

Slides for Chapter 14: Time and Global States



From **Coulouris, Dollimore, Kindberg and Blair**
Distributed Systems:
Concepts and Design

Edition 5, © Addison-Wesley 2012

Introduction [14.1]

- We need to reason about time of events
- No perfect global clock
- Lots of work on clock synchronization, we are skipping (14.3)

Clocks, events, and process states [14.2]

- Refine the model in Chapter 2 to process interactions
- Consider DS a set P of N processes, p_i for $i=1, \dots, N$
- Process p_i has a state s_i that (usually) changes over time
- Process p_i takes a series of actions, from 3 choices
 - Message send
 - Message receive
 - Operation to transform its state
- **Event** \equiv occurrence of a single action that a process carries out as it executes
 - Totally ordered (locally) on a given host,
- $\text{History}(p_i) \equiv h_i \equiv \langle e_i^0, e_i^1, e_i^2, \dots \rangle$ #series of events
- **Note: skipping rest of 14.2 ... on clocks etc and also 14.3**

Logical time and logical clocks [14.4]

- (Going to teach through the VR01 slide set for most of this, then go through the examples here to reinforce)
- Also for vector clocks separate example slides

Figure 14.5

Events occurring at three processes

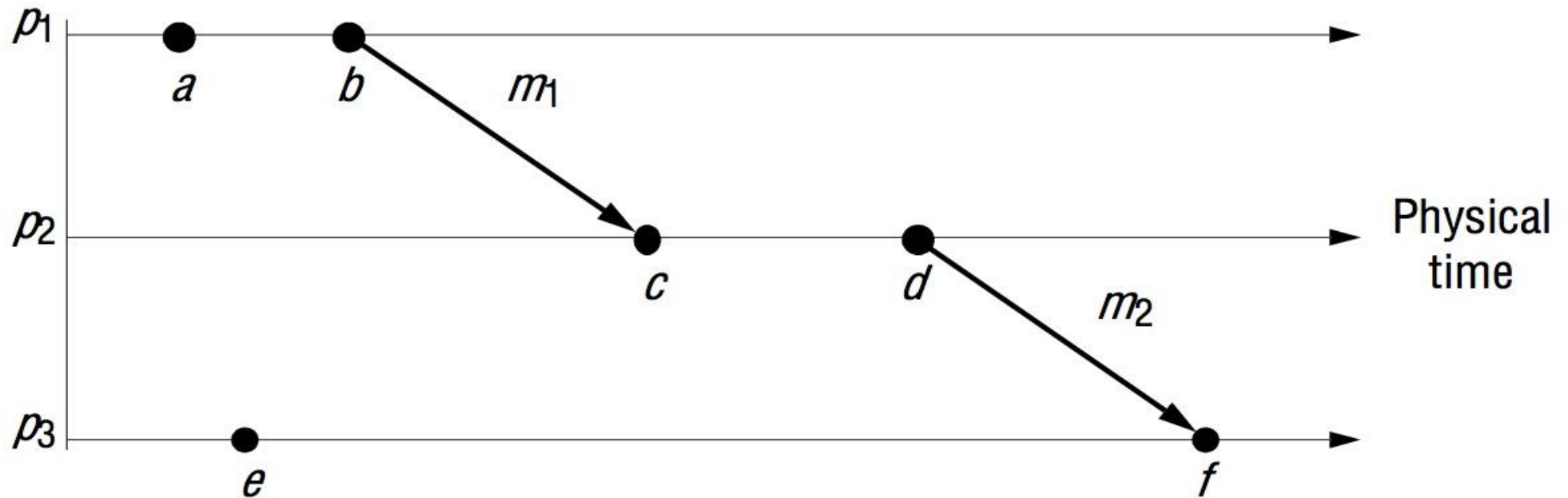
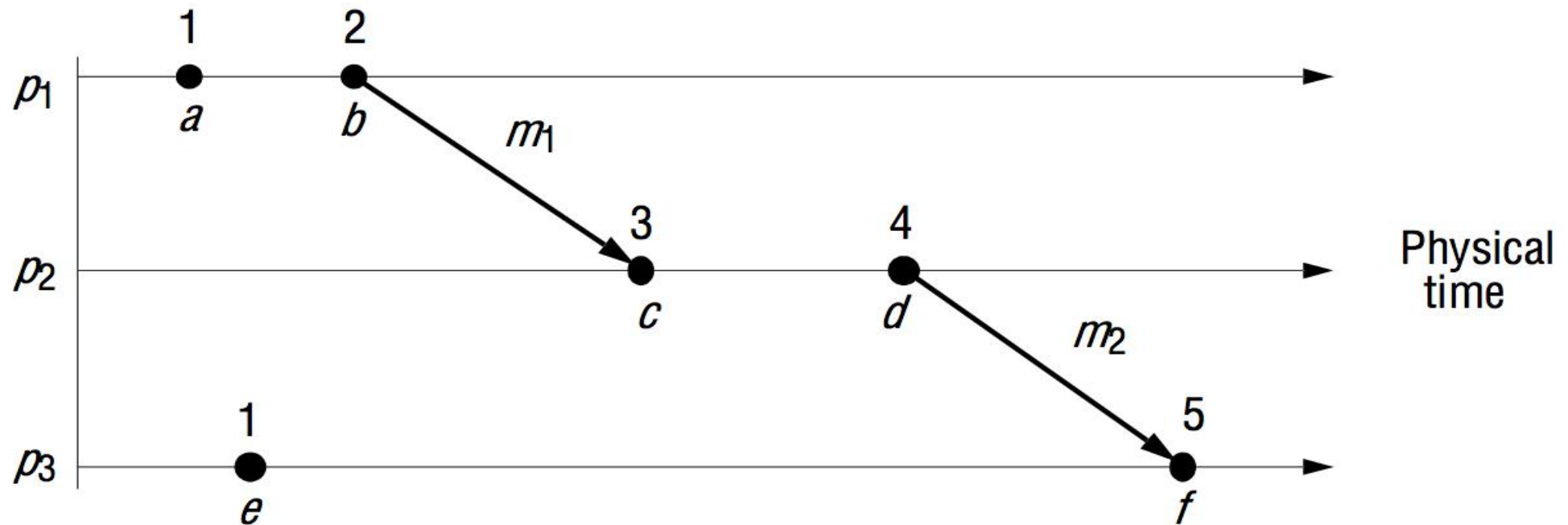


