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}



...but what if that message never comes?

Two options

- Wait for a full Δ after t_{ss}

Two options

- Wait for a full Δ after t_{ss}
- Every process flushes its outgoing channels
  - Sends an empty message with timestamp greater than t_{ss}



works also in an asynchronous system!

Two options

- Wait for a full Δ after t_{ss}
- Every process flushes its outgoing channels
  - Sends an empty message with timestamp greater than 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_i

   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>

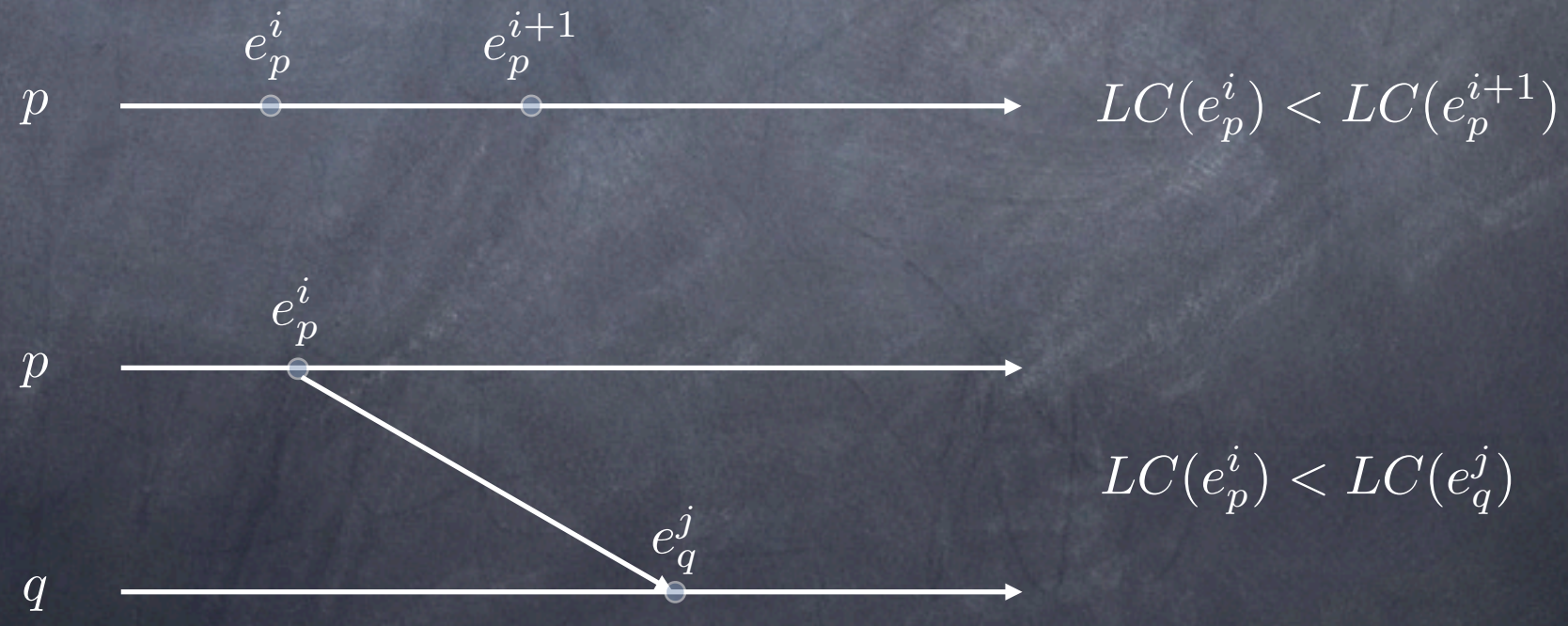
$4.\ e_i \rightarrow e_j \Rightarrow T(e_i) < T(e_j)$

Can the Clock Condition be implemented some other way?

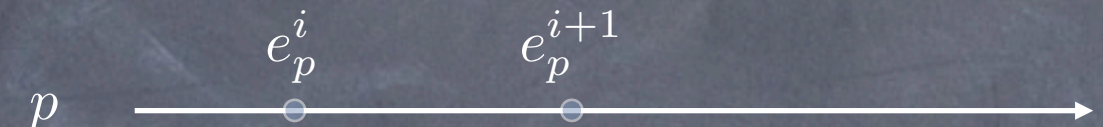
Lamport Clocks

Each process maintains a local variable LC

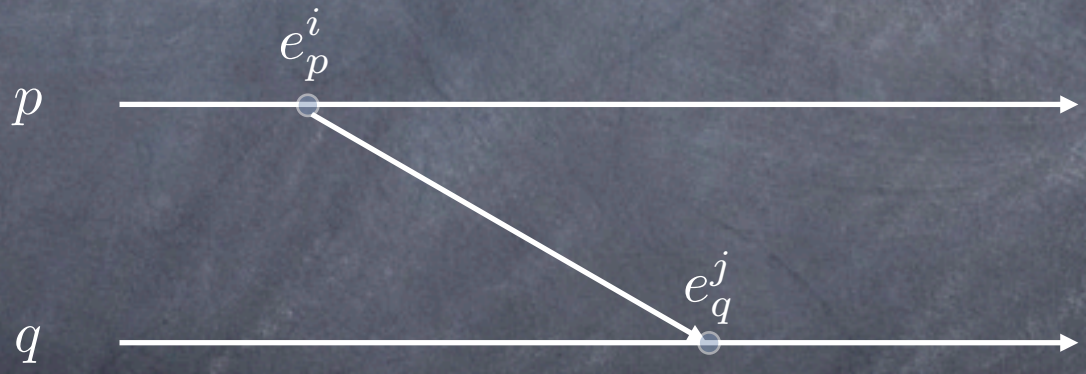
$LC(e) \equiv$ value of LC for event e



Increment Rules



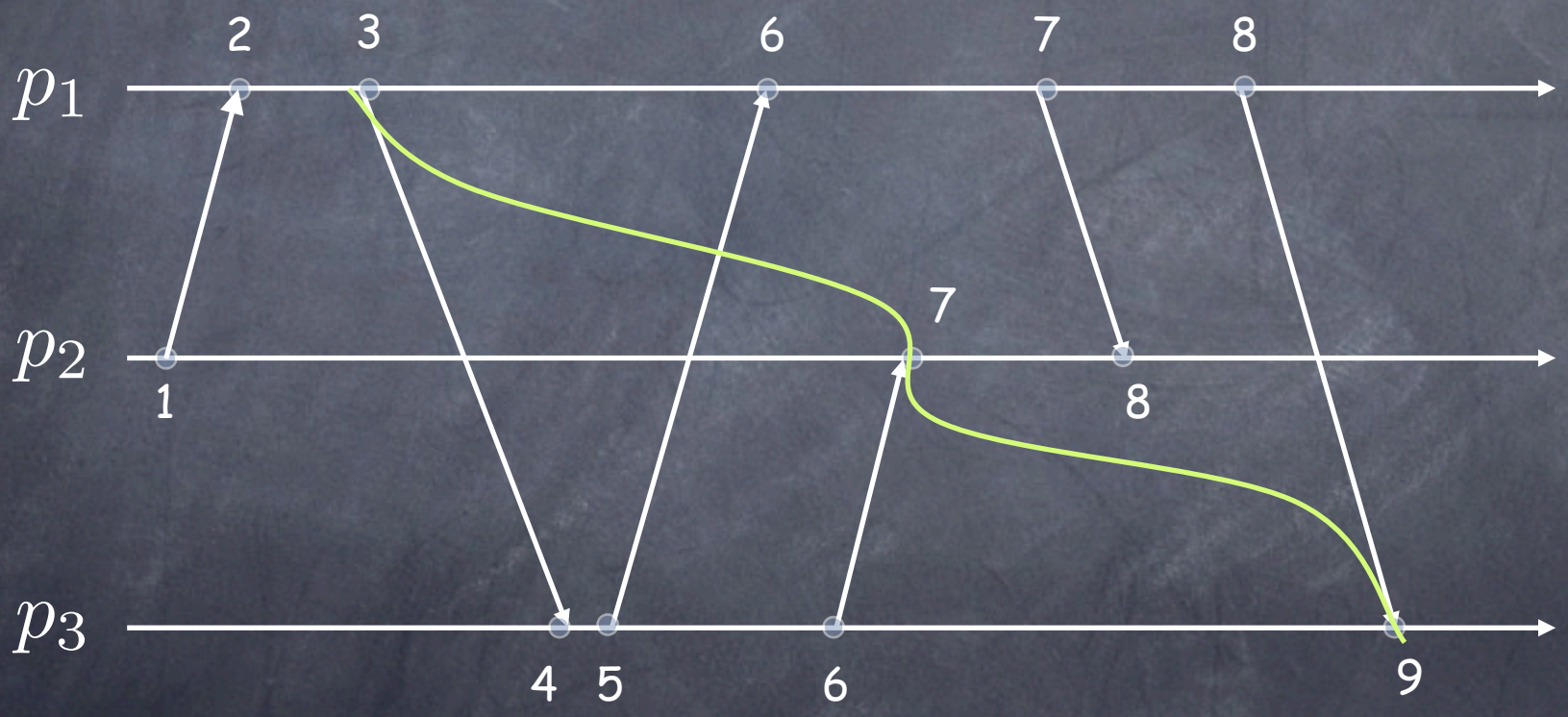
$$LC(e_p^{i+1}) = LC(e_p^i) + 1$$



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

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

Space-Time Diagrams and Logical Clocks



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_i

   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 (logical) timestamp greater than or equal to t_{ss}

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> $LC = t$

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: consider next event e:

- if e is internal/send and $LC = t - 2$
  - execute e and then S



A subtle problem

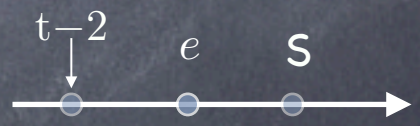
when $LC = t$ do S

doesn't make sense for Lamport clocks!

