UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

Sistemas Distribuídos

Prof. Sérgio T. Carvalho
sergio@inf.ufg.br

Clocks, Global States

Introduction

A common
'Resource'

Q. WHICH REQUEST WAS MADE FIRST?

request A

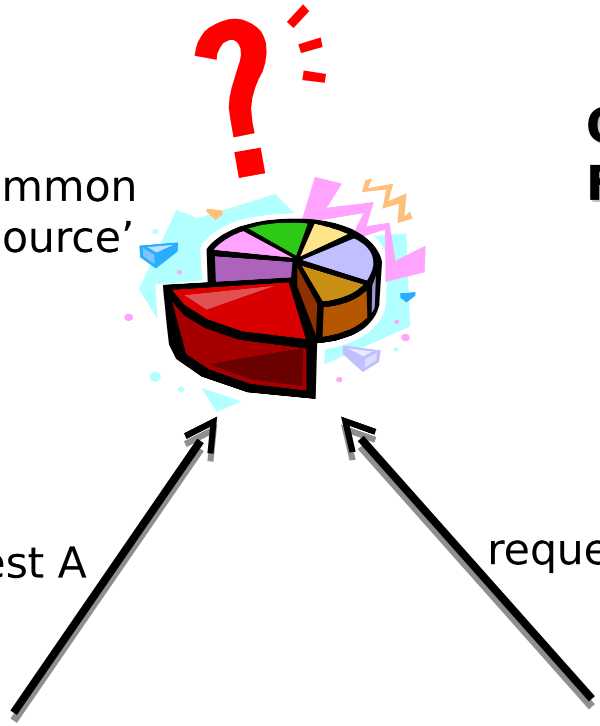
request B

Node A

Node B

Introduction

A common
'Resource'



request A

request B

Node A

Node B

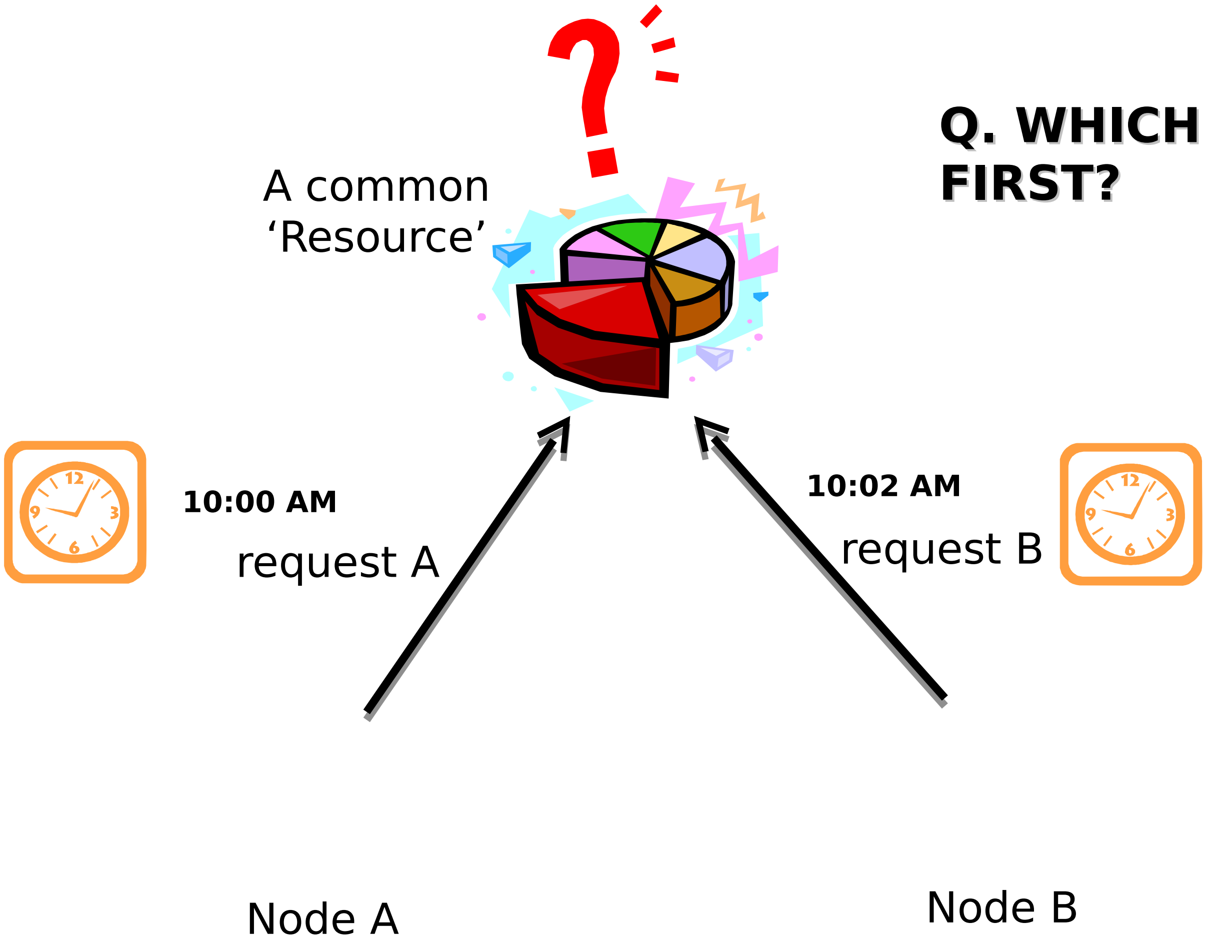
Q. WHICH REQUEST WAS MADE FIRST?

Solution

A Global Clock ??

Global Synchronization?

Introduction



Q. WHICH REQUEST WAS MADE FIRST?

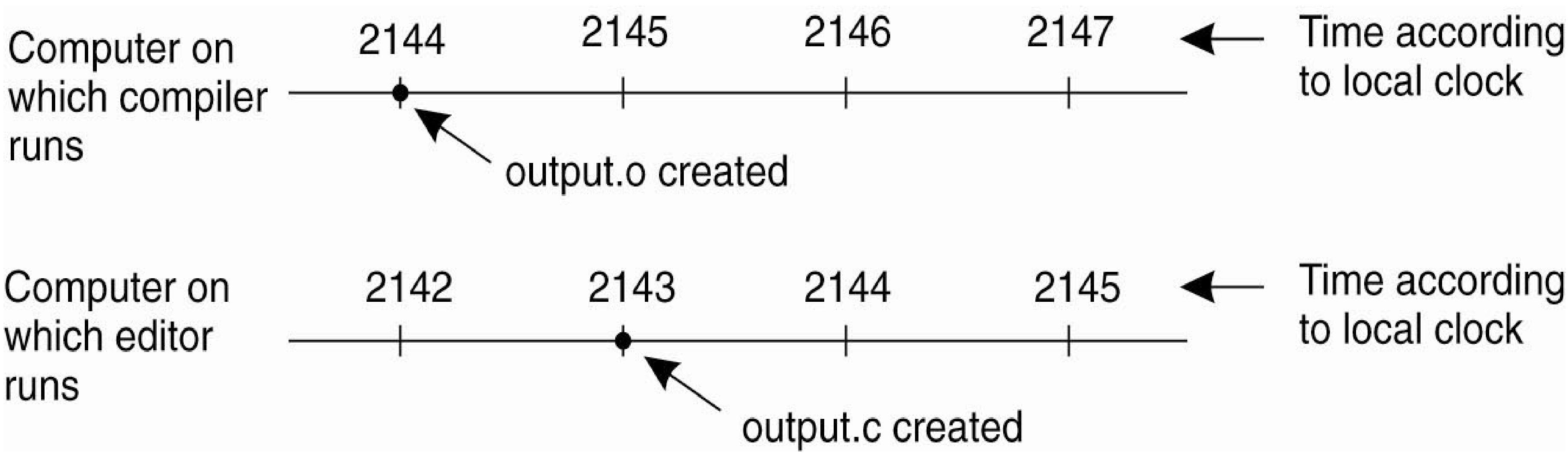
Solution

Individual Clocks?

Are individual clocks accurate, precise?

One clock might run faster/slower?

Introduction



Introduction

- Synchronization in a Distributed System
  - Event Ordering : Which event occurred first?
  - How to sync the clocks across the nodes?
- Can we define notion of ‘happened-before’ without using physical clocks?

