

Consistent global states

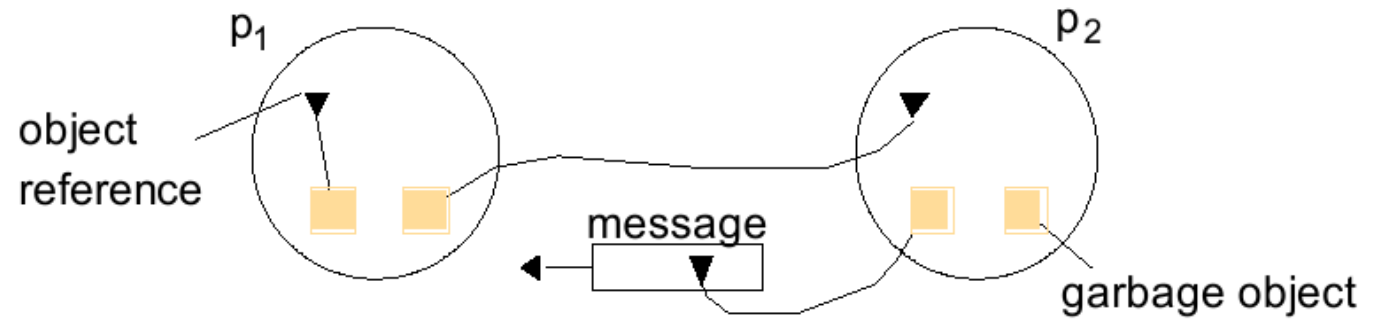
Definizione 1 (Global Predicate Evaluation, GPE). *The problem of detecting whether the global state of a distributed system satisfies some predicate Φ .*

Motivation

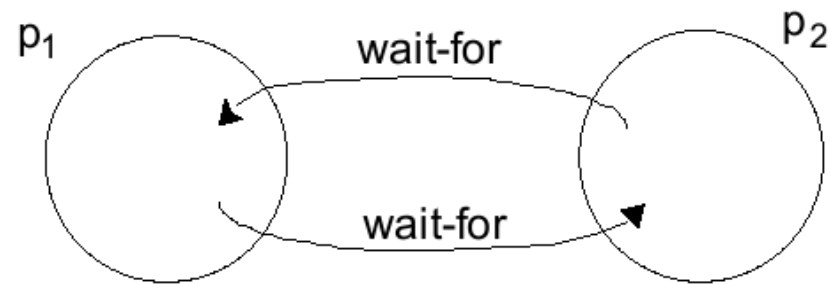
- Many important distributed problems require to execute some notification or reaction when the global state of the system satisfies a given condition.
 - **Monitoring**: Notify an administrator in case of failures
 - **Debugging**: Verify whether an invariant is respected or not
 - **Deadlock detection**: can the computation continue?
 - **Garbage collection**: like Java, but distributed
- Thus, the *ability to construct a global state* and evaluate a predicate over such a state is a core problem in distributed computing.

Examples

a. Garbage collection



b. Deadlock



Why GPE is difficult

A global state obtained through remote observations could be

- **obsolete**: represent an old state of the system.

Solution: build the global state more frequently

- **incomplete**: not “capture” every moment of the system

Solution: build all possible global states

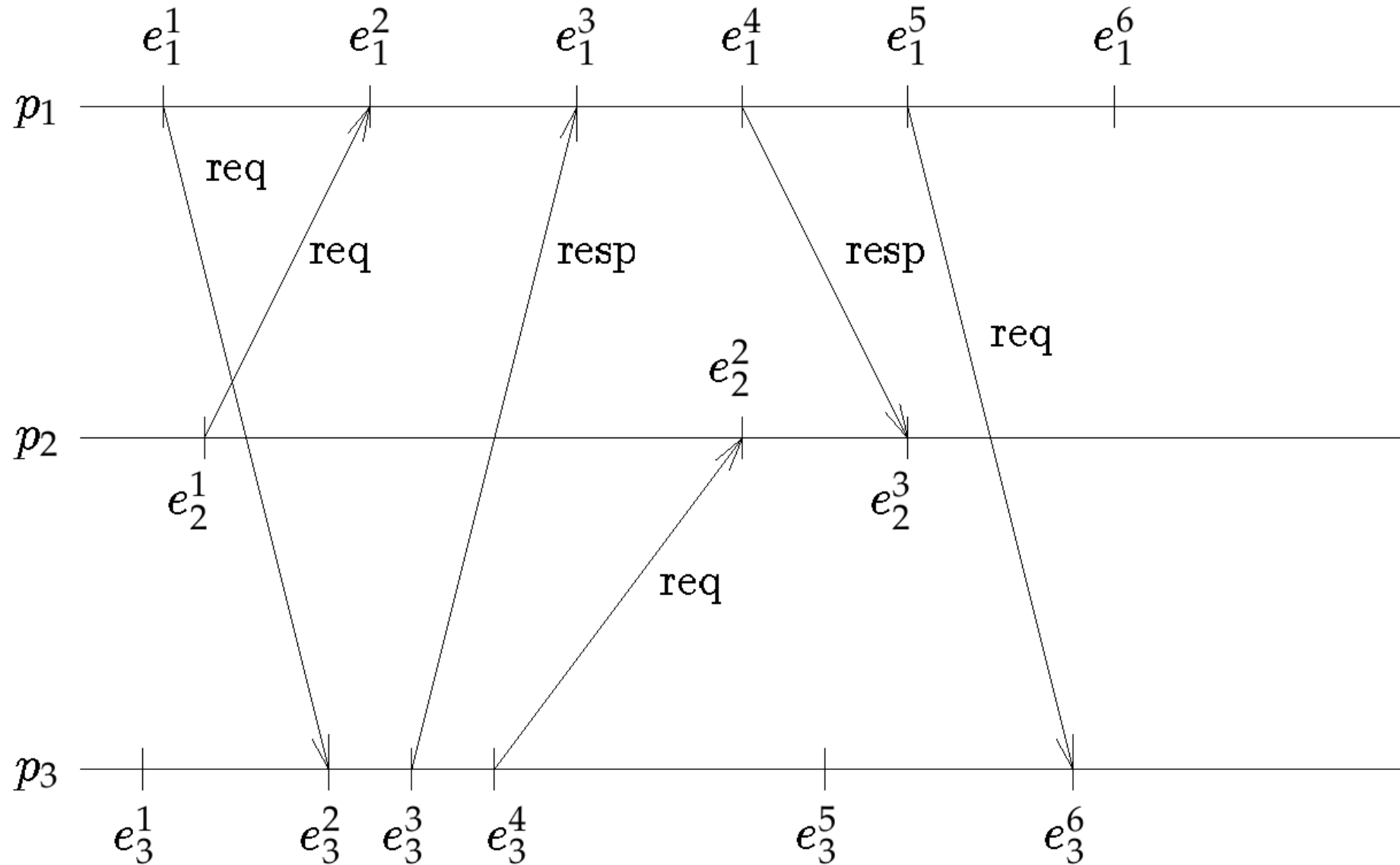
- **inconsistent**: informally, a global state is inconsistent if it could never have been constructed by an idealized observer external to the system.

