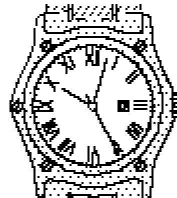
JPU جامعة الجزيرة الخاصة
ALJAZEERA PRIVATE UNIVERSITY

Time



Global States

Time & Global State

- Introduction (Time in Distributed Sys)
- **Physical Clocks**
- Why to synchronize multiple clocks in DS?
- Physical Clock Synchronization
- NTP (Network Time Protocol)
- **Logical Clocks**
- **Distributed Debugging**
- **Global States**
- **Logical Time & Global States**

Introduction (1)

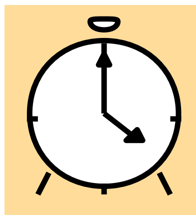
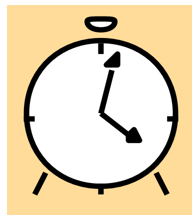
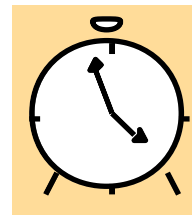
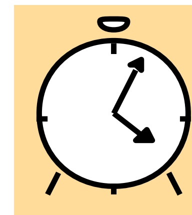
- Time is to order events.
- DS = N processes **&** Communication by messages **&** No Shared memory
- Process P_i State $\mathbf{S_i}$ = (its variable values + its objects in OS)
- Events in DS = **send**(m), **receive**(m), **internal event**.
- No Global Time (problem in DS)

Introduction (2)

- One process (one physical clock) even multi-threaded ➔
  - single, total order for events : $e \longrightarrow_i é$
- History $(p_i) = h_i = <e_i^0, e_i^1, e_i^2, \ldots>$
- No Global Time (problem in DS) = we can not accurately take the event times. So, we do not know the order of events.
- Two different observers may not agree on the interval between 2 events
- Two different observers may see the events in reversed order (Impossible if there is **causality** relation between the 2 events)

Physical Clocks

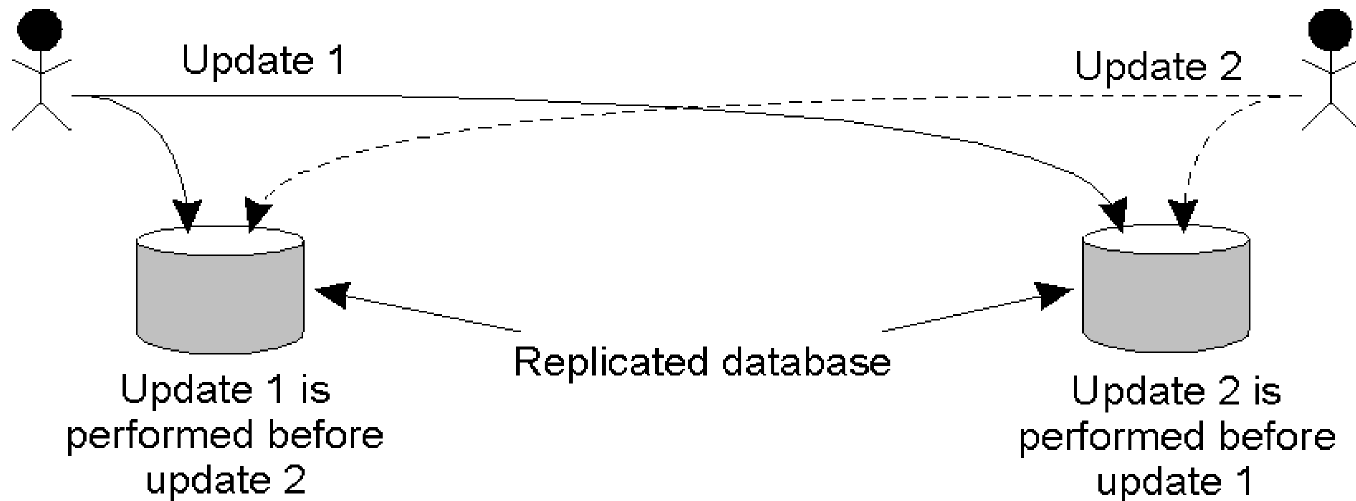
- In DS, each computer has its own clock.
  - Clock = A high-frequency **oscillator** (quartz crystal kept under tension) , **counter**, and **holding register**
- Clock **drift** = crystals run at different frequencies.
- Clock **skew** = the difference between two reads for two clocks **at the same time**.

Network

Why to synchronize multiple clocks in DS?

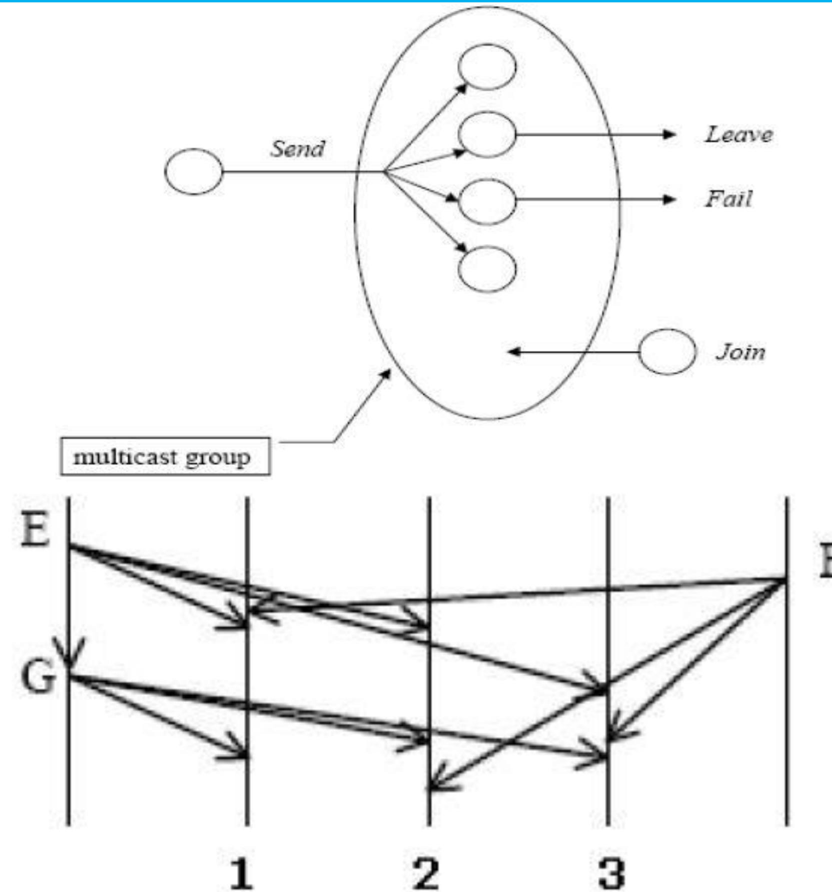
(Ex : Replicated Bank System)



- Balance = $1000
- Update 1 = add($100)
- Update 2 = add(1%)
- ➔ balance 1 = 1110$ && balance 2 = 1111$

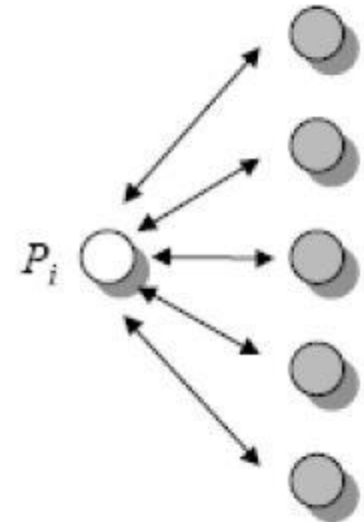
Why to synchronize multiple clocks in DS?