Figure 14.6

Lamport timestamps for the events shown in Figure 14.5

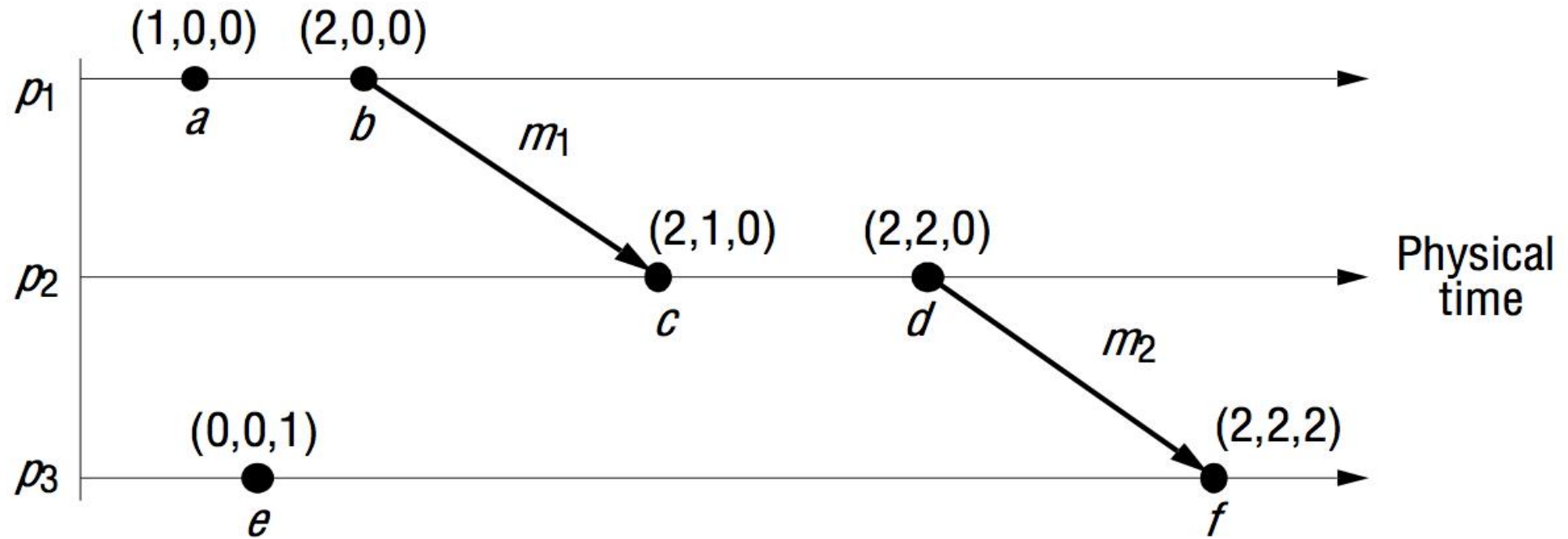


Vector clocks

- Limitation of Lamport clocks: if $L(e) < L(e')$ we can't conclude that $e \rightarrow e'$
- Solution: make the LC scalar a vector
- $V_i[j] \equiv$ number of events that p_j has timestamped
- $V_i[j]$ (for $i \neq j$) \equiv #events at p_j that may have affected p_i and that p_i knows about.
- (Now see 2 slide sets: vector clocks example and also from the Birman book)

