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

by L. Lamport

CS 5204 Operating Systems

Vladimir Glina

Fall 2005

Overview

- Key Points
- Background
- Partial Ordering
- Extension for Total Ordering
- Further Work
- Key Points Reiteration
- Evaluation
- Discussion

Key Points

1. The "happens before" relation on the system event set
2. The events partial ordering on the base of the relation
3. The distributed algorithm for logical clock synchronization
4. The algorithm extension to the case of total events ordering
5. The algorithm application for physical clock synchronization

Background: Distributed System Features

- Spatially separated processes
- Processes communicate through messages
- Message delays are considerable
- Absence of the single timer leads to synchronization problems
  - Example: totally ordered multicast

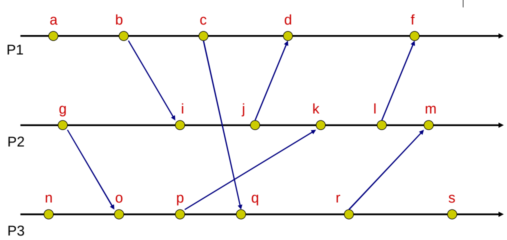
Background: Synchronization Approaches

- Physical Clock Adjustment
  - All clocks show the same actual time
  - Problems:
    - **Most important**: backward time flow possible
    - Sophisticated time services (i.e. WWV); or
    - Reliance on a human operator
- Logical Clock Adjustment
  - Consistency is important, not actual time

Partial Ordering: Basics

- A system is a set of processes P_i
- A process is a set of events *a, b, …* with total ordering
- "Happened before" ($\rightarrow$) relation:
  - $(a \in P)$ && $(b \in P)$ && (a comes before b) $\Rightarrow a \rightarrow b$
  - (P_1 sends a to P_2) && (b is the receipt of P_2 for a) $\Rightarrow a \rightarrow b$
  - $(a \rightarrow b)$ && $(b \rightarrow c) \Rightarrow a \rightarrow c$
- $!(a \rightarrow b)$ && $!(b \rightarrow a) \Rightarrow a$ and b are concurrent
- $!(a \rightarrow a)\ \forall\ a$, so "happened before" is an irreflexive partial ordering on the set of all the system events

Partial Ordering: Example



a → f b → s c → m d || s i || q k || r

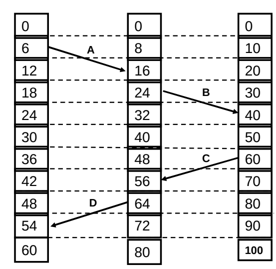
Partial Ordering: Synchronization

- Logical clock: C⟨*a*⟩ = C_j⟨*a*⟩ if *a* ∈ P_j
- ***Check condition:*** for ∀ *a, b*
  a → *b* ⇒ C⟨*a*⟩ < C⟨b⟩ **(not vice versa)**
- The check condition is satisfied if
  - **C1.** (*a, b* ∈ P_i) && (*a* comes before *b)*
    ⇒ C_i⟨*a*⟩ < C_i⟨*b*⟩
  - **C2.** (P_i sends *a* to P_j) && (*b* is the receipt of P_j *to a)*
    ⇒ C_i⟨*a*⟩ < C_j⟨*b*⟩
- **C never decreases!**

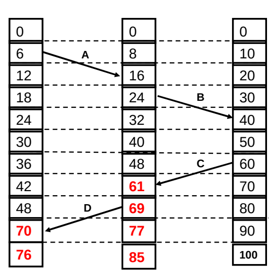
Partial Ordering: Implementation Rules

- **IR1.** Each P_i increments C_i between any two successive events.
- **IR2.**
  a) If *a* is the sending of a message *m* by P_i, then m contains a timestamp $T_m = C_j\langle a\rangle$; **and**
  b) Upon receiving *m*, P_i sets C_j greater than or equal to its present value and greater than T_m

