고려대학교
KOREA UNIVERSITY

KU-The Future

Chapter 14 Time and Global States

신입생 세미나

2021.01.14

석·박사통합과정 김명현

Table of Contents

Section 1

“Introduction”

1. Introduction

Time

Need to measure accurately : for synchronizing clocks with an external source of time

Problems : algorithms that depend upon clock synchronization have several problems

- maintaining the **consistency** of distributed data
- checking the **authenticity** of a request sent to a server
- eliminating the processing of duplicate updates

Section 2

“ Clocks, events and process states ”

2. Clocks, events and process states

Definitions

In distributed system,

- **Process** $p_i (i = 1,2,..,N)$: executing on a single processor, not sharing memory
- p_i's **State** $s_i (i = 1,2,...,N)$: including values of local variables (OS environment, files)
- p_i's **Action** : message send/receive operation, or an operation that transforms s_i
- **Event** : occurrence of a single action

  (Sequence of Events : $relation \rightarrow_i$)
- **History** of Process : $history(p_i) = h_i = < e_i^0, e_i^1, \ldots >$

2. Clocks, events and process states

Clocks

Physical Clock : an electronic device that count oscillations occurring in a crystal

- $H_i(t)$: the node's **hardware** clock value (OS reads, scales and adds an offset)
- $C_i(t) = \alpha H_i(t) + \beta$: **software** clock that approximately measures real, physical time of p_i

  ($C_i(t)$ is not completely accurate)

Clock Resolution : the period between updates of the clock value

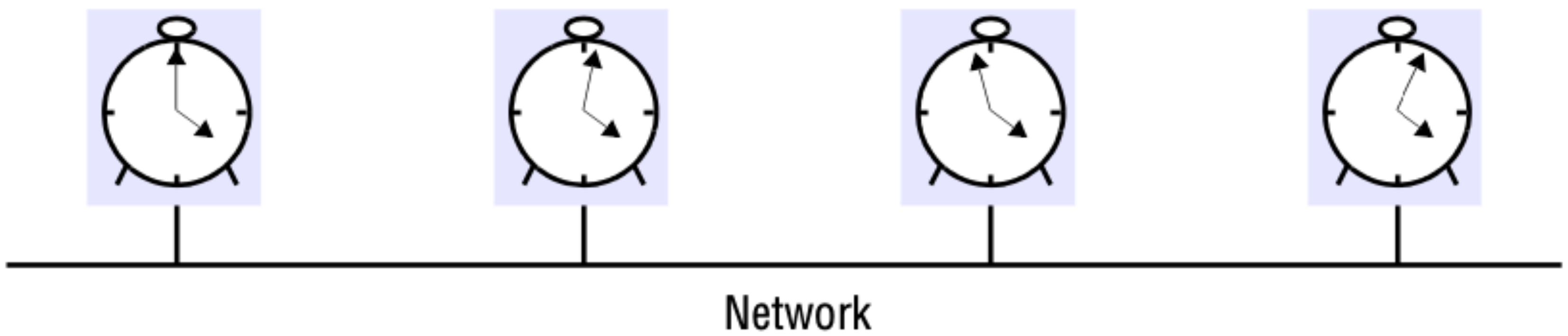
- When the Clock Resolution is smaller than the time interval between successive events,

  they are **different timestamps**.

2. Clocks, events and process states

Clock skew and clock drift

Clock skew : instantaneous difference between reading of 2 clocks



Clock drift : each computers count time at different rates

- the frequencies of oscillation are subject to physical environment *(such as temperature)*

Drift rate : change in the offset between the clock & reference clock

2. Clocks, events and process states

UTC (Coordinated Universal Time)

UTC : external source of highly accurate time, an international standard

- synchronized radio stations, satellites (GPS) broadcast UTC signal
- land-based stations accuracy : 0.1 ~ 10ms
- GPS satellites accuracy : about 1ms

Section 3

“ Synchronizing Physical Clocks ”

3. Synchronizing Physical Clocks

Synchronization

Synchronizing the processes' clock is necessary for accountancy purposes,

- t : real times in interval of real time I
- $S(t)$: source of UTC time
- $C_i(t)$: clock of process p_i (for $i = 1, 2, \ldots, N$)
- D : synchronization bound ($D > 0$)

External Synchronization

- $|S(t) - C_i(t)| < D \leftrightarrow$ **clocks C_i are *accurate* to with in the bound D**

Internal Synchronization

- $|C_i(t) - C_j(t)| < D \leftrightarrow$ **clocks C_i are *agree* to with in the bound D**

3. Synchronizing Physical Clocks

Synchronization

- If the system is **externally sync. with bound** D,

  the system is **internally sync. with bound** $2D$

Correctness for clocks : HW clock H is correct if **drift rate falls within known bound** ρ **> 0**

$$(1-\rho)(t'-t) \ \leq \ H(t') - H(t) \ \leq \ (1+\rho)(t'-t)$$

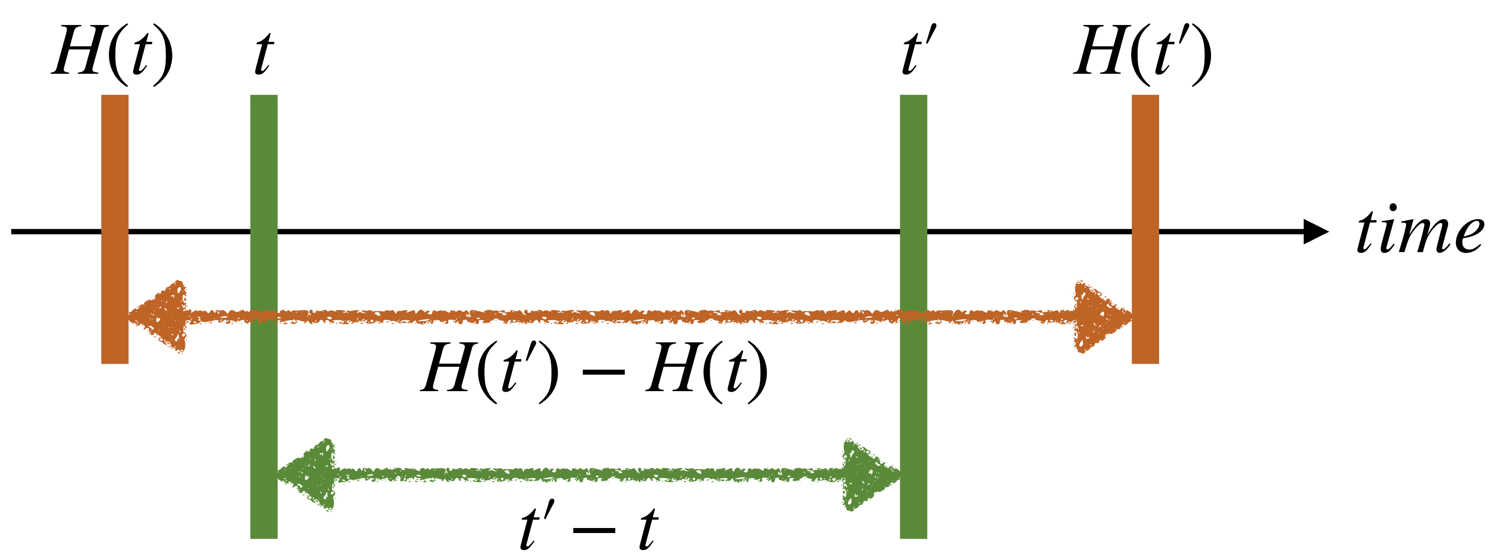
3. Synchronizing Physical Clocks

Synchronization

Correctness for clocks : HW clock H is correct if **drift rate falls within known bound ρ > 0**

$$(1-\rho)(t'-t) \leq H(t')-H(t) \leq (1+\rho)(t'-t)$$

$$(1-\rho) \leq \frac{H(t')-H(t)}{t'-t} \leq (1+\rho)$$



3. Synchronizing Physical Clocks

Synchronization

Monotonicity : the condition that a clock C only ever **advance**s

$$t' > t \Rightarrow C(t') > C(t)$$

Clock Failures

- **Faulty** : when a clock doesn't keep correctness condition
- **Crash Failure** : when the clock stops ticking altogether
- **Arbitrary Failure**

3. Synchronizing Physical Clocks

Synchronization in a synchronous system

Synchronous System : bounds are known for the drift rate of clocks,

the maximum message transmission delay,

the time required to execute each step of a process

- when process p_i send message m and local time t,

  receiver process p_j set its clock to be $t + T_{trans}$

  (T_{trans}: time taken to transmit m between p_i and p_j)

3. Synchronizing Physical Clocks

Synchronization in a synchronous system

- min : a minimum transmission time (always exists)
- max : an upper bound on the time taken to transmit any message
- $u = (max - min)$: the **uncertainty** in the message transmission time

- if receiver set its clock to be $t + min$, the clock skew may be u
- if receiver set its clock to be $t + max$, the clock skew may be also u
- if receiver set its clock to be $t + (\frac{min + max}{2})$, the clock skew may be $\frac{u}{2}$

- generally, the **optimum bound that can be achieve on clock skew** is $u(1 - \frac{1}{N})$

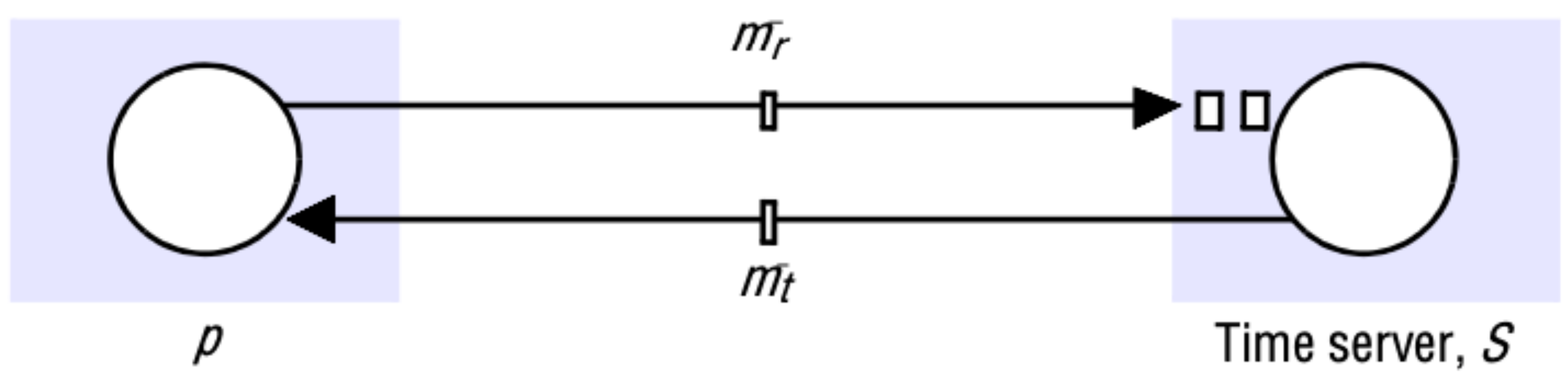
  (N : number of clocks)

3. Synchronizing Physical Clocks

Christian's method

: method for external synchronization that receives UTC source signal from time server

- no upper bound on message transmission delay,
  but round-trip times for message exchanging are often short

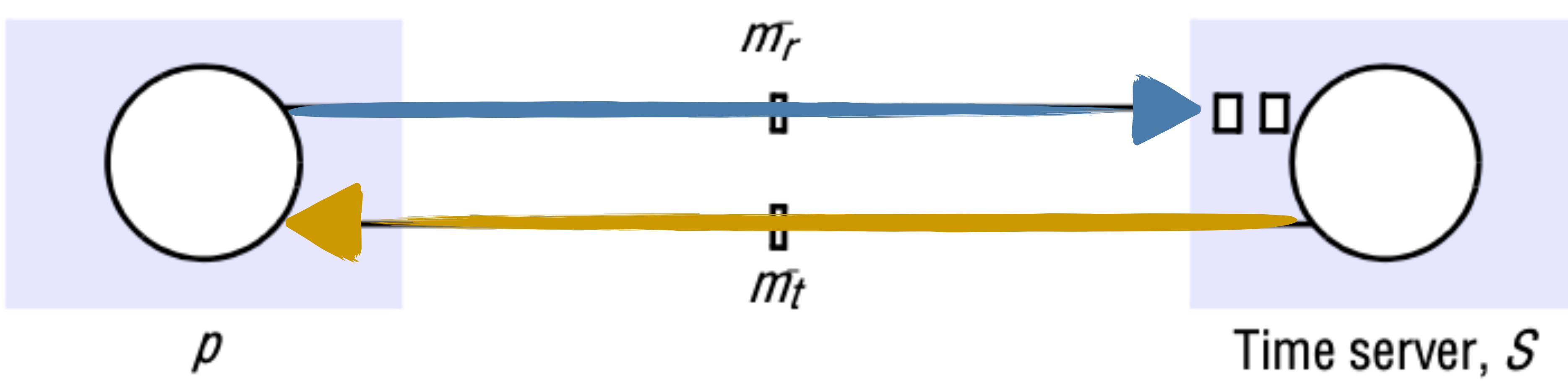


3. Synchronizing Physical Clocks

Christian's method

- m_t : message from time server S, including time value t
- T_{round} : **round-trip time** that taken to sends m_r and receive m_t
- setting clock to be $t + \dfrac{T_{round}}{2}$ makes sense,

  unless m_r, m_t are transmitted over difference networks



3. Synchronizing Physical Clocks

Christian's method

- S' time of sending m_t is in range $[t + min,\ t + (T_{round} - min)]$
- accuracy : $\pm(\frac{T_{round}}{2} - min)$

