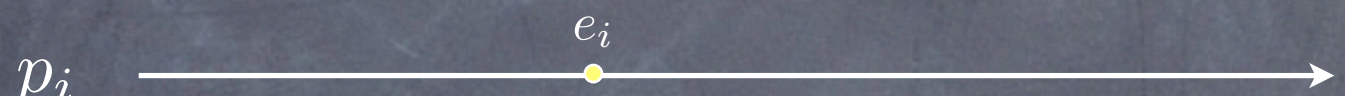
Vector Clocks

- 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 $k_1, k_2, \ldots, k_n$

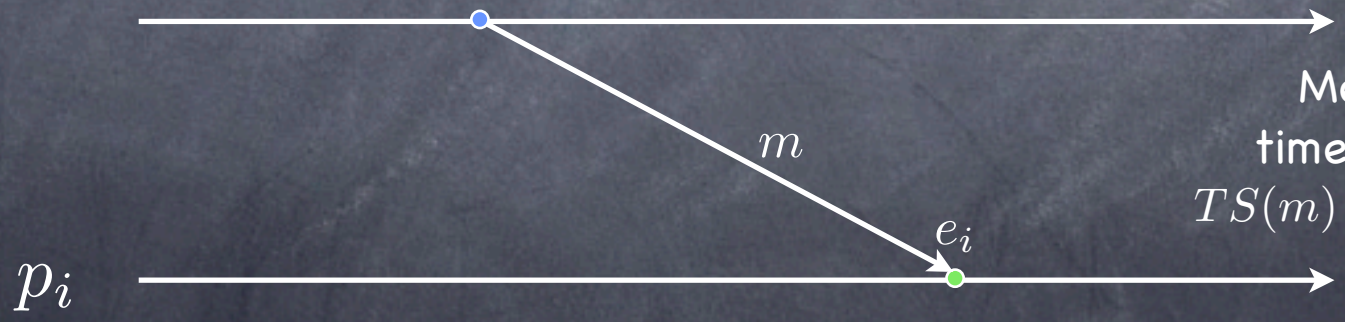
Represent θ using an n-vector VC such that

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

Update rules



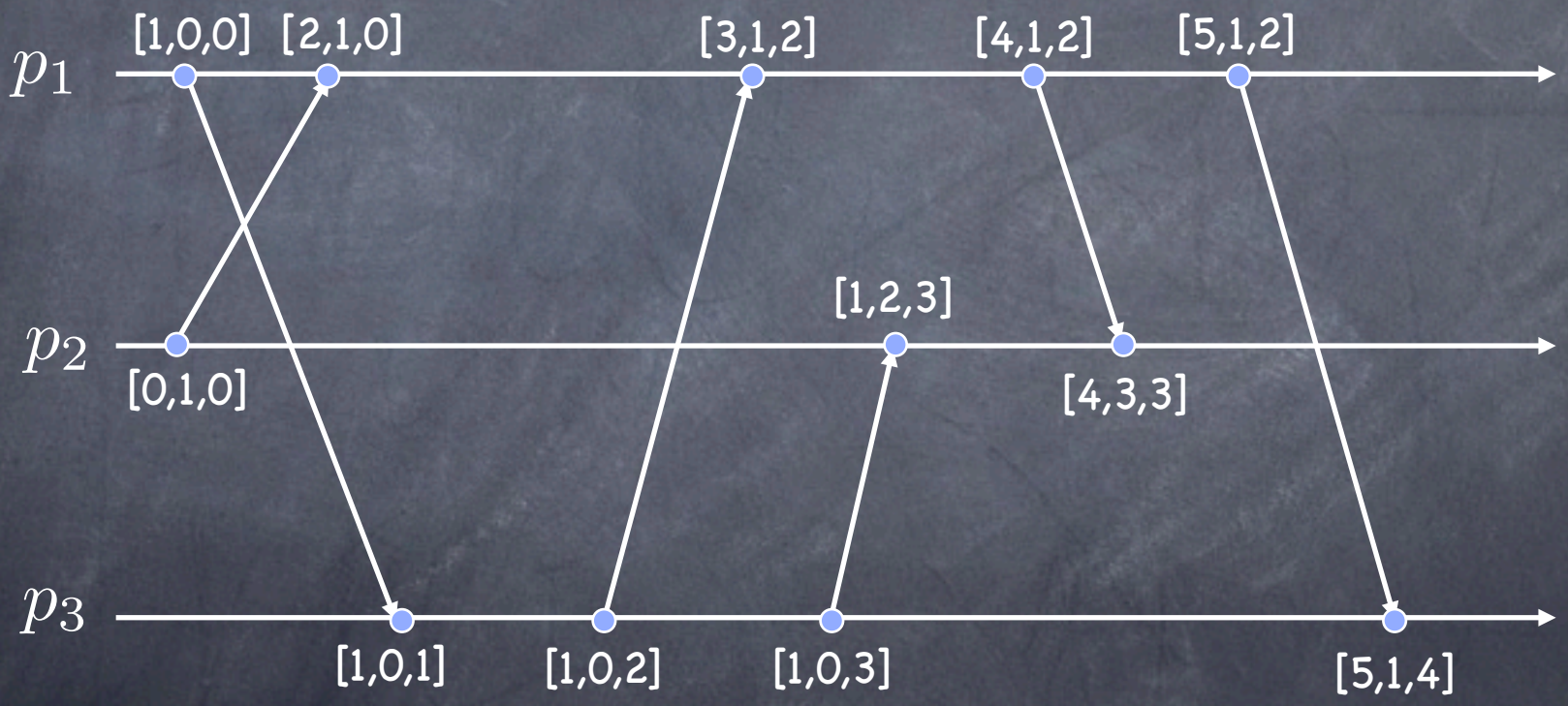
$$VC(e_i)[i] := VC[i] + 1$$



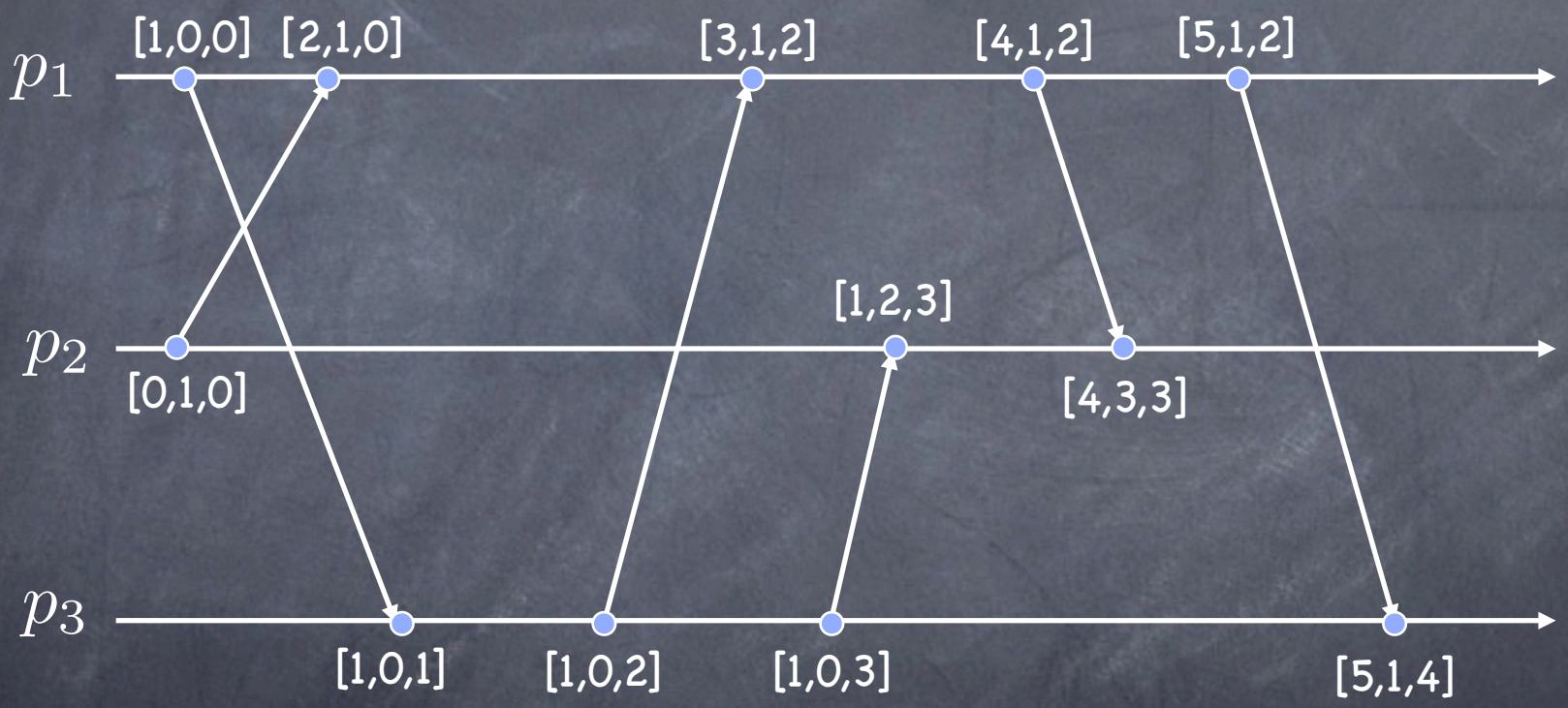
Message m is timestamped with $TS(m) = VC(send(m))$