(Ex : Group Communication)

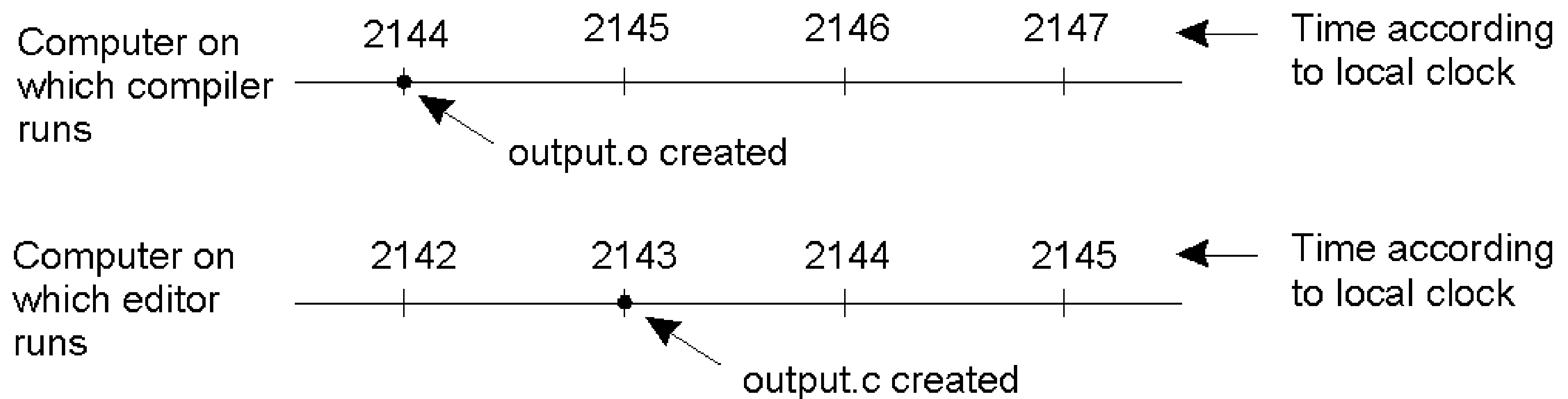


DS?
(Ex : Distributed Critical Section Control)

- **Centralized solution** = Process P_i sends to coordinator
  - If critical section is empty ➔ reply = ok
  - Else, request to queue && when queue is empty send reply
  - When process finishes, it sends Release() to coordinator.
  - **Pb** = coordinator dies
- **Decentralized solution** : Process P_i sends to all other p waits for N replies. ➔ Reply =
  - I don't care (Process doesn't want to enter)
  - I am before him
  - I am after him
  - **Pb** = one process dies



Why to synchronize multiple clocks in DS?
(Ex : Make-file Tool)



- When each machine has its own clock, an event that occurred after another event may be assigned an earlier time.
- Accurate time is needed for some application domains.

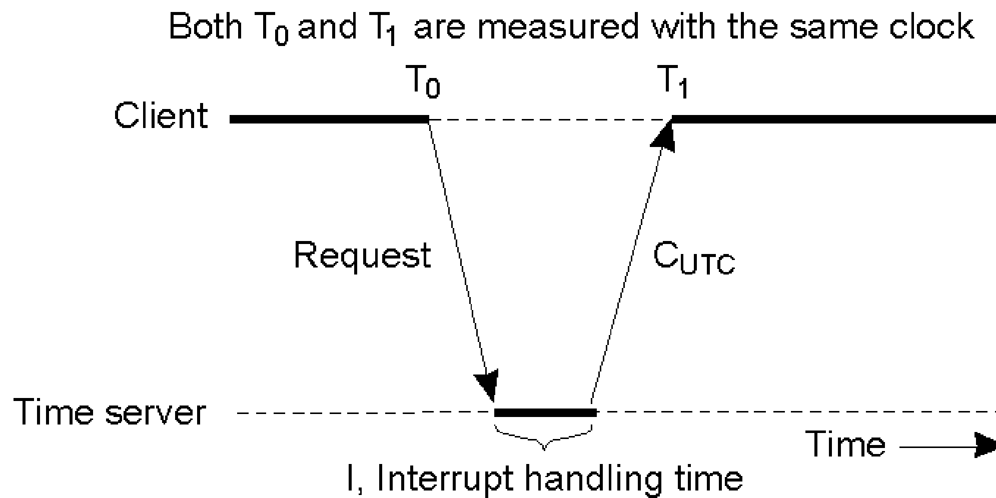
Why to synchronize multiple clocks in DS?
(Some other examples)

- Ex 1 = Replicated Bank System
- Ex 2 = Group Comm.
- Ex 3 = Make-file Tool
- Ex 4 = Distributed Critical Section
- Ex 5 = Distributed Shared Memory (Replicated Data + Group Communication).
- Ex 6 = Transaction Serialization.
- Ex 7 = Distributed Debugging
- Ex 8 = ….

Physical Clock Synchronization

- Centralized / Decentralized algorithms
  - Centralized ➔ Cristian & Berekely Algorithms
  - Decentralized ➔ NTP
- External Synchronization
  - ➔ $| S(t) - C_i(t) | < D$
  - Internet External Time Resource : UTC (Coordinated Universal Time)
    - By GPS → 1µ sec
    - By Radio stations → 0.1-10 m sec
    - By Telephone lines → several m sec
- Internal Synchronization
  - ➔ $| C_i(t) - C_j(t) | < D$

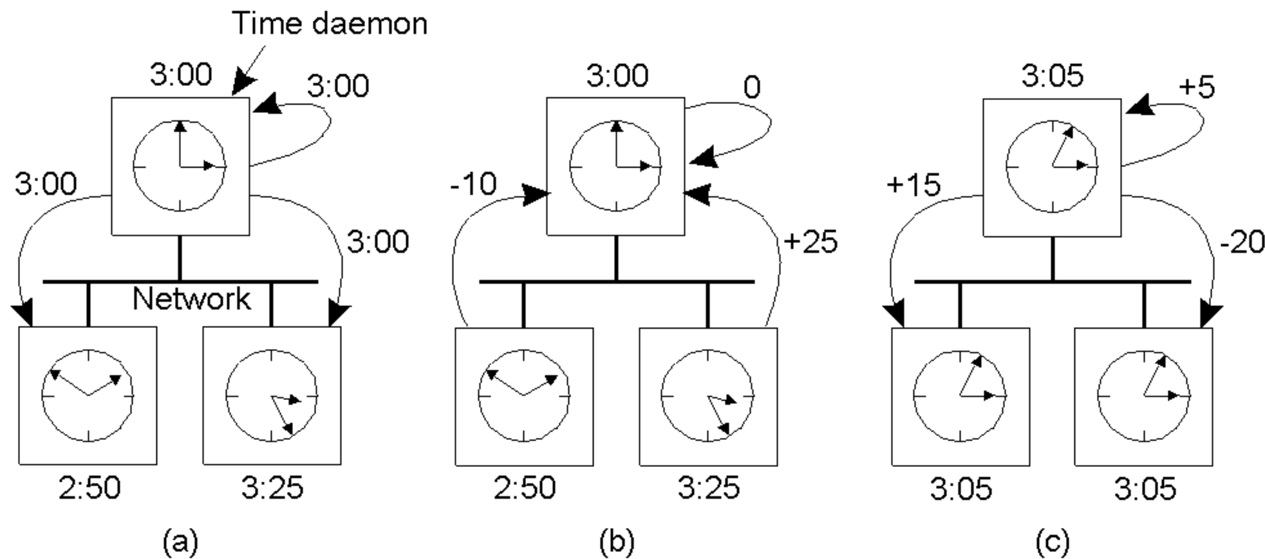
Cristian's method



- time server is passive.
- Periodically, client time = C_{UTC} + (T1 – T0 - I) / 2
- Pb : sender's clock is fast ➔ C_{UTC} < sender's clock (but time must not run backward and changes must introduced gradually)

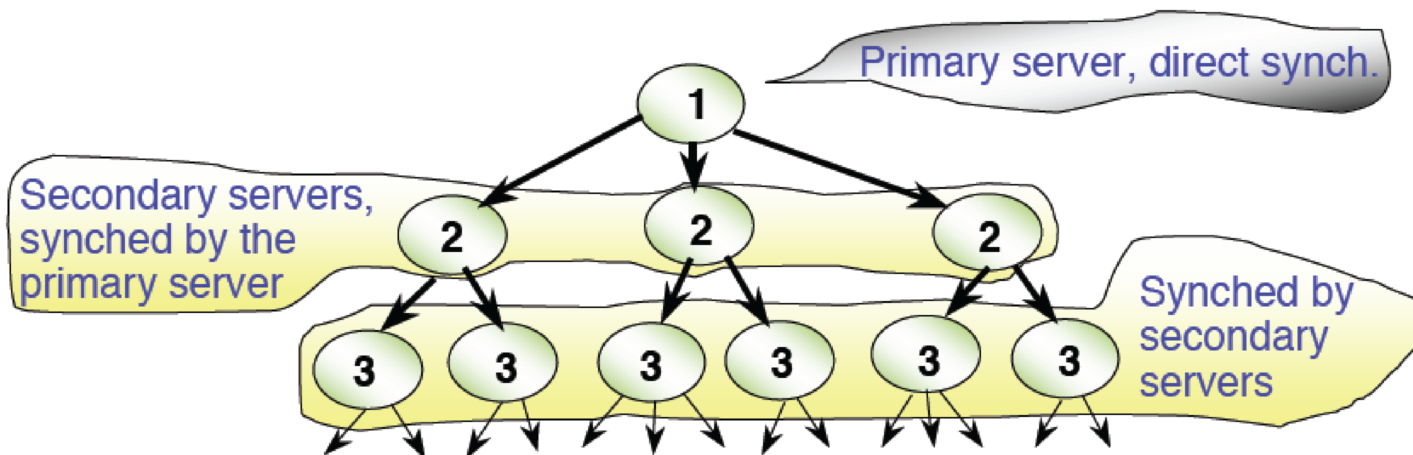
Berkeley Algorithm

- The time daemon asks all the other machines for their clock values (+ Election when daemon failure)
- The machines answer
- The time daemon tells evervone how to adiust their clock