Motivation

- Need to measure time accurately
  - To know the time an event occurred at a computer
  - To do this we need to synchronize its clock with an authoritative external clock

Motivation

- Algorithms for clock synchronization useful for
  - Concurrency control based on timestamp ordering (e.g. in distributed transactions)
  - Authenticity of requests e.g. Kerberos
  - Auditing purposes e.g. in e-commerce scenarios
  - Ordering of events – also useful for maintaining consistency of data (e.g. in replication)

Events and process states

- A distributed system is defined as a collection of N processes p_i, i = 1,2,… N
  - Each process $\mathbf{p_i}$ has a state $\mathbf{s_i}$ consisting of its variables (which it transforms as it executes)
  - Processes communicate only by messages (via a network)
  - Actions of processes: **Send, Receive, Change own state**
  - **Event**: the occurrence of a single action that a process carries out as it executes e.g. Send, Receive, Change state
- “**Happened-before**” relation:
  - Events at a single process $\mathbf{p_i}$, can be placed in a total ordering: $\mathbf{e} \rightarrow \mathbf{e'}$ means **e** happened before **e’**
  - a **history of process** is a series of ordered events: **history(p_i) = <e_i^0, e_i^1, e_i^2, …>**

Events and process states

- The system is composed of a collection of processes
- Each process consists of a sequence of events (instructions/subprogram)

Process P :

`instr1`
`instr2`
`instr3` … (Total Order)

- ‘Sending’ and ‘Receiving’ messages among processes
  - ‘Send’ : an event
  - ‘Receive’ : an event

“Happened Before” Relation

- a *‘Happened Before’* b : **a→b**

1. If *a* and *b* are events in the same process, and *a* comes before *b*, then $a \rightarrow b$.

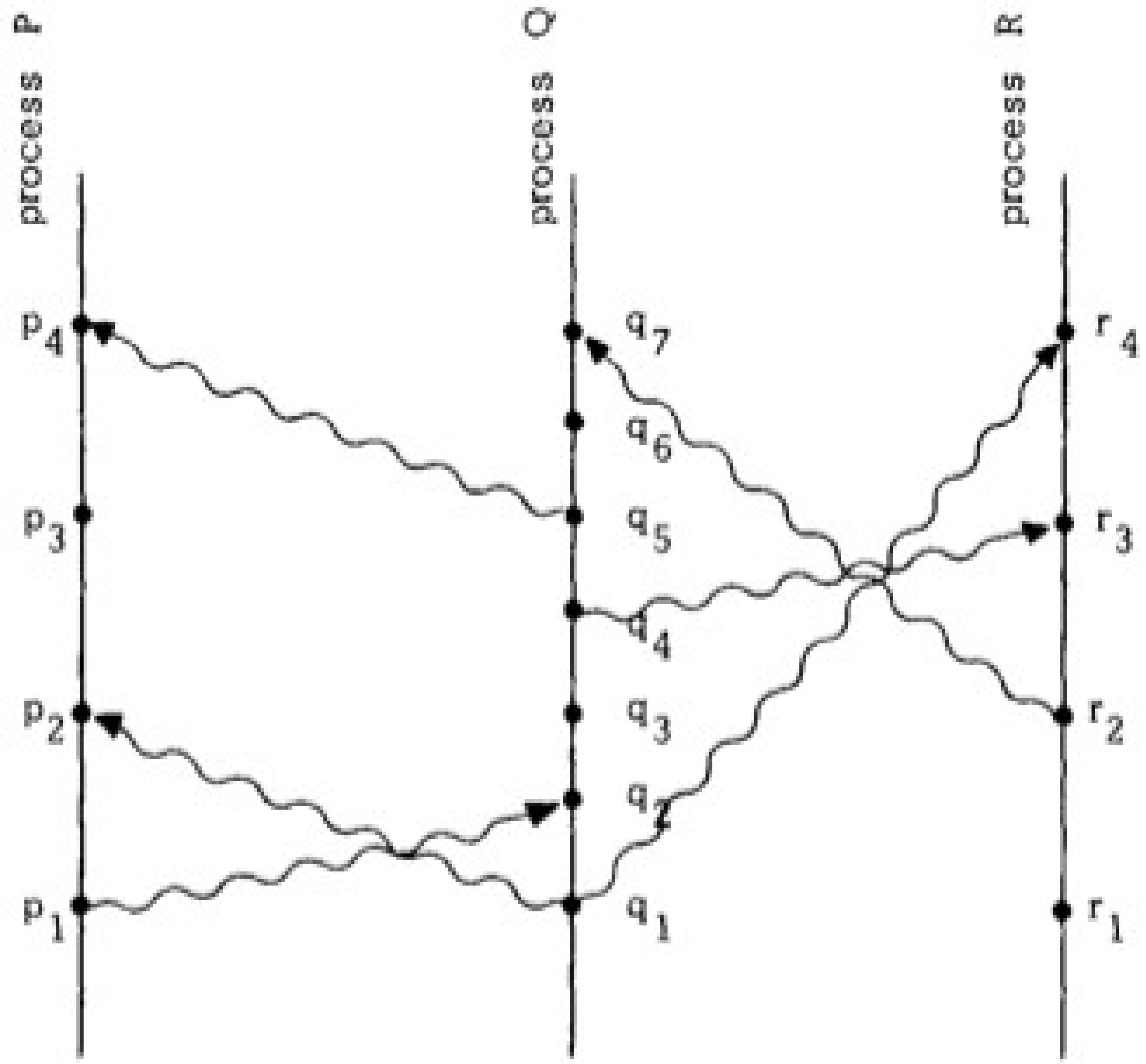
2. If

   a :message sent
   b : receipt of the same message
   then $a \rightarrow b$.

3. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

4. Two distinct events a and b are said to be c*oncurrent if a -/->b and b -/->a*

Space Time Diagram



p1 → **p2**

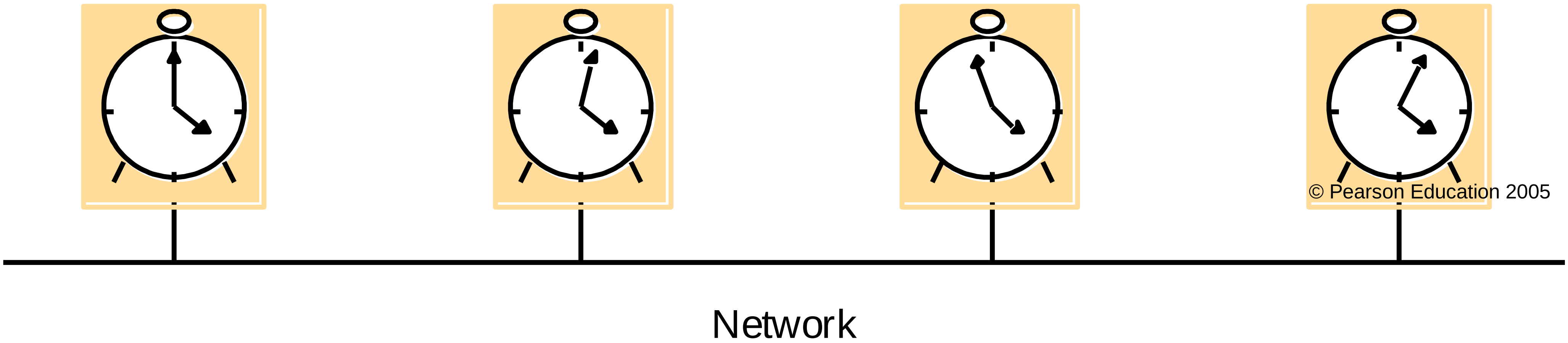
p1 → **q2**

p1 → **r4**

q2 **-/-> p2**

q3 **-/-> p2**

Clock skew and drift



- Computer clocks are not generally in perfect agreement
  - **Skew**: the difference between the times on two clocks (at any instant)

- Computer clocks are subject to clock drift (they count time at different rates)
  - **Clock drift rate**: the difference per unit of time from some ideal reference clock
    - Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10^{-6} secs/sec). High precision quartz clocks drift rate is about 10^{-7} or 10^{-8} secs/sec

Clocks and timestamps

- **Hardware clock:**
  - electronic device counting physical events occurring at a definite frequency (e.g. oscillations in a quartz crystal)
- **Software clock:**
  - At real time, $\boldsymbol{t}$, the OS reads the time on the computer's hardware clock $\boldsymbol{H_i(t)}$
  - It calculates the time on its software clock

$$\boldsymbol{C_i(t) = \alpha H_i(t) + \beta}$$

- **Clock resolution:**
  - period between updates of the clock value: successive events will correspond to different **timestamps** only if the clock resolution is smaller than the time interval between the events

Coordinated Universal Time

- **International Atomic Time** is based on very accurate physical clocks (drift rate 10^{-13} secs/sec)
- **UTC** is an international standard for time keeping
  - It is based on atomic time, but occasionally adjusted to astronomical time
  - It is broadcast from radio stations on land and satellite (e.g. GPS)
- Computers with receivers can synchronize their clocks with these timing signals
  - Signals from land-based stations are accurate to about 0.1-10 millisecond
  - Signals from GPS are accurate to about 1 microsecond

Synchronizing physical clocks

- **External synchronization**
  - Clocks C_i of a set of N computers are synchronized with an external authoritative time source S if:
  - $\mathbf{|S(t) - C_i(t)| < D}$ for $i = 1, 2, \ldots N$ for all $\boldsymbol{t}$ in an interval of real time
  - The clocks C_i are **accurate** to within the bound D.