Figure 14.7

Vector timestamps for the events shown in Figure 14.5

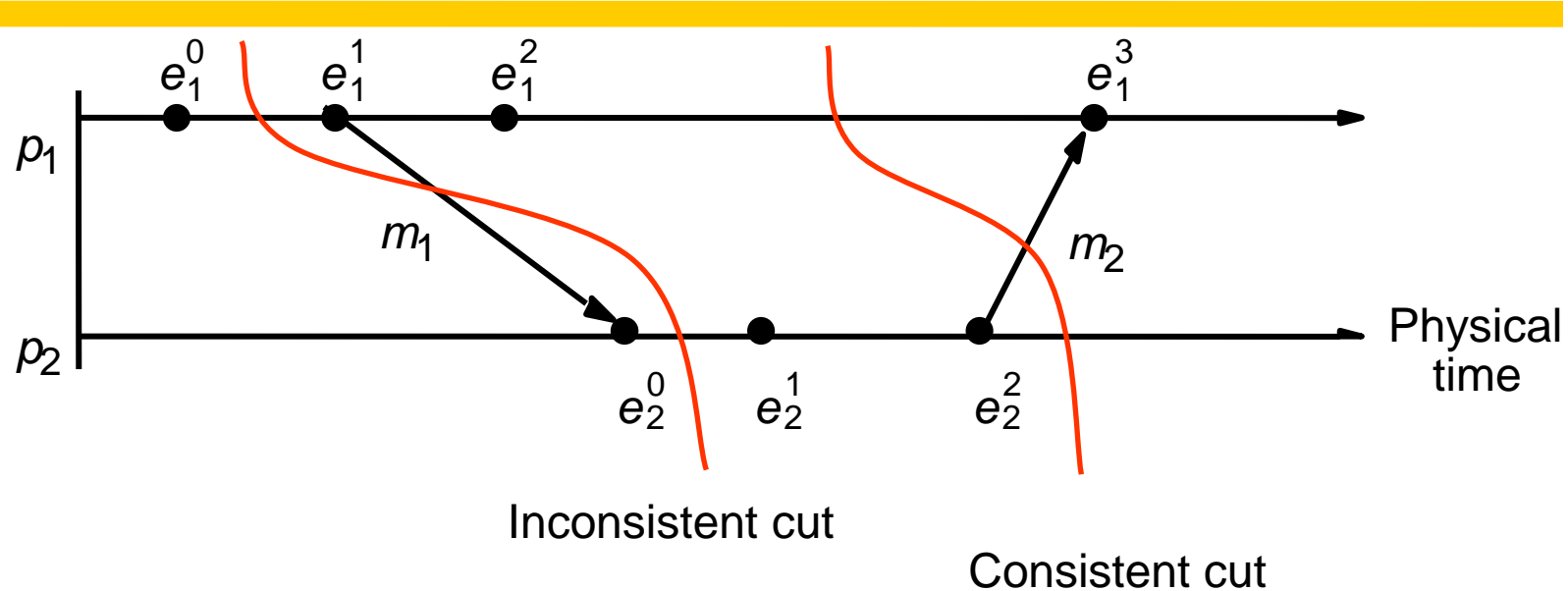


Global states [14.5]

- (See the VR01 slides for this)

Figure 14.9

Cuts



- **Cut** of a system subset of its global history: union of prefixes of process histories
 - **Frontier** of cut: last event in each process's prefix
- Cut C **consistent** if, for every event it contains, all events that "happened before" that event are also contained
 - i.e., for all events e in C , $f \rightarrow e \rightarrow f$ is in C

Consistent cuts

- Recall system goes through $S_0 \rightarrow S_1 \rightarrow S_2 \dots$
 - One different event at one process in $S_i \rightarrow S_{i+1}$
 - Global state then union of process states after a cut
- **Run**: a total ordering of all events in a global history that's consistent with each local history's ordering
 - Not all runs pass through consistent global states
- **Linearization** (AKA **consistent run**): ordering of events in global history consistent with happened-before relationship on the history
 - All linearizations pass only through consistent global states
- **Reachability**: state S' is **reachable** from state S if there is a linearization that passes through S and then S' .

Global state predicates, stability, safety, and liveness [14.5.2]

- Evaluate a **global state predicate** to detect deadlock, etc
 - Function mapping from global states to {True, False}
 - **Stable property**: once predicate true, stays true (opp.: **transitory**)
 - i.e., true from all states reachable from the present state
- **Safety property** (e.g., α): nothing “bad” ever happens
 - E.g., never have deadlock
 - i.e., for all states reachable from initial state, α is False (never True)
- **Liveness property** (e.g., β): something good eventually happens
 - E.g., distributed algorithm eventually terminates
 - i.e., **Liveness w.r.t. β** : for any linearization L starting in state S_0 , β evaluates to True for some state S_L reachable from S_0 .

Snapshot algorithm

- By Chandy and Lamport [1985]: determine global states
- Goal: record a set of process AND channel states such that it is consistent (not strongly consistent)
- Assumptions
 - Neither channels nor processes fail
 - Channels are uni-directional and FIFO ordered
 - Graph of processes and channels strongly connected (path between any 2 processes)
 - Any process may initiate the snapshot at any time
 - Processes don't need to freeze/lock: continue normal operations

Snapshot algorithm (cont.)

- Main ideas

- Terms: **incoming channels** and **outgoing channels** for p_i
- Each process records its state, and for each incoming channel, set of messages sent to it
- For each channel, process records msgs that arrived after its last recorded state and before sender recorded state
 - I.e.,. Record state at different times but account for messages transmitted but not yet received (these are part of the channel stat)
- Use distinguished **marker messages**
 - Tell receiver to save state
 - Way to determine which messages go in channel state
 - To initiate the algorithm, process acts like it received a marker message

Figure 14.10

Chandy and Lamport's 'snapshot' algorithm

Marker receiving rule for process p_i

On p_i 's receipt of a *marker* message over channel c :

if (p_i has not yet recorded its state) it

records its process state now;

records the state of c as the empty set;

turns on recording of messages arriving over other incoming channels;

else

p_i records the state of c as the set of messages it has received over c
since it saved its state.

end if

Marker sending rule for process p_i

After p_i has recorded its state, for each outgoing channel c :

p_i sends one marker message over c

(before it sends any other message over c).