NTP (Network Time Protocol)

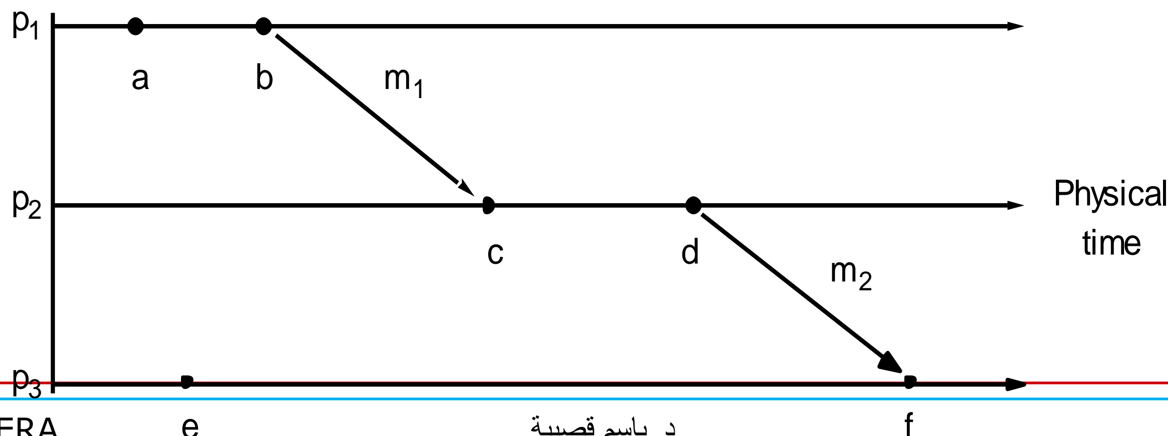
- NTP provides a service enabling clients across the Internet to be synchronized accurately using one of UTC sources.
- Uses a network of time servers to synchronize all processors on a network.
- Time servers are connected by a synchronization subnet tree. The root is adjusted directly . Each node synchronizes its children nodes.



Logical Time

Logical Time

- We can not synchronize perfectly physical clocks. So, we can not give absolute time to events.
- So, we use physical causality (happened-before relationship) to order events (between different processes) :
  - On the same process : **time(a)** < **time(b).** Then, **a** ⟶ **b**
  - **P1** sends **m** to **p2**. Then, **send(m)** ⟶ **receive(m)**
  - (Transitivity) if **a** ⟶ **b** && **b** ⟶ **c**. Then, **a** ⟶ **c**

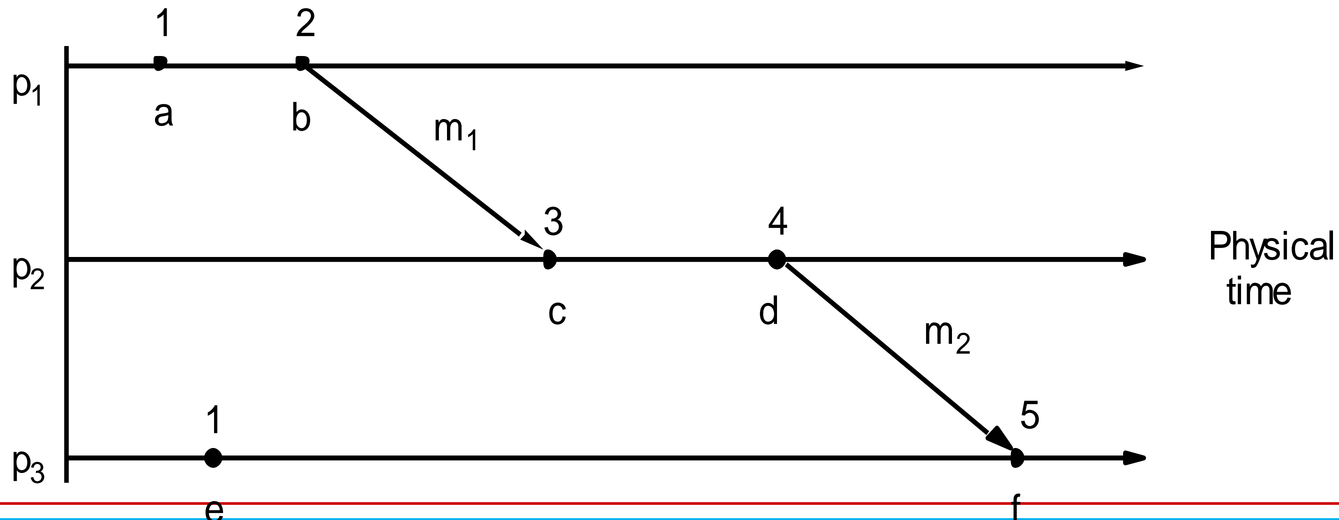


Lamport Method

Logical Clocks (Lamport Method)

- Each process P_i has a logical clock L_i. Initialized $L_i = 0$.
- **R1** : L_i is incremented before each event issued at P_i : $L_i = L_i + 1$
- **R2** : A) when P_i sends message **m**. It add to **m** the value $t = L_i$

  B) the receiving process P_j calculates $L_j = \max(L_j, t)$ and apply **R1** before timestamp the **receive(m)** event.



Lamport Method Consequences

- If **a** $\longrightarrow$ **b**. Then, **L(a)** < **L(b).**
- But, if **L(x)** < **L(y)**. Then, we can not infer that **x** $\longrightarrow$ **y**
- Not all events can be ordered by the $\longrightarrow$ relation
- **a** || **e** are concurrent if **a** $\not\longrightarrow$ **e** and **e** $\not\longrightarrow$ **a**.
  - EX: a and e = 2 events at 2 different processes and there is no messages between these processes

Total order of Lamport clocks

- Different events at different processes can have the same timestamps
- Event total ordering by taking into account the process ID
  - $(T_i, i) < (T_j, j)$ *if* $\underline{T_i < T_j}$ *or* $\underline{T_i == T_j \;\&\&\; I < J}$
  - This order (without physical significance)
  - Sometimes, (useful)
    - EX: to order the process entry to a critical section

Vector Clock

Vector Clock

- Vector clock = solution of Lamport problems
- Vector clock is an array of **N** integer for **N** processes
- Processes use vector clock to timestamp its events.
- processes add vector clock on the sent messages.

Vector Clock Algorithm

- Each process $\mathbf{P_i}$ has its vector clock $\mathbf{V_i}$.
  - Initially, $V_i[j] = 0$ for $j = 1,2, \ldots , N$
- **R1** : $\mathbf{V_i[i]}$ is incremented before each event issued at $\mathbf{P_i}$ **:**

$$\mathbf{V_i[i]= V_i[i] + 1}$$

- **R2** :
  - **A**) when $\mathbf{P_i}$ sends message **m**. It adds the value $\mathbf{t = V_i}$
  - **B**) the receiving process $\mathbf{P_j}$ calculates its $\mathbf{V_j}$ as :

    $V_j[j] = \mathbf{max}\ (V_j[j]\ \mathbf{,}\ t[j])$ for $j = 1,2, \ldots, N$
  - **C**) apply **R1** before timestamp the **receive(m)** event.

Vector Clock Algorithm

- $V_i[i]$: Nb of events that P_i has timestamped
- $V_i[j]$ ($j \neq i$) : Nb of events occurred at P_j and P_i was affected by them.