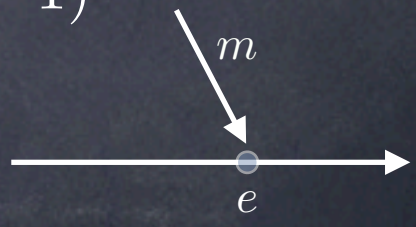
- there is no guarantee that LC will ever be t
- S is anyway executed <u>after</u> $LC = t$

Fixes: consider next event e:

- 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...

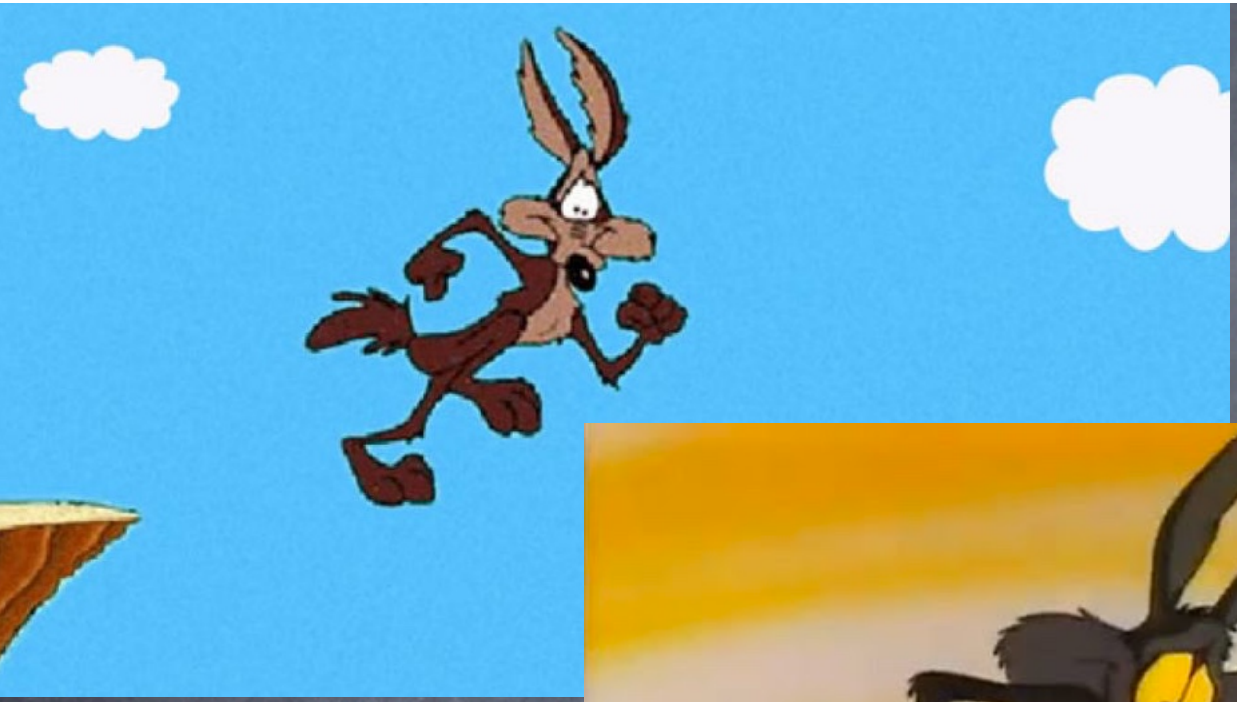
An obvious problem

- 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...



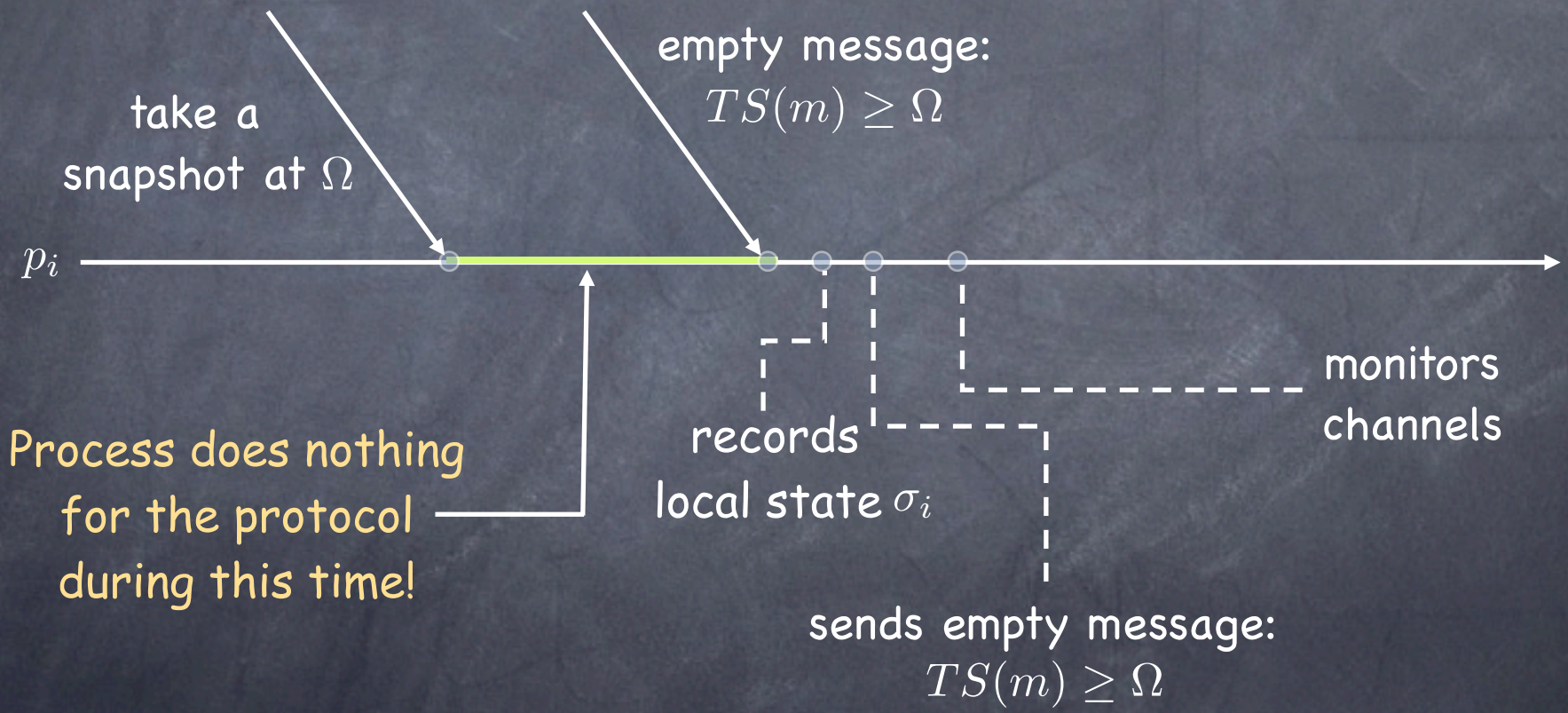



Don't
look down!!

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.

A bit of intuition

- Consider m from p_1 to p_2
- Suppose $receive(m)$ is in the snapshot, but $send(m)$ is not
- Can you draw a contradiction?

