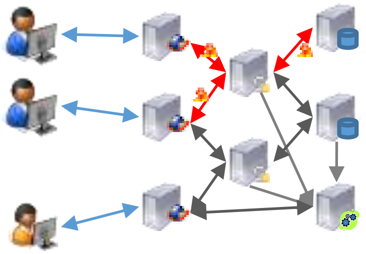


Verteilte Systeme/ Distributed Systems

Artur Andrzejak

8

Logical Time: Repetition Lamport Time

Happened-Before Relation

- Lamport defined the **Happened-Before** relation $\rightarrow$
  - $a \rightarrow b$ event a happened before event b
  - Example: a: message being sent, b: message receipt
  - Transitive: if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

- If a and b occur on different processes <u>that do not exchange messages</u>, then neither $a \rightarrow b$ nor $b \rightarrow a$ are true
- These events are **concurrent**

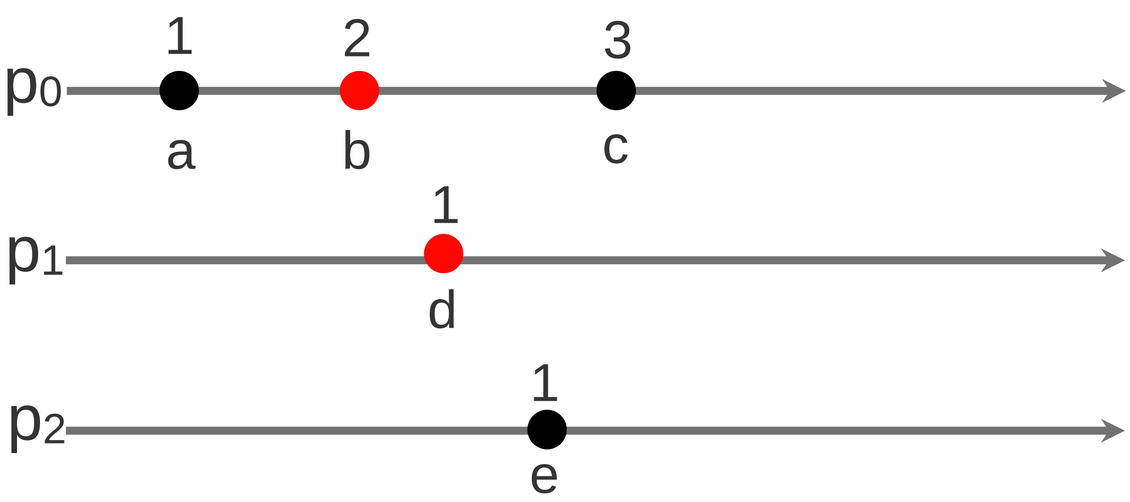
Happened-Before Relation

- Assign Lamport's "clock" value **L**(e) to each event e
  - if a $\rightarrow$ b then L(a) < L(b)
  - Makes sense, since time cannot run backwards

- Lamport's algorithm (informal)
  - Each message carries a timestamp L of the sender's clock
  - When a message arrives:
    - **if** receiver's clock < message timestamp **then**
      set system clock to (message timestamp + 1)
    - **else** do nothing
  - Clock must be advanced between any two events in the same process

Lamport Clock: Problems

- Lamport clock ensures that …
  - If $a \rightarrow b$ then $L(a) < L(b)$
- But $L(a) < L(b)$ it does <u>not</u> imply that $a \rightarrow b$!
  - Therefore, we cannot conclude which events are causally related based on Lamport time only



We have $L(d) < L(b)$, but <u>not</u> $d \rightarrow b$ (it holds: $d \parallel b$)

Logical Time:
Vector Clocks

Following contents are based on the slides of Lecture 10 in the course:
Distributed Computing, Google Code University/ Rutgers University:
By Paul Krzyzanowski, pxk@cs.rutgers.edu, ds@pk.org

Vector Clocks: Papers

- The **causality problem**:
  - "$L(a) < L(b)$ it does not imply that $a \rightarrow b$"
- This problem can be solved by the **vector clocks**
- An independent development of
  - C. J. Fidge, "Timestamps in Message-Passing Systems That Preserve the Partial Ordering," Proceedings of the 11th Australian Computer Science Conference, vol. 10, no. 1, pp. 56–66, 1988.
  - F. Mattern, "Virtual Time and Global States of Distributed Systems," Parallel and Distributed Algorithms, pp. 215–226, 1989.

Vector Clocks: Data Structures

- A **vector clock** for N processes (or distributed nodes) has as data structure an array of N integers
- Each process p_i has its own data structure V_i
- We call the (current) value of V_i the **vector time** of p_i
- The entry $V_r[s]$ (i.e. s-th component in V_r) is the "interpretation" of the logical time of p_s
  - … as seen by the process p_r
- When sending a message, the current vector time of the sender process is sent with it (piggyback = Huckepack)

Vector Clocks: Rules

1. Vector is initialized to 0 at each process

   $V_i\,[j] = 0$ for $i, j = 1, \ldots, N$

2. Process increments its element of the vector in local vector before timestamping event:

   $V_i\,[i] = V_i\,[i] + 1$

3. Message is sent from process P_i with V_i attached to it

4. When P_j receives message, it compares vectors element by element and sets local vector to higher of two values

   $V_j\,[i] = \max(V_i\,[i], V_j\,[i])$ for $i=1, \ldots, N$

Comparing Vector Timestamps

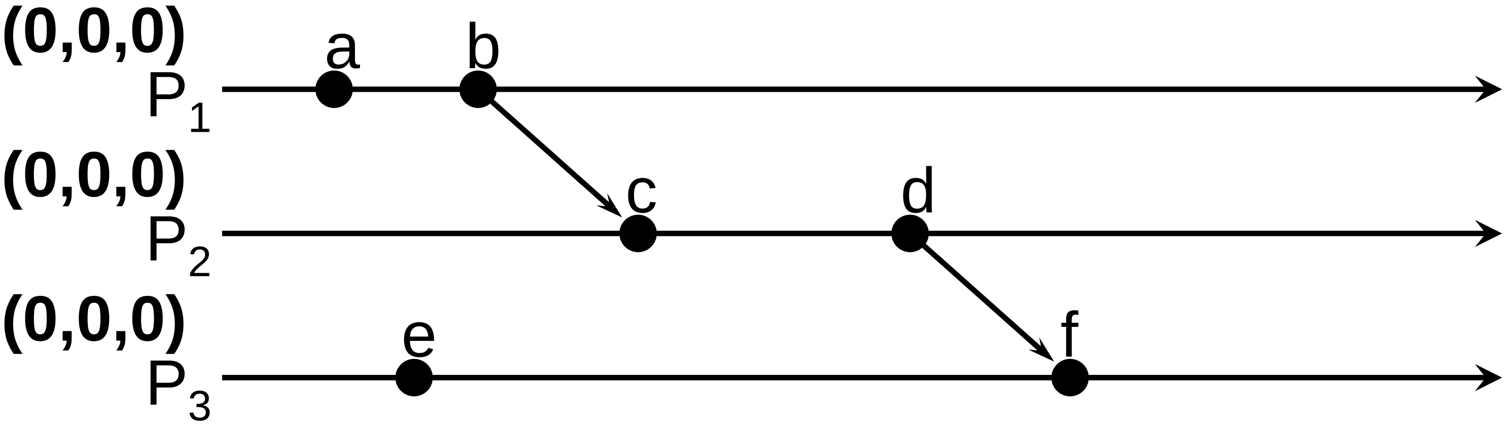
- Define:

  $V = V'$ iff $V[i] = V'[i]$ for all $i = 1 \dots N$
  $V \leq V'$ iff $V[i] \leq V'[i]$ for all $i = 1 \dots N$

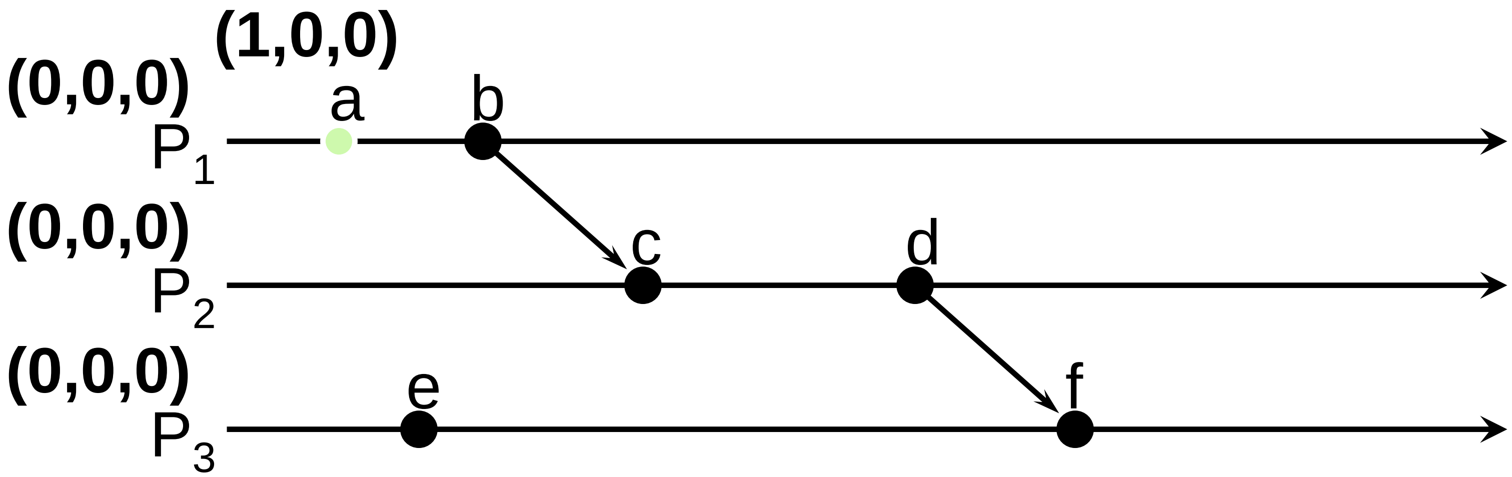
- We have then, for any two events e, e'
  - if $e \to e'$ then $V(e) < V(e')$
    - Just like Lamport's algorithm
  - NEW: if $\mathbf{V(e) < V(e')}$ then $\mathbf{e \to e'}$

- Two events are **concurrent** if
  - **Neither** $V(e) \leq V(e')$ **nor** $V(e') \leq V(e)$