(1,0,0) (2,0,0)
p_1 a b
m_1
(2,1,0) (2,2,0)
p_2 c d
Physical time
m_2
(0,0,1) (2,2,2)
p_3 e f

Example: Vector Logical Time

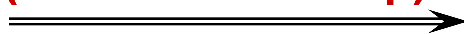
Physical Time

p 1	0,0,0,0	1,0,0,0	2,0,0,0	3,0,2,2	4,0,2,2	
p 2	0,0,0,0	1,1,0,0	1,2,0,0			
p 3	0,0,0,0	2,0,1,0	2,0,2,0	2,2,3,0	4,2,4,2	4,2,5,3
p 4	0,0,0,0	2,0,2,1	2,0,2,2	2,0,2,3		

Messages (vector timestamps): (1,0,0,0), (2,0,0,0), (1,2,0,0), (2,0,2,0), (2,0,2,2), (4,0,2,2), (2,0,2,3)

n,m,p,q — Vector logical clock

(vector timestamp)


Message

Rules for Vector Clock

- **V1 = V2** iff $V1[j] = V2[j]$ for $j = 1, 2, 3, \ldots, N$
- **V1 ≤ V2** iff $V1[j] \leq V2[j]$ for $j = 1, 2, 3, \ldots, N$
- **V1 < V2** iff $V1 \leq V2$ && $V1 \neq V2$

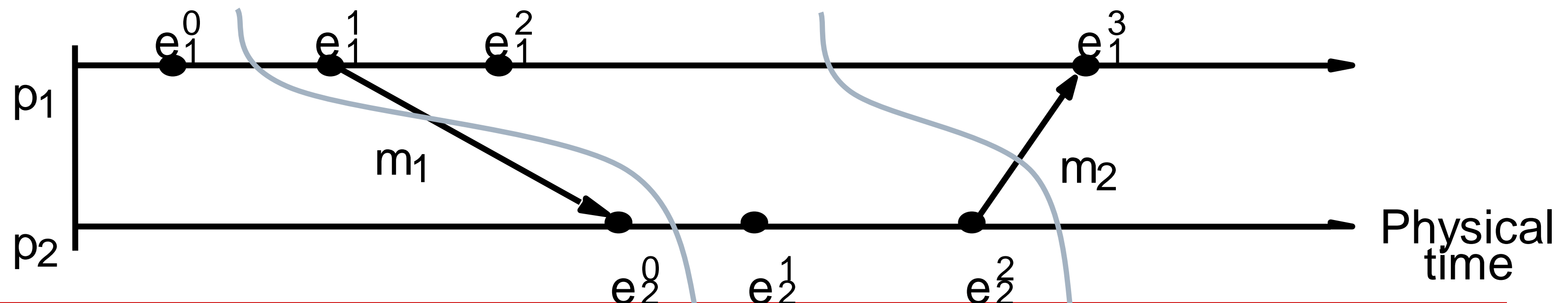
Vector Clock Consequences

- $a \longrightarrow b \quad \Rightarrow \quad V(a) < V(b)$
- $V(a) < V(b) \quad \Rightarrow \quad a \longrightarrow b$
- $a \parallel b$ **if** neither $V(a) \not\leq V(b)$ nor $V(b) \not\leq V(a)$

Consistent Cuts

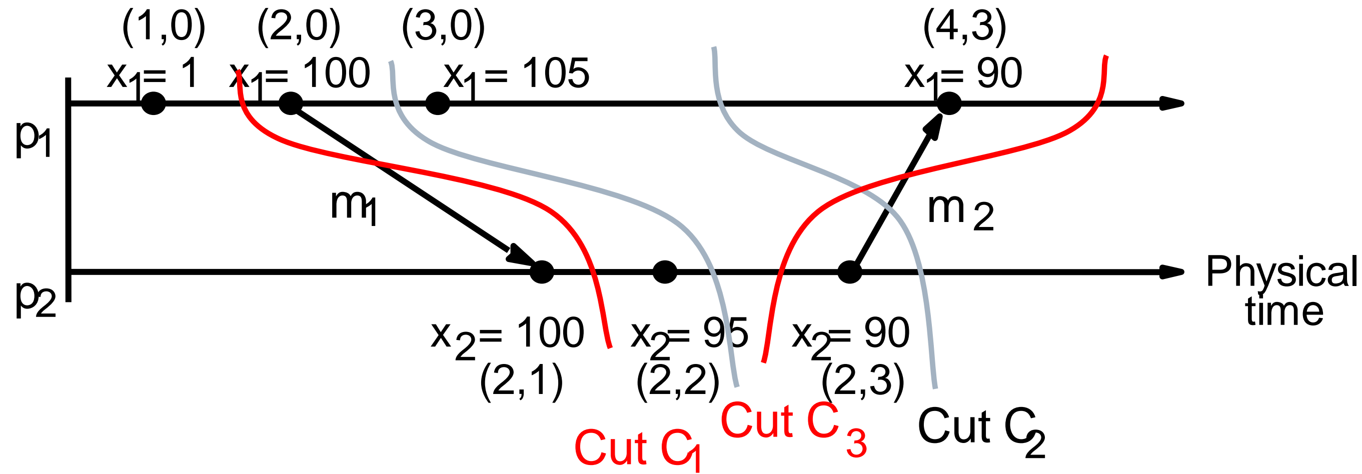
Global states && Consistent cuts

- If all processes had perfectly synchronized clocks ➔ we could agree a time at witch processes record their local states.
- **Cut** = method to assemble a meaningful global state from local states recorded **at different real times.**
- Consistent cut = for each event it contains, it also contains all the events that happened-before that event.
- Consistent Global State = recorded states before a consistent cut.
- Distributed system execution = transitions between system global states



Distributed Debugging
(Logical Time & Global States)

P1	P2
X1 = 1 (1,0)	(2,1) X2 = 100
(2,0) X1 = 100	X2 = 95 (2,2)
(3,0) X1 = 105	X2 = 95 (2,2)
x1 = 90 (4,3)	X2 = 95 (2, 2)
(3,0) X1= 105	X2 = 90 (2, 3)

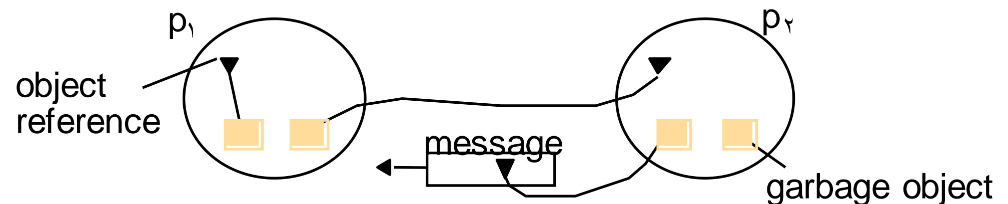


- إن كل فعالية تبعث إلى المراقب حالة متحولاتها المحلية + شعاعها الزمني
- المراقب يحترم بأن تكون عدد أحداث Pj و المعروفة لـ Pi عندما تم إرسال حالة Pj ليست بأكبر من تلك الحاصلة لدى Pi عندما أرسلت حالتها

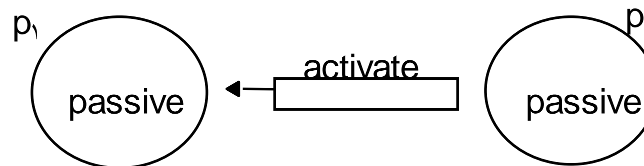
Global States

Global States

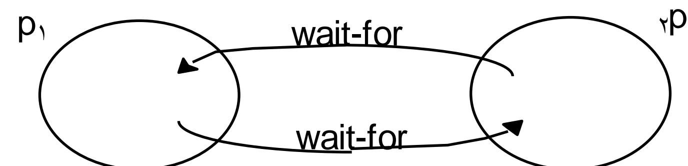
- Global state = assembly of processes local states + communication channel states (<u>in absence of global time</u>)
- Global states allow us to verify a property in Distributed System Execution like :
  - Garbage Collection