Possible causes: Asynchrony of processes and messages.

- From the point of view of any single process, events are ordered uniquely by times shown on the local clock.
- No perfect synchronization across a distributed system \Rightarrow there is no way to use physical time to order any arbitrary pair of events.

Solution: described in this lecture.

Space-Time Diagram of a Distributed Computation



An example - Multi-tier system based on RPC, deadlock detection

Processes in the previous figures use RMIs:

- Client sends a *request* for method execution; blocks.
- Server receives *request*.
- Server executes method; may invoke other methods in other servers, acting as a client.
- Server sends *reply* to client
- Clients receives *reply*; unblocks.

Such a system can deadlock, as RMIs are blocking. It is important to be able to detect when a deadlock occurs.

Monitoring Distributed Computation

- Assumptions (relaxed later):
 - There is a single process p_0 called **monitor** which is responsible for evaluating Φ .
 - We assume that p_0 is distinct from $p_1 \dots p_n$
 - Events executed on behalf of monitoring do not alter canonical enumeration of “real” events.
- Two possible approaches:
 - Proactive monitoring (take a “snapshot”)
 - Passive monitoring (receive notifications)

Run

A **run** of global computation is a total ordering R that includes all the events in the global history and that is consistent with each local history.

- In other words, the events of p_i appear in R in the same order in which they appear in h_i .
- A run corresponds to the notion that events in a distributed computation actually occur in a total order
- A distributed computation may correspond to many runs

A run R is said to be **consistent** if for all events e and e' , $e \rightarrow e'$ implies that e appears before e' in R .

A Passive Approach to GPE

How it works

- At each (relevant - state change) event, a node sends a message to the monitor describing its local state
- The monitor collects messages to reconstruct the global state.

The sequence of events corresponding to the order in which notification messages arrive at the monitor is called an **observation**.

Given the asynchronous nature of our distributed system, *any* permutation of a run R is a possible observation of it.

Observations vs Runs

Definition (**Consistent Observation**) An observation is consistent if it corresponds to a consistent run.

An observation can correspond to:

- A consistent run
- A run which is not consistent
- No run at all

Can you find example of the three cases? Can you explain why this happen?

FIFO Order

Problem Observations may not correspond to a run because a channel between a pair of process may re-order messages in any possible way.

Definition: FIFO Delivery - First-in, First-Out Two messages sent by p_i to p_j must be delivered in the same order in which they were sent:

$$\forall m, m' : \text{send}_i(m) \rightarrow \text{send}_i(m') \Rightarrow \text{deliver}_j(m) \rightarrow \text{deliver}_j(m')$$

FIFO Delivery is not sufficient

Problem If we use FIFO delivery, all the observations taken by p_0 will be *runs*; but there is no guarantee that they are *consistent runs*.

Solution We devise a very simple mechanism, based on strong assumptions, and then we refine it by relaxing those assumptions.

Initial assumptions

- All processes have access to a real-time clock RC ;
let $RC(e)$ be the real-time at which e is executed;
- All messages are delivered within a time δ
- The timestamp of a message m sent through an event $e = send(m)$ is
 $TS(m) = RC(e)$.

DR1: Real-time delivery rule At time t , delivery all received messages m such that $TS(m) \leq t - \delta$ in increasing timestamp order.

Observation O constructed by p_0 using DR1 is guaranteed to be **consistent**.

Safety: Clock Condition for RC $e \rightarrow e' \Rightarrow RC(e) < RC(e')$

Note that $RC(e) < RC(e') \not\Rightarrow e \rightarrow e'$, but this rule is sufficient to obtain consistent observations, as two events $e \rightarrow e'$ are never delivered in the incorrect order.

Liveness: Gap Detection for RC Given two events e and e' along with their clock values $RC(e) < RC(e')$, determine whether some other event e'' exists such that $RC(e) < RC(e'') < RC(e')$.

Note that real-time clocks do not support gap detection; it is the maximum delay of messages that enables it.

Scalar Logical Clocks

Safety: Clock Condition for LC $e \rightarrow e' \Rightarrow LC(e) < LC(e')$

Note that $LC(e) < LC(e') \not\Rightarrow e \rightarrow e'$, but this rule is sufficient to obtain consistent observations, as two events $e \rightarrow e'$ are never delivered in the incorrect order.

Note:

- We say that the timestamp of e is $LC(e)$ without specifying the process;

Good! Are we done? Is this sufficient to write a delivery rule?

Logical Clocks

Problem Logical clocks do not guarantee *gap detection*. So, no message will ever be delivered for fear of receiving a later message with a smaller timestamp.

Solution Additional information is needed; the concept of δ must be reproduced in an asynchronous system.

Stability A message m received by p is called **stable at p** if no message m' with $TS(m') < TS(m)$ can be received in the future by p .

DR2: Deliver Rule for LC Deliver all received messages that are stable at p_0 in increasing timestamp order.

Logical Clocks

Implementation

- Each node communicates with p_0 using FIFO delivery
- When p_0 receives a message from p_i describing an event e with timestamp $TS(e)$, it is sure that it will never receive a message from p_i describing an event e' with $TS(e') \leq TS(e)$.
- Stability of message m at p_0 can be guaranteed when p_0 has received at least one message from all other processes with a timestamp greater or equal than $TS(m)$.

Problems of Logical Clocks

- They add unnecessary delays to observations
- They require a constant flux of messages/events from all nodes
- They only apply to one process (p_0); they do not apply to other processes.

Limitations of Logical Clocks

Clock Condition is not sufficient

- Logical Clocks guarantee that $e \rightarrow e' \Rightarrow LC(e) < LC(e')$;
- This means that $LC(e) < LC(e') \Rightarrow e' \not\rightarrow e$;
- This means that if we receive e' before e , we must delay e' , even if we could predict the timestamps of all events yet to be received
- Smaller timestamps: may be concurrent or may have happened-before

We need a clock mechanism C satisfying:

Strong Clock Condition: $e \rightarrow e' \Leftrightarrow C(e) < C(e')$;

Causal delivery and consistent observations

If p_0 uses a delivery rule satisfying Causal Delivery, then all of its observations will be consistent.

Informal Proof

Definition of Causal Delivery \equiv Definition of a consistent observation

Properties of Vector Clocks

Consistent Cut A cut defined by (c_1, \dots, c_n) is consistent if and only if:

$$\forall i, j \in [1 \dots n] : VC(e_i^{c_i})[i] \geq VC(e_j^{c_j})[i]$$

In other words, a cut is consistent if its frontier does not contain any pairwise inconsistent pair of events.

Example: ?

