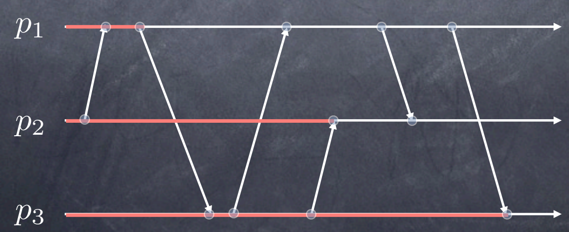
Cuts

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

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots h_n^{c_n}$$



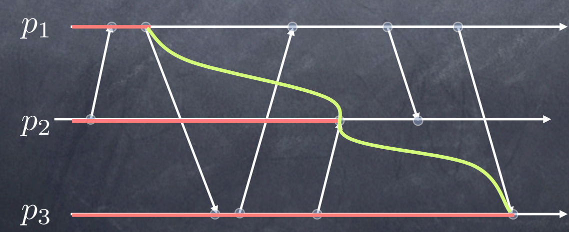
Cuts

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

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots h_n^{c_n}$$

The frontier of C is the set of events

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



Global states and cuts

- The global state of a distributed computation is an n-tuple of local states

$$\Sigma = (\sigma_1, \ldots \sigma_n)$$

- To each cut $(c_1 \ldots c_n)$ corresponds a global state $(\sigma_1^{c_1}, \ldots \sigma_n^{c_n})$

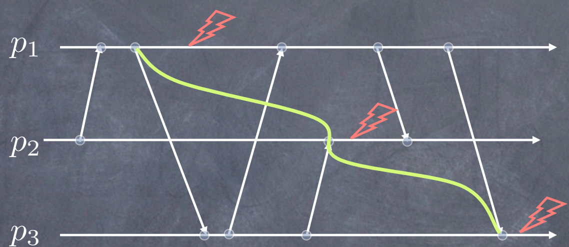
Consistent cuts and consistent global states

- A cut is consistent if

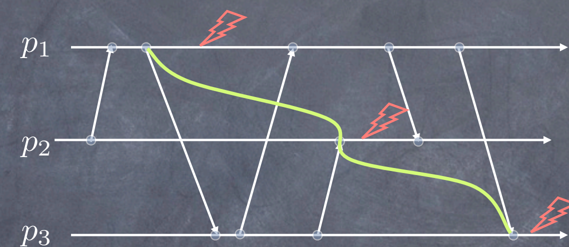
$$\forall e_i, e_j : e_j \in C \land e_i \rightarrow e_j \Rightarrow e_i \in C$$

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

What p_0 sees



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

Our task

- Develop a protocol by which a processor can build a consistent global state
- Informally, we want to be able to take a snapshot of the computation
- Not obvious in an asynchronous system...

Our approach

- Develop a simple synchronous 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. starts recording messages received on each of incoming channels
c. stops recording a channel when it receives first message with timestamp greater than or equal to t_{ss}

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}

Correctness

Theorem Snapshot I produces a consistent cut

Proof Need to prove $e_j \in C \wedge e_i \rightarrow e_j \Rightarrow e_i \in C$

< Definition >
0. $e_j \in C \equiv T(e_j) < t_{ss}$

< Assumption >
1. $e_j \in C$

< Assumption >
2. $e_i \rightarrow e_j$

< 0 and 1>
3. $T(e_j) < t_{ss}$

< Property of real time>
4. $e_i \rightarrow e_j \Rightarrow T(e_i) < T(e_j)$

< 2 and 4>
5. $T(e_i) < T(e_j)$

< 5 and 3>
6. $T(e_i) < t_{ss}$

< Definition >
7. $e_i \in C$

Clock Condition

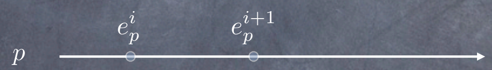
< Property of real time>
4. $e_i \rightarrow e_j \Rightarrow T(e_i) < T(e_j)$

Can the Clock Condition be implemented some other way?

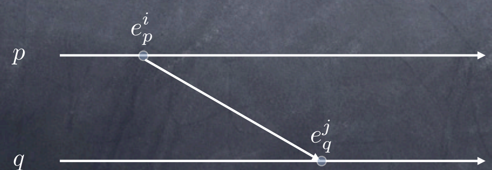
Lamport Clocks

Each process maintains a local variable LC

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

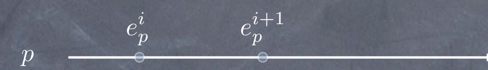


$LC(e_p^i) < LC(e_p^{i+1})$

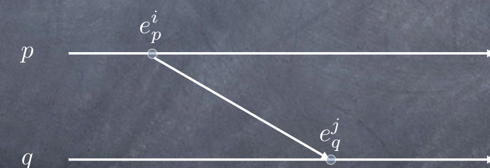


$LC(e_p^i) < LC(e_q^j)$

Increment Rules



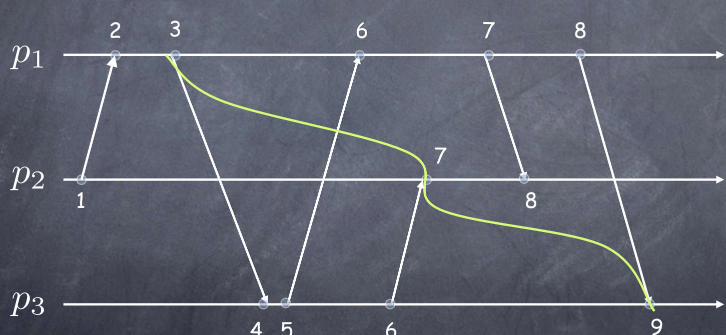
$LC(e_p^{i+1}) = LC(e_p^i) + 1$



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

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

Space-Time Diagrams and Logical Clocks



A subtle problem

when $LC = t$ do S

doesn't make sense for Lamport clocks!

- there is no guarantee that LC will ever be t
- S is anyway executed after $LC = t$

Fixes:

- if e is internal/send and $LC = t - 2$
  - execute e and then S
- if $e = receive(m) \wedge (TS(m) \geq t) \wedge (LC \leq t - 1)$
  - put message back in channel
  - re-enable e ; set $LC = t - 1$; execute S

An obvious problem

- No t_{ss}!
- Choose Ω large enough that it cannot be reached by applying the update rules of logical clocks

An obvious problem

- No t_{ss}!
- Choose Ω large enough that it cannot be reached by applying the update rules of logical clocks

mmmmhhhh...