Vector Timestamps: Example

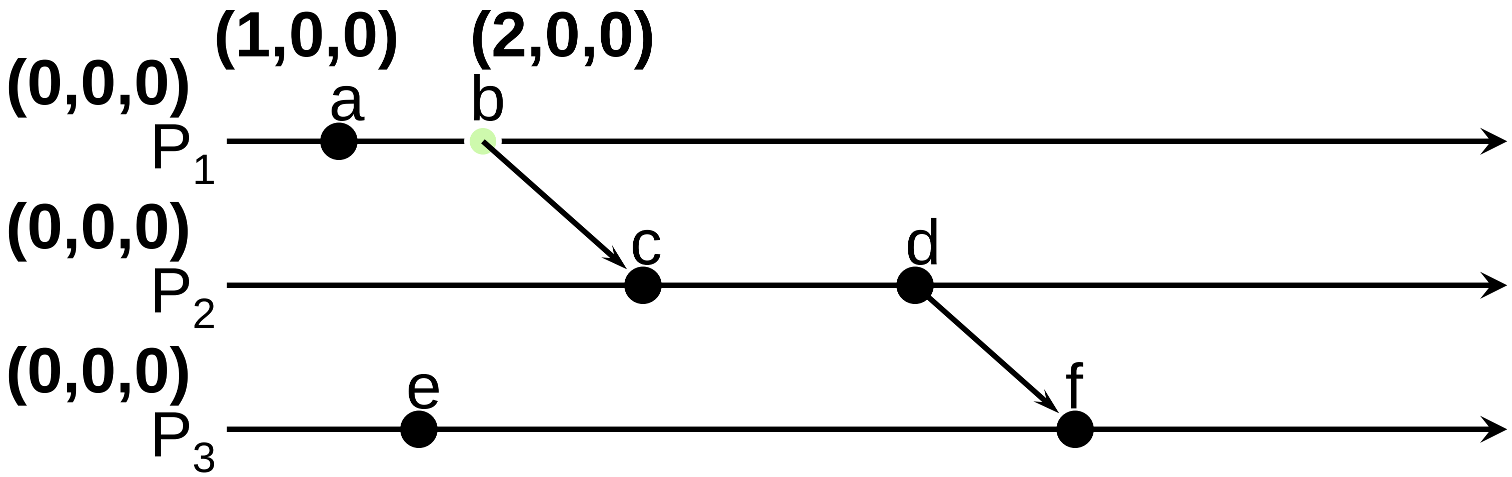


Vector Timestamps: Example



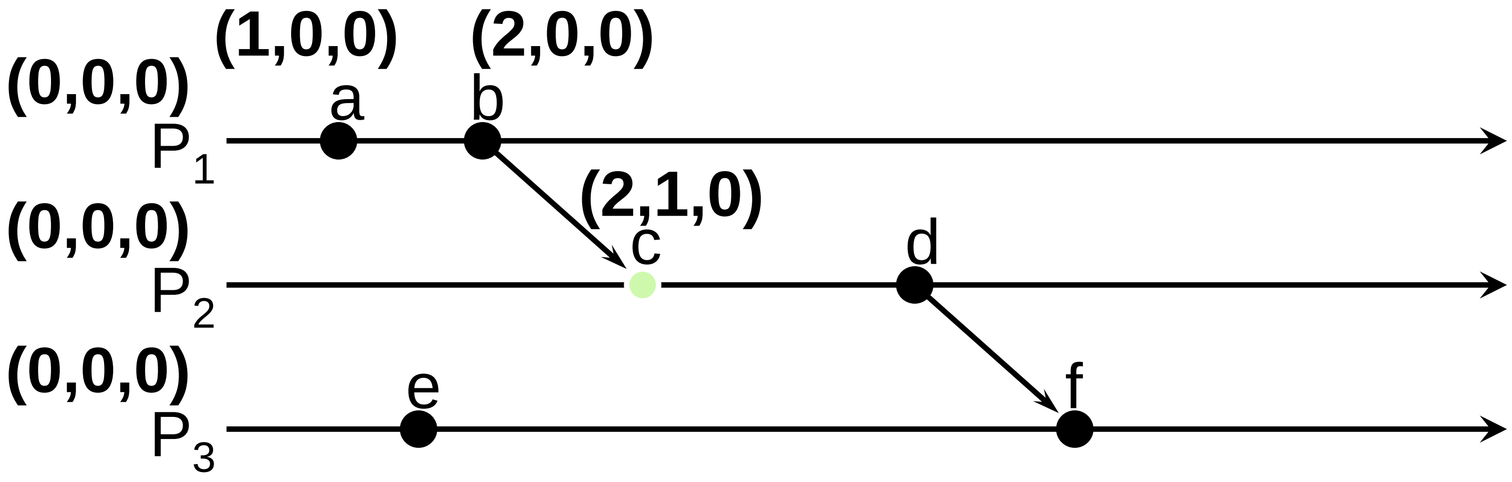
Event	timestamp
a	(1,0,0)

Vector Timestamps: Example



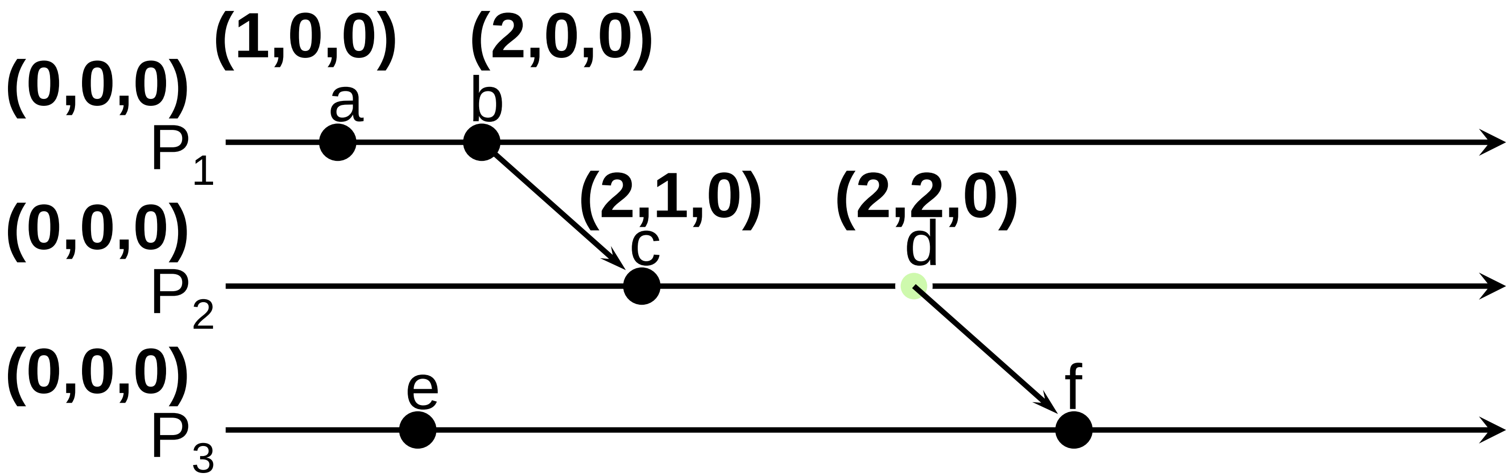
Event	timestamp
a	(1,0,0)
b	(2,0,0)

Vector Timestamps: Example



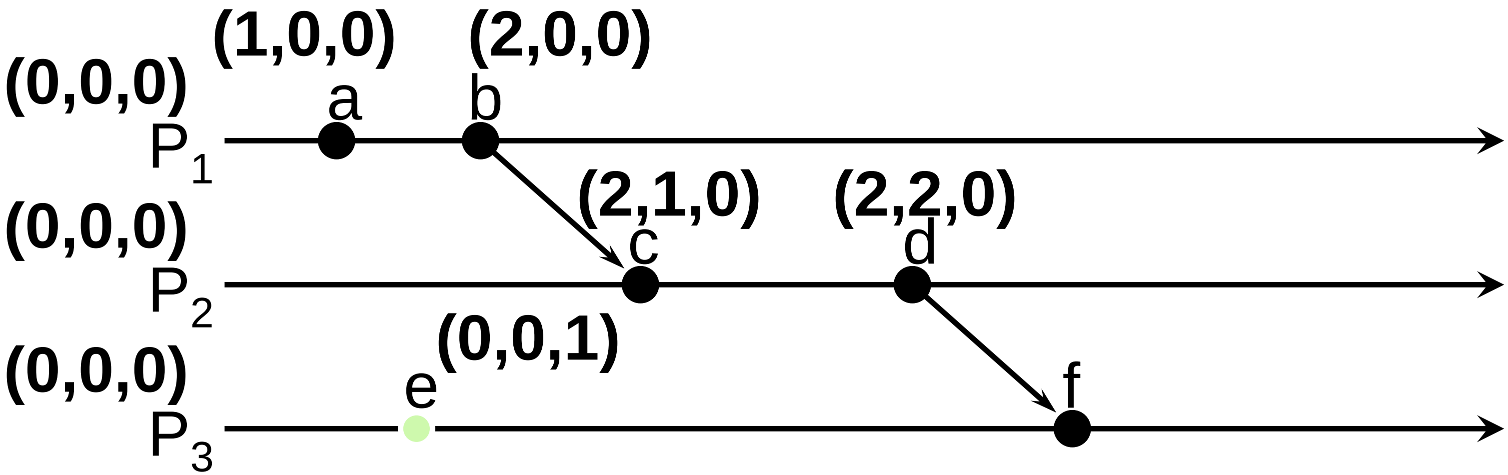
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)