Discussion

- Cristian's method is for only single server
- error occurs when S fails or replies with incorrect time => the Berkeley Algorithm

3. Synchronizing Physical Clocks

The Berkeley Algorithm

: algorithm for internal synchronization

Coordinator computer : act as **master**, which polls slaves periodically

Slaves : multiple computers whose clocks are to be synchronized

- sends local time to master

- master estimates slaves' local clock time by observing T_{round} (similarly to Christian's method), averages the clock values

3. Synchronizing Physical Clocks

NTP (the Network Time Protocol)

- Christian's and Berkeley Algorithm are intended for use within intranets
- purpose of NTP is serving time information protocol over **the Internet**

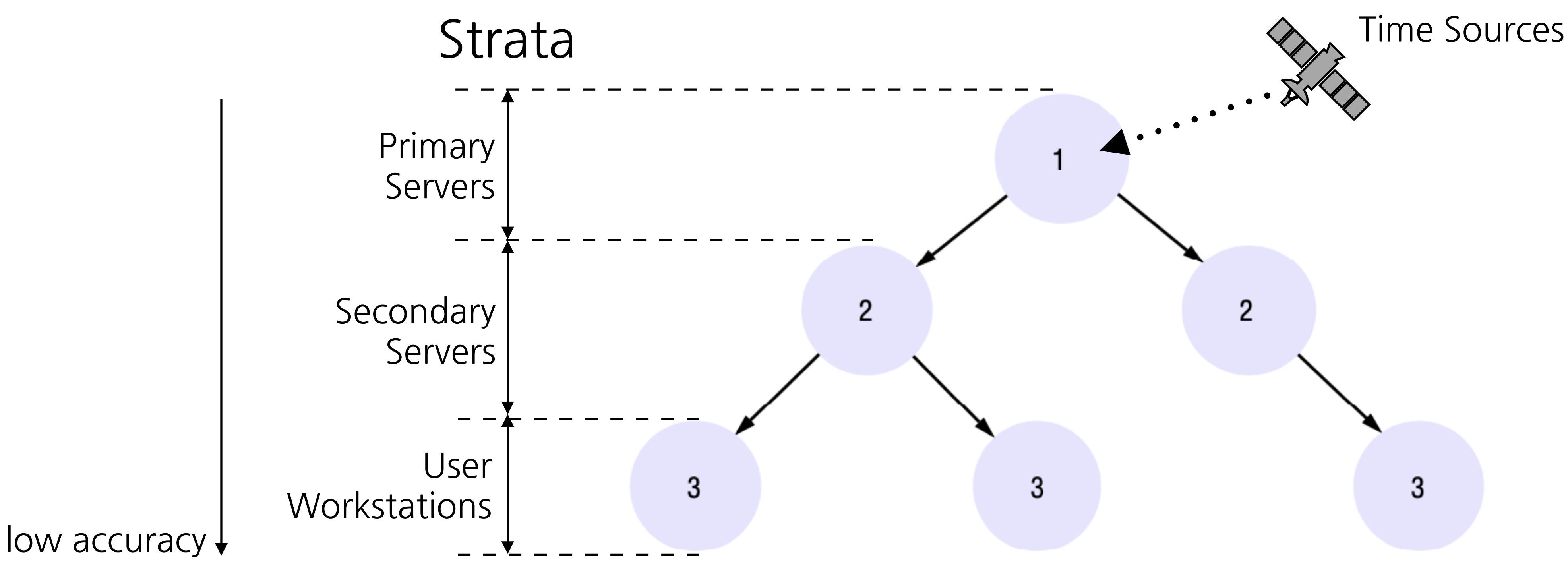
NTP's design aims

- To provide a service enabling clients across the Internet to be synchronized accurately to UTC
- To provide a reliable service that can survive lengthy losses of connectivity
- To enable clients to resynchronize sufficiently frequently to offset the rates of drift found in most computers
- To provide protection against interference with the time service, whether malicious or accidental

3. Synchronizing Physical Clocks

NTP (the Network Time Protocol)

Synchronization Subnet



3. Synchronizing Physical Clocks

NTP (the Network Time Protocol)

Multicast mode

- for **high-speed LAN**, relatively **low accuracy**
- servers periodically multicasts the time to the servers on LAN

Procedure-call mode

- suitable where higher accuracies are required than multicast mode
- one server accepts requests from other computers (replies message with its timestamp)

Symmetric mode

- for higher levels of the synchronization subnet (needs **highest accuracy**)
- a pair of servers exchange messages bearing time information

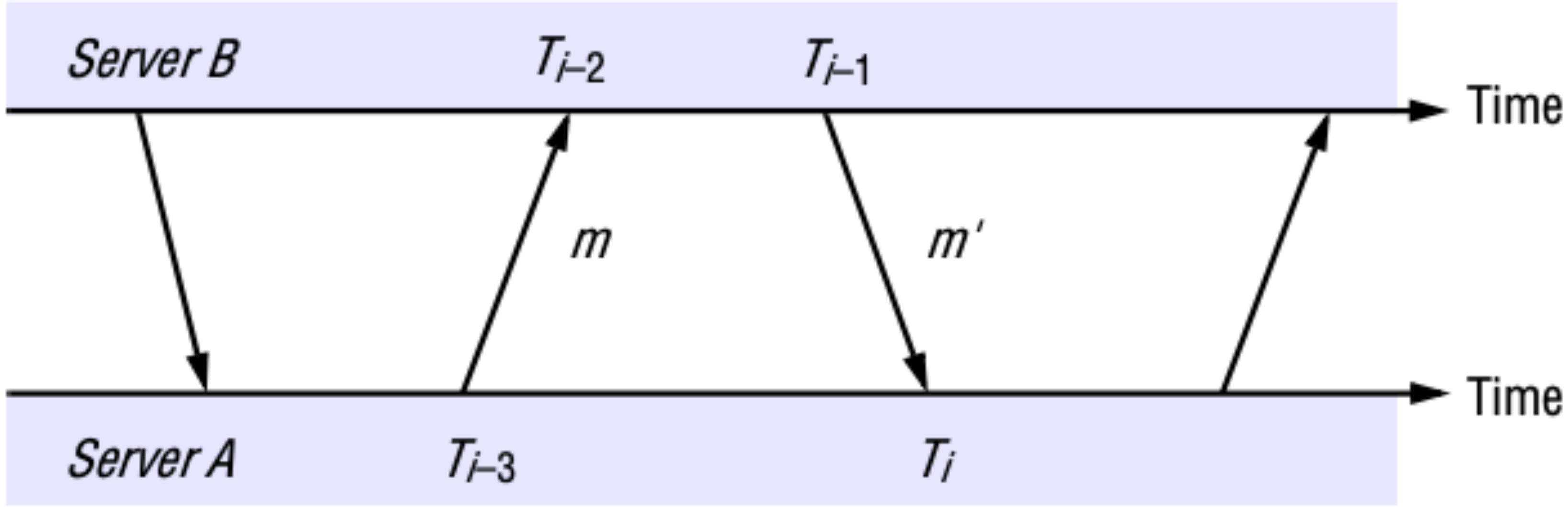
3. Synchronizing Physical Clocks

NTP (the Network Time Protocol)

- messages in all modes are delivered by using **UDP**

Messages in procedure-call mode & symmetric mode bears:

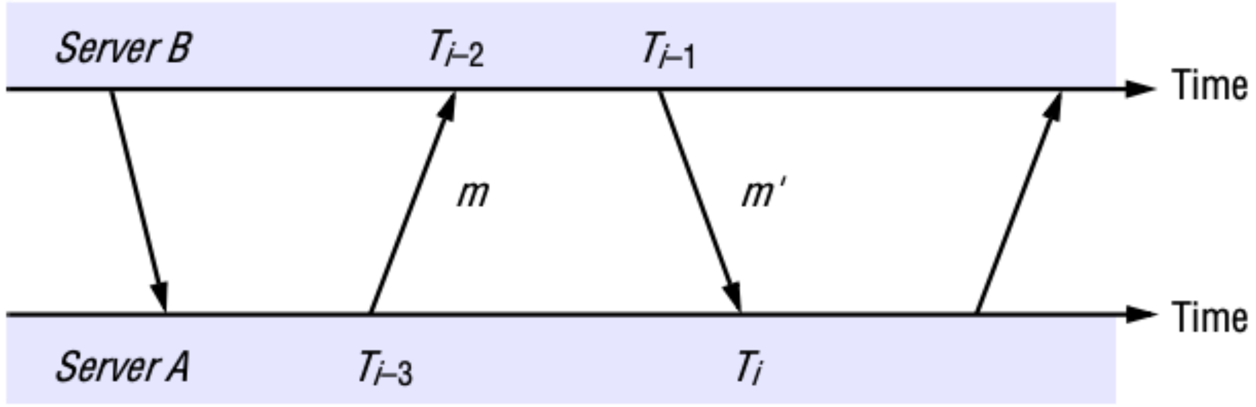
- local times when the previous NTP message between the pair was sent/received
- local times when the current message was transmitted



3. Synchronizing Physical Clocks

NTP (the Network Time Protocol)

- o_i : an **estimate of the actual offset** between the two clocks
- d_i : delay, the **total transmission time** for the two messages
- o : **true offset** of the clock at B relative to that at A
- t, t' : actual transmission times for m, m'



$$T_{i-2} = T_{i-3} + t + o\ ,\ T_i = T_{i-1} + t' - o$$

$$d_i = t + t' = (T_{i-2} - T_{i-3}) + (T_i - T_{i-1})$$

$$o = o_i + \frac{t' - t}{2}\ ,\ \text{where } o_i = \frac{T_{i-2} - T_{i-3} + T_{i-1} - T_i}{2} = \frac{t - t'}{2}$$

NTP (the Network Time Protocol)

$$o = o_i + \frac{t' - t}{2} \text{, where } o_i = \frac{T_{i-2} - T_{i-3} + T_{i-1} - T_i}{2} = \frac{t - t'}{2}$$

$$o_i - \frac{d_i}{2} \leq o \leq o_i + \frac{d_i}{2} \quad (t, t' \geq 0)$$

- o_i is an **estimate** of the offset, d_i is a **measure of the accuracy** of this estimate

Filter dispersion : a statistical quantity which represents **the quality of this estimate**

- NTP servers apply a data filtering algorithm to successive pairs $<o_i, d_i>$,

  the algorithm estimates the offset o, and calculates Filter dispersion

- High filter dispersion → relatively Unreliable data

3. Synchronizing Physical Clocks

NTP (the Network Time Protocol)

- NTP servers engages in message exchanges with several of peers to control local clock,
- applies a **peer-selection algorithm** to examine values with each of several peers

Synchronization dispersion : the **sum of the filter dispersions**,
which measured between the server & the root of the synchronization subnet

- peers exchange synchronization dispersion in messages

Section 4

“ Logical time & Logical clocks ”

4. Logical time and logical clocks

Happened-before Relation

Since we cannot synchronize clocks perfectly across a distributed system,

we cannot in general use physical time to find out the order of any arbitrary pair of events

- If two events that occurred at the same process p_i,

  then they occurred in the order in which p_i observes them

  $\Rightarrow$ **the order** $\rightarrow_i$

- Whenever a message is sent between processes,

  the **sending event occurred before the receiving event**

4. Logical time and logical clocks

Happened-before Relation

- HB1 : If ∃ process p_i : $e \rightarrow_i e'$, then $e \rightarrow e'$
- HB2 : For any message m, $send(m) \rightarrow receive(m)$
- HB3 : If e, e' and e'' are events such that $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

