Clocks, Event Ordering, and Global Predicate Computation

Example of Global Predicate

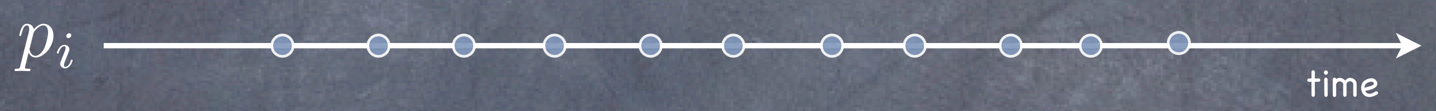
- Setting: Locks in distributed system
    - Objects locked by nodes and moved to the node that is currently modifying it
    - Nodes requesting the object/lock, send a message to the current node locking it and blocks for a response
- How do we detect deadlocks in this scenario?

Events and Histories

- Processes execute sequences of events
- Events can be of 3 types: local, send, and receive
- e_p^i is the i-th event of process p
- The local history h_p of process p is the sequence of events executed by process p

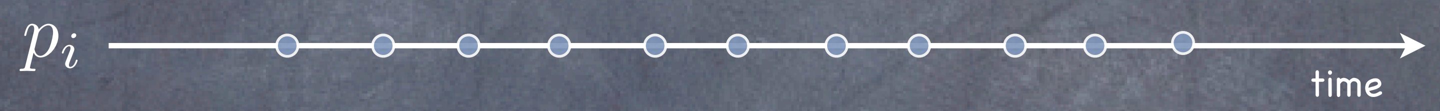
Ordering events

- Observation 1:
  - Events in a local history are totally ordered

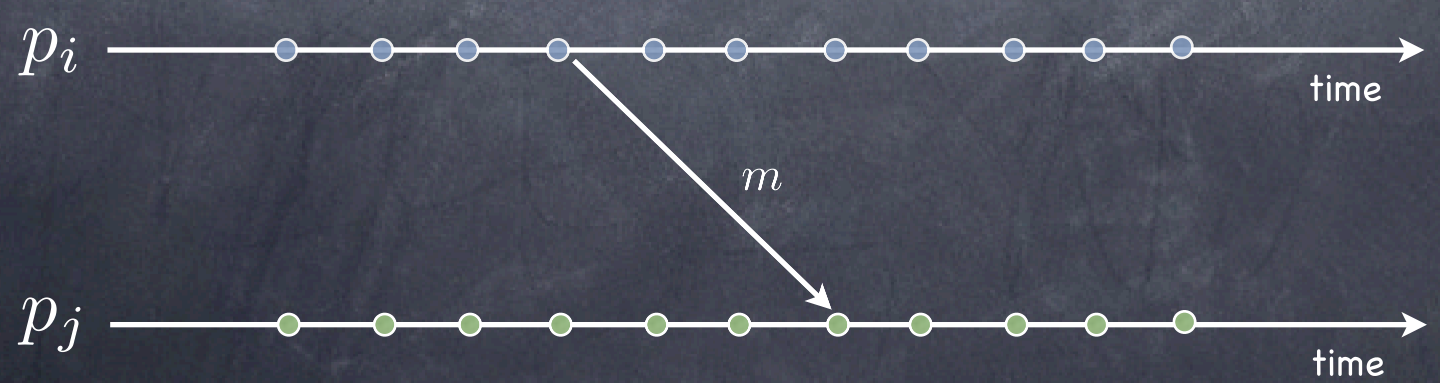


Ordering events

- Observation 1:
  - Events in a local history are <u>totally ordered</u>



- Observation 2:
  - For every message m, $send(m)$ precedes $receive(m)$



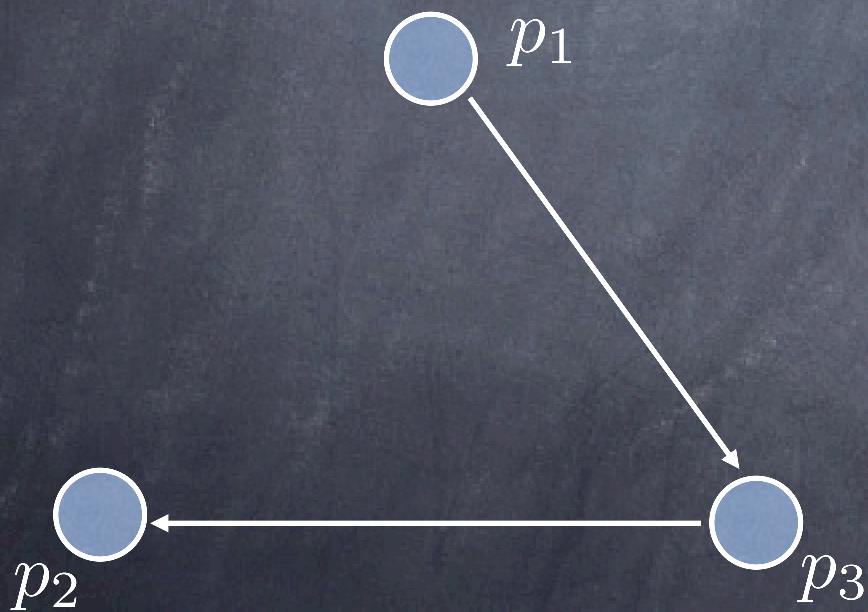
Happened-before (Lamport[1978])

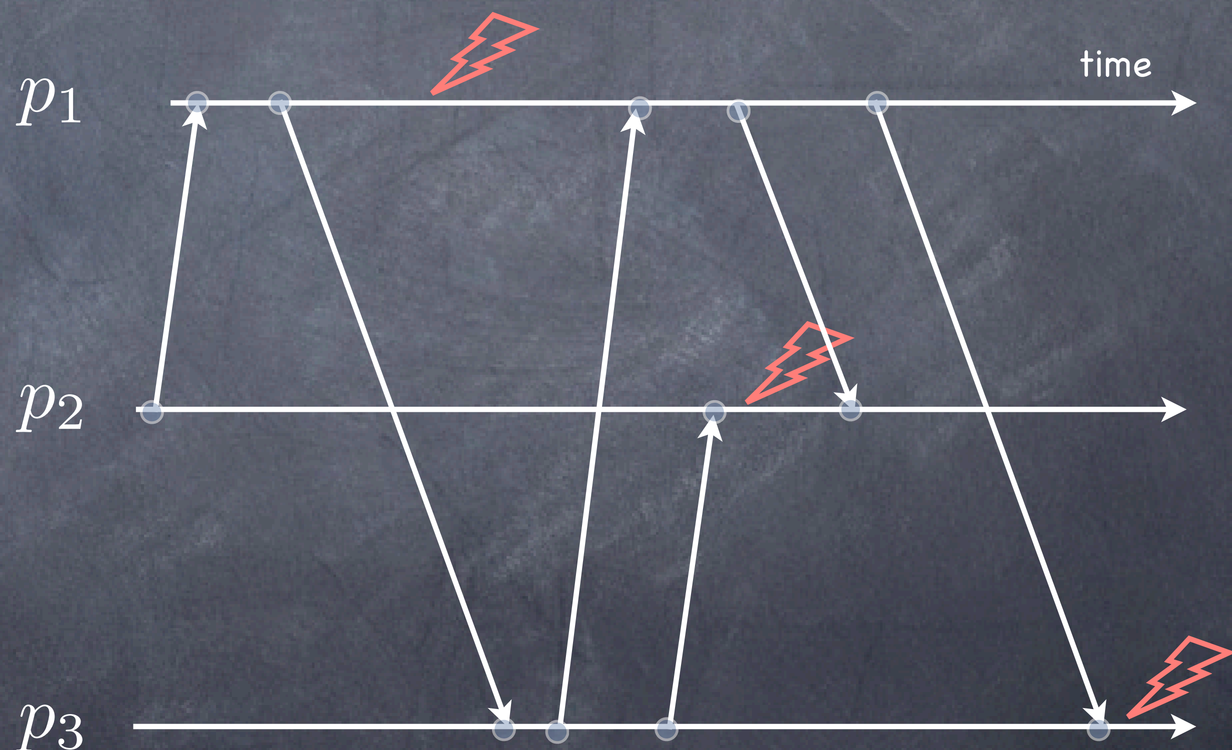
A binary relation $\rightarrow$ defined over events

