DoC 437 - 2009

Distributed Algorithms

Part 7: Logical Time

Asynchronous logical time

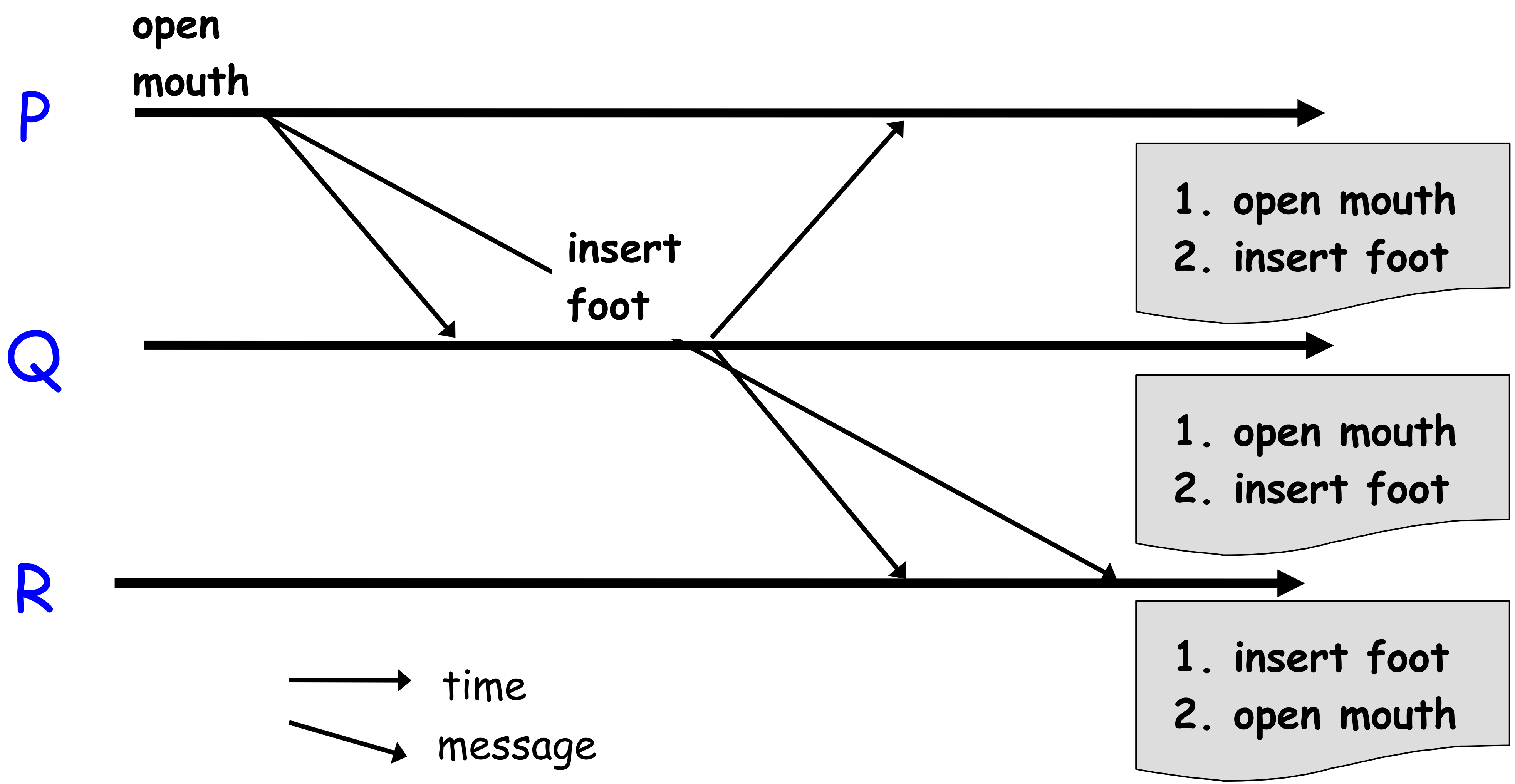
- In a distributed system, it is often necessary to establish relationships among events occurring at different processes

  was event *a* in process *P* responsible for causing event *b* in process *Q*?

  is event *a* in process *P* unrelated to event *b* in process *Q*?

