

Synchronization

Lamport Logical Clock and Clock Vectors

Pedro F. Souto (`pfs@fe.up.pt`)

May 5, 2021

Roadmap

The Happened-Before Relation

Lamport Clocks and Timestamps

Vector Clocks and Timestamps

Roadmap

The Happened-Before Relation

Lamport Clocks and Timestamps

Vector Clocks and Timestamps

Events

- ▶ Often, there is no need for synchronized clocks
 - ▶ Sometimes, we just need to order **events**

Event is an action that takes place in the execution of an algorithm

Sending of a message

Delivery of a message

Computation step

Execution of a distributed algorithm by process i can be defined as a sequence of events:

$$e_i^0 e_i^1 e_i^2 \dots$$

- ▶ Each event in an execution is an instance of a given **event type**

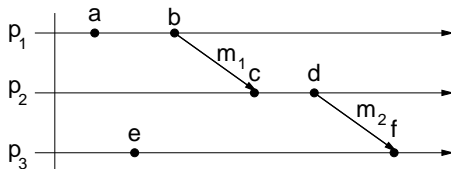
The Happened-Before Relation (\rightarrow)

HB1 If e and e' are events that happen in that order in the same process, then $e \rightarrow e'$

HB2 If e is the sending of a message m and e' is the reception of that message, then $e \rightarrow e'$

HB3 If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

- That is, the HB relation is transitive



- The HB relation is a partial order. For example:

$$a \not\rightarrow e \wedge e \not\rightarrow a$$

- Events like a and e are **concurrent**: $a || e$

- The HB relation captures the **potential causality** between events

Roadmap

The Happened-Before Relation

Lamport Clocks and Timestamps

Vector Clocks and Timestamps

Lamport Clocks and Timestamps

Lamport clock is a logical clock used to assign (Lamport) timestamps to events

- Each process in the system has its own Lamport clock L_i

(Lamport) clock condition For any events e and e' :

$$e \rightarrow e' \Rightarrow L(e) < L(e')$$

Or equivalently:

$$L(e) \geq L(e') \Rightarrow e \not\rightarrow e'$$

Satisfying the Clock Condition

C1 If e and e' are events in process p_i and occur in that order, then $L_i(e) < L_i(e')$

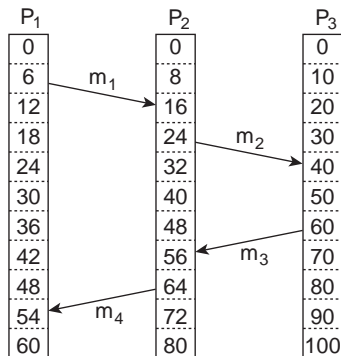
C2 If

e is the sending of a message by process p_i , and

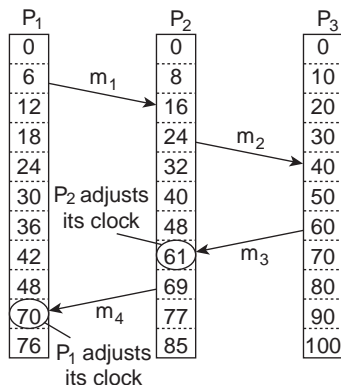
e' is the receipt of that message by a process p_j

then $L_i(e) < L_j(e')$

(Physical Clocks vs. Lamport Clocks)



(a)



(b)

- Free-running physical clocks cannot be used as Lamport clocks
 - If the clock resolution is small enough C1 is easy to ensure
 - The problem is with C2

Lamport Clocks and Timestamps (2/3)

- ▶ The timestamp of an event at process i is assigned by L_i , the Lamport clock at the process i

Lamport Clock Update Rules To satisfy the Clock Condition a Lamport clock must be updated **before** assigning its value to the event as follows:

LC1 if e is not the receiving of a message, just increment L_i

LC2 if e is the receiving of a message m :

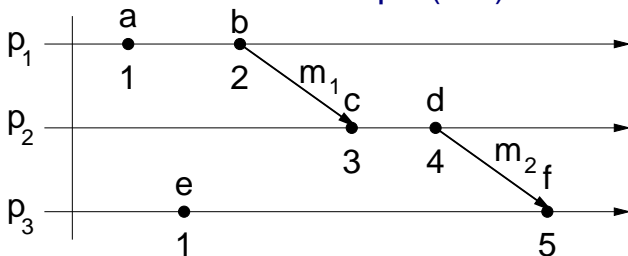
$$L_i = \max(L_i, TS(m)) + 1$$

where $TS(m)$ is the timestamp of the corresponding sending event

- ▶ Implementing LC2 requires piggybacking the timestamp of the sending event on every message