Example of snapshot algorithm

- Two processes, trade in widgets, over two unidirectional channels
- Process p1 sends orders for widgets to p2 with its payment (\$10/widget)
- Process p2 sends widget along other channel

Figure 14.11

Two processes and their initial states

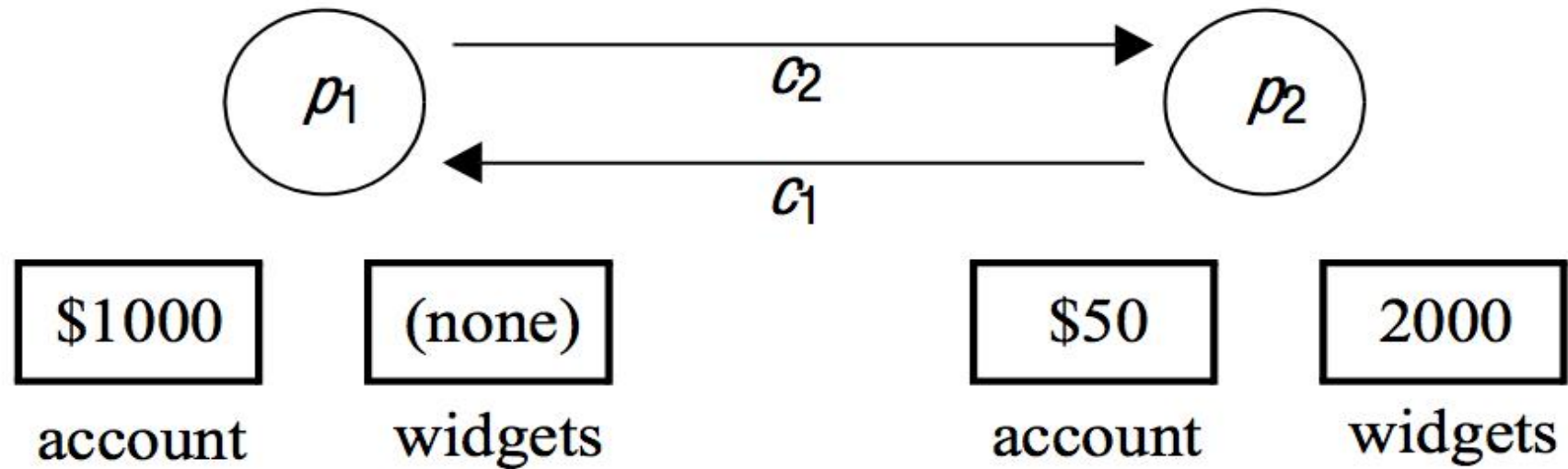
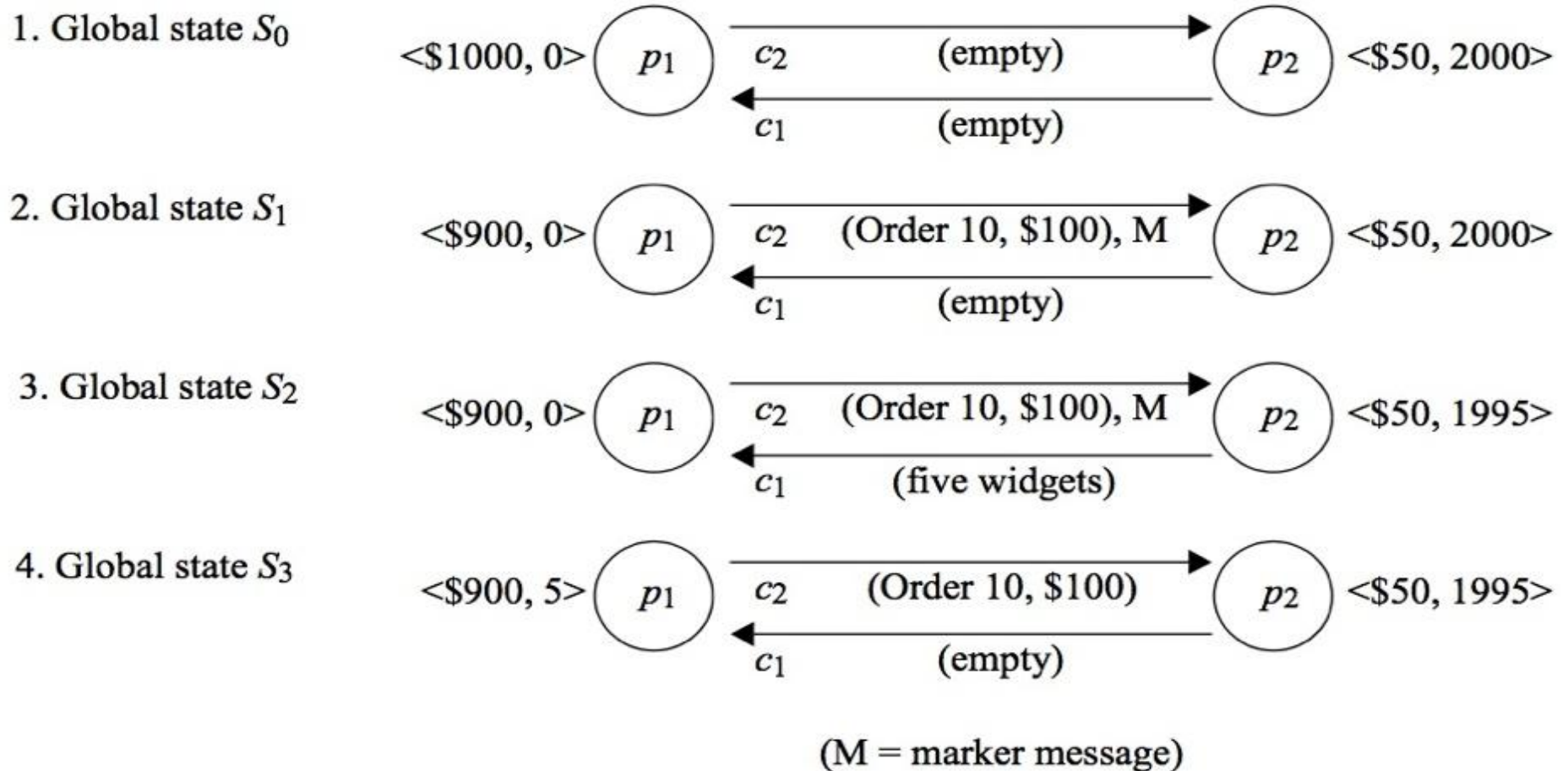