Vector Timestamps: Example



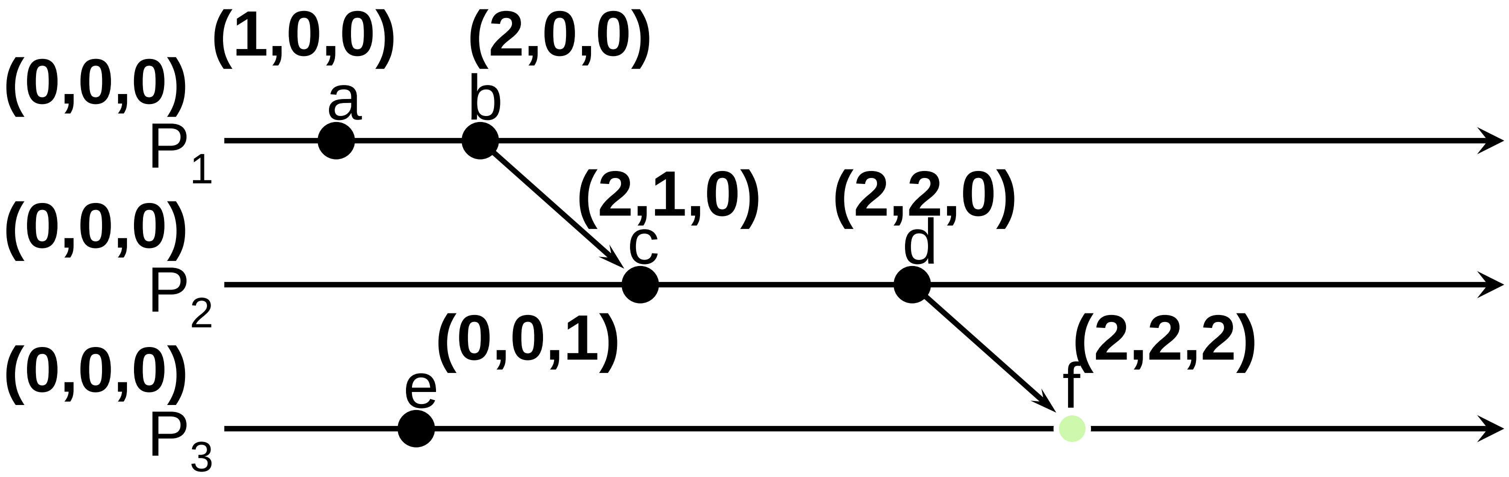
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)

Vector Timestamps: Example



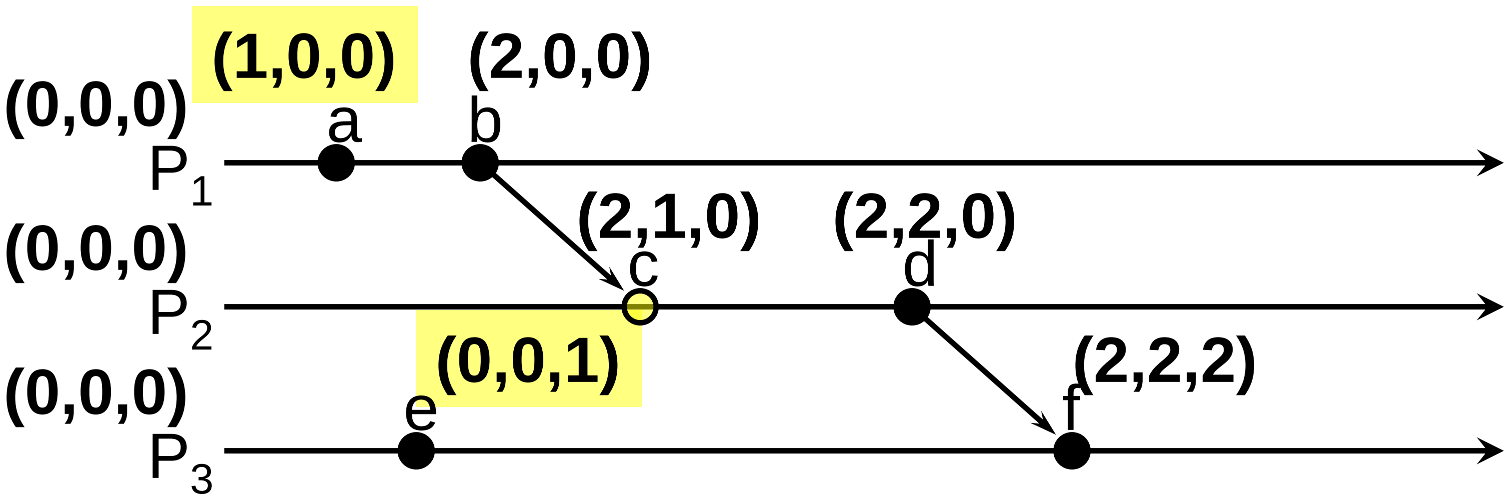
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)

