Global Predicate Detection and Event Ordering

Our Problem

To compute predicates over the state of a distributed application

Model

- Message passing
- No failures
- Two possible timing assumptions:
  1. Synchronous System
  2. Asynchronous System
     - No upper bound on message delivery time
     - No bound on relative process speeds
     - No centralized clock

Clock Synchronization

External Clock Synchronization

keep processor clock within some maximum deviation from an external time source.

- can exchange of info about timing events of different systems
- can take actions at real-time deadlines
- synchronization within 0.1 ms

Internal Clock Synchronization:

keep processor clocks within some maximum deviation from each other.

- can measure duration of distributed activities that start on one process and terminate on another
- can totally order events that occur on a distributed system

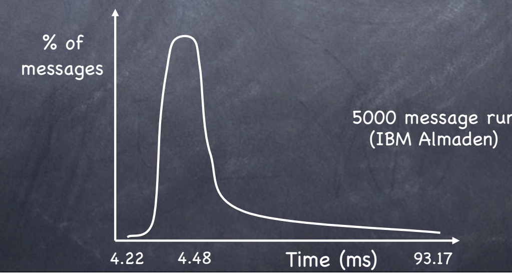
Synchronizion clocks: Take 1

- Assume an upper bound max and a lower bound min on message delivery time
- Guarantee that processes stay synchronized within max - min

Synchronizion clocks: Take 1

- Assume an upper bound max and a lower bound min on message delivery time
- Guarantee that processes stay synchronized within max - min

Problem:

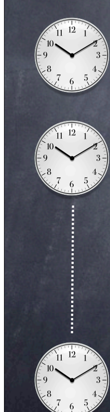


Clock Synchronization: Take 2

- No upper bound on message delivery time...
- ...but lower bound min on message delivery time
- Use timeout maxp to detect process failures
- slaves send messages to master
- Master averages slaves value; computes fault-tolerant average

Precision: 4 maxp - min

Probabilistic Clock Synchronization (Cristian)



- Master-Slave architecture
- Master is connected to external time source
- Slaves read master's clock and adjust their own

How accurately can a slave read the master's clock?

The Idea

- Clock accuracy depends on message roundtrip time
  - if roundtrip is small, master and slave cannot have drifted by much!
- Since no upper bound on message delivery, no certainty of accurate enough reading...
- ... but very accurate reading can be achieved by repeated attempts

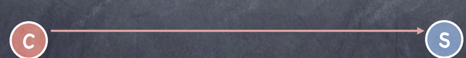
Asynchronous systems

- Weakest possible assumptions
  - cfr. "finite progress axiom"
- Weak assumptions $\equiv$ less vulnerabilities
- Asynchronous $\neq$ slow
- "Interesting" model wrt failures (ah ah ah!)

Client-Server

Processes exchange messages using
Remote Procedure Call (RPC)

A client requests a service by sending the server a message. The client blocks while waiting for a response



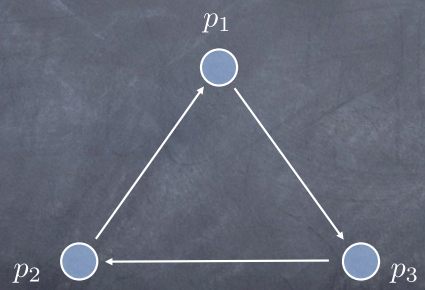
Client-Server

Processes exchange messages using
Remote Procedure Call (RPC)

A client requests a service by sending the server a message. The client blocks while waiting for a response

The server computes the response (possibly asking other servers) and returns it to the client

Deadlock!



Goal

Design a protocol by which a processor can determine whether a global predicate (say, deadlock) holds

Wait-For Graphs

- Draw arrow from p_i to p_j if p_j has received a request but has not responded yet

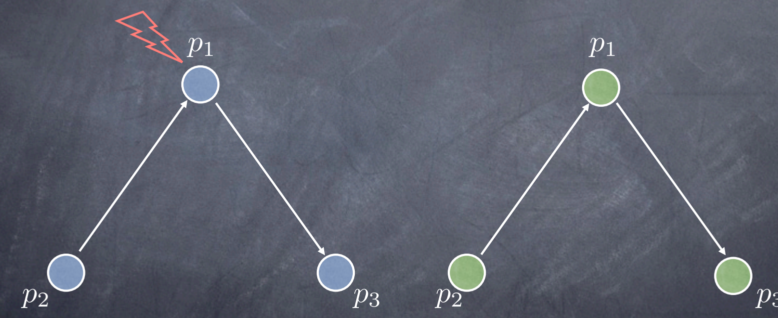
Wait-For Graphs

- Draw arrow from p_i to p_j if p_j has received a request but has not responded yet
- Cycle in WFG $\Rightarrow$ deadlock
- Deadlock $\Rightarrow \Diamond$ cycle in WFG