Key observation:

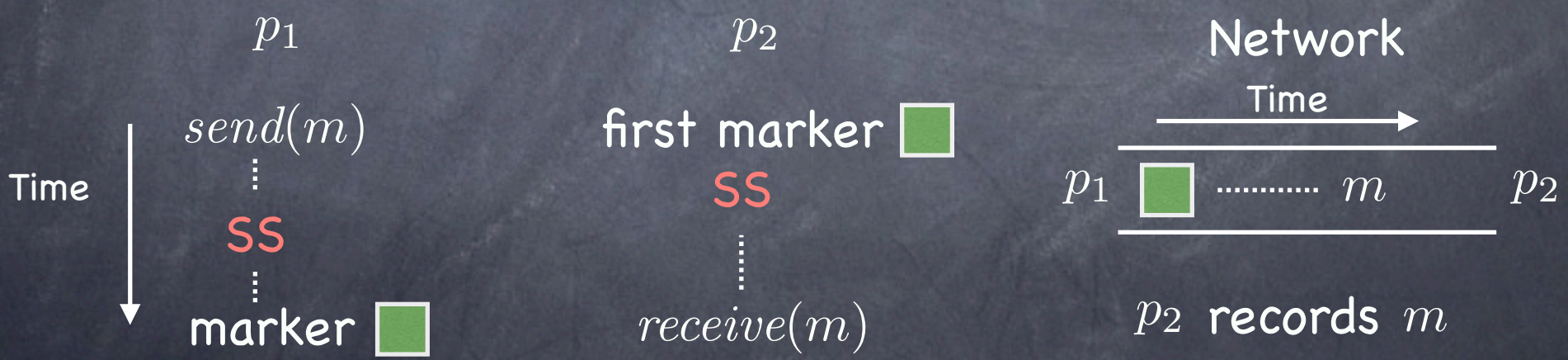
"take a snapshot" marker separates
"before SS" messages from "after SS" messages

Case I

- Consider m from p_1 to p_2

e_1 ⟶ e_2

- Suppose $send(m)$ is in the snapshot, but $receive(m)$ is not



$e_1 \in C$

$e_2 \notin C$

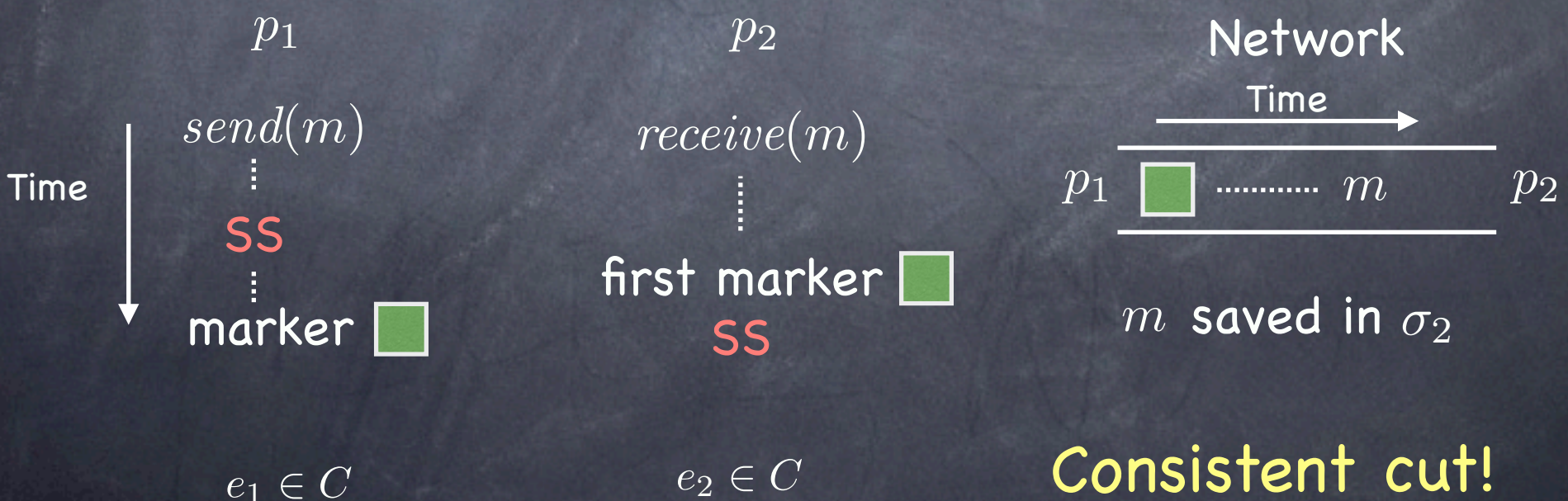
Consistent cut!

Case II

- Consider m from p_1 to p_2

e_1 ⟶ e_2

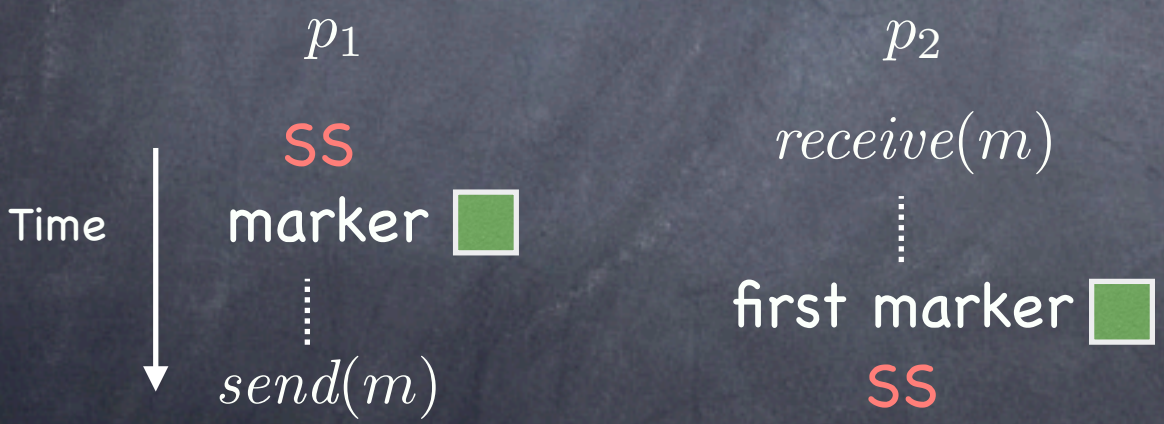
- Suppose $send(m)$ is in the snapshot, and so is $receive(m)$

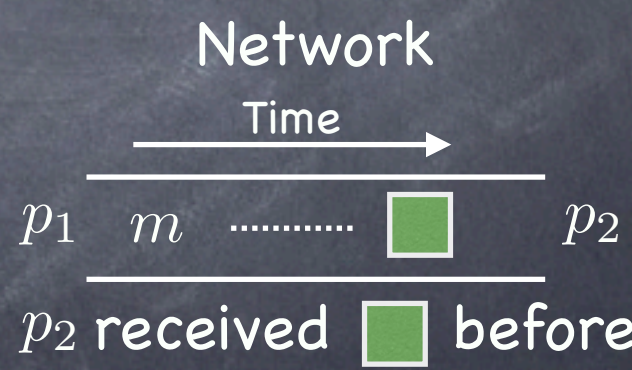


Case III

- Consider m from p_1 to p_2
- Suppose $receive(m)$ is in the snapshot, but $send(m)$ is not

e_2 ← e_1





p_2 received before receiving m: hence m is not in p_2's snapshot!

Case III is impossible!

Snapshots: a perspective

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

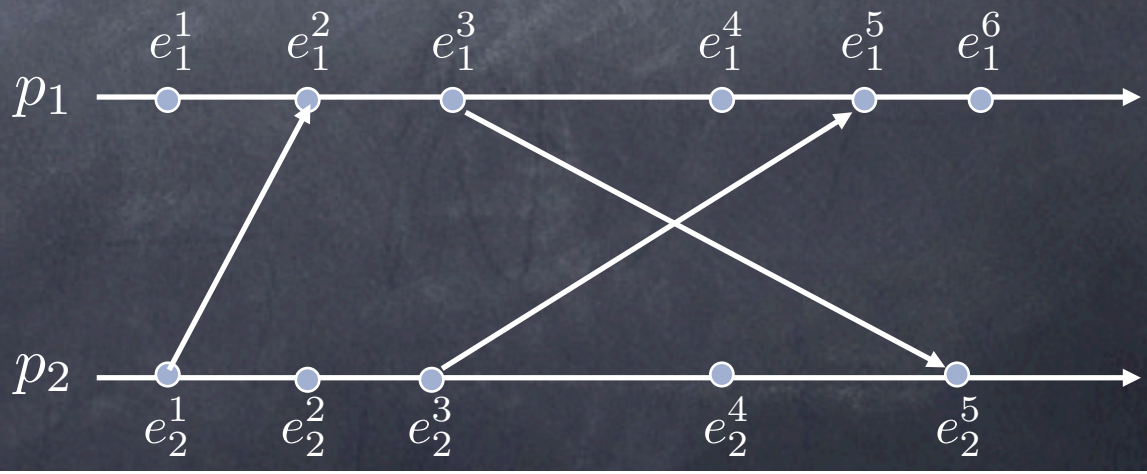
Snapshots: a perspective

- 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

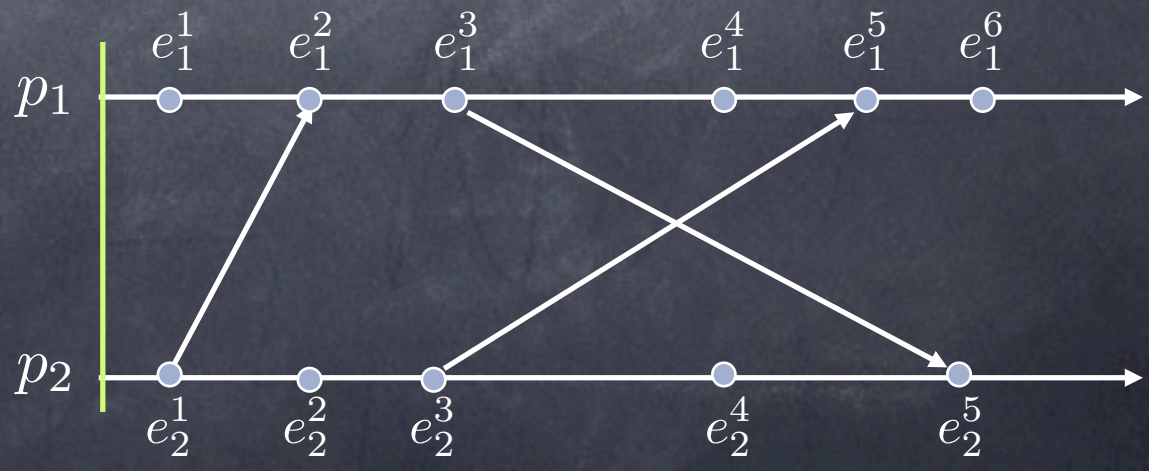
- The global state Σ^s saved by the snapshot protocol is a consistent global state
- But did it ever occur during the computation?
  - a distributed computation provides only a partial order of events
  - many total orders (runs) are compatible with that partial order
  - all we know is that Σ^s could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