An obvious problem

- No t_{ss}!
- Choose Ω large enough that it cannot be reached by applying the update rules of logical clocks

mmmmhhhh...

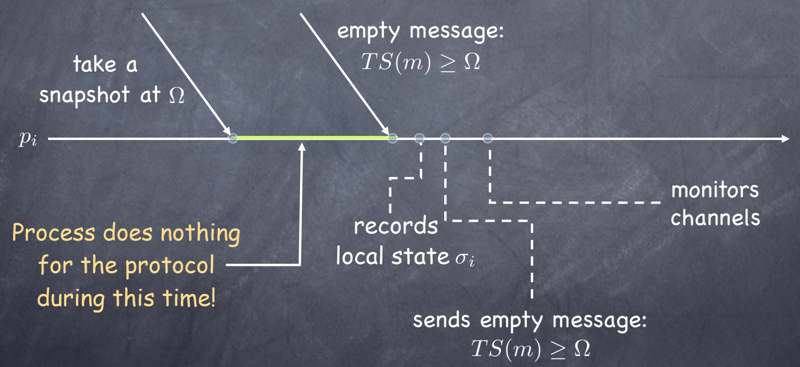
- Doing so assumes
  - upper bound on message delivery time
  - upper bound relative process speeds

We better relax it...

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



Use empty message to announce snapshot!

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.

Snapshots: a perspective

- The global state Σ^s saved by the snapshot protocol is a consistent global state

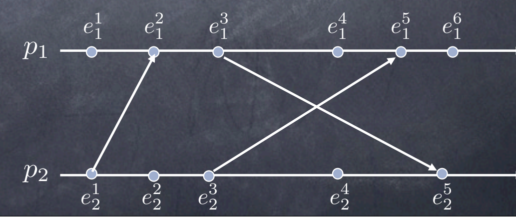
Snapshots: a perspective

- The global state Σ^s saved by the snapshot protocol is a consistent global state
- But did it ever occur during the computation?
  - a distributed computation provides only a partial order of events
  - many total orders (runs) are compatible with that partial order
  - all we know is that Σ^s could have occurred

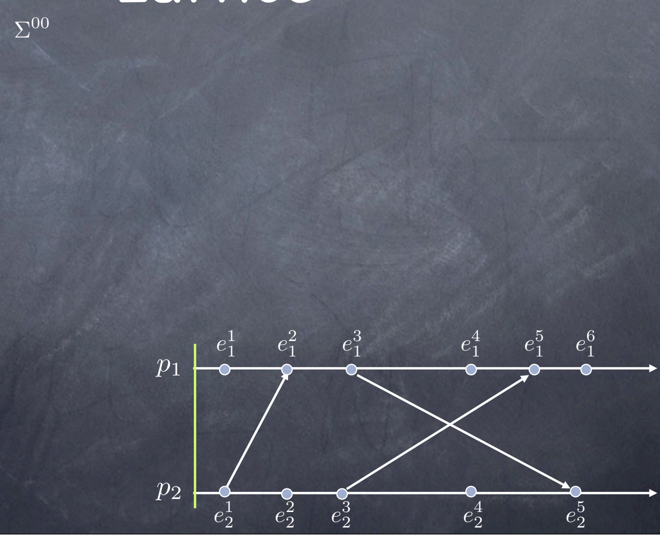
Snapshots: a perspective

- The global state Σ^s saved by the snapshot protocol is a consistent global state
- But did it ever occur during the computation?
  - a distributed computation provides only a partial order of events
  - many total orders (runs) are compatible with that partial order
  - all we know is that Σ^s could have occurred
- We are evaluating predicates on states that may have never occurred!