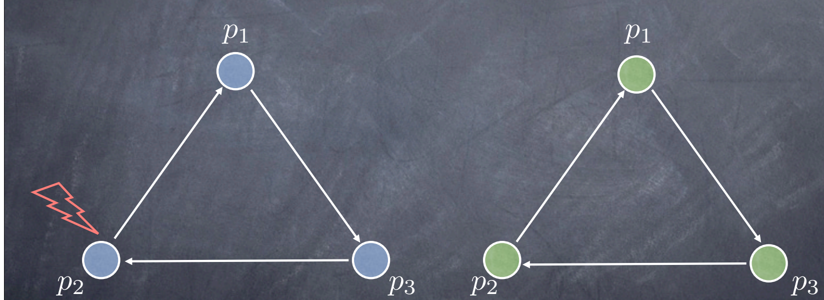
The protocol

- p_0 sends a message to $p_1 \ldots p_3$
- On receipt of p_0's message, p_i replies with its state and wait-for info

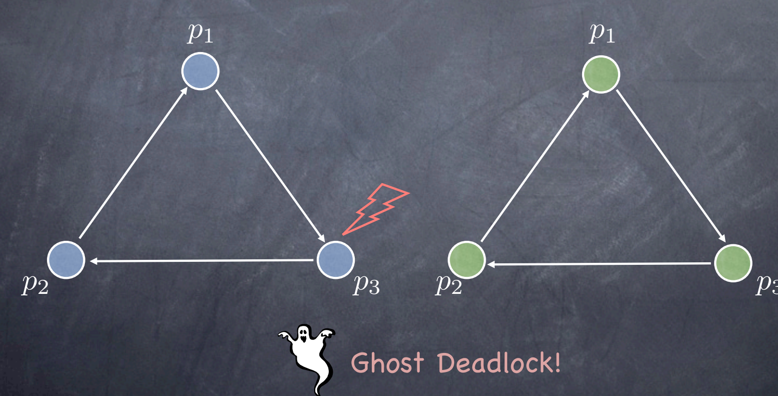
An execution



An execution



An execution



Houston, we have a problem...

- Asynchronous system
  - no centralized clock, etc. etc.
- Synchrony useful to
  - coordinate actions
  - order events
- Mmmmhhh...

Events and Histories

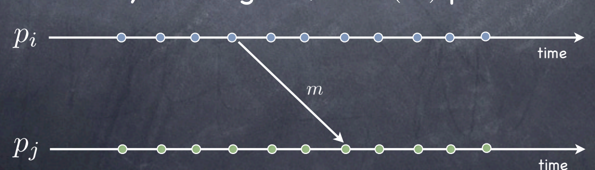
- Processes execute sequences of events
- Events can be of 3 types: local, send, and receive
- e_p^i is the i-th event of process p
- The local history h_p of process p is the sequence of events executed by process p
  - h_p^k : prefix that contains first k events
  - h_p^0 : initial, empty sequence
- The history H is the set $h_{p_0} \cup h_{p_1} \cup \ldots h_{p_{n-1}}$

NOTE: In H, local histories are interpreted as sets, rather than sequences, of events

Ordering events

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

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



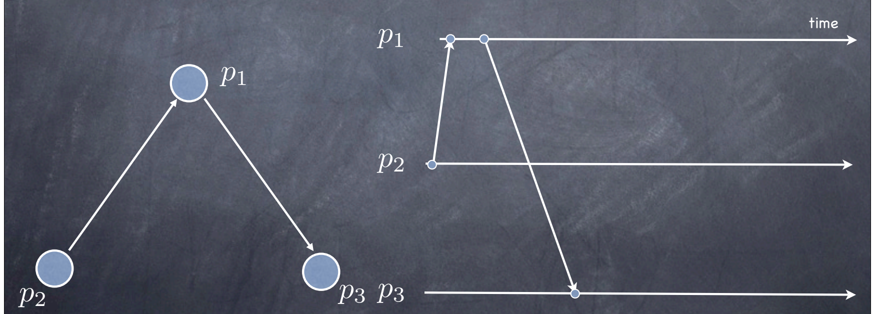
Happened-before (Lamport[1978])