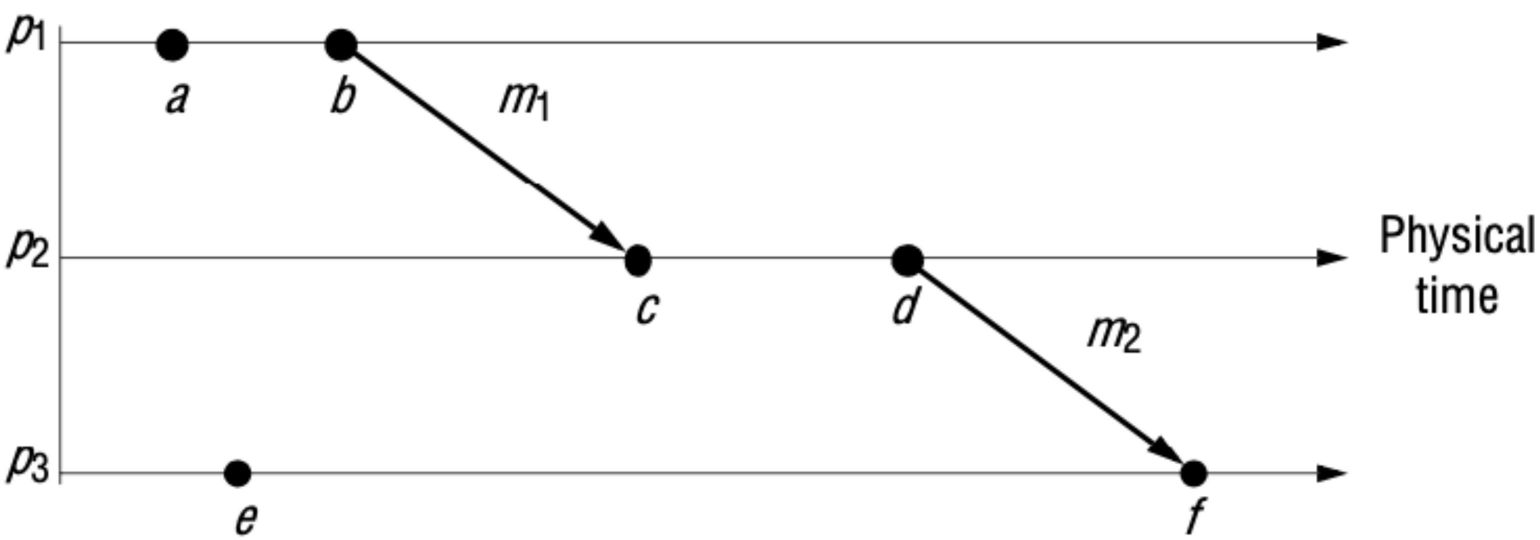
$a \rightarrow b$ (by HB1)

$b \rightarrow c$ (by HB2)

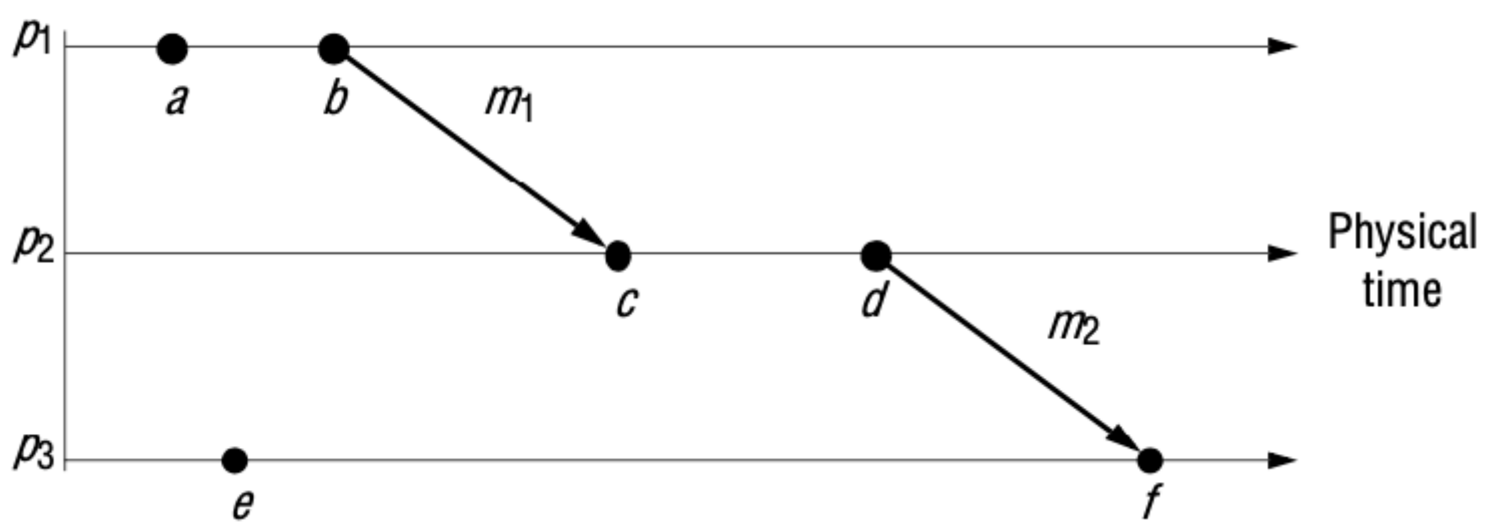
$c \rightarrow d$ (by HB1)

$d \rightarrow f$ (by HB2)

$\therefore\ a \rightarrow f$ (by HB3)



Happened-before Relation



$a \nrightarrow e, e \nrightarrow a$

$\Rightarrow a \parallel e$ (a and e are **concurrent**)

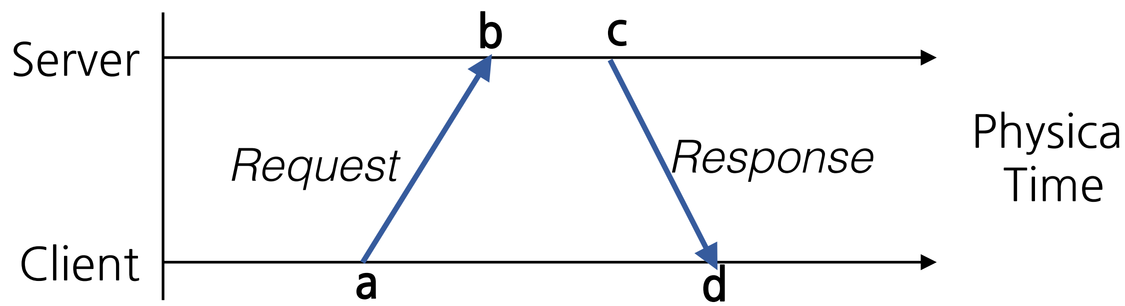
4. Logical time and logical clocks

Happened-before Relation

Limitation

- Cannot model when the data flow in ways other than by message passing
- when a server receives a request message and subsequently sends a reply,

  b, c can be related by $\rightarrow$ even though there is no real connection between them



4. Logical time and logical clocks

Logical clocks

: a simple mechanism by which the happened-before ordering can be captured numerically

Each process p_i keeps its own **Logical Clock** L_i,

which it uses to apply **Lamport timestamps** to events

- $L_i(e)$: timestamp of event e at process p_i
- $L(e)$: timestamp of event e at whatever process it occurred at

4. Logical time and logical clocks

Logical clocks

To capture the happened-before relation $\rightarrow$,

processes update their logical clocks in messages as:

- LC1 : L_i is incremented before each event is issued at process p_i :

$$L_i := L_i + 1$$

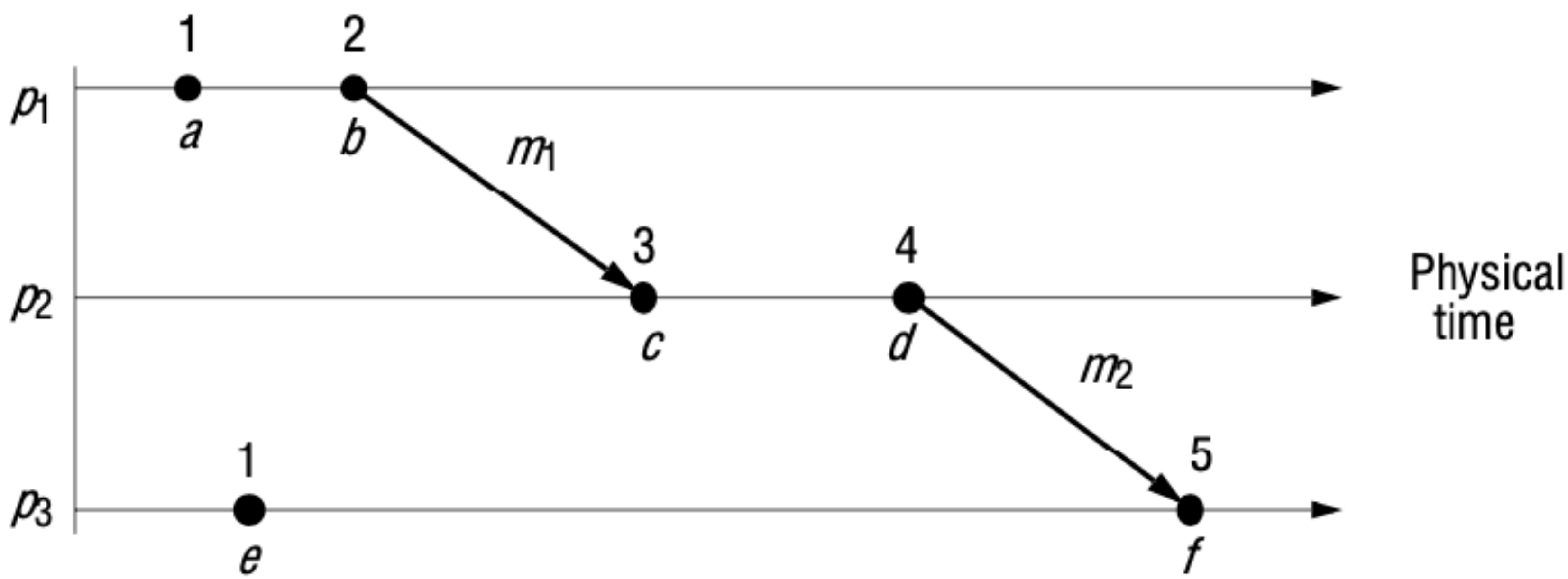
- LC2 : (a) When a process p_i sends a message, m bears $t = L_i$

  (b) On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$ and then

  applies LC1 before timestamping the event $receive(m)$

4. Logical time and logical clocks

Logical clocks



- *If events e, e' are related to each other, then* $e < e' \Rightarrow L(e) < L(e')$
- but, the **converse is not always true**
  - Counterexample : $L(b) > L(e)$ but $b \parallel e$