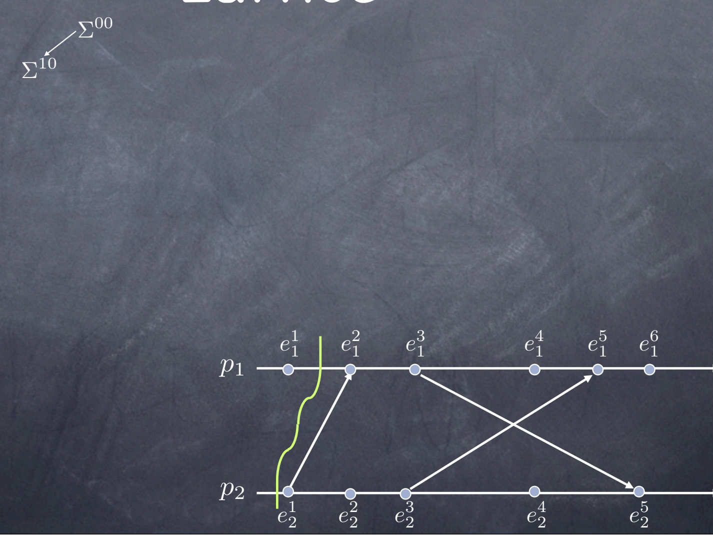
An Execution and its Lattice

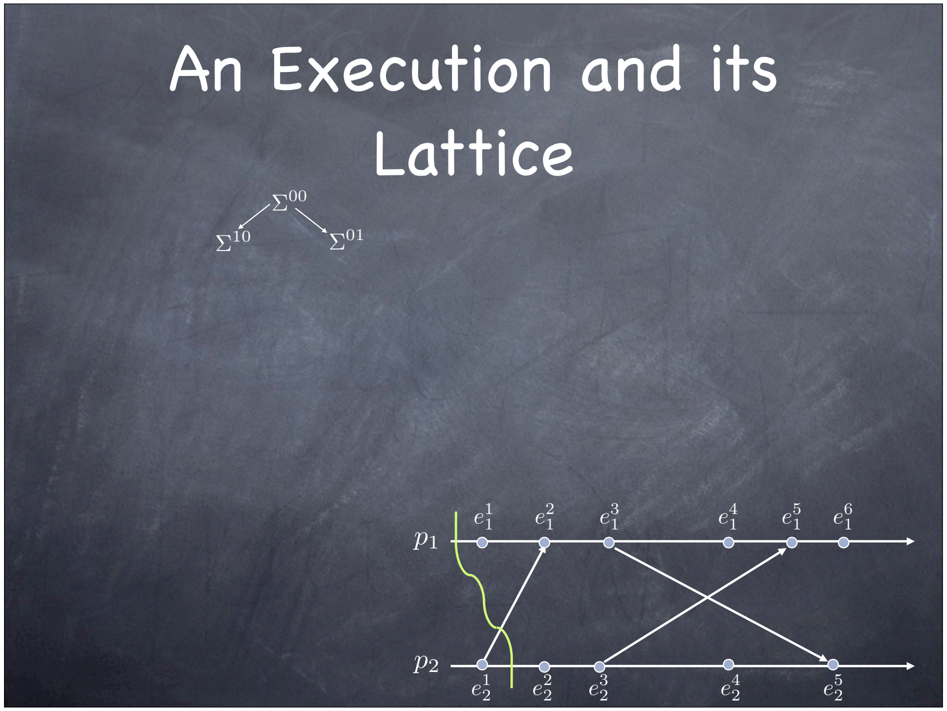


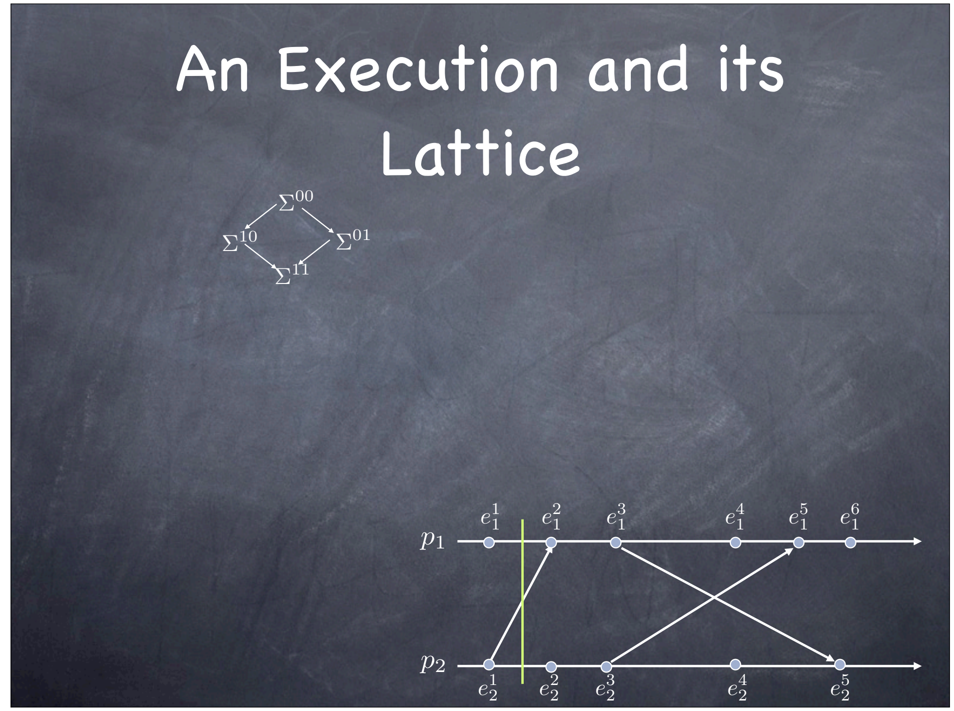
An Execution and its Lattice

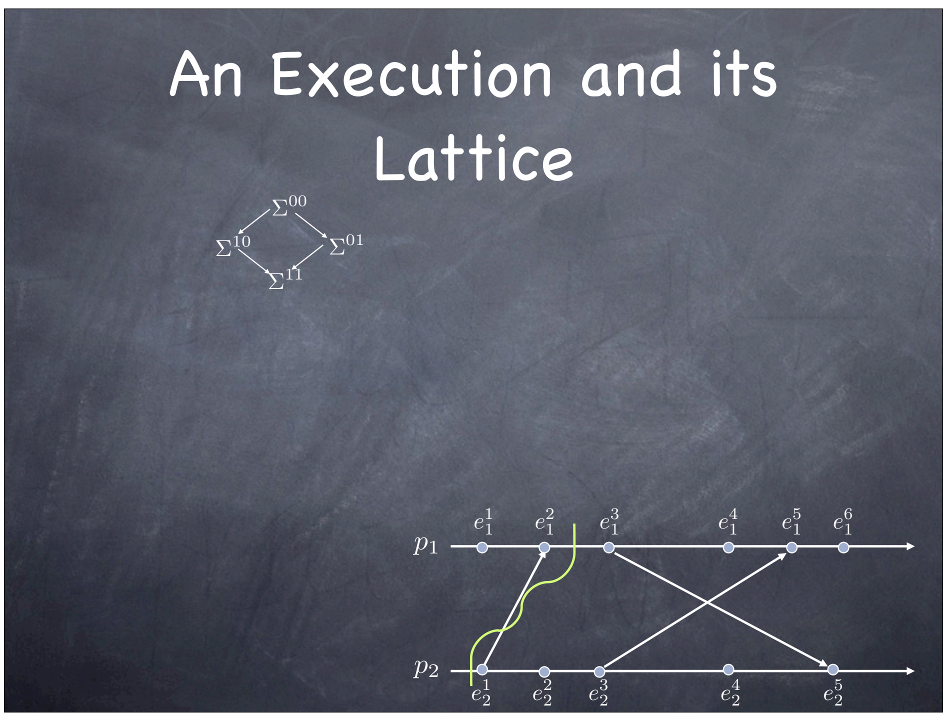


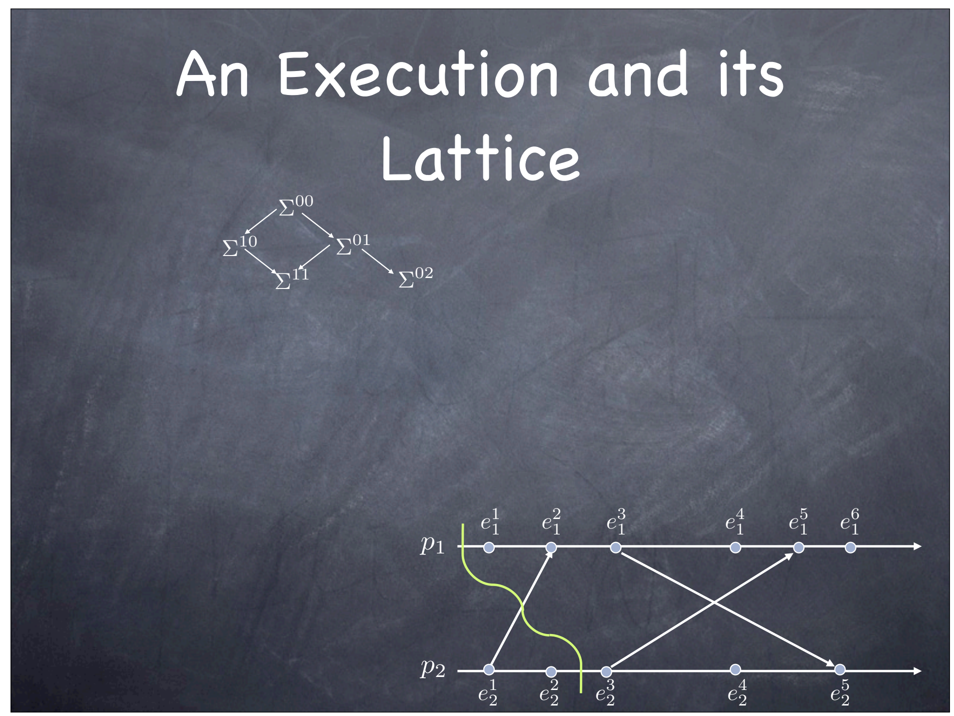
An Execution and its Lattice



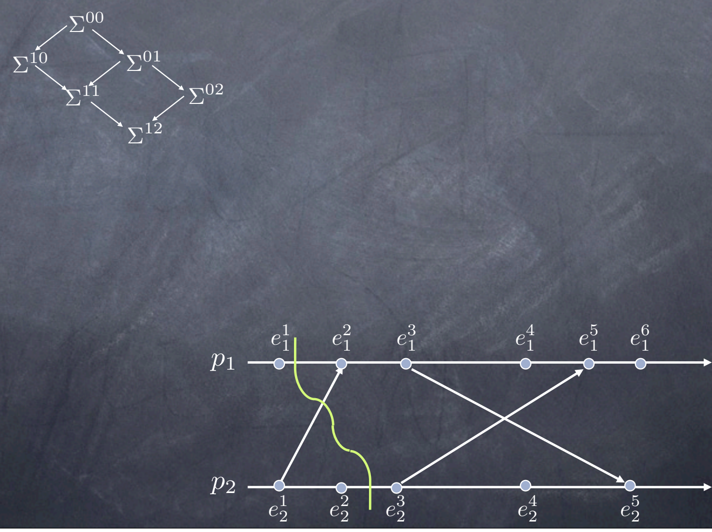

An Execution and its Lattice
Σ^{00}
Σ^{10} Σ^{01}
p_1 p_2
e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
e_2^1 e_2^2 e_2^3 e_2^4 e_2^5