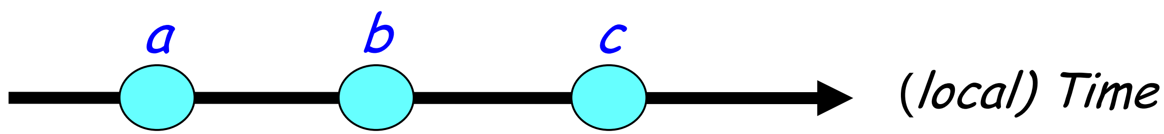
- We discuss the partial ordering relation "*happened before*" defined over the set of events

Example: email



Assumptions

- Processes communicate only via messages
- Events of each individual process form a totally ordered sequence



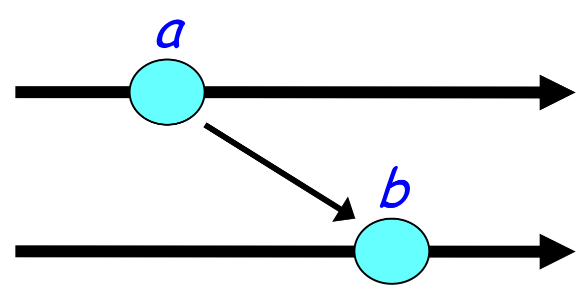
- Sending and receiving messages are the only events of interest

The “happens before” relation $\rightarrow$

- The relation $\rightarrow$ on the set of events satisfies the following conditions…

  if a and b are events in the same process, and a comes before b, then $a \rightarrow b$

  if a is the sending of a message by one process and b is the receipt of the same message by another, then $a \rightarrow b$



  if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$ (transitivity)

Concurrent, partial order of events

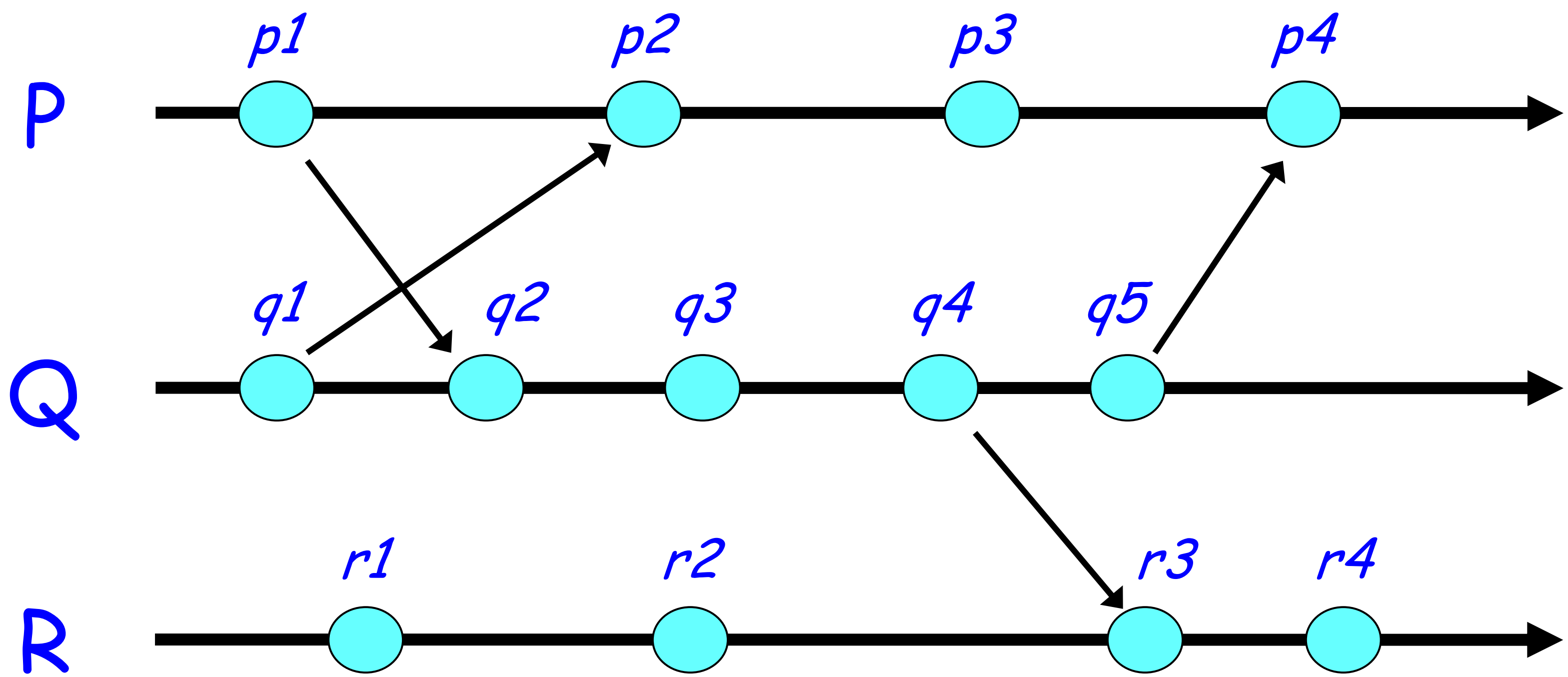
- Two distinct events a and b are said to be *concurrent* $(a \,||\, b)$ if $\neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$

  also termed "*independent*"

