

# Logical Clocks and Causal Ordering

**CS60002: Distributed Systems**



**Pallab Dasgupta**

Dept. of Computer Sc. & Engg.,

Indian Institute of Technology Kharagpur

# Why do we need global clocks?

- For causally ordering events in a distributed system
  - **Example:**
    - Transaction T transfers Rs 10,000 from S1 to S2
    - Consider the situation when:
      - State of S1 is recorded after the deduction and state of S2 is recorded before the addition
      - State of S1 is recorded before the deduction and state of S2 is recorded after the addition
- Should not be confused with the clock-synchronization problem



What data is being transmitted? 0101?



Yes, if this is the clock



If this is the clock, then 01110001

*The receiver needs to know the clock of the sender*

# Ordering of Events

---

## Lamport's *Happened Before* relationship:

For two events  $a$  and  $b$ ,  $a \rightarrow b$  if

- *$a$  and  $b$  are events in the same process and  $a$  occurred before  $b$ , or*
- *$a$  is a send event of a message  $m$  and  $b$  is the corresponding receive event at the destination process, or*
- *$a \rightarrow c$  and  $c \rightarrow b$  for some event  $c$*

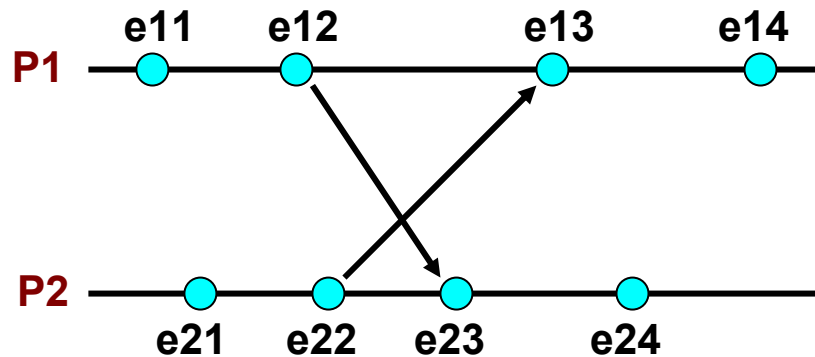
# Causally Related versus Concurrent

## Causally related events:

Event a causally affects event b if  $a \rightarrow b$

## Concurrent events:

- Two distinct events a and b are said to be concurrent ( denoted by  $a||b$  ) if  $a \not\rightarrow b$  and  $b \not\rightarrow a$



e11 and e21 are concurrent

e14 and e23 are concurrent

e22 causally affects e14

A space-time diagram

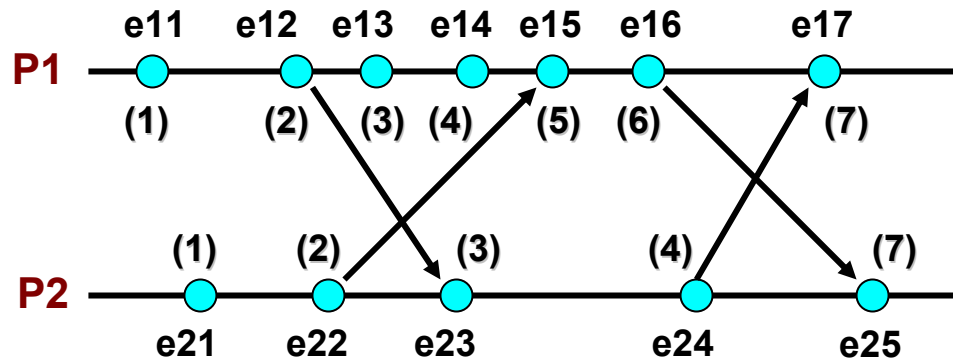
# Lamport's Logical Clock

*Each process  $i$  keeps a clock  $C_i$*

- Each event  $a$  in  $i$  is time-stamped  $C_i(a)$ , the value of  $C_i$  when  $a$  occurred
- $C_i$  is incremented by 1 for each event in  $i$
- In addition, if  $a$  is a send of message  $m$  from process  $i$  to  $j$ , then on receive of  $m$ ,

$$C_j = \max (C_j, C_i(a)+1)$$

# How Lamport's clocks advance



# Points to note

---

- if  $a \rightarrow b$ , then  $C(a) < C(b)$
- $\rightarrow$  is a partial order
- Total ordering possible by arbitrarily ordering concurrent events by process numbers

# Limitation of Lamport's Clock

---

$a \rightarrow b$  implies  $C(a) < C(b)$

**BUT**

$C(a) < C(b)$  doesn't imply  $a \rightarrow b$  !!