1. if $e_i^k, e_i^l \in h_i$ and $k < l$, then $e_i^k \rightarrow e_i^l$
2. if $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$
3. if $e \rightarrow e'$ and $e' \rightarrow e''$ then $e \rightarrow e''$

Space-Time diagrams

A graphic representation of a distributed execution





H and $\rightarrow$ impose a partial order

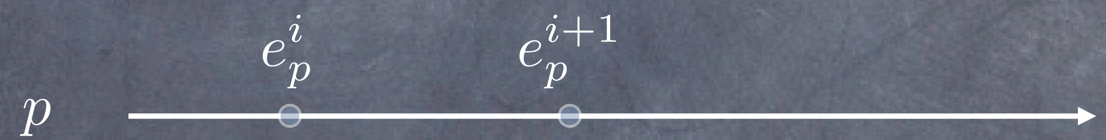
Global States & Clocks

- Need to reason about global states of a distributed system
- Global state: processor state + communication channel state
- Consistent global state: causal dependencies are captured
- Use virtual clocks to reason about the timing relationships between events on different nodes

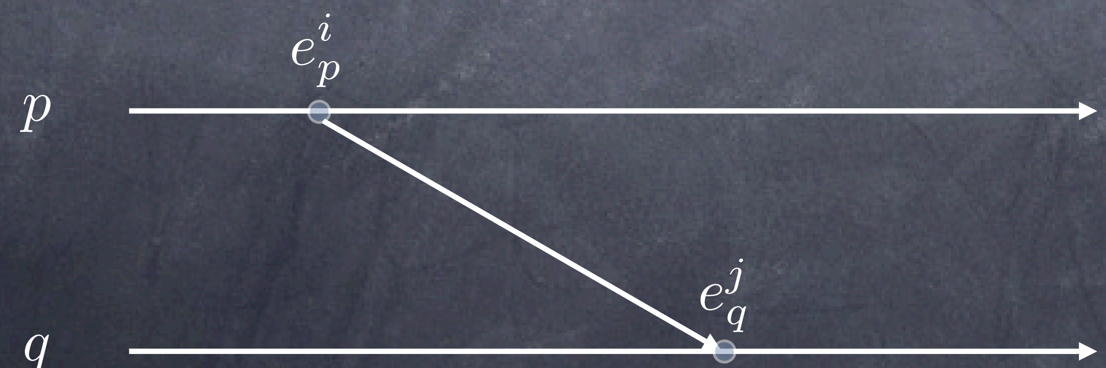
Lamport Clocks

Each process maintains a local variable LC

$LC(e) \equiv$ value of LC for event e

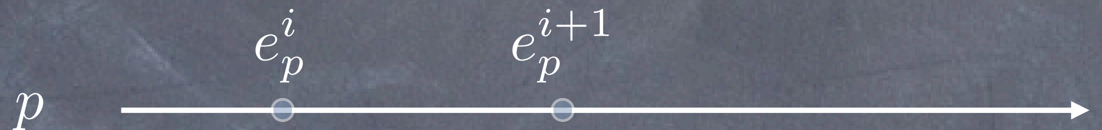


$LC(e_p^i) < LC(e_p^{i+1})$