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

Example



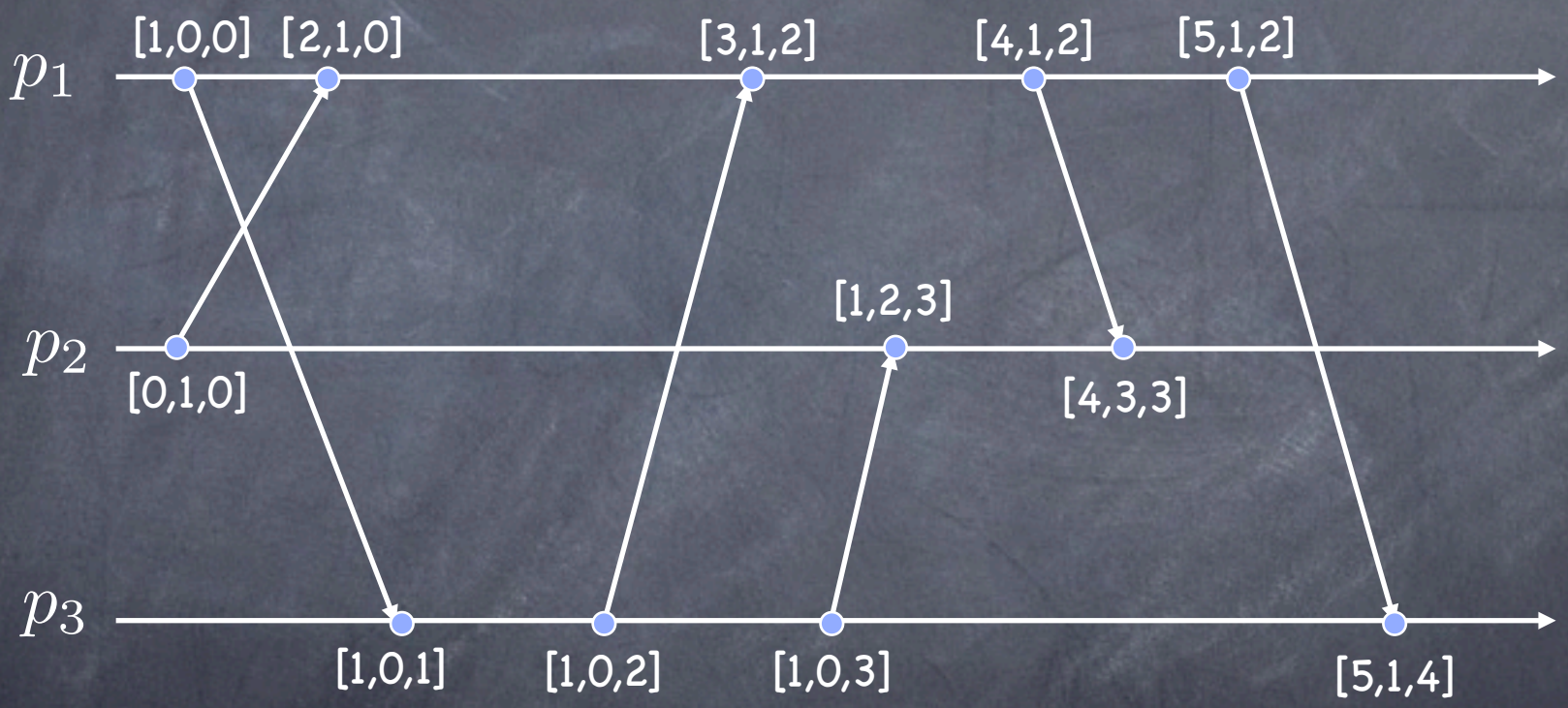
Operational interpretation



$VC(e_i)[i]$ =

$VC(e_i)[j]$ =

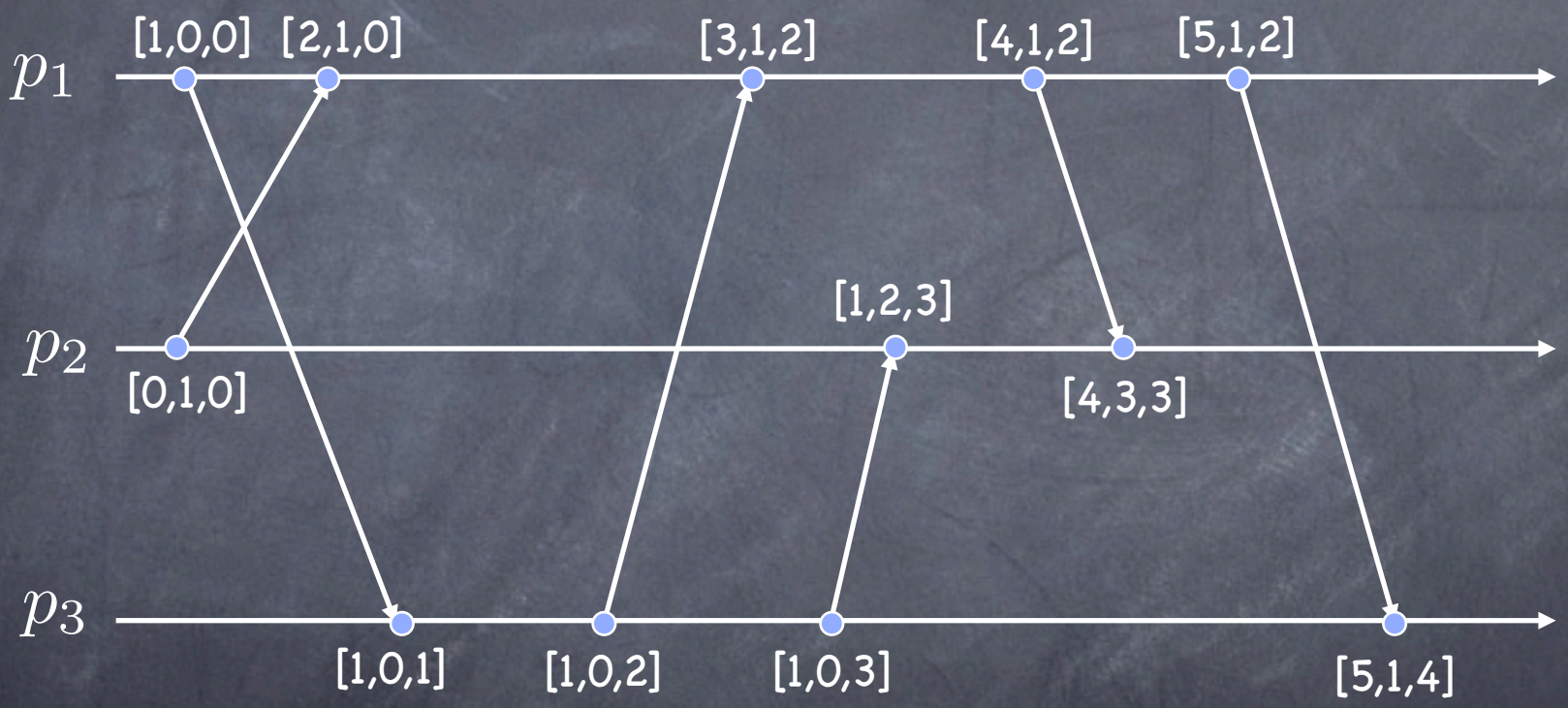
Operational interpretation



$VC(e_i)[i]$ = 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

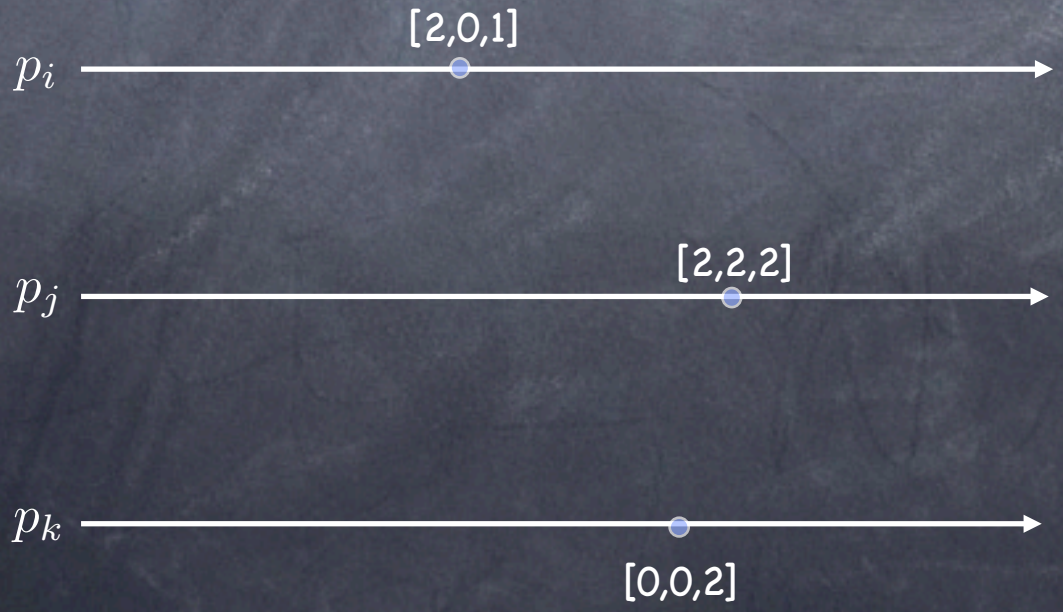
Given e_i of p_i and e_j of p_j, if $VC(e_i)[k] < VC(e_j)[k]$ for some $k \neq j$, then there exists e_k s.t.

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.

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

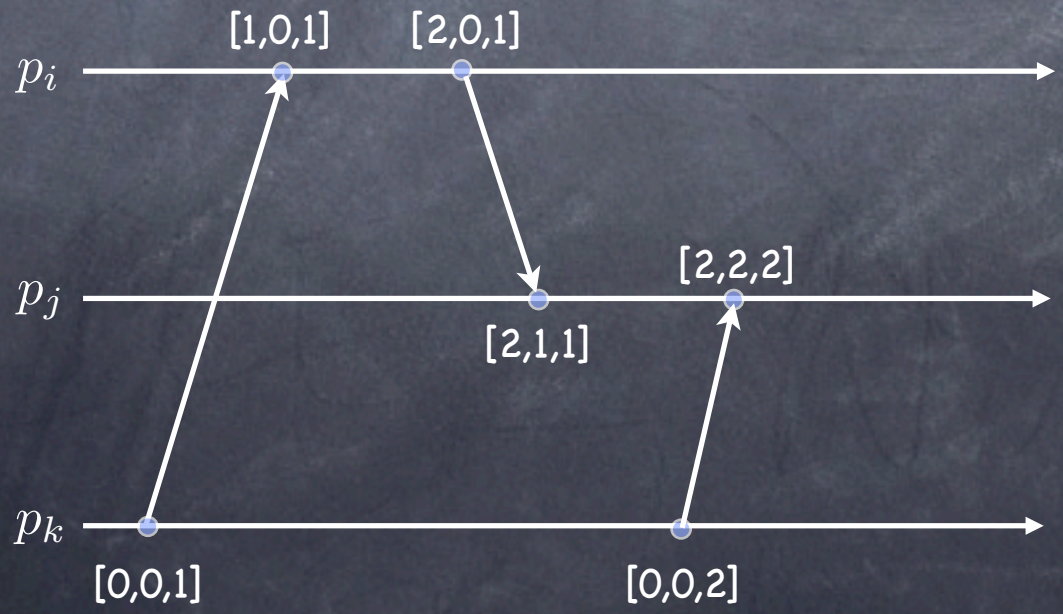


VC properties: weak gap detection

- 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

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



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

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

- 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.

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

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?

Deliver m_j after all the messages that precede m_j in its causal history!

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

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

So, again... Deliver m after all the causal history that precedes m !

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:

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

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

Properties

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

> "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)

The challenges of non-stable predicates

- Consider a non-stable predicate Φ encoding, say, a safety property. We want to determine whether Φ holds for our program.

The challenges of non-stable predicates

- Consider a non-stable predicate Φ encoding, say, a safety property. We want to determine whether Φ holds for our program.
- Suppose we apply Φ to Σ^s

The challenges of non-stable predicates

- Consider a non-stable predicate Φ encoding, say, a safety property. We want to determine whether Φ holds for our program.
- Suppose we apply Φ to Σ^s
- Φ holding in Σ^s does not preclude the possibility that our program violates safety!

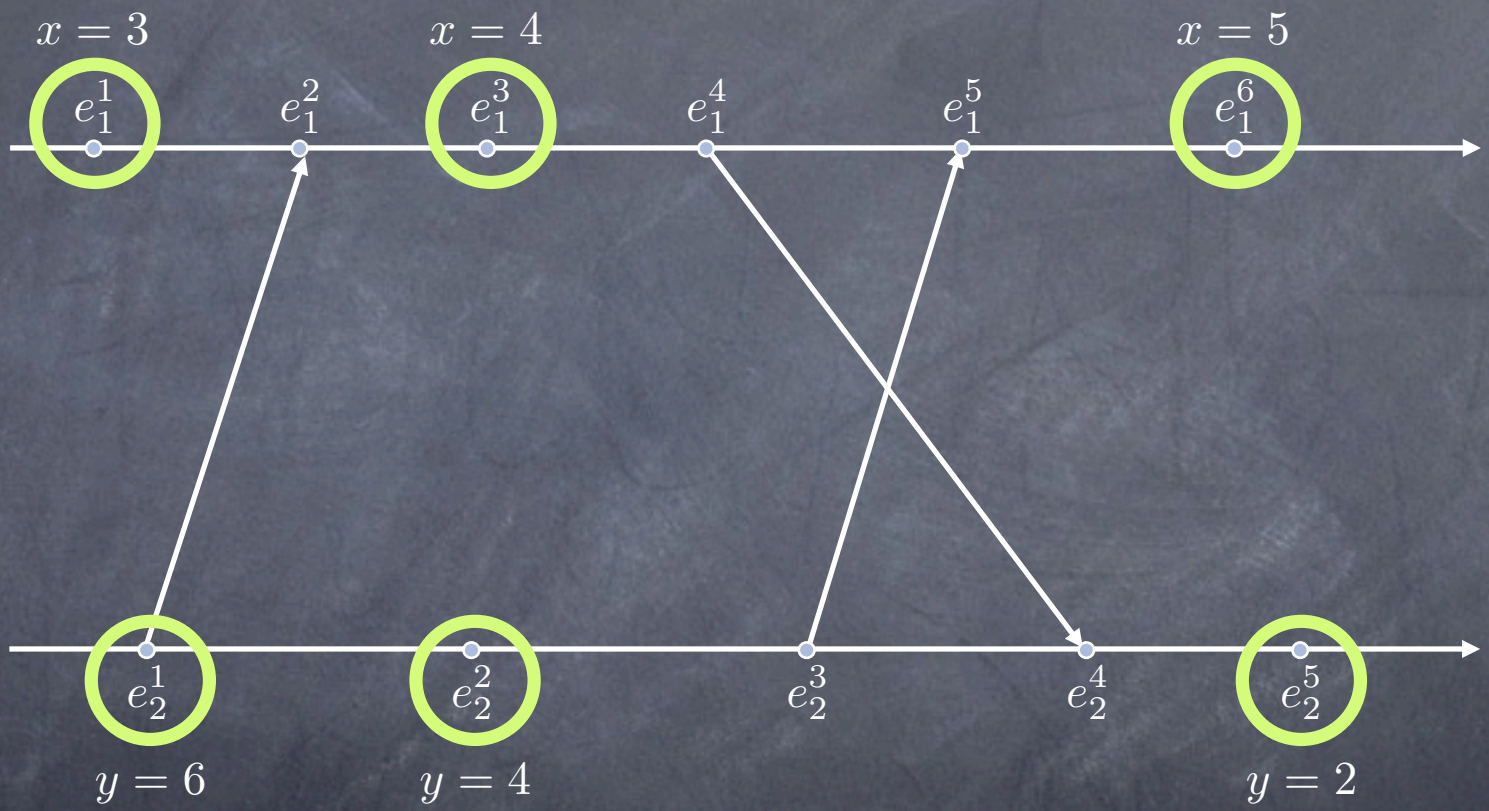
The challenges of non-stable predicates

- Consider now a different non-stable predicate Φ. We want to determine whether Φ ever holds during a particular computation.
- Suppose we apply Φ to Σ^s

The challenges of non-stable predicates

- Consider now a different non-stable predicate Φ. We want to determine whether Φ ever holds during a particular computation.
- Suppose we apply Φ to Σ^s
- Φ holding in Σ^s does not imply that Φ ever held during the actual computation!

Example



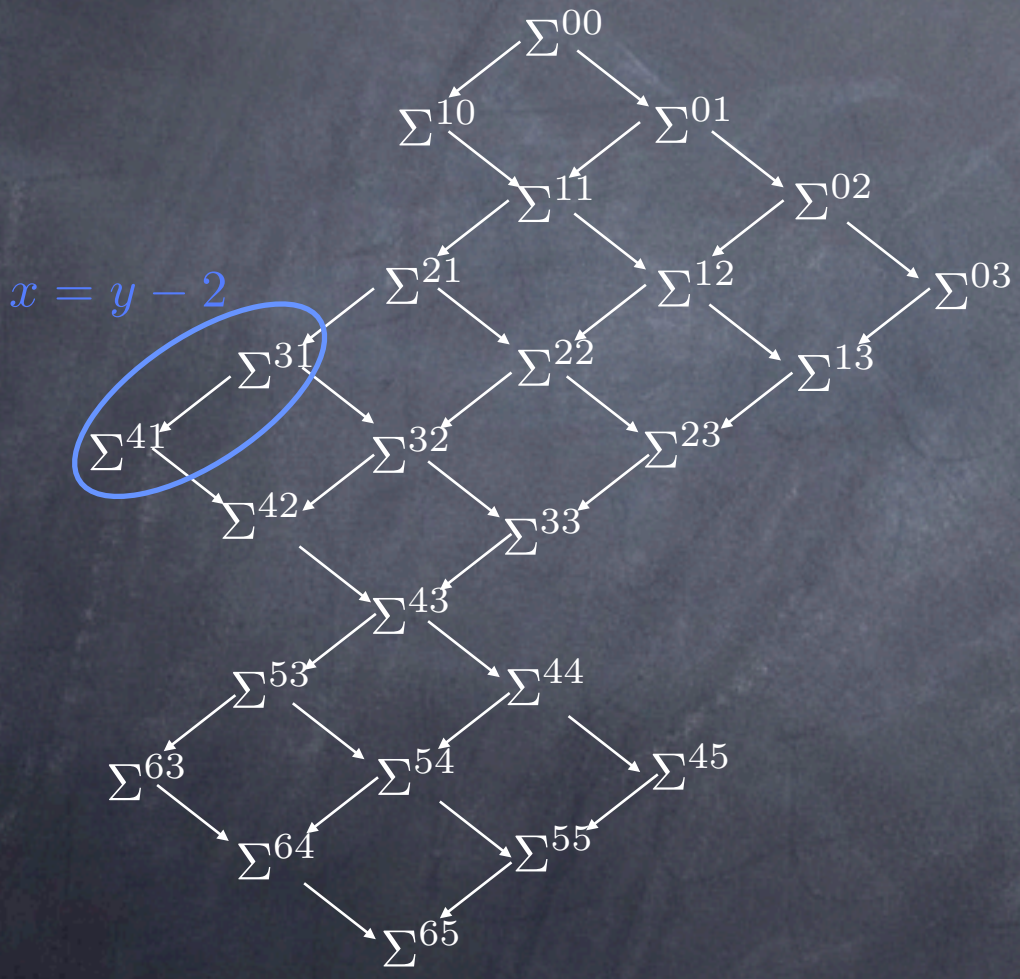
Detect whether the following predicates hold:

$x = y$ $\quad$ $x = y - 2$

Assume that initially:

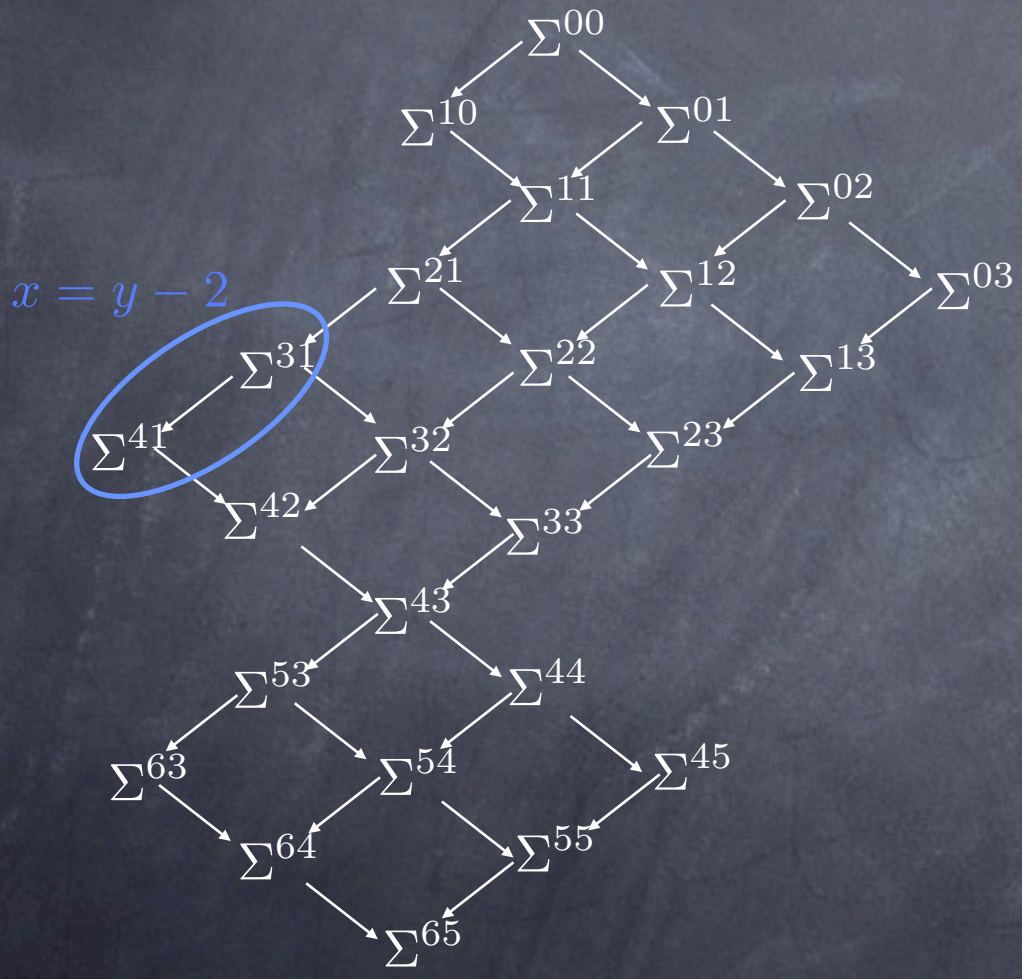
$x = 0; y = 10$

Possibly



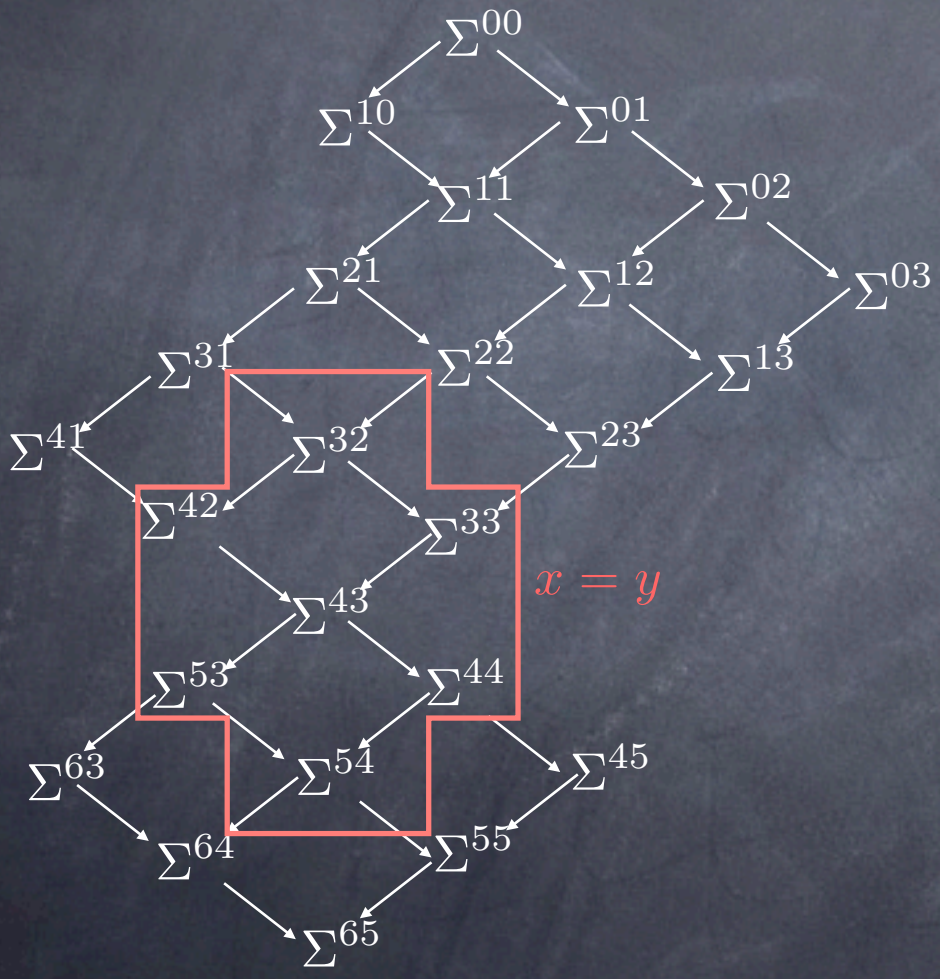
- If Σ^s is Σ^{31} or Σ^{41}, $x = y - 2$ is detected, but it may never have occurred

Possibly



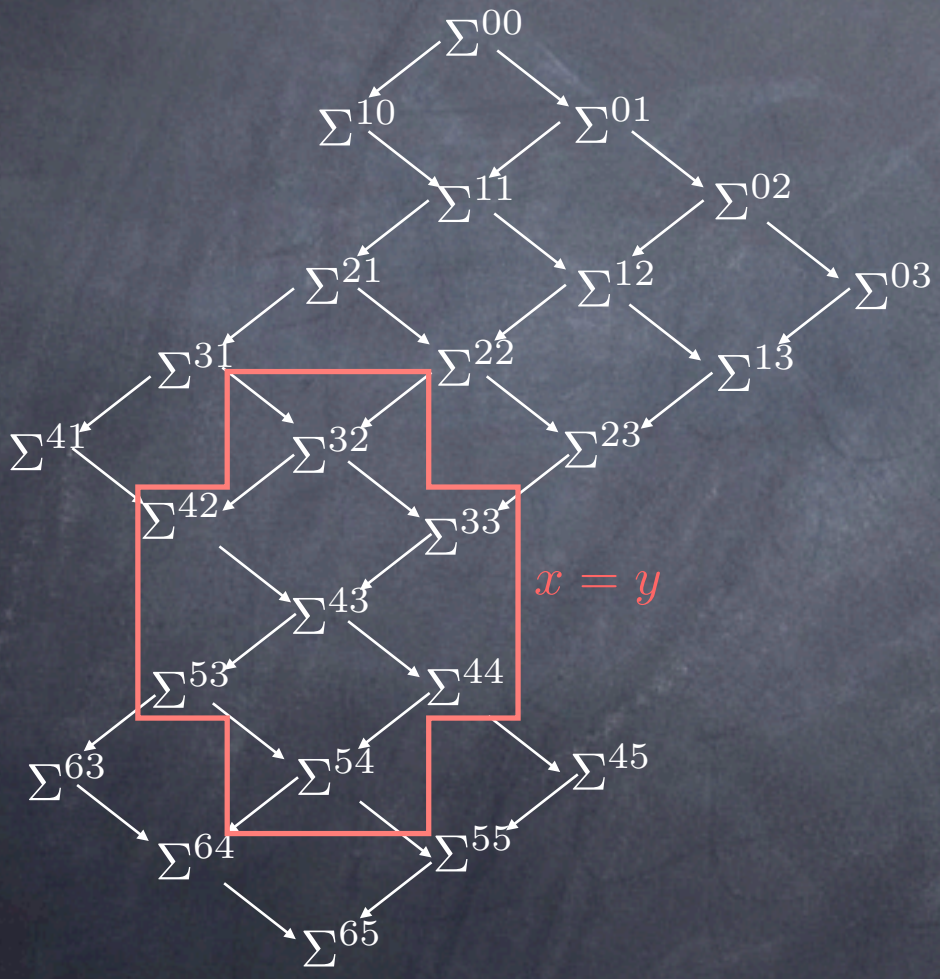
- If Σ^s is Σ^{31} or Σ^{41}, $x = y - 2$ is detected, but it may never have occurred
- Possibly(Φ)
  There exists a consistent observation of the computation O such that Φ holds in a global state of O

Definitely



- We know that $x = y$ has occurred, but it may not be detected if tested before Σ^{32} or after Σ^{54}

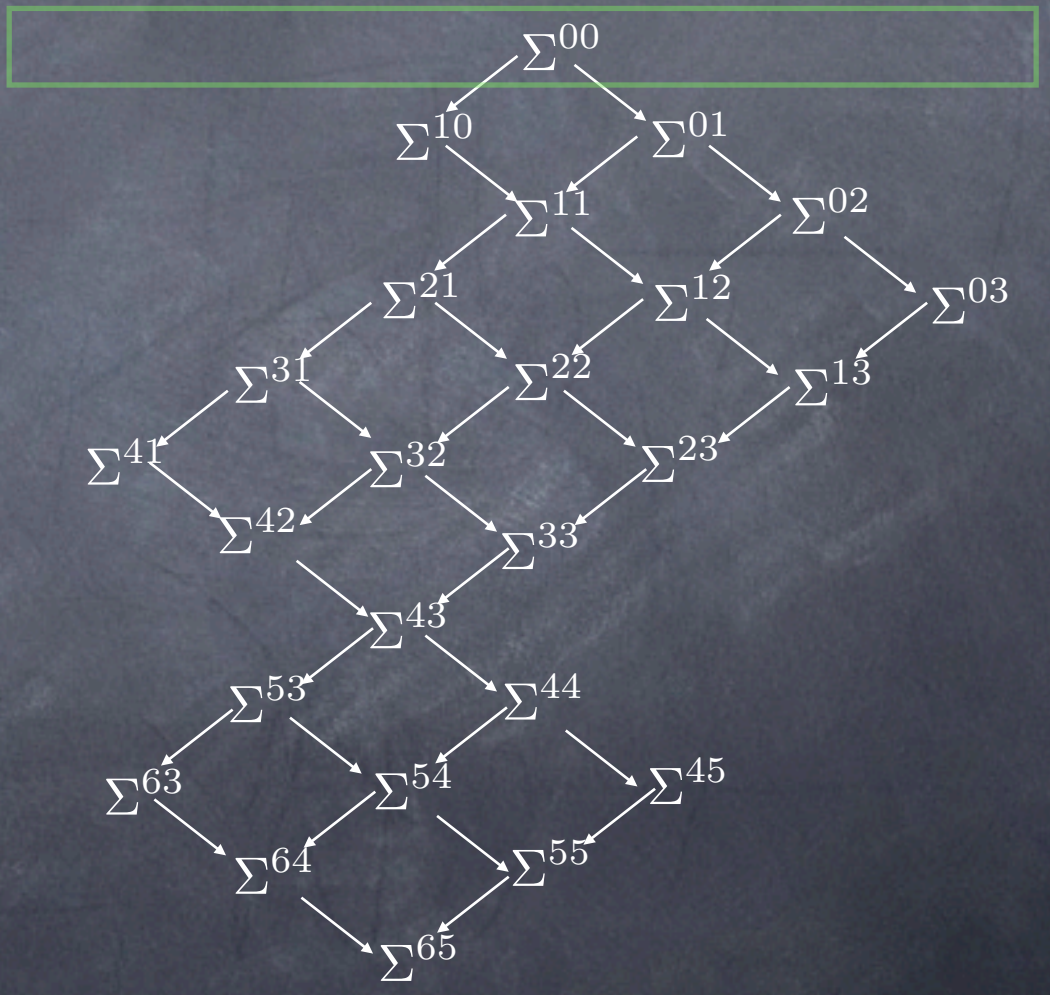
Definitely



- We know that $x = y$ has occurred, but it may not be detected if tested before Σ^{32} or after Σ^{54}
- Definitely(Φ)
  For every consistent observation O of the computation, there exists a global state of O in which Φ holds

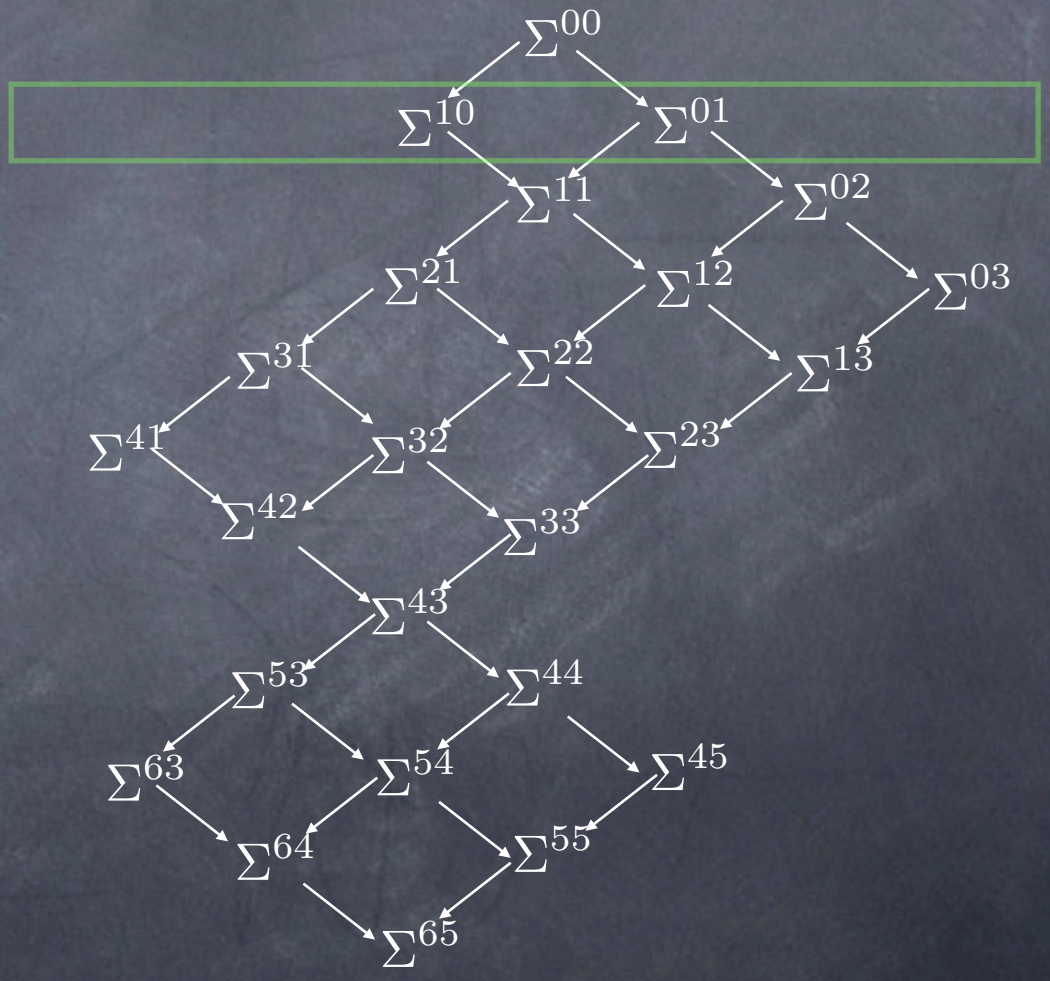
Computing Possibly

- Scan lattice, level after level
- If Φ holds in one global state, then Possibly(Φ)



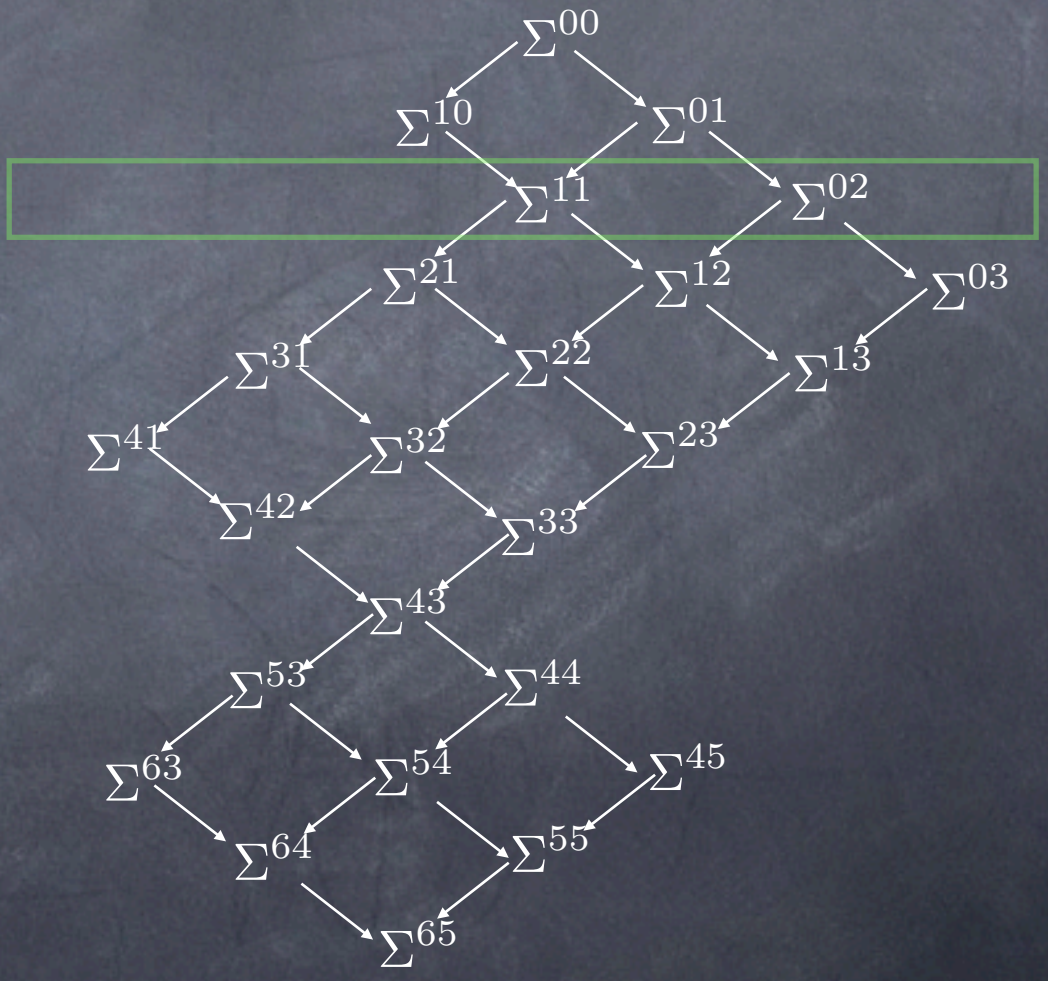
Computing Possibly

- Scan lattice, level after level
- If Φ holds in one global state, then Possibly(Φ)