- The relation $\rightarrow$ defines a *partial order* over the set of events, since some events may be unrelated by $\rightarrow$

- The relation $a \rightarrow b$ means that it is possible for event a to *causally affect* event b

Space/time diagram



$a \rightarrow b$: path from a to b in the diagram, moving forward in time along the process and message lines

$p1 \rightarrow r4$ $\quad q4 \rightarrow r3$ $\quad p2 \rightarrow p4$ $\quad q3 \,||\, p3$ $\quad q3 \,||\, r2$

Logical clocks: assigning numbers

- A clock C_i for each process P_i is a function that assigns a number (a "time") $C_i(a)$ to event a in P_i

- The entire system of clocks is represented by the function C that assigns to any event b the number $C(b)$, where $C(b) = C_j(b)$ if b is an event in P_j

- The general clock condition

  for any events a, b: if $a \to b$ then $C(a) < C(b)$

Satisfying the clock condition

- The clock condition is satisfied by…

  CL1: if a and b are events in P_i and $a \rightarrow b$, then $C_i(a) < C_i(b)$

  CL2: if a is the sending of a message by P_i and b is the receipt of that message by P_j, then $C_i(a) < C_j(b)$

- Holds for any pair of events in any two processes

For any events a, b: if $C(a) < C(b)$, does $a \rightarrow b$?

Implementing logical clocks

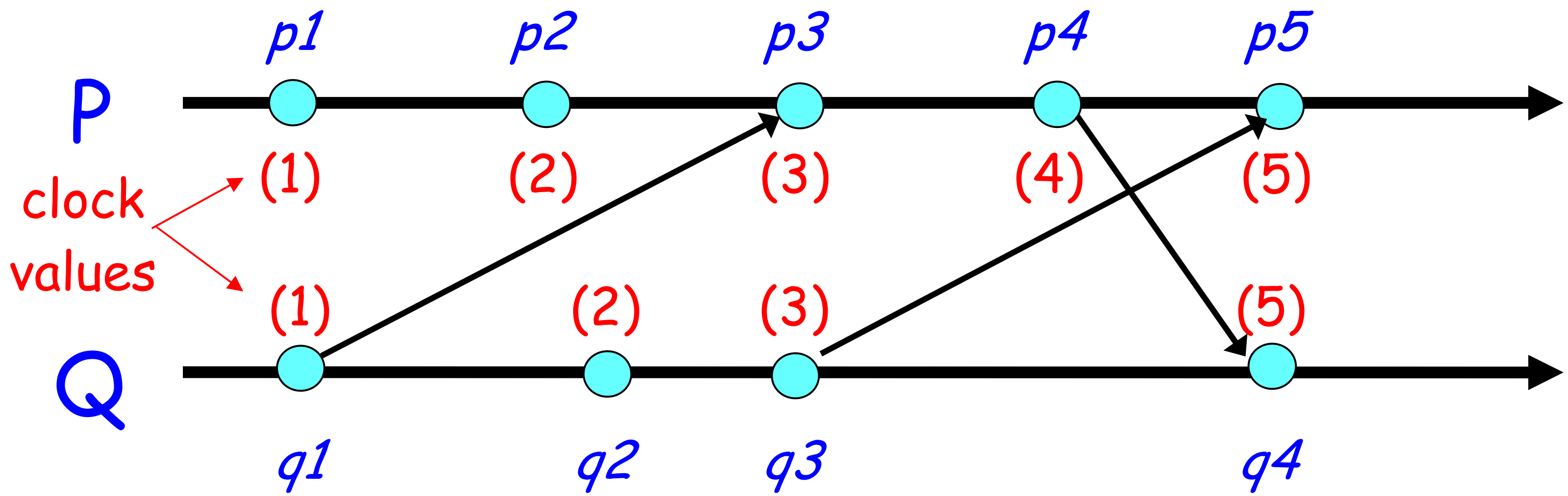
- Implementation rules…

  IR1: each process P_i increments C_i immediately after the occurrence of a local event

  IR2: if a is an event representing the sending of a message m by P_i to P_j, then m contains the timestamp $T_m = C_i(a)$

  IR3: if b is an event representing the receiving of a message m by P_j, then $C_j(b) := \max(C_j, T_m+1)$

Logical clocks lead to virtual time



- Virtual time, as implemented by logical clocks, advances with the occurrence of events and is therefore *discrete*
- If no events occur, virtual time stops

The total time order relation $\Rightarrow$

- Logical clocks place events into a *partial order* consistent with causality
- Convenient to have a *total order* on events, so we use the process identifier to "break ties"