$LC(e_p^i) < LC(e_q^j)$

Increment Rules



$$LC(e_p^{i+1}) = LC(e_p^i) + 1$$

$$LC(e_q^j) = max(LC(e_q^{j-1}), LC(e_p^i)) + 1$$

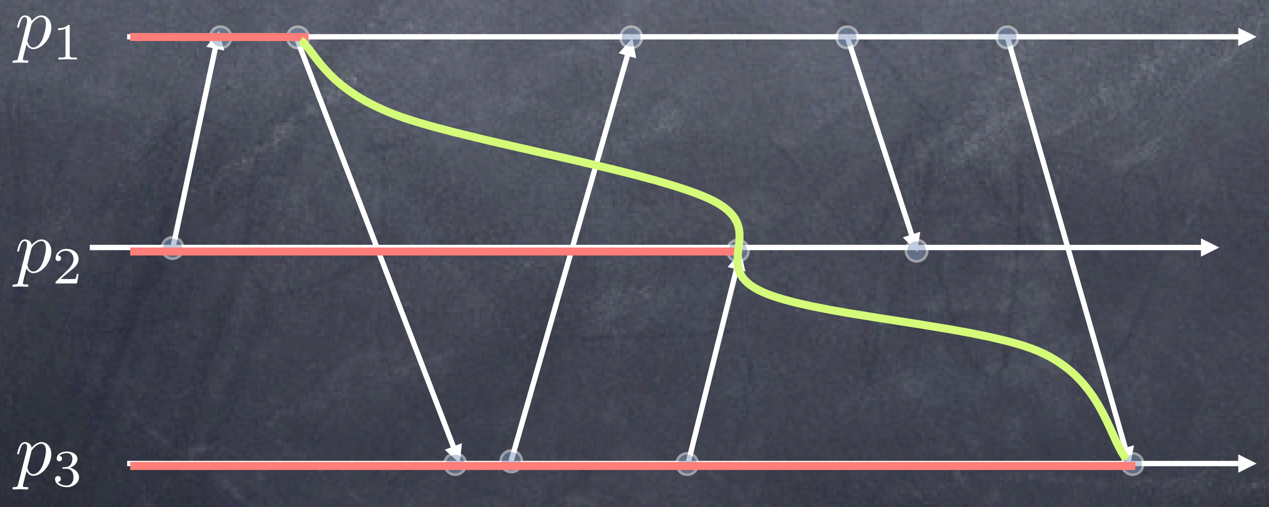
Timestamp m with $TS(m) = LC(send(m))$

Cuts

A cut C is a subset of the global history of H

The frontier of C is the set of events

$$e_1^{c_1}, e_2^{c_2}, \ldots e_n^{c_n}$$



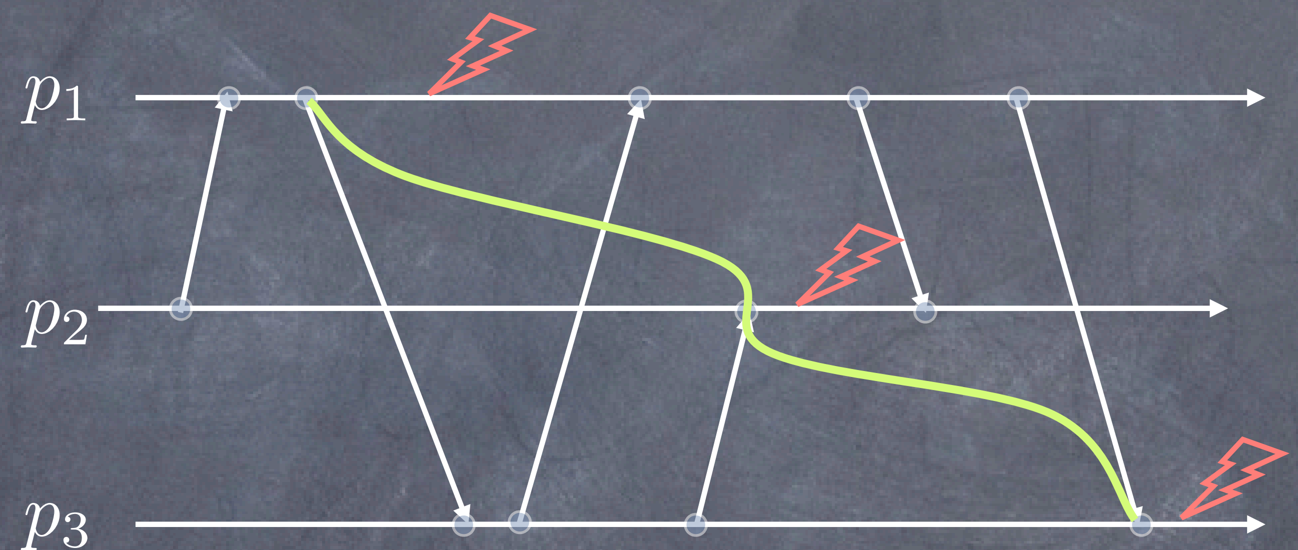
Consistent cuts and consistent global states

- A cut is consistent if

  $$\forall e_i, e_j : e_j \in C \wedge e_i \rightarrow e_j \Rightarrow e_i \in C$$

- A consistent global state is one corresponding to a consistent cut

What p_0 sees



Not a consistent global state: the cut contains the event corresponding to the receipt of the last message by p_3 but not the corresponding send event

Global Consistent States

- Can we use Lamport Clocks as part of a mechanism to get globally consistent states?

