Theoretical Foundations of Distributed Operating Systems

Neeraj Mittal

The University of Texas at Dallas

June 1, 2012

Outline

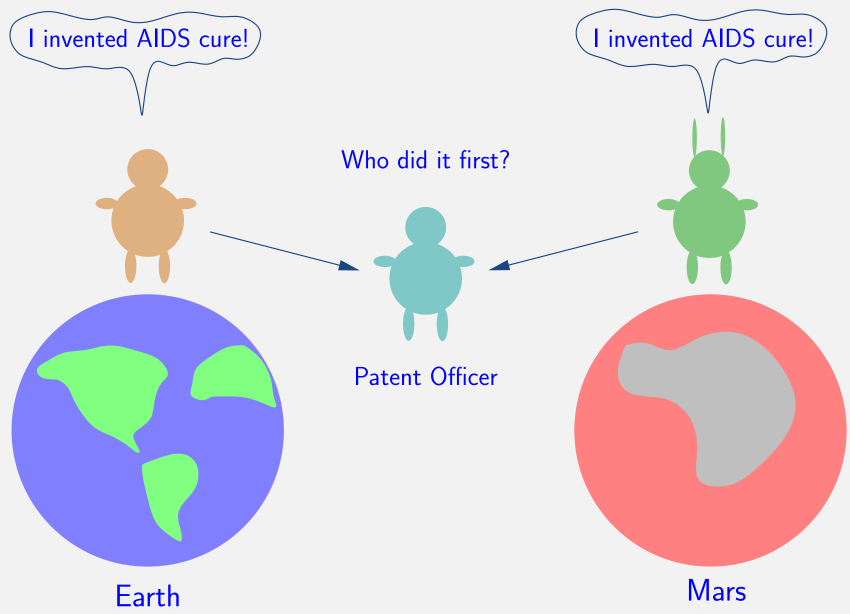
- Overview
- Ordering Events
- Abstract Clocks
- Ordering of Messages
- State of a Distributed System
- Monitoring a Distributed System

Two Important Characteristics

- Absence of Global Clock
  - there is no common notion of time
- Absence of Shared Memory
  - no process has up-to-date knowledge about the system

Absence of Global Clock

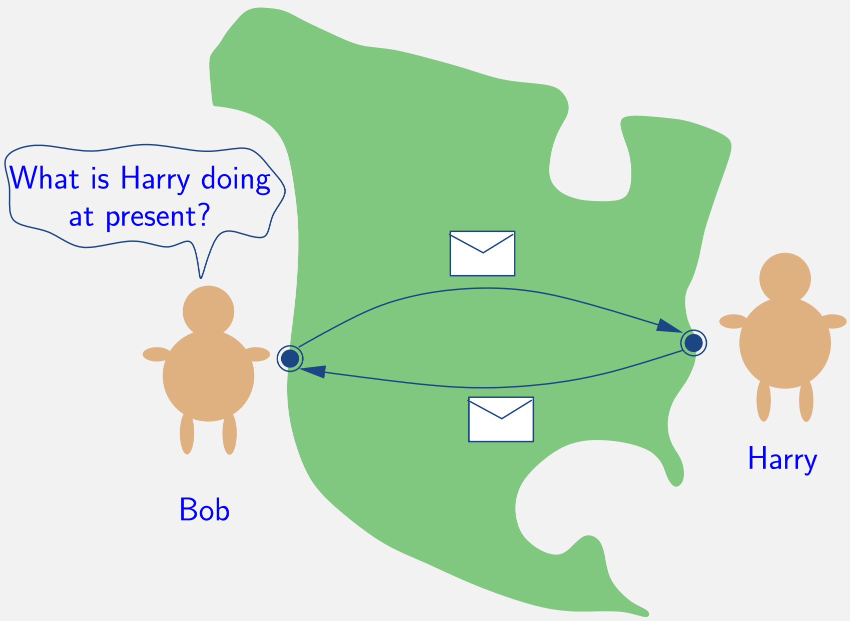
- Different processes may have different notions of time



Problem: How do we order events on different processes?

Absence of Shared Memory

- A process does not know current state of other processes



Problem: How do we obtain a coherent view of the system?

When is it possible to order two events?