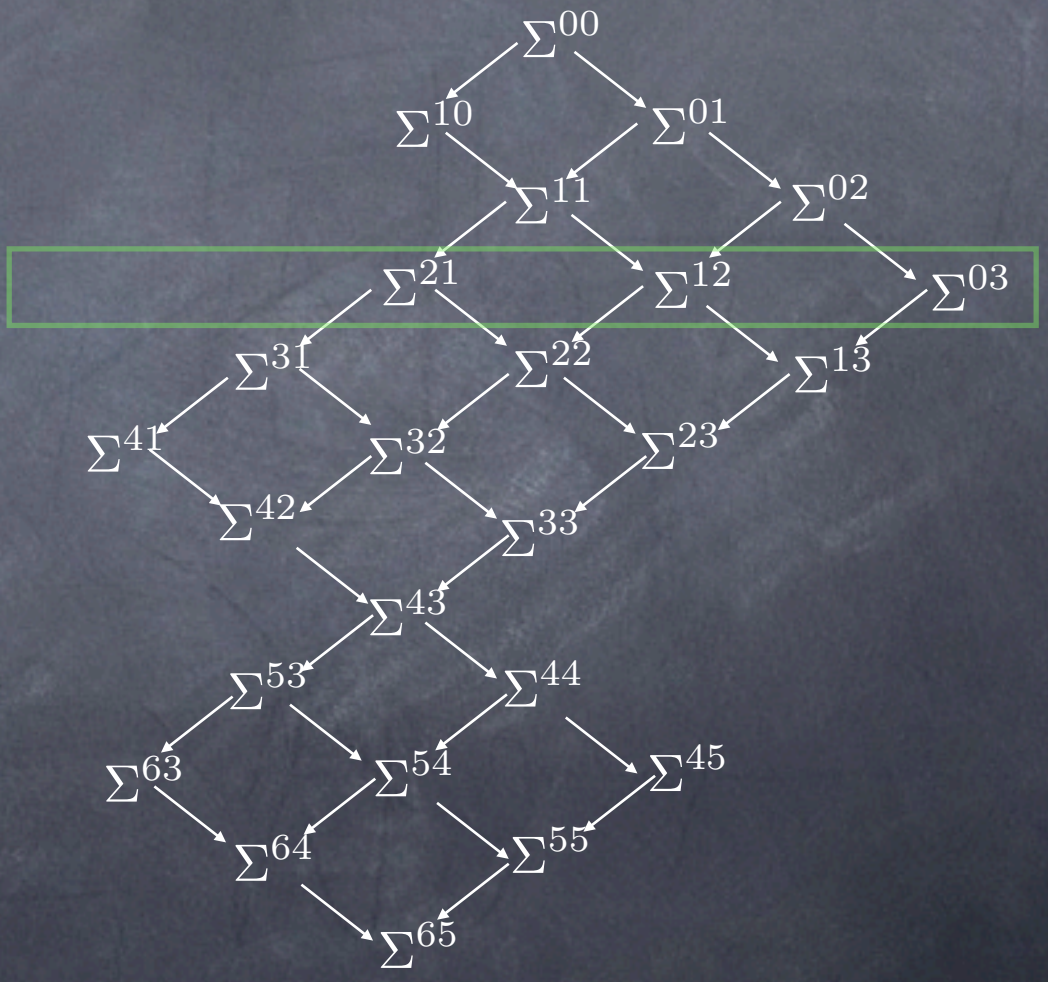
Computing Possibly

- Scan lattice, level after level
- If Φ holds in one global state, then Possibly(Φ)



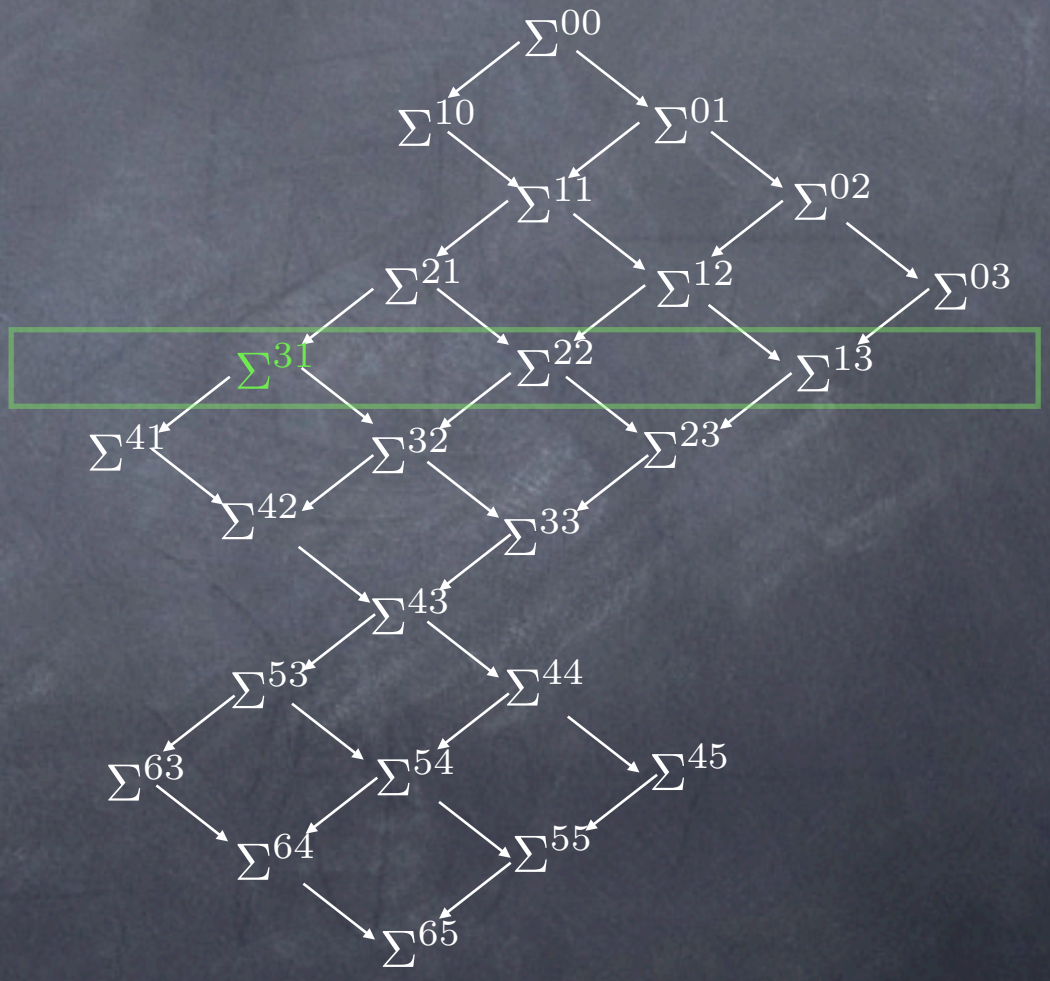
Computing Possibly

- Scan lattice, level after level
- If Φ holds in one global state, then Possibly(Φ)



Computing Possibly

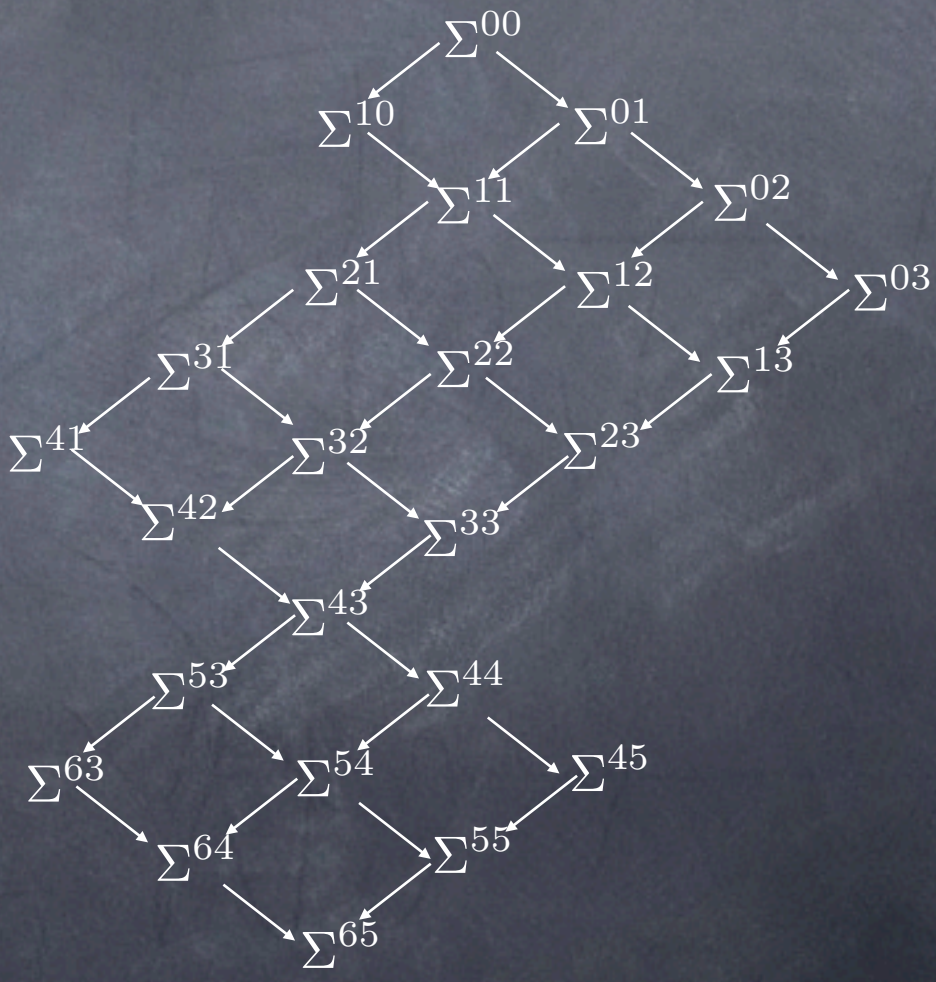
- Scan lattice, level after level
- If Φ holds in one global state, then Possibly(Φ)



Possibly$(x = y - 2)$

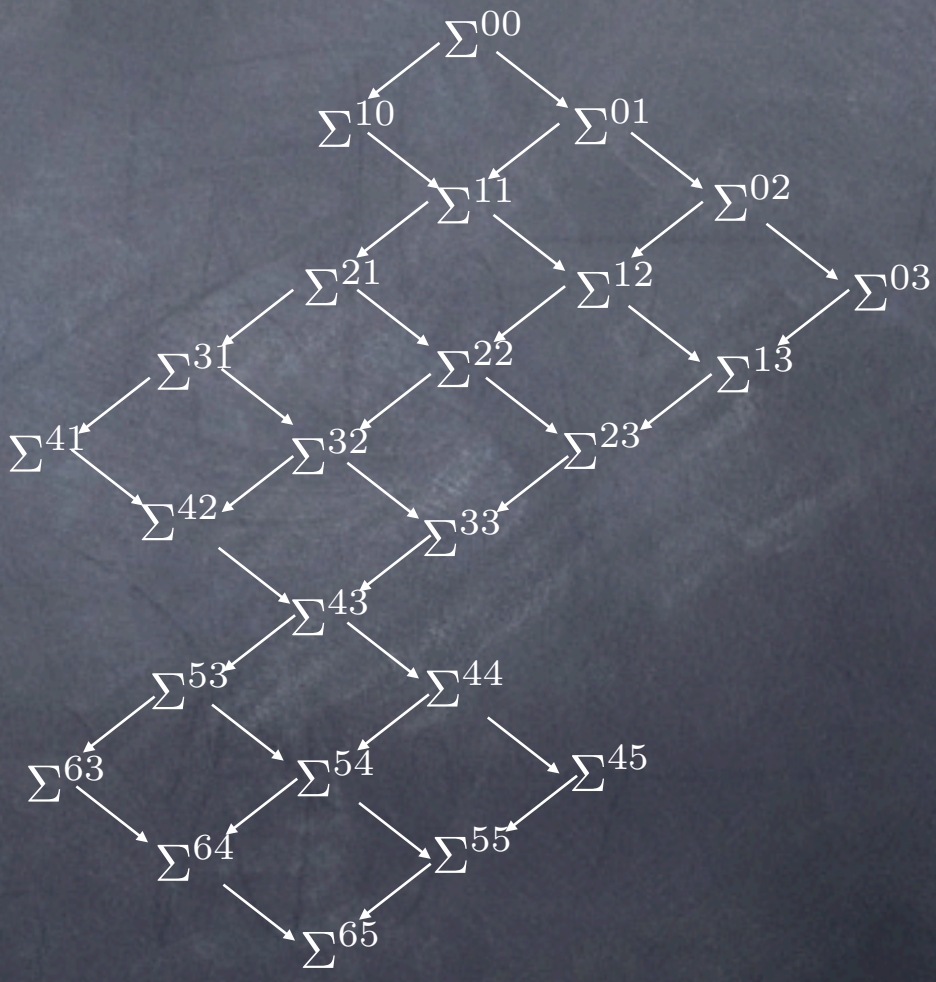
Computing Definitely

- Scan lattice, level after level



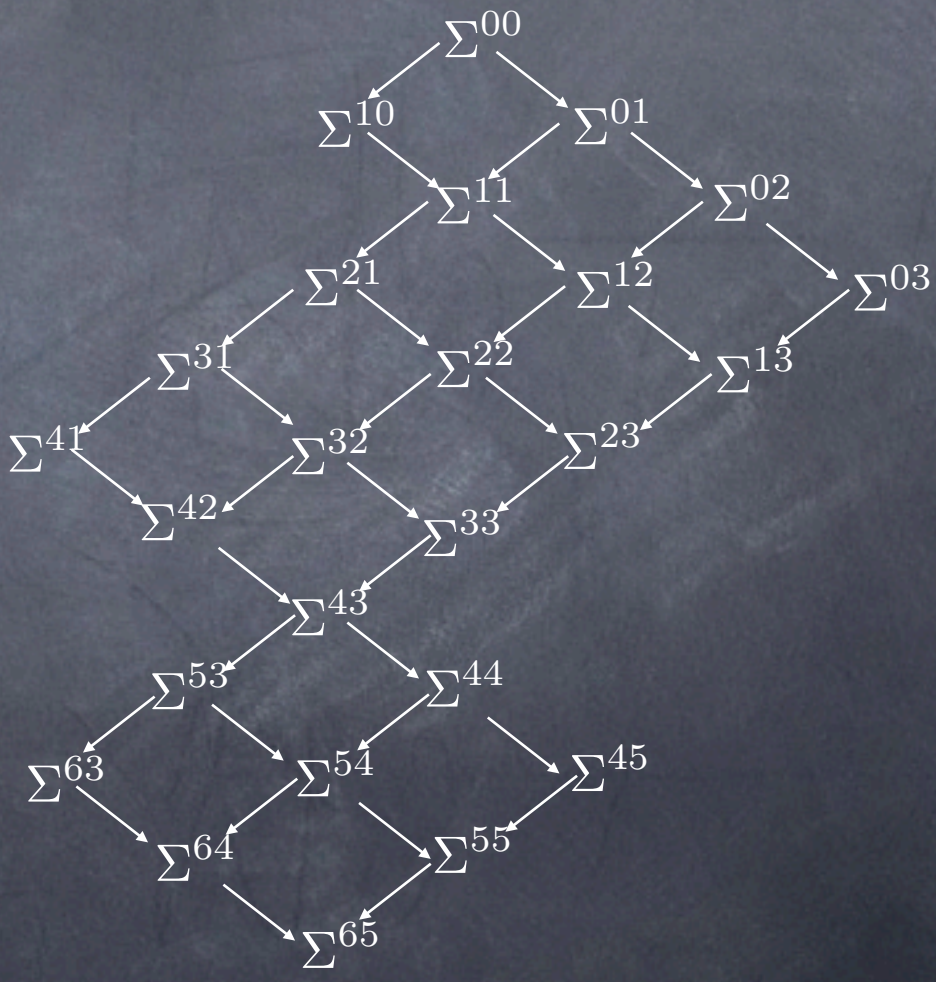
Computing Definitely

- Scan lattice, level after level
- Given a level, only expand nodes that correspond to states for which $\neg\Phi$



