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

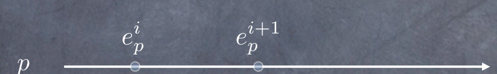
< Property of real time>
$4.\ e_i \rightarrow e_j \Rightarrow T(e_i) < T(e_j)$

Can the Clock Condition be implemented some other way?

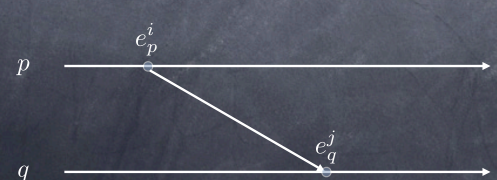
Lamport Clocks

Each process maintains a local variable LC

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

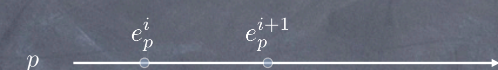


$LC(e_p^i) < LC(e_p^{i+1})$

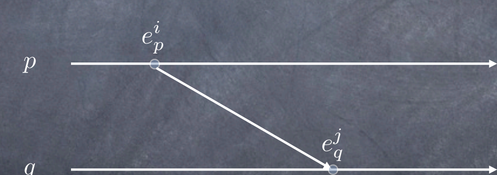


$LC(e_p^i) < LC(e_q^j)$

Increment Rules



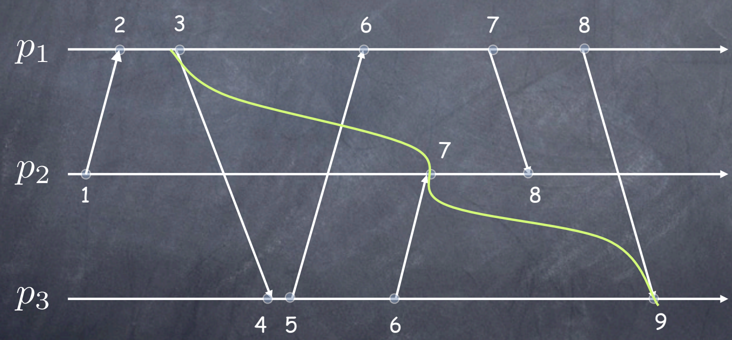
$LC(e_p^{i+1}) = LC(e_p^i) + 1$



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

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 <u>after</u> $LC = t$

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...

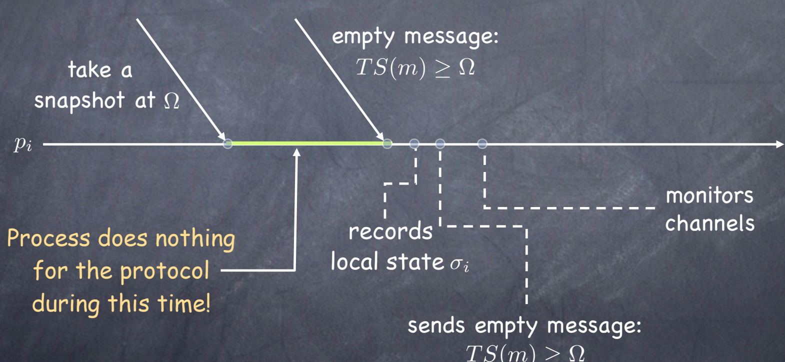
- Doing so assumes
  - upper bound on message delivery time
  - upper bound relative process speeds

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" <u>for the first time</u> 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

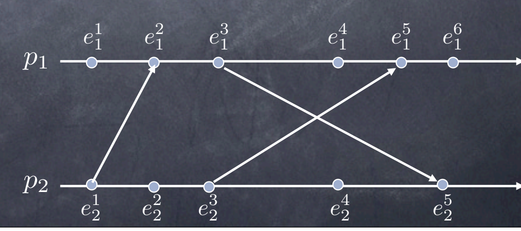
Snapshots: a perspective

- 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

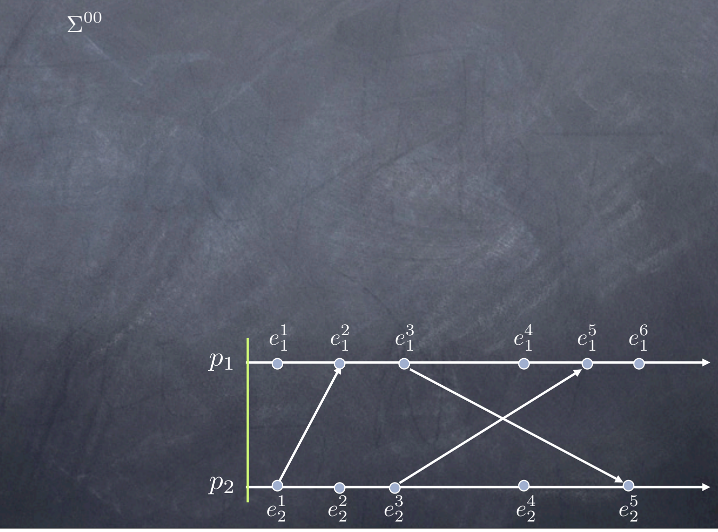
Snapshots: a perspective

- 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!