  - Termination

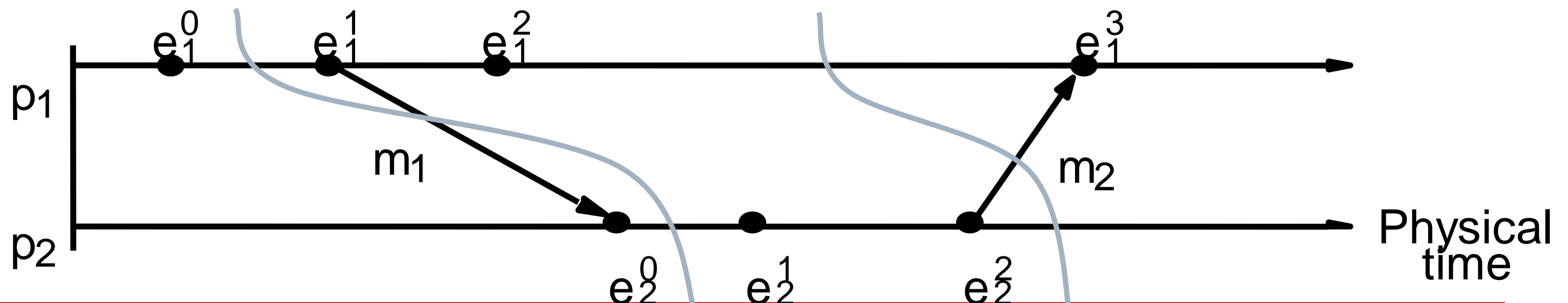


  - Deadlock

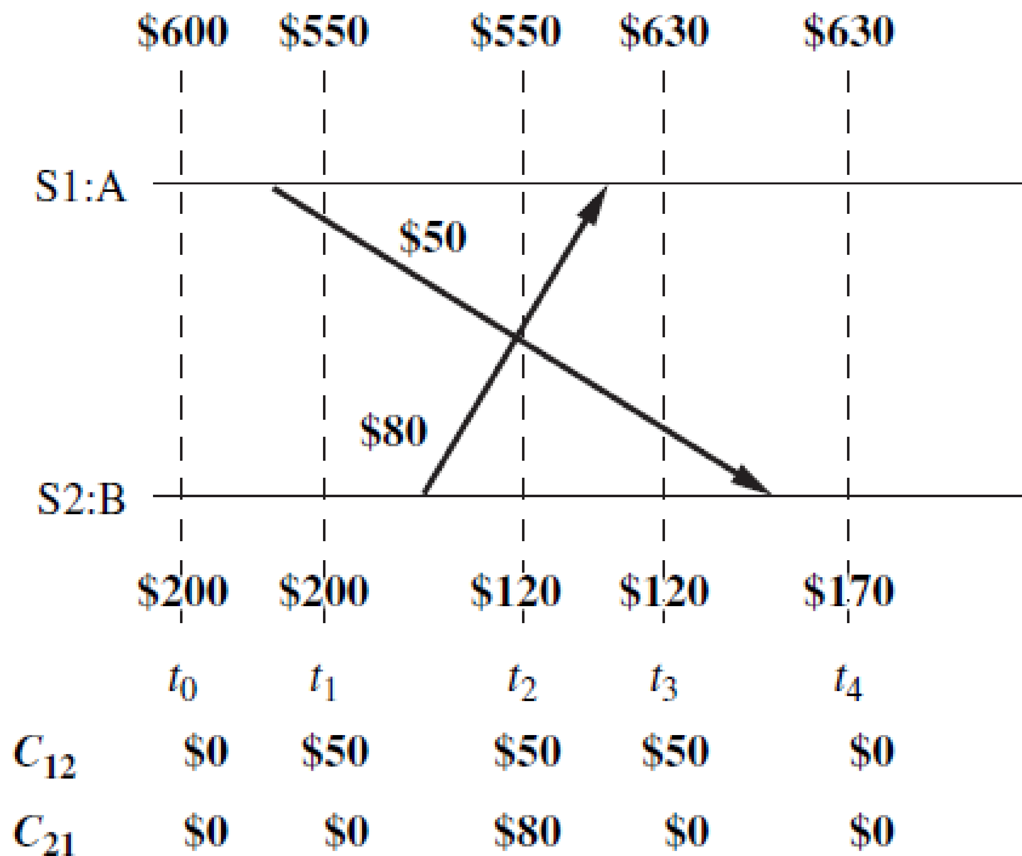


Consistent cuts

- If all processes had perfectly synchronized clocks ➔ we could agree a time at witch processes record their local states.
- **Cut** = method to assemble a meaningful global state from local states recorded **at different real times.**
- Consistent cut = for each event it contains, it also contains all the events that happened-before that event.
- Consistent Global State = recorded states before a consistent cut.
- Distributed system execution = transitions between system global states

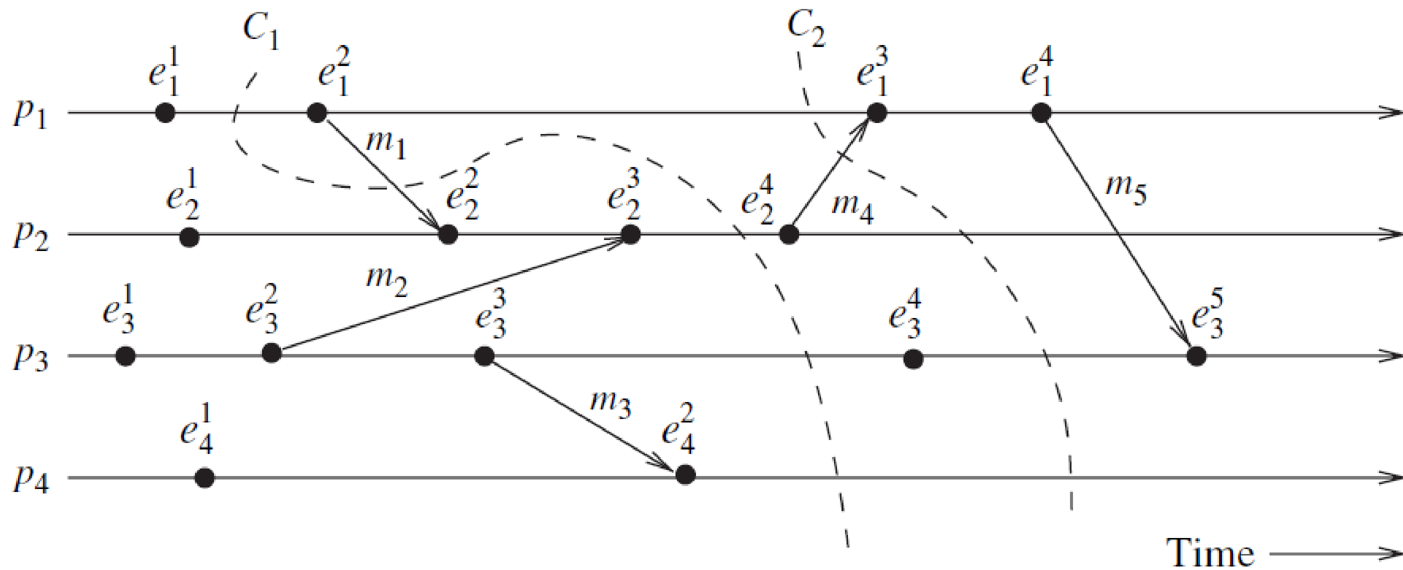


Ex : Banking Sys



Recording Consistent State (Consistent Cut)

- C1 = *send(mij)∈LSi*
  - ⇒ mij ∈ Scij ⊕ *rec(mij)∈ LSj*
- C2 = *send(mij)∉Lsi* ⇒ mij ∉Scij ∧ re*c(mij)∉LSj*

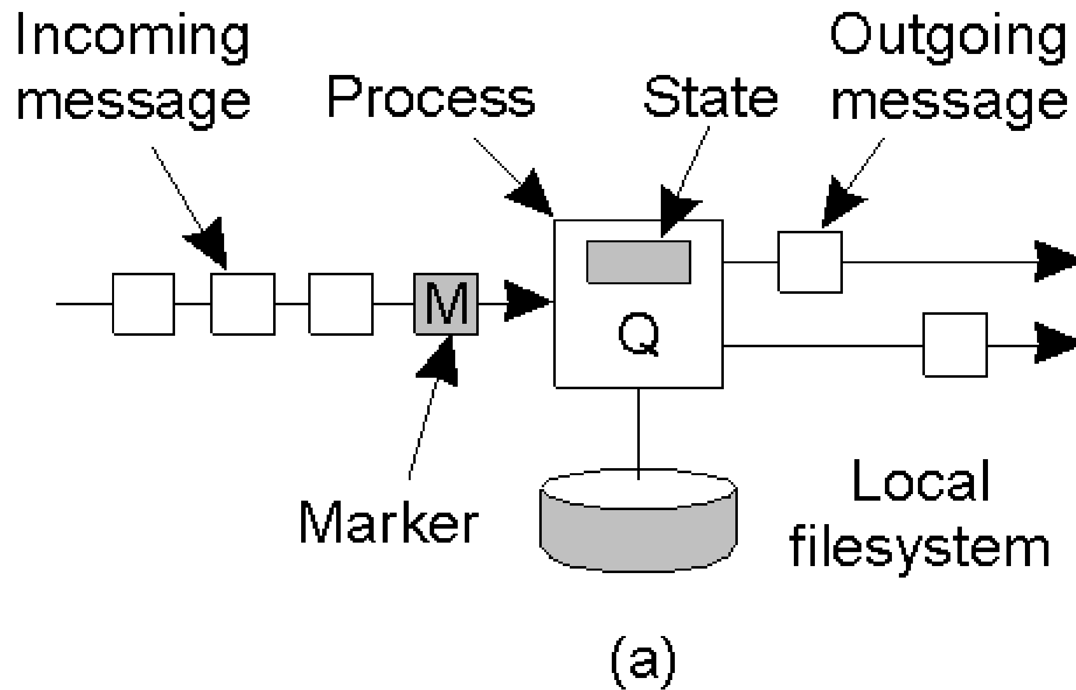


Snapshot Algorithm Bases to Recording Consistent State

- **B1** = Which messages to/or not to be recorded in the snapshot (in a channel state or in a process state)
  - Any message that is sent by a process before recording its snapshot must be recorded
  - Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot
- **B2** = How to determine when a process takes its snapshot
  - A process **pj** must record its snapshot before processing a message **mij** that was sent by process **pi** after recording its snapshot