Properties of Vector Clocks

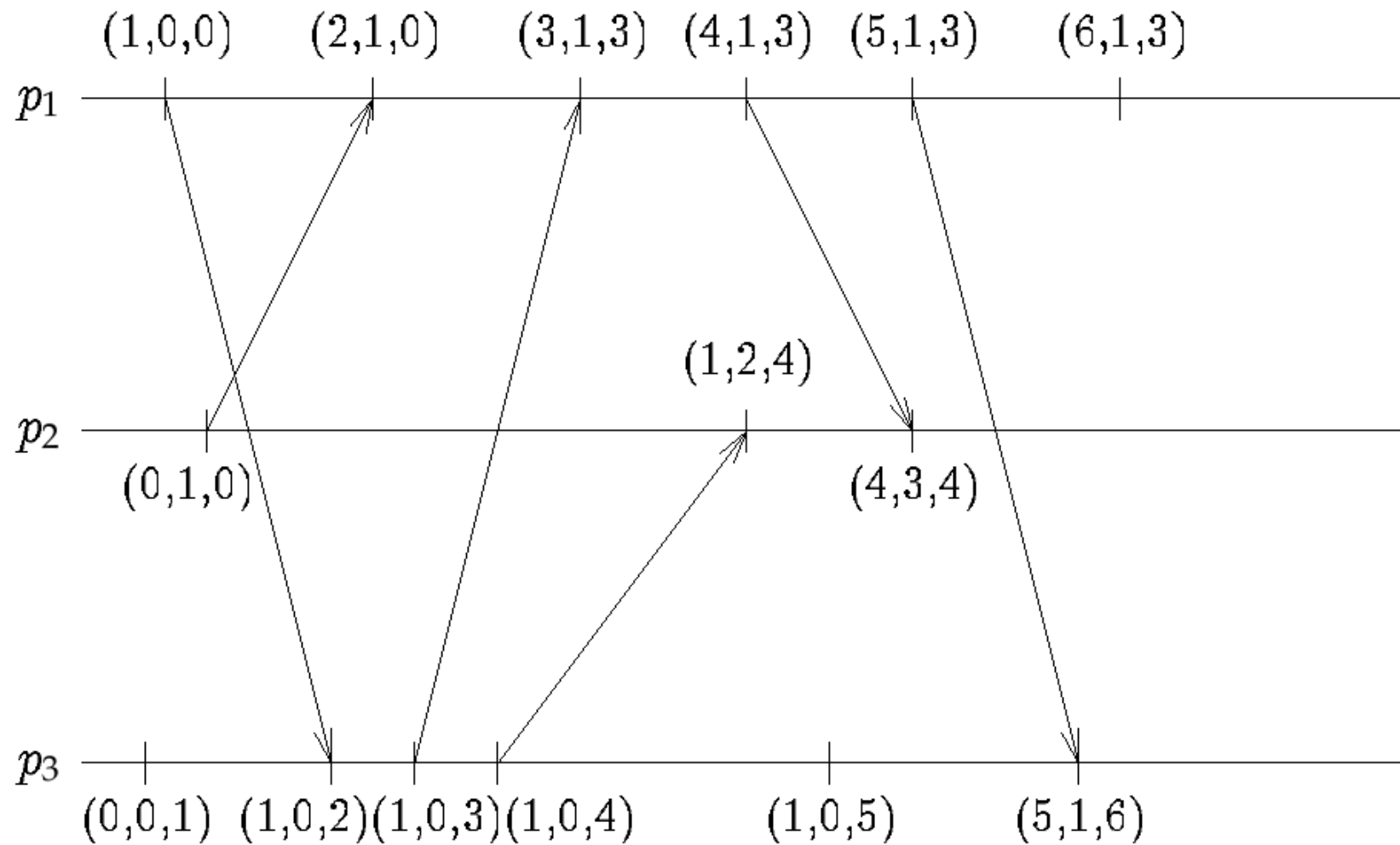
Weak Gap Detection Given events e_i and e_j , if $VC(e_i)[k] < VC(e_j)[k]$ for some $k \neq j$, then there exists an event e_k such that:

$$(e_k \not\rightarrow e_i) \wedge (e_k \rightarrow e_j)$$

- This property derives from the vector clock update rule and the Strong Clock Condition.
- It can be used to determine if the causal gap between two events admits a third event.
- For arbitrary processes p_i, p_j, p_k , it does not enable to say that $e_i \rightarrow e_k \rightarrow e_j$;
- For $i = k$, $e_i \rightarrow e_k \rightarrow e_j$.

Weak Gap Detection

$$VC(e_i)[k] < VC(e_j)[k] \wedge k \neq j \Rightarrow e_k \not\rightarrow e_i \wedge e_k \rightarrow e_j$$



Implementing Causal Delivery with Vector Clocks

Variables maintained at p_0

- \mathcal{M} : the set of messages received but not yet delivered to p_0
- an array D , initialized to 0's, such that $D[k]$ contains $TS(m)[k]$ where m is the last message delivered by p_0 from process p_k .

When a message is deliverable? A message $m \in \mathcal{M}$ from process p_j is deliverable as soon as p_0 can verify that there is no other message m' (neither in \mathcal{M} nor in the channels) such that $send(m') \rightarrow send(m)$.

Implementing Causal Delivery with Vector Clocks

Two conditions to be verified:

- There is no earlier message from p_j that has not been delivered yet.

How can we express this condition?

- $\forall k \neq j$, let m' be the last message from p_k delivered by p_0 ($D[k] = TS(m')[k]$); we must be sure that no message m'' from p_k exists such that: $send_k(m') \rightarrow send_k(m'') \rightarrow send_j(m)$

How can we express this condition?

Implementing Causal Delivery with Vector Clocks

Two conditions to be verified:

- There is no earlier message from p_j that has not been delivered yet.

Causal Delivery Rule, Part 1 $D[j] == TS(m)[j] - 1$

- $\forall k \neq j$, let m' be the last message from p_k delivered by p_0 ($D[k] = TS(m')[k]$); we must be sure that no message m'' from p_k exists such that: $send_k(m') \rightarrow send_k(m'') \rightarrow send_j(m)$

How can we express this condition?

Implementing Causal Delivery with Vector Clocks

Two conditions to be verified:

- There is no earlier message from p_j that has not been delivered yet.