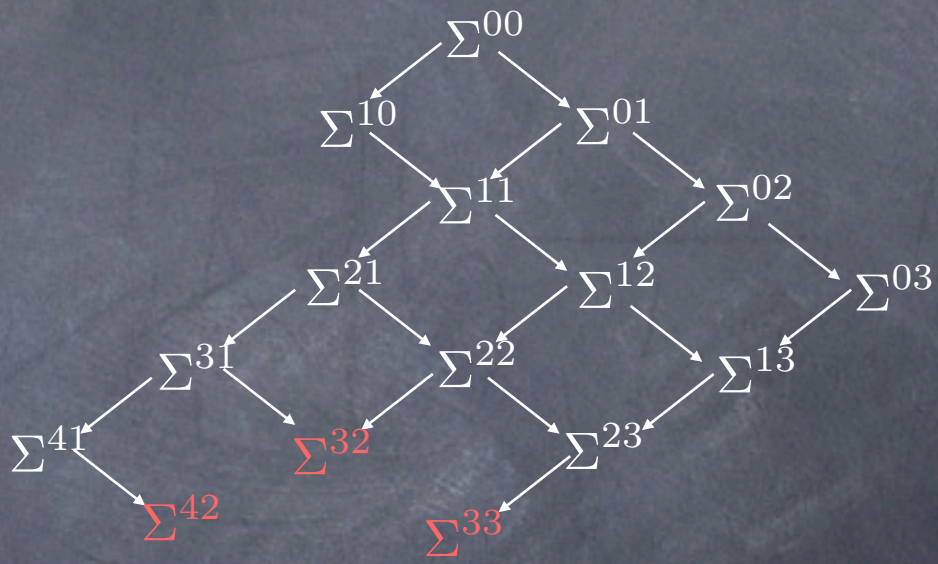
Computing Definitely

- Scan lattice, level after level
- Given a level, only expand nodes that correspond to states for which $\neg\Phi$
- If no such state, then Definitely(Φ)
- If reached last state Σ^l, and $\neg\Phi(\Sigma^l)$, then ¬Definitely(Φ)



Computing Definitely

- Scan lattice, level after level
- Given a level, only expand nodes that correspond to states for which $\neg\Phi$
- If no such state, then Definitely(Φ)
- If reached last state Σ^l, and $\neg\Phi(\Sigma^l)$, then ¬Definitely(Φ)



Definitely $(x = y)$

Building the lattice: collecting local states

- To build the global states in the lattice, p_0 collects local states from each process.
- p_0 keeps the set of local states received from p_i in a FIFO queue Q_i

Key questions:

1. when is it safe for p_0 to discard a local state σ_i^k of p_i?
2. Given level i of the lattice, how does one build level $i+1$?

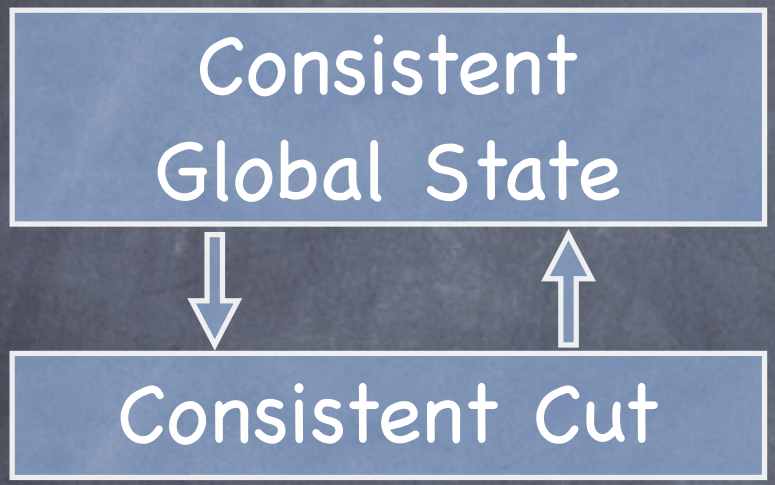
Garbage-collecting local states

- For each local state σ_i^k, we need to determine:
  - $\Sigma_{min}(\sigma_i^k))$, the earliest consistent state that σ_i^k can belong to
  - $\Sigma_{max}(\sigma_i^k))$, the latest consistent state that σ_i^k can belong to

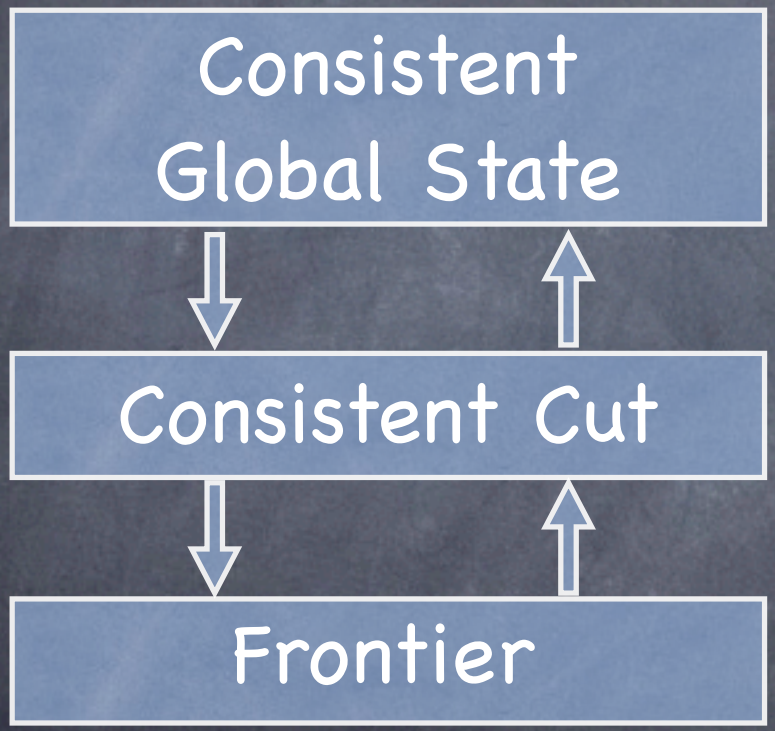
Defining "earliest" and "latest"

Consistent Global State

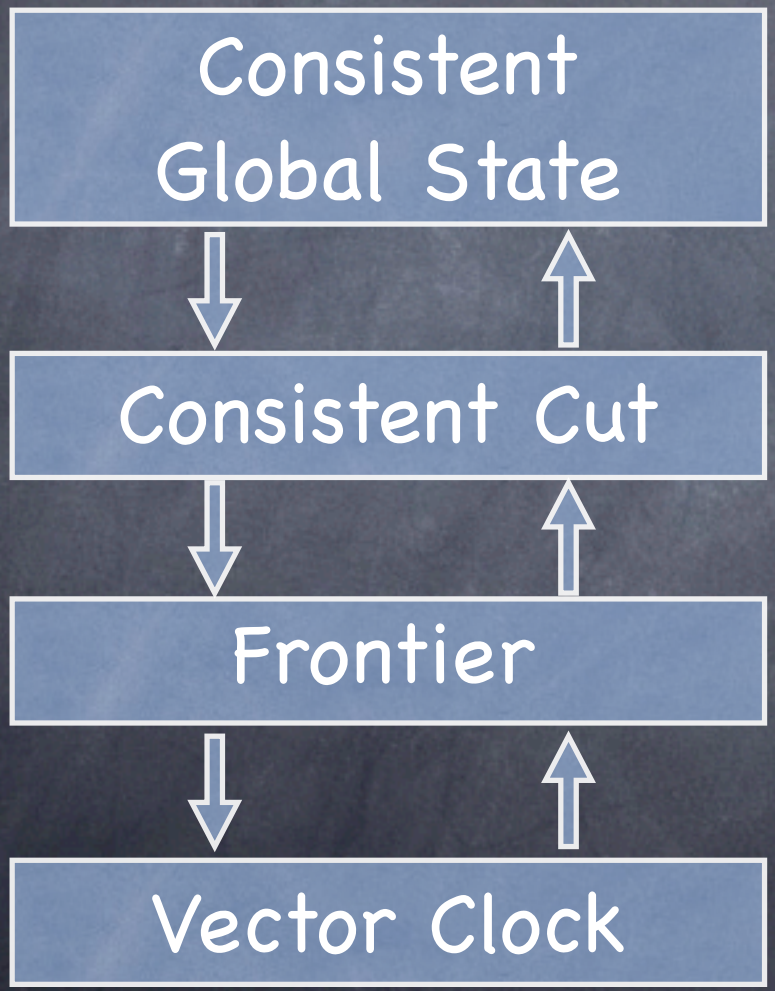
Defining "earliest" and "latest"



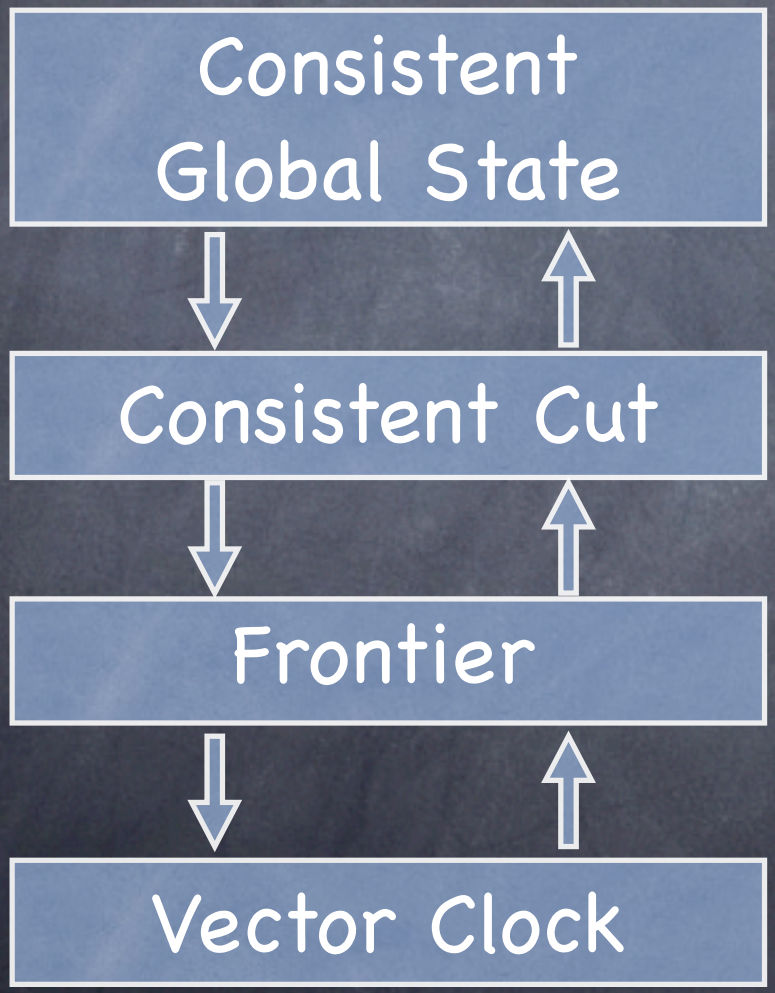
Defining "earliest" and "latest"



Defining "earliest" and "latest"



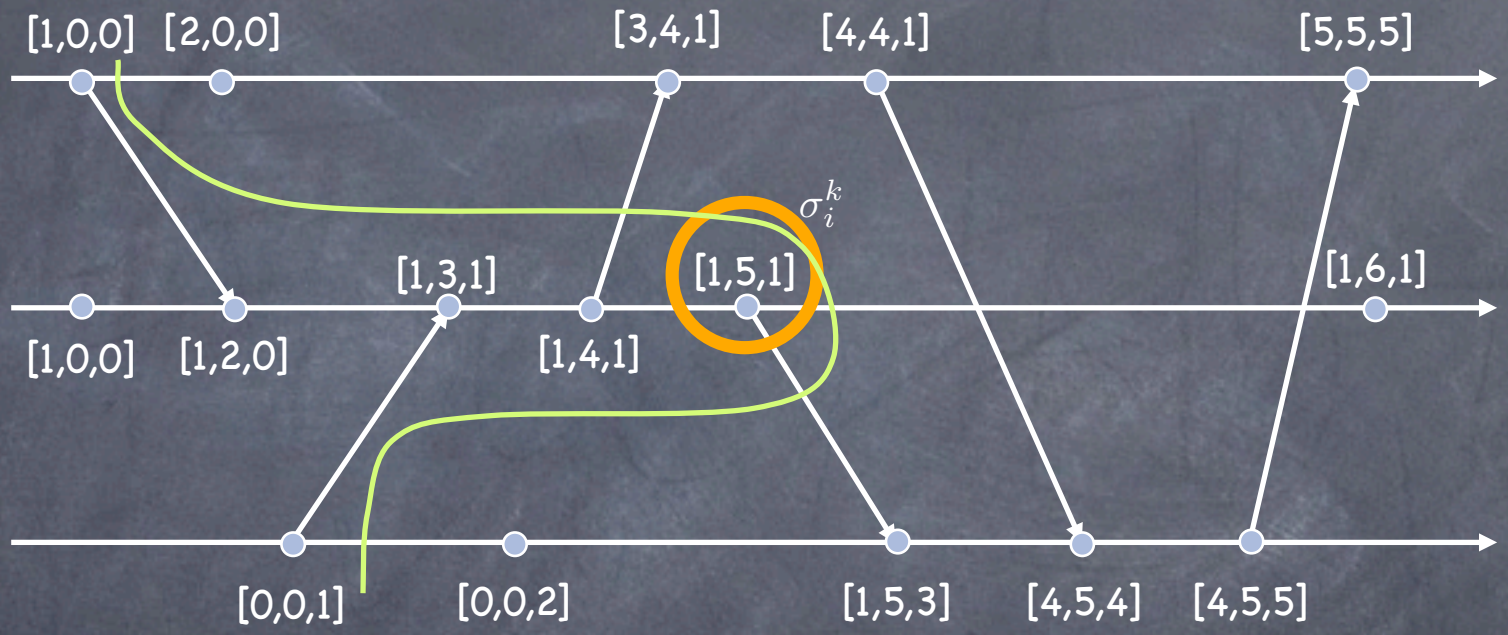
Defining "earliest" and "latest"



Associate a vector clock with each consistent global state

- $\Sigma_{min}(\sigma_i^k))$ is the consistent global state with the lowest vector clock that has σ_i^k on its frontier
- $\Sigma_{max}(\sigma_i^k))$ is the one with the highest