"snapshot" algorithm

- Any process may initiate the algorithm
- There is a path between any two processes



"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***

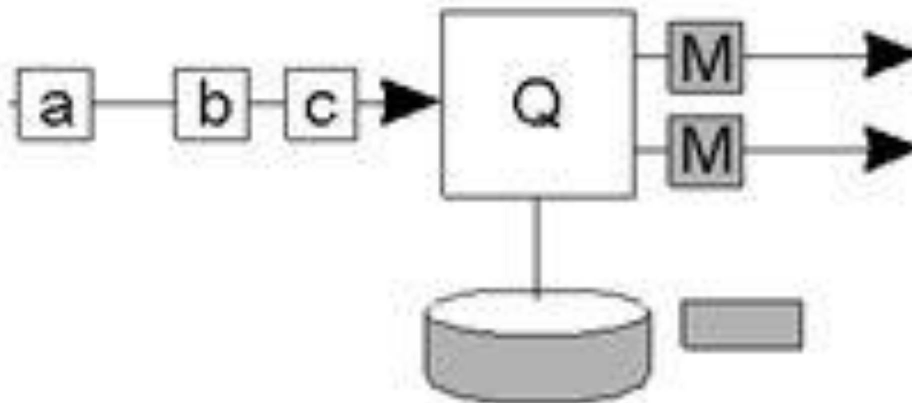
  After p_i has recorded its state, for each outgoing channel *c*:

  p_i sends one marker message over *c*

  (before it sends any other message over *c*).

“snapshot” algorithm (step 1)

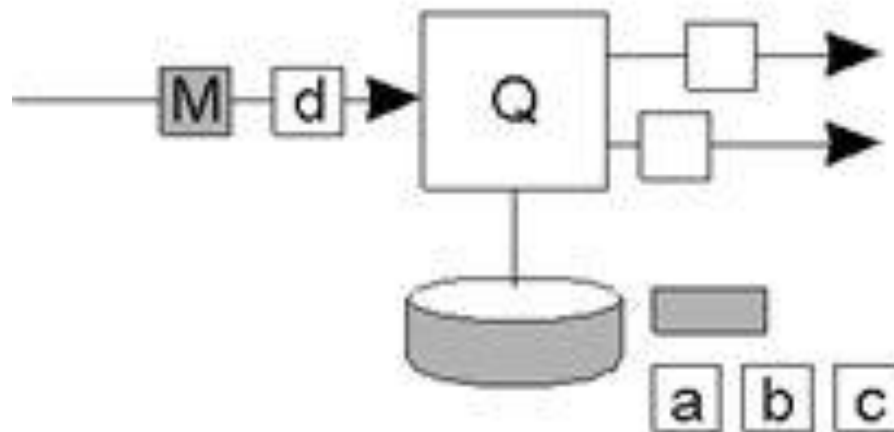
- Processes can continue their execution and send / receive messages while the algorithm runs.



(b)

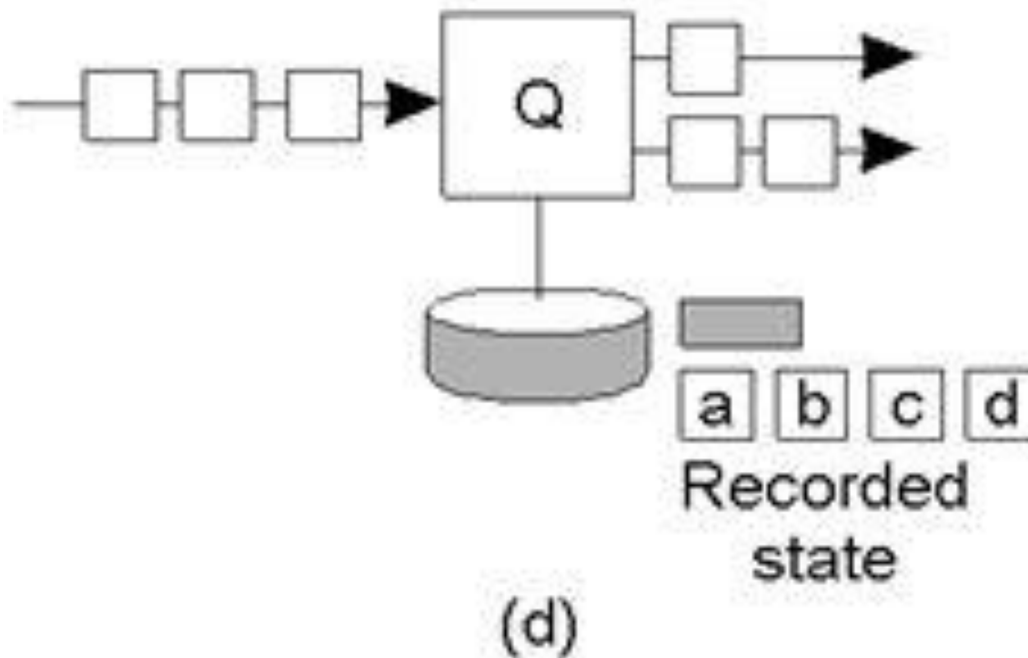
“snapshot” algorithm (step 2)

- while any process can initiate the algorithm (multiple executions). So, we need ID for M

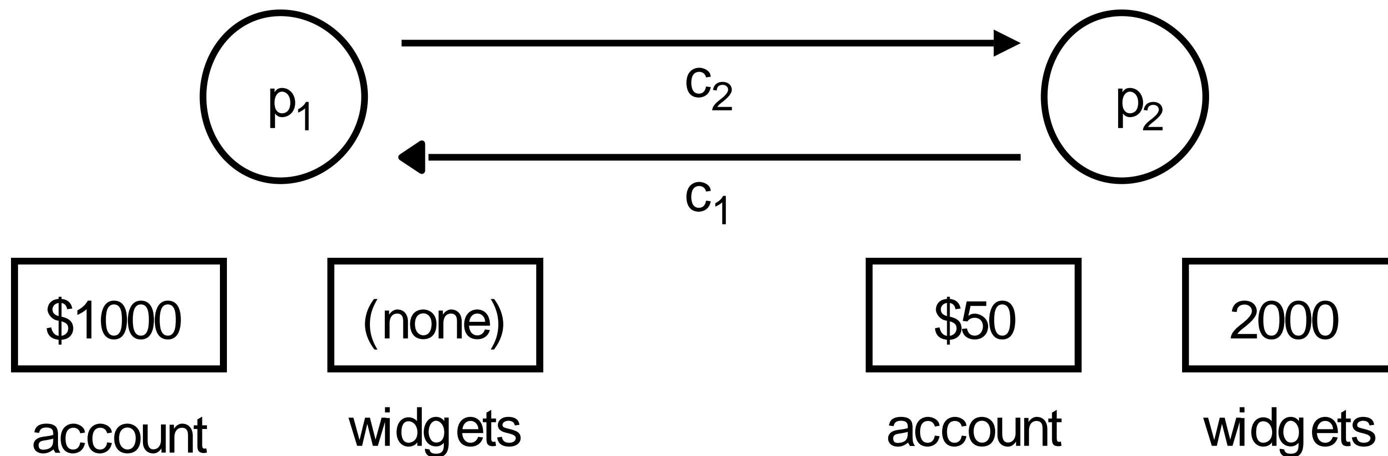


“snapshot” algorithm (step 3)