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 $e \rightarrow e''$

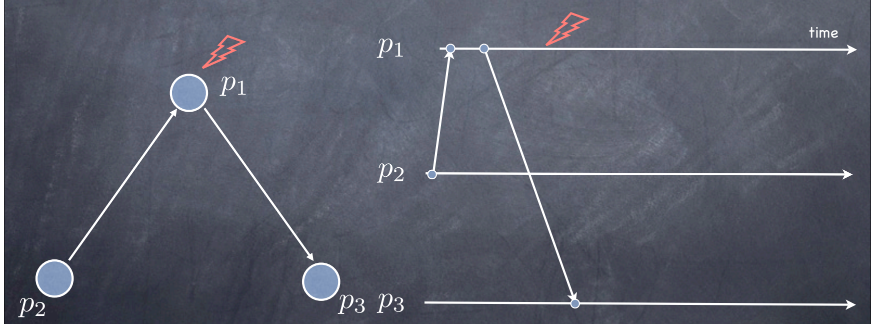
Space-Time diagrams

A graphic representation of a distributed execution



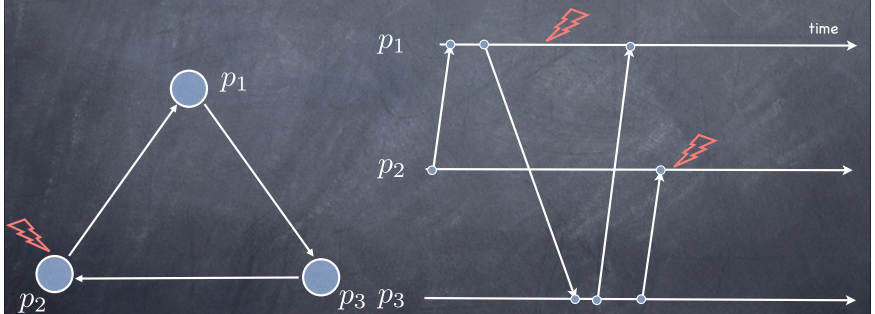
Space-Time diagrams

A graphic representation of a distributed execution



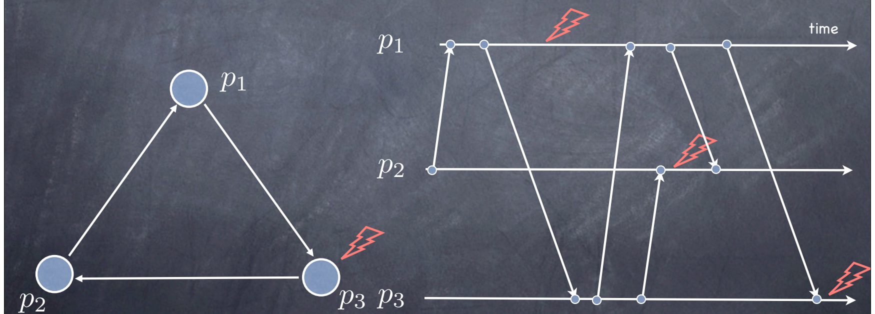
Space-Time diagrams

A graphic representation of a distributed execution



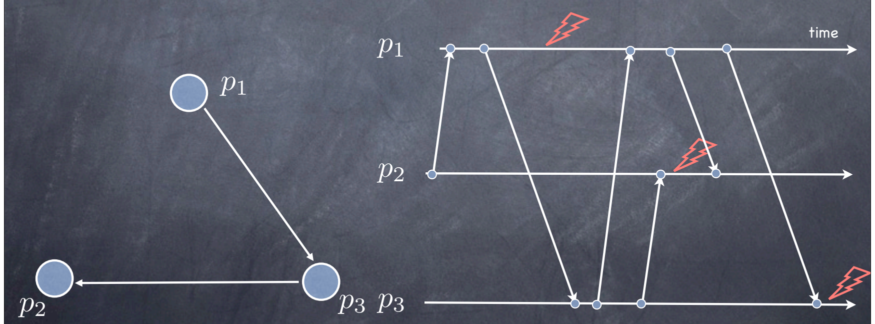
Space-Time diagrams

A graphic representation of a distributed execution



Space-Time diagrams

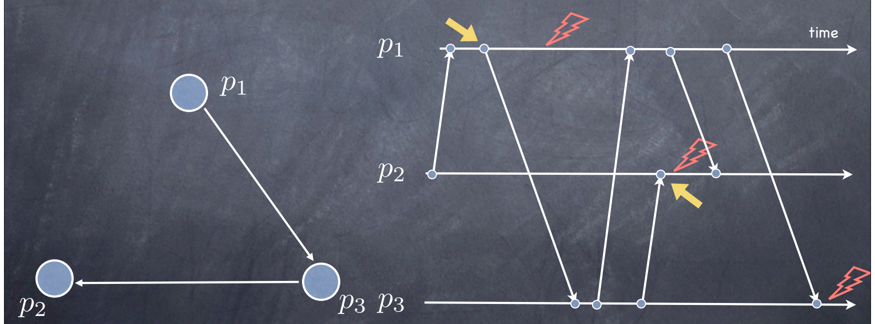
A graphic representation of a distributed execution



H and → impose a partial order

Space-Time diagrams

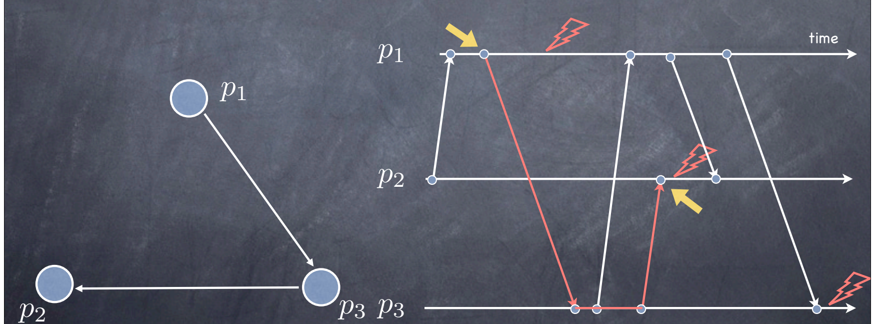
A graphic representation of a distributed execution