- **Internal synchronization**
  - The clocks C_i of a set of N computers are synchronized with one another if:
  - $\mathbf{| C_i(t) - C_j(t) | < D}$ for $i,j = 1, 2, \ldots N$ for all $\boldsymbol{t}$ in an interval of real time
  - The clocks C_i **agree** within the bound D.

Clock correctness

- A hardware clock, H is said to be **correct** if its drift rate is within a known bound $\rho > 0$. (e.g. 10^{-6} secs/ sec)
  - the error in measuring the interval between real times t and t' is bounded: $\mathbf{(1 - \rho)\ (t' - t) \le H(t') - H(t) \le (1 + \rho)\ (t' - t)}$ (where $t' > t$)
  - forbids jumps in time readings of hardware clocks
- For software clocks, weaker condition of **monotonicity**
  - $\boldsymbol{t' > t \Rightarrow C(t') > C(t)}$
    - e.g. required by Unix *make*
    - can achieve monotonicity with a hardware clock that runs fast by adjusting the values of α and β in $C_i(t) = \alpha H_i(t) + \beta$
- *a faulty* clock is one that does not obey its correctness condition
  - ***crash failure*** - a clock stops ticking
  - ***arbitrary*** **failure** - any other failure e.g. jumps in time, Y2K

Synchronization in a synchronous system

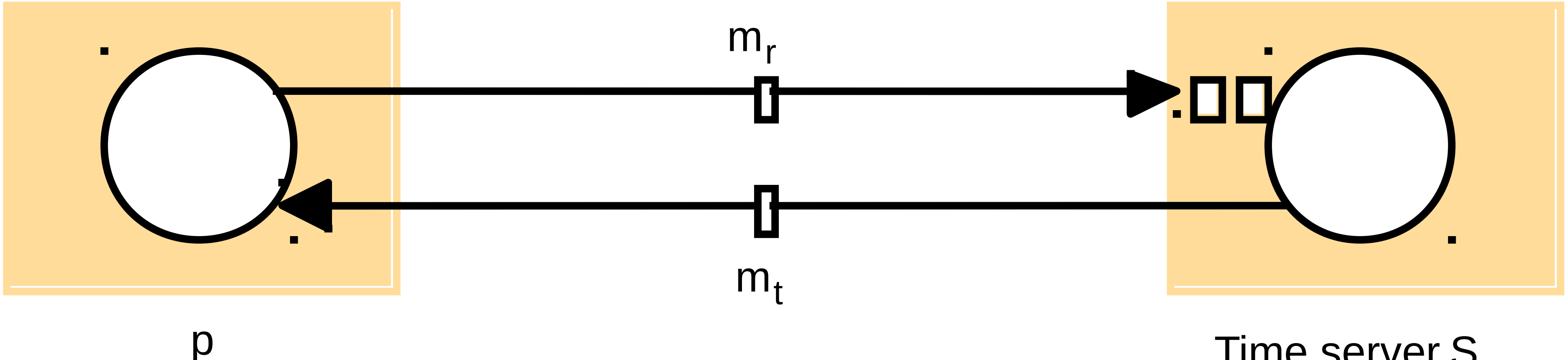
- Recall: a synchronous distributed system is one in which the following bounds are defined:
  - the time to execute each step of a process has known lower and upper bounds
  - each message transmitted over a channel is received within a known bounded time
  - each process has a local clock whose drift rate from real time has a known bound

- **Internal synchronization in a synchronous system:**
- One process p1 sends its local time t to process p2 in a message m,
- p2 could set its clock to $\mathbf{t + T_{trans}}$ where $\mathbf{T_{trans}}$ is the time to transmit m
- T_{trans} is unknown but $\mathbf{min \leq T_{trans} \leq max}$
- uncertainty **u = max-min**. Set clock to **t + (max - min)/2** then **skew ≤ u/2**

Cristian's method for an asynchronous system

- A time server *S* receives signals from a UTC source
  - Process *p* requests time in m_r and receives *t* in m_t from *S*
  - *p* sets its clock to $\boldsymbol{t + T_{round}/2}$
  - **Accuracy** $\pm (T_{round}/2 - min)$:
    - because the earliest time *S* puts *t* in message m_t is *min* after *p* sent m_r.
    - the latest time was *min* before m_t arrived at *p*
    - the time by *S*'s clock when m_t arrives is in the range $[t+min, t + T_{round} - min]$

T_{round} is the round trip time recorded by p
min is an estimated minimum transmission time

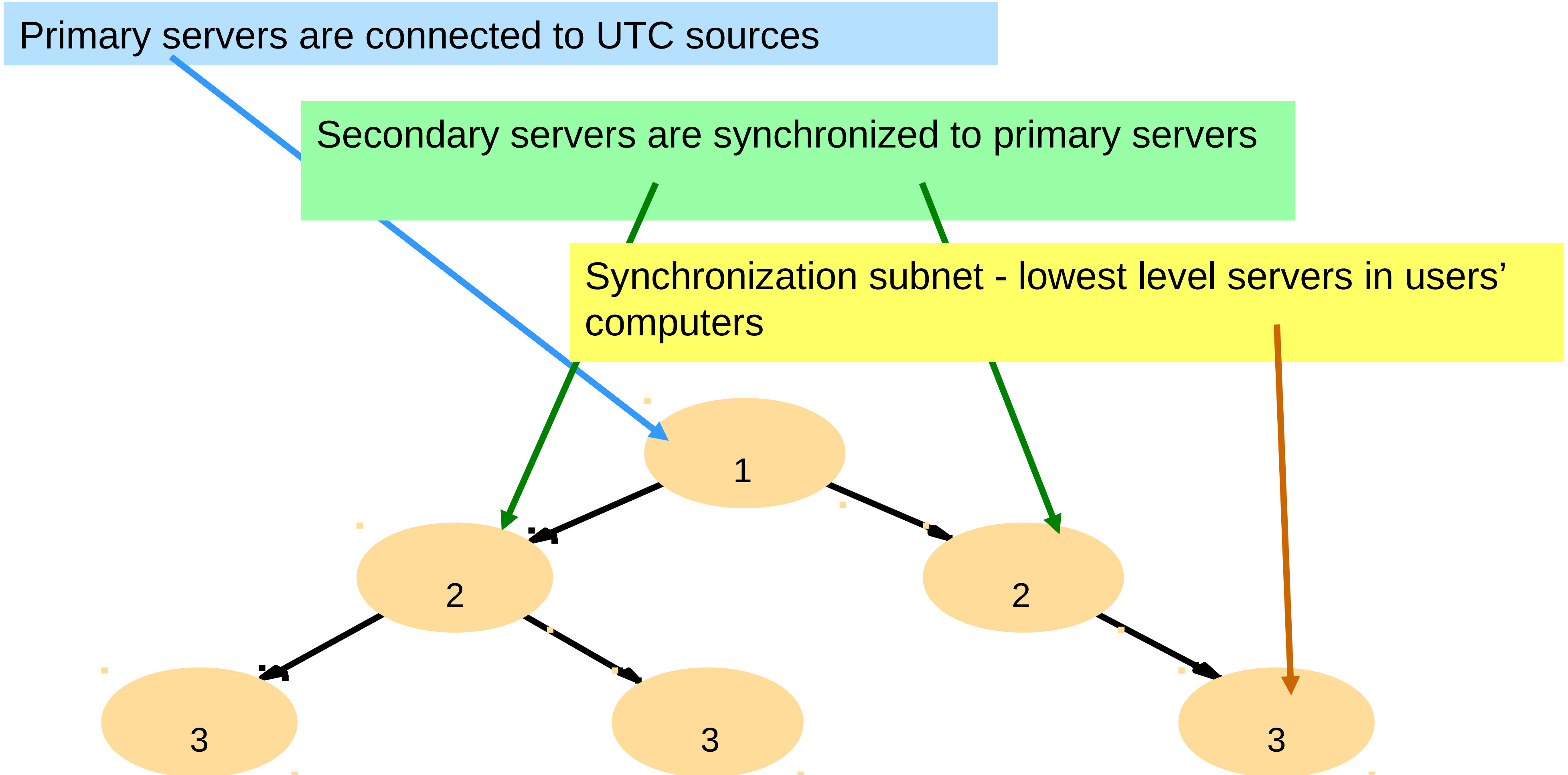


Berkeley algorithm

- Cristian's algorithm -
  - a single time server might fail, so they suggest the use of a group of synchronized servers
  - it does not deal with faulty servers
- **Berkeley algorithm**
  - An algorithm for internal synchronization of a group of computers
  - A master polls to collect clock values from the others (slaves)
  - The master uses round trip times to estimate the slaves' clock values
  - It takes an average (eliminating any above some average round trip time or with faulty clocks)
  - It sends the **required adjustment** to the slaves (better than sending the time which depends on the round trip time)
  - Measurements
    - 15 computers, clock synchronization 20-25 millisecs drift rate < 2x10-5
    - If master fails, can elect a new master to take over (not in bounded time)