An Execution and its Lattice

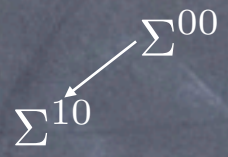


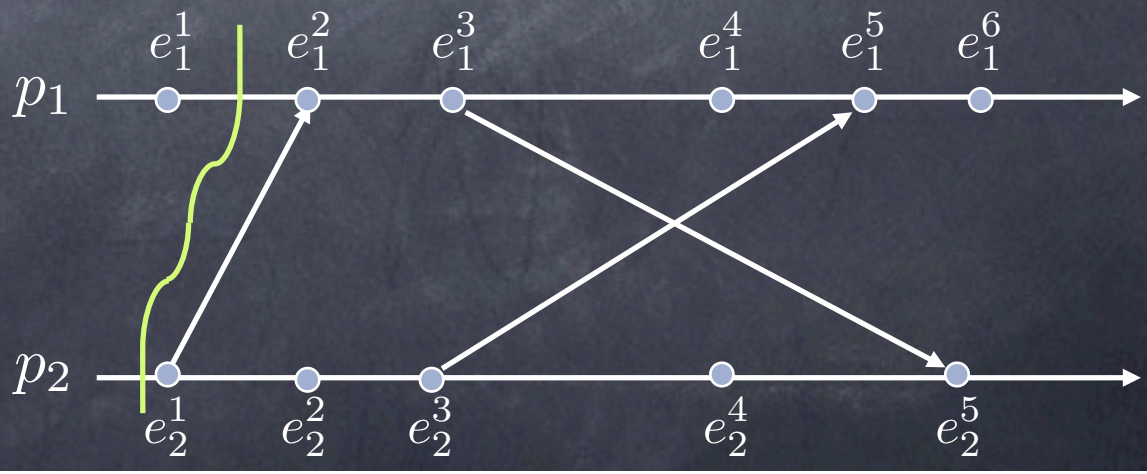
An Execution and its Lattice

Σ^{00}

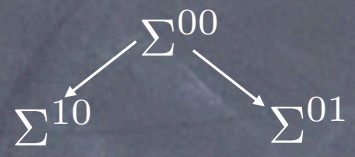


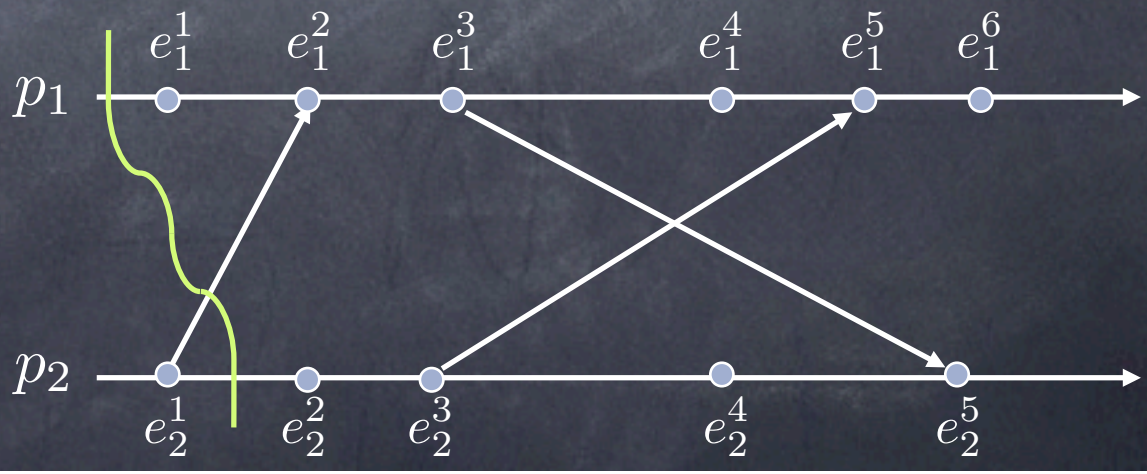
An Execution and its Lattice



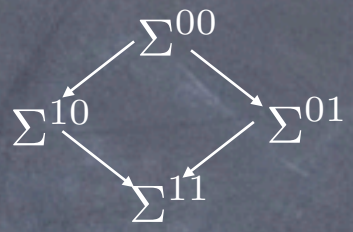


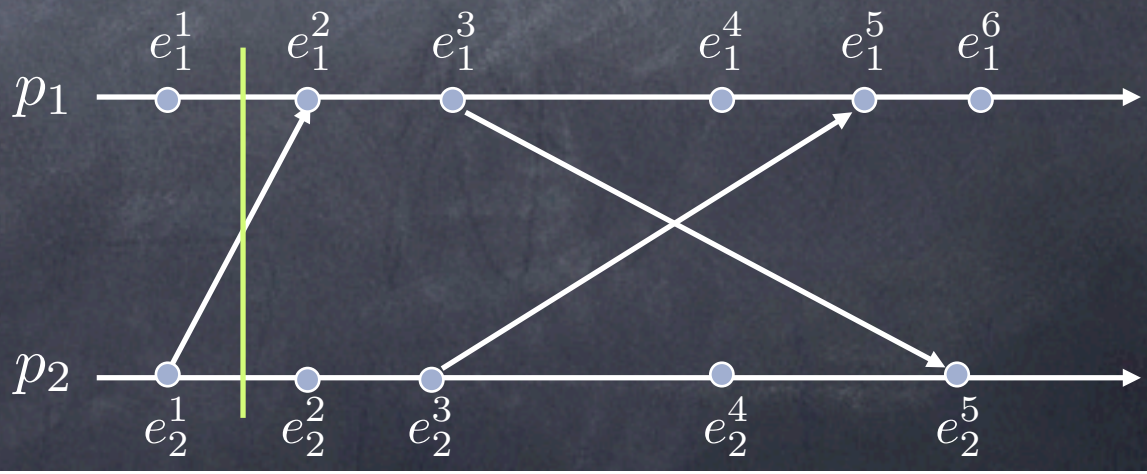
An Execution and its Lattice