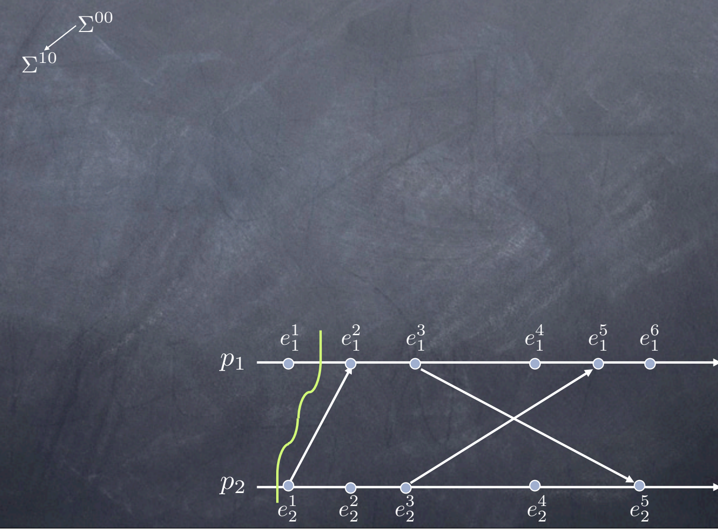
An Execution and its Lattice



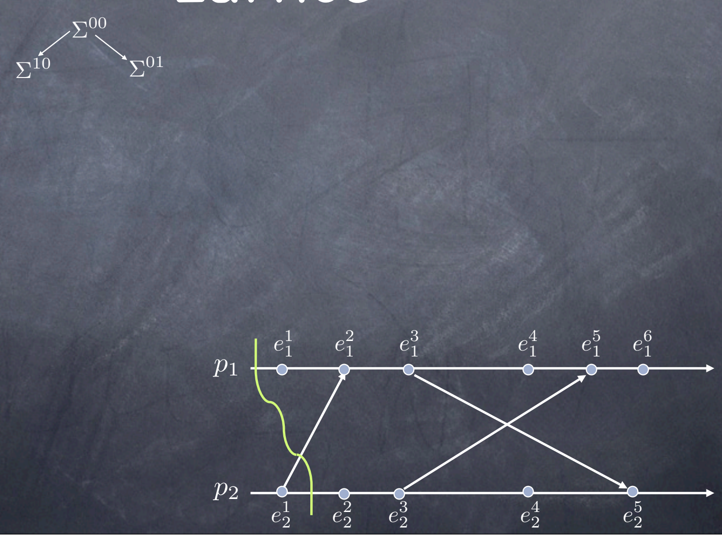
An Execution and its Lattice



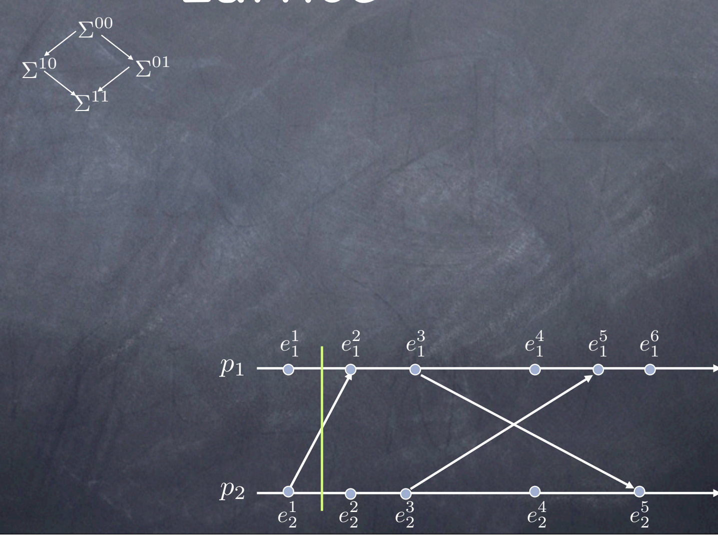
An Execution and its Lattice

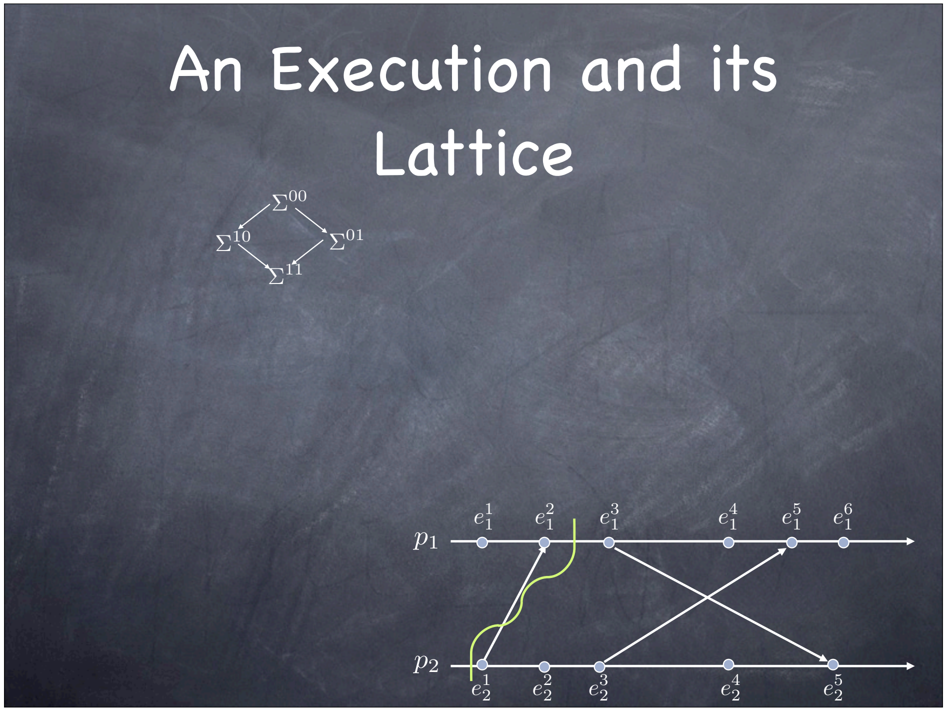


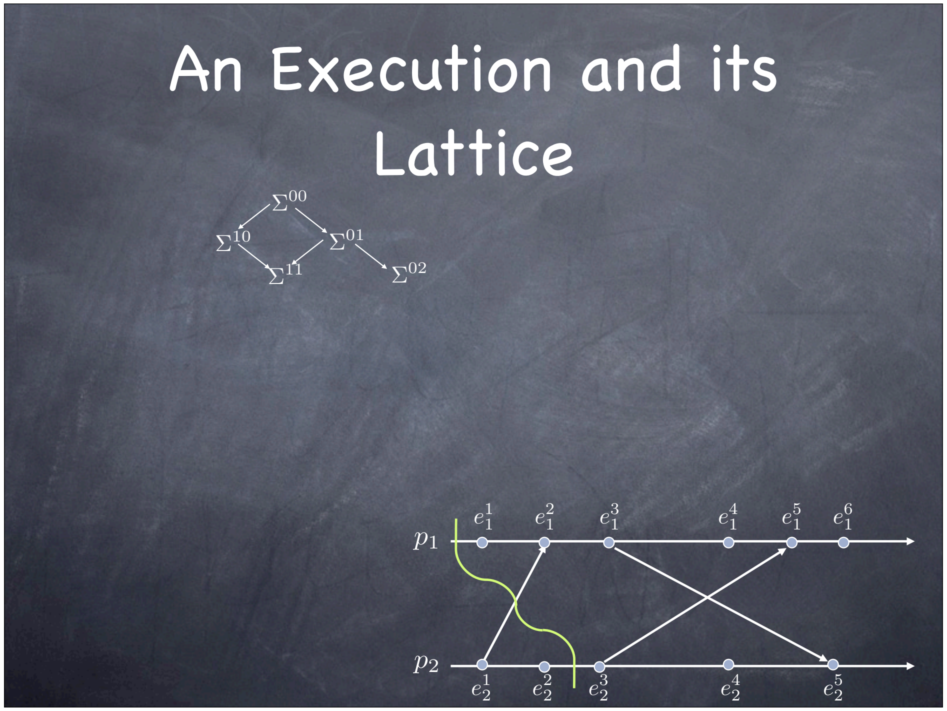
An Execution and its Lattice