Figure 14.12

The execution of the processes in Figure 14.11



Note: (1) S_0 is when p_1 sends marker (3) p_1 had previously ordered five widgets; sent before M received by p_2 (5) After above, final recorded state includes five widgets in c_1 , yet system did not go through this state

Distributed debugging [14.6]

- Problem: recording system's global state to make useful statements about whether a transitory state occurred in an actual execution
 - Capture trace info and do *post hoc* analysis
- Chandy and Lamport's [1985] snapshot algorithm earlier used to collect states
 - Send to monitor process (considered outside the system)
 - Algorithm by Marzullo and Neiger [1991]

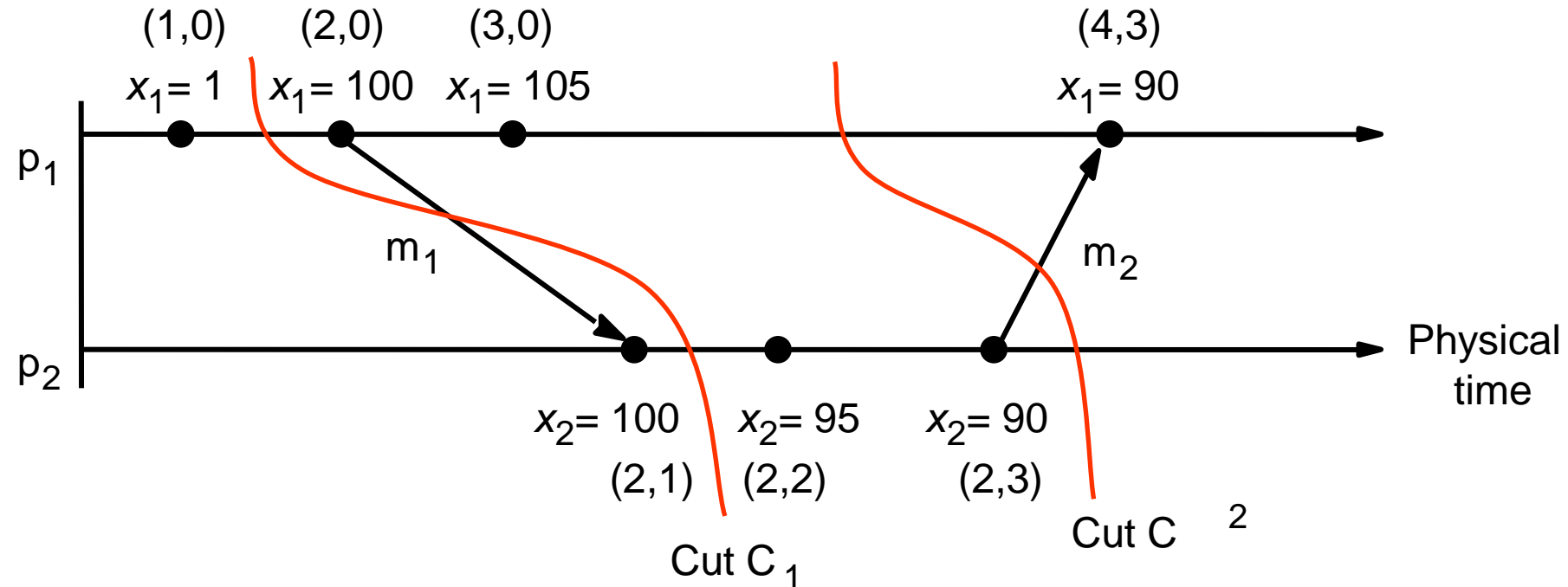
Distributed debugging (cont.)