$a \Rightarrow b$ iff either

(i) $C_i(a) < C_j(b)$ or

(ii) $C_i(a) = C_j(b)$ and $P_i < P_j$

note: if $a \rightarrow b$ then $a \Rightarrow b$

Distributed mutual exclusion

- The basic model

  fixed set of processes share a resource and communicate using a fully connected asynchronous message passing network

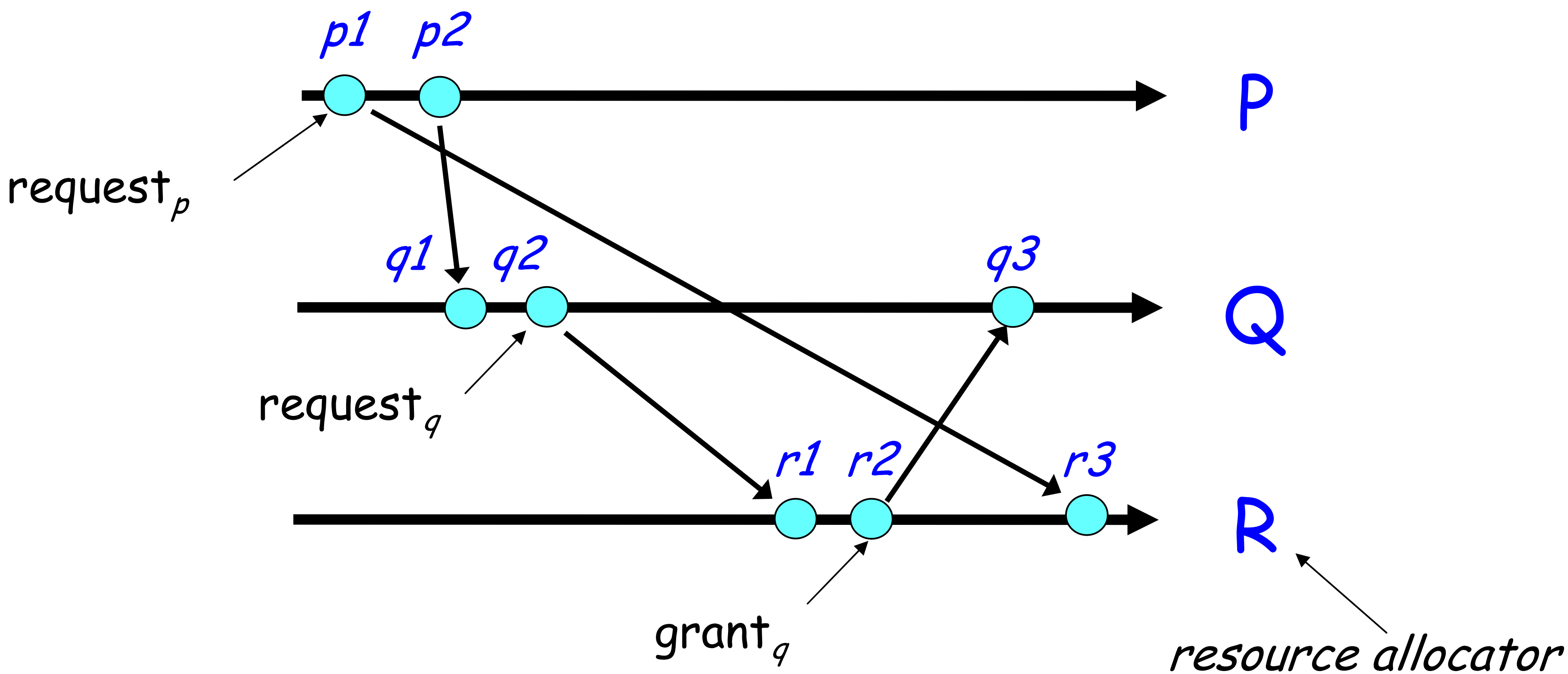
- The desired outcome

  mutual exclusion: granted process must release resource before resource granted to another

  ordered access: requests granted in order made

  no starvation: if every granted process eventually releases, then every request is eventually satisfied

Centralized solution



Which property does this scheme violate?

What is the significance of the p2/q1 events?