Causal Delivery Rule, Part 1 $D[j] == TS(m)[j] - 1$

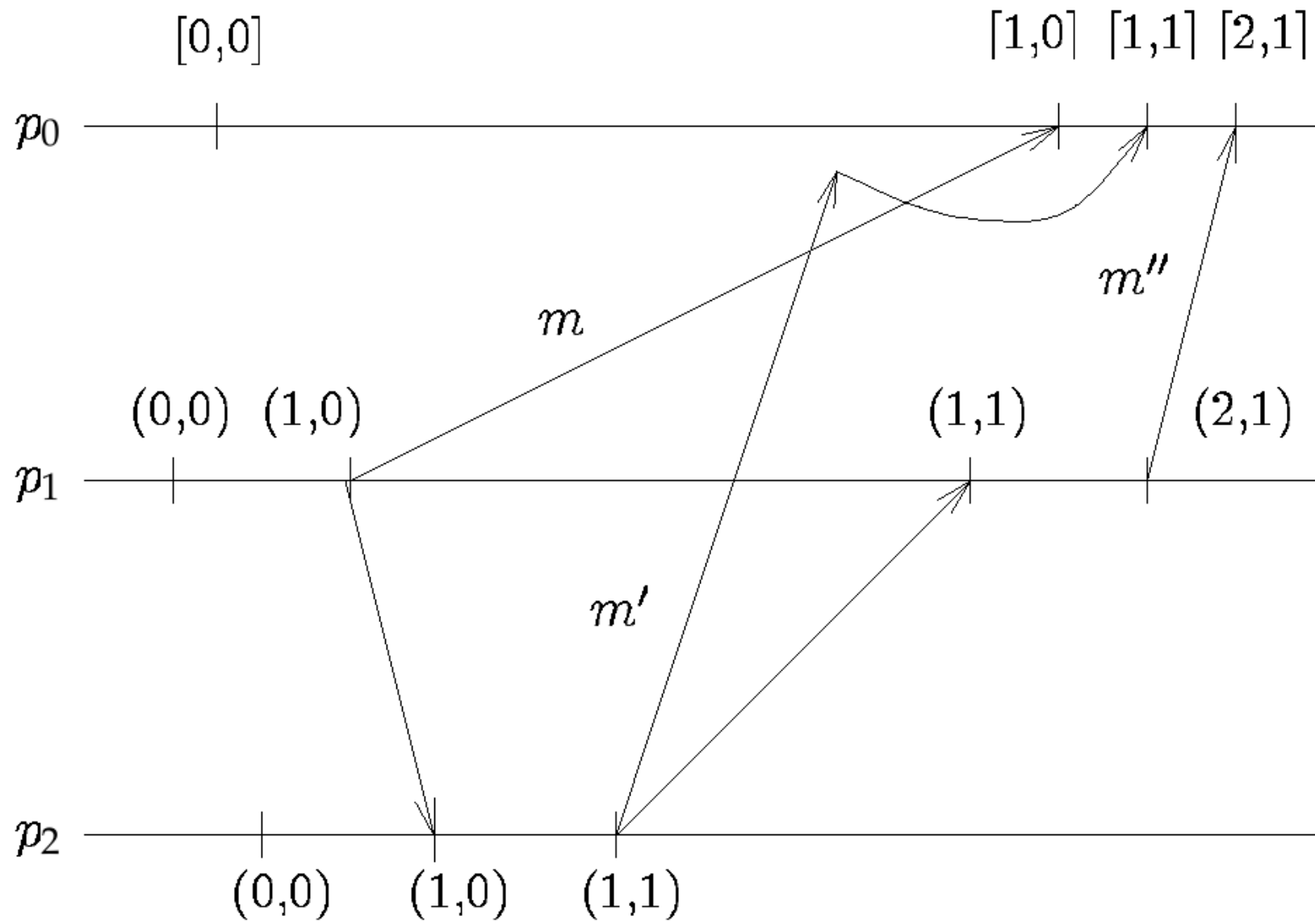
- $\forall k \neq j$, let m' be the last message from p_k delivered by p_0 ($D[k] = TS(m')[k]$); we must be sure that no message m'' from p_k exists such that: $send_k(m') \rightarrow send_k(m'') \rightarrow send_j(m)$

Causal Delivery Rule, Part 2: $\forall k \neq j : D[k] \geq TS(m)[k]$

It follows from Weak Gap Detection

Note: both parts must hold!

Example



Final Comments

- We have seen how to implement Causal Delivery “many-to-one”
- The same rules apply if we implement a mechanism for implementing “one-to-many” (reliable broadcast)
- Discussion about scalability

Snapshot Protocol

Problem

- **Goal:** To build the global state after an explicit request of the monitor.
- **How:** By taking “pictures” (snapshot) of the local state when instructed
- **Challenge:** To build a consistent global state.

Applications

- Failure recovery: a global state (**checkpoint**) is periodically saved; recovery from a failure is done by restoring the system to the last saved global state.
- Distributed garbage collection
- Monitoring distributed events (e.g., industrial process control)

Solution: Chandy and Lamport Snapshot Algorithm

- This particular protocol enables to reason about “channel states”
- GPE can be delayed with respect to passive monitoring
- Assumption: processes communicate through FIFO channels

Snapshot Protocol

Channel State

- For each channel between p_i and p_j
 - $x_{i,j}$ contains those messages that p_i has sent but p_j has not received yet.
 - $x_{j,i}$ contains those messages that p_j has sent but p_i has not received yet.
- Note: channel state can be obtained by storing appropriate information in the local state, but it is complicated.

Recorded information

Each process will record its local state σ_i and the content of its *incoming* channels $x_{j,i}$.

Snapshot Protocol

Technique: as before, we start with a synchronous protocol and we later relax our assumptions

- Access to a real time clock RC ;
- Message delays are bounded by some known value δ ;
- Relative process speeds are bounded;
- Message m is tagged with a timestamp $TS(m) = RC(e)$, where $e = send(m)$.

Snapshot Protocol, v.1

1. p_0 chooses a time t_s far enough in the future that a message containing t_s sent by p_0 will be received by all other processes before t_s ;

$$t_s = \text{now}() + \delta$$

2. p_0 sends a message “*take a snapshot at t_s* ” to all processes in Π ;
3. When $RC_i = t_s$:
 - (a) p_i records its local state;
 - (b) sends an empty message to all processes in Π ;
 - (c) starts to record all messages received over each incoming channel.
4. When p_i receives a message m from j such that $TS(m) \geq t_s$, it stops recording messages for incoming channel j
5. Each p_i then sends its recorded local state and the channel states to p_0 .

Snapshot Protocol, v.1

Liveness The empty messages at 3(b) guarantee liveness.

Safety This protocol constructs a consistent global state; actually, this global state did in fact occur. Formally:

Let C_s be the cut associated to the constructed global state;

$$(e \in C_s) \wedge (e' \rightarrow e) \Rightarrow$$

$$(e \in C_s) \wedge (RC(e') < RC(e)) \Rightarrow \text{C.C. : } e' \rightarrow e \Rightarrow RC(e') < RC(e)$$

$$(e' \in C_s) \quad C_s \text{ Def. : } e \in C_s \Leftrightarrow RC(e) < t_s$$

Can we use logical clocks instead of a real time clock?

Snapshot Protocol, v.2

Substituting real-time clocks with logical clocks requires some adjustments.