An Execution and its Lattice
Σ^{00}
Σ^{10} Σ^{01}
Σ^{11}
p_1 p_2
e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
e_2^1 e_2^2 e_2^3 e_2^4 e_2^5

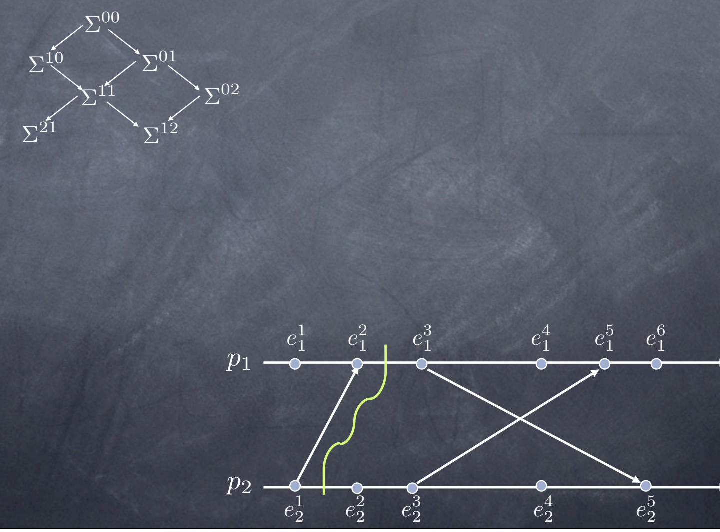

An Execution and its Lattice
Σ^{00}
Σ^{10} Σ^{01}
Σ^{11}
p_1 p_2
e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
e_2^1 e_2^2 e_2^3 e_2^4 e_2^5


An Execution and its Lattice
Σ^{00}
Σ^{10} Σ^{01}
Σ^{11} Σ^{02}
p_1 p_2
e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
e_2^1 e_2^2 e_2^3 e_2^4 e_2^5

