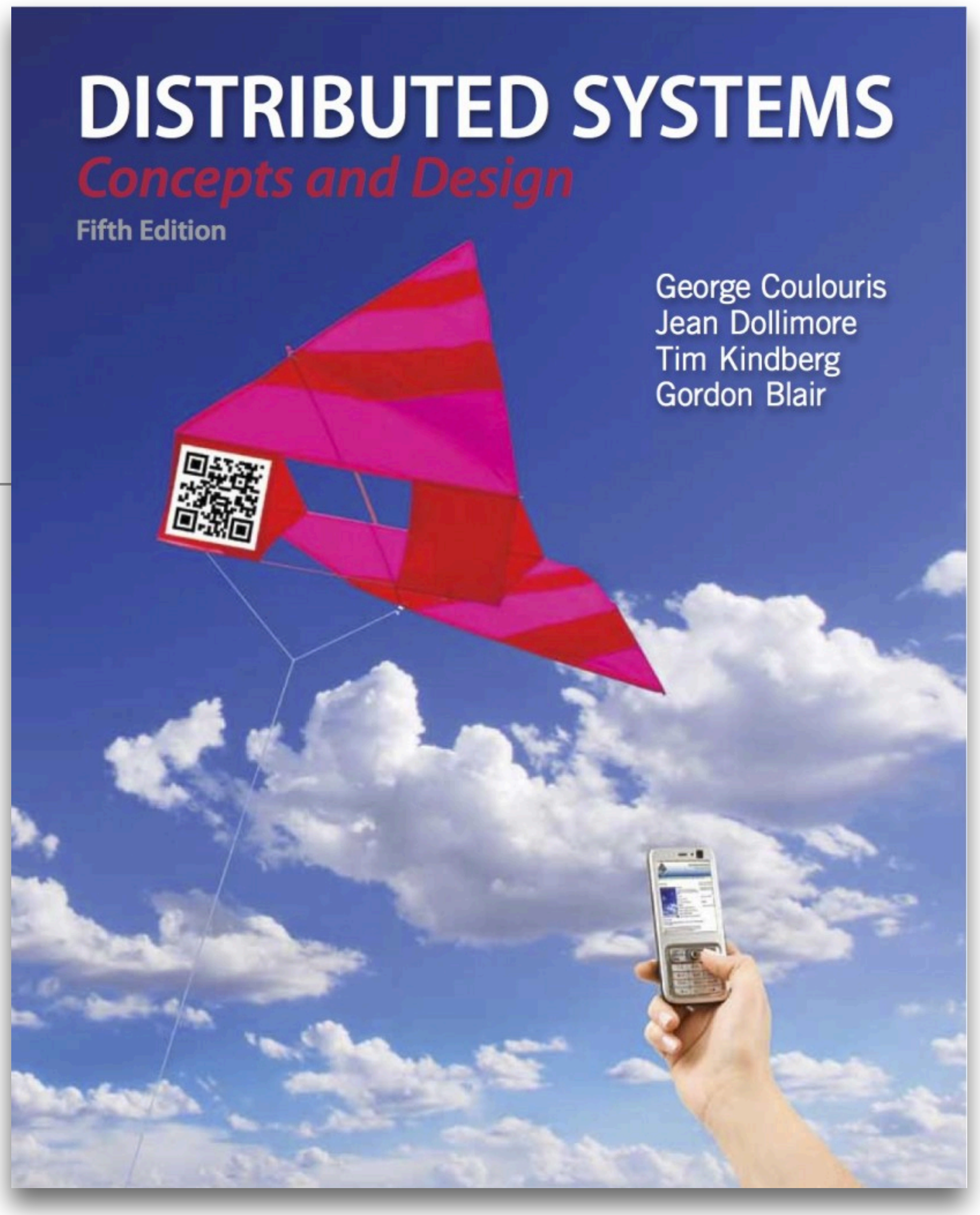
Distributed Systems 2023-2024:
Time, Coordination and Agreement

Wouter Joosen & Tom Van Cutsem
DistriNet KU Leuven
November 2023

KU LEUVEN DistriNet

Background reading



- CDK5 handbook
  - Chapter 14: sections 14.1, 14.2, 14.4
  - Chapter 15: section 15.2

- Recommended course notes by prof. Martin Kleppmann (Cambridge University):
  - https://www.cl.cam.ac.uk/teaching/2223/ConcDisSys/dist-sys-notes.pdf
  - Sections 3.3 and 4.1