Global Snapshot

- Develop a simple global snapshot protocol
- Refine protocol as we relax assumptions
- Record:
  - processor states
  - channel states
- Assumptions:
  - FIFO channels
  - Each m timestamped with with $T(send(m))$

Snapshot I

i. p_0 selects t_{ss}

ii. p_0 **sends** "take a snapshot at t_{ss}" **to** all processes

iii. **when** clock of p_i reads t_{ss} **then** p

a. records its local state σ_i

b. sends an empty message along its outgoing channels

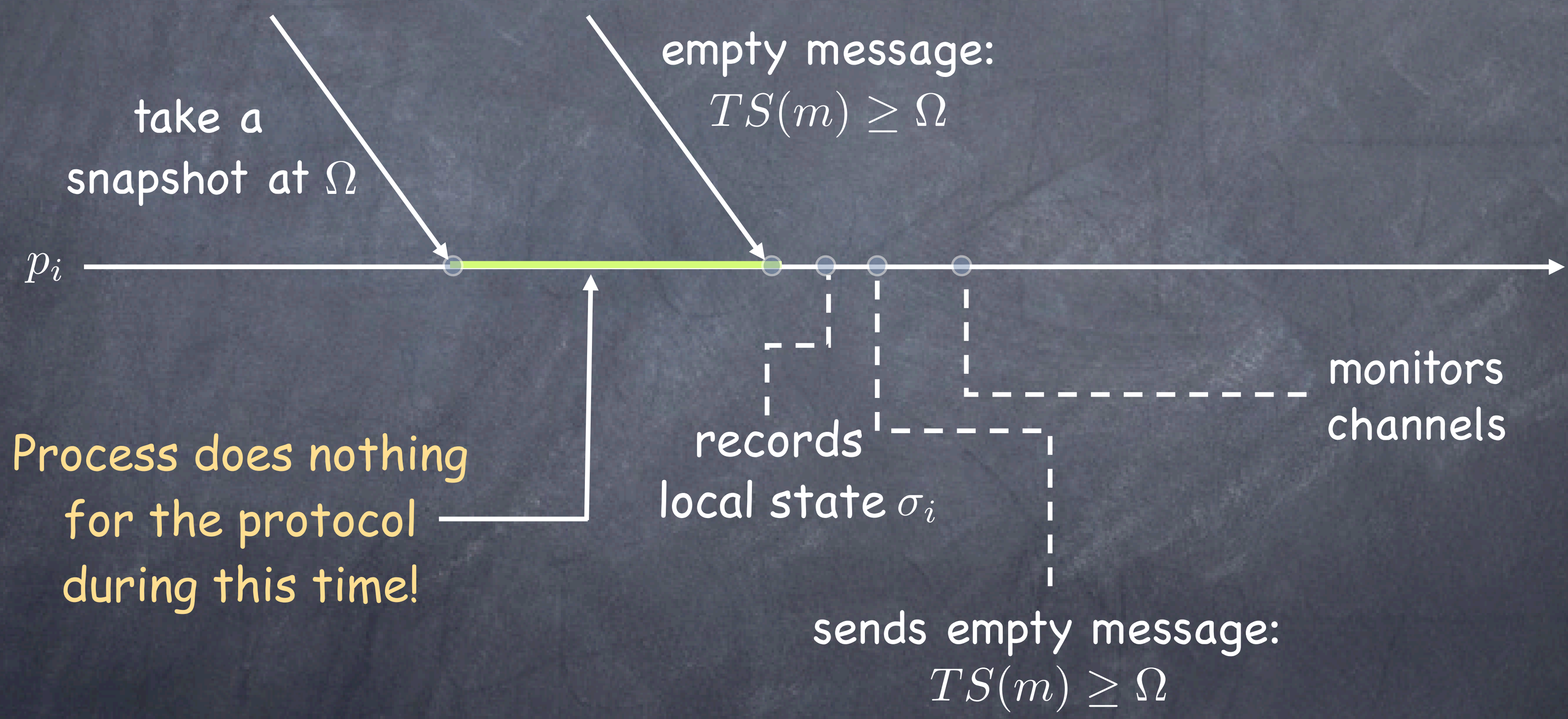
c. starts recording messages received on each of incoming channels