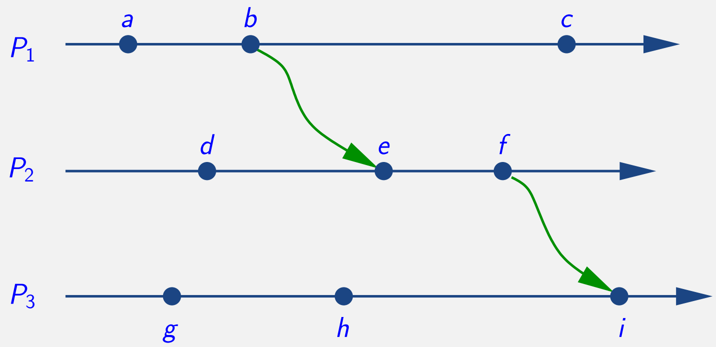
- Three cases:

  1. Events executed on the same process:
     - if e and f are events on the same process and e occurred before f, then e *happened-before* f
  2. Communication events of the same message:
     - if e is the send event of a message and f is the receive event of the same message, then e *happened-before* f
  3. Events related by transitivity:
     - if event e happened-before event g and event g happened-before event f, then e *happened-before* f

Outline

Happened-Before Relation

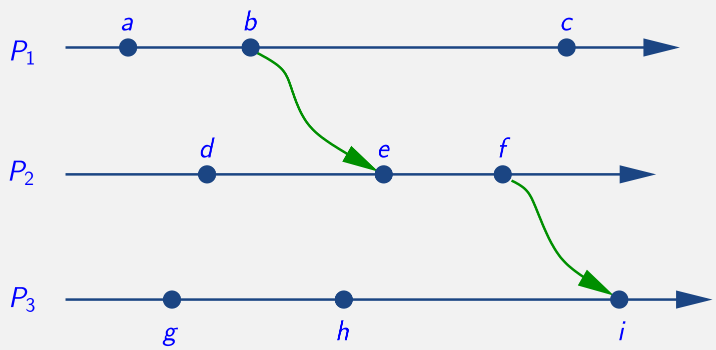
- Happened-before relation is denoted by $\rightarrow$
- Illustration:



  - Events on the same process:
    examples: $a \rightarrow b$, $a \rightarrow c$, $d \rightarrow f$
  - Events of the same message:
    examples: $b \rightarrow e$, $f \rightarrow i$
  - Transitivity:
    examples: $a \rightarrow e$, $a \rightarrow i$, $e \rightarrow i$

Concurrent Events

- Events not related by happened-before relation
- Concurrency relation is denoted by $\parallel$
- Illustration:



  - Examples: $a \parallel d$, $d \parallel h$, $c \parallel e$
- Concurrency relation is not transitive:
  example: $a \parallel d$ and $d \parallel c$ but $a \not\parallel c$

Outline

Different Kinds of Clocks

- Logical Clocks
  - used to totally order all events
- Vector Clocks
  - used to track happened-before relation
- Matrix Clocks
  - used to track what other processes know about other processes
- Direct Dependency Clocks
  - used to track direct causal dependencies

Logical Clock

- Implements the notion of virtual time
- Can be used to totally order all events
- Assigns timestamp to each event in a way that is consistent with the happened-before relation:

$$e \to f \implies C(e) < C(f)$$

$C(e)$: timestamp for event e
$C(f)$: timestamp for event f

Implementing Logical Clock

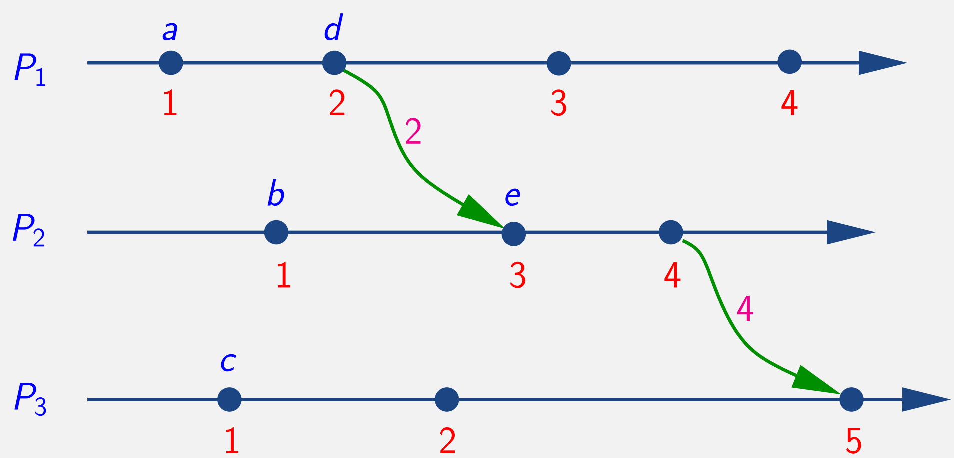
- Each process has a local scalar clock, initialized to zero
  - C_i denotes the local clock of process P_i
- Action depends on the type of the event
- Protocol for process P_i:
  - On executing an interval event:

    $C_i := C_i + 1$
  - On sending a message m:

    $C_i := C_i + 1$
    piggyback C_i on m
  - On receiving a message m:

    let t_m be the timestamp piggybacked on m
    $C_i := \max\{C_i, t_m\} + 1$

Implementing Logical Clock: An Illustration



Limitation of Logical Clock

- Logical clock
  cannot be used to determine whether two events are concurrent
  $$e \parallel f \text{ does not imply } C(e) = C(f)$$

when equal
concurrent
but when concurrent
not equal

Vector Clock

- Captures the happened-before relation
- Assigns timestamp to each event such that:

$$e \to f \iff C(e) < C(f)$$

$C(e)$: timestamp for event e
$C(f)$: timestamp for event f

Comparing Two Vectors

- Vectors are compared component-wise:
  - Equality:
$$V = V' \quad \text{iff} \quad \langle \forall i : V[i] = V'[i] \rangle$$
  - Less Than:
$$V < V' \quad \text{iff} \quad \langle \forall i : V[i] \leq V'[i] \rangle \wedge \langle \exists i : V[i] < V'[i] \rangle$$

- Example:
$$\begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} < \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix} \text{ and } \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} < \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} \text{ but } \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \not< \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}$$

Implementing Vector Clock

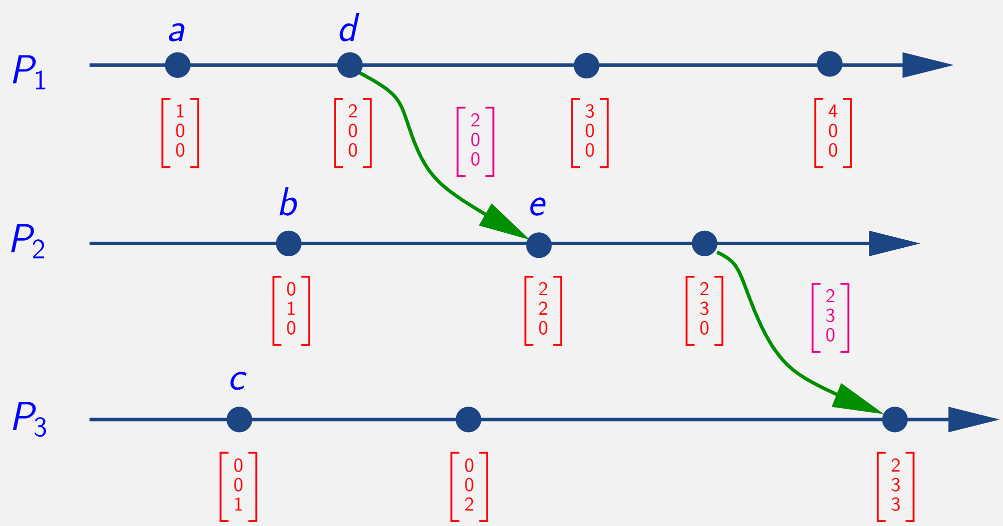
- Each process has a local vector clock
  - C_i denotes the local clock of process P_i
- Action depends on the type of the event
- Protocol for process P_i:
  - On executing an interval event:

    $C_i[i] := C_i[i] + 1$
  - On sending a message m:

    $C_i[i] := C_i[i] + 1$
    piggyback C_i on m
  - On receiving a message m:

    let t_m be the timestamp piggybacked on m
    $\forall k \quad C_i[k] := \max\{C_i[k], t_m[k]\}$
    $C_i[i] := C_i[i] + 1$

Implementing Vector Clock: An Illustration



Properties of Vector Clock

- How many comparisons are needed to determine whether an event e happened-before another event f?
  - As many as N integers may need to be compared in the worst case, where N is the number of processes
  - Can we do better?
  - Suppose we know the processes on which events e and f occurred (say, P_i and P_j respectively)

$$e \rightarrow f$$

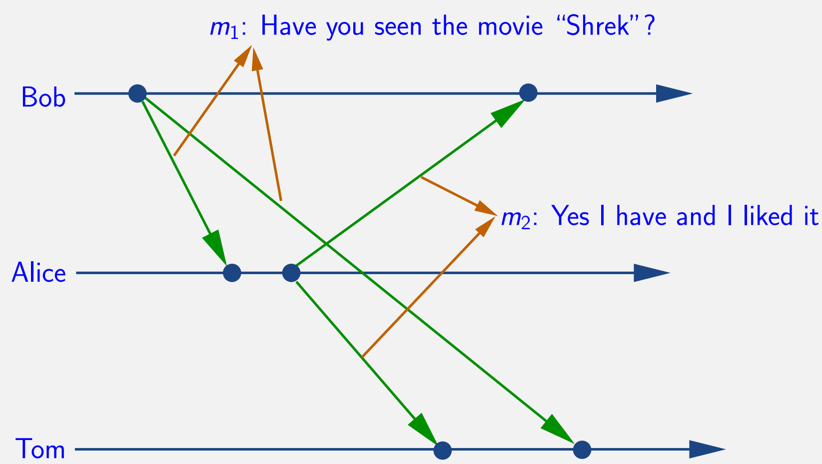
if and only if

$$(i = j) \wedge (C(e)[i] < C(f)[i]) \bigvee (i \neq j) \wedge (C(e)[i] \leq C(f)[i])$$

Outline

Ordering of Messages

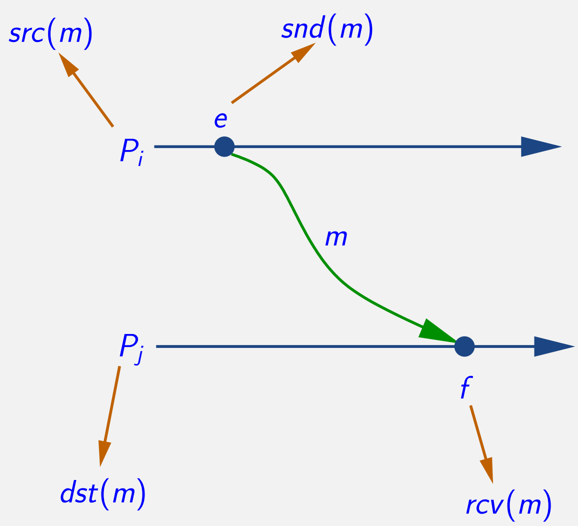
- For many applications, messages should be delivered in certain order to be interpreted meaningfully
- Example:



  - m_2 cannot be interpreted until m_1 has been received
  - Tom receives m_2 before m_1: an undesriable behavior

Useful Notations

- For a message m:
  - $src(m)$: source process of m
  - $dst(m)$: destination process of m
  - $snd(m)$: send event of m
  - $rcv(m)$: receive event of m



Causal Delivery of Messages

- A message w causally precedes a message m if $snd(w) \rightarrow snd(m)$
- An execution of a distributed system is said to be causally ordered if the following holds for every message m:

  every message that *causally precedes* m and is *destined for the same process* as m is *delivered before* m

  Mathematically, for every message w:

$$(snd(w) \rightarrow snd(m)) \wedge (dst(w) = dst(m)) \\ \Longrightarrow \\ rcv(w) \rightarrow rcv(m)$$

for FIFO source also should be same

A Causally Ordered Delivery Protocol

- Proposed by Birman, Schiper and Stephenson (BSS)
- Assumption:
  - communication is broadcast based: a process sends a message to every other process
- Each process maintains a vector with one entry for each process:
  - let V_i denote the vector for process P_i
  - the j^{th} entry of V_i refers to the number of messages that have been broadcast by process P_j that P_i knows of