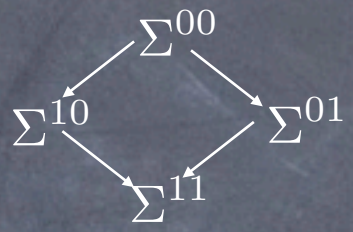


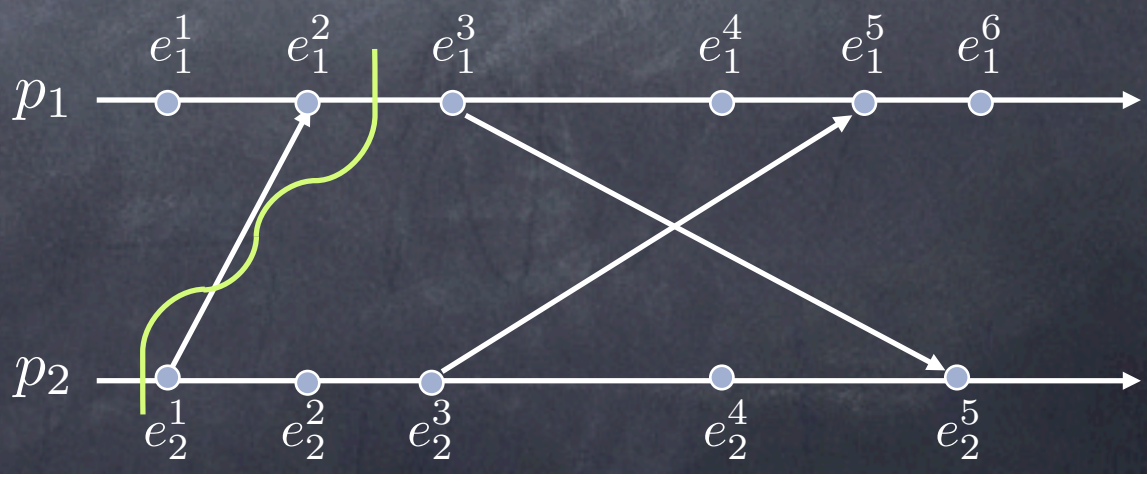
An Execution and its Lattice



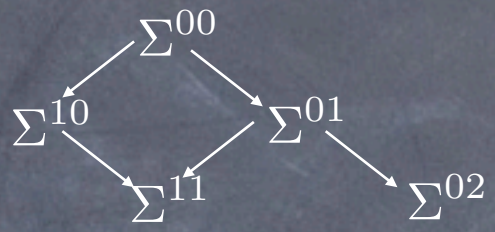


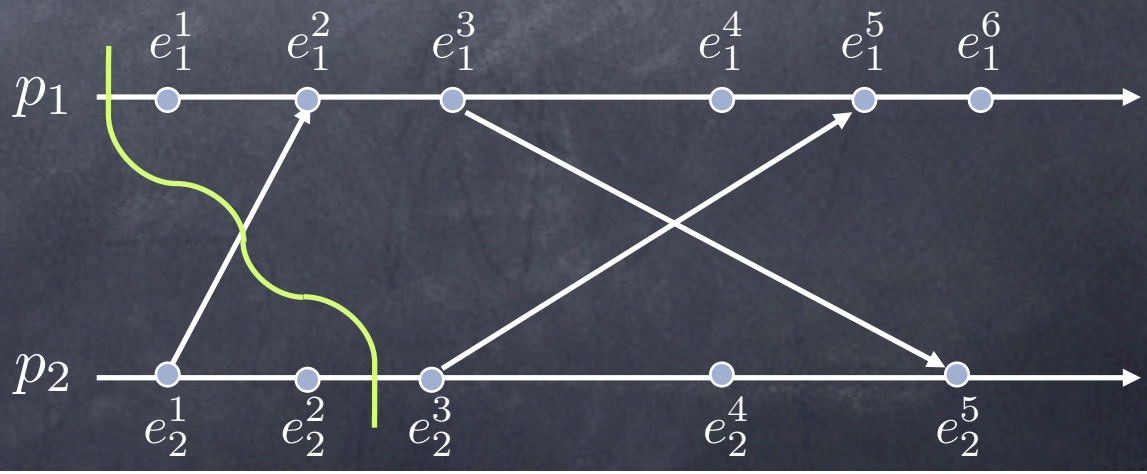
An Execution and its Lattice



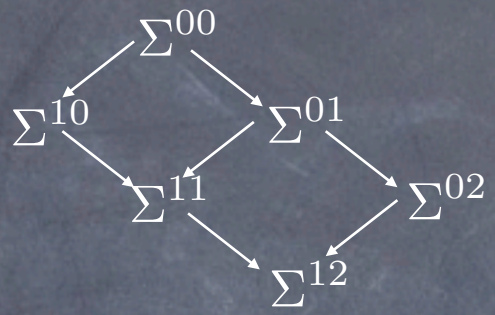


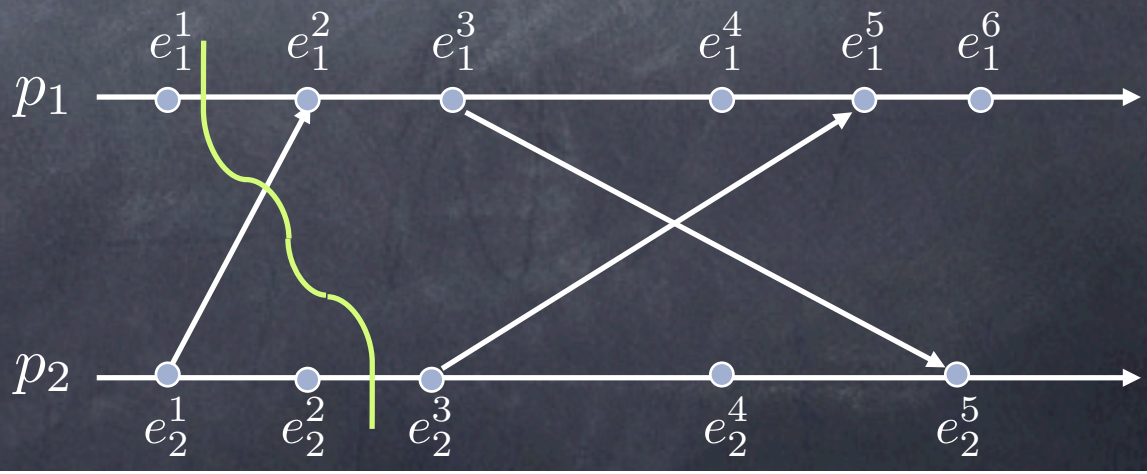
An Execution and its Lattice



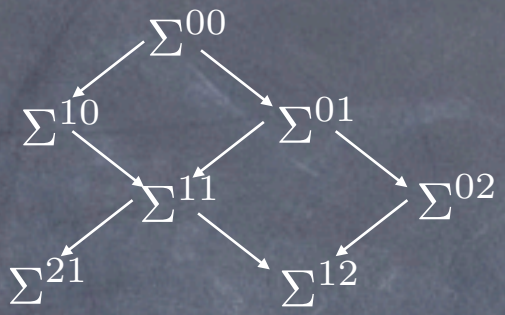