Computing Σ_{min}

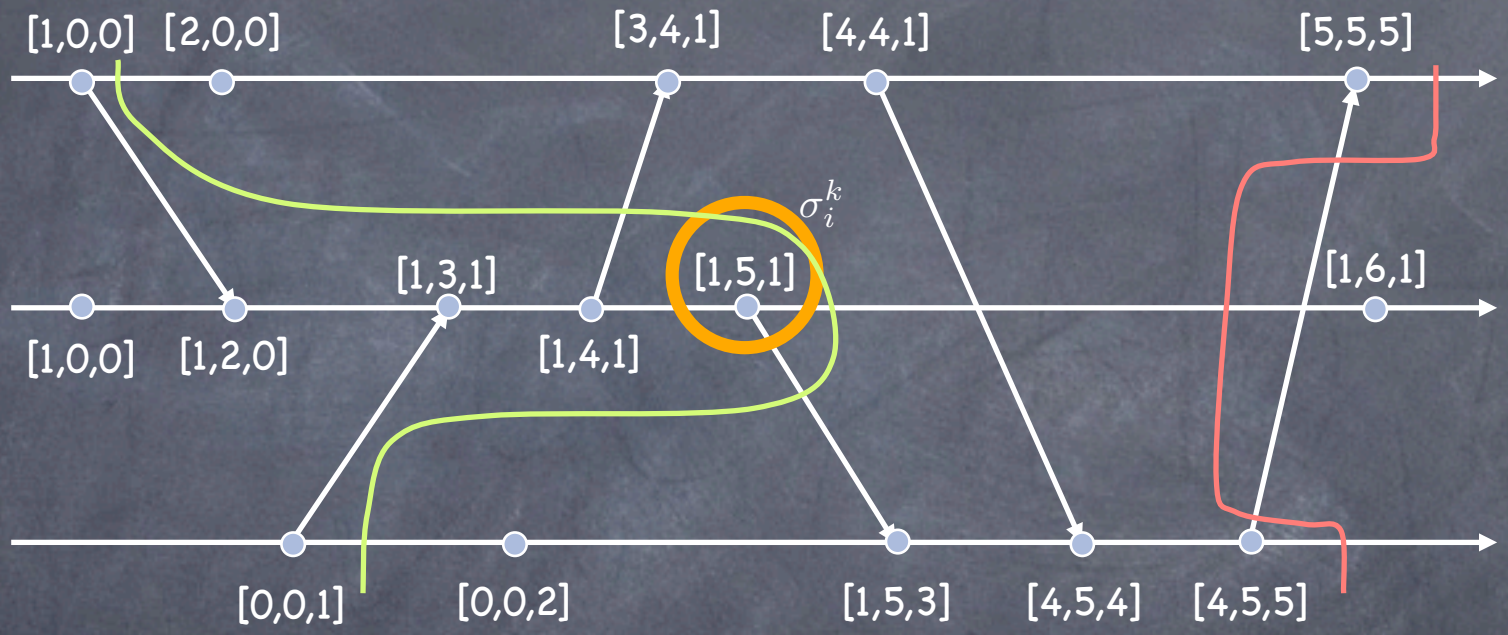


- Label σ_i^k with $VC(e_i^k)$

$$\Sigma_{min}(\sigma_i^k) = (\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots, \sigma_n^{c_n}) : \forall j : c_j = VC(\sigma_i^k)[j]$$

- $\Sigma_{min}(\sigma_i^k)$ and σ_i^k have the same vector clock!

Computing Σ_{max}

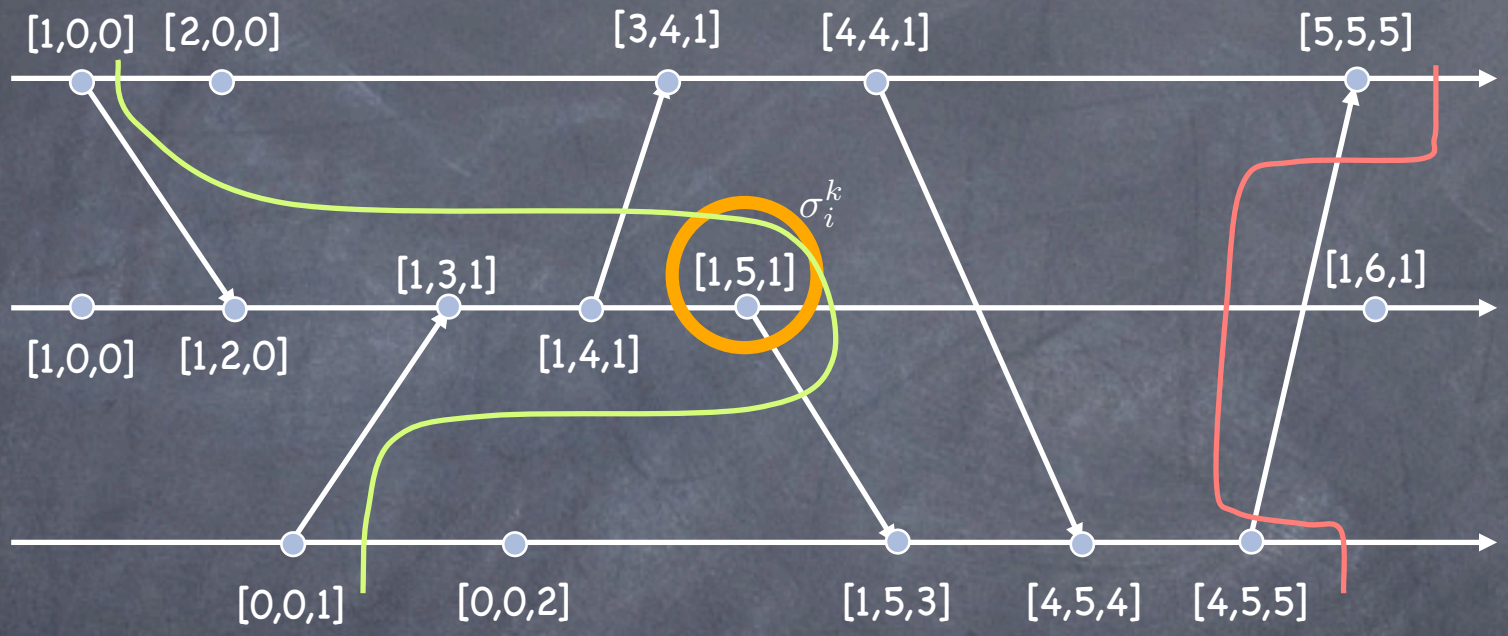


$$\Sigma_{max}(\sigma_i^k) = (\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots \sigma_n^{c_n}) :$$

$$\wedge \forall j : VC(\sigma_j^{c_j})[i] \leq VC(\sigma_i^k)[i]$$

$$\wedge((\sigma_j^{c_j} = \sigma_j^{c_f}) \vee VC(\sigma_j^{c_j+1})[i] > VC(\sigma_i^k)[i]))$$

Computing Σ_{max}



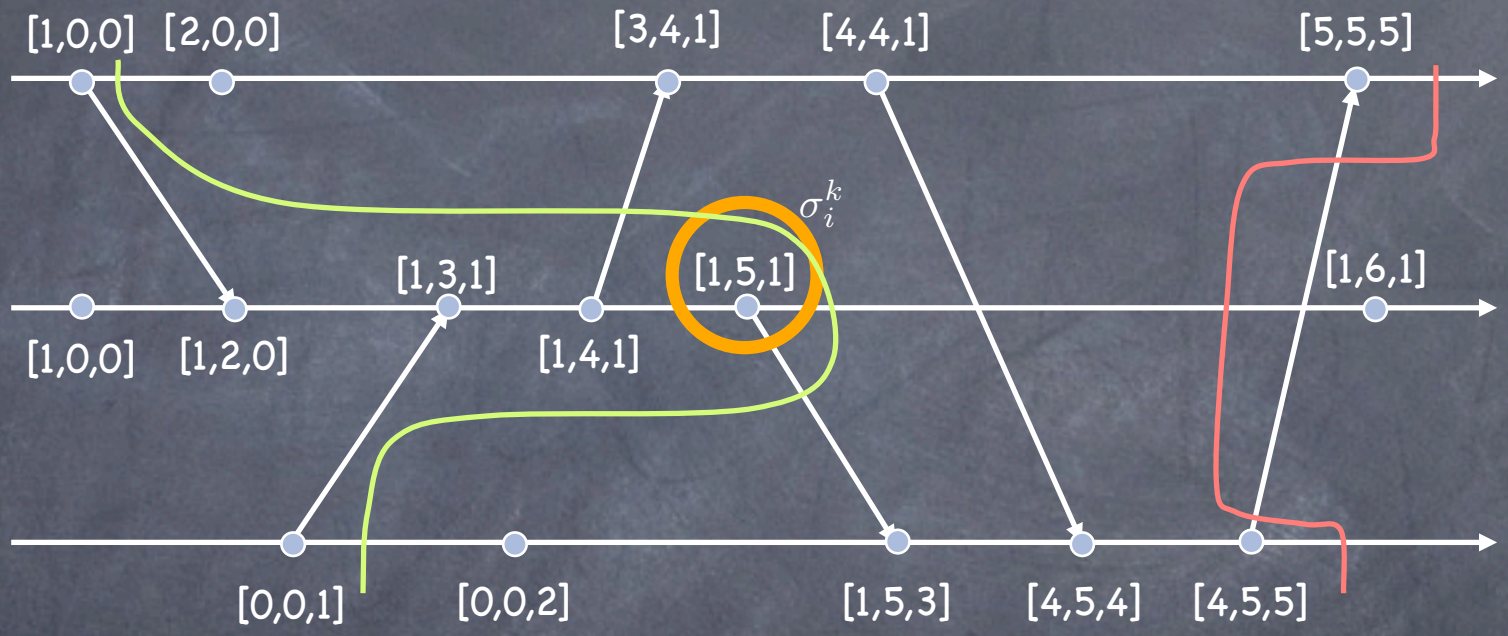
$$\Sigma_{max}(\sigma_i^k) = (\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots \sigma_n^{c_n}) :$$

$$\wedge \forall j : VC(\sigma_j^{c_j})[i] \leq VC(\sigma_i^k)[i]$$

$$\wedge((\sigma_j^{c_j} = \sigma_j^{c_f}) \vee VC(\sigma_j^{c_j+1})[i] > VC(\sigma_i^k)[i]))$$

set of local states one for each process, s.t.

Computing Σ_{max}



$$\Sigma_{max}(\sigma_i^k) = (\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots \sigma_n^{c_n}) :$$

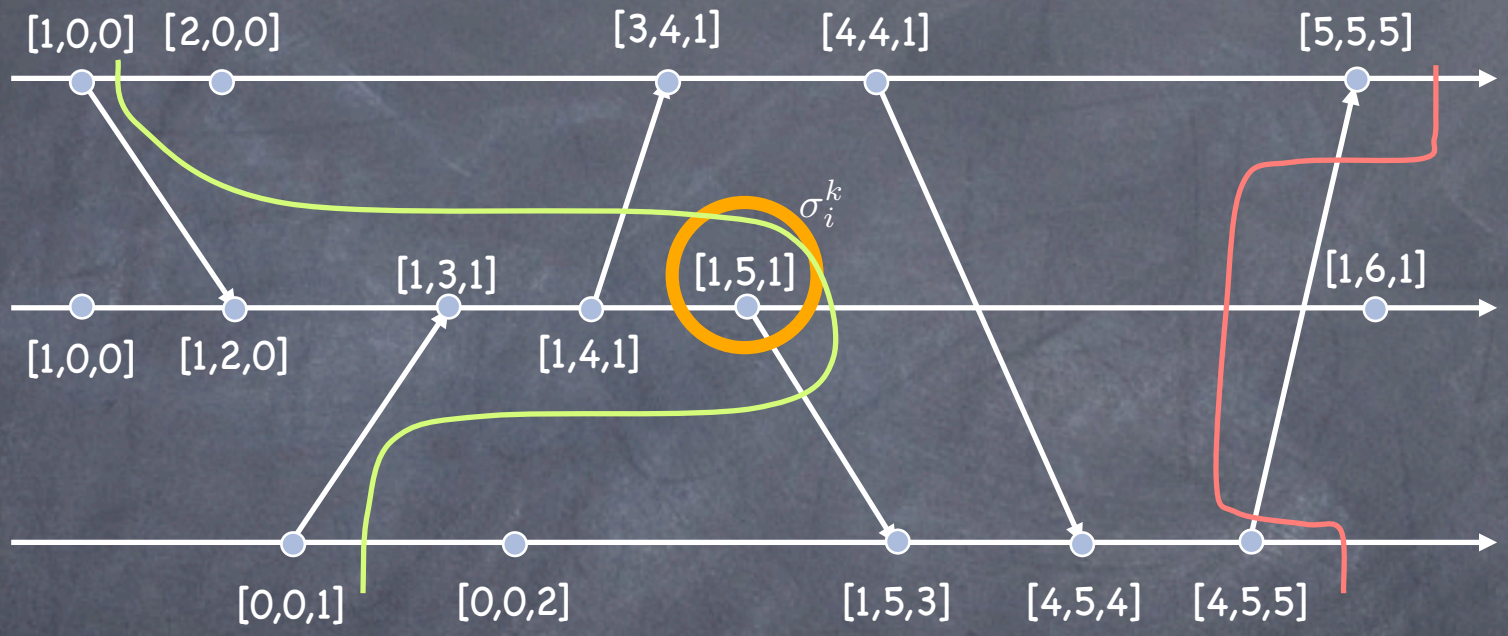
$$\wedge \forall j : VC(\sigma_j^{c_j})[i] \leq VC(\sigma_i^k)[i]$$

$$\wedge((\sigma_j^{c_j} = \sigma_j^{c_f}) \vee VC(\sigma_j^{c_j+1})[i] > VC(\sigma_i^k)[i]))$$

set of local states one for each process, s.t.

all local states are pair-wise consistent with σ_i^k

Computing Σ_{max}



$$\Sigma_{max}(\sigma_i^k) = (\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots \sigma_n^{c_n}) :$$

$$\wedge \forall j : VC(\sigma_j^{c_j})[i] \leq VC(\sigma_i^k)[i]$$

$$\wedge((\sigma_j^{c_j} = \sigma_j^{c_f}) \vee VC(\sigma_j^{c_j+1})[i] > VC(\sigma_i^k)[i]))$$

set of local states one for each process, s.t.

all local states are pair-wise consistent with σ_i^k

and they are the last such state

State-Machine Replication

Modeling faults

- Mean Time To Failure/ Mean Time To Recover
  - close to hardware
- Threshold: f out of n
  - makes condition for correct operation explicit
  - measures fault-tolerance of architecture, not single components
- Set of explicit failure scenarios