- Goal: determine cases where global state predicate ϕ
 - Was **definitely True** at some point in the execution
 - Was **possibly True** at some point in the execution
- “Definitely” applies to actual execution, not run extrapolated from it
 - Can consider all linearizations H of the observed events
 - **Possibly ϕ** : exists a consistent global state S through which a linearization of H passes such that $\phi(S)$ is True
 - **Definitely ϕ** : for all linearizations L of H , exists a consistent global state S through which L passes such that $\phi(S)$ is True

Collecting the state [14.6.1]

- Procs p_i send in initial state, then periodically later ones
 - Does not interfere with execution, only delays a bit
 - Only need to send updates when change in variable used in ϕ
 - Monitor proc records state msgs from each p_i in queue Q_i

Figure 14.14 Vector timestamps and variable values for the execution of Figure 14.9



Example : $|x_i - x_j| \leq \delta$ for all i, j in $[1, N]$
E.g. $\delta = 50$ & send only “large adjustments” next slides..
Inconsistent cut C_1 show violation that never happened... but C_2 did

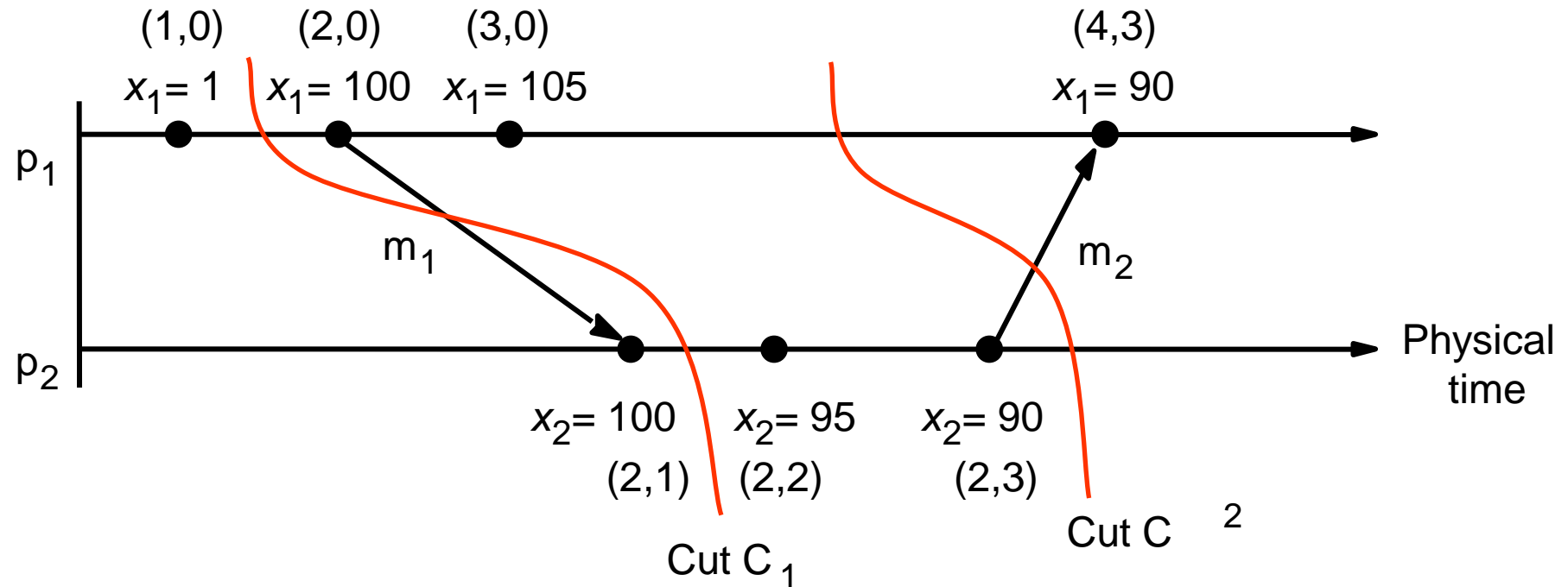
Observing consistent global states [14.6.2]

- Recall a cut C **consistent** if, for every event it contains, all events that “happened before” that event are also contained
 - i.e., for all events e in C , $f \rightarrow e \rightarrow f$ is in C
- Fig 14.14 & only send when adjustments “large enough”
 - Upon receipt, process updates its value to that of sender
- To know of cut is consistent, processes also send vector clocks with (changed) state

Observing consistent global states (cont.)

- Let
 - $S = \{s_1, s_2, \dots, s_N\}$ be a global state at monitor, from the state msgs
 - $V(s_i)$ vector timestamp of state s_i received from process p_i
- Then S is a consistent global state iff
 - $V(s_i)[i] \geq V(s_j)[i]$ for i, j in $[1, N]$**
 - I.e., # of p_i 's events known at p_j when it sent s_j is no more than the number of events that had occurred at p_i when it sent s_i .
 - I.e., if one proc's state depends on another (by happened-to), then global state also encompasses state upon which it depends
 - How to represent? Lattices (2 slides away)
 - Condition depicted next...