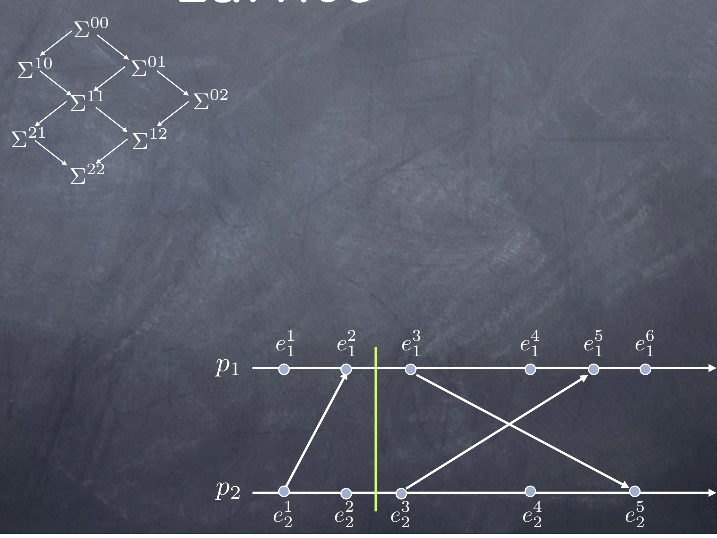
An Execution and its Lattice



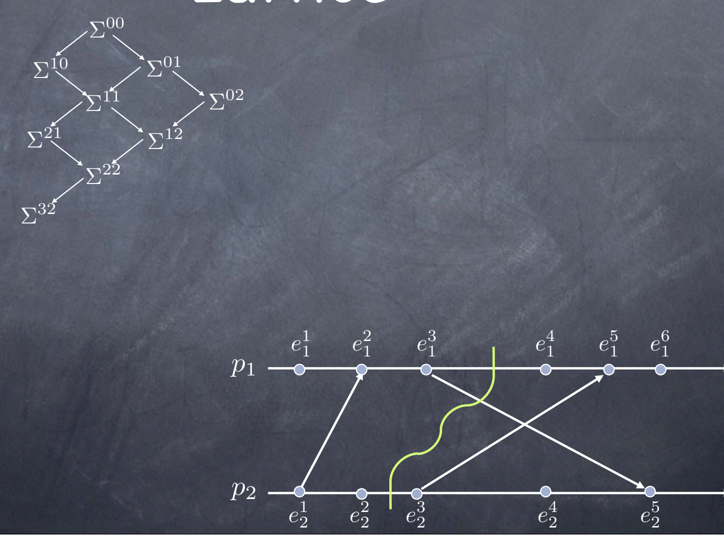
An Execution and its Lattice



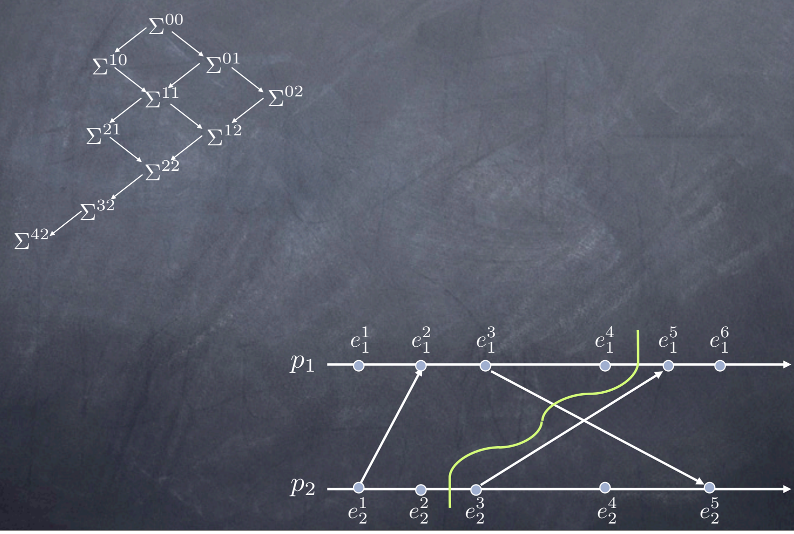
An Execution and its Lattice



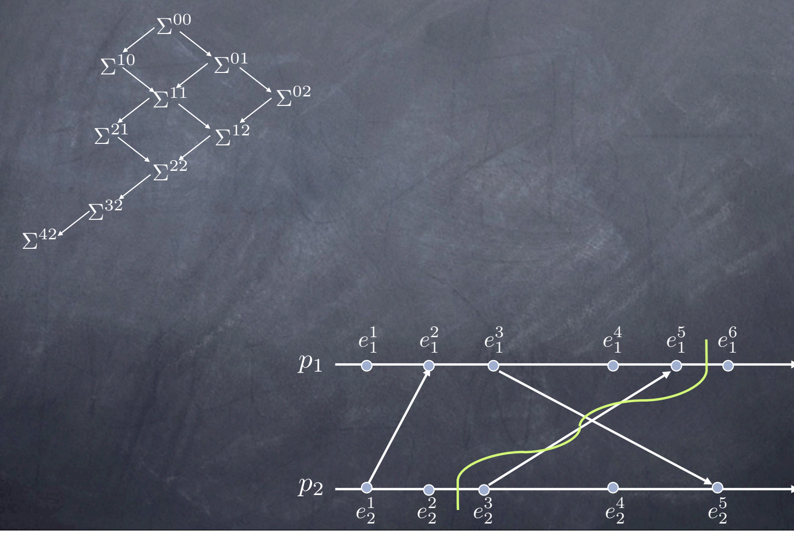
An Execution and its Lattice



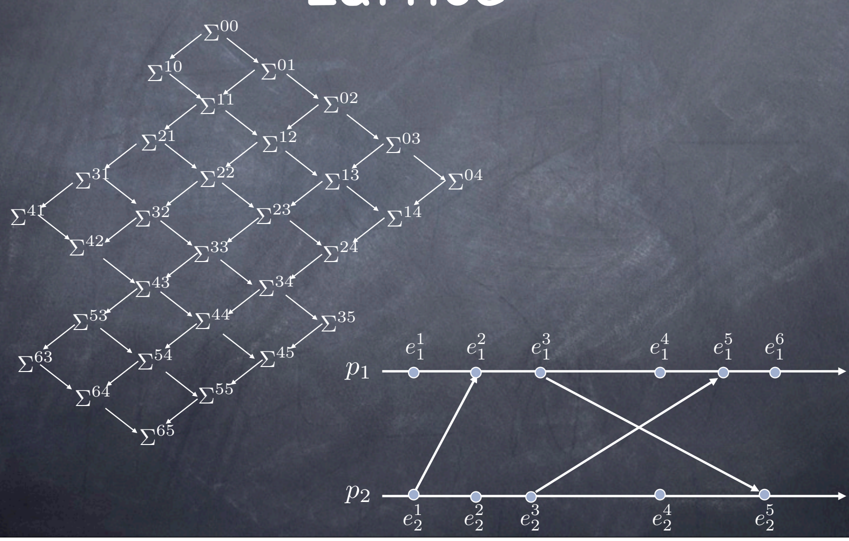
An Execution and its Lattice



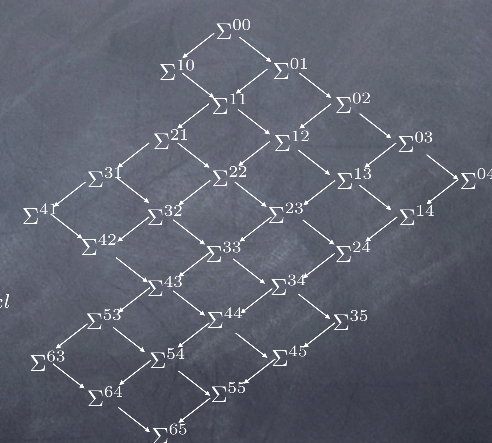
An Execution and its Lattice



An Execution and its Lattice

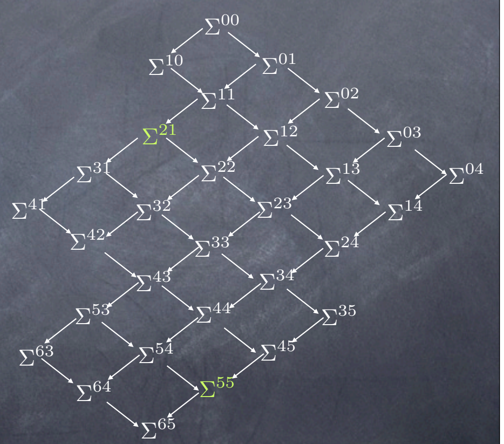


Reachability



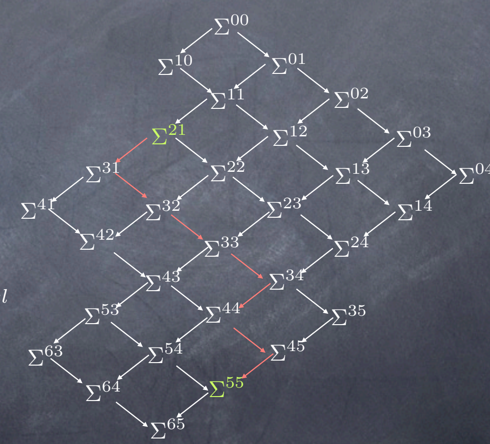
Σ^{kl} is reachable from Σ^{ij} if there is a path from Σ^{ij} to Σ^{kl} in the lattice

Reachability



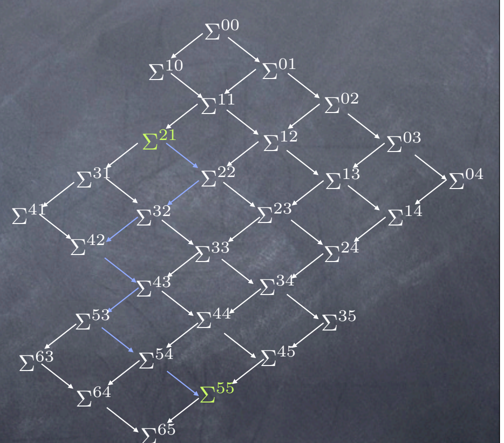
Σ^{kl} is reachable from Σ^{ij} if there is a path from Σ^{ij} to Σ^{kl} in the lattice

Reachability



Σ^{kl} is reachable from Σ^{ij} if there is a path from Σ^{ij} to Σ^{kl} in the lattice

Reachability



Σ^{kl} is reachable from Σ^{ij} if there is a path from Σ^{ij} to Σ^{kl} in the lattice

$$\Sigma^{ij} \rightsquigarrow \Sigma^{kl}$$

So, why do we care about Σ^s again?

- Deadlock is a stable property

  Deadlock $\Rightarrow \Box$ Deadlock

- If a run R of the snapshot protocol starts in Σ^i and terminates in Σ^f, then $\Sigma^i \rightsquigarrow_R \Sigma^f$

So, why do we care about Σ^s again?

- Deadlock is a stable property

  Deadlock $\Rightarrow \Box$ Deadlock

- If a run R of the snapshot protocol starts in Σ^i and terminates in Σ^f, then $\Sigma^i \leadsto_R \Sigma^f$