The BSS Protocol

- Protocol for process P_i:
  - On broadcasting a message m:

    piggyback V_i on m
    $V_i[i] := V_i[i] + 1$

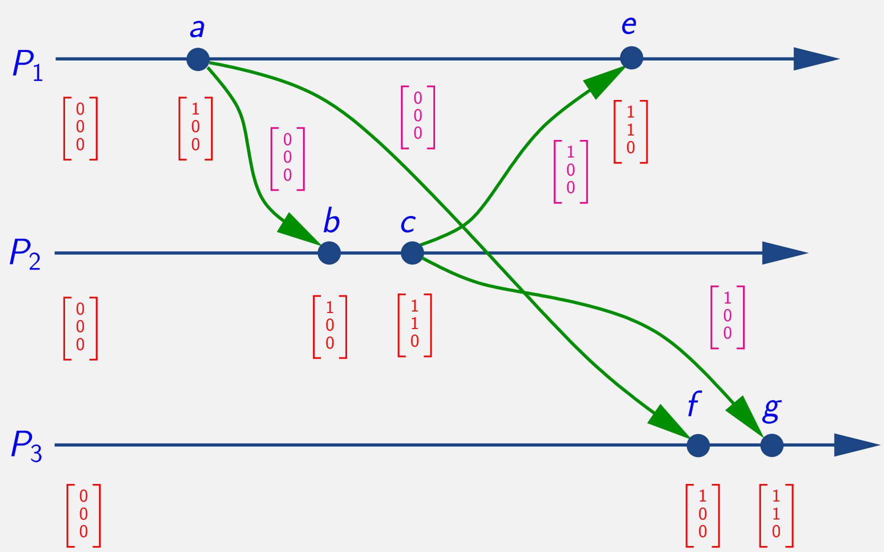
  - On arrival of a message m from process P_j:

    let V_m be the vector piggybacked on m
    deliver m once $V_i \geq V_m$

  - On delivery of a message m sent by process P_j:

    $V_i[j] := V_i[j] + 1$

The BSS Protocol: An Illustration



Another Causally Ordered Delivery Protocol

- Proposed by Raynal, Schiper and Toueg (RST)
- Assumption:
  - communication is unicast based: a process sends a message to only one other process
- Each process P_i maintains a matrix M_i of size $N \times N$
  - N is the number of processes in the system
- The entry (j, k) in the matrix M_i captures the following knowledge:
  - Suppose $M_i[j, k] = x$
  - Interpretation: As far as process P_i knows, process P_j has sent x messages to process P_k
- Initially, all entries in all matrices are set to 0

The RST Protocol

- Protocol for process P_i:
  - On sending message m to process P_j:

    piggyback matrix M_i on message m
    increment $M_i[i, j]$ by 1

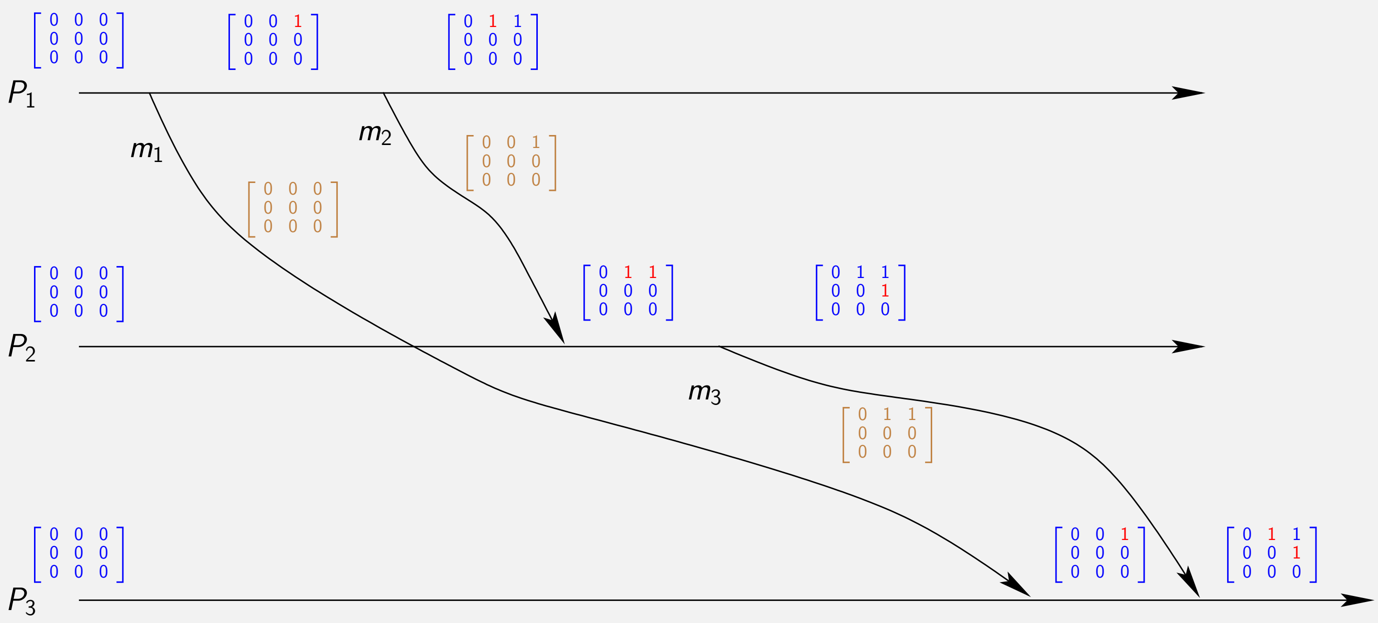
  - On receiving message m from process P_j carrying matrix M_m:

    buffer the message until $M_i[*, i] \geq M_m[*, i]$

  - On delivery of message m sent by process P_j carrying matrix M_m:

    merge M_i and M_m
    $\forall x, y : M_i[x, y] := \max(M_i[x, y], M_m[x, y])$
    increment $M_i[j, i]$ by 1

The RST Protocol: An Illustration



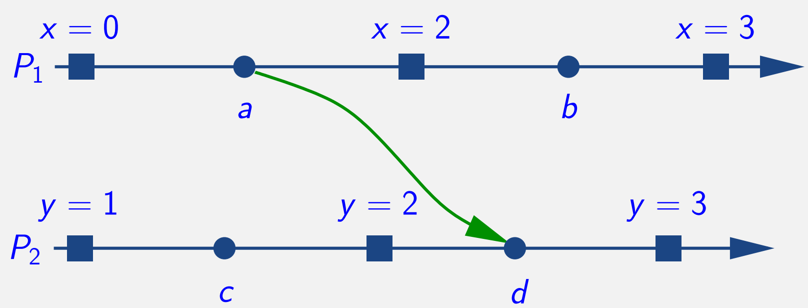
Outline

State of a Distributed System

- State of a distributed system is a collection of states of all its processes and channels:
  - a process state is given by the values of all variables on the process
  - a channel state is given by the set of messages in transit in the channel
    - can be determined by examining states of the two processes it connects
- State of a process is called local state or local snapshot
- State of the system is called global state or global snapshot

Events and Local States

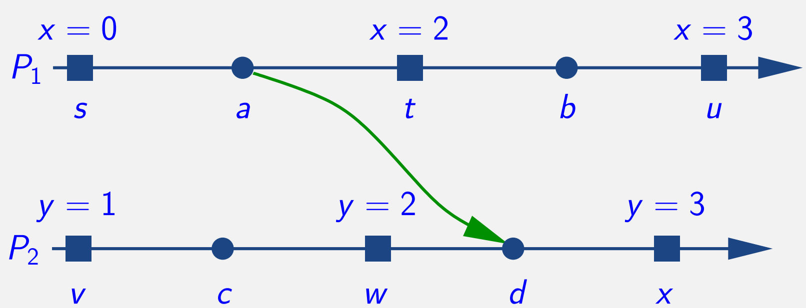
- A process changes its state by executing an event
- Example:



  - Process P_1 changes its state from $x = 0$ to $x = 2$ on executing event a
  - Process P_2 changes its state from $y = 2$ to $y = 3$ on executing event d

When is a Global State Meaningful?

- Example:



  - Is it possible for x to be 0 and y to be 3 at the same time?
  - Does $\{s, x\}$ form a meaningful global state?