*So not a true clock !!*



# Solution: *Vector Clocks*

Each process  $P_i$  has a clock  $C_i$ , which is a vector of size  $n$   
The clock  $C_i$  assigns a vector  $C_i(a)$  to any event  $a$  at  $P_i$

## Update rules:

- $C_i[i]++$  for every event at process  $i$
- If  $a$  is send of message  $m$  from  $i$  to  $j$  with vector timestamp  $t_m$ , then on receipt of  $m$ :  
$$C_j[k] = \max(C_j[k], t_m[k]) \text{ for all } k$$

# Partial Order between Timestamps

For events  $a$  and  $b$  with vector timestamps  $t^a$  and  $t^b$ ,

- Equal:  $t^a = t^b$  iff  $\forall i, t^a[i] = t^b[i]$
- Not Equal:  $t^a \neq t^b$  iff  $\exists i, t^a[i] \neq t^b[i]$
- Less or equal:  $t^a \leq t^b$  iff  $\forall i, t^a[i] \leq t^b[i]$
- Not less or equal:  $t^a \not\leq t^b$  iff  $\exists i, t^a[i] > t^b[i]$
- Less than:  $t^a < t^b$  iff  $(t^a \leq t^b \text{ and } t^a \neq t^b)$
- Not less than:  $t^a \not< t^b$  iff  $\neg(t^a \leq t^b \text{ and } t^a \neq t^b)$
- Concurrent:  $t^a \parallel t^b$  iff  $(t^a \not< t^b \text{ and } t^b \not< t^a)$

# Causal Ordering

---

- $a \rightarrow b$  iff  $t_a < t_b$
- Events  $a$  and  $b$  are causally related iff  $t_a < t_b$  or  $t_b < t_a$ , else they are concurrent
- Note that this is still not a total order

# Use of Vector Clocks in Causal Ordering of Messages

---

- If  $\text{send}(m1) \rightarrow \text{send}(m2)$ , then every recipient of both message  $m1$  and  $m2$  must “deliver”  $m1$  before  $m2$ .
  - “deliver” – when the message is actually given to the application for processing

# Birman-Schiper-Stephenson Protocol

- To broadcast  $m$  from process  $i$ , increment  $C_i(i)$ , and timestamp  $m$  with  $VT_m = C_i[i]$
- When  $j \neq i$  receives  $m$ ,  $j$  delays delivery of  $m$  until
  - $C_j[i] = VT_m[i] - 1$  and
  - $C_j[k] \geq VT_m[k]$  for all  $k \neq i$
  - Delayed messages are queued in  $j$  sorted by vector time. Concurrent messages are sorted by receive time.
- When  $m$  is delivered at  $j$ ,  $C_j$  is updated according to vector clock rule.

# Problem of Vector Clock

---

- **Message size increases since each message needs to be tagged with the vector**
- **Size can be reduced in some cases by only sending values that have changed**

# Global State Recording

## CS60002: Distributed Systems



Pallab Dasgupta

Dept. of Computer Sc. & Engg.,

Indian Institute of Technology Kharagpur

# Global State Collection

---

- **Applications:**
  - **Checking “stable” properties, checkpoint & recovery**
- **Issues:**
  - **Need to capture both node and channel states**
  - **system cannot be stopped**
  - **no global clock**