- Deadlock in Σ^s implies deadlock in Σ^f

So, why do we care about Σ^s again?

- Deadlock is a stable property

  Deadlock $\Rightarrow \Box$ Deadlock

- If a run R of the snapshot protocol starts in Σ^i and terminates in Σ^f, then $\Sigma^i \leadsto_R \Sigma^f$
- Deadlock in Σ^s implies deadlock in Σ^f
- No deadlock in Σ^s implies no deadlock in Σ^i

Same problem, different approach

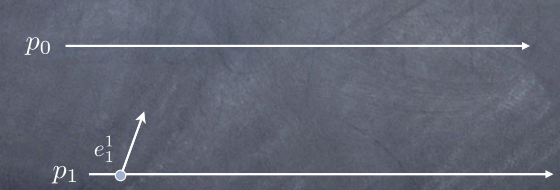
- Monitor process does not query explicitly
- Instead, it passively collects information and uses it to build an observation.

  (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.

Observations: a few observations

- An observation puts no constraint on the order in which the monitor receives notifications



Observations: a few observations

- An observation puts no constraint on the order in which the monitor receives notifications

Observations: a few observations

- An observation puts no constraint on the order in which the monitor receives notifications

Observations: a few observations

- An observation puts no constraint on the order in which the monitor receives notifications

To obtain a run, messages must be delivered to the monitor in FIFO order

Observations: a few observations

- An observation puts no constraint on the order in which the monitor receives notifications

To obtain a run, messages must be delivered to the monitor in FIFO order
What about consistent runs?