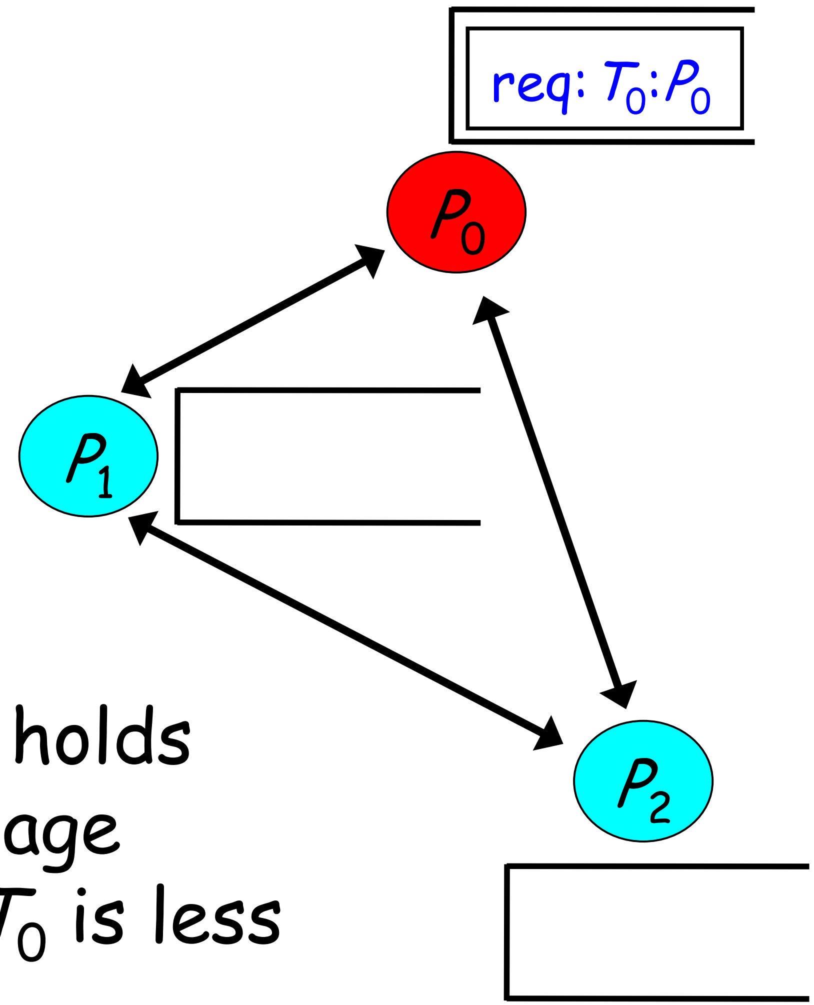
Decentralized solution using ⇒

- Assume FIFO channels between processes

- General scheme

  each process maintains its own request queue

  initially, queues are empty, except for P_0, which currently holds the resource and has the message [req: T_0:P_0], where timestamp T_0 is less than the value of any clock



Distributed mutual exclusion

decentralized ⇒ algorithm

1. Request at process P_i

   send [req: T_m:P_i] to every other process

2. Receipt of [req: T_m:P_i] at process P_j

   place in request queue and send [ack: $T_{\underline{m}'}$:P_j] to P_i

3. Release resource at process P_i

   remove any [req: T_m:P_i] from queue and send [rel: $T_{\underline{m}'}$:P_j] to every other process