H and → impose a partial order

Space-Time diagrams

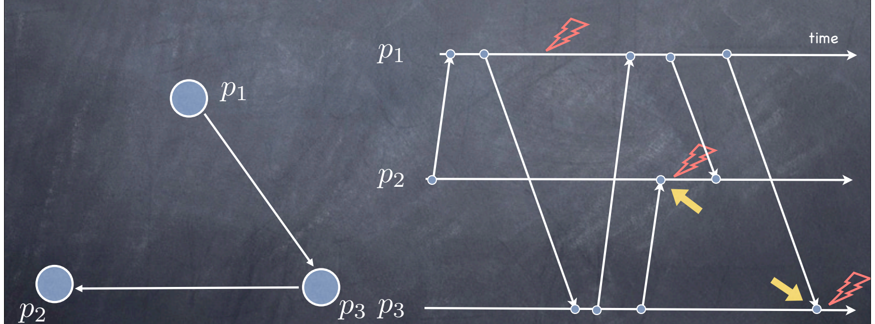
A graphic representation of a distributed execution



H and → impose a partial order

Space-Time diagrams

A graphic representation of a distributed execution



H and → impose a partial order

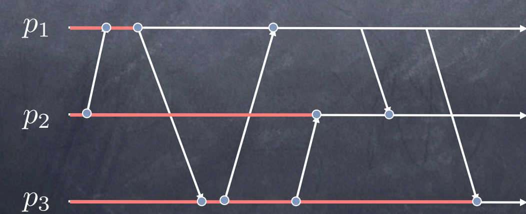
Runs and Consistent Runs

- A run is a total ordering of the events in H that is consistent with the local histories of the processors
  - Ex: $h_1, h_2, \ldots, h_n$ is a run
- A run is consistent if the total order imposed in the run is an extension of the partial order induced by $\rightarrow$
- A single distributed computation may correspond to several consistent runs!

Cuts

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

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



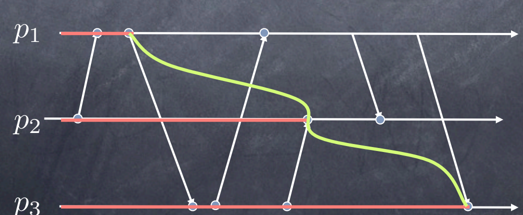
Cuts

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 $(\sigma_1^{c_1}, \ldots \sigma_n^{c_n})$

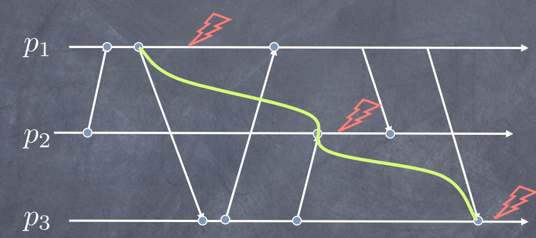
Consistent cuts and consistent global states

- A cut is consistent if

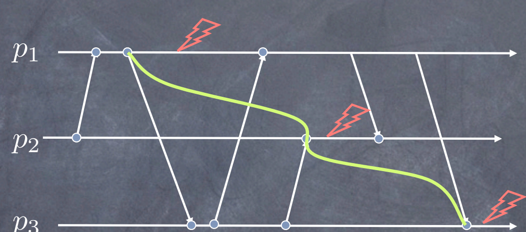
  $$\forall e_i, e_j : e_j \in C \wedge e_i \rightarrow e_j \Rightarrow e_i \in C$$

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

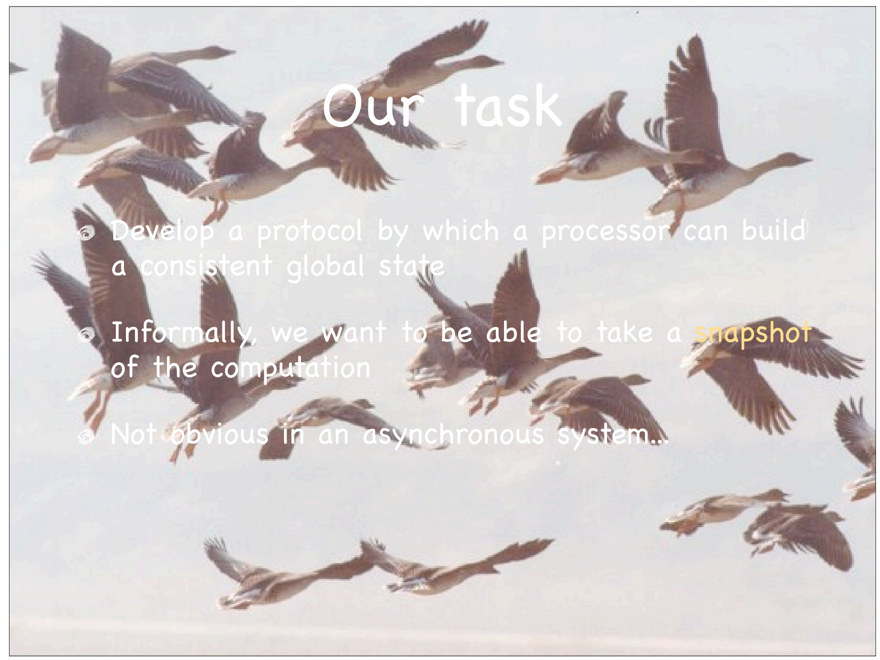
What p_0 sees



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 $e_j \in C \land e_i \rightarrow e_j \Rightarrow e_i \in C$