Vector Timestamps: Example



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

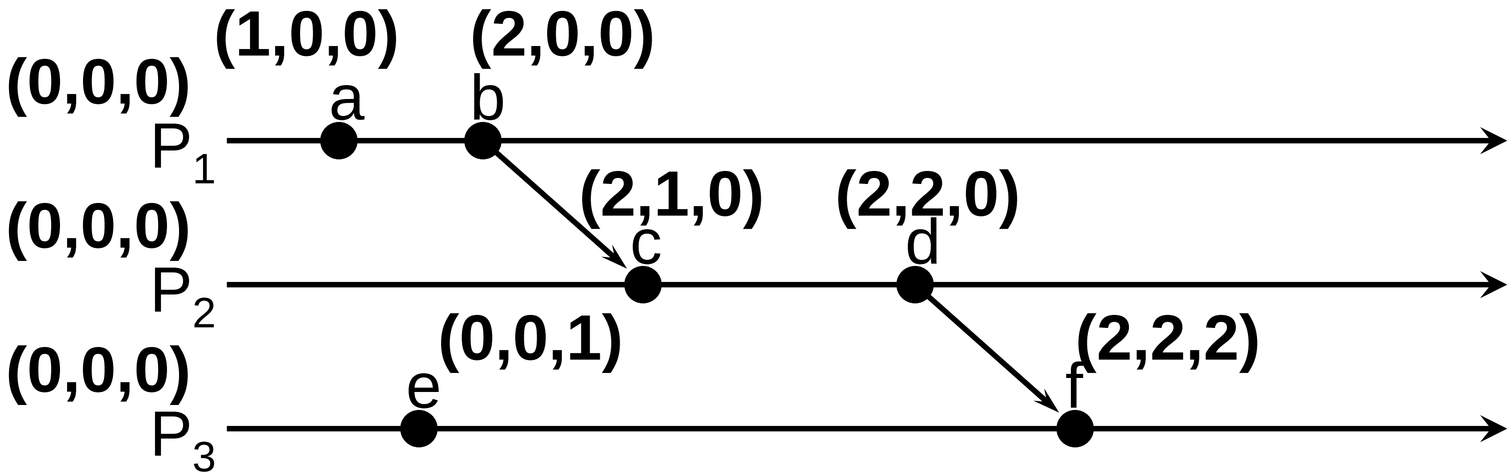
Vector Timestamps: Concurrent Events



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

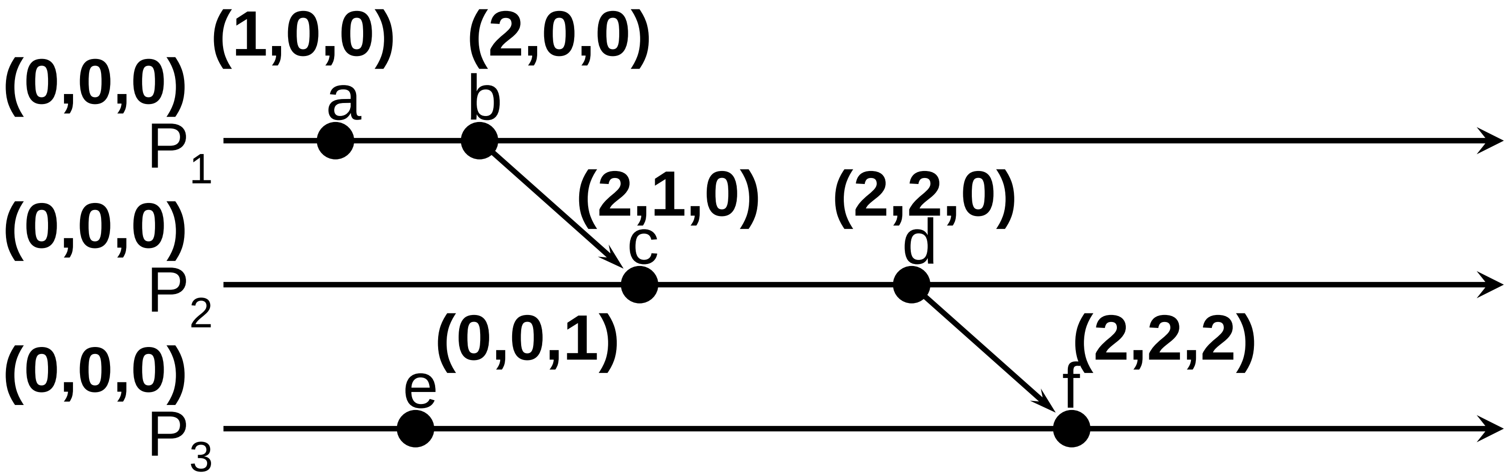
Vector Timestamps: Concurrent Events



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

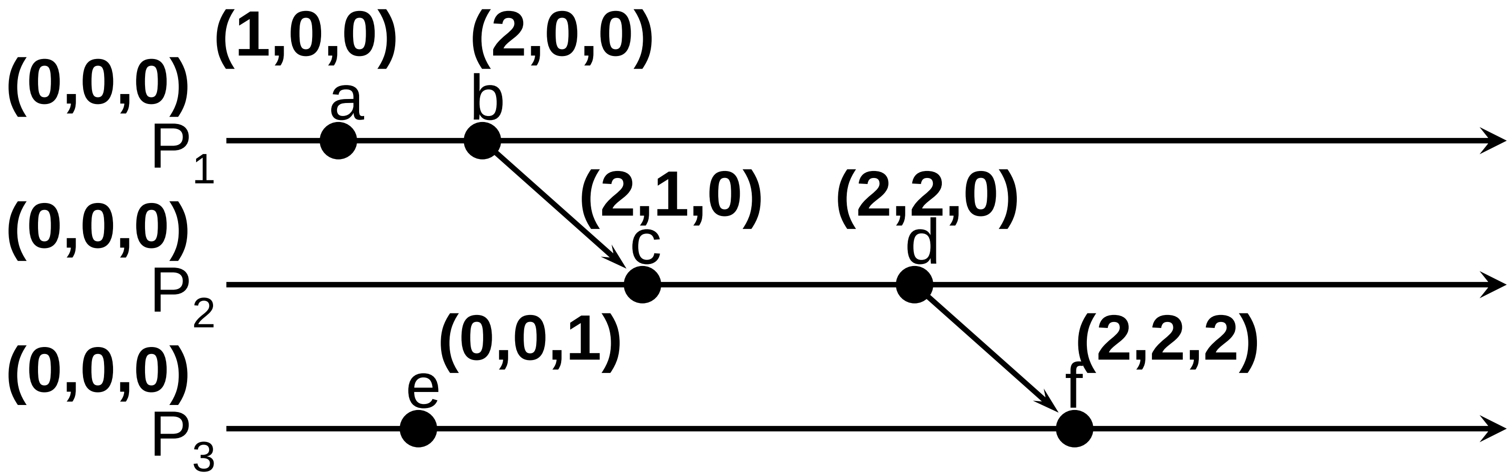
Vector Timestamps: Concurrent Events



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

Vector Timestamps: Concurrent Events



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

Summary: Logical Clocks & Partial Ordering

- Causality
  - If $a \rightarrow b$ then event a can affect event b
- Concurrency
  - If neither $a \rightarrow b$ nor $b \rightarrow a$ then one event cannot affect the other
- Partial Ordering
  - Causal events are sequenced
- Total Ordering
  - All events are sequenced