Network time protocol (NTP)

- A time service for the Internet - synchronizes clients to UTC



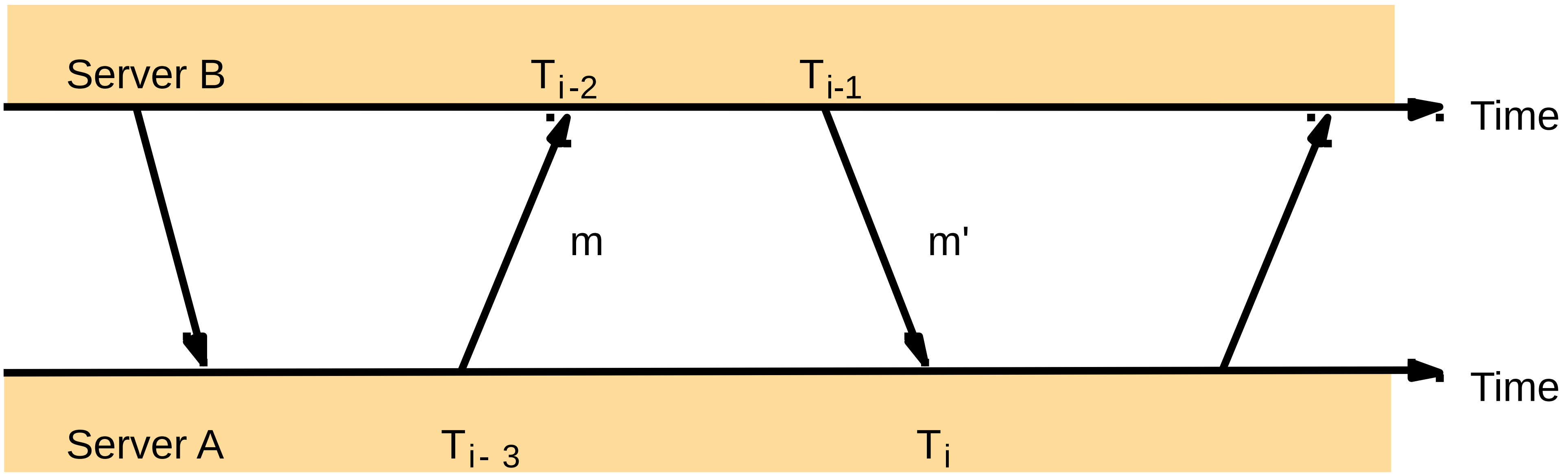
Synchronization of NTP servers

- The synchronization subnet can reconfigure if failures occur
  - a primary that loses its UTC source can become a secondary
  - a secondary that loses its primary can use another primary

- Modes of synchronization:
  - **Multicast**
    - A server within a high speed LAN multicasts time to others which set clocks assuming some delay (not very accurate)
  - **Procedure call**
    - A server accepts requests from other computers (like Cristiain's algorithm). Higher accuracy. Useful if no hardware multicast.
  - **Symmetric**
    - Pairs of servers exchange messages containing time information
    - Used where very high accuracies are needed (e.g. for higher levels)

Messages exchanged between NTP peers

- All modes use UDP
- Each message bears timestamps of recent events:
  - Local times of *Send* and *Receive* of previous message
  - Local times of *Send* of current message
- Recipient notes the time of receipt T_i (we have T_{i-3}, T_{i-2}, T_{i-1}, T_i)
- In symmetric mode there can be a non-negligible delay between messages



What are Logical Clocks?

- A **logical clock** is a mechanism for capturing chronological sequence and causal relationships of events in a distributed system.

- Clock Implementation
  - Data structure
  - Clock Update Protocol

- Logical clock algorithms of note are:
  - Scalar clocks
  - Vector clocks
  - Matrix clocks

“Happened Before” Relation

- a *‘Happened Before’* b : **a→b**

1. If *a* and *b* are events in the same process, and *a* comes before *b*, then $a \rightarrow b$.

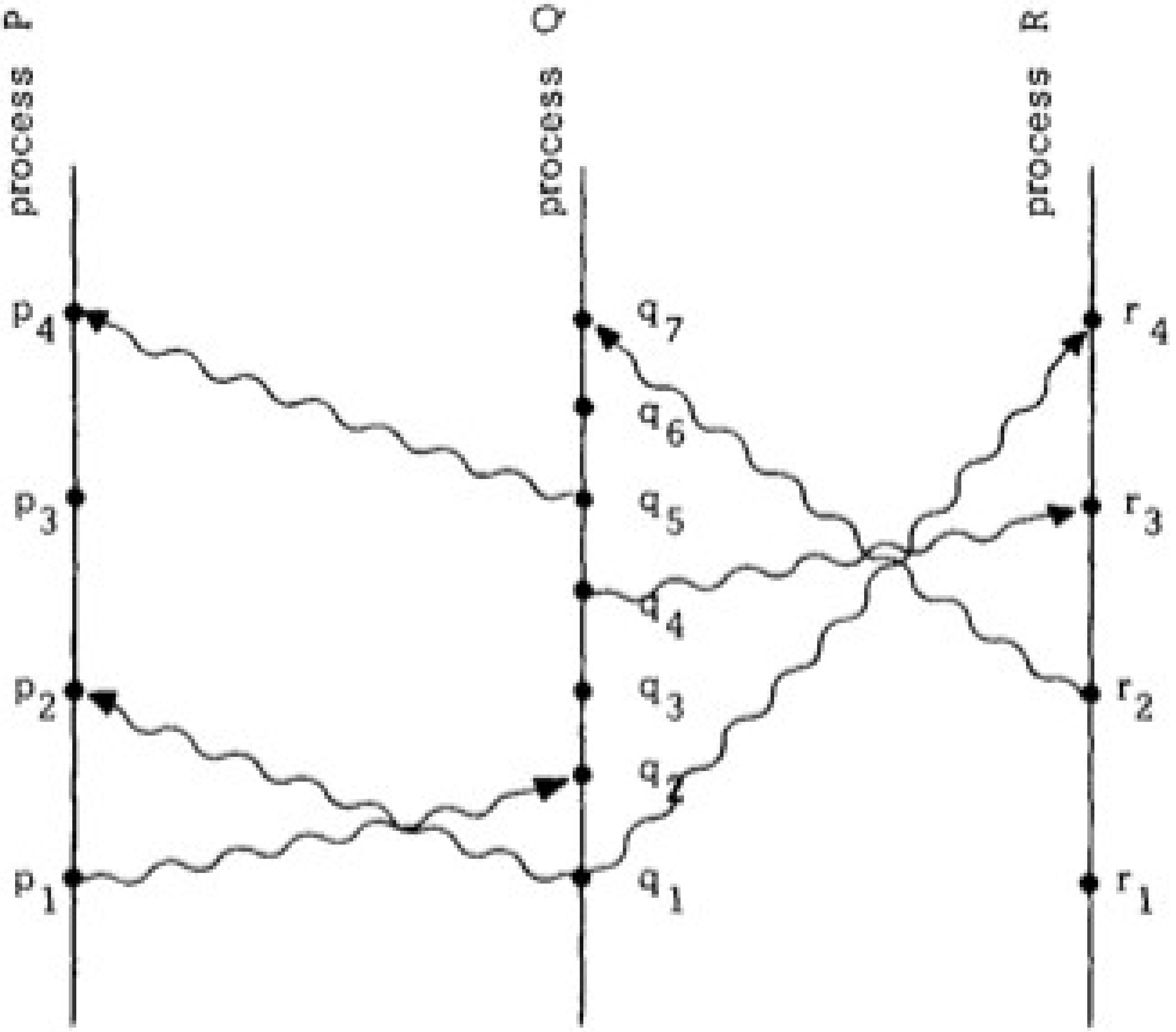
2. If

   a :message sent
   b : receipt of the same message
   then $a \rightarrow b$.

3. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

4. Two distinct events a and b are said to be c*oncurrent if a -/->b and b -/->a*

Space Time Diagram



p1 → **p2**

p1 → **q2**

p1 → **r4**

q2 **-/-> p2**

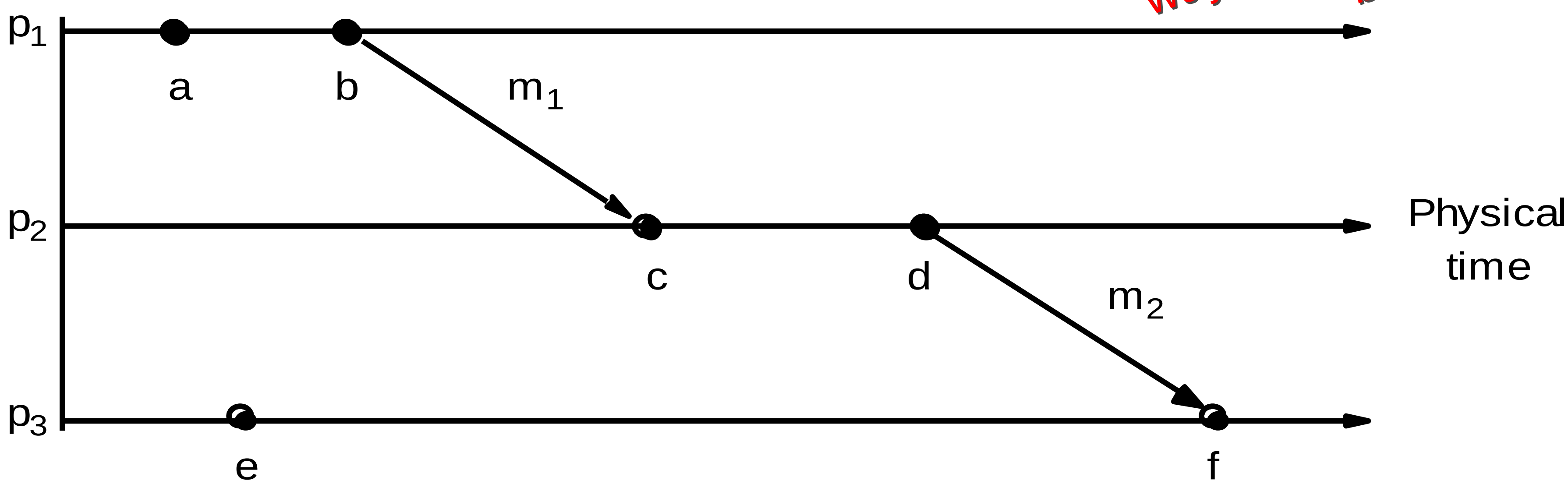
q3 **-/-> p2**

Logical time and logical clocks

- Instead of synchronizing clocks, event ordering can be used:

1. If two events occurred at the same process p_i (i = 1, 2, … N) then they occurred in the order observed by p_i, (recall happened-before relation)
2. when a message, *m* is sent between two processes, *send(m)* happened before *receive(m)*
3. The happened before relation is transitive

Not all events are related in this way! (consider “a” and “e” below)



Logical Clocks (1/2)

Problem: How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?

Solution: attach a timestamp $C(e)$ to each event e, satisfying the following properties:

P1: If a and b are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.

P2: If a corresponds to sending a message m, and b to the receipt of that message, then also $C(a) < C(b)$.

Logical Clocks (2/2)

Each process P_i maintains a **local** counter C_i and adjusts this counter according to the following rules:

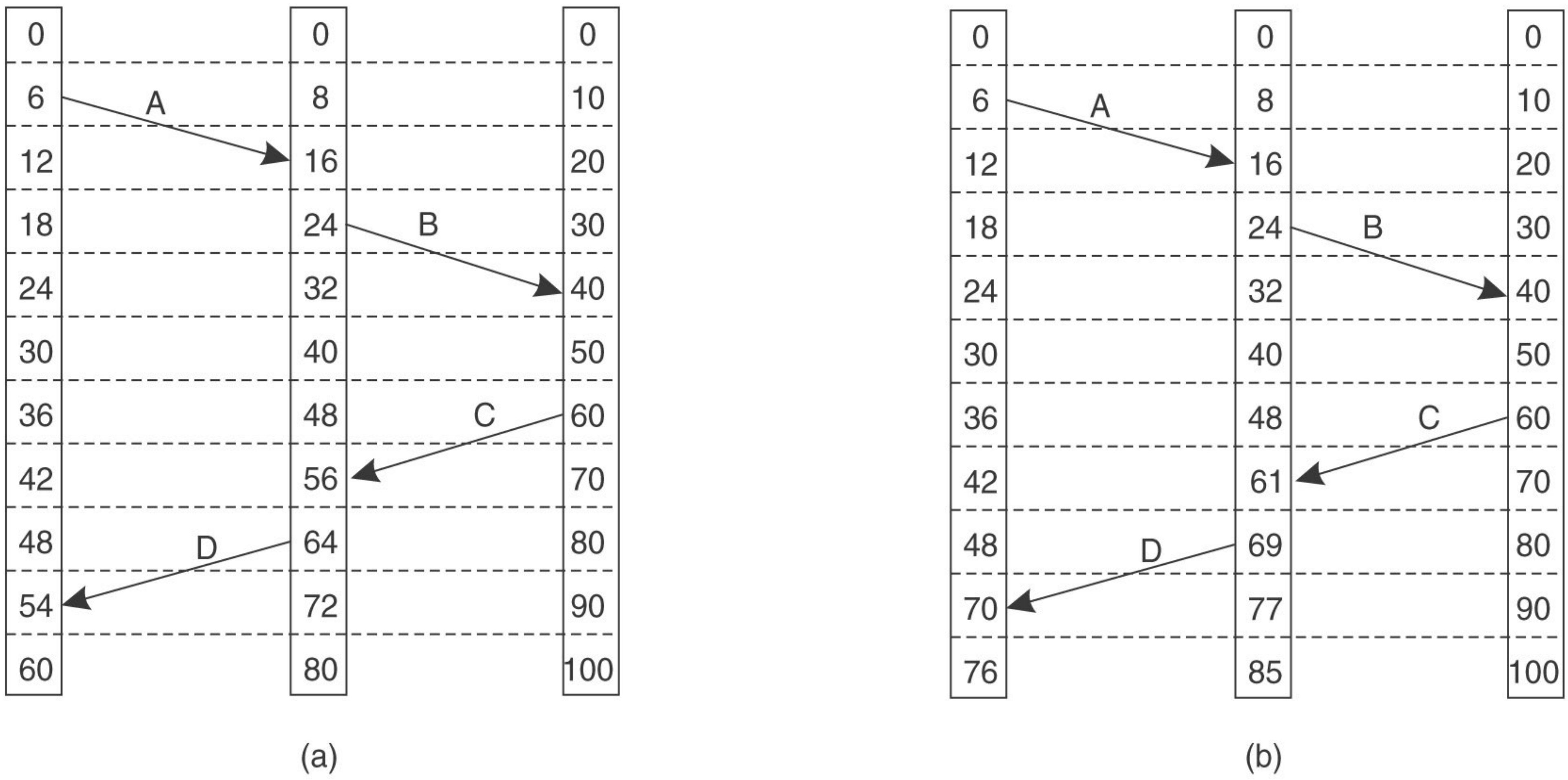
1: For any two successive events that take place within P_i, C_i is incremented by 1.

2: Each time a message m is sent by process P_i, the message receives a timestamp $T_m = C_i$.

3: Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j:

$$C_j \leftarrow \max\{C_j + 1, T_m + 1\}.$$

Logical Clocks

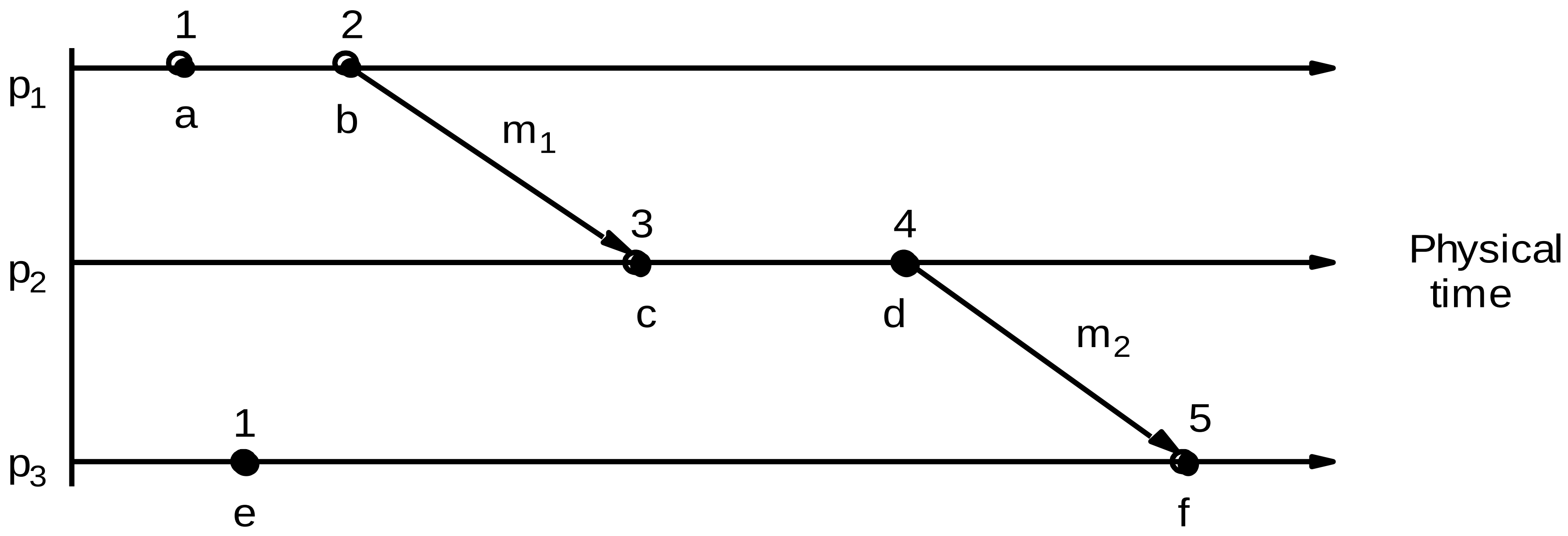


Correction of clocks

Lamport's logical clocks

- A logical clock is a monotonically increasing software counter. It need not relate to a physical clock.

- Each process p_i has a logical clock, C_i which can be used to apply logical timestamps to events



Total Ordering with Logical Clocks

Problem: it can still occur that two events happen at the same time. Avoid this by attaching a process number to an event:

$$P_i \text{ timestamps event } e \text{ with } C_i(e).i$$

Then: $C_i(a).i$ before $C_j(b).j$ if and only if:

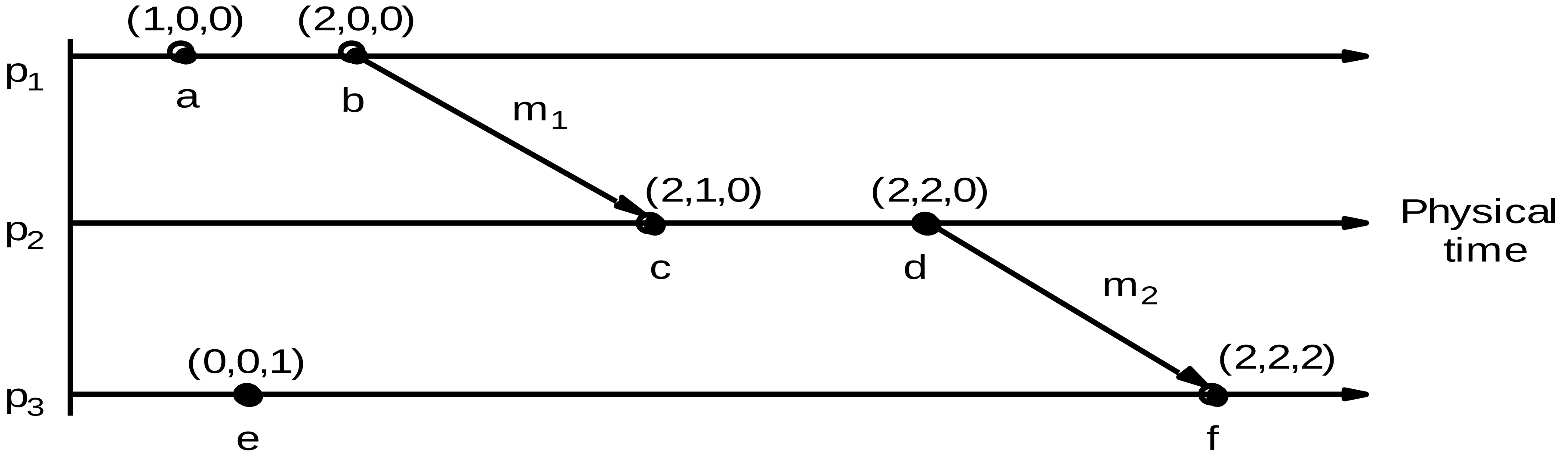
1: $C_i(a) < C_j(a)$; or

2: $C_i(a) = C_j(b)$ and $i < j$

Vector clocks

A problem with Lamport's clocks: C(e) < C(e') doesn't imply e -> e'

- Vector clock V_i at process p_i is an array of *N* integers
- VC1: initially $V_i[j] = 0$ for $i, j = 1, 2, \ldots N$
- VC2: before p_i timestamps an event it sets $V_i[i] := V_i[i] + 1$
- VC3: p_i piggybacks $t = V_i$ on every message it sends
- VC4: when p_i receives (m,t) it sets $V_i[j] := \max(V_i[j]\ , t[j])$ $j = 1, 2, \ldots N$ *(then before next event adds I to own element using VC2)*



Vector clocks

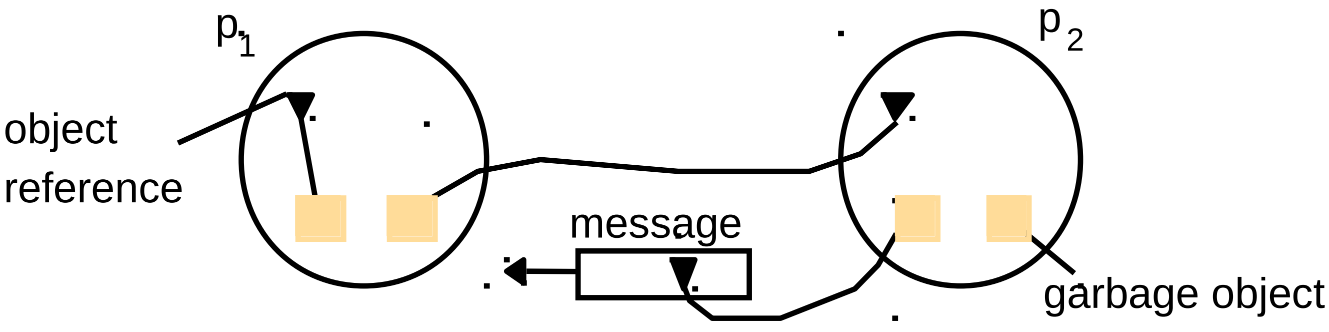
- At p_i
  - $V_i[i]$ is the number of events p_i timestamped locally
  - $V_i[j]$ is the number of events that have occurred at p_j (that has potentially affected p_i)

Need for global state

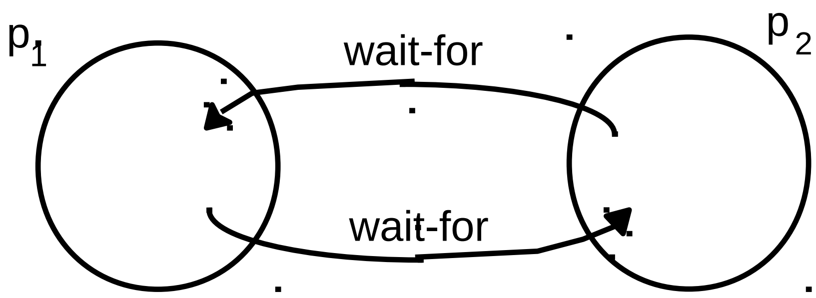
- Finding out whether a particular property in the system is true (i.e. whether the system in a particular global state)
  - *Distributed garbage collection*
  - *Distributed deadlock detection*
  - *Distributed debugging*

- Problem: absence of global time
  - *use the idea of process histories instead!*
    - $history(p_i) = \langle \underbrace{e_i^0, e_i^1, e_i^2}, \dots \rangle$
  - *Global state corresponds to a set of initial prefixes of the individual process histories*