- The construct “ **when** $LC = t_s$ **do** S ” does not make sense with LC s.
 - LC s are not continuous (e.g., they can jump from $t_s - 1$ to $t_s + 1$)
 - When $LC = t_s$, the event that caused the clock update is already done.
- Solution: before p_i executes an event e :
 - If e is an internal or send event, and $LC = t_s - 2$, then p_i executes e and then starts executing S .
 - If $e = receive(m) \wedge TS(m) \geq t$ and $LC < t_s - 1$, then p_i :
 - * puts $LC = t_s - 1$;
 - * executes S ;
 - * executes e .

Snapshot Protocol, v.2

Substituting real-time clocks with logical clocks requires some adjustments.

- We assume that we can find a logical time t_s large enough that a message containing it sent by p_0 will be received by all other processes before t_s ;
- Impossible in an asynchronous dist. sys.
- We will relax this assumption later.

Snapshot Protocol, v.2

1. p_0 chooses a logical time t_s large enough
2. p_0 sends a message “*take a snapshot at t_s* ” to all processes in Π
3. p_0 sets its logical clock to t_s ;
4. When $LC_i = t_s$:
 - (a) p_i records its local state;
 - (b) sends an empty message to all processes in Π ;
 - (c) starts to record all messages received over each incoming channel.
5. When p_i receives a message m from j such that $TS(m) \geq t_s$, it stops recording messages for incoming channel j
6. Each p_i then sends its recorded local state and the channel states to p_0 .

Snapshot Protocol, v.3

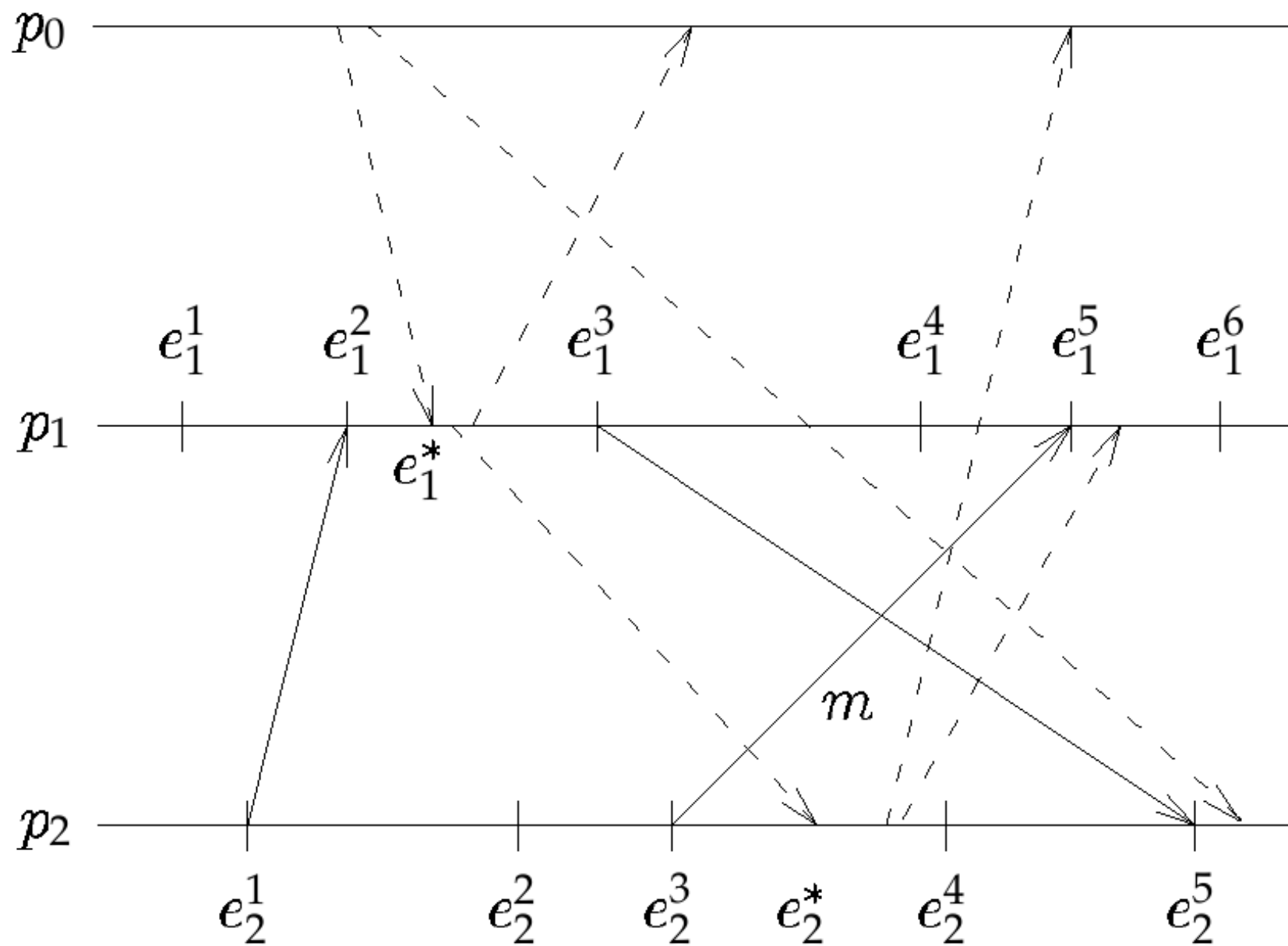
We now remove the need for t_s .

- A process may receive an empty message from a node before the “take snapshot at t_s ” is actually received
- In other words, it may be already aware of the snapshot protocol.
- We remove the “at t_s ” and we use “take snapshot” messages instead of empty ones.
- We can now remove logical clocks completely, as messages are not timestamped any more.

Snapshot Protocol, v.3

1. p_0 sends a message “*take snapshot*” to itself ;
2. the **first time** p_i receives a “*take snapshot*” message from a process p_j
 - p_i records its local state σ_i ;
 - sends “*take snapshot*” to all processes in Π ;
 - $x_{k,i} = \emptyset \quad \forall k \neq i$;
3. when p_i receives m different from “*take snapshot*” from $p_k, k \neq j$
 - $x_{k,i} := x_{k,i} \cup \{m\}$
4. when p_i receives a “*take snapshot*” from $p_k, k \neq j$ beyond the **first time**:
 - p_i stops recording messages in $x_{k,i}$;
5. when p_i has received a “*take snapshot*” from all processes
 - p_i then sends its recorded local state and the channel states to p_0 .

Example



Proof of correctness

A global state built using the snapshot algorithm is consistent.

Stable Predicates

Problem

- Let Σ be a global state built by one of the methods presented
- It represents a state of the past, that may have no bearing to the present
- Does it make sense to evaluate predicate Φ on it?

A special case: stable predicates

Many systems properties have the characteristics that once they become true, they remain true.