Global States

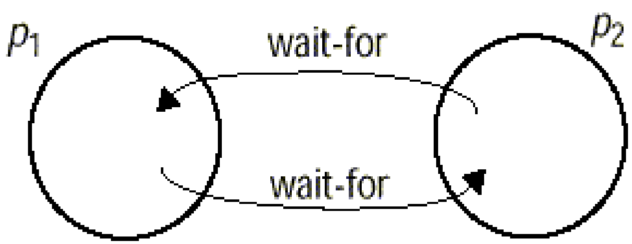
Some slides are based on the part 2 of the course:
Distributed Software Systems - Winter 2004/2005
Stefan Leue, Uni Münster

Global States

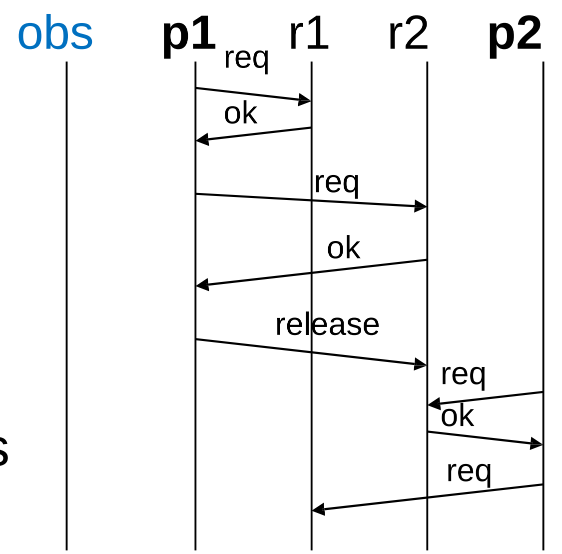
- The local state of a single process is easy to obtain
  - Just “freeze” the execution, collect contents of CPU-registers, RAM content of the user space, and the OS Process Control Block (PCB)
- Consider <u>many processes in a distributed system</u>
  - Is it enough to collect all local states at a certain point of time?
- Two problems:
  - There might be some messages still in delivery
  - “A certain point of time” – difficult/impossible to determine consistently for all processes (see time sync)

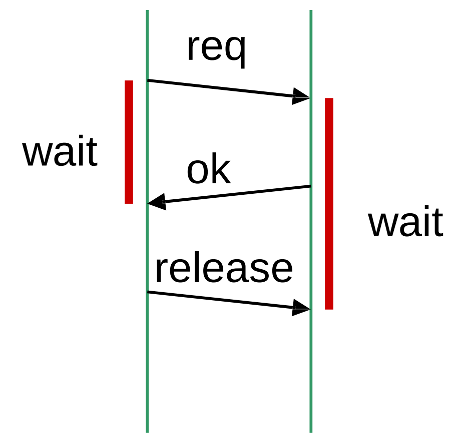
Motivation for Knowing the Global State

- 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?



© Pearson Education 2001

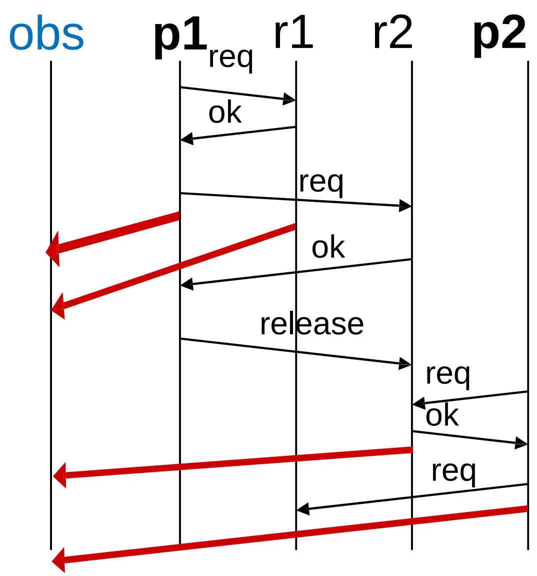




Problem: system state changes during the observation, hence we may get an inaccurate observation result

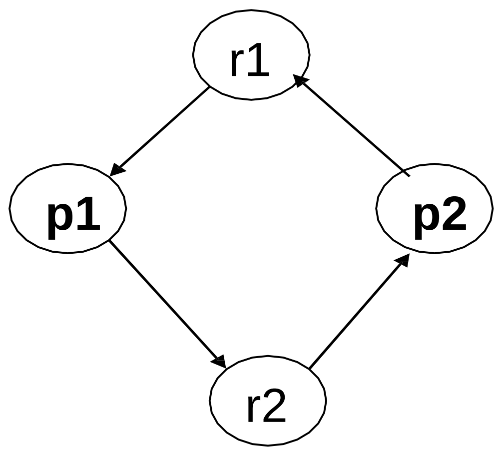
Motivation for Knowing the Global State

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



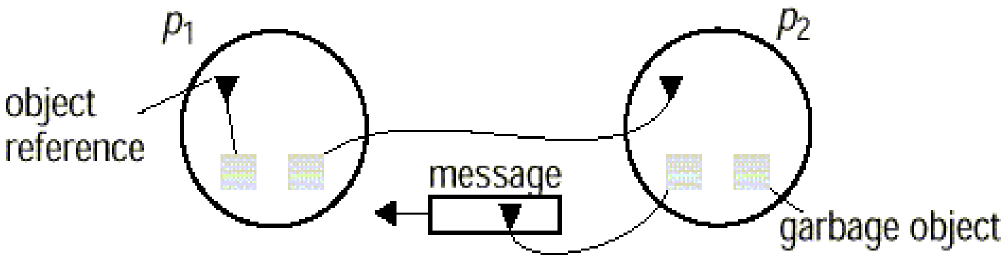
What observer obs knows:

- **p1** holds r1, waits for r2
- r1 is assigned to **p1**
- r2 is assigned to **p2**
- **p2** holds r2, waits for r1

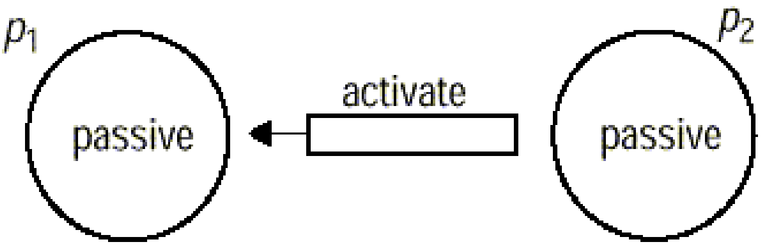


More Problems Requiring Global State

- **Distributed garbage collection**: is there any reference to an object left?