< Definition >	< 0 and 1>	< 5 and 3>
$0.\ e_j \in C \equiv T(e_j) < t_{ss}$	$3.\ T(e_j) < t_{ss}$	$6.\ T(e_i) < t_{ss}$
< Assumption >	< Property of real time>	< Definition >
$1.\ e_j \in C$	$4.\ e_i \rightarrow e_j \Rightarrow T(e_i) < T(e_j)$	$7.\ e_i \in C$
< Assumption >	< 2 and 4>	
$2.\ e_i \rightarrow e_j$	$5.\ T(e_i) < T(e_j)$	

Clock Condition

< Property of real time>

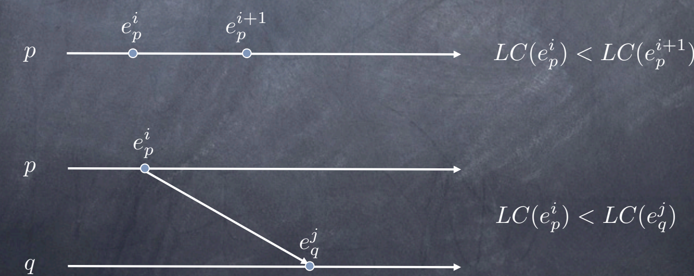
4. $e_i \rightarrow e_j \Rightarrow T(e_i) < T(e_j)$

Can the Clock Condition be implemented some other way?

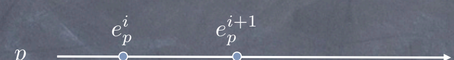
Lamport Clocks

Each process maintains a local variable LC

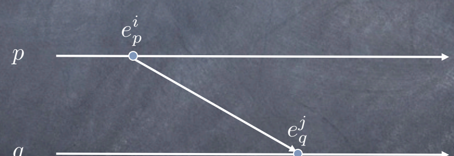
$LC(e) \equiv$ value of LC for event e



Increment Rules



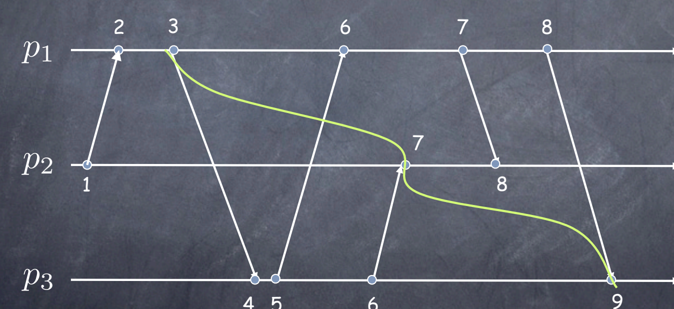
$LC(e_p^{i+1}) = LC(e_p^i) + 1$



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

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

Space-Time Diagrams and Logical Clocks



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...

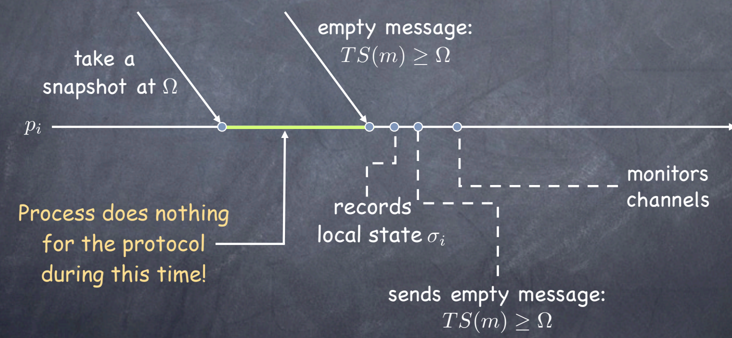
- Doing so assumes
  - upper bound on message delivery time
  - upper bound relative process speeds

We better relax it...