www.porchfest.org

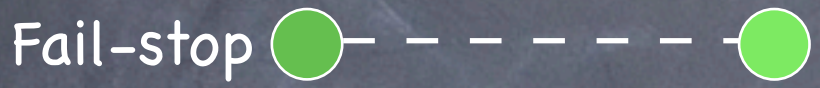
Noon to 6:00 pm

Fall Creek and Northside

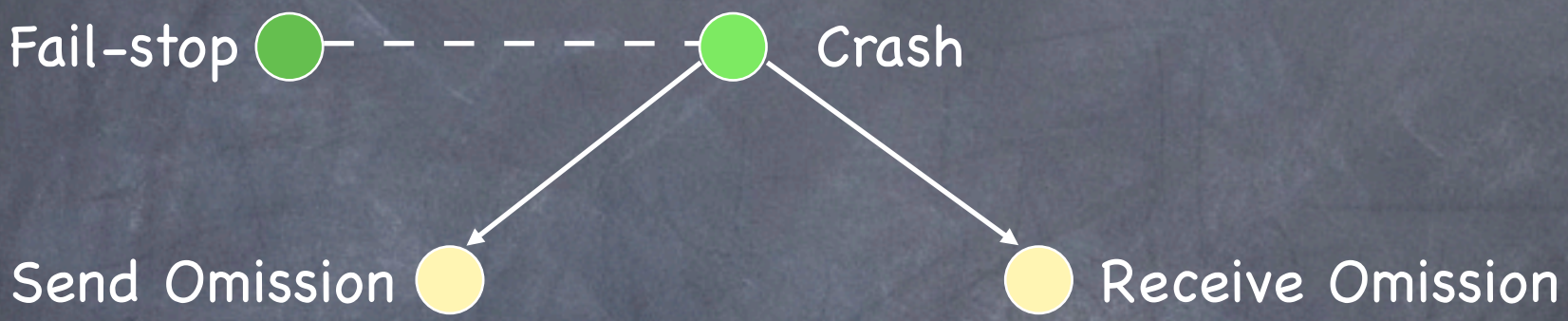
A hierarchy of failure models

- Crash

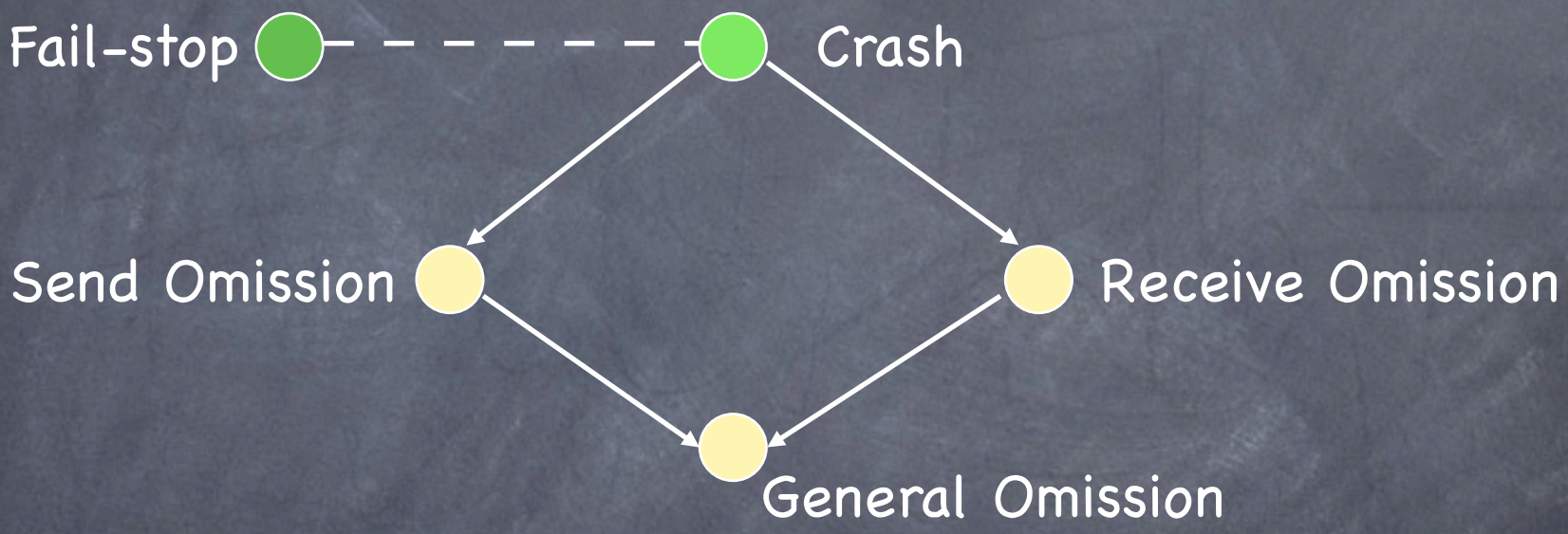
A hierarchy of failure models



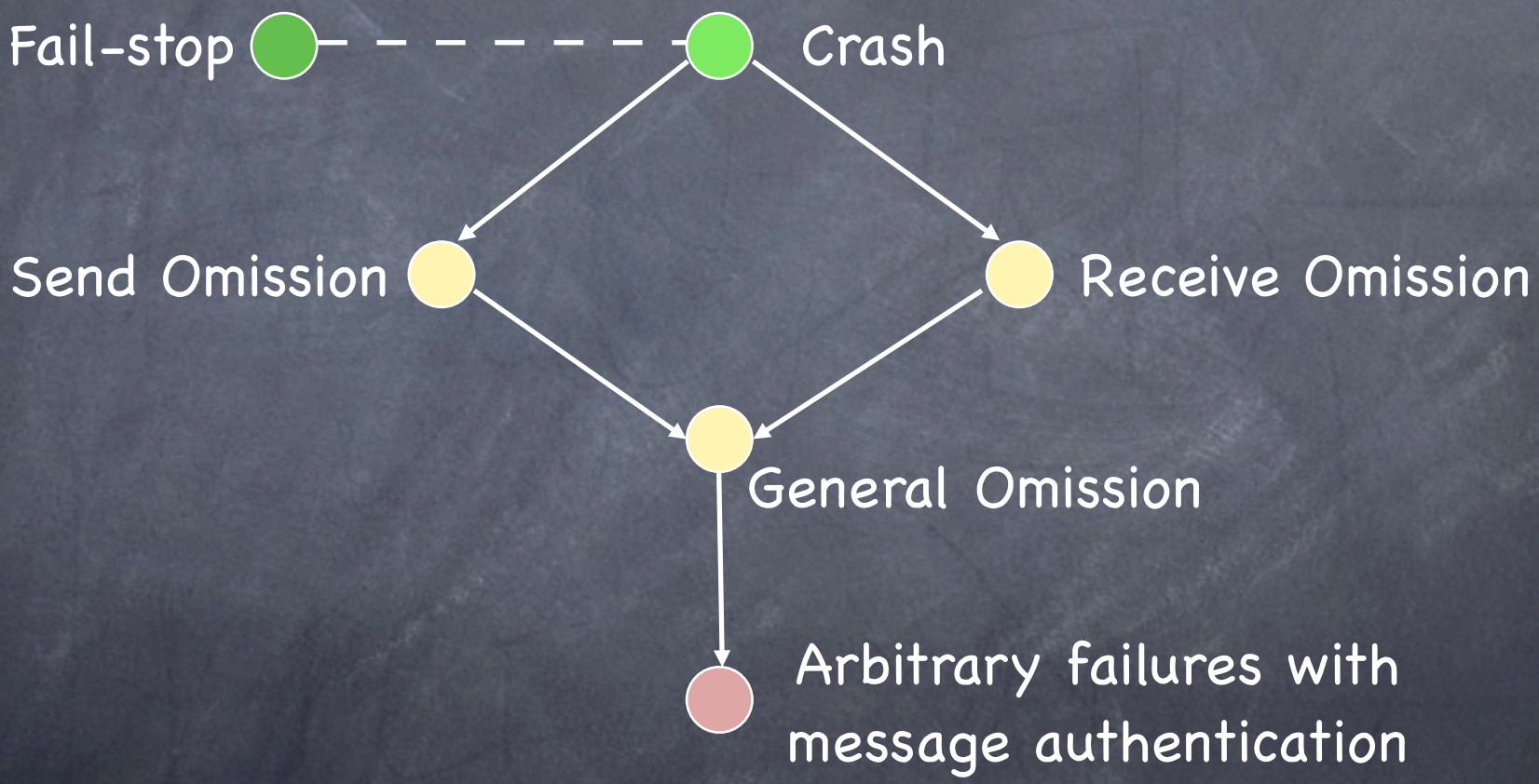
A hierarchy of failure models



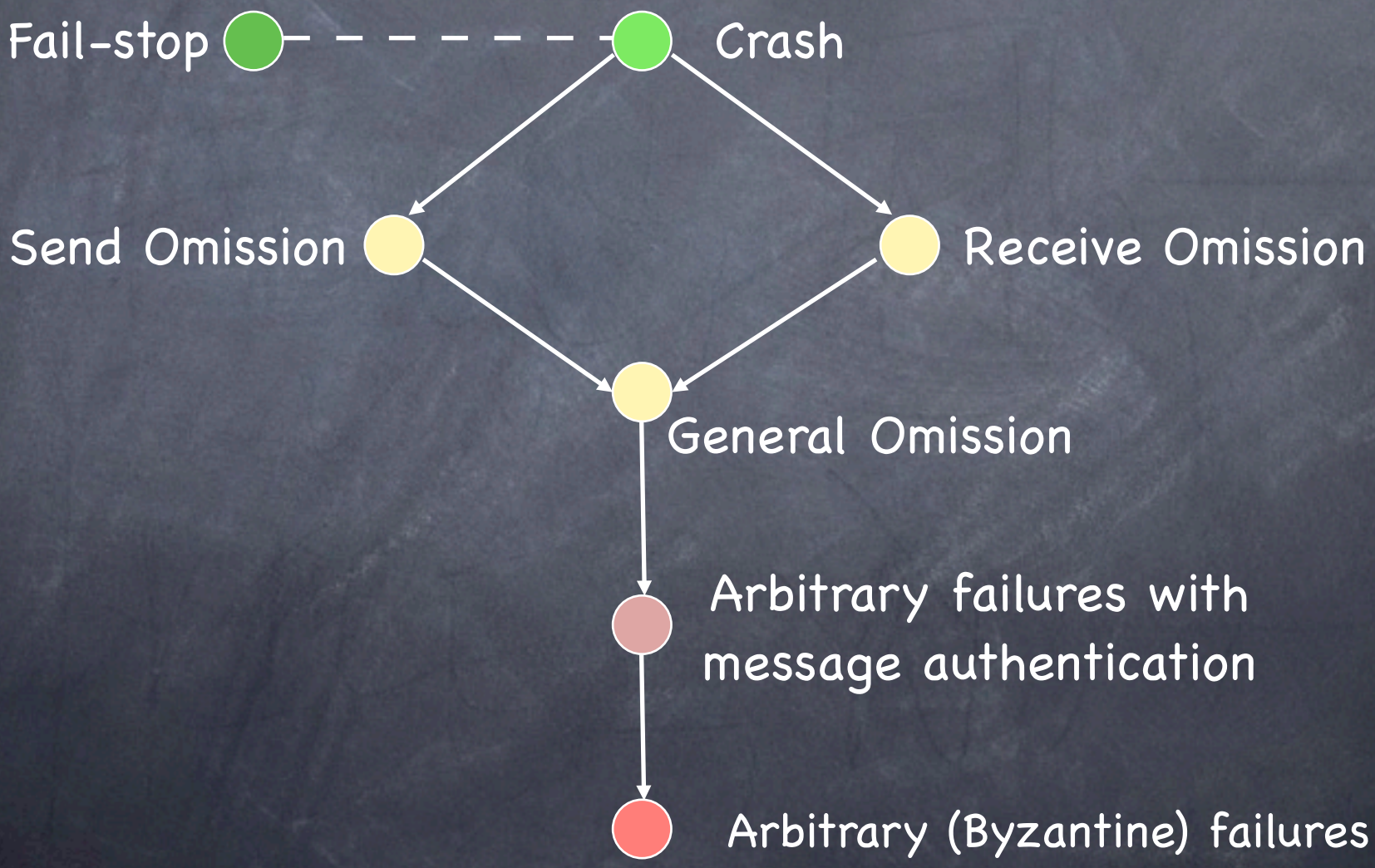
A hierarchy of failure models



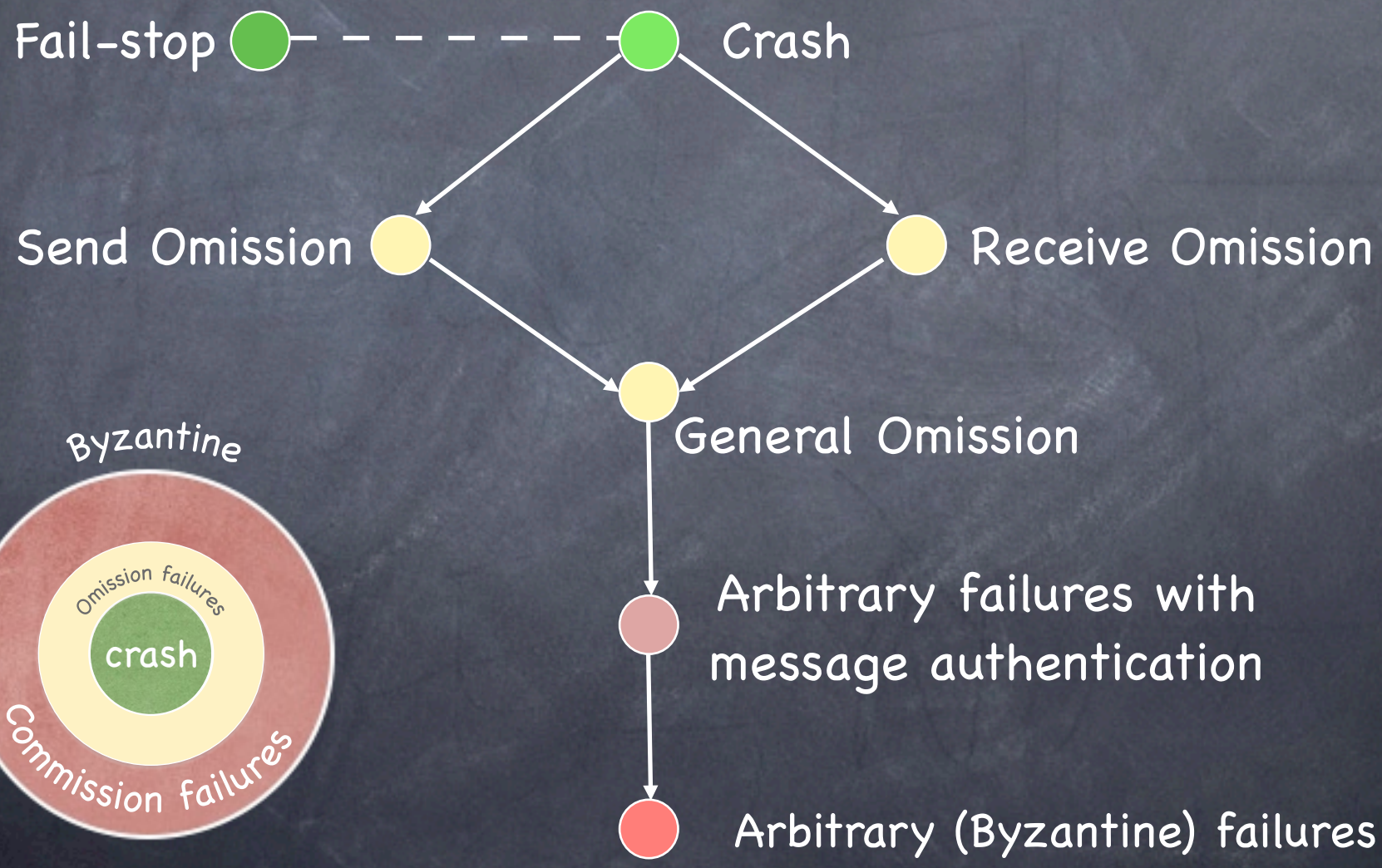
A hierarchy of failure models



A hierarchy of failure models



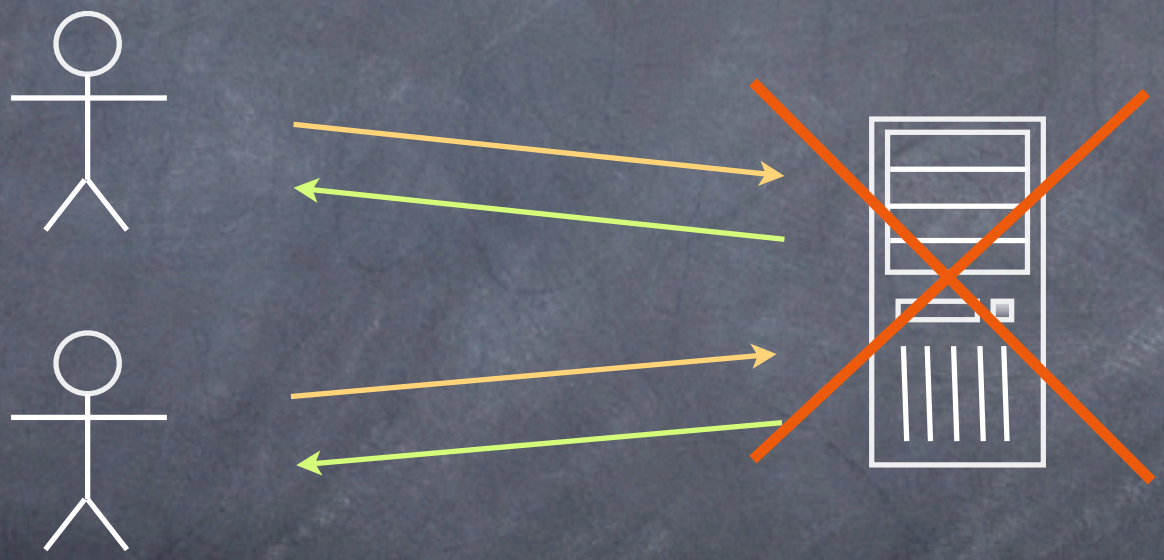
A hierarchy of failure models



The Problem

Clients

Server



Solution: replicate server!