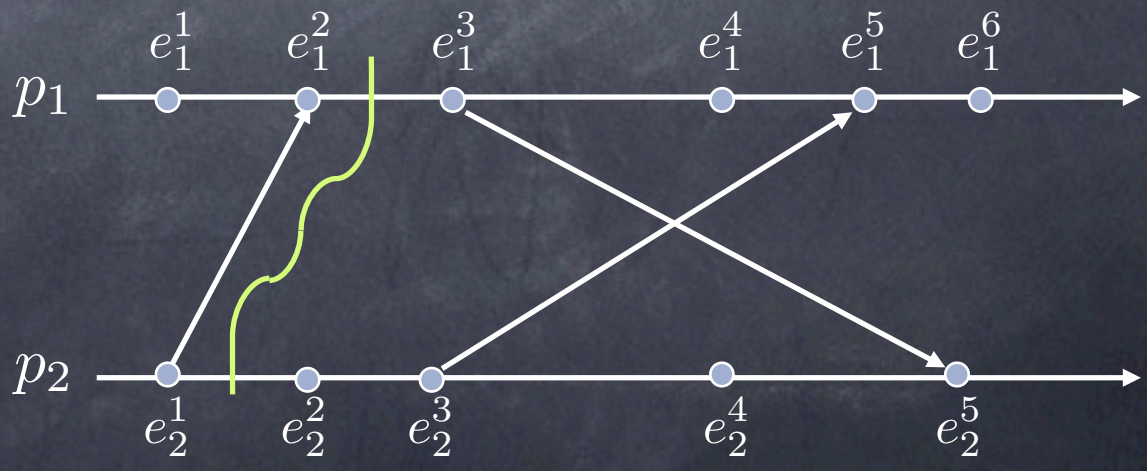
An Execution and its Lattice



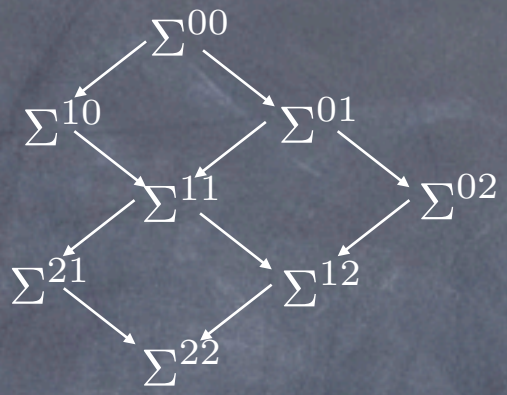


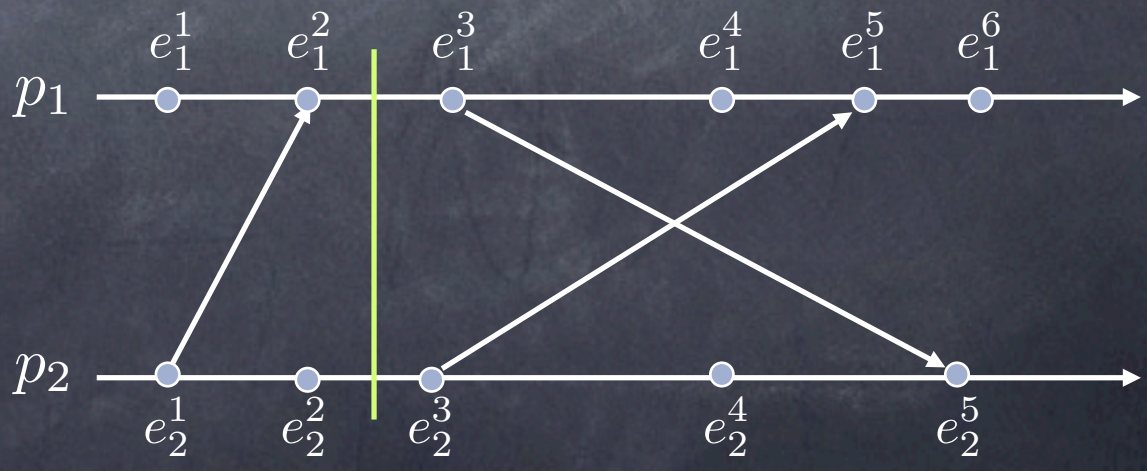
An Execution and its Lattice



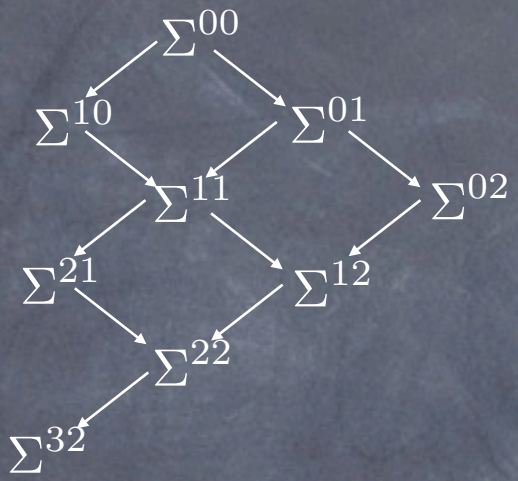


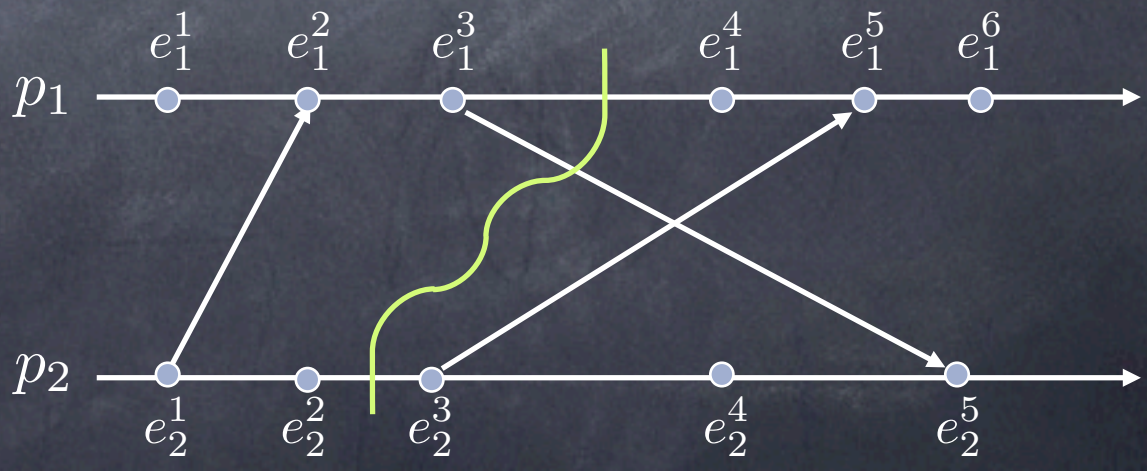
An Execution and its Lattice



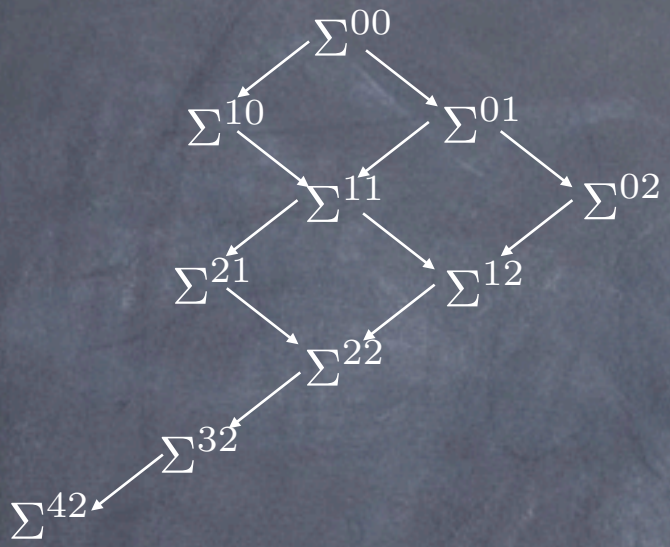


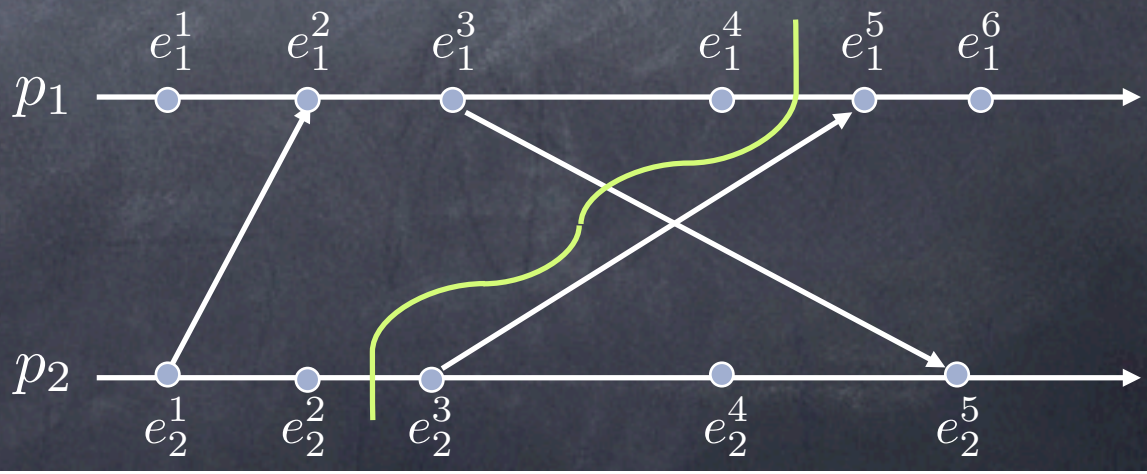
