Time and Order in Distributed Systems

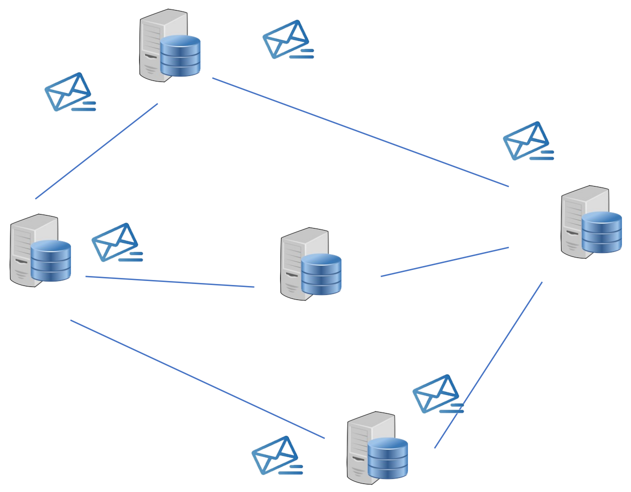
CS4405 – Analysis of Concurrent and Distributed Programs

Burcu Kulahcioglu Ozkan

TU Delft

Events in distributed systems

- Processes operate on their local memory and communicate by exchanging messages:
  - A process performs some local computation
  - A process sends a message
  - A process receives a message



Time and order of events in distributed systems

Why do we need to order the events?

- Encoding history ("happens before" relationships)
- Transactions in a database
- Consistency of distributed data
- Debugging (finding the root cause of a bug)
- . . .

Reminder: Partial vs Total order

Strict partial order:

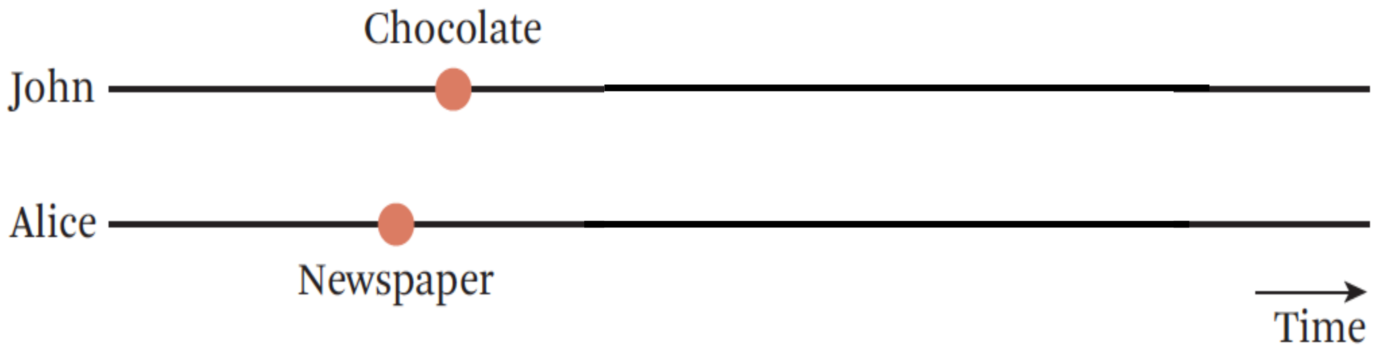
- **Irreflexivity**: $\forall a. \neg a < a$ (items not comparable with self)
- **Transitivity**: if $a \leq b$ and $b \leq c$ then $a \leq c$
- **Antisymmetry**: if $a \leq b$ and $b \leq a$ then $a = b$

Strict total order:

- An additional property: $\forall a, b, a \leq b \vee b \leq a \vee a = b$

Time in centralized vs distributed systems

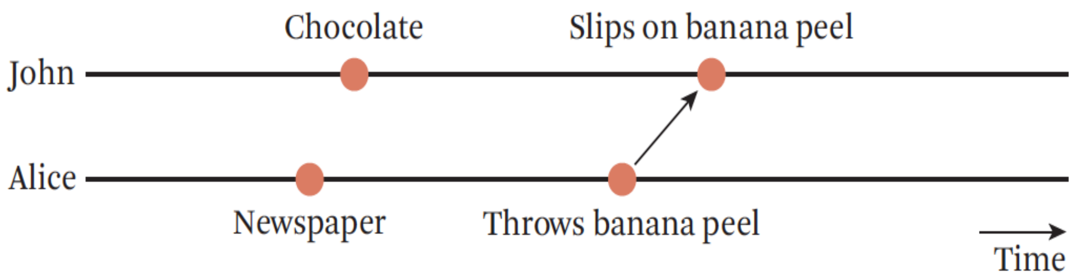
- Centralized systems: System calls to kernel, monotonically increasing time values.
- Distributed systems: Achieving agreement on time is not trivial!