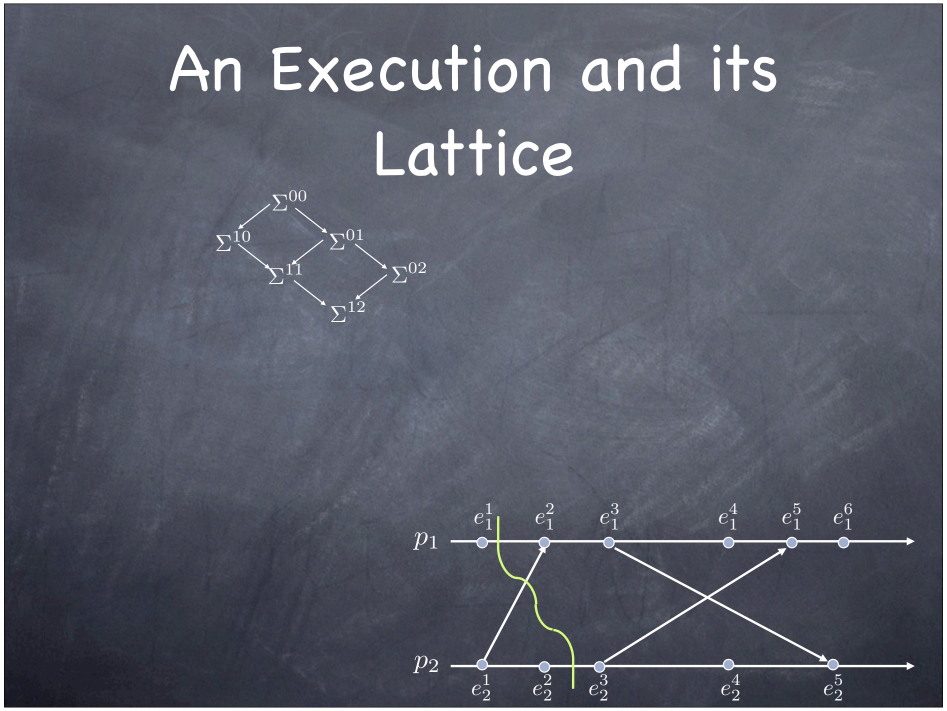


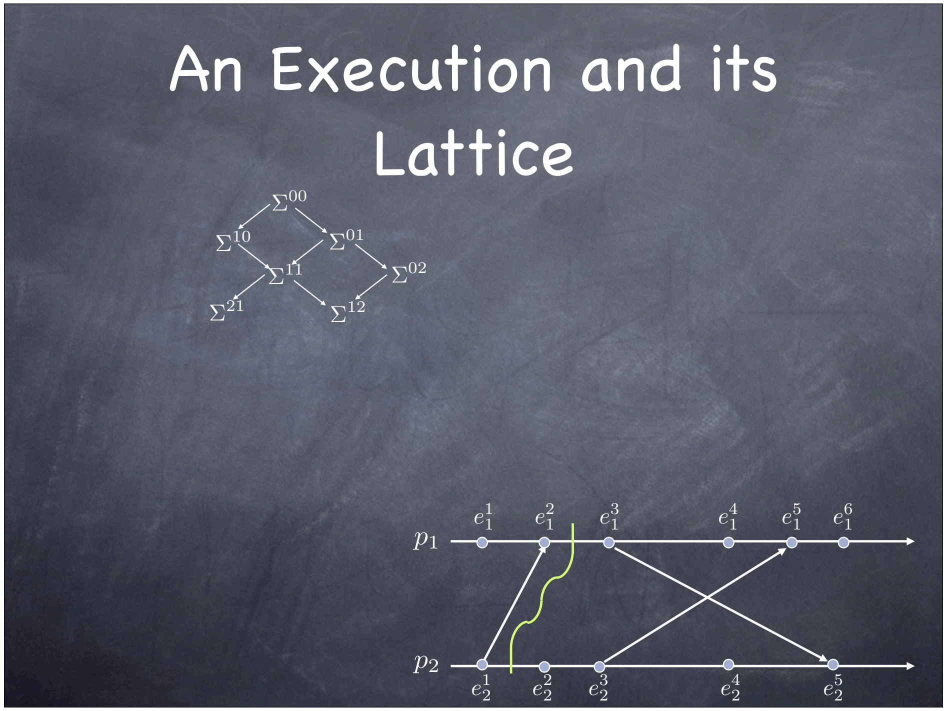
An Execution and its Lattice



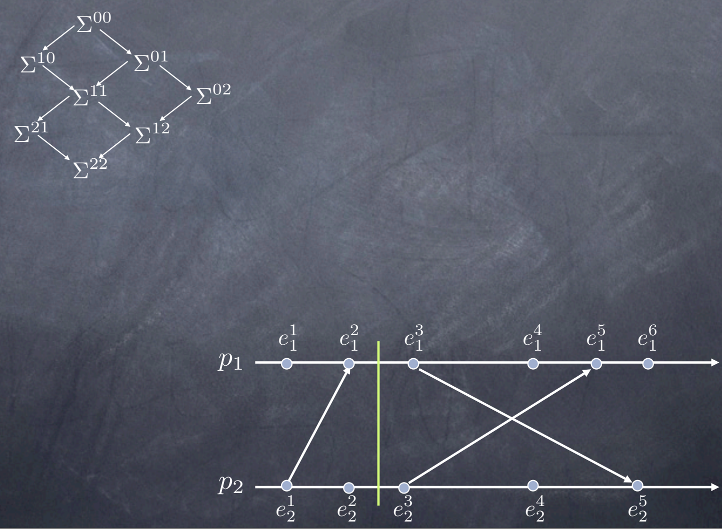

An Execution and its Lattice
Σ^{00}
Σ^{10}
Σ^{01}
Σ^{11}
p_1
p_2
e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
e_2^1 e_2^2 e_2^3 e_2^4 e_2^5


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