Revisiting Happened-Before Relation

- Earlier, happened-before relation was defined on events
- The relation can be extended to local states as follows:
  - let s be a local state of process P_i
  - let t be a local state of process P_j
  - $s \rightarrow t$ if there exist events e and f such that:
    1. P_i executed e after s,
    2. P_j executed f before t, and
    3. $e \rightarrow f$
- If $s \rightarrow t$, then s and t cannot co-exist at the same time
  Also, s must occur before t
- If $s \parallel t$, then s and t can co-exist simultaneously

A Consistent Global State

- For a global state G, let $G[i]$ refer to the local state of process P_i in G
- A global state G is meaningful or consistent if

$$\forall i, j : i \neq j : (G[i] \not\rightarrow G[j]) \wedge (G[j] \not\rightarrow G[i])$$

$$\equiv$$

$$\forall i, j : i \neq j : G[i] \parallel G[j]$$

Recording a Consistent Global Snapshot

- Proposed by Chandy and Lamport (CL)
- Assumptions and Features:
  - channels satisfy first-in-first-out (FIFO) property
  - channels are not required to be bidirectional
  - communication topology may not be fully connected
- Messages exchanged by the underlying computation (whose snapshot is being recorded) are called application messages
- Messages exchanged by the snapshot algorithm are called control messages

The CL Protocol

- *Initially:* all processes are colored blue
- *Eventually:* all processes become red
- On changing color from blue to red:

  record local snapshot
  send a marker message along all outgoing channels

- On receiving marker message along incoming channel C:

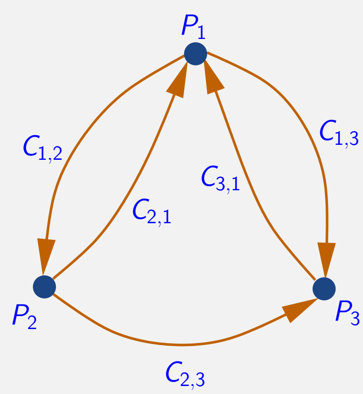
  if color is blue then
    change color from blue to red
  endif
  record state of channel C as application messages received along C since turning red

The CL Protocol (Continued)

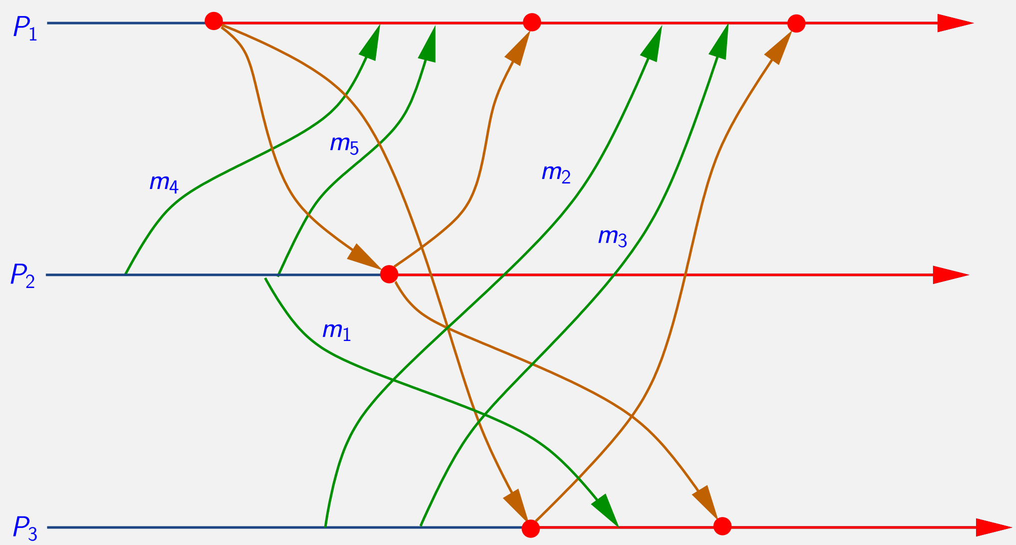
- Any process can initiate the snapshot protocol by spontaneously changing its color from blue to red
  - there can be multiple initiators of the snapshot protocol
- Global Snapshot: local snapshots of processes *just after they turn red*
- In-Transit Messages: *blue* application messages received by processes *after they have turned red*
- Why is the global snapshot consistent?
  - Assume an application message has the same color as its sender
  - Can a blue process receive a red application message?