Snapshot II

- processor p_0 selects Ω
- p_0 sends "take a snapshot at Ω" to all processes and 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

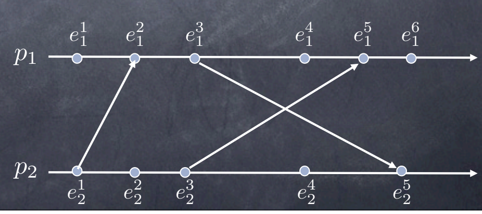
- The global state Σ^s saved by the snapshot protocol is a consistent global state
- But did it ever occur during the computation?
  - a distributed computation provides only a partial order of events
  - many total orders (runs) are compatible with that partial order
  - all we know is that Σ^s could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

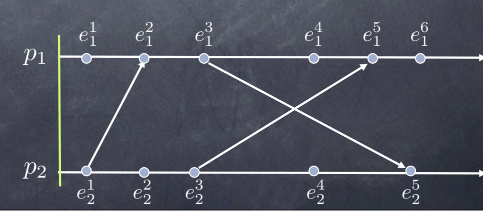
- We are evaluating predicates on states that may have never occurred!

An Execution and its Lattice

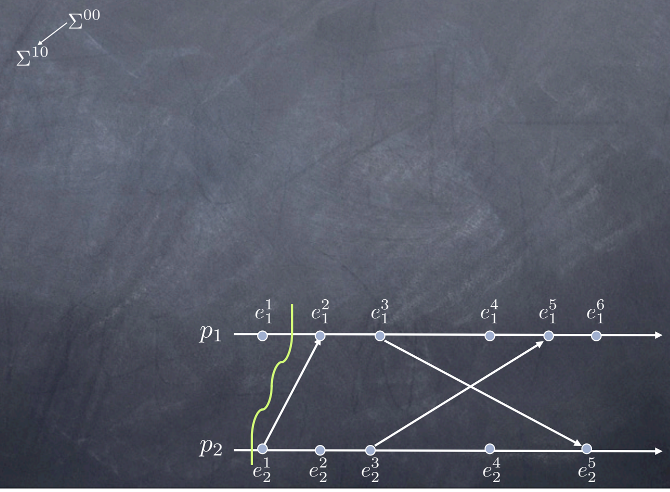


An Execution and its Lattice