Causal delivery

FIFO delivery guarantees:

$send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$

Causal delivery

FIFO delivery guarantees:

$send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$

Causal delivery generalizes FIFO:

$send_i(m) \rightarrow send_k(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$

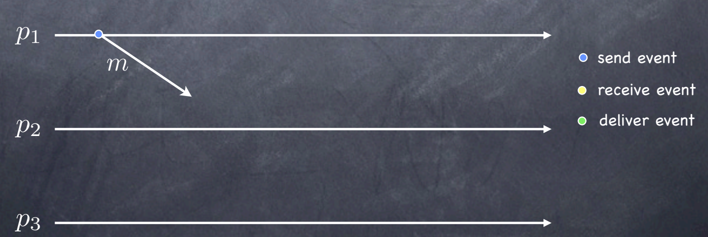
Causal delivery

FIFO delivery guarantees:

$send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$

Causal delivery generalizes FIFO:

$send_i(m) \rightarrow send_k(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$



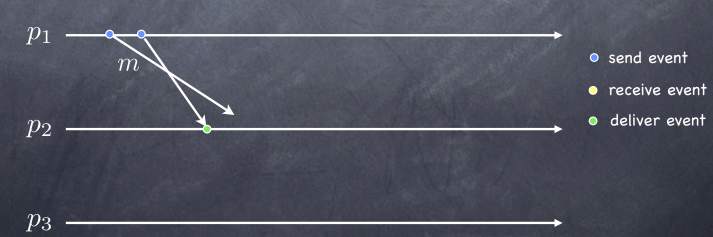
Causal delivery

FIFO delivery guarantees:

$send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$

Causal delivery generalizes FIFO:

$send_i(m) \rightarrow send_k(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$



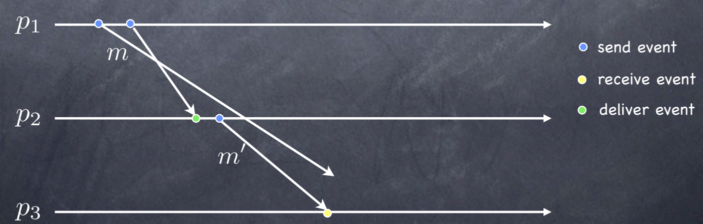
Causal delivery

FIFO delivery guarantees:

$send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$

Causal delivery generalizes FIFO:

$send_i(m) \rightarrow send_k(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$



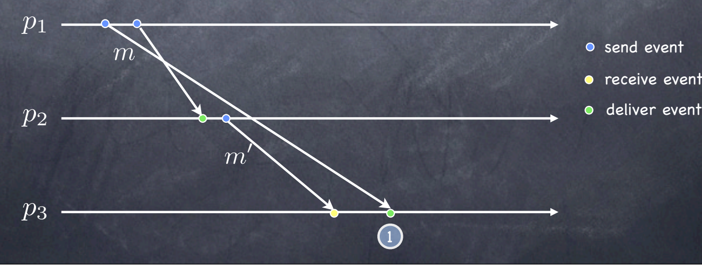
Causal delivery

FIFO delivery guarantees:

$send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$

Causal delivery generalizes FIFO:

$send_i(m) \rightarrow send_k(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$



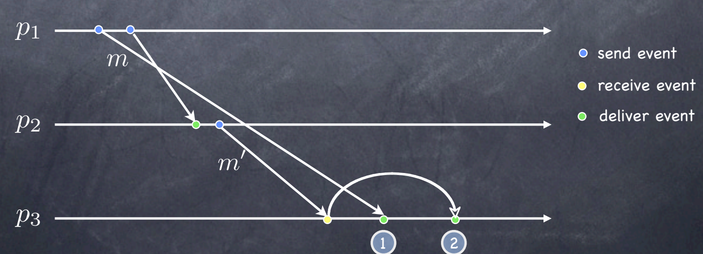
Causal delivery

FIFO delivery guarantees:

$send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$

Causal delivery generalizes FIFO:

$send_i(m) \rightarrow send_k(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$



Causal Delivery in Synchronous Systems

We use the upper bound Δ on message delivery time

Causal Delivery in Synchronous Systems

We use the upper bound Δ on message delivery time

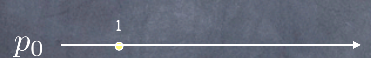
DR1: At time t, p_0 delivers all messages it received with timestamp up to $t - \Delta$ in increasing timestamp order

Causal Delivery with Lamport Clocks

DR1.1: Deliver all received messages in increasing (logical clock) timestamp order.

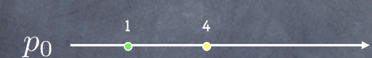
Causal Delivery with Lamport Clocks

DR1.1: Deliver all received messages in increasing (logical clock) timestamp order.



Causal Delivery with Lamport Clocks

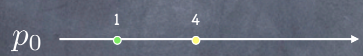
DR1.1: Deliver all received messages in increasing (logical clock) timestamp order.



Should p_0 deliver?

Causal Delivery with Lamport Clocks

DR1.1: Deliver all received messages in increasing (logical clock) timestamp order.



Should p_0 deliver?

Problem: Lamport Clocks don't provide gap detection

Given two events e and e' and their clock values $LC(e)$ and $LC(e')$—where $LC(e) < LC(e')$ determine whether some event e'' exists s.t.

$$LC(e) < LC(e'') < LC(e')$$

Stability

DR2: Deliver all received stable messages in increasing (logical clock) timestamp order.

A message m received by p is stable at p if p will never receive a future message m' s.t.

$$TS(m') < TS(m)$$

Implementing Stability

- Real-time clocks
  - wait for Δ time units

Implementing Stability

- Real-time clocks
  - wait for Δ time units
- Lamport clocks
  - wait on each channel for m s.t. $TS(m) > LC(e)$
- Design better clocks!

Clocks and STRONG Clocks

- Lamport clocks implement the clock condition:

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

- We want new clocks that implement the strong clock condition:

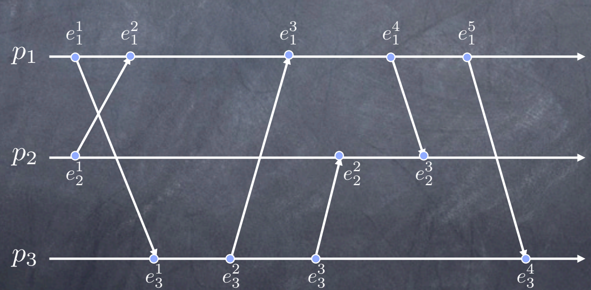
$$e \rightarrow e' \equiv SC(e) < SC(e')$$

Causal Histories

- The causal history of an event e in $(H, \rightarrow)$ is the set
$$\theta(e) = \{e' \in H \mid e' \rightarrow e\} \cup \{e\}$$