Partial Ordering: Unregulated Clocks



Partial Ordering: Corrected Clocks



Total Ordering: Definition

- ‹ is an **arbitrary** total ordering of processes
- "Happen before" for total ordering(⟹):
  (*a* ε P_i) && (*b* ε P_j) ⇒ a ⟹ b iff
  - C_i⟨*a*⟩ < C_j⟨ *b*⟩, ***or***
  - P_i ‹ P_j
- The total ordering depends on C_i and is not unique

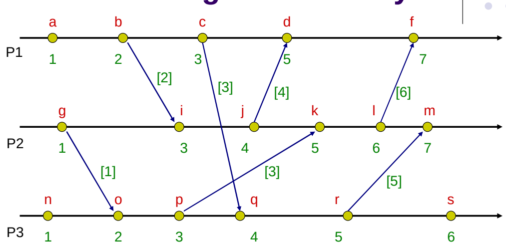
Total Ordering: Synchronization

1. P_i broadcasts the message T_m:P_i (request resource) and puts it on its request queue.
2. When P_j receives T_m:P_i, it puts the message on its request queue and sends the acknowledgment to P_i.
3. To release the resource, P_i removes T_m:P_i from its queue, broadcasts a timestamped release message.
4. When P_j receives the release message, it removes T_m:P_i from its queue.
5. P_i is granted the resource when
   1) It has T_m:P_i in its queue ordered before any other request in the queue by the relation $\Longrightarrow$; and
   2) P_i has received a message from every other process timestamped later than T_m.

Further Work: Vector Timestamps

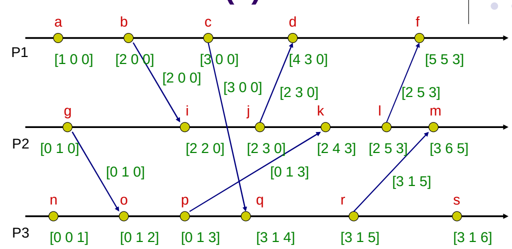
- Lamport clock is:
  - Consistent: $a \rightarrow b \;\Rightarrow C\langle a\rangle < C\langle b\rangle$
  - **Not**: $C\langle a\rangle < C\langle b\rangle \Leftrightarrow a \rightarrow b$ (not strongly consistent)
- Vector timestamps (VT) are strongly consistent
- VT address **potential** causality
  - Allow to say if ***a* happened** before *b*, but not if ***a* caused** *b*
- VT say how many events have occurred so far at all processes
- VT solve the totally-ordered multicasting problem

Lack of Strong Consistency



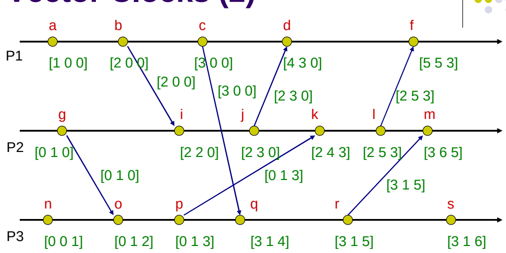
d || s q || i k || r
5 < 6 4 > 3 5 = 5

Vector Clocks (1)



$a \rightarrow f$ $b \rightarrow s$ $c \rightarrow m$
[1 0 0] < [5 5 3] [2 0 0] < [3 1 6] [3 0 0] < [3 6 5]

Vector Clocks (2)



d || s q || i k || r
[4 3 0] < [3 1 6] [3 1 4] < [2 2 0] [2 4 3] < [3 1 5]

Key Points Reiteration

1. The "happens before" relation on the system event set
2. The events partial ordering on the base of the relation
3. The distributed algorithm for logical clock synchronization
4. The algorithm extension to a case of total events ordering
5. The algorithm application for physical clock synchronization

Evaluation

- The logical clocks idea is very appealing
- Virtually no revision on previous work
- Nice to have more mathematically strict extension on total ordering, if possible

Discussion

Thank you!

Any questions?