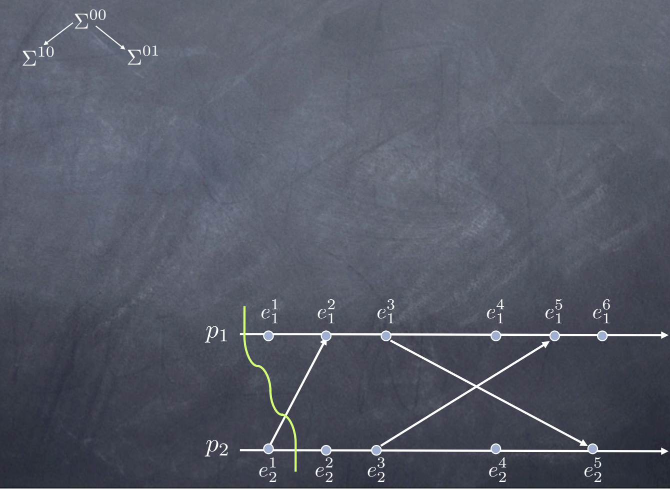
Σ^{00}



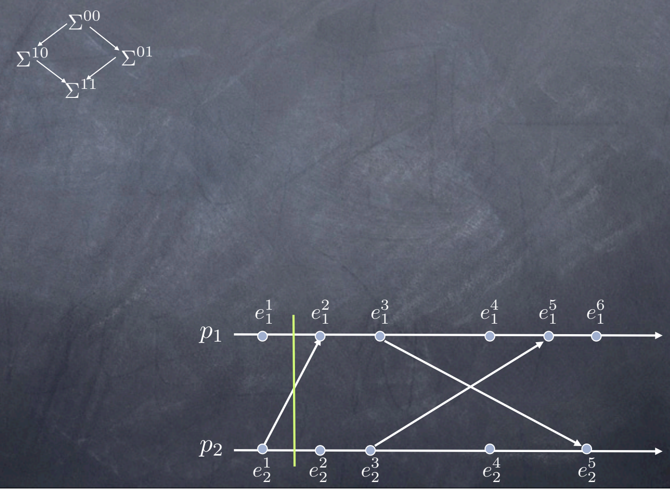
An Execution and its Lattice



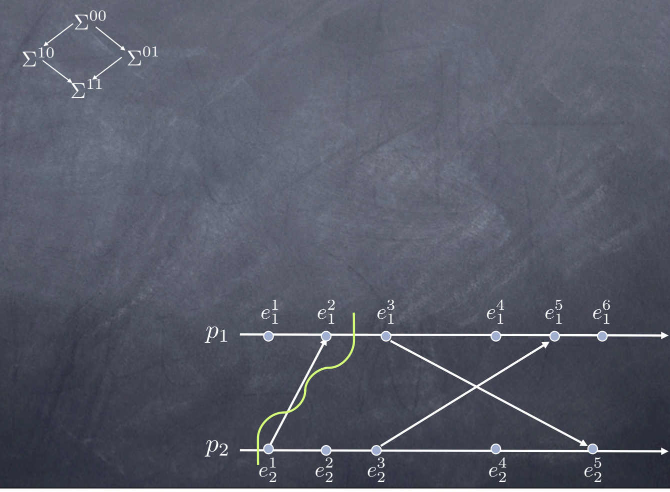
An Execution and its Lattice



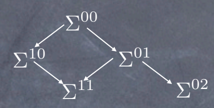
An Execution and its Lattice

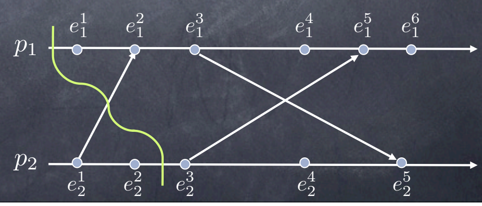


An Execution and its Lattice

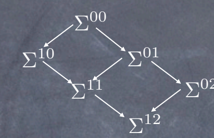


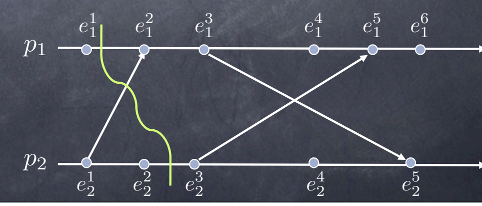
An Execution and its Lattice



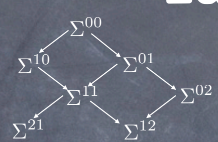


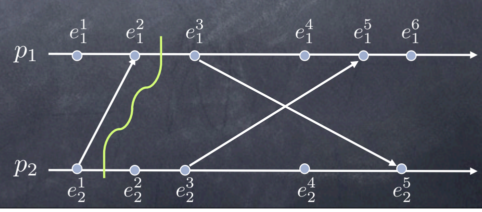
An Execution and its Lattice



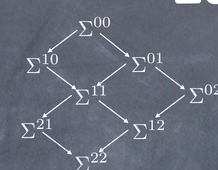


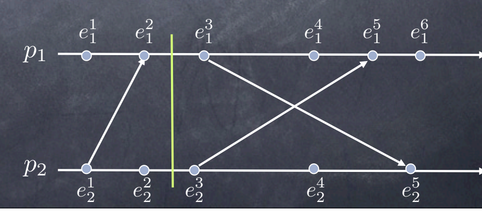
An Execution and its Lattice



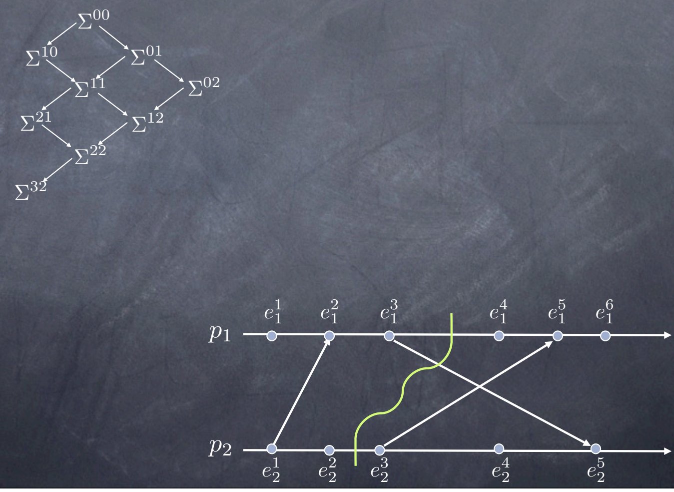


An Execution and its Lattice