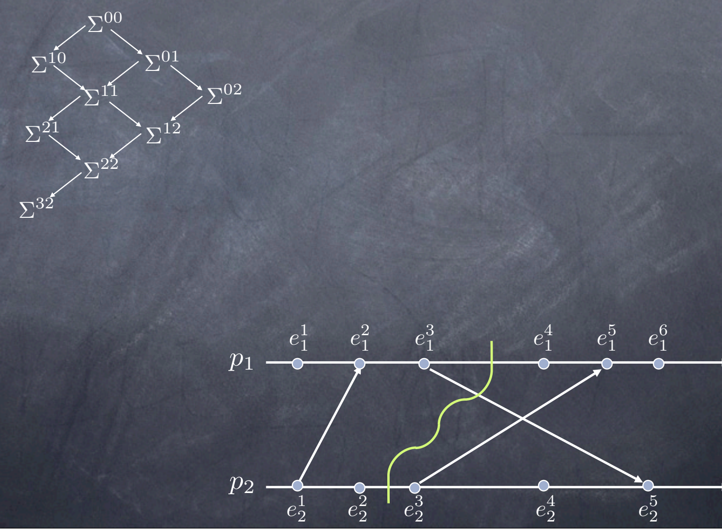

An Execution and its Lattice
Σ^{00}
Σ^{10}
Σ^{01}
Σ^{11}
Σ^{02}
Σ^{12}
p_1
p_2
e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
e_2^1 e_2^2 e_2^3 e_2^4 e_2^5


An Execution and its Lattice
Σ^{00}
Σ^{10}
Σ^{01}
Σ^{11}
Σ^{02}
Σ^{21}
Σ^{12}
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