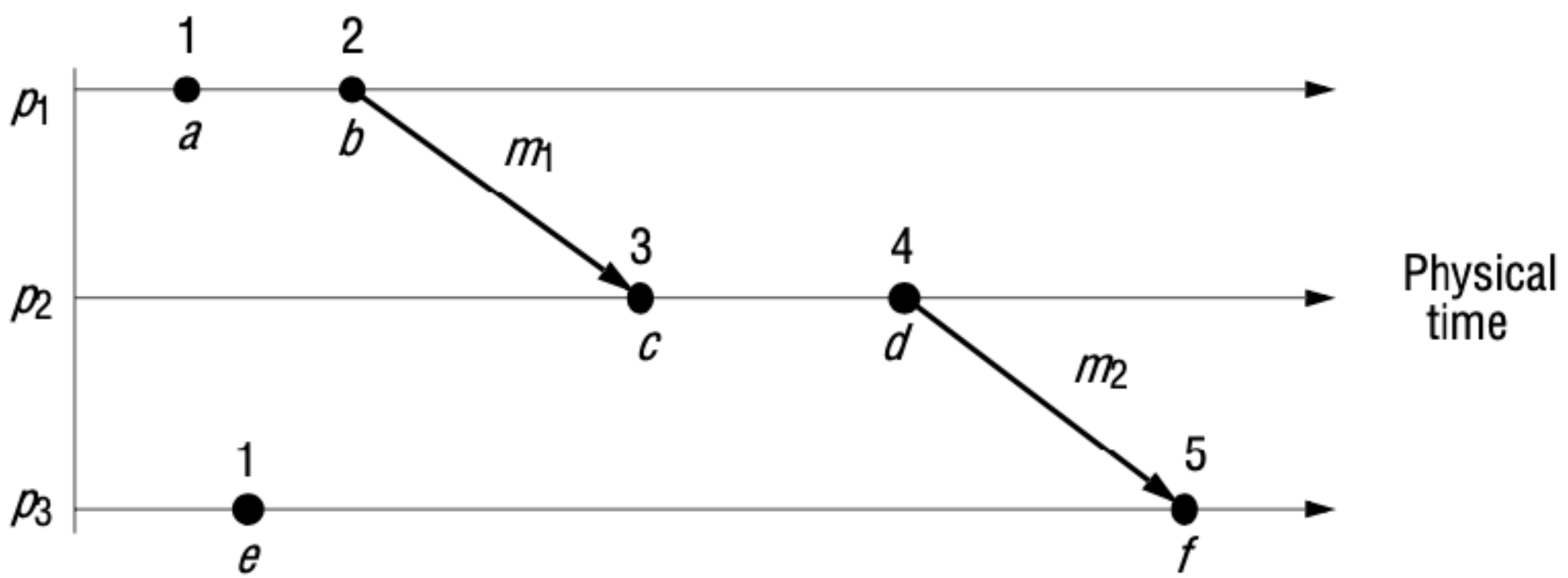
4. Logical time and logical clocks

Totally ordered logical clocks

Sometimes we need **a total order on the set of events**,

but some pairs of distinct events have numerically identical Lamport timestamps:

$L(a) = 1,\ L(e) = 1$



4. Logical time and logical clocks

Totally ordered logical clocks

Global logical timestamp : timestamp for ordering **entire set of events**

- e : an event occurring at process p_i with local timestamp T_i
- e' : an event occurring at process p_j with local timestamp T_j
- (T_i, i) : global logical timestamp

we define $(T_i, i) < (T_j, j)$ *iff either* ($T_i < T_j$) *or* ($T_i = T_j$ *when* $i < j$)

(in previous figure, $a < e$)

- there's <u>no general physical significance</u>, but sometimes useful:

  to order the entry of processes to a critical section

4. Logical time and logical clocks

Vector clocks

Lamport's logical clock has a shortcoming:

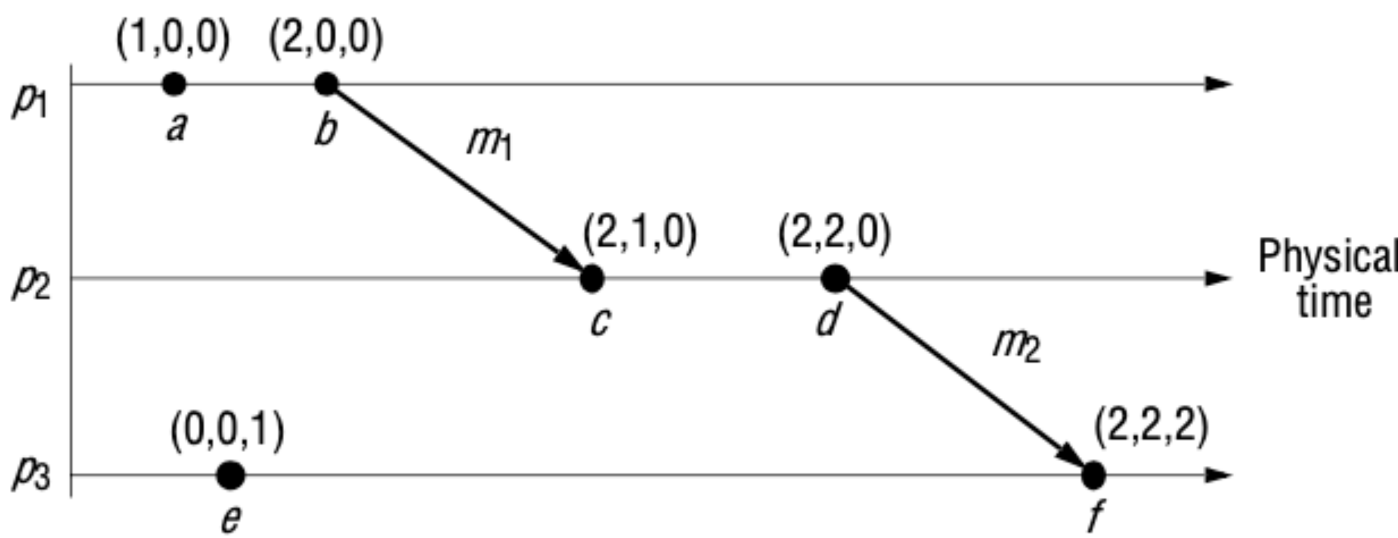
the fact that from $L(e) < L(e')$ we cannot conclude that $e \rightarrow e'$

Vector clock for a system of N processes is **an array of N integers**

- each processes keeps its own vector clock V_i
- processes piggyback vector timestamps on the messages they send to one another
- Rules for updating the Vector clock:
  - VC1 : Initially, $V_i[j] = 0$, for $i, j = 1, 2, \ldots, N$
  - VC2 : Just before p_j timestamps an event, it sets $V_i[i] := V_i[i] + 1$
  - VC3 : p_i includes the value $t = V_i$ in every message it sends
  - VC4 : When p_i receives a timestamp t in a message, it sets $V_i[j] := max(V_i[j], t[j])$

4. Logical time and logical clocks

Vector clocks



Rule of comparing vector timestamps:

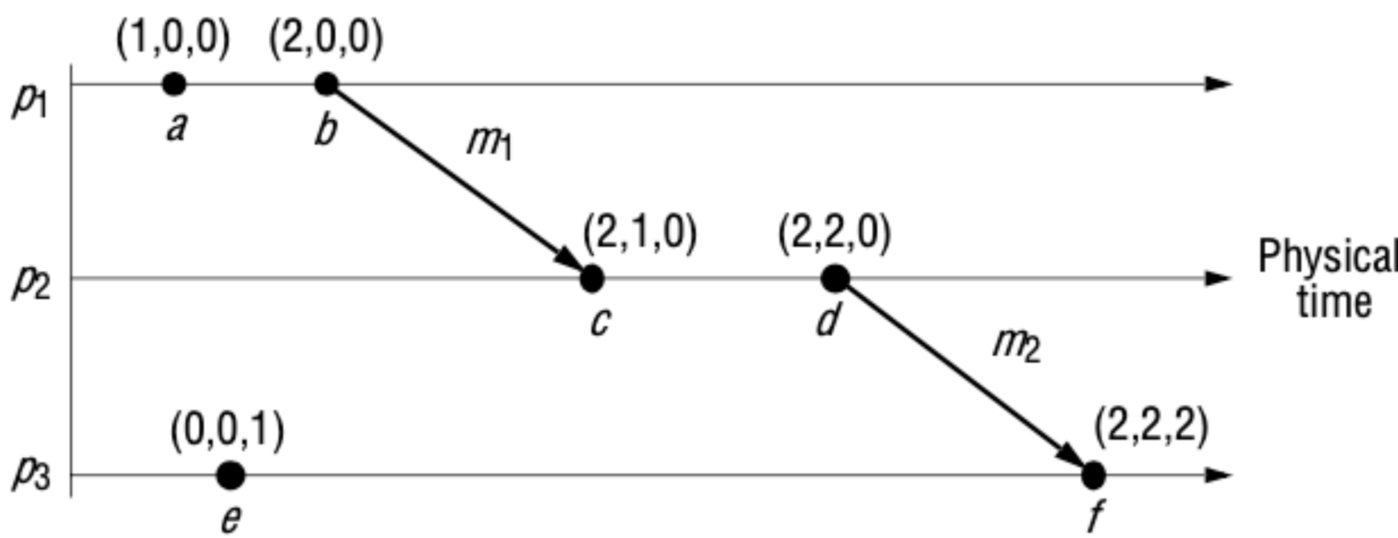
$$V = V' \iff V[j] = V'[j] \ \textit{for}\ j = 1,2,\ldots,N$$

$$V \leq V' \iff V[j] \leq V'[j] \ \textit{for}\ j = 1,2,\ldots,N$$

$$V < V' \iff (V \leq V') \wedge (V \neq V')$$

4. Logical time and logical clocks

Vector clocks



when $V(e)$ is the **vector timestamp** applied by the process at which e occurs,

$e \rightarrow e' \Rightarrow V(e) < V(e')$ (also the **converse is true**)

$V(a) < V(f)$ can be seen from the fact that $a \rightarrow f$

$c \parallel e$ can be seen from the facts that neither $V(c) \leq V(e)$ nor $V(e) \leq V(c)$

4. Logical time and logical clocks

Vector clocks

Disadvantages

- needs **an amount of storage and message payload**

⇒ some techniques exist for storing and transmitting smaller amounts of data:

Raynal and Singhal's ***Matrix clocks*** (processes keep estimates of other processes' vector times)

Section 5

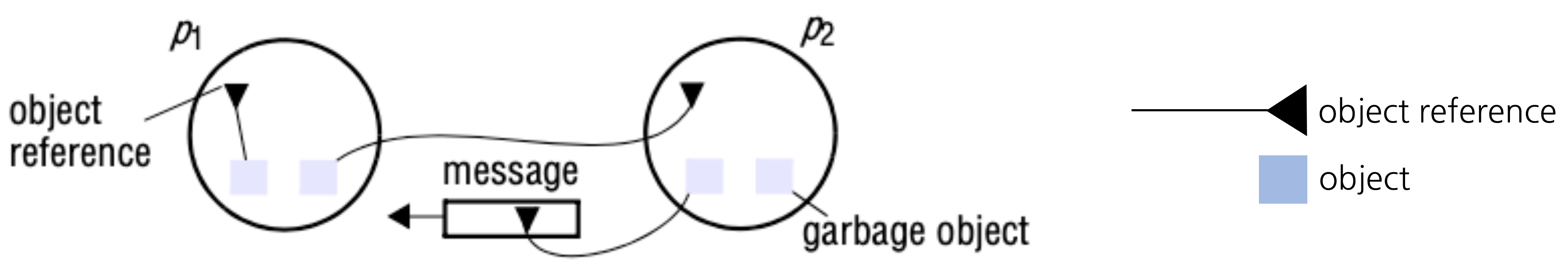
“ Global States ”

5. Global States

Problems of Distributed systems

Distributed garbage collection

- if there are **no longer any references** to an object, it is considered to be **Garbage**



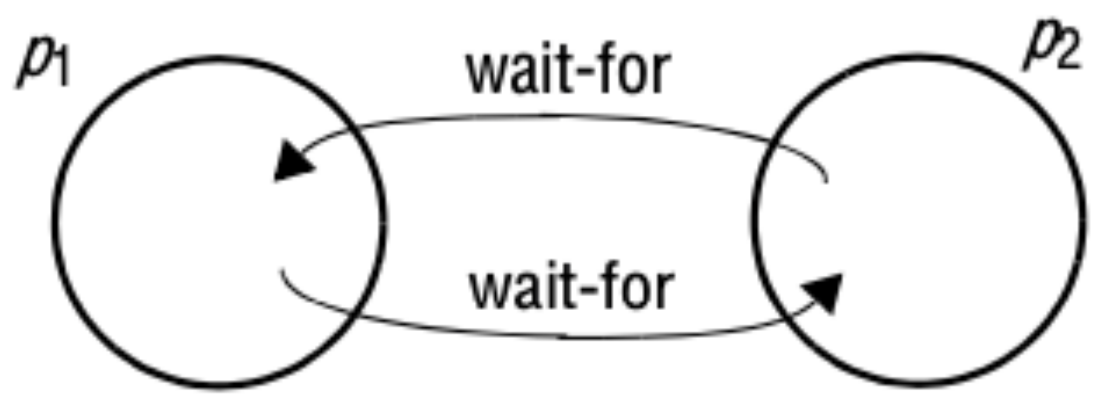
- in p_2, there is a garbage object with no references
- a reference can be in a message

5. Global States

Problems of Distributed systems

Distributed deadlock detection

- occurs when each of a collection of processes waits for another process to send it a message, and where there is a cycle in the graph of this ***waits-for relationship***