d. stops recording a channel when it receives first message with timestamp greater than or equal to t_{ss}

Snapshot II

- processor p_0 selects Ω
- p_0 sends "take a snapshot at Ω" to all processes; it waits for all of them to reply and then sets its logical clock to Ω
- when clock of p_i reads Ω then p_i
  - records its local state σ_i
  - sends an empty message along its outgoing channels
  - starts recording messages received on each incoming channel
  - stops recording a channel when receives first message with timestamp greater than or equal to Ω

Relaxing synchrony



Snapshot III

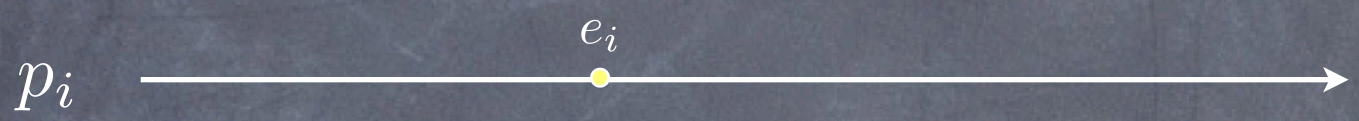
- processor p_0 sends itself "take a snapshot "
- when p_i receives "take a snapshot" for the first time from p_j :
  - records its local state σ_i
  - sends "take a snapshot" along its outgoing channels
  - sets channel from p_j to empty
  - starts recording messages received over each of its other incoming channels
- when p_i receives "take a snapshot" beyond the first time from p_k:
  - stops recording channel from p_k
- when p_i has received "take a snapshot" on all channels, it sends collected state to p_0 and stops.

Same problem, different approach

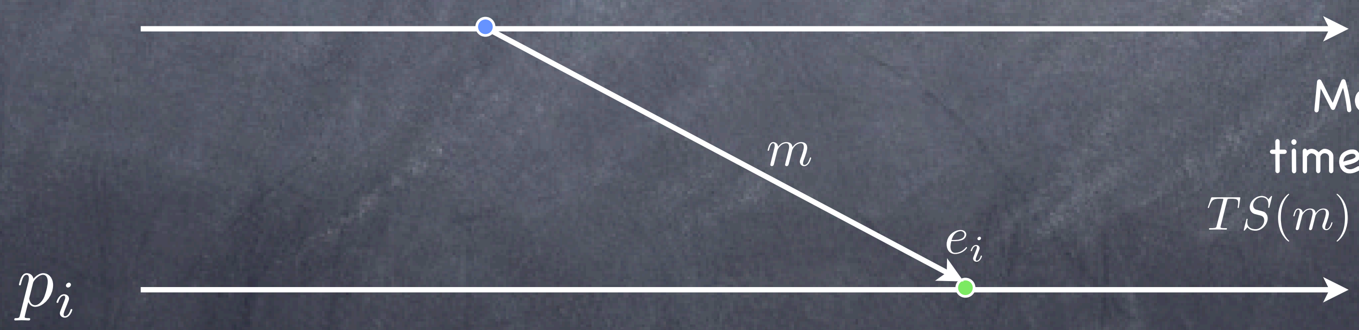
- Monitor process does not query explicitly
- Instead, it passively collects information and uses it to build an observation.
  (reactive architectures, Harel and Pnueli [1985])

An observation is an ordering of event of the distributed computation based on the order in which the receiver is notified of the events.