The CL Protocol: An Illustration

- Three processes: P_1, P_2 and P_3
- Five channels: $C_{1,2}$, $C_{1,3}$, $C_{2,1}$, $C_{2,3}$ and $C_{3,1}$



The CL Protocol: An Illustration (Continued)

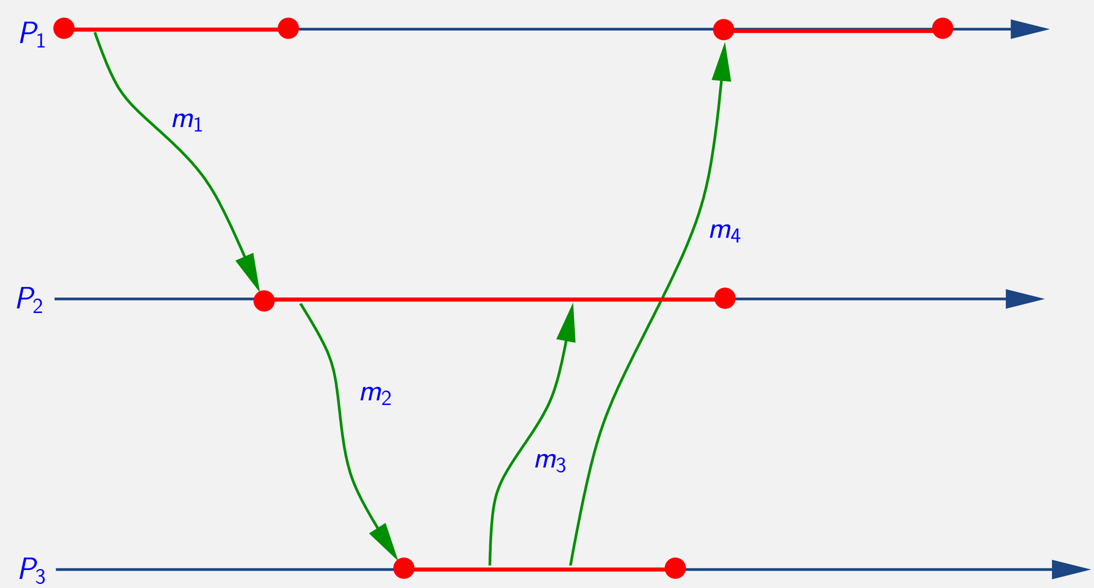


Outline

A Subclass of Distributed Computation

- Many distributed computations obey the following paradigm:
  - A process is either in active state or passive state
  - A process can send an application message *only when it is active*
  - An active process can become passive at any time
  - A passive process, on receiving an application message, becomes active
- Intuitively:
  - If a process is active, then it is doing some work
  - If process is passive, then it is idle
  - An active process uses an application message to send a part of its work to another process

Distributed Computation: An Illustration



Termination Detection

- To detect if the computation has finished doing all the work
  - all processes have become passive, and
  - all channels have become empty
- Different types of computations:
  - diffusing: only one process is active in the beginning
  - non-diffusing: any subset of processes can be active in the beginning
    - no process knows which processes are active and which processes are passive

Huang's Protocol

- *Assumption:* computation is diffusing
  - the initially active process is called the coordinator
    - coordinator is responsible for detecting termination
- *Initially:*
  - coordinator has a weight of 1
  - all other processes have a weight of 0
- *Invariants:*
  - total amount of weight in the system is 1
  - a non-coordinator process has a non-zero weight *if and only if* it is active
  - a channel has a non-zero weight *if and only if* it is non-empty
    - weight of a channel is the sum of the weight of all messages in it

Huang's Protocol (Continued)

- Actions:
  - On sending an application message:
    - send half of the weight along with the message
  - On receiving an application message:
    - add the weight of the message to the current weight
  - On becoming passive:
    - send the current weight to the coordinator
- Coordinator announces termination once:
  - it has become passive and
  - it has collected all the weight

Huang's Protocol: An Illustration