IMP. Incrementing a Lamport clock is **not** an event

Lamport Clocks and Timestamps (3/3)



- ▶ If we need to order all events we can use the pair (**extended Lamport timestamp**):

$$(L(e), i)$$

where i is the process where e happens

- ▶ How would you define that order, so that it "extends" the HB relation?
- ▶ Although total, this order is somewhat arbitrary
- ▶ Actually, Lamport claims that the reason for Lamport clocks is precisely to obtain a total order.

State Machine

- Indeed, a total order allows to solve any synchronization problem

Idea Specify the synchronization in terms of a state machine:

Set of commands, C

Set of states, S

State transition function, $t : C \times S \rightarrow S'$

- I.e., the execution of a command, c , changes the current state s to a new state s' , formally: $t(c, s) = s'$

Synchronization is achieved, if all processes execute the same set of commands in the same order

A process can execute a command timestamped T when it has learned of all commands issued by all other processes with timestamps less than or equal to T . The precise algorithm is straightforward, and we will not bother to describe it.

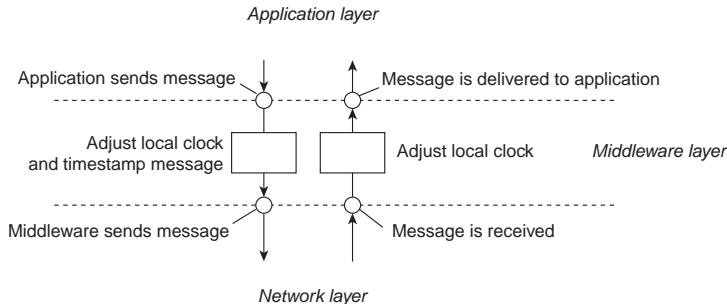
The problem is that the failure of a single process blocks the system

Use of Lamport Clocks: Total Order Multicast (1/2)

Problem How to ensure that if m and m' are delivered by processes i and j , then they deliver m and m' in the same order?

Solution Use **extended** Lamport timestamps to timestamp messages, and **deliver** the messages in the order of these timestamps

Deliver vs. Receive this is similar to what happens with TCP to ensure order in point-to-point communication



Use of Lamport Clocks: Total Order Multicast (2/2)

Assumptions The communication channel is:

Reliable

FIFO

- ▶ Each process keeps its own Lamport clock
- ▶ The only relevant events are the sending and receiving of multicast messages
- ▶ Just before multicasting a message, the LC is incremented and its value used to timestamp the message
- ▶ Upon receiving a message m , a process:
 1. Inserts the message in a queue of messages ordered by their extended Lamport timestamps;
 2. Updates the Lamport clock to $LC = \max(TS(m), LC)$
- ▶ A message is delivered to the application only when:
 - ▶ It is at the head of the queue
 - ▶ It is **stable**
 - ▶ If there is a message on the queue from every other process
- ▶ To reduce the delivery delay, can use acknowledgments

Roadmap

The Happened-Before Relation

Lamport Clocks and Timestamps

Vector Clocks and Timestamps

Vector Clocks (1/3)

Observation The main limitation of Lamport clocks is that:

$$L(e) < L(e') \not\Rightarrow e \rightarrow e'$$

Idea Use an array of timestamps, one per processor

- ▶ Due to Mattern, Fidge and Schmuck

Rules Each process p_i keeps its own vector V_i , which it updates as follows:

VC1 if e is not the receiving of a message, just increment $V_i[i]$

VC2 if e is the receiving of a message m :

$$V_i[i] = V_i[i] + 1$$

$$V_i[j] = \max(V_i[j], TS(m)[j])$$

- ▶ **Initially** $V_i[j] = 0$, for all j
- ▶ The timestamp of the sending event is piggybacked on the corresponding message ($TS(m) = V(send(m))$)

Basically

$V_i[i]$ is the number of events timestamped by p_i

$V_i[j]$ is the number of events in p_j that p_i **knows** about

Vector Clocks (2/3)