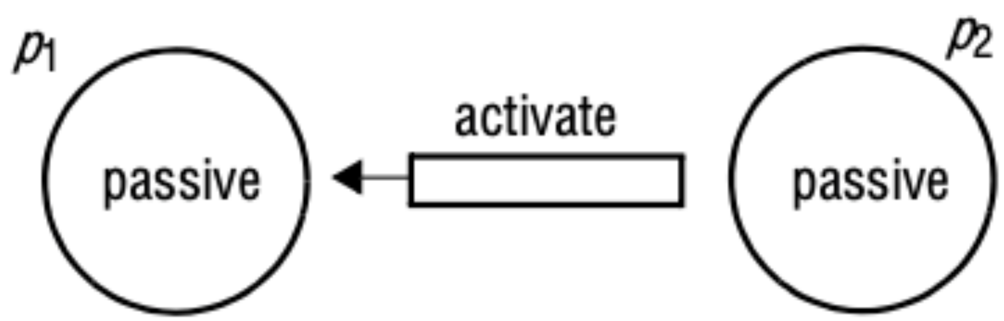
5. Global States

Problems of Distributed systems

Distributed termination detection

: when the processes are in passive state, we may not conclude that the algorithm has terminated

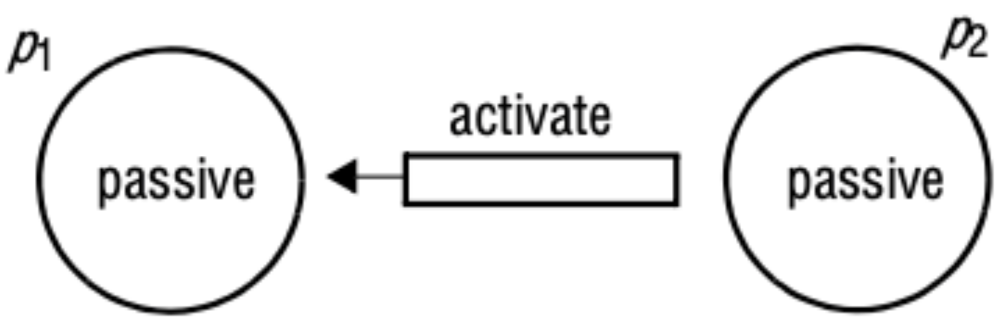
- **Passive process** : not engaged in any activity of its own,
  but is prepared to respond with a value requested by the other



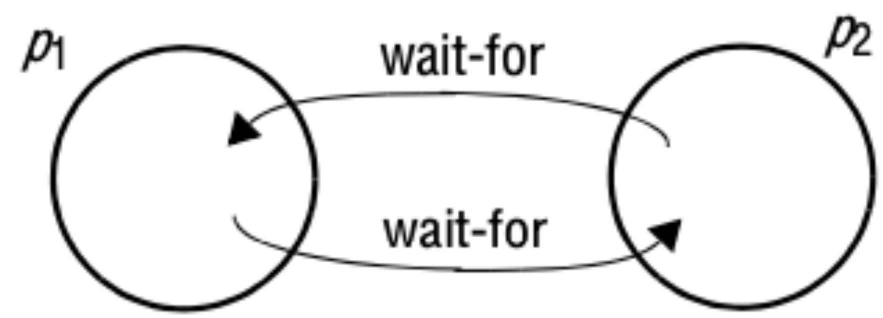
5. Global States

Problems of Distributed systems

Distributed termination detection vs **Distributed deadlock detection**



[termination detection]



[deadlock detection]

- deadlock may affect only in **system, whereas all processes must have terminated**
- a deadlocked process is attempting to perform a further action, for which another process waits;
  a **passive process is not engaged in any activity**

5. Global States

Problems of Distributed systems

Distributed debugging

- Distributed systems are complex to debug,

  and care needs to be taken in establishing what occurred during the execution

5. Global States

Global states and Consistent cuts

Global state : the state of the collection of processes

- ascertaining a global state is much harder to address, because of the **absence of global time**
- it is possible to assemble a meaningful global state from local states, but we some definitions:

$$history(p_i) = h_i = < e_i^0, e_i^1, \ldots >$$

$h_i^k = < e_i^0, e_i^1, \ldots, e_i^k >$ (any finite prefix of the process's history)

- each event is an **internal action of the process**, or is **the sending or receipt of a message**
- each process can record the events, and the succession of states it passes through

- s_i^k : the **state** of process p_i immediately **before the kth event** occurs

5. Global States

Global states and Consistent cuts

Global history

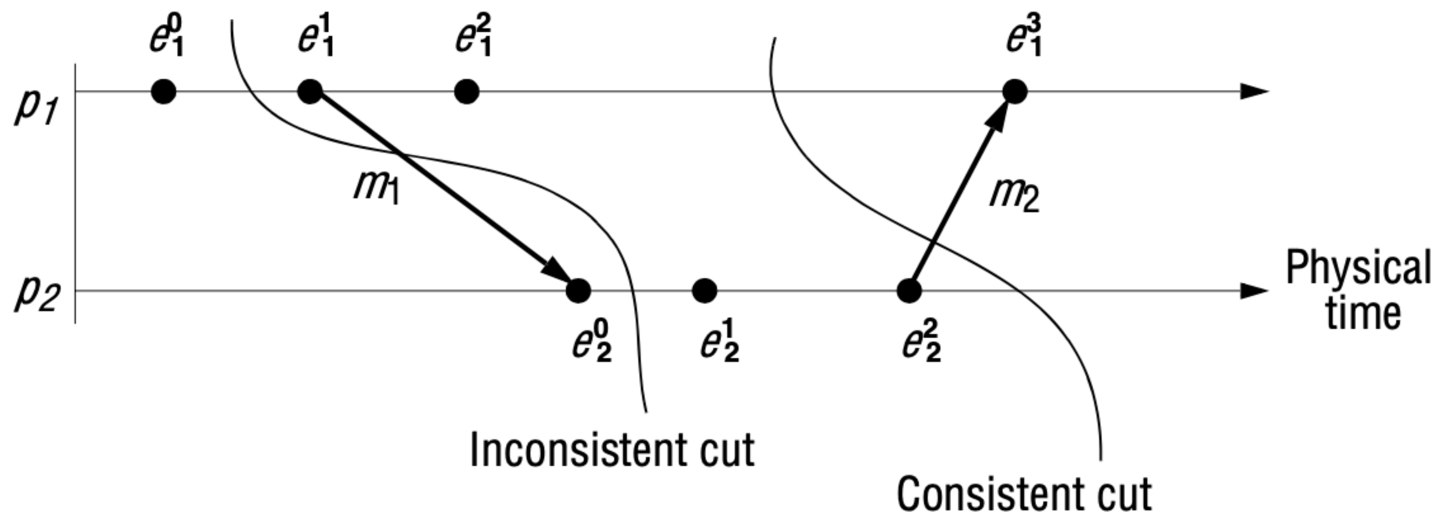
$$H = h_0 \cup h_1 \cup \ldots \cup h_{N-1}$$

Cut of the system's execution is a **subset of its global history** that is a union of prefixes of process histories:

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_N^{c_N}$$

- $e_i^{c_i}$: state of p_i immediately after the last event processed by p_i in the cut
- **Frontier** is the set of events $\{e_i^{c_i} : i = 1,\ldots,N\}$

Global states and Consistent cuts



- the figure above shows two cuts:

one with frontier $< e_1^0, e_2^0 >$ & another with frontier $< e_1^2, e_2^2 >$

- the leftmost cut is an **inconsistent cut**, because it is showing an ***effect* without a *cause***

(p_2 includes the receipt of the m_1, but p_1 doesn't include the sending of $m1$)

- the rightmost cut is a **consistent cut**

5. Global States

Global states and Consistent cuts

If a cut C is consistent, it also contains all the events that happened-before that event:

for all events $e \in C,\ f \rightarrow e\ \Rightarrow f \in C$

we may characterize the execution of a distributed system

as **a series of transitions between global states** of the system:

$S_0 \rightarrow S_1 \rightarrow \ldots$

- transition : one event occurs at some single process

5. Global States

Global states and Consistent cuts

Run : a total ordering of all the events in a global history that is consistent with each local history's ordering

Linearization (Consistent Run) : an ordering of the events in a global history that is consistent with this happened-before relation $\rightarrow$ on H

- linearization is also a run
- not all runs pass through consistent global states,
  but all linearizations pass only through consistent global states

- if there is a linearization that passes through state S and S',
  then S' ***is reachable from*** S

5. Global States

Global state predicates, stability, safety and liveness

Global state predicate : a function that maps from the set of **global states** of processes to **{T/F}**

- detects a condition such as **deadlock** or **termination**
- the predicates are all **stable**

    (once the system enters a state *True*, it remains *True* in all future states)

5. Global States

Global state predicates, stability, safety and liveness

Safety

- (let α is an undesirable property that is a predicate of the deadlocked system's global state)
- Safety with respect to α evaluates to **False for all states** S **reachable from** S_0
- guarantee that something good will happen, eventually

Liveness

- (let β is an undesirable property that is a property of reaching termination)
- Safety with respect to β is the property that, for any **linearization** L **starting in the state** S_0,

  β evaluates to **True for some state** S_L **reachable from** S
- guarantee that something bad will never happen

5. Global States

The **Snapshot** algorithm of Chandy & Lamport

Snapshot Algorithm

- the goal is to **record a set of process and channel states** for processes p_i

  even though the combination of states may never have occurred at the same time,

  the global state is **consistent**

5. Global States

The **Snapshot** algorithm of Chandy & Lamport

Assumptions of the algorithm

- Neither channels nor processes fail : **communication is reliable**
- Channels are **unidirectional** and provide **FIFO**-ordered message delivery
- The **graph** of processes and channels is **strongly connected**

  (always a path exists between any two processes)
- Any processes may initiate a global snapshot at any time
- Processes **may continue their executions** while the snapshot takes place

- There are *incoming channel*s and *outgoing channel*s

5. Global States

The **Snapshot** algorithm of Chandy & Lamport

Each process records:

- its **state**
- for each **incoming channel**s, a set of messages sent to it
- any messages that arrived after it recorded its state and before the sender recorded its own state

This arrangement allows us to record the **states of processes at different times**,

but to account for the differentials between process states in messages transmitted but not yet received

(if p_i has sent a message m to p_j, but p_j has not received it,

then we account for m as **belonging to the state of the channel between them**)

The **Snapshot** algorithm of Chandy & Lamport

Marker messages

- special message that used for proceeding the algorithm
- distinct from any other messages
- the processes may send/receive markers, while they proceed with their normal execution

- roles of the Marker:
    - as a **prompt for the receiver to save its own state**, if it has not already done so
    - as a means of **determining which messages to include** in the channel state

The **Snapshot** algorithm of Chandy & Lamport

Snapshot algorithm - *Marker Sending Rule*

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).

- process must **send a marker after they have recorded** their state,
  but **before they send any other messages**

  (record → send marker → send message)

The **Snapshot** algorithm of Chandy & Lamport

Snapshot algorithm - *Marker Receiving Rule*

Marker receiving rule for process p_i

On receipt of a *marker* message at p_i 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