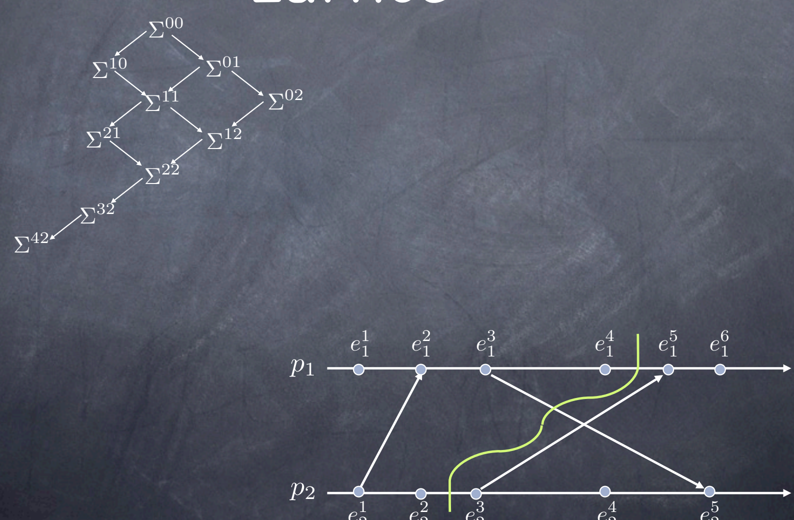
An Execution and its Lattice



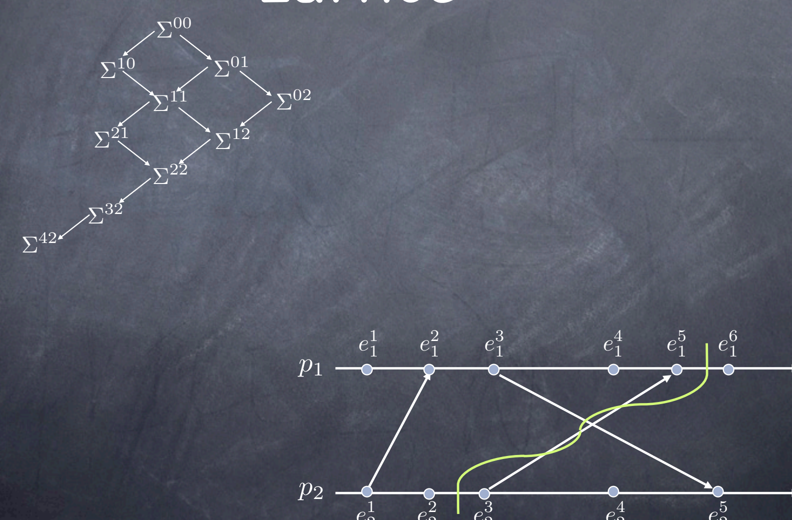
An Execution and its Lattice



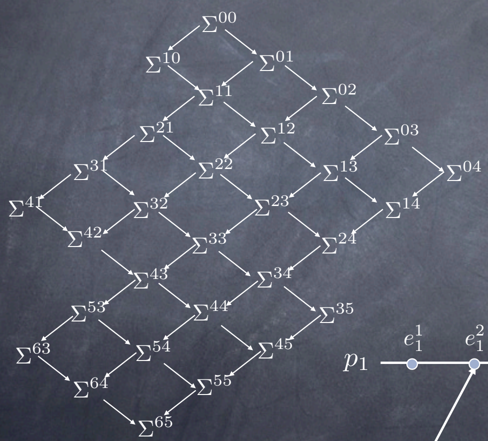
An Execution and its Lattice

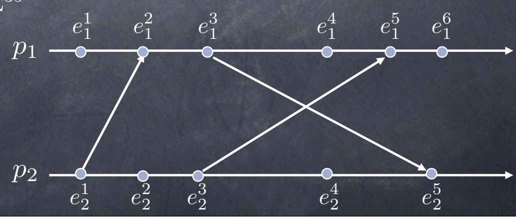


An Execution and its Lattice



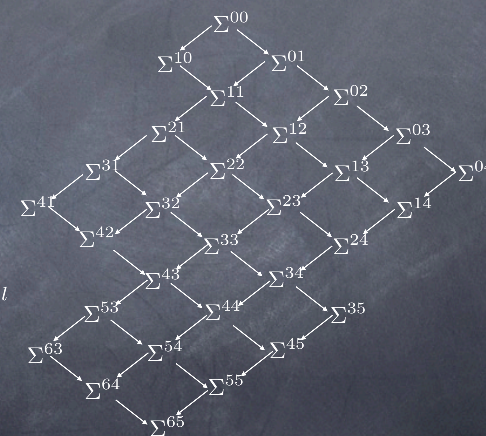
An Execution and its Lattice



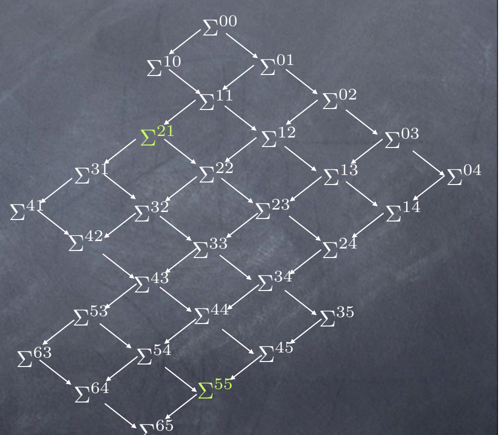


Reachability

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

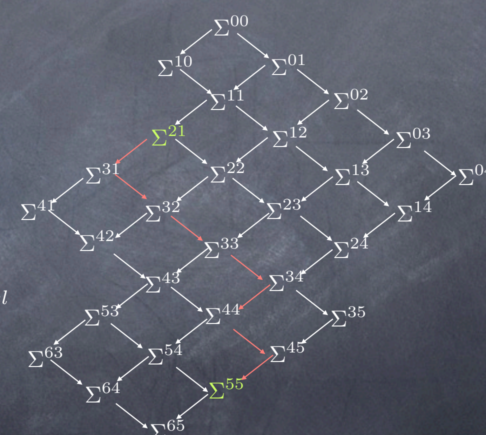


Reachability



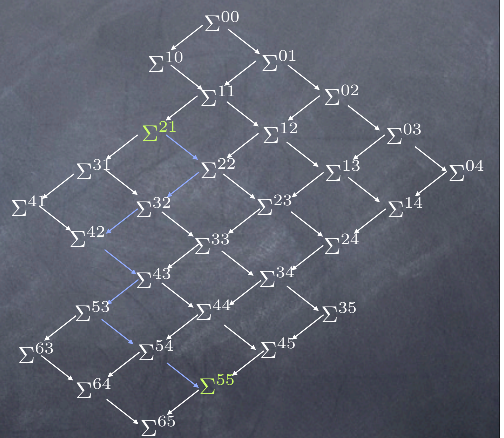
Σ^{kl} is reachable from Σ^{ij} if there is a path from Σ^{ij} to Σ^{kl} in the lattice