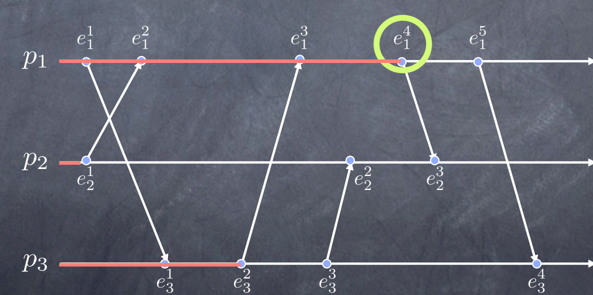
Causal Histories

- The causal history of an event e in $(H, \rightarrow)$ is the set
$$\theta(e) = \{e' \in H \mid e' \rightarrow e\} \cup \{e\}$$



Causal Histories

- The causal history of an event e in $(H, \rightarrow)$ is the set
$$\theta(e) = \{e' \in H \mid e' \rightarrow e\} \cup \{e\}$$



$$e \rightarrow e' \equiv \theta(e) \subset \theta(e')$$

How to build $\theta(e)$

Each process p_i:

- initializes θ: $\theta := \emptyset$
- if e_i^k is an internal or send event, then
  $$\theta(e_i^k) := \{e_i^k\} \cup \theta(e_i^{k-1})$$
- if e_i^k is a receive event for message m, then
  $$\theta(e_i^k) := \{e_i^k\} \cup \theta(e_i^{k-1}) \cup \theta(send(m))$$

Pruning causal histories

- Prune segments of history that are known to all processes (Peterson, Bucholz and Schlichting)
- Use a more clever way to encode $\theta(e)$

Vector Clocks

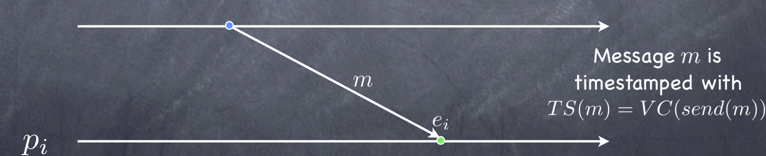
- Consider $\theta_i(e)$, the projection of $\theta(e)$ on p_i
- $\theta_i(e)$ is a prefix of h^i: $\theta_i(e) = h_i^{k_i}$ – it can be encoded using k_i
- $\theta(e) = \theta_1(e) \cup \theta_2(e) \cup \ldots \cup \theta_n(e)$ can be encoded using $k_1, k_2, \ldots, k_n$

Represent θ using an n-vector VC such that

$$VC(e)[i] = k \Leftrightarrow \theta_i(e) = h_i^{k_i}$$

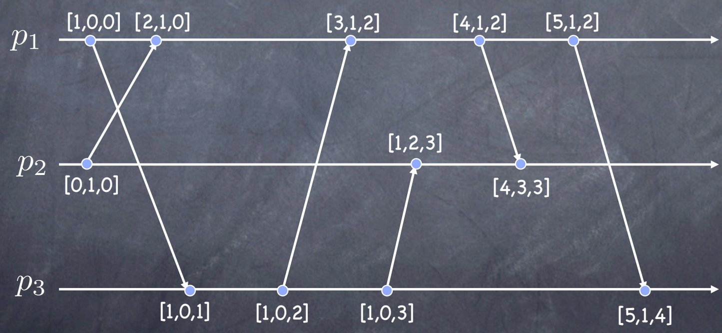
Update rules

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

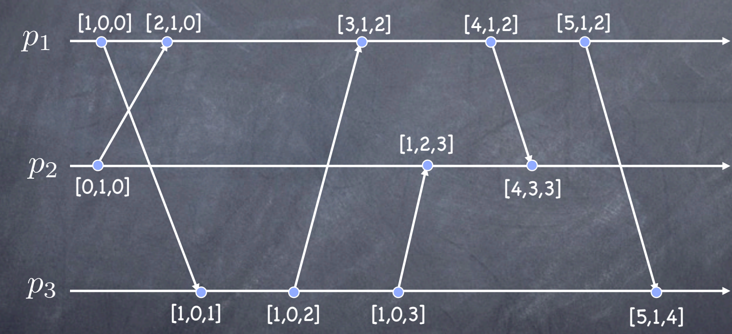


$$VC(e_i) := max(VC, TS(m))$$
$$VC(e_i)[i] := VC[i] + 1$$

Example



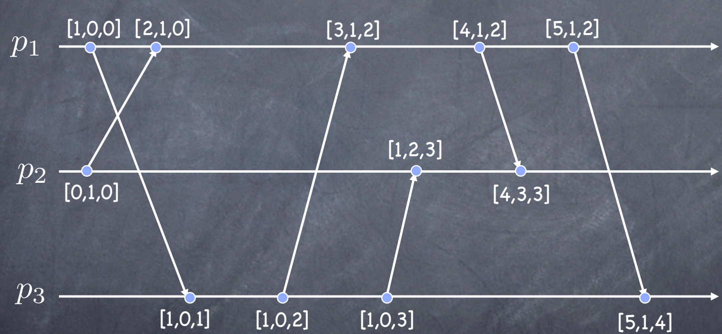
Operational interpretation



$VC(e_i)[i]$ =

$VC(e_i)[j]$ =

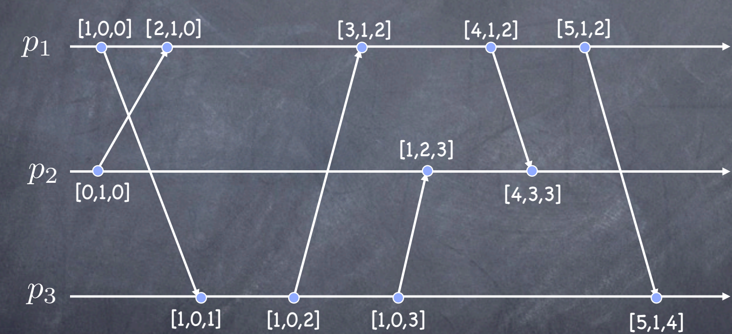
Operational interpretation



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

$VC(e_i)[j]$ =

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