An Execution and its Lattice



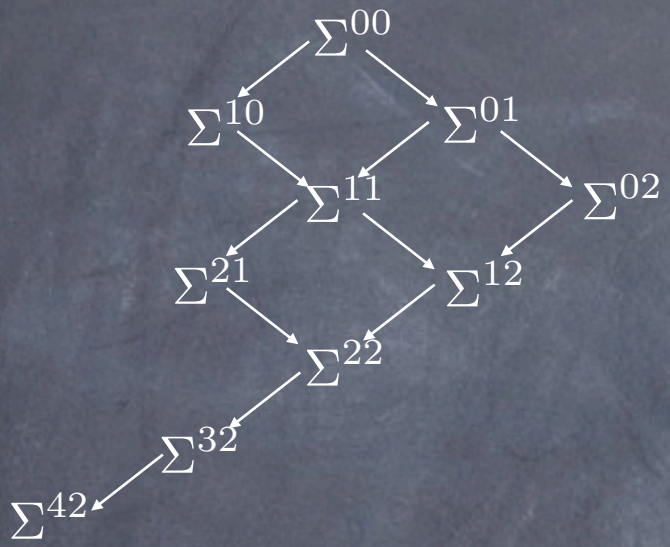


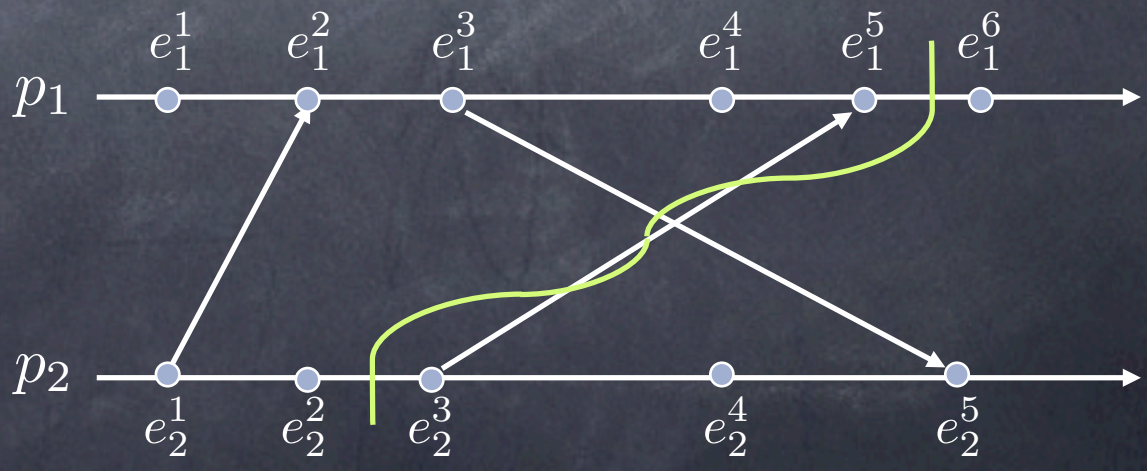
An Execution and its Lattice



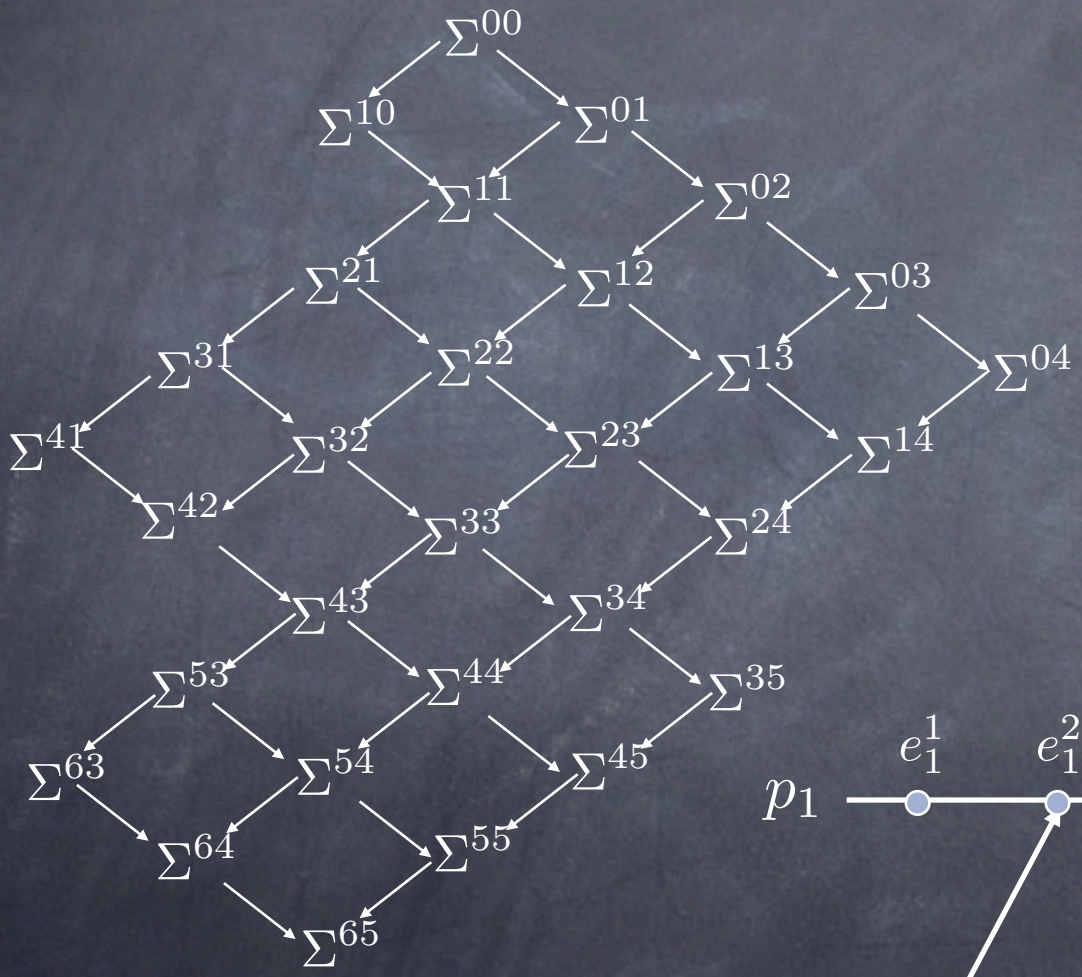


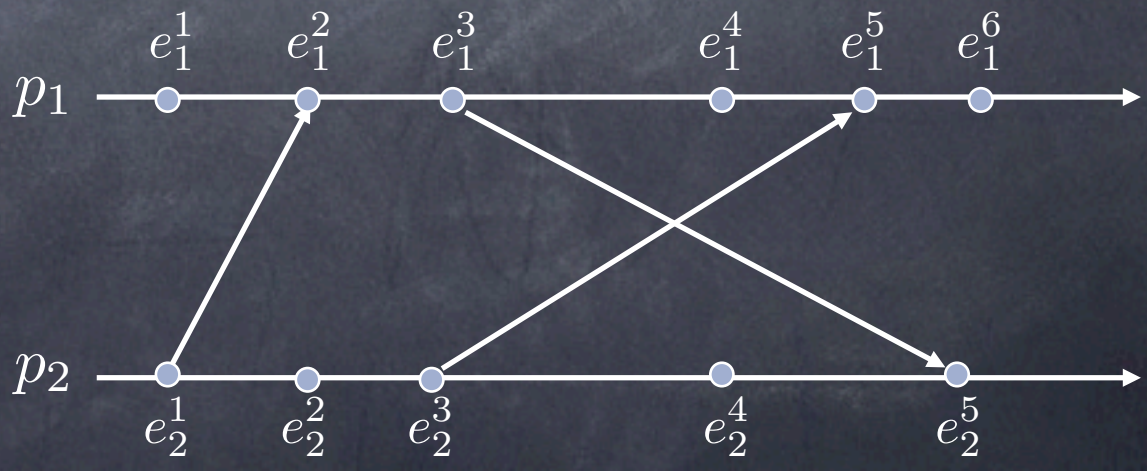
An Execution and its Lattice





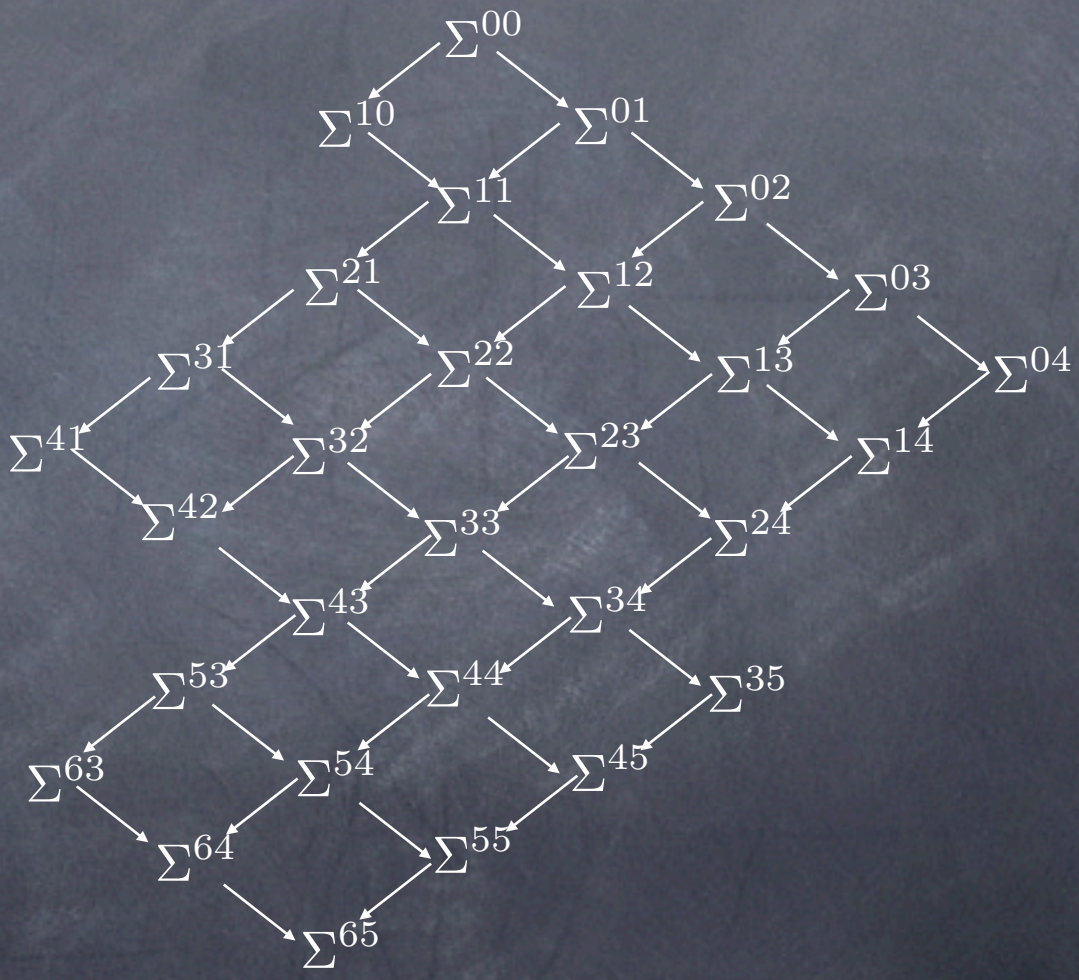
An Execution and its Lattice





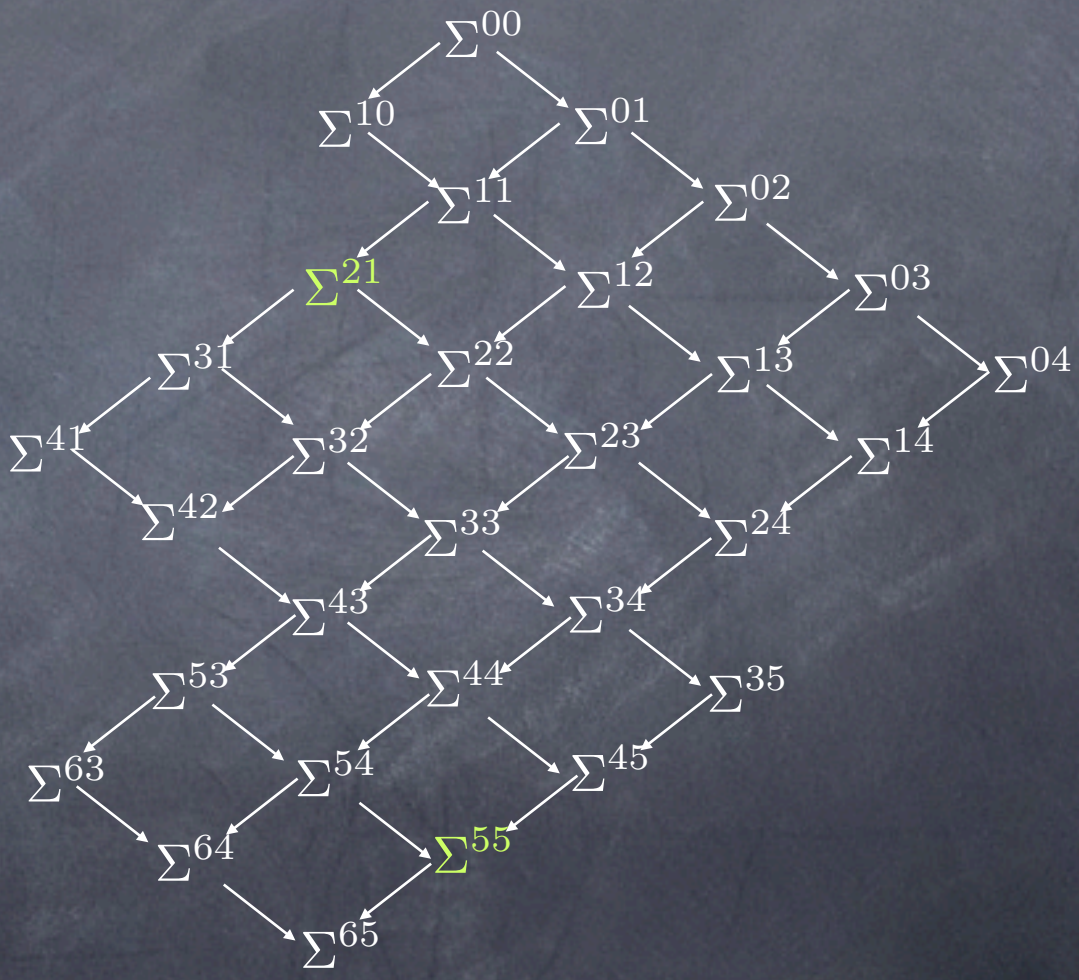