Distributed mutual exclusion

decentralized ⇒ algorithm

4. Receipt of [rel: T_m:P_i] at process P_j

   remove any [req: T_x:P_i] from request queue

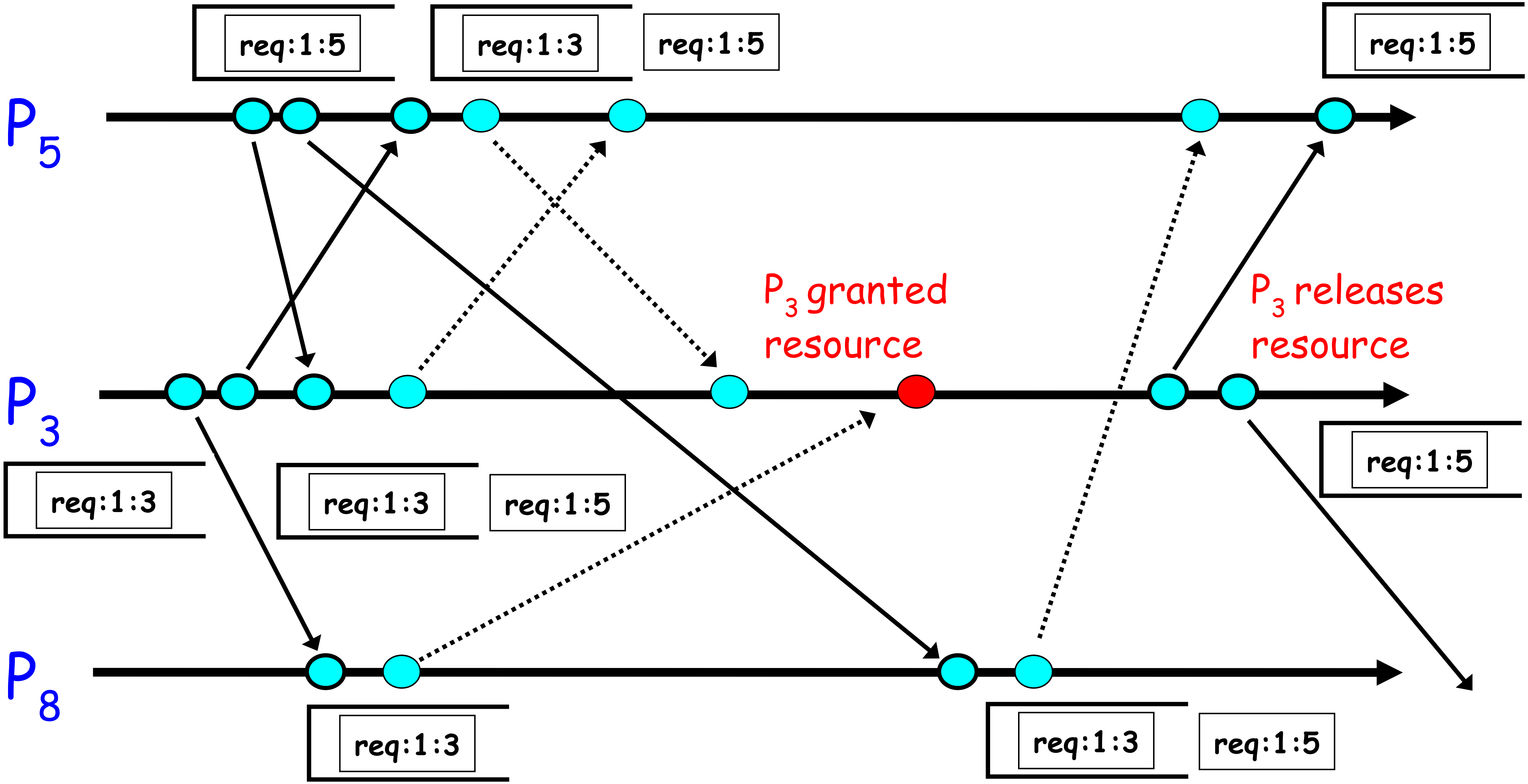
5. Grant the resource at process P_i if…

   i. [req: T_m:P_i] in the queue is "older" than any other request in the queue by relation ⇒ and

   ii. P_i has received from every other process a message time stamped "later" than T_m

Distributed mutual exclusion

example



Proof of mutual exclusion

- By contradiction

  assume *P* and *Q* have been granted the resource concurrently; 5(i) and 5(ii) must hold at both sites

  implies that at some instant *t*, both *P* and *Q* have their requests at the top of their respective queues

  assume *P*'s request has smaller timestamp than *Q*'s

  - by 5(ii) and FIFO property, at instant *t*, the request of *P* must be present in the queue of *Q*; since it has a smaller timestamp it must be at the top of *Q*'s queue

  - but by 5(i), *Q*'s request must be at the top of *Q*'s queue, so contradiction

Communication complexity

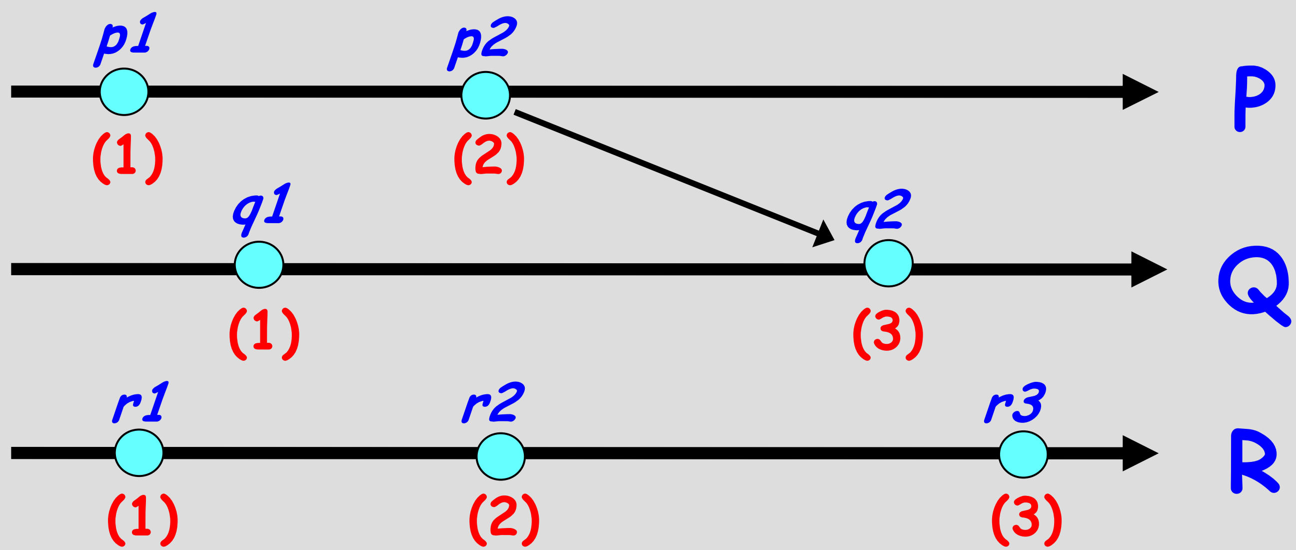
- Cycle of acquiring and releasing the shared resource

n-1 request messages + n-1 acknowledgment messages + n-1 release messages = 3(n-1)