Update rules



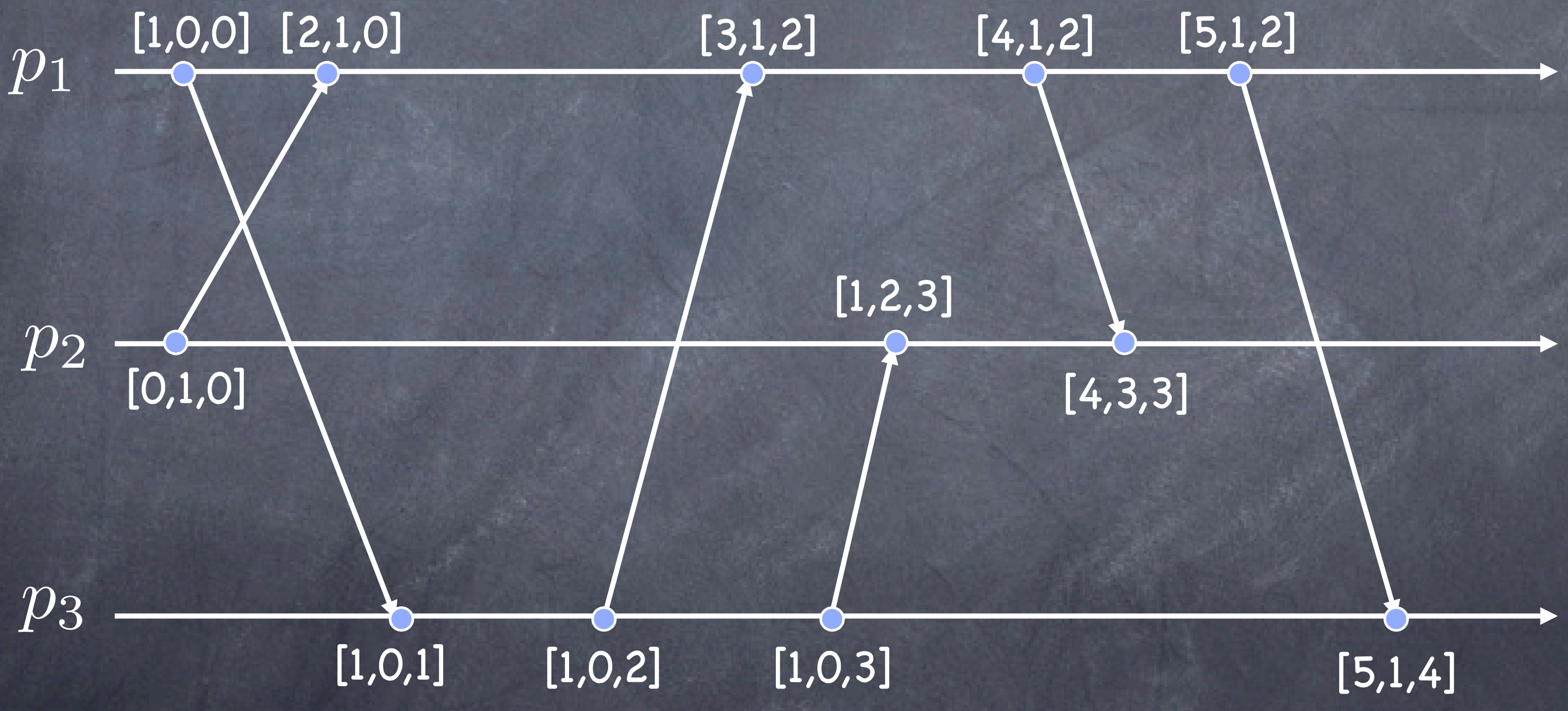
$$VC(e_i)[i] := VC[i] + 1$$



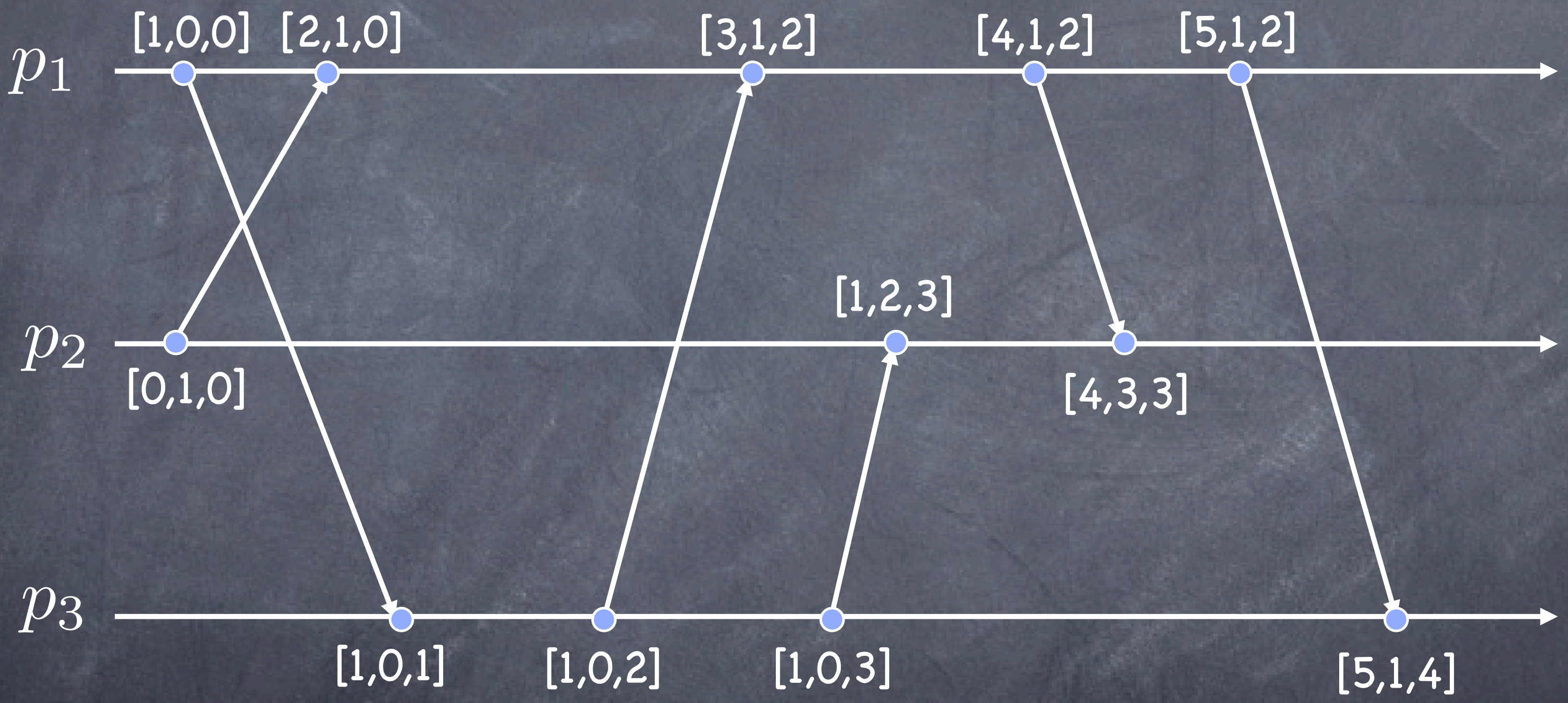
Message m is timestamped with $TS(m) = VC(send(m))$

$$VC(e_i) := max(VC, TS(m))$$
$$VC(e_i)[i] := VC[i] + 1$$

Example



Operational interpretation



$VC(e_i)[i]$ = no. of events executed by p_i up to and including e_i

$VC(e_i)[j]$ = no. of events executed by p_j that happen before e_i of p_i

The protocol

- p_0 maintains an array $D[1, \ldots, n]$ of counters
- $D[i] = TS(m_i)[i]$ where m_i is the last message delivered from p_i

Rule: Deliver m from p_j as soon as both of the following conditions are satisfied:

$$D[j] = TS(m)[j] - 1$$

$$D[k] \geq TS(m)[k], \forall k \neq j$$

Summary

- Lamport clocks and vector clocks provide us with good tools to reason about timing of events in a distributed system
- Global snapshot algorithm provides us with an efficient mechanism for obtaining consistent global states