- Process finishes when it receives M on all its incoming channels



“snapshot” algorithm - Example



“snapshot” algorithm - Example

1. Global state S_0

<$1000, 0> p_1 — c_2 (empty) — p_2 <$50, 2000>

c_1 (empty)

2. Global state S_1

<$900, 0> p_1 — c_2 (Order 10, $100), M — p_2 <$50, 2000>

c_1 (empty)

3. Global state S_2

<$900, 0> p_1 — c_2 (Order 10, $100), M — p_2 <$50, 1995>

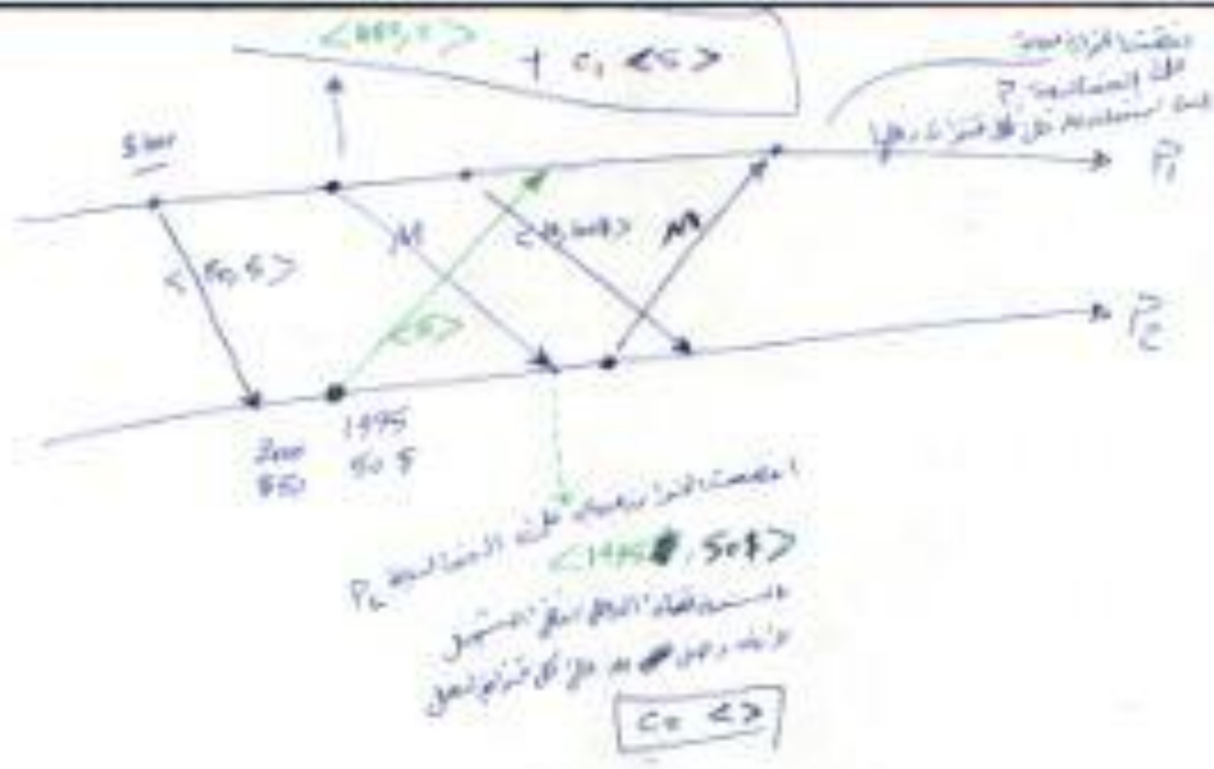
c_1 (five widgets)