Limitation of Lamport's basic clocks

- If $a \rightarrow b$ then $C(a) < C(b)$
- However, if a and b are in different processes, then it is not necessarily the case that if $C(a) < C(b)$ then $a \rightarrow b$

$C(p1) < C(q2)$
and
$C(p1) < C(r2)$
however
$p1 \rightarrow q2$
and
$p1 \,||\, r2$



Vector time

- Each P_i has a *vector* VC_i with entry for each P
- Implementation rules…

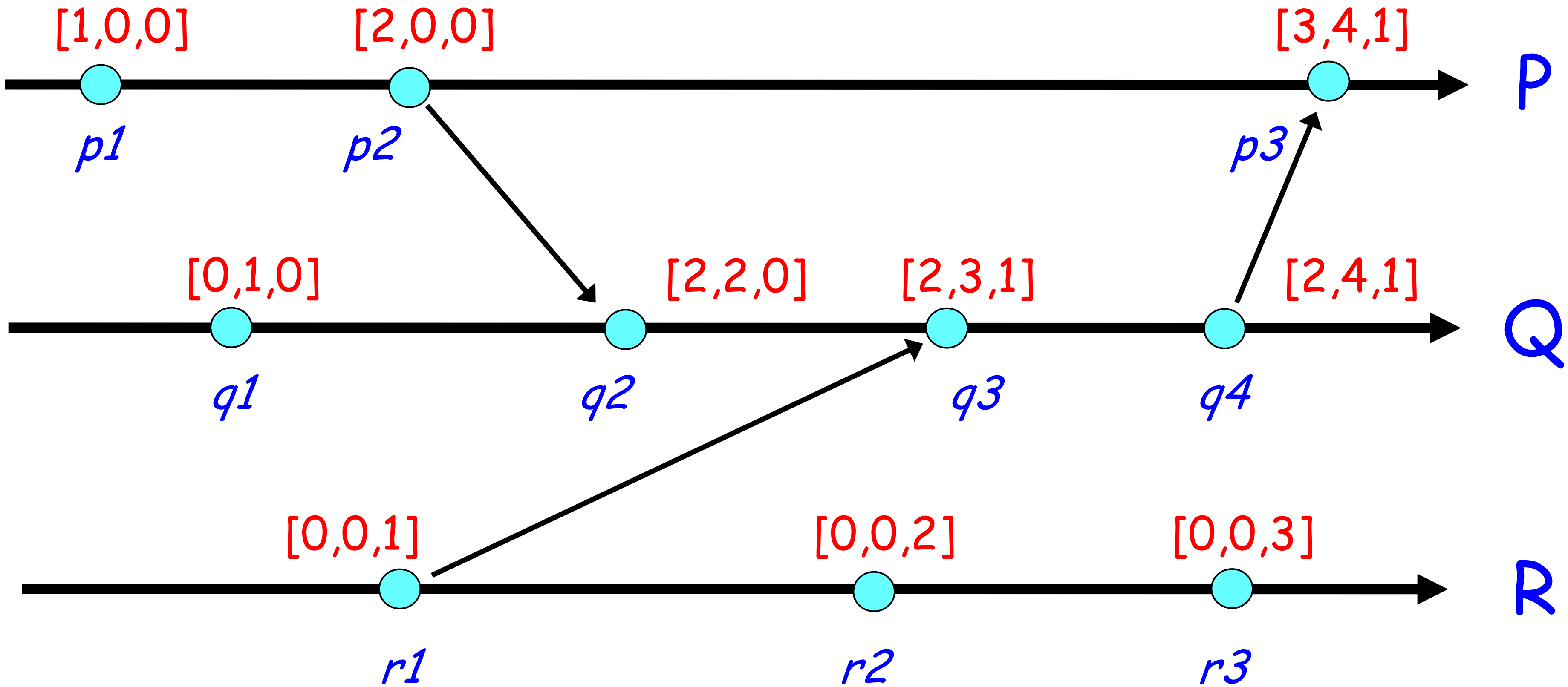
IR1: each process P_i increments $VC_i[i]$ immediately after the occurrence of a local event

IR2: if a is a send event in P_i for message m, then m contains the timestamp $VT_m = VC_i$ at a