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 $\Sigma^i \rightsquigarrow_R \Sigma^f$

- 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 $\Sigma^i \rightsquigarrow_R \Sigma^f$

- 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

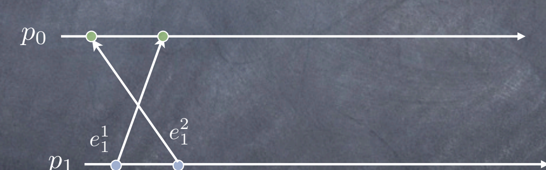
- An observation puts no constraint on the order in which the monitor receives notifications

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:

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

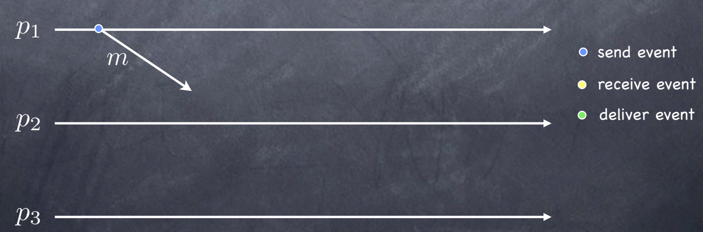
Causal delivery

FIFO delivery guarantees:

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

Causal delivery generalizes FIFO:

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



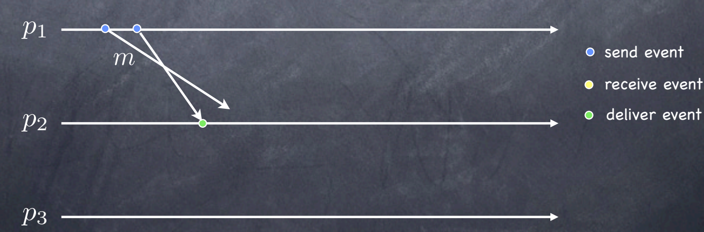
Causal delivery

FIFO delivery guarantees:

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

Causal delivery generalizes FIFO:

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



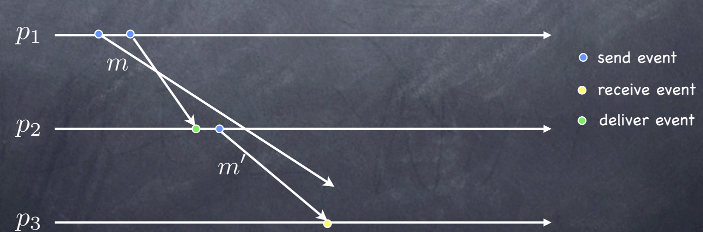
Causal delivery

FIFO delivery guarantees:

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

Causal delivery generalizes FIFO:

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



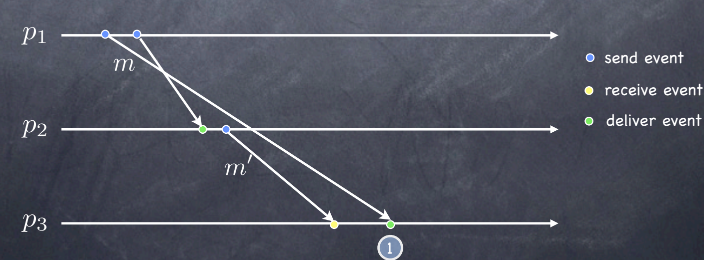
Causal delivery

FIFO delivery guarantees:

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

Causal delivery generalizes FIFO:

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



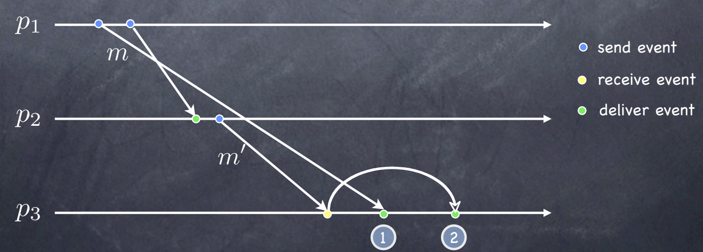
Causal delivery

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

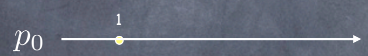
DR1: At time t, p_0 delivers all messages it received with timestamp up to $t - \Delta$ in increasing timestamp order

Causal Delivery with Lamport Clocks

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

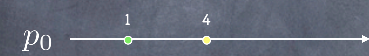
Causal Delivery with Lamport Clocks

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



Causal Delivery with Lamport Clocks

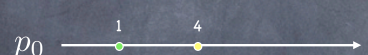
DR1.1: Deliver all received messages in increasing (logical clock) timestamp order.