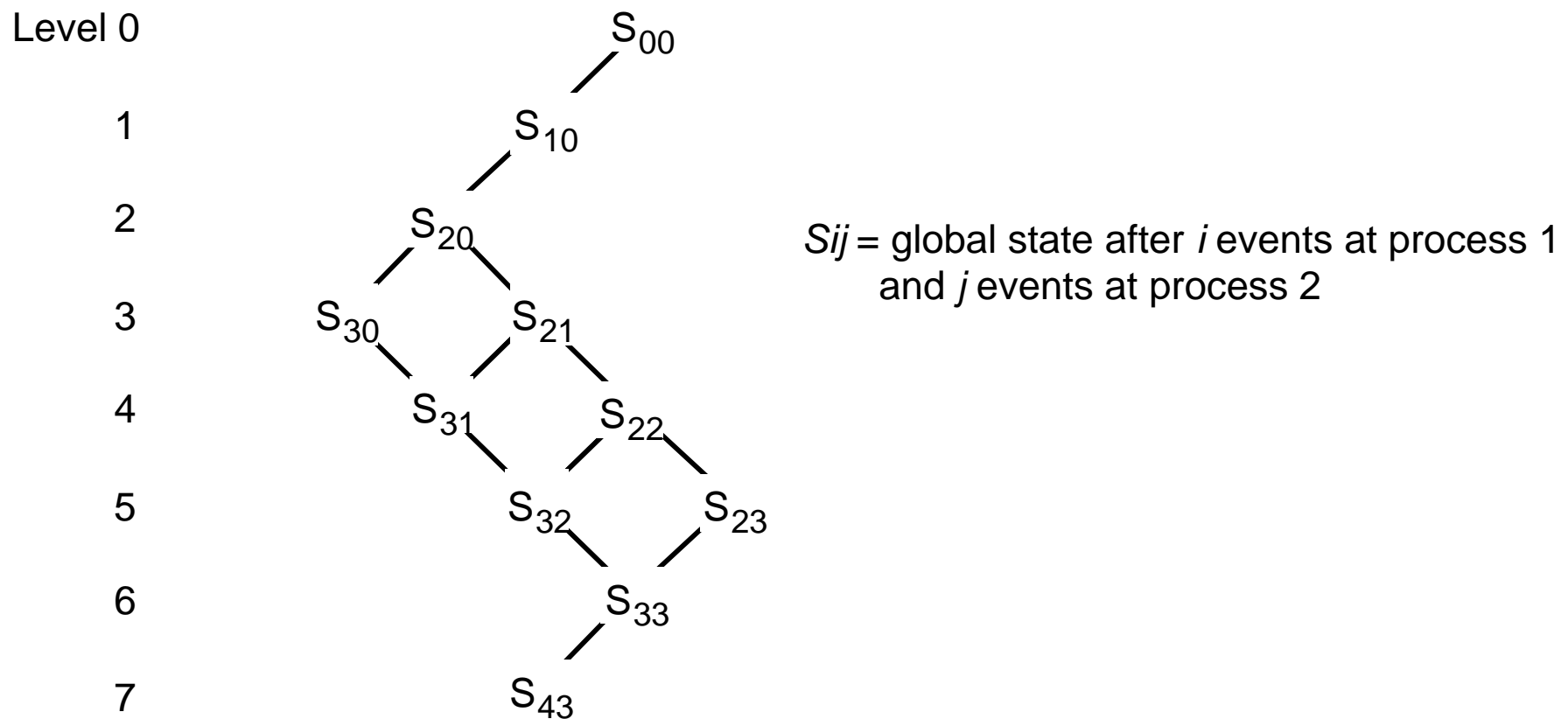
Figure 14.14 REDUX Vector timestamps and variable values for the execution of Figure 14.9



- Consistent cut iff **$V(s_i)[i] \geq V(s_j)[i]$ for i, j in $[1, N]$**
 - I.e., # of p_i 's events known at p_j when it sent s_j is no more than the number of events that had occurred at p_i when it sent s_i .
 - I.e., if one proc's state depends on another (by happened-to), then global state also encompasses state upon which it depends

Figure 14.15

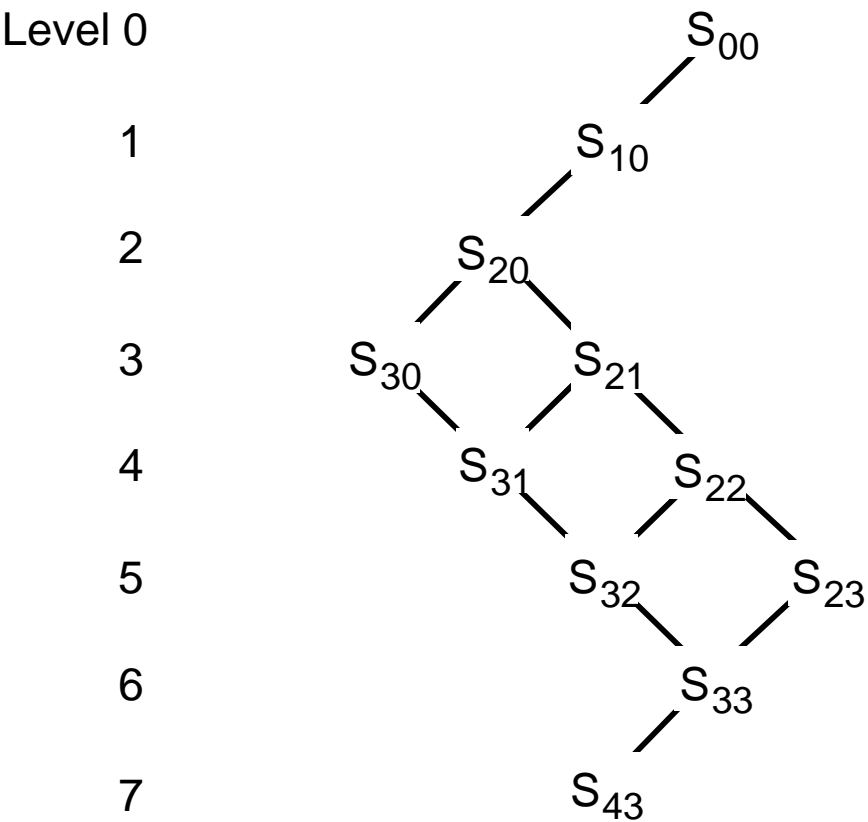
The lattice of global states for the execution of Figure 14.14