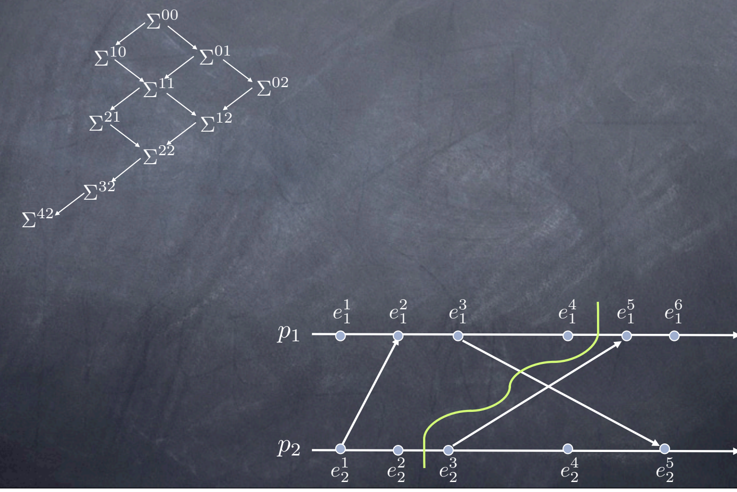




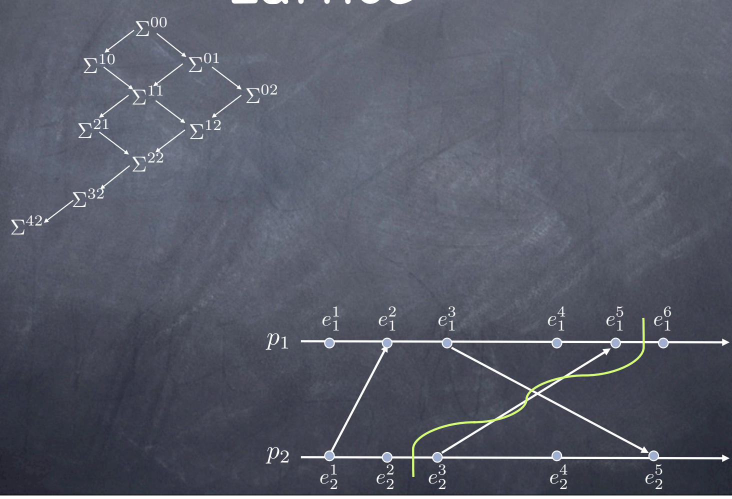
An Execution and its Lattice



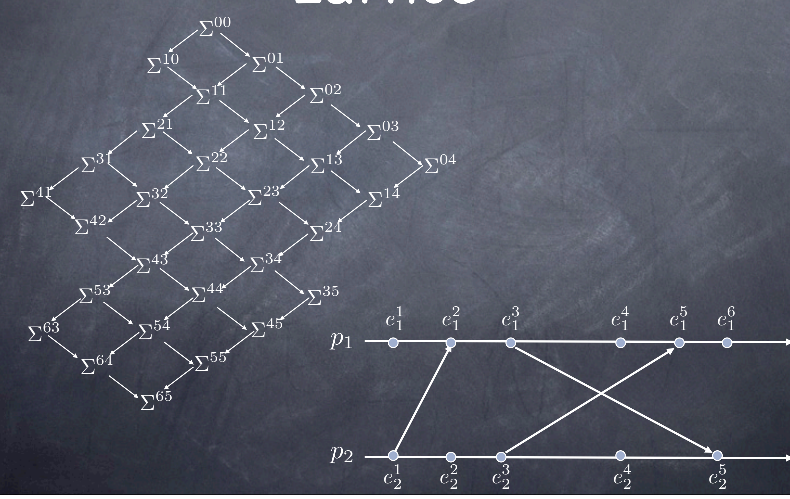
An Execution and its Lattice



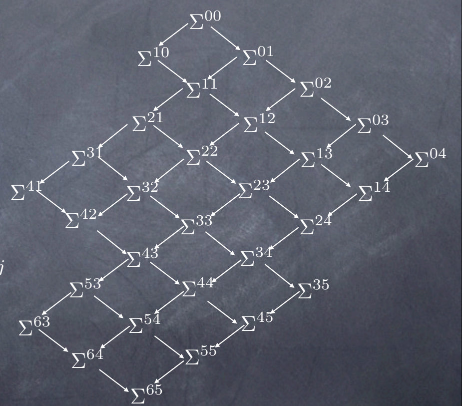
An Execution and its Lattice



An Execution and its Lattice

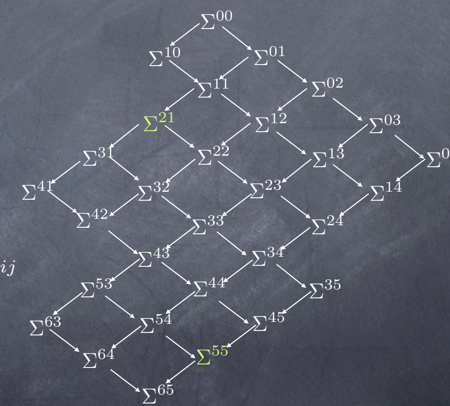


Reachability



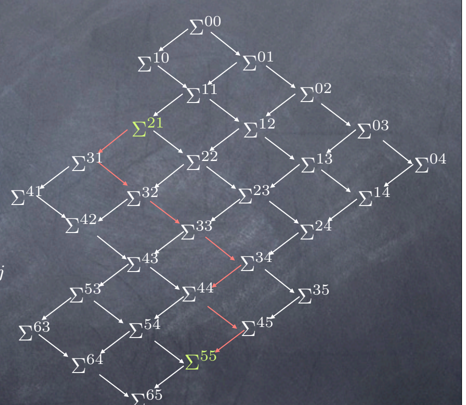
Σ^{kl} is reachable from Σ^{ij} if there is a path from Σ^{kl} to Σ^{ij} in the lattice

Reachability



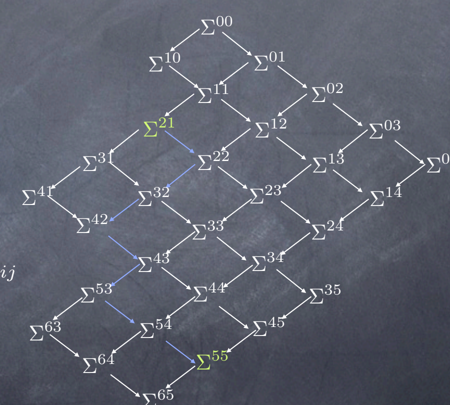
Σ^{kl} is reachable from Σ^{ij} if there is a path from Σ^{kl} to Σ^{ij} in the lattice

Reachability



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

Reachability



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