Replication in space

- Run parallel copies of a unit
- Vote on replica output
- Failures are masked
- High availability, but at high cost

Replication in time

- When a replica fails, restart it (or replace it)
- Failures are detected, not masked
- Lower maintenance, lower availability
- Tolerates only benign failures

Non-determinism

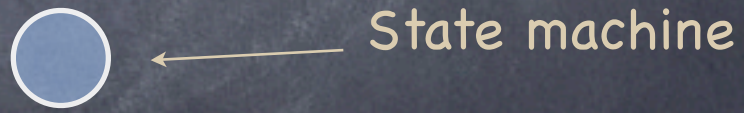
An event is non-deterministic if the state that it produces is not uniquely determined by the state in which it is executed

Handling non-deterministic events at different replicas is challenging

- Replication in time requires to reproduce during recovery the original outcome of all non-deterministic events
- Replication in space requires each replica to handle non-deterministic events identically

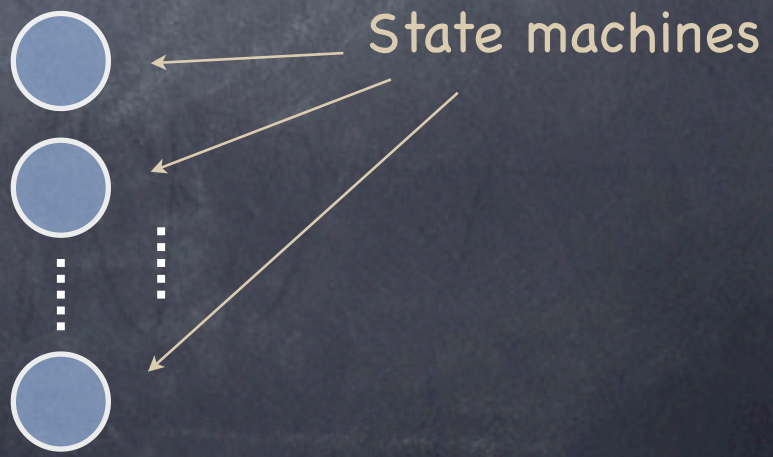
The Solution

1. Make server deterministic (state machine)



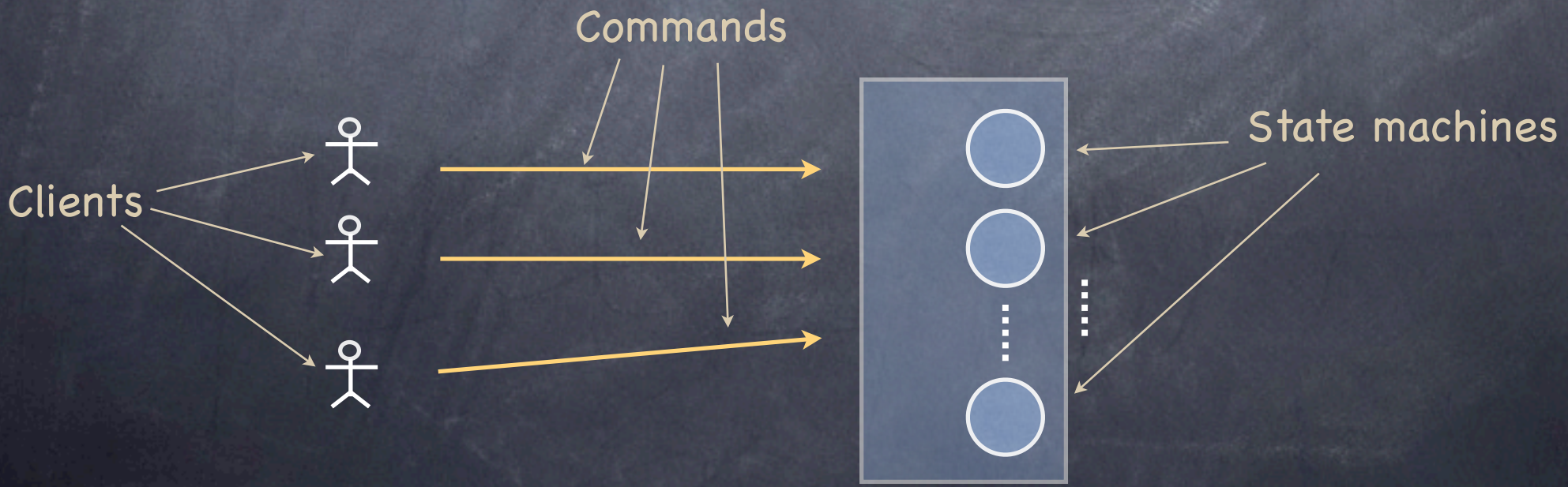
The Solution

1. Make server deterministic (state machine)
2. Replicate server



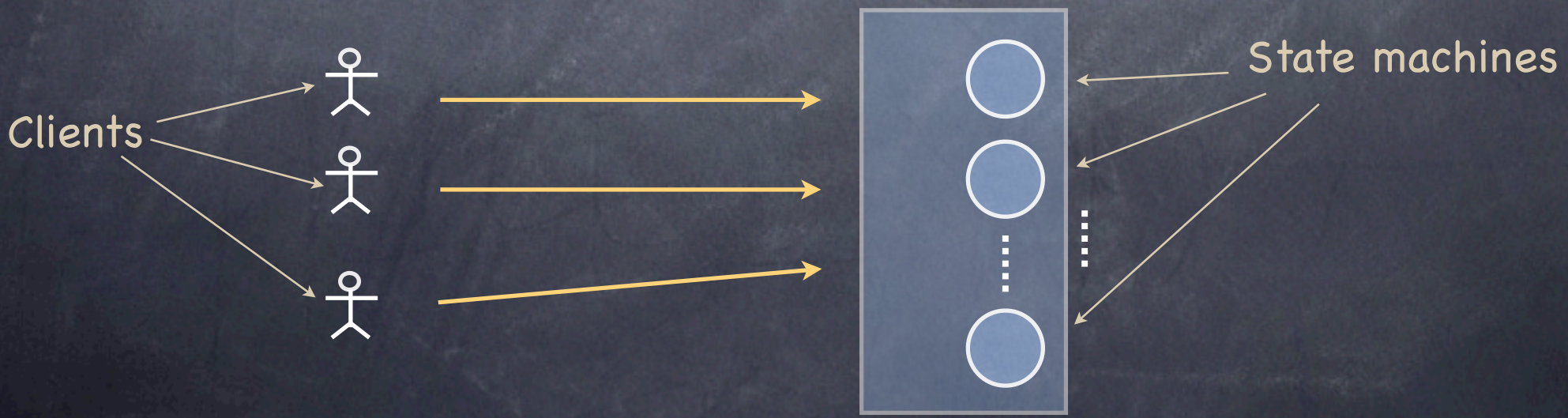
The Solution

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure correct replicas step through the same sequence of state transitions



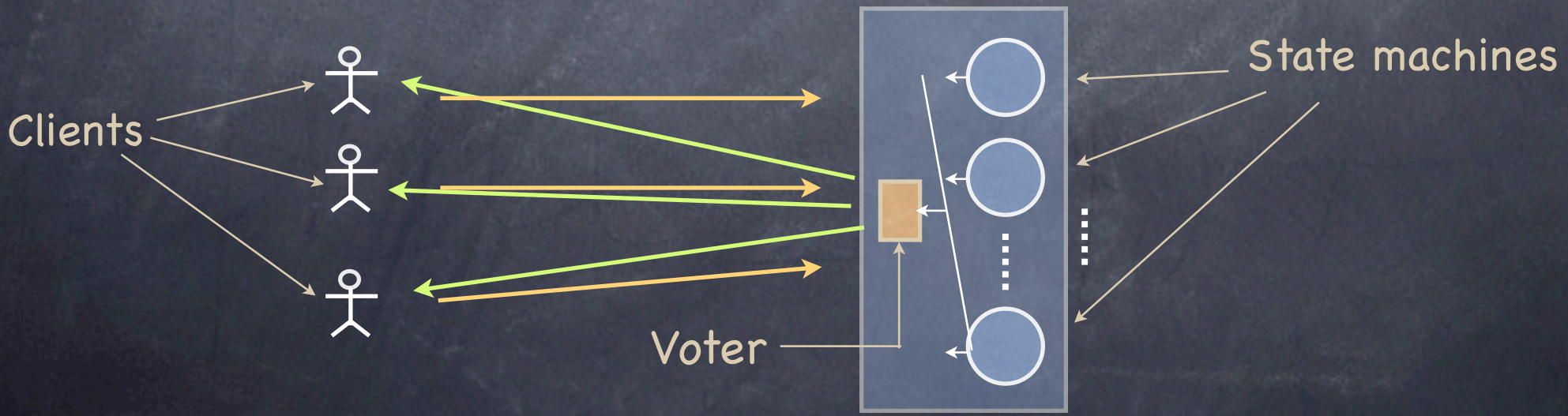
The Solution

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure correct replicas step through the same sequence of state transitions
4. Vote on replica outputs for fault-tolerance

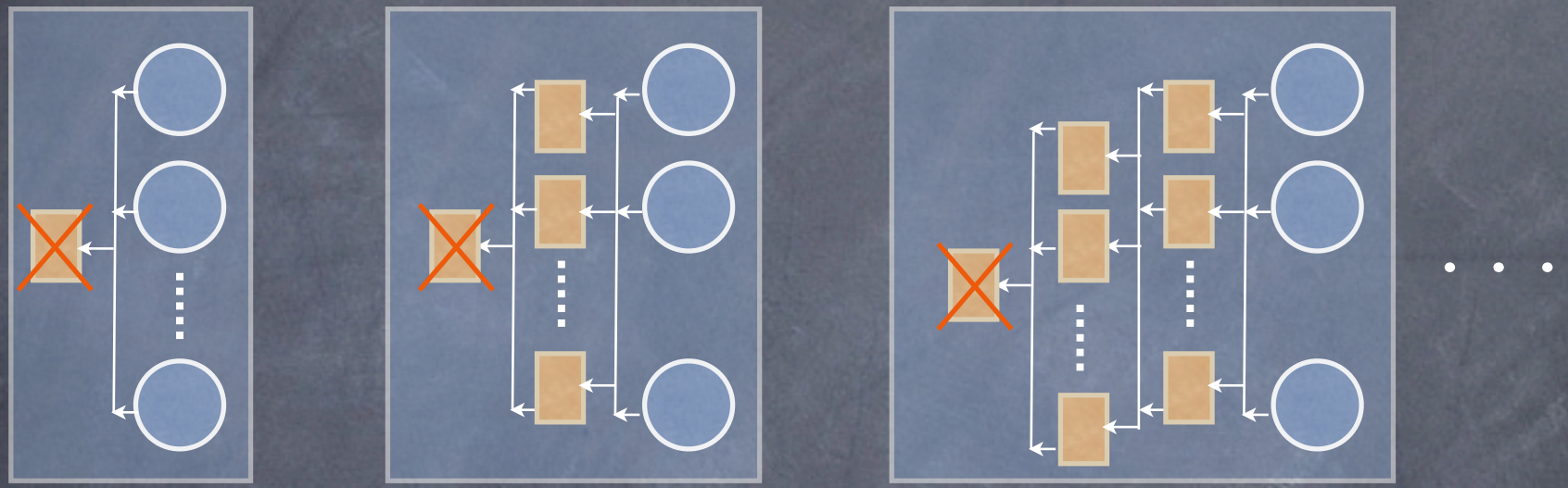


The Solution

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure correct replicas step through the same sequence of state transitions
4. Vote on replica outputs for fault-tolerance

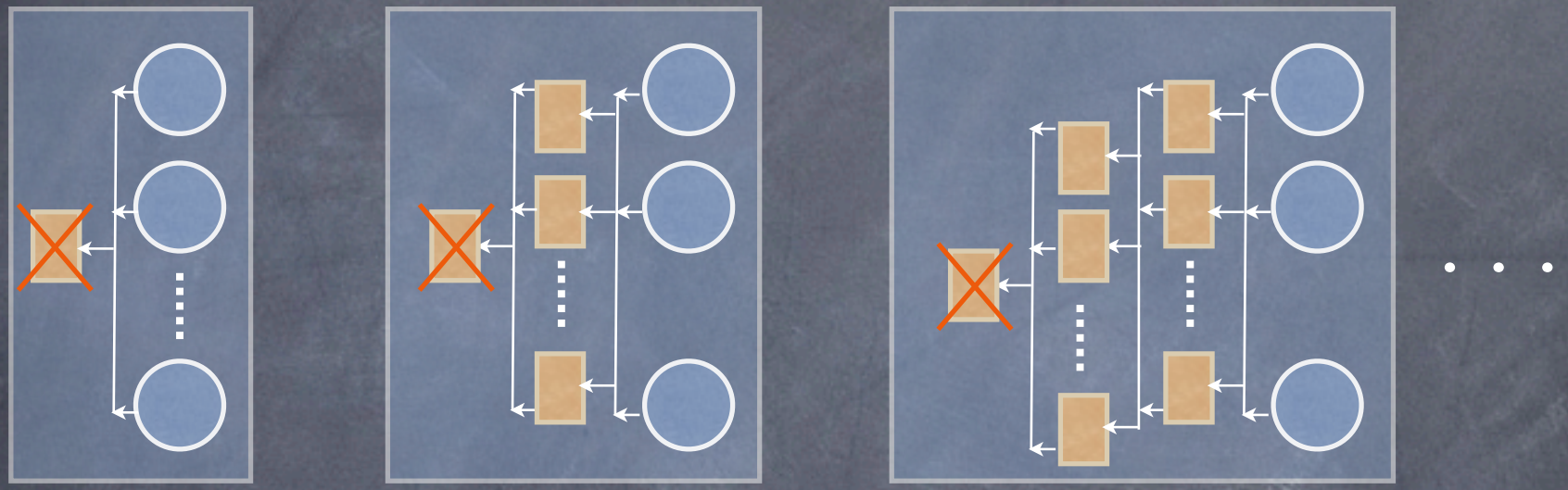


A conundrum

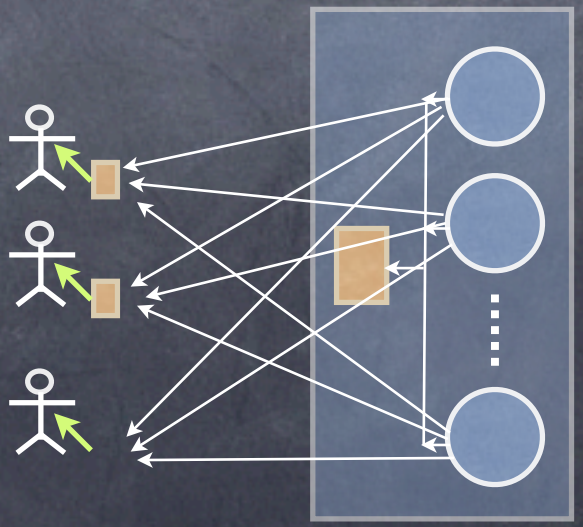


A: voter
and client
share fate!

A conundrum



A: voter
and client
share fate!



State Machines

- Set of state variables + Sequence of commands
- A command
  - Reads its read set values (opt. environment)
  - Writes to its write set values (opt. environment)
- A deterministic command
  - Produces deterministic wsvs and outputs on given rsv
- A deterministic state machine
  - Reads a fixed sequence of deterministic commands

Replica Coordination

All non-faulty state machines receive all commands in the same order

- Agreement: Every non-faulty state machine receives every command
- Order: Every non-faulty state machine processes the commands it receives in the same order

Primary-Backup

The Idea

- Clients communicate with a single replica (primary)
- Primary:
  - sequences clients' requests
  - updates as needed other replicas (backups) with sequence of client requests or state updates
  - waits for acks from all non-faulty clients
- Backups use timeouts to detect failure of primary
- On primary failure, a backup is elected as new primary

Primary-backup and non-determinism

- Non-deterministic commands executed only at the primary
- Backups receive either
  - state updates (non-determinism?)
  - command sequence (non-determinism?)