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:

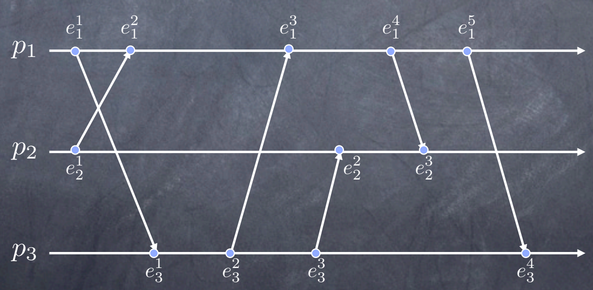
$$e \rightarrow e' \equiv SC(e) < SC(e')$$

Causal Histories

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

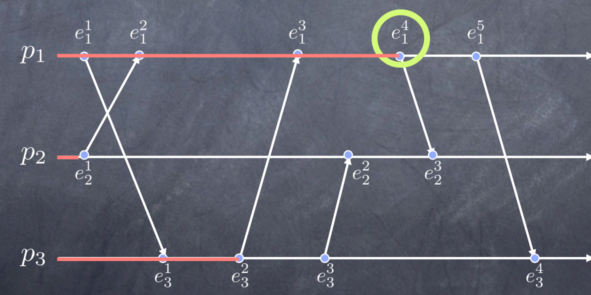
Causal Histories

- 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

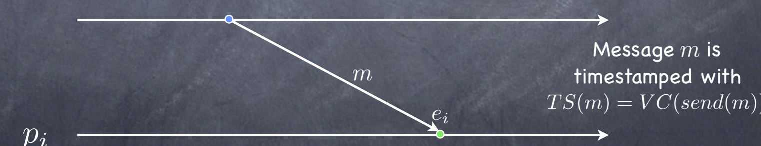
- Consider $\theta_i(e)$, the projection of $\theta(e)$ on p_i
- $\theta_i(e)$ is a prefix of h^i: $\theta_i(e) = h_i^{k_i}$ – it can be encoded using k_i
- $\theta(e) = \theta_1(e) \cup \theta_2(e) \cup \ldots \cup \theta_n(e)$ can be encoded using $k_1, k_2, \ldots, k_n$

Represent θ using an n-vector VC such that

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

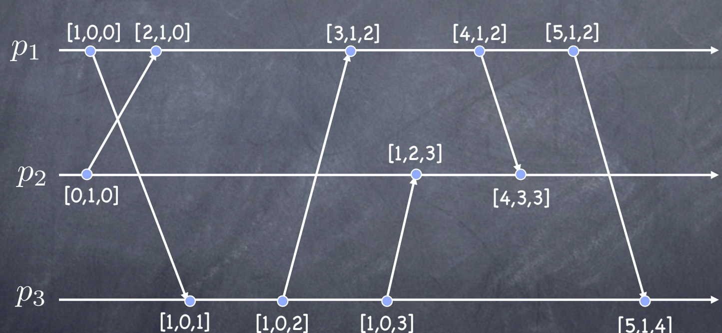
Update rules

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

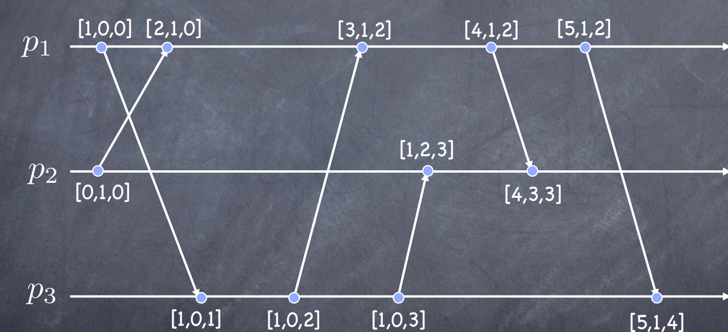


$$VC(e_i) := max(VC, TS(m))$$
$$VC(e_i)[i] := VC[i] + 1$$

Example



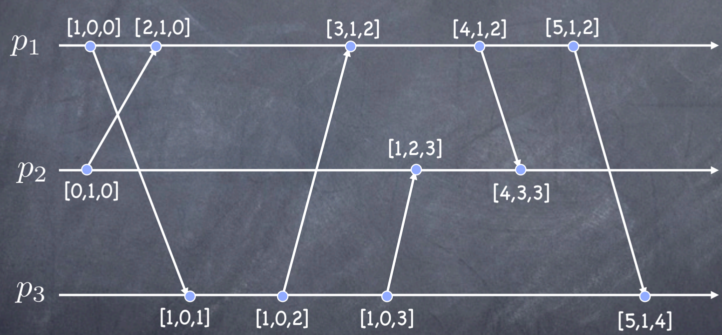
Operational interpretation



$VC(e_i)[i]$ =

$VC(e_i)[j]$ =

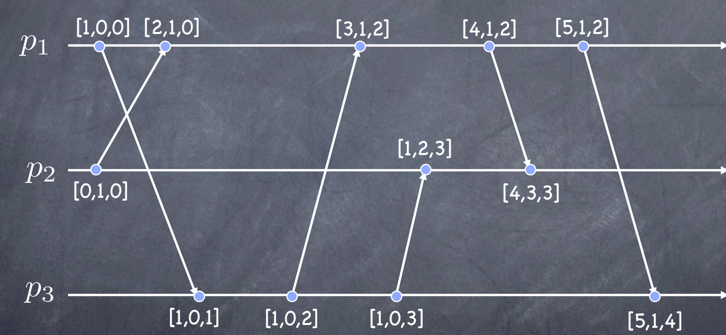
Operational interpretation



$VC(e_i)[i]$ = 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

VC properties: event ordering

Given two vectors V and V', less than is defined as:

$$V < V' \equiv (V \neq V') \wedge (\forall k : 1 \leq k \leq n : V[k] \leq V'[k])$$