Detecting global properties



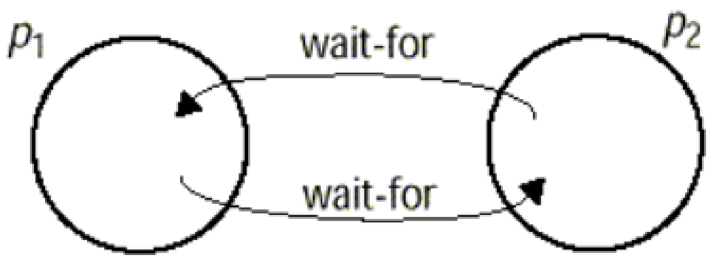
a. Garbage collection



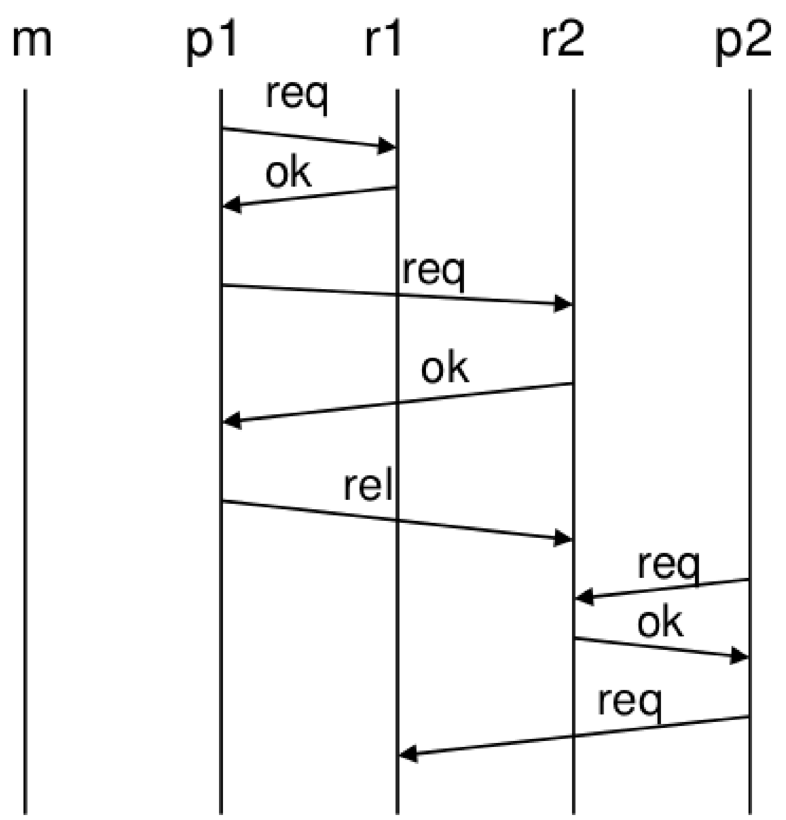
b. Deadlock

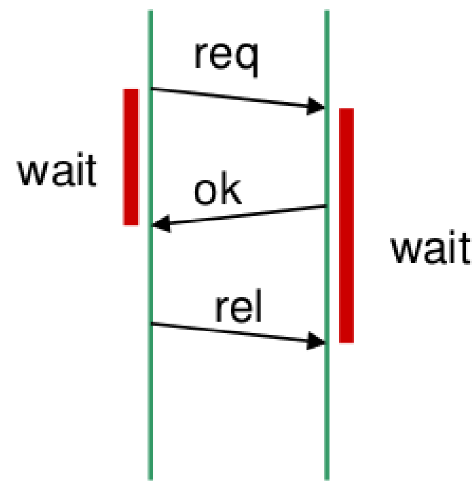
Problems that would require the view on a global state

- Distributed deadlock detection: is there a cyclic wait-for-graph amongst processes and resources in the system?



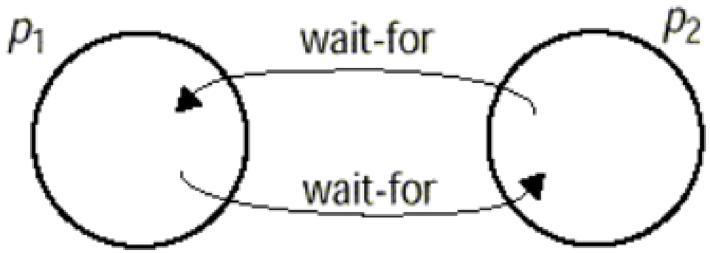
  - problem: system state changes while we conduct observation, hence we may get an inaccurate observation result



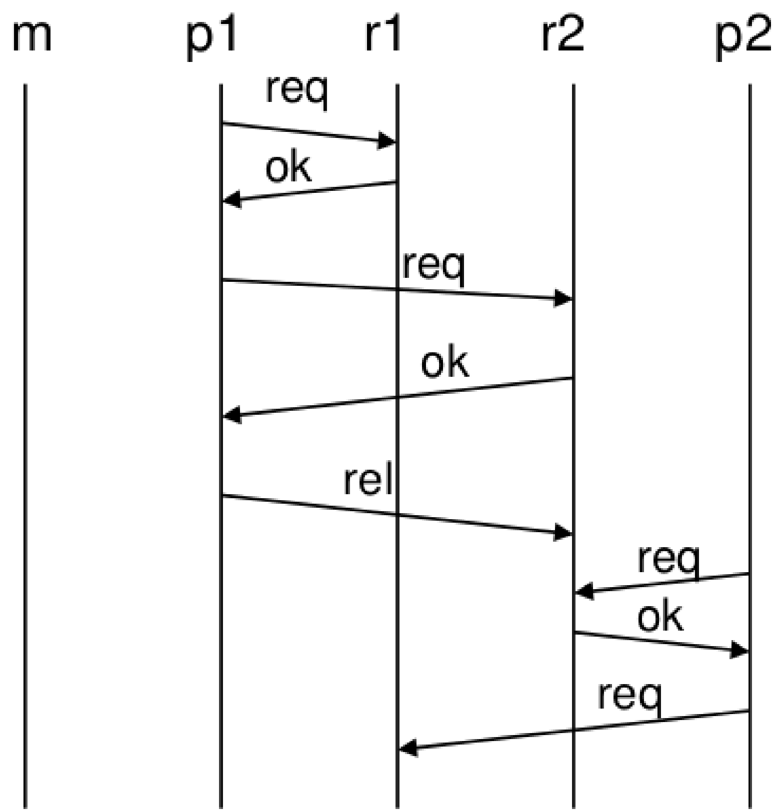


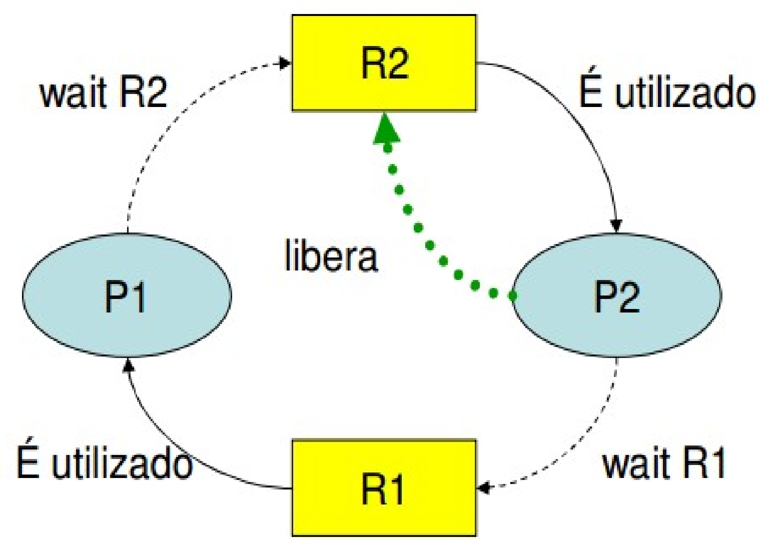
Problems that would require the view on a global state

- Distributed deadlock detection: is there a cyclic wait-for-graph amongst processes and resources in the system?