Let V and V' be vector timestamps

Vector Timestamps Comparison We define the relation $<$ between vector timestamps:

$$V < V' \text{ iff } \forall_j : V[j] \leq V'[j] \wedge \exists_i : V[i] < V'[i]$$

Vector Clock Condition

$$\begin{aligned} e \rightarrow e' &\Rightarrow V(e) < V(e') \\ V(e) < V(e') &\Rightarrow e \rightarrow e' \end{aligned}$$

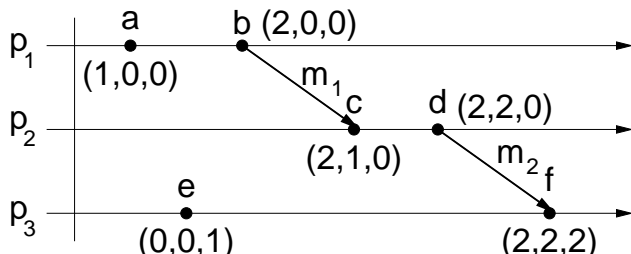
On the other hand:

$$\neg(V(e) < V(e')) \wedge \neg(V(e') < V(e)) \Rightarrow e \parallel e'$$

Conclusion Vector timestamps can be used to determine whether the HB relation holds between any two pairs of events

- Lamport clocks allow to conclude **only** if the HB does **not** hold

Vector Clocks (3/3)



- ▶ a and e are concurrent events
- ▶ The main issue with vector clocks is that we need n *timestamps* per event, whereas *Lamport clocks* need only one
 - ▶ But there is no way to avoid it (Charron-Bost).
- ▶ **Hidden** communication channels can lead to **anomalous** behavior
 - ▶ I.e. to the violation of the Clock Condition for both Lamport clocks and vector clocks.
 - ▶ Lamport claims that only synchronized (physical) clocks may eliminate such anomalies

Vector Clocks Use: Causal Order Multicast (1/2)

Problem How to ensure that if $m \rightarrow m'$ and process i delivers m' , then it must have previously delivered m

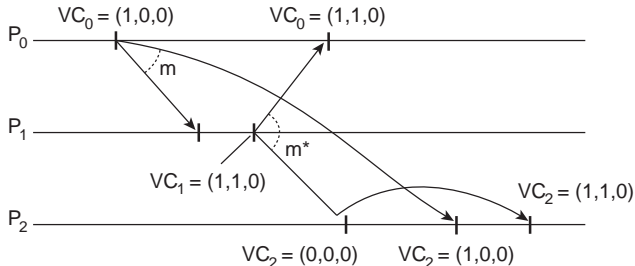
Solution Use vector clocks to timestamp messages, and **deliver** the messages in the order of these timestamps (as defined above)

- ▶ Each process keeps its own vector clock
- ▶ The only relevant events are the sending and receiving of multicast messages
- ▶ Just before multicasting a message, the sender updates its VC after which it timestamps the message
- ▶ Upon receiving a message m , a receiver inserts the message in the queue of received messages
- ▶ Process i delivers message m to the application only when:

$$V_i[j] \geq TS(m)[j], \forall j \neq k, \text{ where } k \text{ is the sender of } m$$
$$V_i[k] = TS(m)[k] - 1$$

After which it should update its VC (no need to increment $V_i[i]$)

Vector Clocks Use: Causal Order Multicast (2/2)



Observations

- ▶ $VC_i[i]$, counts the number of messages multicast by p_i
- ▶ $VC_i[j]$, $i \neq j$, counts the number of messages multicast by p_j that p_i has delivered to the application
- ▶ The communication channels need to be reliable, but not FIFO. Why?

Question does the total order multicast protocol (with Lamport timestamps) also ensure causal order?

Further Reading

- ▶ Tanenbaum and van Steen, Section 6.2 of *Distributed Systems*, 2nd Ed.
- ▶ Leslie Lamport, *Time, Clocks and the Ordering of Events in a Distributed System*, Communications of the ACM 21(7): 558-565 (1978)
- ▶ F. Mattern, "Virtual Time and Global States of Distributed Systems", in Proc. Workshop on Parallel and Distributed Algorithms, Elsevier, pp. 215-226.
- ▶ C. Baquero and N. Preguiça, "Why logical clocks are easy", Communications of the ACM 59(4): 43-47 (2016)