- **Distributed termination detection**: is there either an active process left or is any process activation message in transit?



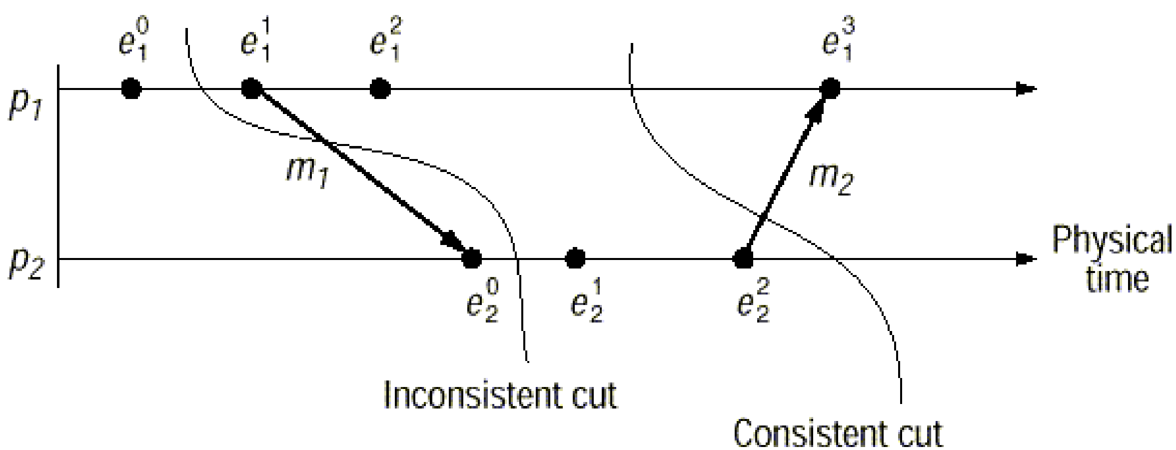
Observability - Assumptions

We can observe and record:

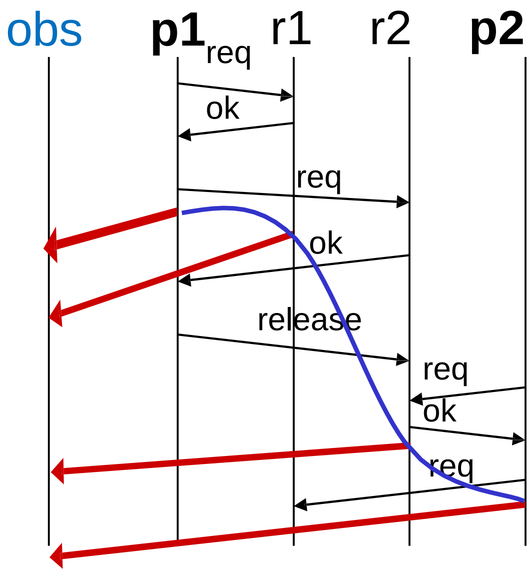
- Process states and communication status
- Global states cannot be observed based on physical clock timestamps
  - Due to inability to synchronize clocks
- We can observe events and local states of processes, and thereby infer the states of the communication channels

Cuts

- Assemble an assumed global system state from state information of the processes,
  - Such that the resulting "cut" through the system is consistent
- **Consistent cuts** - conditions:
  - CC1. Only events that could have happened simultaneously (concurrent events) are part of the same cut, and
  - CC2. The cut includes no events that are the effect of another event in the $\rightarrow$ relation without that the cause is also part of the cut

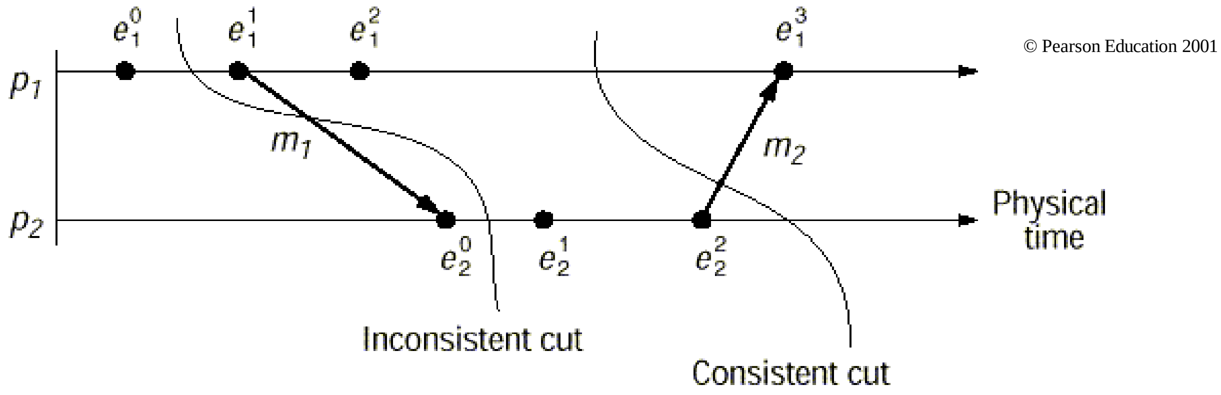


Recall our Deadlock Example



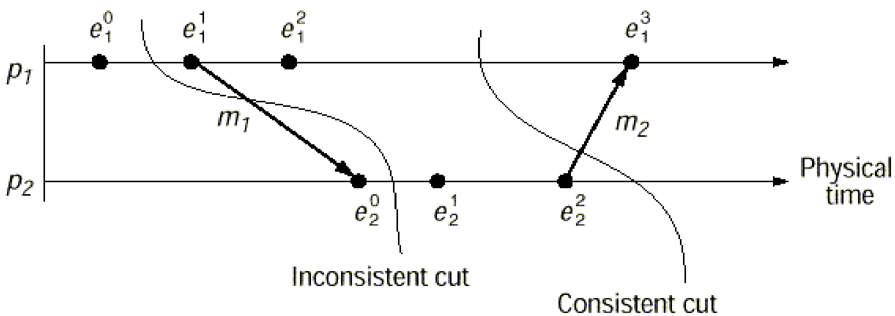
- Consistent or not? Why?