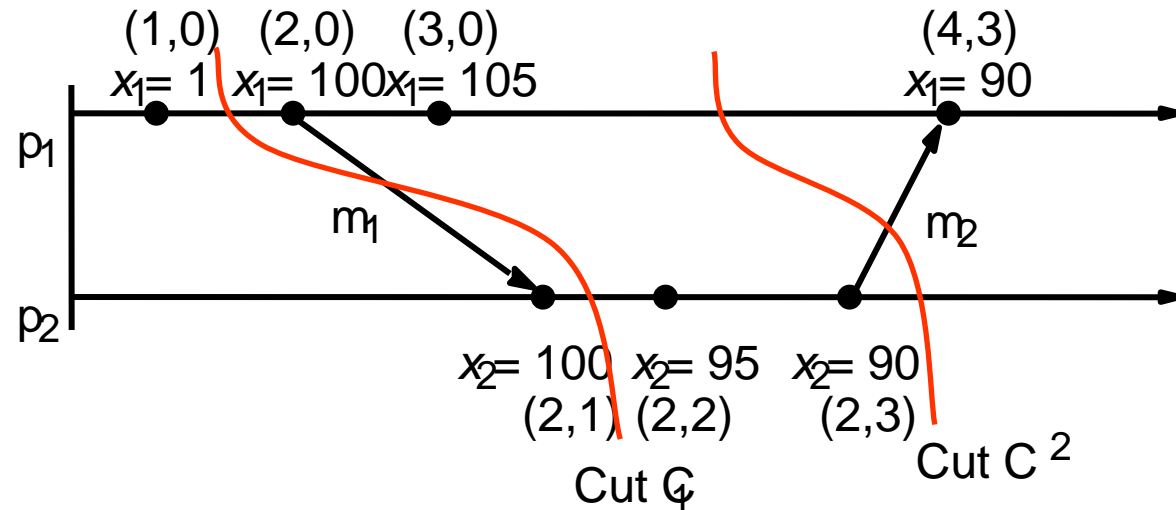
- **Lattice**: a partially ordered set represented graphically (loose defn)
- Captures reachability between consistent global states
- A linearization traverses from top to bottom, one level down only.
- Eg. Above is all consistent global states in the history

Figure 14.15 Redux

The lattice of global states for the execution of Figure 14.14



S_{ij} = global state after i events at process 1 and j events at process 2



Evaluating with the lattice

- Lattice shows us all linearizations corresponding to a history
- Evaluating possibly ϕ
 - Start at initial stage & step through all consistent states
 - Evaluate ϕ at each stage, stop when it evaluates to True
- Evaluating definitely ϕ
 - Try to find a set of states through which all linearizations must pass
 - Then check if the set's states all evaluate ϕ to True; done if find
 - E.g., $\phi(S_{30})$ and $\phi(S_{21})$ both true, and one or other must be passed through for all executions

Figure 14.16: Algorithms to eval. *possibly* ϕ and *definitely* ϕ

NOTE: infinite depth

1. Evaluating *possibly* ϕ for global history H of N processes

$L := 0$;

$States := \{ (s_1^0, s_2^0, \dots, s_N^0) \}$;

while ($\phi(S) = False$ for all $S \in States$)

$L := L + 1$;

$Reachable := \{ S' : S' \text{ reachable in } H \text{ from some } S \in States \wedge \text{level}(S') = L \}$;

$States := Reachable$

end while

output "*possibly* ϕ ";

S' set where one event diff. from S

Reachable iff $V(s_j)[j] \geq V(s'_j)[j]$ for $i \neq j$ in $[1, N]$

Can find all states: traverse state queue messages Q_{i1}

2. Evaluating *definitely* ϕ for global history H of N processes

$L := 0$;

if ($\phi(s_1^0, s_2^0, \dots, s_N^0)$) *then* $States := \{ \}$ *else* $States := \{ (s_1^0, s_2^0, \dots, s_N^0) \}$;

while ($States \neq \{ \}$)

$L := L + 1$;

$Reachable := \{ S' : S' \text{ reachable in } H \text{ from some } S \in States \wedge \text{level}(S') = L \}$;

$States := \{ S \in Reachable : \phi(S) = False \}$

end while

output "*definitely* ϕ ";

Figure 14.17

Evaluating *definitely* ϕ

Level 0

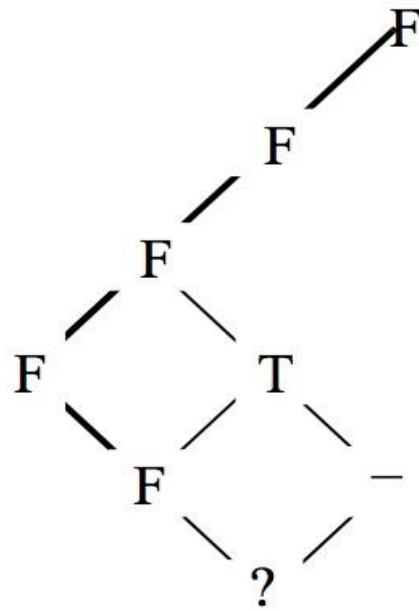
1

2

3

4

5



$F = (\phi(S) = \text{False}); T = (\phi(S) = \text{True})$

Only traverse states eval F

E.g., Level 3 only one (bold lines)

E.g., Level 4 only one, right one not reachable from F

If $\phi(?)$ is True then definitely ϕ holds