5. Global States

The **Snapshot** algorithm of Chandy & Lamport

Snapshot algorithm

- any process **may begin the algorithm at any time**

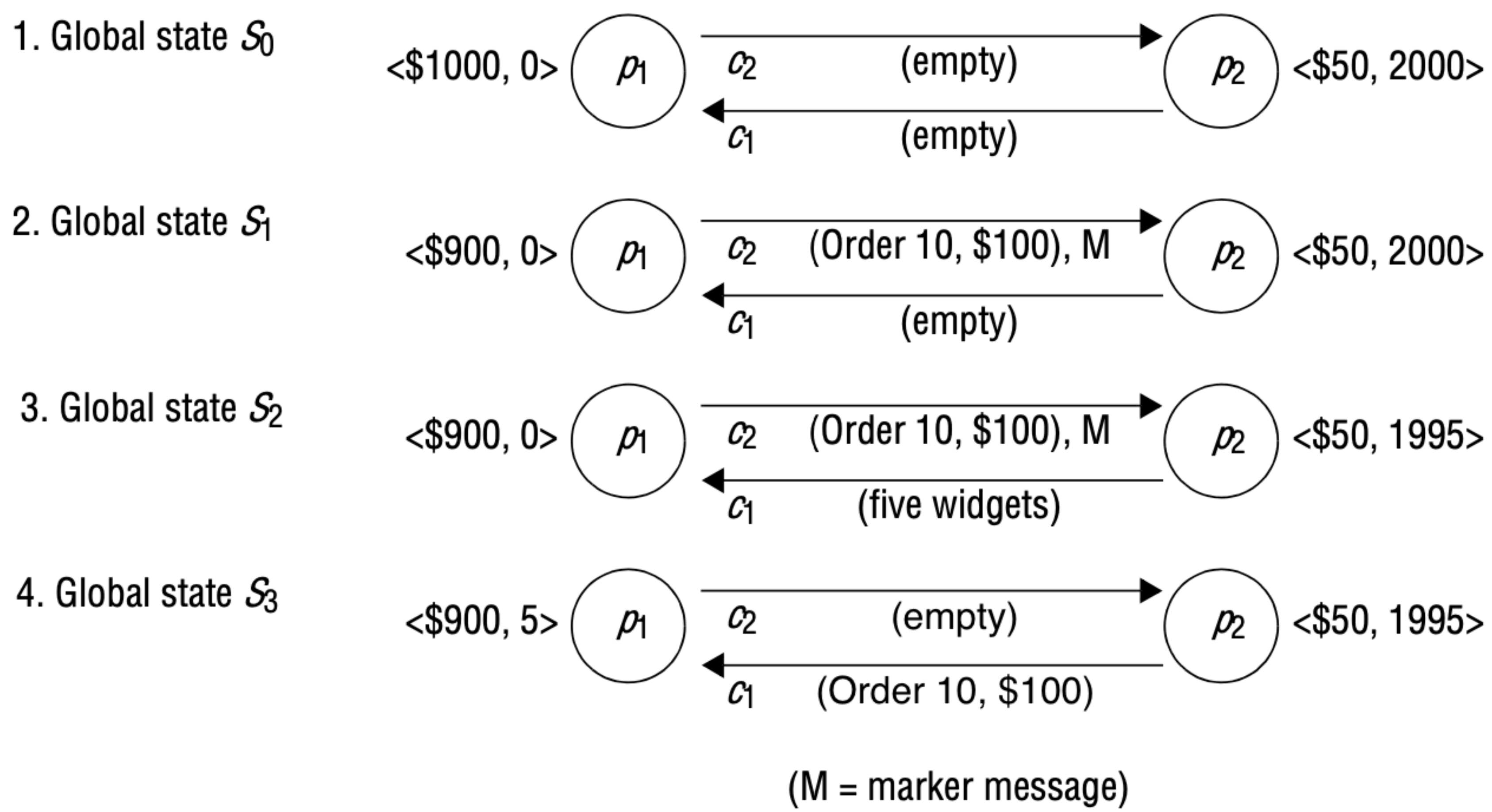
  ⇒ it acts as though it has received a marker (over a nonexistent channel)

- several processes **may initiate recording concurrently** in this way

5. Global States

The **Snapshot** algorithm of Chandy & Lamport

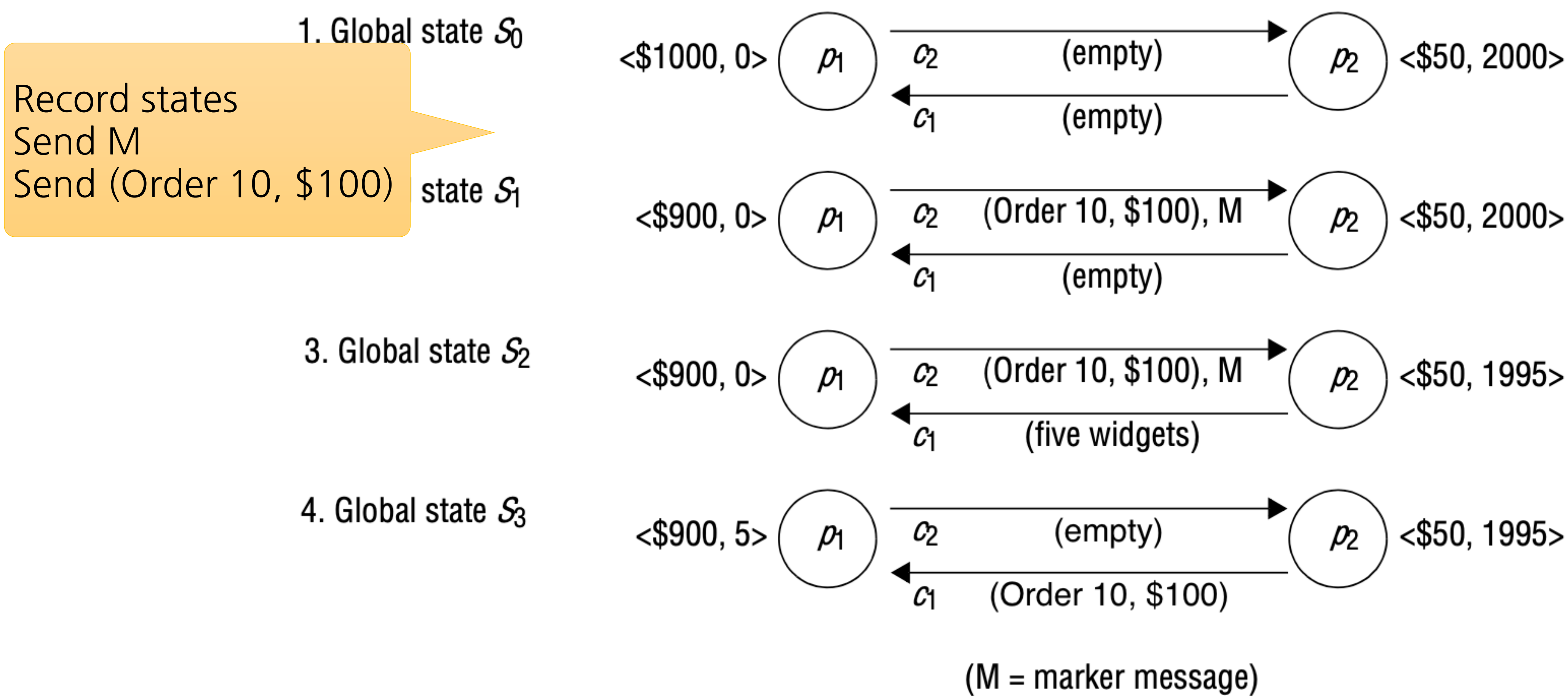
Snapshot algorithm



5. Global States

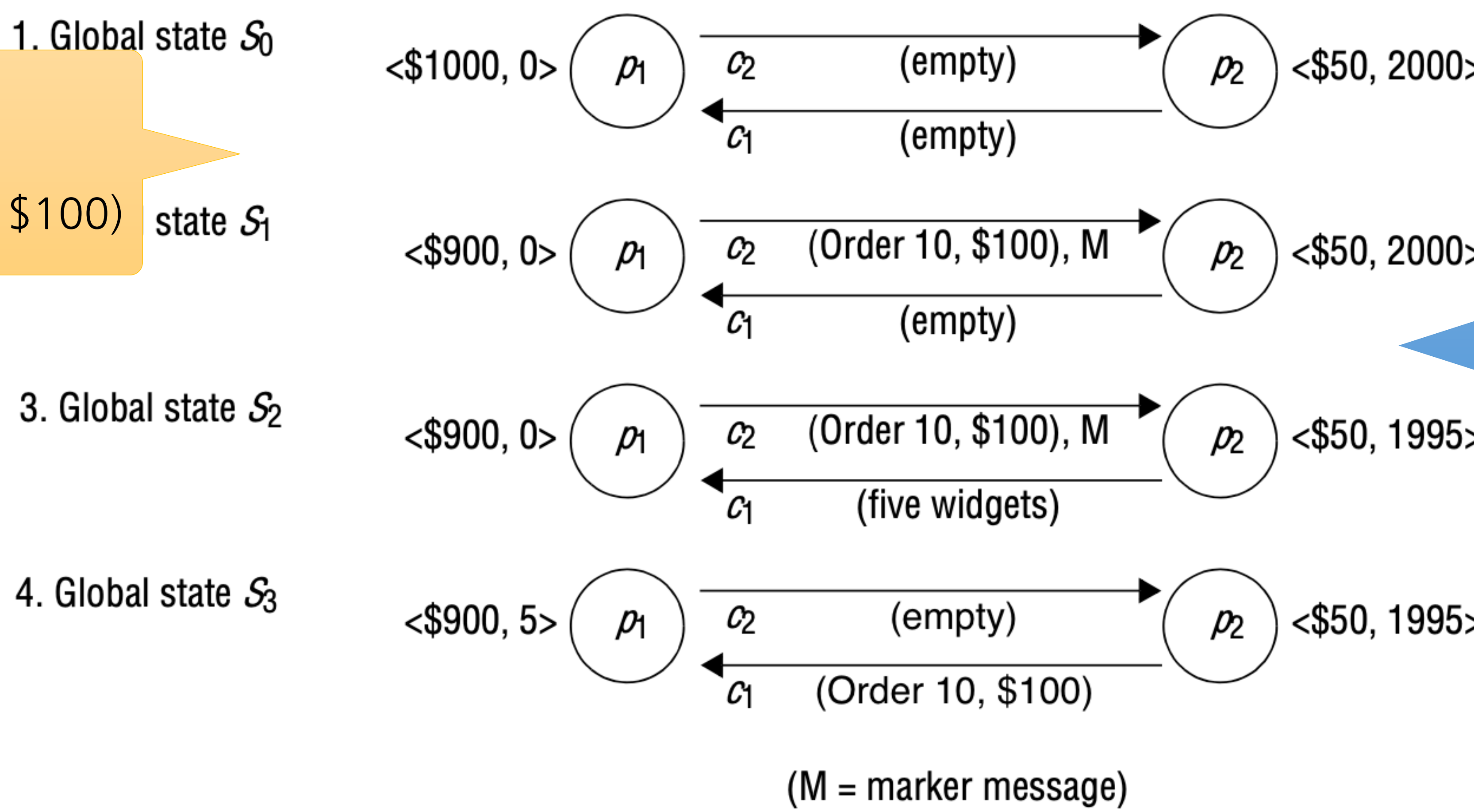
The **Snapshot** algorithm of Chandy & Lamport

Snapshot algorithm



The **Snapshot** algorithm of Chandy & Lamport

Snapshot algorithm



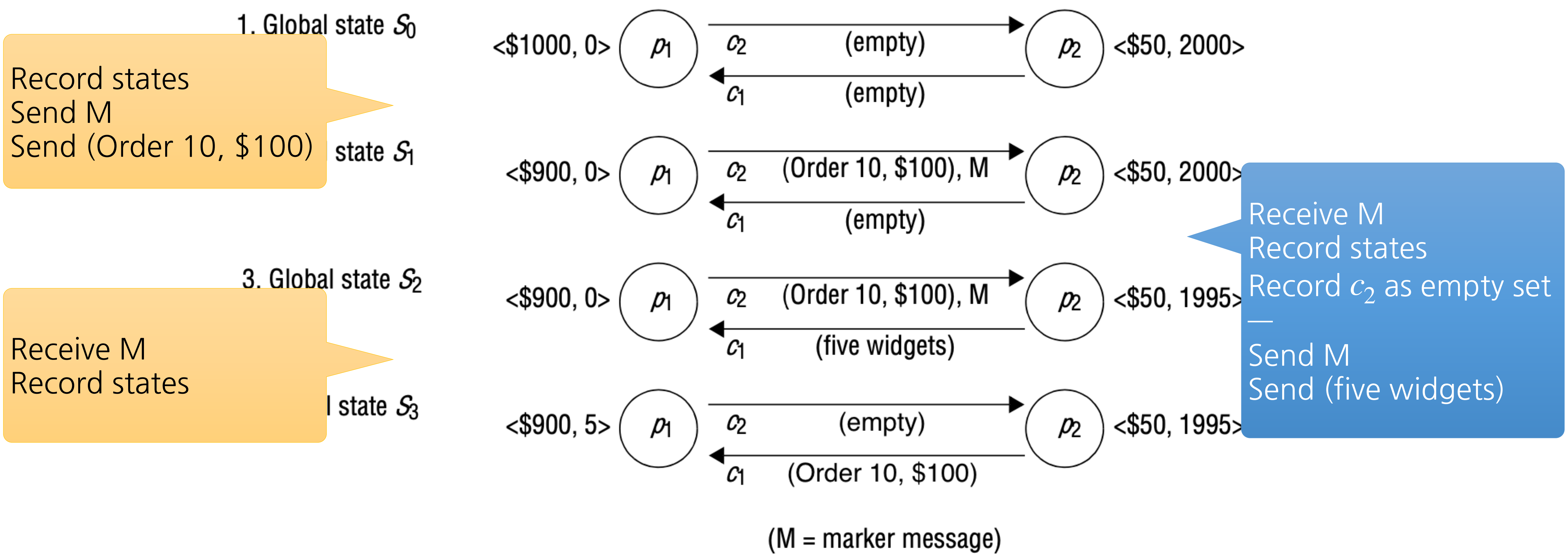
Record states
Send M
Send (Order 10, $100)