- Deadlock
- Garbage collection
- Termination

A distributed computation may have many runs

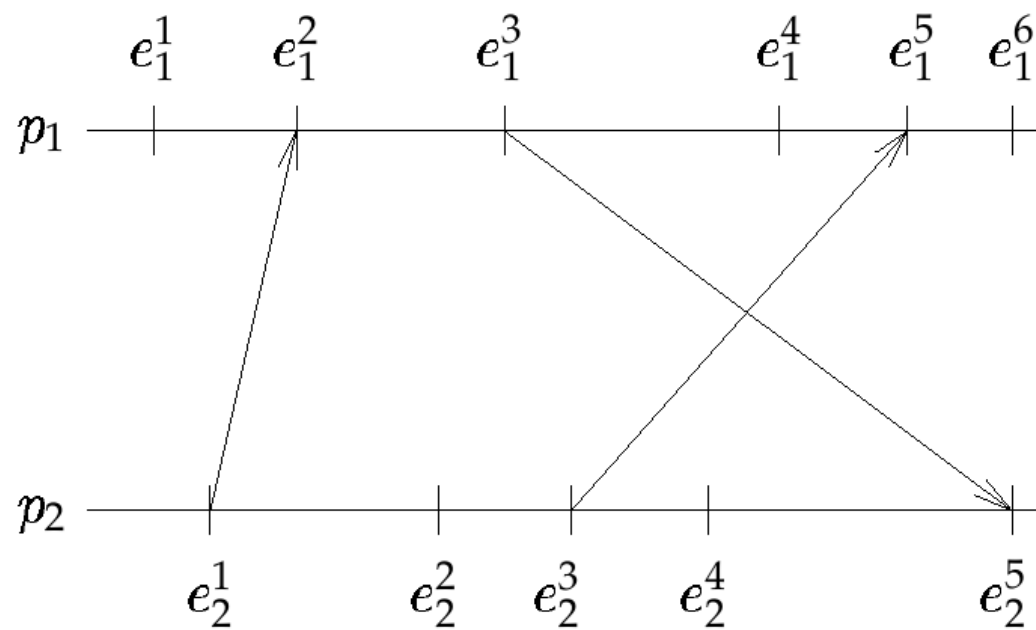
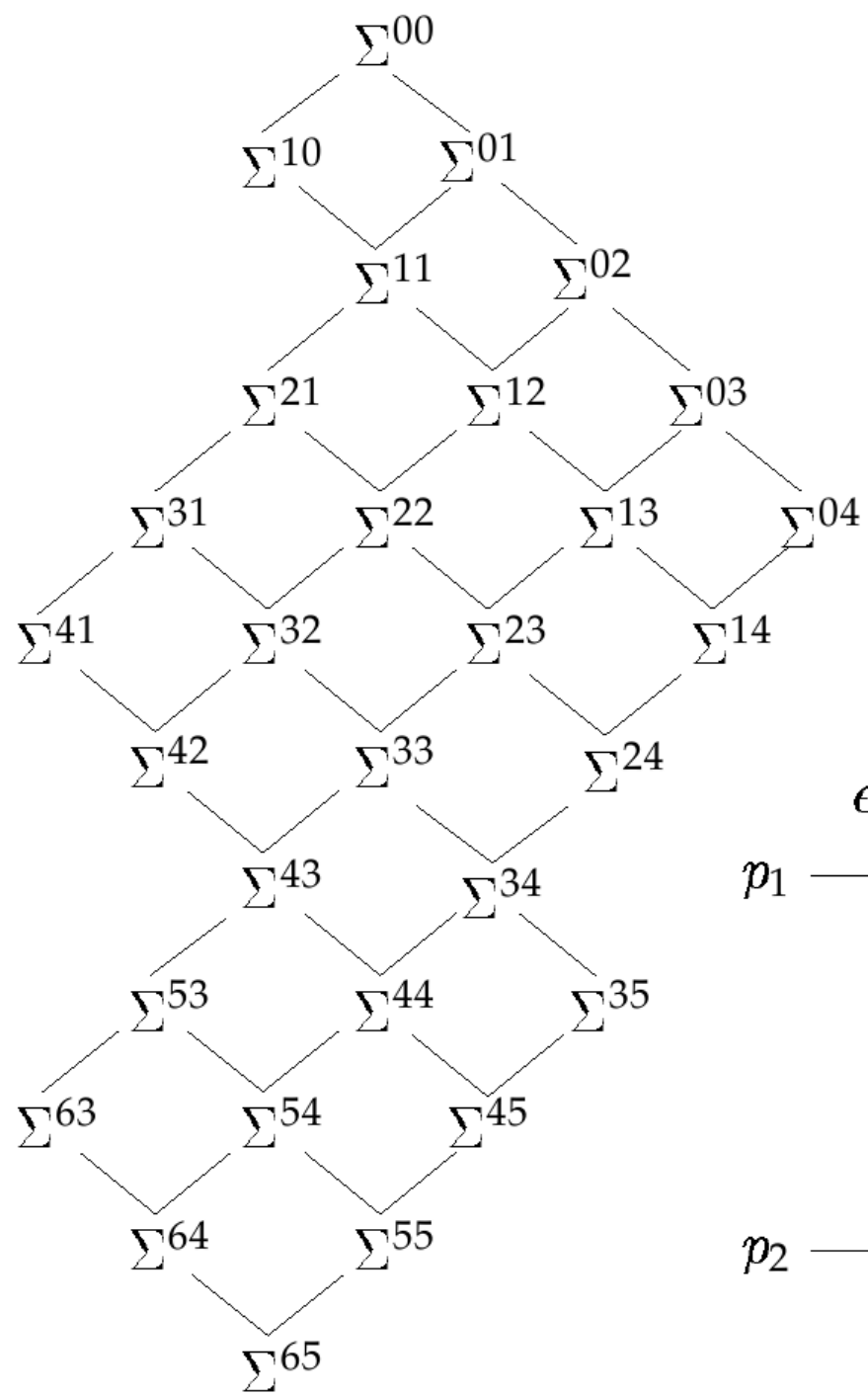
Leads-to relation

- A consistent run $R = e^1 e^2 \dots$ results in a sequence of consistent global states $\Sigma^0 \Sigma^1 \Sigma^2 \dots$, where Σ^0 denotes the initial global state.
- We say that a global state Σ **leads to** a global state Σ' , denoted $\Sigma \rightsquigarrow_R \Sigma'$ in a consistent run R if:
 - R results in a sequence of global states $\Sigma^0 \Sigma^1 \Sigma^2 \dots$;
 - $\Sigma = \Sigma^i, \Sigma' = \Sigma^j, i < j$.
- We write $\Sigma \rightsquigarrow \Sigma'$ if there is a run R such that $\Sigma \rightsquigarrow_R \Sigma'$.

A distributed computation may have many runs

Lattice

- The set of all consistent global states of a computation along with the leads-to relation defines a **lattice**;
- n orthogonal axis, one per process;
- $\Sigma^{k_1 \dots k_n}$ shorthand for the global state $(\sigma_1^{k_1}, \dots, \sigma_n^{k_n})$;
- The **level** of $\Sigma^{k_1 \dots k_n}$ is equal to $k_1 + \dots + k_n$.
- A path in the lattice is a sequence of global states of increasing levels that corresponds to a consistent run.