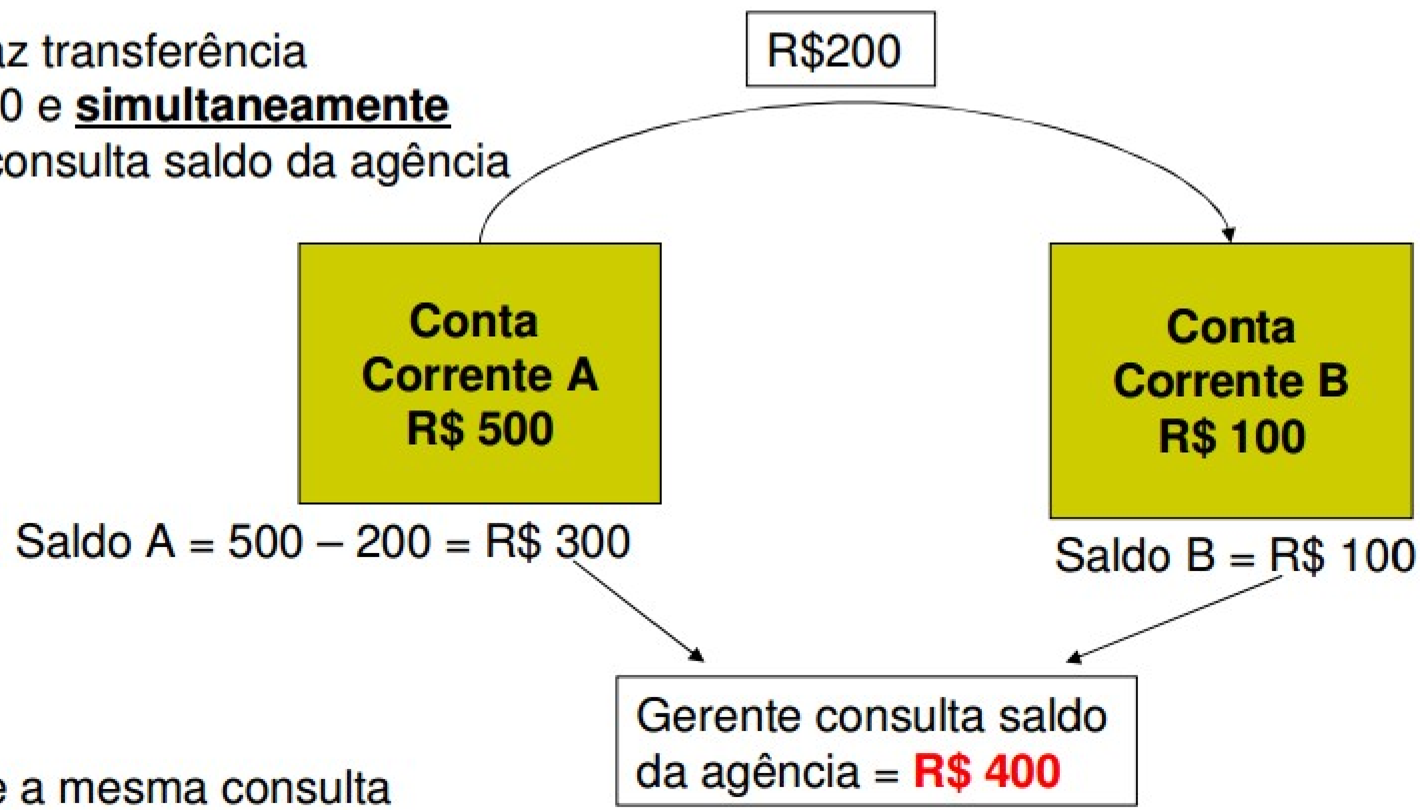
  - problem: system state changes while we conduct observation, hence we may get an inaccurate observation result





Need for global state

Cliente faz transferência
de R$ 200 e **simultaneamente**
gerente consulta saldo da agência



Considere a mesma consulta
realizada pelo gerente
imediatamente antes ou
imediatamente após a
transferência = saldo era **R$600**

é preciso observar o
estado dos canais de
comunicação

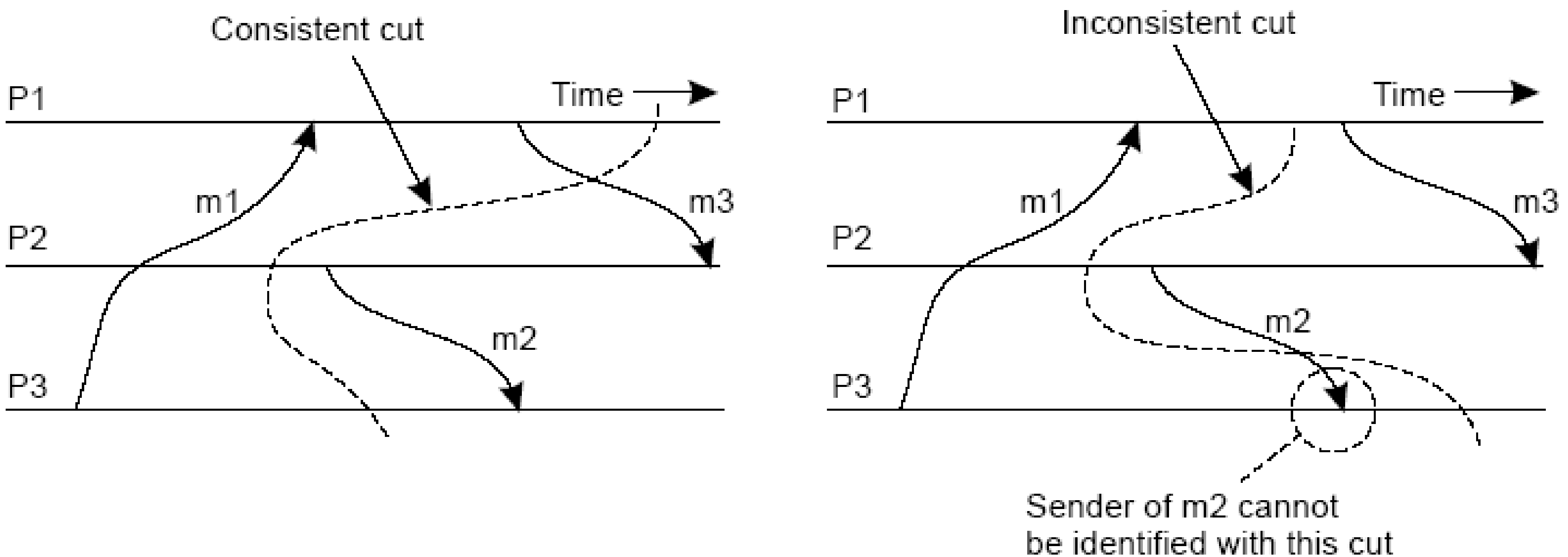
Need for global state

- Essential problem is the absence of global time
  - Perfect clock synchronization cannot be achieved -> global states cannot be observed from recorded state at agreed time
- Global state from local states recorded at different real times?
- History of process p_i: $h_i = < e_i^0, e_i^1, e_i^2, \ldots >$
  - each event e_i^k is either an internal action of process or sending or receiving a message over communication channels

Need for global state

- S_i^k is the state of process p_i immediately before the kth event occurs
- Processes record sending and receiving of all messages as part of their state
- 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 union of prefixes of process history:
  - $C = h_1^{c1} \cup h_2^{c2} \cup \ldots \cup h_N^{cN}$

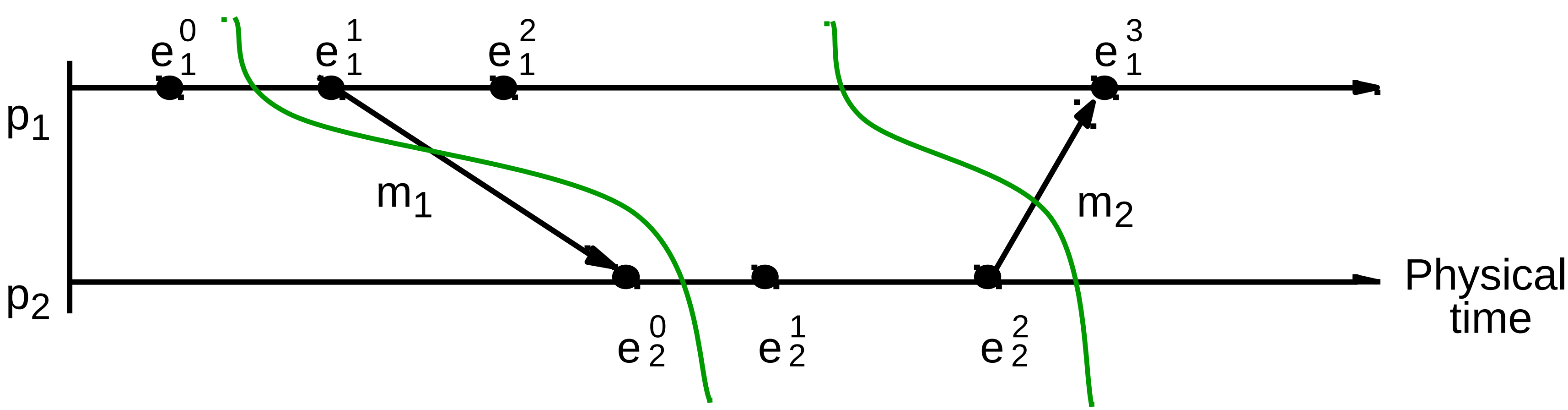
Global States



(a) (b)

Global States

- Inconsistent cut
  - I.e. p_2 includes the receipt of the message m_1, but at p_1 it does not include the sending of message
- Consistent cut
  - I.e. Includes both sending and receipt of message m_1, includes sending but not receipt of message m_2
  - Cut *C* is consistent if, for each event *e* such it contains, it also contains all events happened-before that event
    - For all events $e \in C, f \rightarrow e \Rightarrow f \in C$



Inconsistent cut

Consistent cut

Global States

- Consistent global state
  - State that corresponds to a consistent cut
  - Global system state $S = (s_1, s_2, .., s_N)$

- Global state sequences
  - Execution of system as series of transitions between global states of the system
    - $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow ...$
    - In each transition precisely one event occurs at some single process in the system
    - Linearization is an ordering of events in a global history that is consistent with happened-before relation on *H*
    - *S'* is reachable from a state *S* if there is a linearization that passes through *S* and then *S'*