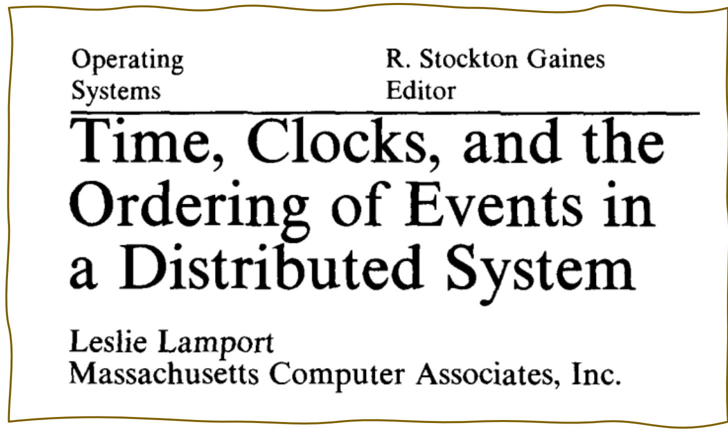


Logical time

- Idea: Instead of using the precise clock time, capture the events relationship between a pair of events

- Based on causality: If some event possibly causes another event, then the first event happened-before the other



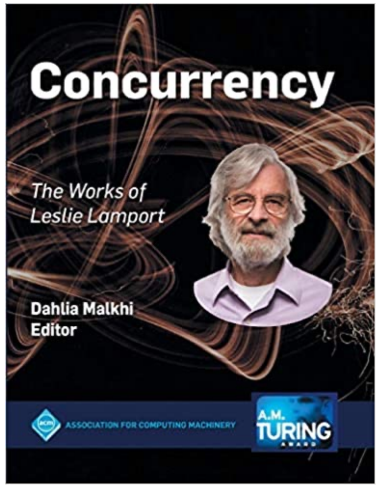

Operating Systems

R. Stockton Gaines Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

CACM, 1978



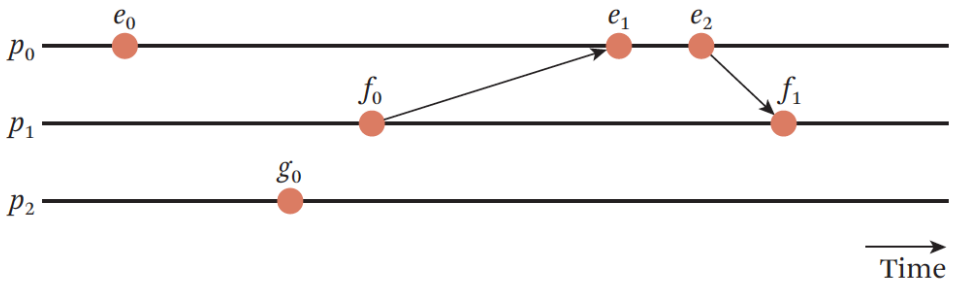
Happens-before relation between events

Happens-before relation captures dependencies between events:

- If a and b are events in the same node, and a occurs before b, then $a \to b$
- If a is the event of sending a message and b is the event of receiving that message, then $a \to b$
- The relation is transitive.

It is a strict partial order: it is irreflexive, antisymmetric and transitive.

Two events not related to happened-before are *concurrent*.

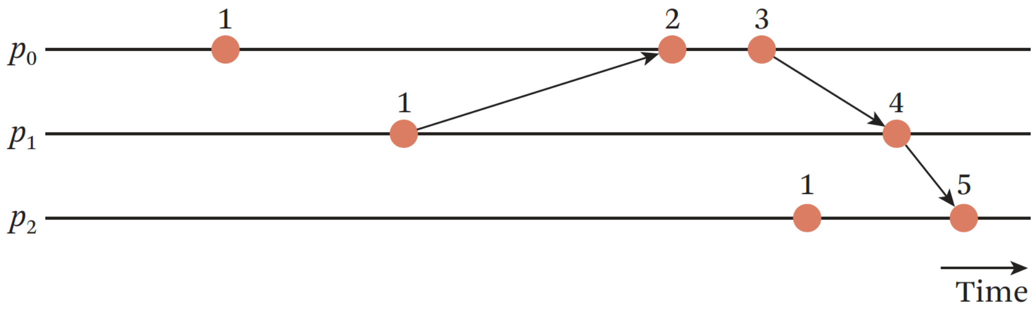


Lamport timestamps

Lamport introduced the eponymous logical timestamps in 1978:

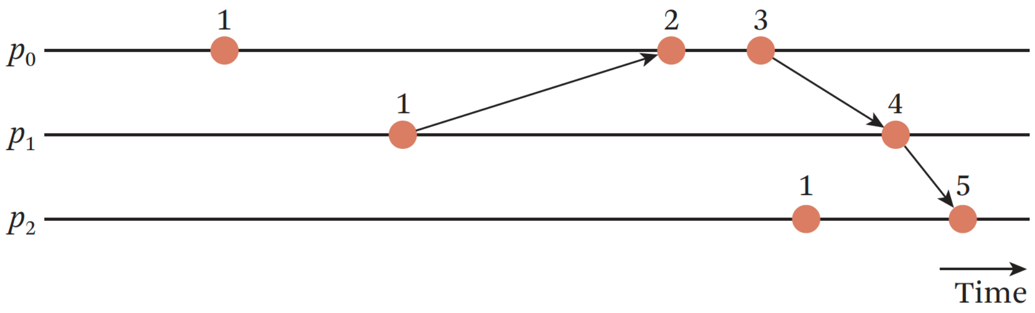
- Each individual process p maintains a counter: $LT(p)$.
- When a process p performs an action, it increments $LT(p)$.
- When a process p sends a message, it includes $LT(p)$ in the message.
- When a process p receives a message from a process q, that message includes the value of $LT(q)$; p updates its $LT(p)$ to the $\max(LT(p), LT(q)) + 1$

For two events a and b, if $a \rightarrow b$, then $LT(a) < LT(b)$.

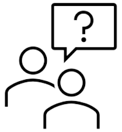


Lamport timestamps

For two events a and b, if $a \rightarrow b$, then $LT(a) < LT(b)$.



If $LT(a) < LT(b)$, then **it does not mean** that $a \rightarrow b$.



Why?

Why is the LT invariant not symmetric?

Another example scenario with 4 nodes that exchange events:

Initial state of timestamps: $[A(0), B(0), C(0), D(0)]$

E1. A sends to C: $[A(1), B(0), C(0), D(0)]$

E2. C receives from A: $[A(1), B(0), C(2), D(0)]$

E3. C sends to A: $[A(1), B(0), C(3), D(0)]$

E4. A receives from C: $[A(4), B(0), C(3), D(0)]$

E5. B sends to D: $[A(4), B(1), C(3), D(0)]$

E6. D receives from B: $[A(4), B(1), C(3), D(2)]$

At this point, $LT(E6) < LT(E4)$, but it does not mean that $E6 \rightarrow E4$!

Events 4 and 6 are independent.

Vector Clocks

Vector clocks can maintain causal order.

On a system with N nodes, each node i maintains a vector V_i of size N.

- $V_i[i]$ is the number of events that occurred at node i
- $V_i[j]$ is the number of events that node i knows occurred at node j

All nodes vector clocks start at [0, …, 0]

They are updated as follows:

- Local events increment $V_i[i]$
- When i sends a message to j, it includes V_i
- When j receives V_i, it updates all elements of V_j to $V_j[a] = \max(V_i[a], V_j[a])$

Vector clocks

Initial state of timestamps: $[A(0,0,0,0), B(0,0,0,0), C(0,0,0,0), D(0,0,0,0)]$

E1. A sends to C: $[A(1,0\ 0\ 0), B(0,0,0,0), C(0,0,0,0), D(0,0,0,0)]$

E2. C receives from A:$[A(1,0\ 0\ 0), B(0,0,0,0), C(1,0,1,0), D(0,0,0,0)]$

E3. C sends to A:$[A(1,0\ 0\ 0), B(0,0,0,0), C(1,0,2,0), D(0,0,0,0)]$

E4. A receives from C:$[A(2,0\ 2\ 0), B(0,0,0,0), C(1,0,2,0), D(0,0,0,0)]$

E5. B sends to D: $[A(2,0\ 2\ 0), B(0,1,0,0), C(1,0,2,0), D(0,0,0,0)]$

E6. D receives from B:$[A(2,0\ 2\ 0), B(0,1,0,0), C(1,0,2,0), D(0,1,0,1)]$

Vector clock guarantees

- Comparing vector clocks: Given V_i and V_j :
  - $V_i = V_j$ iff $V_i[k] = V_j[k]$ for all k
  - $V_i \leq V_j$ iff $V_i[k] \leq V_j[k]$ for all k
  - (Concurrency) $V_i \ || \ V_j$ otherwise

- For two events a and b and their vector clocks $VC(a)$ and $VC(\mathrm{b})$:
  - if $a \rightarrow b$, then $VC(a) < VC(b)$
  - if $VC(a) < VC(b)$, then $a \rightarrow b$

Vector clocks are expensive to maintain: they require $O(n)$ timestamps to be exchanged with each communication.

- However, we cannot do better than $O(n)$

Causally dependent events

Why compute causal dependency between events?