Receive M
Record states
Record c_2 as empty set
—
Send M
Send (five widgets)

5. Global States

The **Snapshot** algorithm of Chandy & Lamport

Snapshot algorithm



5. Global States

The **Snapshot** algorithm of Chandy & Lamport

Termination of the Snapshot algorithm

- we assume that a process that:

  has received a marker message records its state **within a finite time**,

  sends marker messages over each outgoing channel **within a finite time**

- since we assume that the graph of processes and channels to be **strongly connected**,

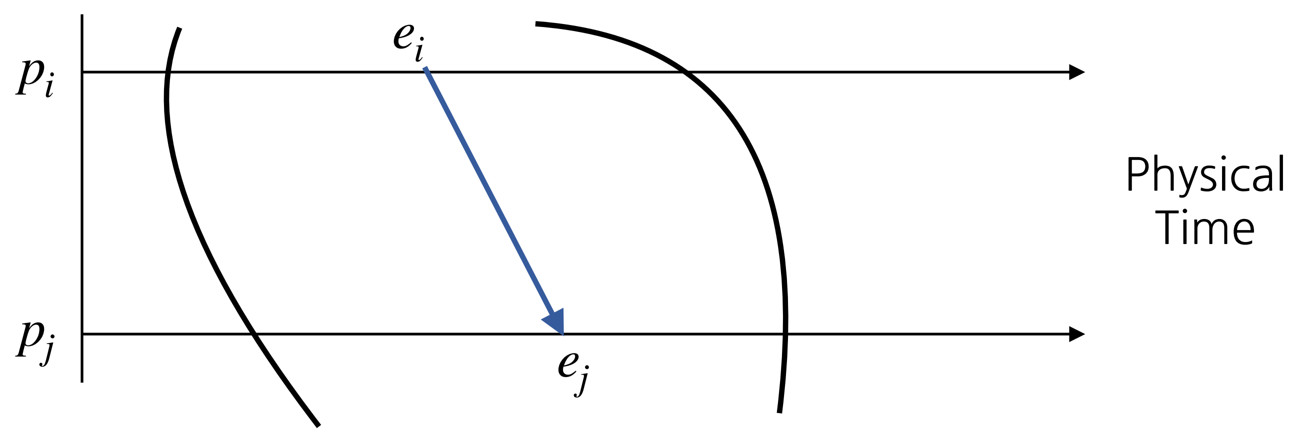
  it follows that all processes will have recorded states **a finite time after some process initially records its state**

5. Global States

The **Snapshot** algorithm of Chandy & Lamport

Characterizing the observed state

- Snapshot algorithm **selects a *cut* from the history** of the execution
- and this *cut*, which is the state recorded by the algorithm, is **consistent**



- let e_i and e_j, : $e_i \rightarrow e_j$
- if e_j is in the cut, then e_i is in the cut
- if e_j occurred before p_j recorded its state, then e_i must have occurred before p_i recorded its state

5. Global States

The **Snapshot** algorithm of Chandy & Lamport

$Sys = e_0, e_1, \ldots$: the linearization of the system as it executed

S_{init} : the global state immediately **before the first process recorded its state**

S_{snap} : the **recorded** global state

S_{final} : the global state **when the snapshot algorithm terminates**,

immediately after the last state-recording action

$Sys' = e_0', e_1', \ldots$: a permutation of Sys such that all 3 states (S_{init}, S_{snap}, S_{final}) occurs

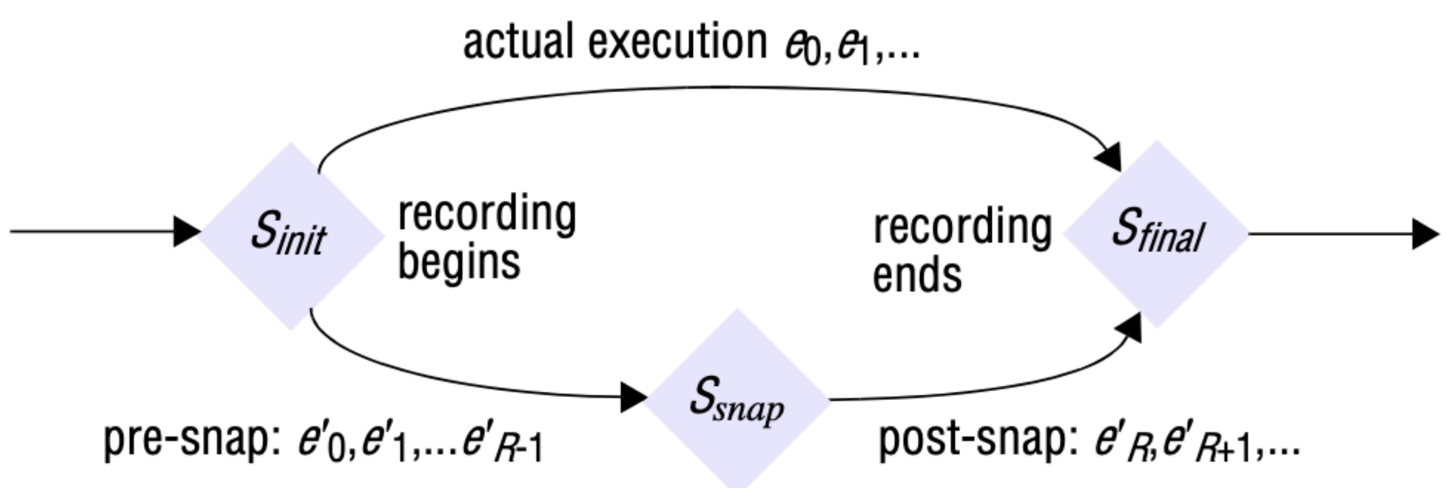
- S_{snap} is reachable from S_{init}, S_{final} is reachable from S_{snap}

5. Global States

The **Snapshot** algorithm of Chandy & Lamport

$Sys' = e_0', e_1', \ldots$: a permutation of Sys such that all 3 states (S_{init}, S_{snap}, S_{final}) occurs

- S_{snap} is reachable from S_{init}, S_{final} is reachable from S_{snap}



pre-snap : events which occurred at p_i before it recorded its state (before S_{snap})

post-snap : events which occurred at p_i after it recorded its state (after S_{snap})

(if events occur at different processes, a post-snap may occur before a pre-snap)

5. Global States

The **Snapshot** algorithm of Chandy & Lamport

the reachability property of the snapshot algorithm is **useful for detecting stable predicates**

- (***stable*** : once true, stays true forever afterwards)
- non-stable predicate we establish as being *True* in S_{snap} may or may not have been *True* in the actual execution
- if a stable predicate is *True* in S_{snap}, all reachable state in S is *True*

Stable liveness example

- computation has terminated

Stable non-safety example

- there is a deadlock

Section 6

“ Distributed Debugging ”

6. Distributed Debugging

Introduction of Distributed Debugging

we now examine the problem of recording a system's global state,

so that we may make useful statements about **whether a transitory state occurred**

in an actual execution

- x_i : variable in process p_i
- **safety condition** $|x_i - x_j| \leq \delta$ is to be met even though a process may change the value of its

  variable at any time

6. Distributed Debugging

Introduction of Distributed Debugging

a distributed system controlling a system of pipes in a factory,

where we are interested in whether all the valves were open at some time

(valves are controlled by different processes)

- in general, we cannot observe the values or the states of the valves **simultaneously**

- the challenge is to monitor system's execution over time

  to **capture *trace* information** rather than a single snapshot,

  so that we can establish post hoc whether the required safety condition was violated

6. Distributed Debugging

Introduction of Distributed Debugging

Chandy & Lamport's snapshot algorithm collects state in a **distributed fashion**,

processes send the state to a **monitor process**

Marzullo and Neiger's algorithm is centralized

- processes send their states to a **process called *monitor***,
  which assembles globally consistent states from what it receive
- consider monitor to lie outside the system, observing its execution

6. Distributed Debugging

Introduction of Distributed Debugging

our aim is to determine cases:

- where a given global state predicate ϕ was ***definitely*** **True at some point** in the execution or
- where ϕ was ***possibly*** **True**

(H : history of the system's execution, L : linearization)

possibly ϕ :

there is a consistent global state S through which a linearization of H passes such that $\phi(S)$ is *True*

definitely ϕ : **from all** linearizations L of H,

there is a consistent global state S through which L passes such that $\phi(S)$ is *True*

6. Distributed Debugging

Introduction of Distributed Debugging

when using snapshot algorithm and obtain the global state S_{snap},

if $\phi(S_{snap})$ happens to be *True*, then we may assert **possibly** ϕ

$\neg possibly\ \phi$: **for all consistent global states** S, $\phi(S)$ evaluates to ***False***

we may conclude $definitely\ (\neg\phi)$ from $\neg possibly\ \phi$,

may not conclude $\neg possibly\ \phi$ from $definitely\ (\neg\phi)$

- (the latter is the assertion that $\neg\phi$ holds at some state on every linearization)

6. Distributed Debugging

Introduction of Distributed Debugging

How the process **states are collected**

How the monitor **extracts consistent global states**

How the monitor **evaluates** $possibly\ \phi$

How the monitor **evaluates** $definitely\ \phi$

Collecting the state

State message

- observed processes p_i **sends their state** from time to time in state messages
- monitor records the state messages from each process p_i in a **separate queue** Q_i
- sending state messages may delay the normal execution, but it does not interfere with it

Optimizations to reduce the state message traffic

- only send **the relevant state** to the monitor
  (because the global state predicate may depend only on certain parts of the processes' states)

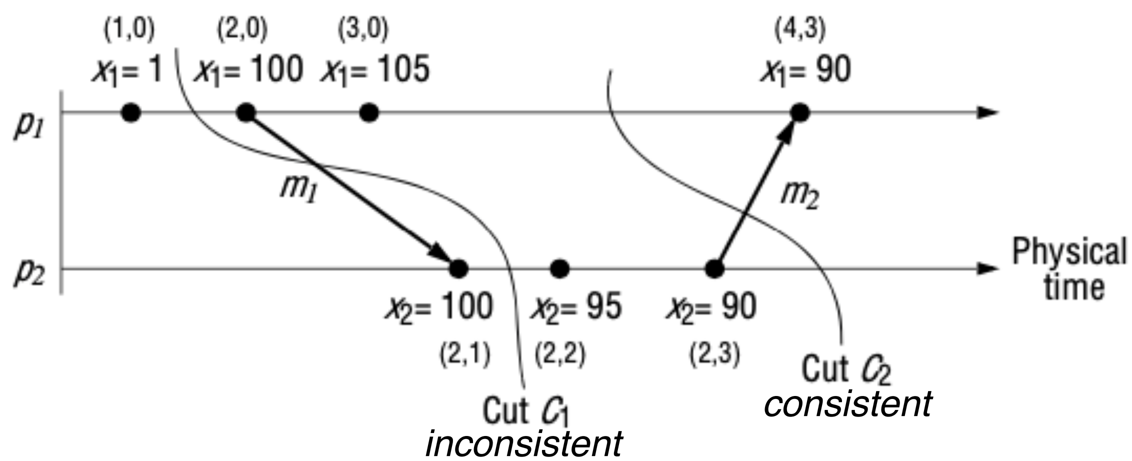
- only send their state at times **when the predicate ϕ may become True or cease to be True**
  (no need to send changes that do not affect the predicate's value)