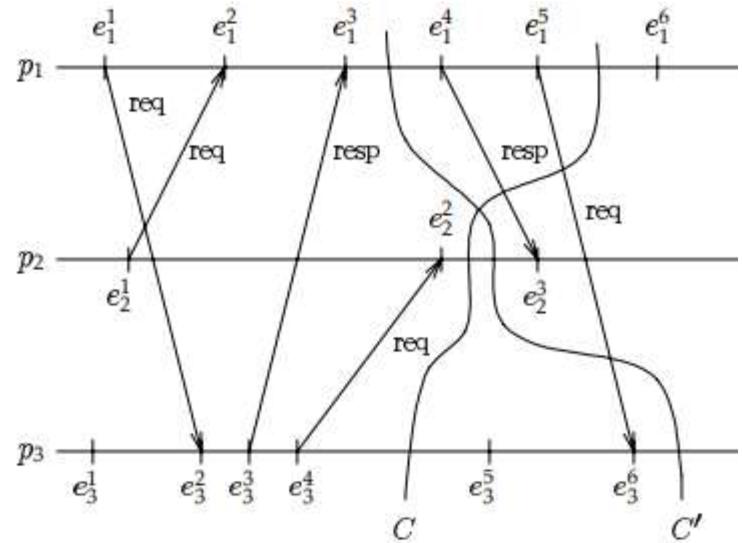
Formal Definitions



- Let **history** of a process i be **h_i** = $< e_i^0 , e_i^1 , e_i^2 ,\ldots>$
  - Each event e_i^k corresponds to either a send, a receive or an internal action
  - We record the send and receive events as a part of the state for recovering the channel information
- Let be the **global history H** = $(h_1, h_2, .., h_N)$

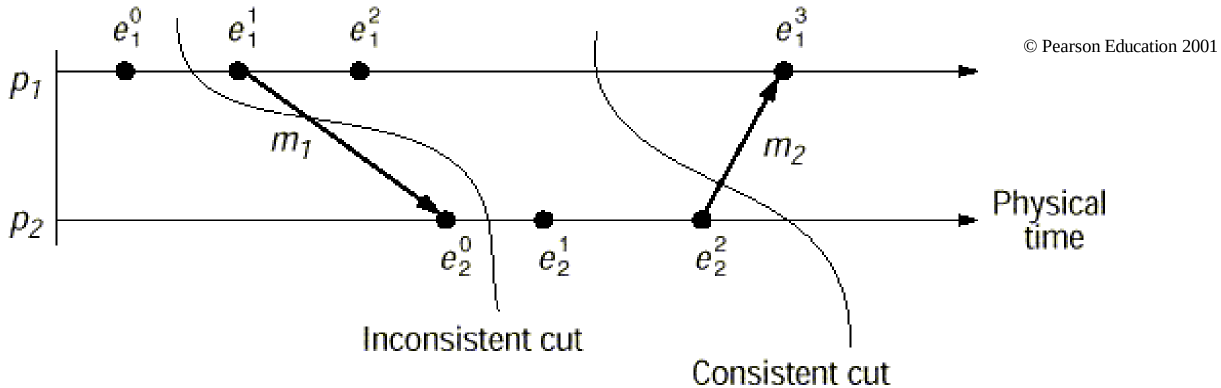
Formal Definition: Cut





- A **cut** is a tuple $C = (h_1^{c1}, h_2^{c2}, .., h_N^{cN})$ such that for all i, h_i^{ci} is a **prefix** of h_i , i.e. a sequence of all events with indices 0 to c_i
- The set of last events included in a cut C is called the **frontier of the cut**

Formal Definition: Consistent Cut

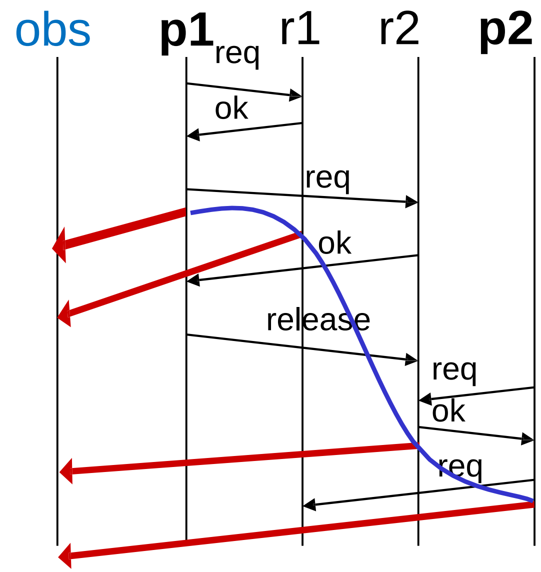


- A a cut C is **consistent** if for all events e and e' holds:

$$(e \in C) \; and \; (e' \rightarrow e) \Rightarrow e' \in C$$

- In other words, a consistent cut is left closed under the causal precedence relation →

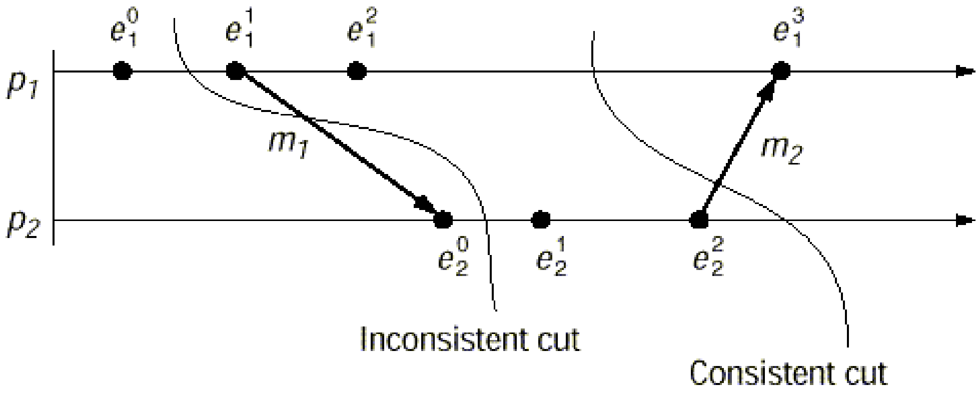
Deadlock Example



- Graphically: if all arrows that intersect the cut have their bases to the left and heads to the right of it, then the cut is consistent

Consistent Global States

- In a given cut C, let s_i be the state for p_i immediately succeeding the last event of p_i in C
  - I.e. the state of p_i after the frontier-event of p_i in C
- We call tuple $S = (s_1, s_2, .., s_N)$ a **global system state**
- A **consistent** global system state is one that corresponds to a consistent cut



Thank you.

Additional Slides