Reachability

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



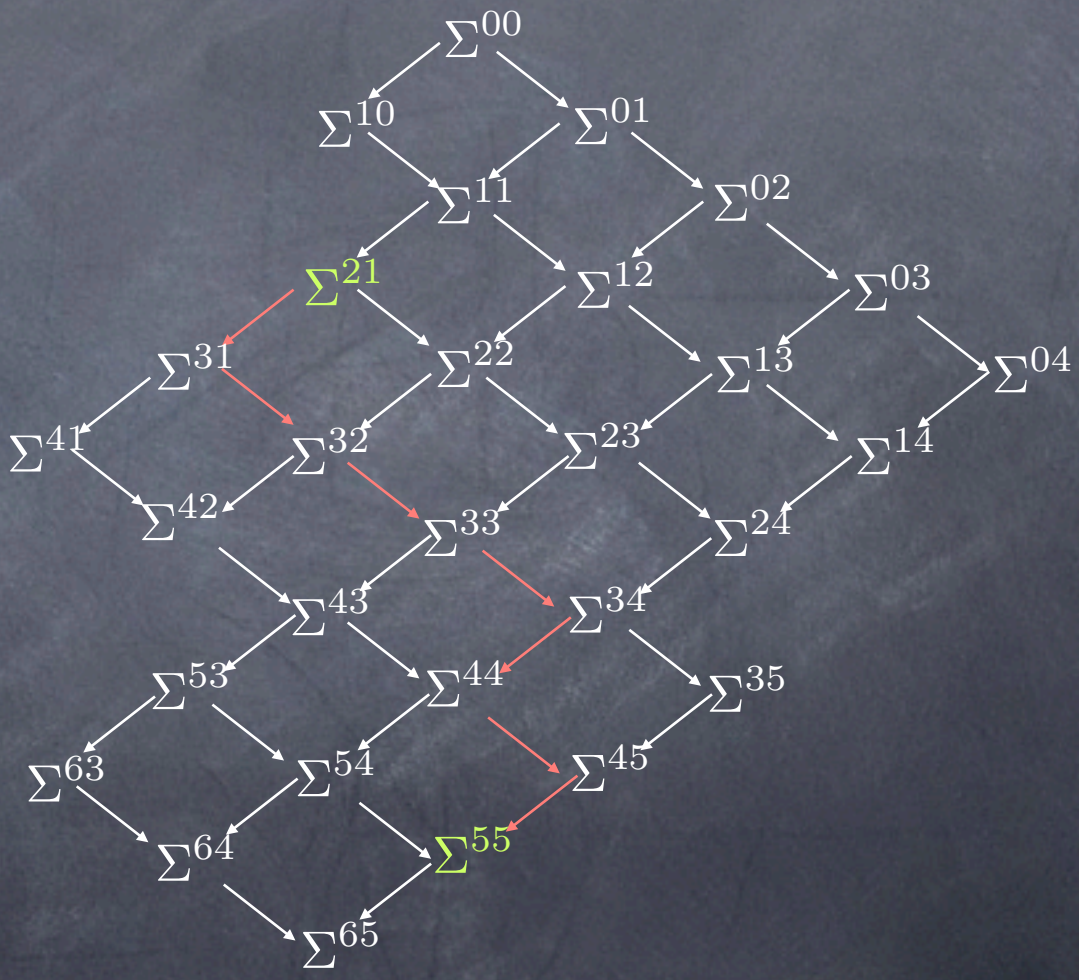
Reachability

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



Reachability

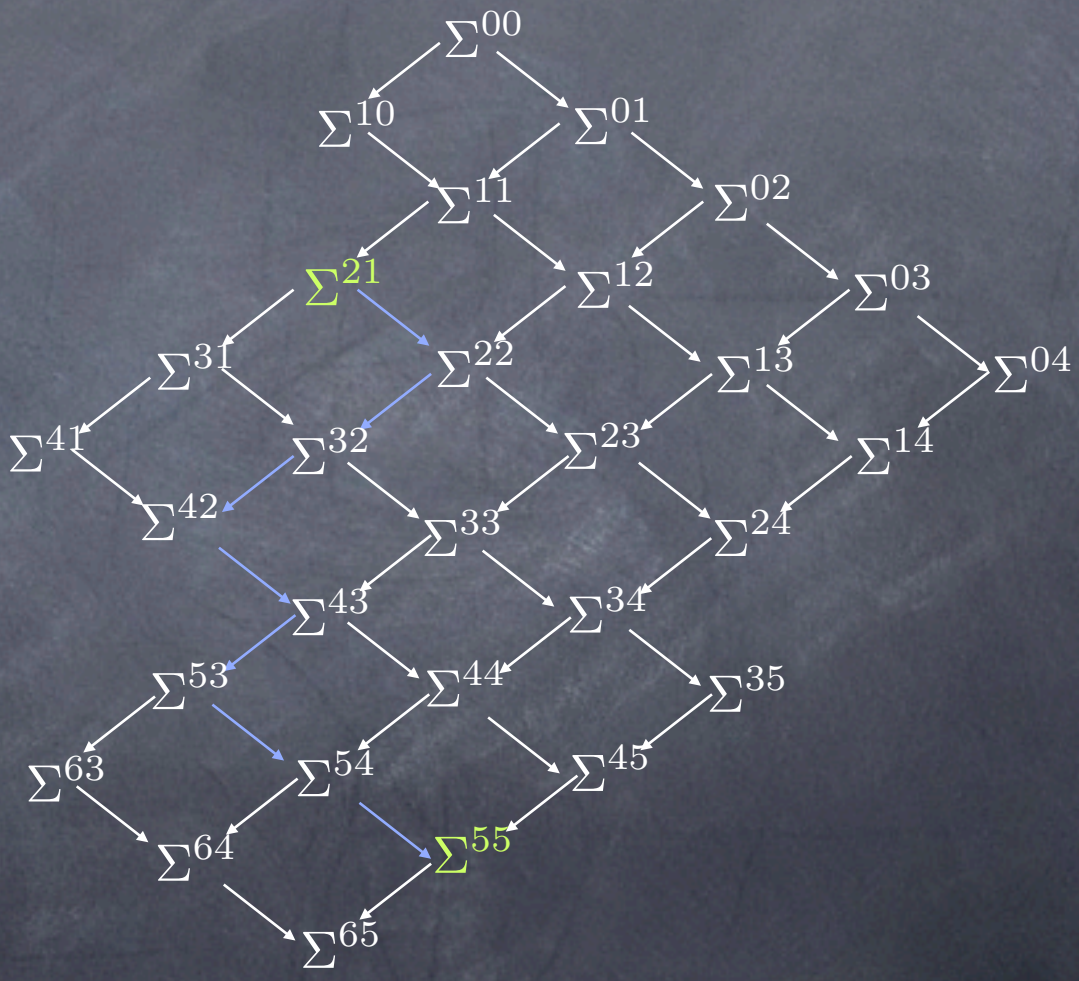
Σ^{kl} is reachable from Σ^{ij} if there is a path from Σ^{ij} to Σ^{kl} in the lattice



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?