Stable Predicates

Consider a global state construction protocol:

- Let Σ^a be the global state in which the protocol is initiated;
- Let Σ^f be the global state in which the protocol terminates;
- Let Σ^s be the global state constructed by the protocol

Since $\Sigma^a \rightsquigarrow \Sigma^s \rightsquigarrow \Sigma^f$, if Φ is stable, then:

$$\Phi(\Sigma^s) = \mathbf{true} \quad \Rightarrow \quad \Phi(\Sigma^f) = \mathbf{true}$$

$$\Phi(\Sigma^s) = \mathbf{false} \quad \Rightarrow \quad \Phi(\Sigma^a) = \mathbf{false}$$

Deadlock Detection

Code

- Server code
- Server code, modified for snapshot protocol
- Monitor code for snapshot protocol
- Server code, modified for passive protocol
- Monitor code for passive protocol

Notes

- No need to store also channel state in this case

process p(i)

Queue pending := **new** Queue()

boolean working := **false**

while true do

while working **or** pending.size() = 0 **do**

receive <m, j>

case m.type **of**

 REQUEST:

 pending.enqueue(<m, j>)

 RESPONSE:

 <m', j'> := nextState(m, j)

 working := (m'.type = REQUEST)

send m' **to** p(j')

while not working **and** (pending.size() > 0) **do**

 <m, j> := pending.dequeue()

 <m', j'> := nextState(m, j)

 working := (m'.type = REQUEST)

send m' **to** p(j')

Deadlock Detection through Snapshot

Approach

- All channels are based on FIFO delivery
- Add code to deal with Snapshot messages

Pros and Cons

- Generates overhead only when deadlock is suspected
- Introduces a latency between deadlock and detection

process p(i)

Queue pending := **new** Queue()

boolean working := **false**

boolean[1..n] blocking := **false**, ..., **false**

while true do

while working **or** pending.size() = 0 **do**

receive <m, j>

case m.type **of**

 REQUEST:

 blocking[j] := **true**

 pending.enqueue(<m, j>)

 RESPONSE:

 <m', j'> := nextState(m, j)

 working := (m'.type = REQUEST)

send m' **to** p(j')

if (m'.type = RESPONSE) **then**

 blocking[j'] := **false**

SNAPSHOT :

if $s = 0$ **then**

send <SNAPSHOT, blocking> **to** $p(0)$

send <SNAPSHOT> **to** $p(1), \dots, p(i-1), p(i+1), \dots, p(n)$

$s := (s + 1) \bmod n$

while not working **and** pending.size() > 0 **do**

$\langle m, j \rangle :=$ pending.dequeue()

$\langle m', j' \rangle :=$ NextState(m, j)

working := ($m'.type = request$)

send m' **to** $p(j')$

if ($m'.type = RESPONSE$) **then**

blocking[j'] := **false**

```
process p(0)
  boolean [1..n, 1..n] wfg
  while true do
    // wait until deadlock is suspected;
    send <SNAPSHOT> to p(1), ..., p(n);
    for k := 1 to n do
      receive <m, j>
      wfg[j] := m.data;
    if (cycle in wfg) then system is deadlocked
```

Deadlock Detection through Passive Monitoring

Approach

- Sends a message to p_0 for each relevant event
- Communication with p_0 based on causal delivery

Pros and Cons

- Simpler approach, but complexity is hidden by the causal delivery mechanism
- Latency limited to message delays
- Higher overhead

```
process p(i)
```

```
Queue pending := new Queue()
```

```
boolean working := false
```

```
while true do
```

```
  while working or pending.size() = 0 do
```

```
    receive <m, j>
```

```
    case m.type of
```

```
      REQUEST:
```

```
        send <REQUESTED, i, j> to p(0)
```

```
        pending.enqueue(<m, j>)
```

```
      RESPONSE:
```

```
        <m', j'> := nextState(m, j)
```

```
        working := (m'.type = REQUEST)
```

```
        send m' to p(j')
```

```
        if (m'.type = RESPONSE) then
```

```
          send <RESPONDED, i, j> to p(0)
```

```
while not working and (pending.size() > 0) do  
  <m, j> := pending.dequeue()  
  <m', j'> := NextState(m, j)  
  working := (m'.type = REQUEST)  
  send m' to p(j')  
  if (m'.type = RESPONSE) then  
    send <RESPONDED, i, j> to p(0)
```

```
process p(0)
  boolean [1..n, 1..n] wfg
  while true do
    receive <m, j>
      if (m.type = RESPONDED) then
        wfg[m.from, m.to] := false
      else
        wfg[m.from, m.to] := true
      if (cycle in wfg) then system is deadlocked
```

Non-stable Predicates

Problems of non-stable predicates

- The condition encoded by the predicate may not persist long enough for it to be true when the predicate is evaluated;
- If a predicate Φ is found to be true by the monitor, we do not know whether Φ *ever* held during the *actual* run.

Conclusions

- Evaluating a non-stable predicate over a single computation makes no sense
- The evaluation must be extended to the entire lattice of the computation.

