS 505, Spring 2007

Clocks

1. physical clocks
   - Protocols to control drift exist, but physical clock timestamps cannot assign an ordering to "nearly concurrent" events.

2. logical clocks
   - Simple timestamps guaranteed to respect causality: "A's current time is later than the timestamp of any event A knows about, no matter where it happened or who told A about it."

3. vector clocks
   - Order(N) timestamps that say exactly what A knows about events on B, even if A heard it from C.

4. matrix clocks
   - Order(N^2) timestamps that say what A knows about what B knows about events on C.
   - Acknowledgement vectors: an O(N) approximation to matrix clocks.

Event Ordering

- When there is no common memory or clock, it is sometimes impossible to say which of two events occurred first.
- The happened-before relation is a partial ordering of events in distributed systems such that
  1. If A and B are events in the same process, and A was executed before B, then A \(\rightarrow\) B.
  2. If A is the event of sending a message by one process and B is the event of receiving that by another process, then A \(\rightarrow\) B.
  3. If A \(\rightarrow\) B and B \(\rightarrow\) C, then A \(\rightarrow\) C.
- If two events A and B are not related by the \(\Rightarrow\) relation, then they are executed concurrently (no causal relationship)

Causality Example: Event Ordering
Causality and Logical Time

- **Constraint:** The update ordering must respect potential causality.
  - Communication patterns establish a happened-before order on events, which tells us when ordering might matter.
  - Event $e_1$ happened-before $e_2$ iff $e_1$ could possibly have affected the generation of $e_2$: we say that $e_1 < e_2$.
    - $e_1 < e_2$ iff $e_1$ was “known” when $e_2$ occurred.
    - Events $e_1$ and $e_2$ are potentially causally related.

Logical Clocks [Lamport]

- **Solution:** timestamp updates with logical clocks. Timestamping updates with the originating node’s logical clock $LC$ induces a partial order that respects potential causality.
  - **Clock condition:** $e_1 < e_2$ implies that $LC(e_1) < LC(e_2)$
    1. Each site maintains a monotonically increasing clock value $LC$.
    2. Globally visible events (e.g., updates) are timestamped with the current $LC$ value at the generating site. Increment local $LC$ on each new event: $LC = LC + 1$
    3. Piggyback current clock value on all messages. Receiver resets local $LC$: if $LC_s > LC_r$, then $LC_r = LC_s + 1$
  - Use processor ids to break ties to create a total ordering.

Logical Clocks: Example

- $LC$ update advances receiver’s clock if it is “running slow” relative to sender.
- $A6-A10$: receiver’s clock is unaffected because it is “running fast” relative to sender.
- $C5$: $LC$ update advances receiver’s clock if it is “running slow” relative to sender.

Causality and Updates: Example

- $A1 < B2 < C3$  
  $B3 < A4$  
  $C3 < A5$

Use processor ids to break ties to create a total ordering.
Update Ordering

- **Problem:** how to ensure that all sites recognize a fixed order on updates, even if updates are delivered out of order?

- **Solution:** Assign timestamps to updates at their accepting site, and order them by source timestamp at the receiver.
  - Assign nodes unique IDs: break ties with the origin node ID.
  - Problem: What (if different) ordering exists between updates accepted by different sites?
    - Comparing physical timestamps is arbitrary: physical clocks drift.
    - Even a protocol to maintain loosely synchronized physical clocks cannot assign a meaningful ordering to events that occurred at “almost exactly the same time”.

**Example: Lamport’s Algorithm**

- Three processes, each with its own clock. The clocks run at different rates.
- Lamport’s Algorithm corrects the clock.

  - Note: \(ts(A) < ts(B)\) does not imply \(A\) happened before \(B\).
  - What if we use this to synchronize physical clocks?

**Motivation for Vector Clocks**

- **Logical clocks** induce an order consistent with causality, but
  - the converse of the clock condition does not hold: it may be that \(LC(e_j) < LC(e_j)\) even if \(e_1\) and \(e_2\) are concurrent.
    - If \(A\) could know anything \(B\) knows, then it must be \(LC_A > LC_B\).
    - But if \(LC_A > LC_B\) then this doesn’t make it so; i.e., “false positives”.
    - Concurrent updates may be ordered unnecessarily.
  - We need a clock mechanism that is necessary and **sufficient** in capturing causality.

**Vector Clocks**

- **Vector clocks** (AKA vector timestamps or version vectors) are a more detailed representation of what a site might know.
  1. In a system with \(N\) nodes, each site keeps a vector timestamp \(TS[i]\) as well as a logical clock \(LC\).
    - \(TS[i]\) at site \(i\) is the most recent value of site \(j\)’s logical clock that site \(i\) “heard about”.
    - \(TS[i] = LC_j\); each site keeps its own \(LC\) in \(TS[i]\).
  2. When site \(i\) generates a new event, it increments its logical clock:
    - \(TS[i] = TS[i] + 1\)
  3. A site \(r\) observing an event (e.g., receiving a message) from site \(s\) sets its \(TS_r\) to the pairwise maximum of \(TS_s\) and \(TS_r\).
    - For each site \(i\), \(TS[i] = \max(TS[i], TS[j])\)
Vector Clocks: Example

Question: what if I have two updates to the same data item, and neither timestamp dominates the other?

Vector Clocks and Causality

- Vector clocks induce an order that exactly reflects causality.
  - Tag each event $e$ with current $TS$ vector at originating site.
  - $e_i \text{ happened-before } e_j$ if and only if $TS(e_j) \text{ dominates } TS(e_i)$
  - “Every event or update visible when $e_i$ occurred was also visible when $e_j$ occurred.”
  - Proof?
  - Vector timestamps allow us to ask if two events are concurrent, or if one happened-before the other.
  - If $e_i < e_j$ then $LC(e_i) < LC(e_j)$ and $TS(e_j)$ dominates $TS(e_i)$.
  - “If $TS(e_j)$ does not dominate $TS(e_i)$ then it is not true that $e_i < e_j$.”

The Need for Propagating Acknowledgments

- Vector clocks tell us what $B$ knows about $C$, but they do not reflect what $A$ knows about what $B$ knows about $C$.
  - Nodes need this information to determine when it is safe to discard/stabilize updates.
  - $A$ can always tell if $B$ has seen an update $u$ by asking $B$ for its vector clock and looking at it.
    - If $u$ originated at site $i$, then $B$ knows about $u$ if and only if $TS_u$ covers its accept stamp $LC_i$. $TS_u[i] \geq LC_i$.
  - $A$ can only know that every site has seen $u$ by looking at the vector clocks for every site.
    - Even if $B$ recently received updates from $C$, $A$ cannot tell (from looking at $B$’s vector clock) if $B$ got $u$ from $C$ or if $B$ was already aware of $u$ when $C$ contacted it.

Solution: Matrix Clocks

- Matrix clocks extend vector clocks to capture “what $A$ knows about what $B$ knows about $C$”.
  - Each site $i$ maintains a matrix $MC_i(N,N)$.
    - Row $j$ of $i$’s matrix clock $MC_i$ is the most recent value of $j$’s vector clock $TS_j$ that $i$ has heard about.
    - $MC[i, j] = LC_i$, and $MC[i, j]^* = TS_j$.
    - $MC[i, k]$ = what $i$ knows about what $j$ knows about what happened at $k$.
  - If $A$ sends a message to $B$, then $MC_B$ is set to the pairwise maximum of $MC_A$ and $MC_B$.
    - If $A$ knows that $B$ knows $u$, then after $A$ talks to $C$, $C$ knows that $B$ knows $u$ too.