# Notations

---

## Some notations:

- $LS_i$ : Local state of process  $i$
- $\text{send}(m_{ij})$  : Send event of message  $m_{ij}$  from process  $i$  to process  $j$
- $\text{rec}(m_{ij})$  : Similar, receive instead of send
- $\text{time}(x)$  : Time at which state  $x$  was recorded
- $\text{time}(\text{send}(m))$  : Time at which  $\text{send}(m)$  occurred

# Definitions

- $\text{send}(m_{ij}) \in \text{LS}_i$       iff  $\text{time}(\text{send}(m_{ij})) < \text{time}(\text{LS}_i)$
- $\text{rec}(m_{ij}) \in \text{LS}_j$       iff  $\text{time}(\text{rec}(m_{ij})) < \text{time}(\text{LS}_j)$
- $\text{transit}(\text{LS}_i, \text{LS}_j)$   
=  $\{ m_{ij} \mid \text{send}(m_{ij}) \in \text{LS}_i \text{ and } \text{rec}(m_{ij}) \notin \text{LS}_j \}$
- $\text{inconsistent}(\text{LS}_i, \text{LS}_j)$   
=  $\{ m_{ij} \mid \text{send}(m_{ij}) \notin \text{LS}_i \text{ and } \text{rec}(m_{ij}) \in \text{LS}_j \}$

# Definitions

- **Global state: collection of local states**

$$GS = \{LS1, LS2, \dots, LS_n\}$$

- **GS is consistent iff**

for all  $i, j, 1 \leq i, j \leq n,$

$$\text{inconsistent}(LS_i, LS_j) = \Phi$$

- **GS is transitless iff**

for all  $i, j, 1 \leq i, j \leq n,$

$$\text{transit}(LS_i, LS_j) = \Phi$$

- **GS is strongly consistent if it is consistent and transitless.**

# Chandy-Lamport's Algorithm

---

- Uses special marker messages.
- One process acts as initiator, starts the state collection by following the marker sending rule below.
- Marker sending rule for process P:
  - P records its state and
  - For each outgoing channel C from P on which a marker has not been sent already, P sends a marker along C before any further message is sent on C

# Chandy Lamport's Algorithm contd..

---

- **When Q receives a marker along a channel C:**
  - **If Q has not recorded its state then Q records the state of C as empty; Q then follows the marker sending rule**
  - **If Q has already recorded its state, it records the state of C as the sequence of messages received along C after Q's state was recorded and before Q received the marker along C**

## Notable Points

---

- **Markers sent on a channel distinguish messages sent on the channel before the sender recorded its states and the messages sent after the sender recorded its state**
- **The state collected may not be any state that actually happened in reality, rather a state that “could have” happened**
- **Requires FIFO channels**
- **Message complexity  $O(|E|)$ , where  $E$  = no. of links**

# Termination Detection

**CS60002: Distributed Systems**



**Pallab Dasgupta**

Dept. of Computer Sc. & Engg.,

Indian Institute of Technology Kharagpur

# Termination Detection

- **Model**

- processes can be active or idle
- only active processes send messages
- idle process can become active on receiving a computation message
- active process can become idle at any time
  
- **Termination: all processes are idle and no computation message are in transit**
- **Can use global snapshot to detect termination also**



# Huang's Algorithm

- One controlling agent, has weight 1 initially
- All other processes are idle initially and has weight 0
- Computation starts when controlling agent sends a computation message to a process
- An idle process becomes active on receiving a computation message
- $B(DW)$  – computation message with weight  $DW$ . Can be sent only by the controlling agent or an active process
- $C(DW)$  – control message with weight  $DW$ , sent by active processes to controlling agent when they are about to become idle

# Weight Distribution and Recovery

- Let current weight at process =  $W$
- Send of B(DW):
  - Find  $W_1, W_2$  such that  $W_1 > 0, W_2 > 0, W_1 + W_2 = W$
  - Set  $W = W_1$  and send B( $W_2$ )
- Receive of B(DW):
  - $W = W + DW$ ;
  - if idle, become active
- Send of C(DW):
  - send C( $W$ ) to controlling agent
  - Become idle
- Receive of C(DW):
  - $W = W + DW$
  - if  $W = 1$ , declare “termination”