- Strong Clock Condition: $e \rightarrow e' \equiv VC(e) < VC(e')$
- Simple Strong Clock Condition:
  Given e_i of p_i and e_j of p_j, where $i \neq j$
  $$e_i \rightarrow e_j \equiv VC(e_i)[i] \leq VC(e_j)[i]$$
- Concurrency
  Given e_i of p_i and e_j of p_j, where $i \neq j$
  $$e_i \parallel e_j \equiv (VC(e_i)[i] > VC(e_j)[i]) \wedge (VC(e_j)[j] > VC(e_i)[j])$$

VC properties: consistency

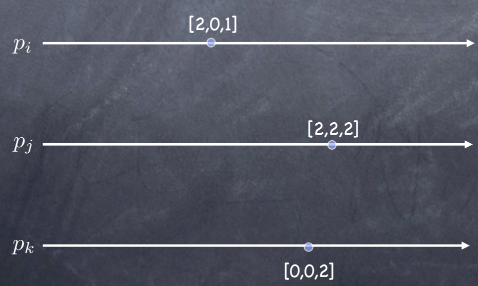
- Pairwise inconsistency
  Events e_i of p_i and e_j of p_j $(i \neq j)$ are pairwise inconsistent (i.e. can't be on the frontier of the same consistent cut) if and only if
  $$(VC(e_i)[i] < VC(e_j)[i]) \vee (VC(e_j)[j] < VC(e_i)[j])$$
- Consistent Cut
  A cut defined by $(c_1, \ldots, c_n)$ is consistent if and only if
  $$\forall i, j : 1 \leq i \leq n, 1 \leq j \leq n : (VC(e_i^{c_i})[i] \geq VC(e_j^{c_j})[i])$$

VC properties: weak gap detection

- Weak gap detection

  Given e_i of p_i and e_j of p_j, if $VC(e_i)[k] < VC(e_j)[k]$ for some $k \neq j$, then there exists e_k s.t

  $$\neg(e_k \to e_i) \wedge (e_k \to e_j)$$

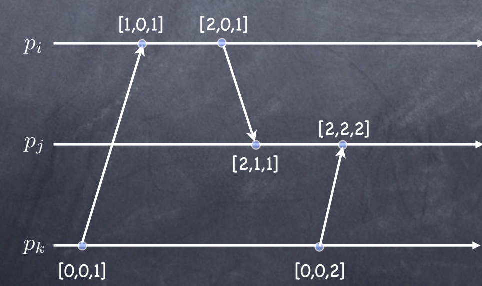


VC properties: weak gap detection

- Weak gap detection

  Given e_i of p_i and e_j of p_j, if $VC(e_i)[k] < VC(e_j)[k]$ for some $k \neq j$, then there exists e_k s.t

  $$\neg(e_k \to e_i) \wedge (e_k \to e_j)$$



VC properties: strong gap detection

- Weak gap detection

  Given e_i of p_i and e_j of p_j, if $VC(e_i)[k] < VC(e_j)[k]$ for some $k \neq j$, then there exists e_k s.t

  $$\neg(e_k \to e_i) \wedge (e_k \to e_j)$$

- Strong gap detection

  Given e_i of p_i and e_j of p_j, if $VC(e_i)[i] < VC(e_j)[i]$ then there exists e_i' s.t.

  $$(e_i \to e_i') \wedge (e_i' \to e_j)$$

VCs for Causal Delivery

- Each process increments the local component of its VC only for events that are notified to the monitor
- Each message notifying event e is timestamped with $VC(e)$
- The monitor keeps all notification messages in a set M

Stability

Suppose p_0 has received m_j from p_j.
When is it safe for p_0 to deliver m_j?

Stability

Suppose p_0 has received m_j from p_j.
When is it safe for p_0 to deliver m_j?

- There is no earlier message in M

  $\forall m \in M : \neg(m \rightarrow m_j)$

Stability

Suppose p_0 has received m_j from p_j.
When is it safe for p_0 to deliver m_j?

- There is no earlier message in M

  $\forall m \in M : \neg(m \rightarrow m_j)$

- There is no earlier message from p_j

$TS(m_j)[j] = 1+$ no. of p_j messages delivered by p_0

Stability

Suppose p_0 has received m_j from p_j.
When is it safe for p_0 to deliver m_j?

- There is no earlier message in M

  $\forall m \in M : \neg(m \rightarrow m_j)$

- There is no earlier message from p_j

$TS(m_j)[j] = 1+$ no. of p_j messages delivered by p_0

- There is no earlier message m_k'' from p_k, $k \neq j$

  see next slide...

Checking for m''_k

- Let m'_k be the last message p_0 delivered from p_k
- By strong gap detection, m''_k exists only if

$$TS(m'_k)[k] < TS(m_j)[k]$$

- Hence, deliver m_j as soon as

$$\forall k : TS(m'_k)[k] \geq TS(m_j)[k]$$

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

DR3: Deliver m from p_j as soon as both of the following conditions are satisfied:

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

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

Properties

Property: a predicate that is evaluated over a run of the program

"every message that is received was previously sent"

Not everything you may want to say about a program is a property:

"the program sends an average of 50 messages in a run"

Safety properties

- "nothing bad happens"
  - no more than k processes are simultaneously in the critical section
  - messages that are delivered are delivered in causal order
  - Windows never crashes
- A safety property is "prefix closed":
  - if it holds in a run, it holds in every prefix

Liveness properties

- "something good eventually happens"
  - a process that wishes to enter the critical section eventually does so
  - some message is eventually delivered
  - Windows eventually boots
- Every run can be extended to satisfy a liveness property
  - if it does not hold in a prefix of a run, it does not mean it may not hold eventually

A really cool theorem

Every property is a combination of a safety property and a liveness property

(Alpern & Schneider)