IR3: if b is a receive event in P_j for message m, then $\forall K,\ VC_i[k] := \max(VC_j[k], VT_m[k])$

Vector time

example



Causally related events

- Vector timestamps represent *causality* precisely
- For two vector timestamps VT_a and VT_b ...

$$VT_a \neq VT_b \text{ iff } \exists i,\ VT_a[i] \neq VT_b[i]$$
$$VT_a \leq VT_b \text{ iff } \forall i,\ VT_a[i] \leq VT_b[i]$$
$$VT_a < VT_b \text{ iff } (VT_a \leq VT_b \text{ and } VT_a \neq VT_b)$$

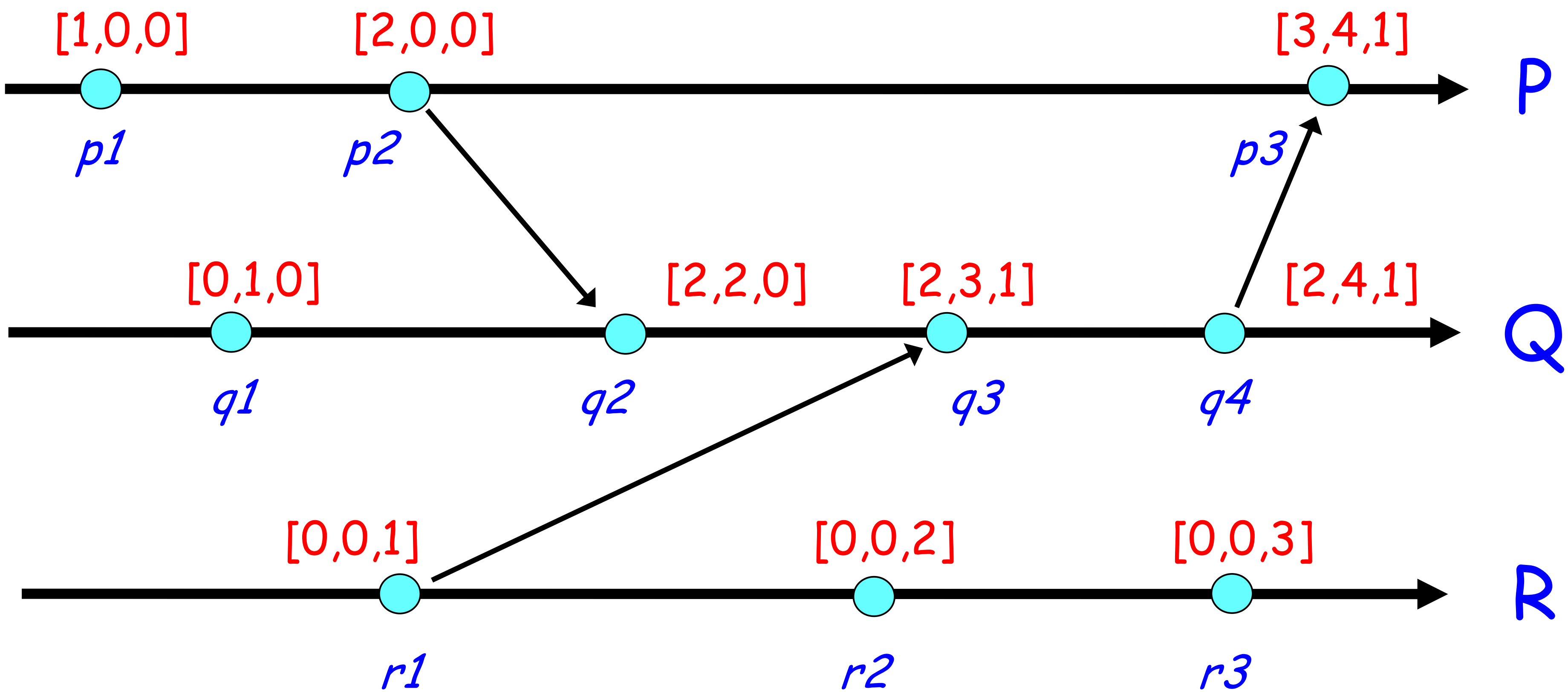
- Events a and b are causally related if...

$$VT_a < VT_b \text{ or } VT_b < VT_a$$

- So, $a \rightarrow b$ iff $VT_a < VT_b$

Causally related events

example



We can observe that *p1* || *r3*
since ¬[1,0,0] < [0,0,3]
and ¬[0,0,3] < [1,0,0]

We can also observe that *r1* → *p3*
since [0,0,1] < [3,4,1]

What's an example application?

- Distributed databases
- Goal is to apply updates to database replicas in the same order to preserve consistency
- Use causal ordering of update messages to recover the global order locally