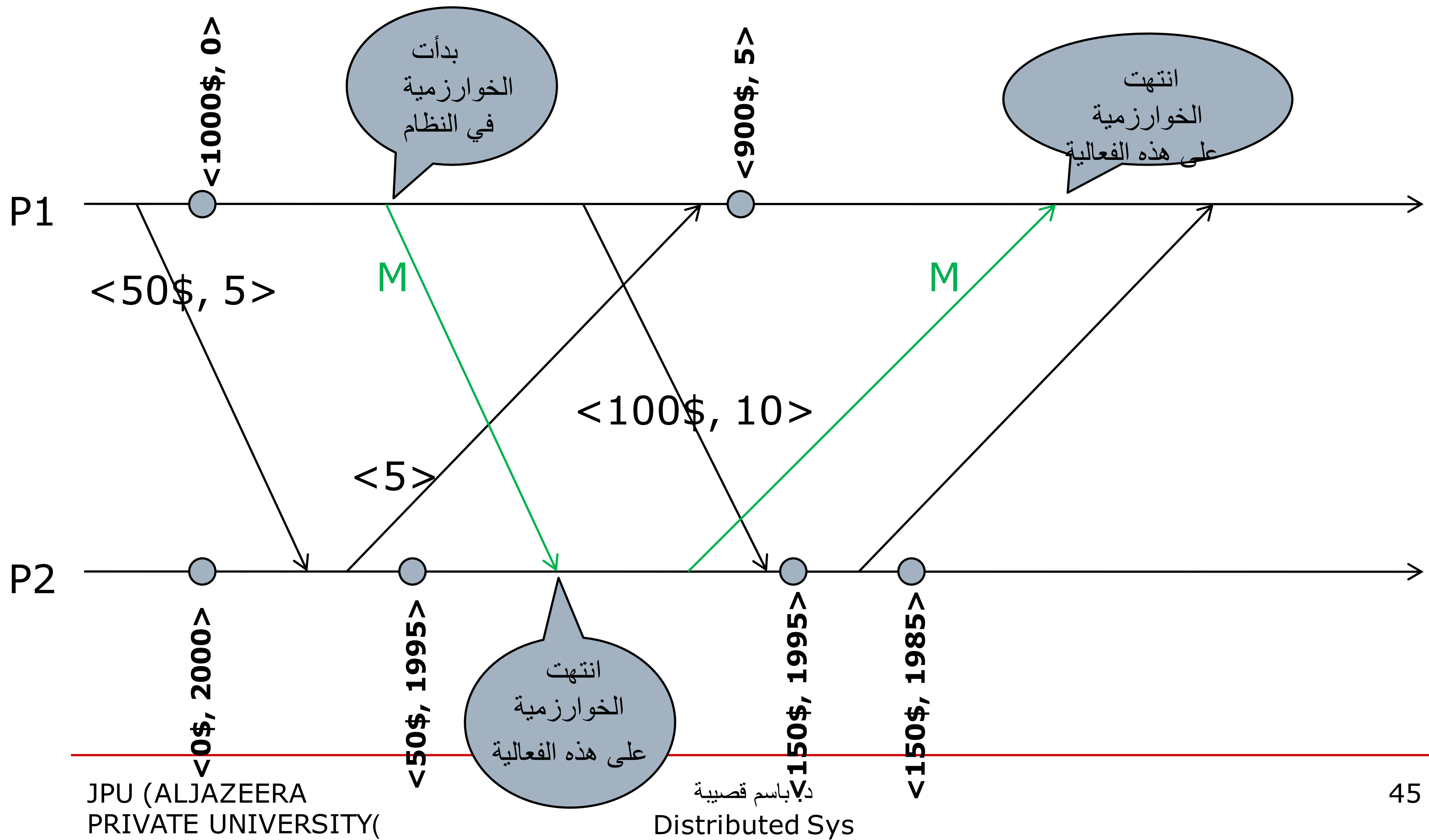
4. Global state S_3

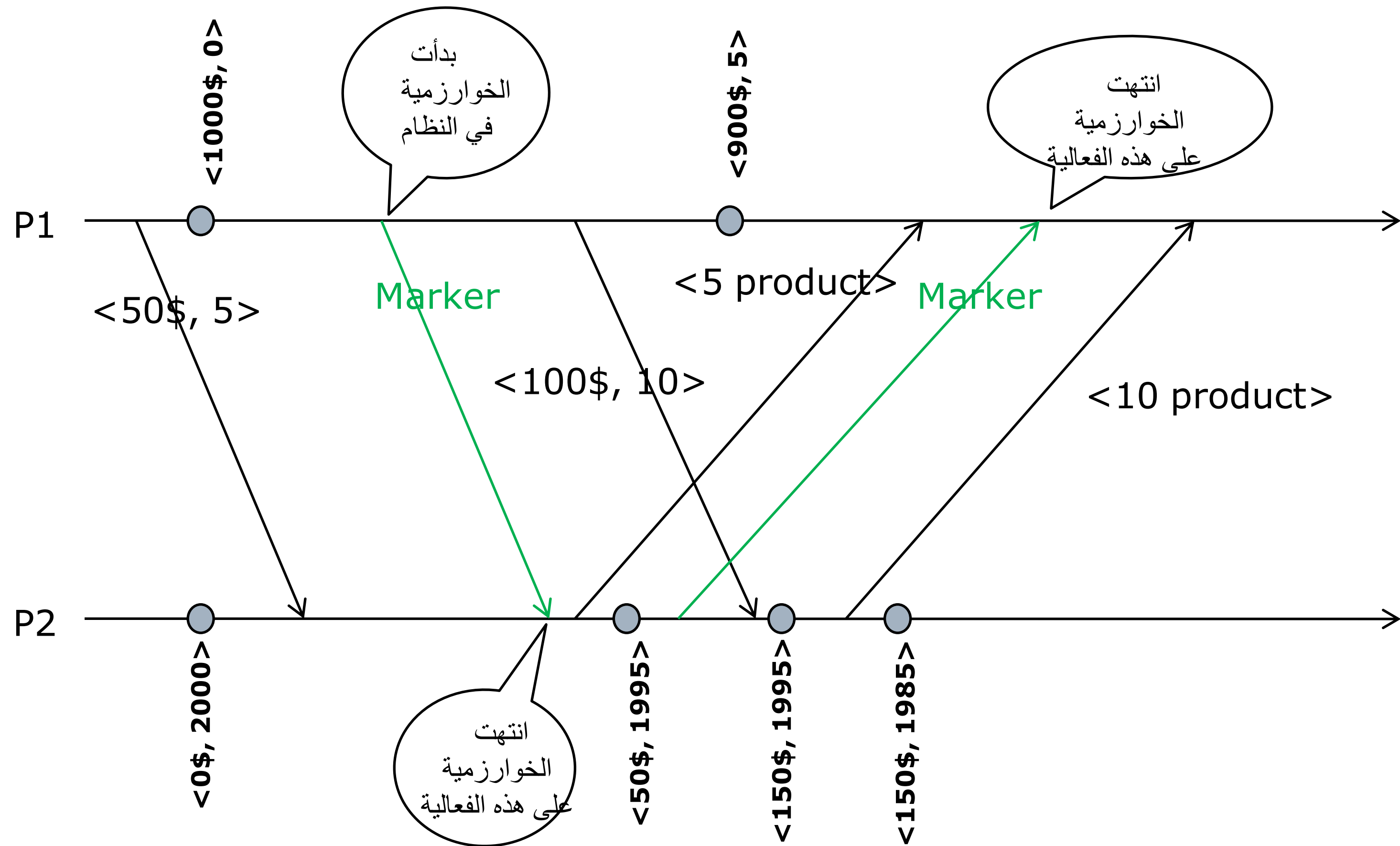
<$900, 5> p_1 — c_2 (Order 10, $100) — p_2 <$50, 1995>

c_1 (empty)

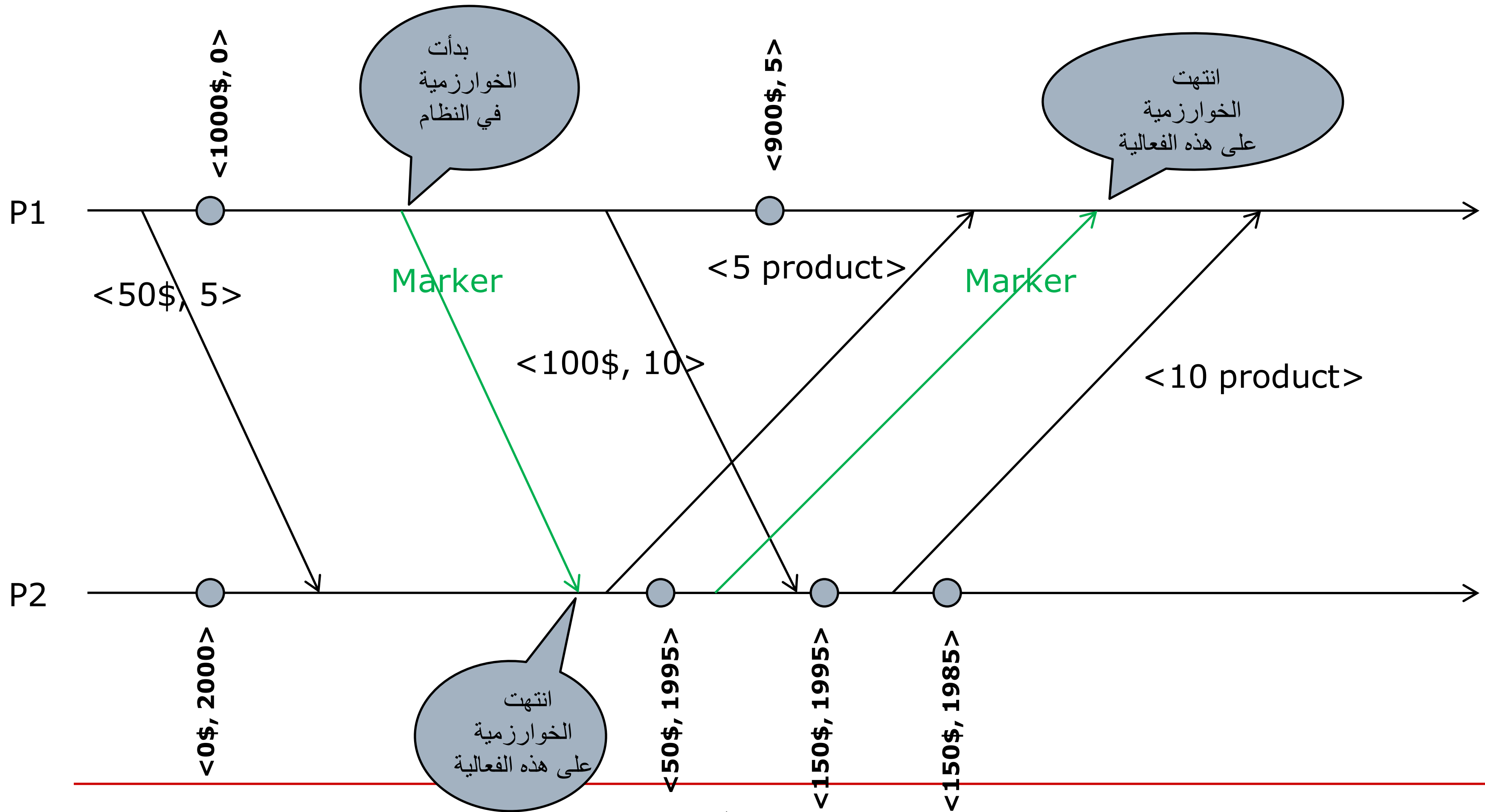
‘snap’ Global State = P1 <$1000, 0>, P2 <$50, 1995>, C1 <5>, C2 <>


P_1
P_2
M
M


<1000$, 0>
بدأت الخوارزمية في النظام
<900$, 5>
انتهت الخوارزمية على هذه الفعالية
P1
<50$, 5>
M
M
<100$, 10>
<5>
P2
<0$, 2000>
<50$, 1995>
انتهت الخوارزمية على هذه الفعالية
<150$, 1995>
<150$, 1985>



P1
P2
<1000$, 0>
بدأت الخوارزمية في النظام
<900$, 5>
انتهت الخوارزمية على هذه الفعالية
<50$, 5>
Marker
<5 product>
Marker
<100$, 10>
<10 product>
<0$, 2000>
انتهت الخوارزمية على هذه الفعالية
<50$, 1995>
<150$, 1995>
<150$, 1985>



P1
P2
<1000$, 0>
بدأت
الخوارزمية
في النظام
<900$, 5>
انتهت
الخوارزمية
على هذه الفعالية
<50$, 5>
Marker
<5 product>
Marker
<100$, 10>
<10 product>
<0$, 2000>
انتهت
الخوارزمية
على هذه الفعالية
<50$, 1995>
<150$, 1995>
<150$, 1985>

P1

<1000$, 0>

بدأت الخوارزمية في النظام

انتهت الخوارزمية على هذه الفعالية

<900$, 5>

Marker

<50$, 5>

<100$, 10>

<10 products>

<5 products>

Marker

P2

<0$, 2000>

<50$, 1995>

انتهت الخوارزمية على هذه الفعالية

<150$, 1995>

<150$, 1985>

Questions ?