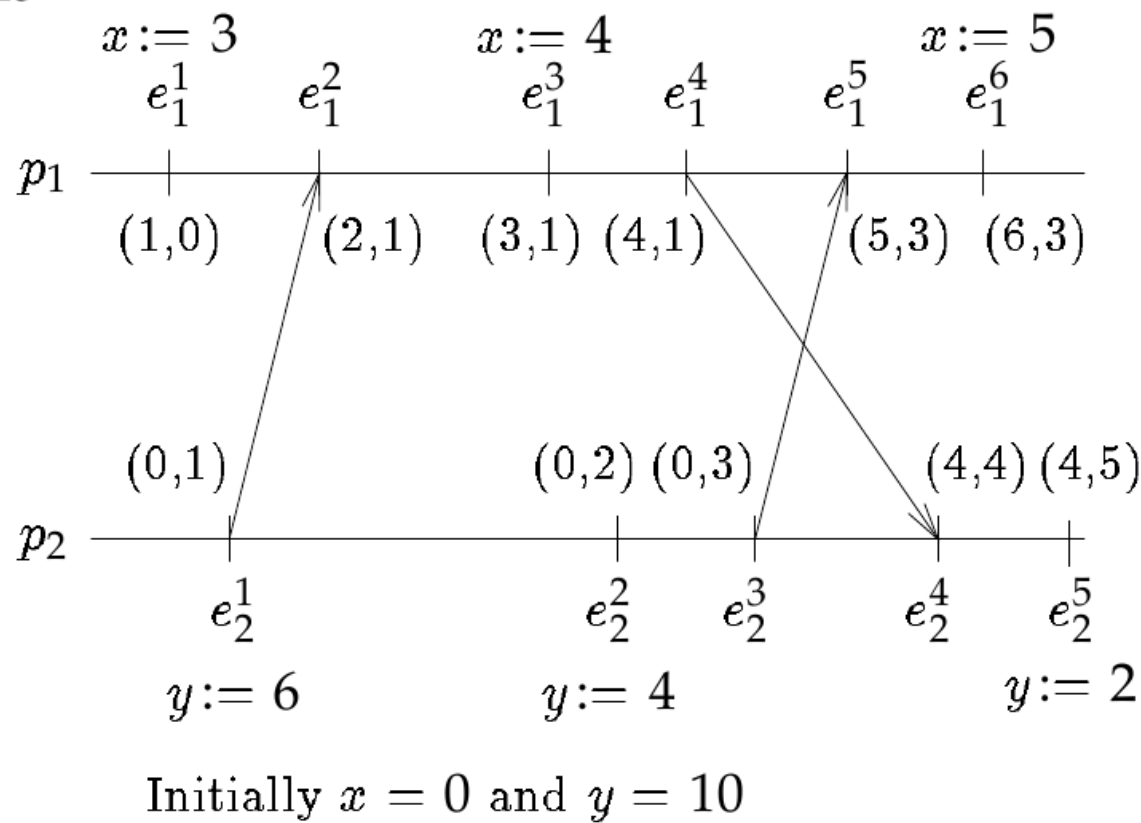
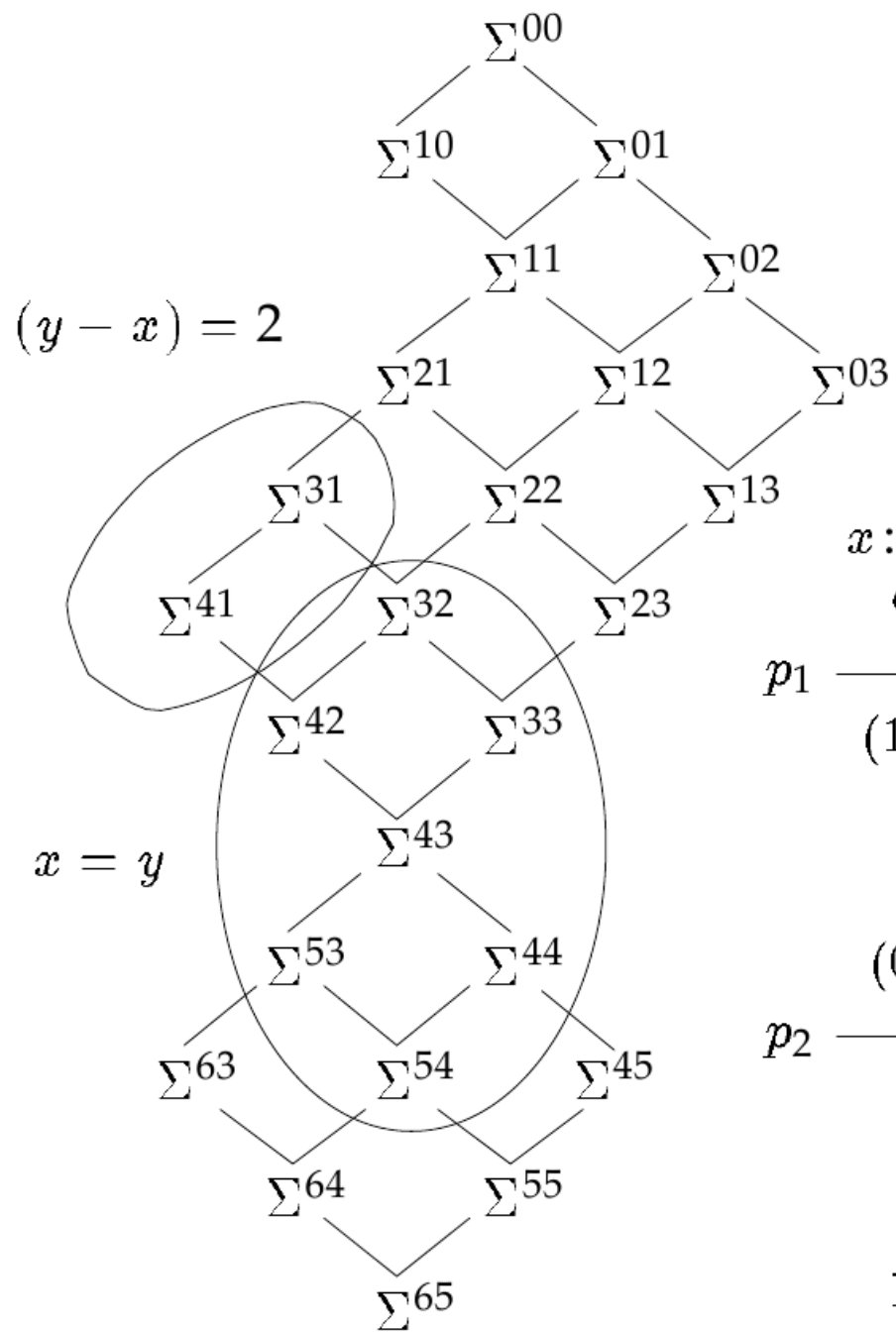
Predicates over entire computations

It is possible to evaluate a predicate over an entire computation using an observation obtained by a passive monitor.

- **Possibly(Φ)**: There exists a consistent observation O of the computation such that Φ holds in a global state of O .
- **Definitely(Φ)**: For every consistent observation O of the computation, there exists a global state of O in which Φ holds.

Examples **Possibly** $((y - x) = 2)$, **Definitely** $(x = y)$

Debugging If **Possibly** (Φ) is true, and Φ identifies some erroneous state of the computation, then there is a bug, even if it is not observed during an actual run.



Predicates over entire computations

Properties **Possibly** and **Definitely** are not duals:

$$\neg \mathbf{Possibly}(\Phi) \not\equiv \mathbf{Definitely}(\neg\Phi)$$

$$\neg \mathbf{Definitely}(\Phi) \not\equiv \mathbf{Possibly}(\neg\Phi)$$

Example $\mathbf{Possibly}(x \neq y)$, $\mathbf{Definitely}(x = y)$

Algorithms for detecting Possibly and Definitely

- We use the passive approach in which processes send notifications of events relevant to Φ to the monitor p_0 ;
- Events are tagged with vector clocks;
- The monitor collects all the events and builds the lattice of global states.

How?

- To detect **Possibly**(Φ): if there exists one global state in which Φ is true, then return **true**, otherwise **false**.
- To detect **Definitely**(Φ): mark nodes where Φ is true with a value 1, the other nodes with value 0. If the cost of the shortest path between the initial state and the final state is larger than 0, return **true**, otherwise **false**.

Algorithms for detecting Possibly and Definitely

Problems

- The number of states grows exponentially with the number of total events.
- Techniques can be used to reduce the number of events
 - Only those relevant to Φ
 - Forcing periodic synchronization
 - Reducing the complexity of predicates (conjunction of local predicates)