6. Distributed Debugging

Observing consistent global states

the monitor must assemble consistent global states against which it evaluates ϕ

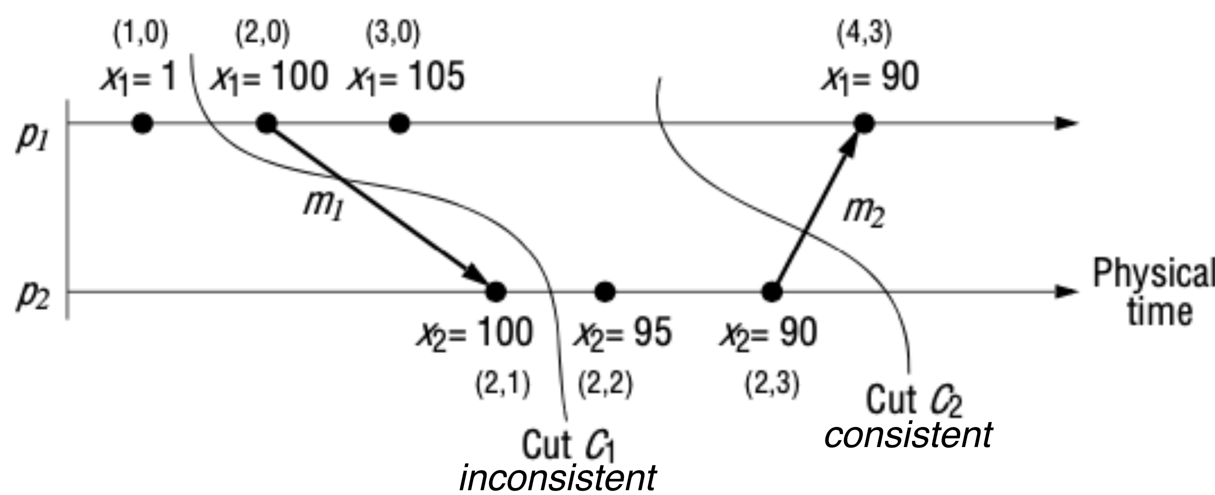
example:



consistent cut
for all events e in the cut C, $f \rightarrow e \Rightarrow f \in C$

- initially, $x_1 = x_2 = 0$
- the requirement is $|x_1 - x_2| \leq 50$

Observing consistent global states



- one of processes **adjust the value** of its variable,
  it sends **the value in a state message** to the monitor

- if monitor uses values in cut $C_1, x_1 = 1, x_2 = 100$, the constraint $|x_1 - x_2| \leq 50$ has broken
  ⇒ this state of affairs **never occurred**

6. Distributed Debugging

Observing consistent global states

Monitor can distinguish consistent global states from inconsistent global states,
by examining **vector clock values** in state messages

$S = (s_1, s_2, \ldots, s_N)$: **global state** drawn from the state messages that the monitor has received

$V(s_i)$: **vector timestamp of the state** s_i received from p_i

$V(s_i)[i] \geq V(s_j)[i]$ (for $i, j = 1,2,\ldots,N$) (condition CGS)

- $V(s_i)[i]$ means the number of p_i's events known at p_i when it sent s_i
- $V(s_j)[i]$ means the number of p_i's events known at p_j when it sent s_j

6. Distributed Debugging

Observing consistent global states

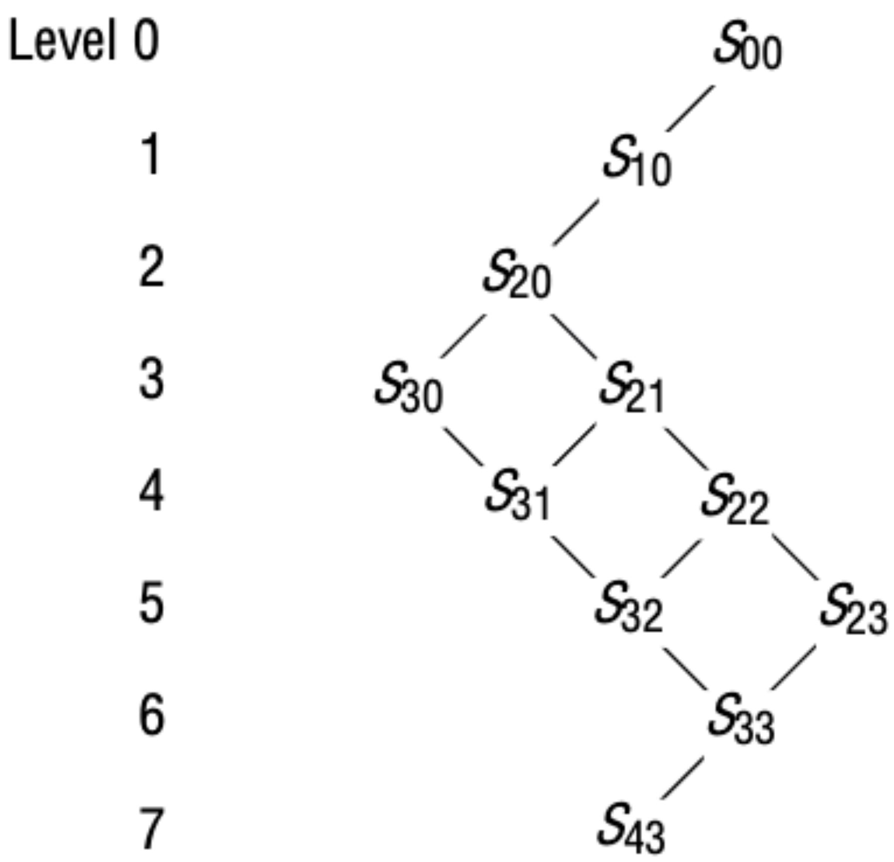
the relation of **reachability**

between consistent global states

- **nodes** : consistent global states
- **edges** : possible transitions between states

- inconsistent global states like S_{01} does not appear
- S_{ij} is in level $(i + j)$
- any global state is reachable from it on the next level

(S_{22} is reachable from S_{20}, not S_{30})



(s_{ij} = global state after e_1^i and e_2^j)

6. Distributed Debugging

Observing consistent global states

Evaluating *possibly* ϕ

: the monitor **traverses all consistent states reachable** from initial state with evaluating $\phi(S)$,

while $\phi(S)$ evaluates *False*

(**stops traversal when $\phi(S)$ evaluates *True***)

Evaluating *definitely* ϕ

: the monitor must attempt to find a set of states through **which all linearizations must pass**,

and at each of **which ϕ evaluates to *True***

Evaluating *possibly* ϕ

the monitor must traverse the lattice of reachable states, starting from the initial state $(s_1^0, \ldots, s_1^N)$

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

$L := 0$;
States := { $(s_1^0, s_2^0, \ldots, s_N^0)$ };
while ($\phi(S) = False$ for all $S \in States$)
  $L := L + 1$;
  Reachable := { S' : S' reachable in H from some $S \in States \wedge level(S') = L$ };
  States := *Reachable*
end while
output "*possibly* ϕ";

$S' = (s_1, \ldots, s_i', \ldots, s_N)$: a consistent state in the next level reachable from $S = (s_1, \ldots, s_N)$

S' is reachable from $S \Longleftrightarrow V(s_j)[j] \geq V(s_i')[j]$ (condition CGS)

the algorithm assumes that the execution is *infinite* (it may be adapted for a finite execution easily)

Evaluating *definitely* ϕ

the monitor must traverse the lattice of reachable states, starting from the initial state $(s_1^0, \ldots, s_1^N)$

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

$L := 0;$
if $(\phi(s_1^0, s_2^0, \ldots, s_N^0))$ *then States* $:= \{\}$ *else States* $:= \{ (s_1^0, s_2^0, \ldots, s_N^0) \};$
while (*States* $\neq \{\}$)
    $L := L + 1;$
    Reachable $:= \{ S' : S'$ reachable in H from some $S \in States \ \wedge \ level(S') = L \};$
    States $:= \{ S \in Reachable : \phi(S) = False \}$
end while
output "*definitely* ϕ";

States : a set which contains those **states at the current level**

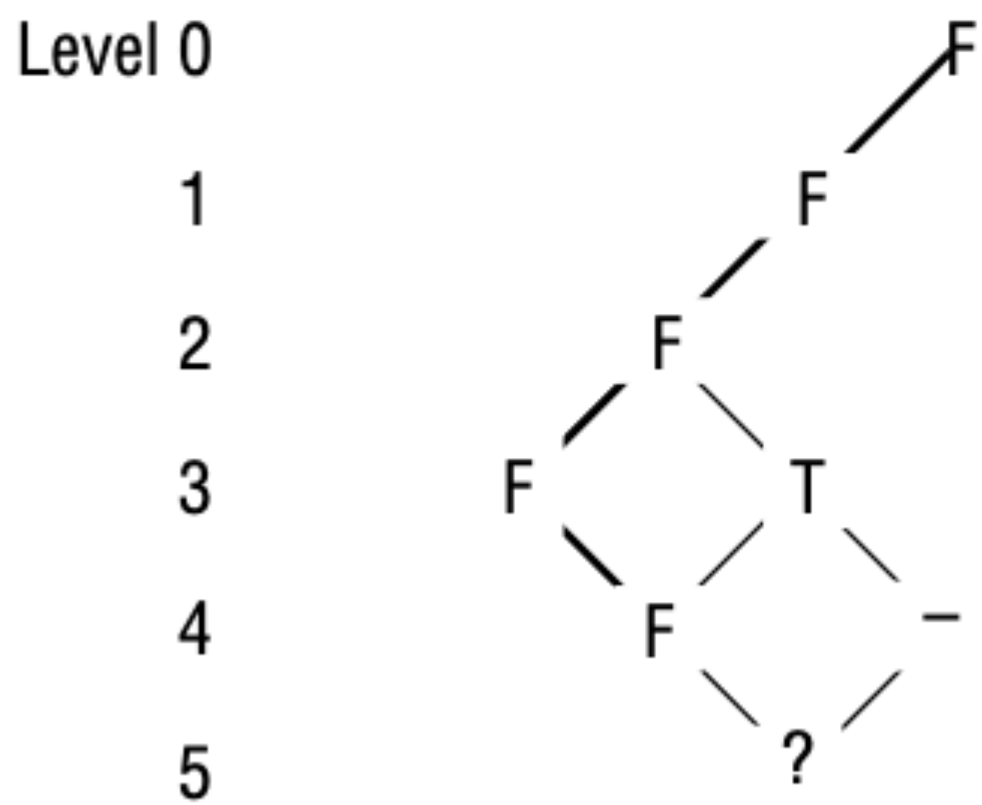
that may be reached on a linearization from the initial state

by **traversing only states for which** ϕ **evaluates to *False***

Evaluating *definitely* ϕ

at level 3, the set *States* consists of only one state which is marked in bold lines

if ϕ evaluates to True in the state at level 5, we may conclude *definitely* ϕ



Evaluating *definitely* ϕ - cost

N : the number of observed processes

k : the maximum number of events at a single process

the algorithms entail $O(k^N)$ **comparison**s

space cost : $O(kN)$

- but the monitor may delete a message containing s_i from queue Q_i

  when no other item of state arriving from another process could possibly be involved in a consistent global state containing s_i

- when $V(s_j^{last})[i] > V(s_i)[i]$ for $j = 1,2,\ldots,N, j \neq i$

  (s_{last} : the last state that the monitor has received from p_j)

6. Distributed Debugging

Evaluating *possibly* ϕ and *definitely* ϕ in synchronous systems

the algorithms work in an **asynchronous system** (no timing assumptions)

- monitor may **examine a consistent global state** S, for which any two local states s_i, s_j occurred an arbitrarily long time apart in the actual execution

In **synchronous system**, suppose that:

- processes' **physical clocks are internally synchronized** within a known bound
- processes provide **physical timestamps** and **vector timestamps**

then, monitor need **consider only consistent global states** (existed simultaneously)

- with good enough clock synchronization, these will number many less then all globally consistent states

6. Distributed Debugging

Evaluating *possibly* ϕ and *definitely* ϕ in synchronous systems

p_i $(i = 1, \ldots, N)$: the observed processes

p_0 : the monitor

C_i $(i = 0, \ldots, N)$: physical clocks

these are synchronized to with in a known bound $D > 0$:

$$|C_i(t) - C_j(t)| < D \; for \; i, j = 0, \ldots, N$$

6. Distributed Debugging

Evaluating *possibly* ϕ and *definitely* ϕ in synchronous systems

the observed processes **send their vector time and physical time** to the monitor

the monitor now applies a condition that **not only tests for consistency of global state** S, but also tests whether each pair of states could have **happened at the same real time**

$V(s_i)[i] \geq V(s_j)[i]$ *and* s_i, s_j *could have occurred at the same real time*

s

Evaluating *possibly* ϕ and *definitely* ϕ in synchronous systems

$L_i(s_i)$: local time when the next state transition occurs at p_i

- p_i is in the state s_i from $C_i(s_i)$ to $L_i(s_i)$

for s_i and s_j to have obtained at the same real time:

$C_i(s_i) - D \leq C_j(s_j) \leq L_i(s_i) + D$ or vice versa(swapping i, j)

the monitor must calculate a value $L_i(s_i)$

- when the monitor has received a state message for p_i 's next state s'_i, then $L_i(s_i) = C_i(s'_i)$
- if otherwise, the monitor estimates $L_i(s_i)$ as $C_0 - max + D$
  - (C_0 : the monitor's current local clock)
  - (max : the maximum transmission time for a state message)

End