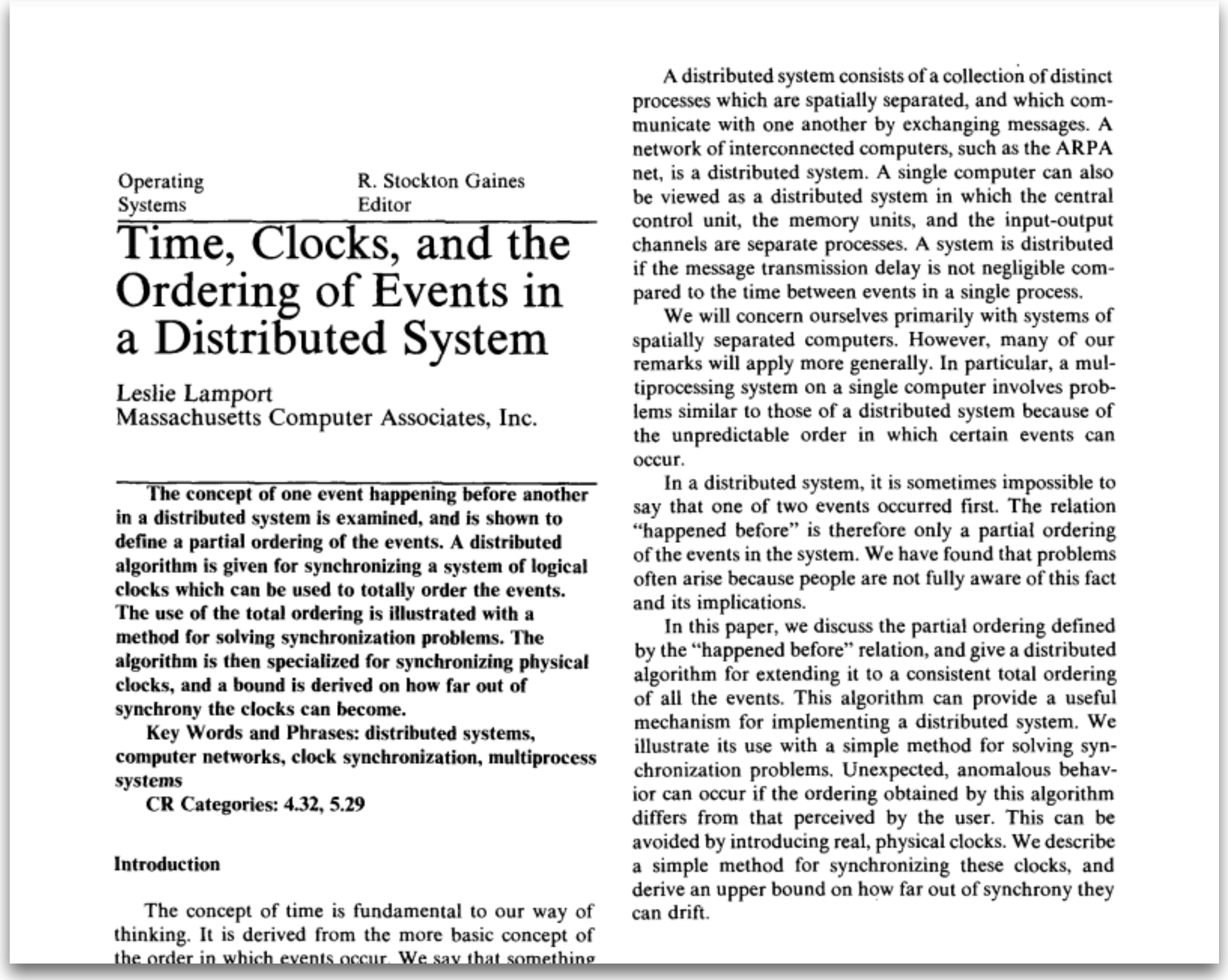
- Optional course notes by prof. Paul Krzyzanowski (Rutgers University):
  - Lamport and Vector clocks: https://people.cs.rutgers.edu/~pxk/417/notes/logical-clocks.html
  - Mutual exclusion: https://people.cs.rutgers.edu/~pxk/417/notes/mutex.html

Outline

- **Time, clocks, event ordering**: how to get processes to agree on the order of events, even if there is no shared global clock?
- **Distributed mutual exclusion**: how to get processes to agree on who has exclusive access to a resource, even in the absence of a central coordinator?
- **Group communication** and **reliable multicast**: how to get processes to agree on a set of messages to be delivered, even in the face of unreliable or slow network links?

Time, clocks and the ordering of events in a distributed system

A bit of history



Operating Systems

R. Stockton Gaines Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

Seminal 1978 distributed systems paper [1]

Leslie Lamport

[1] Leslie lamport: Time, Clocks and the Ordering of Events in a Distributed System, Communications of the ACM, 1978

Time and clocks

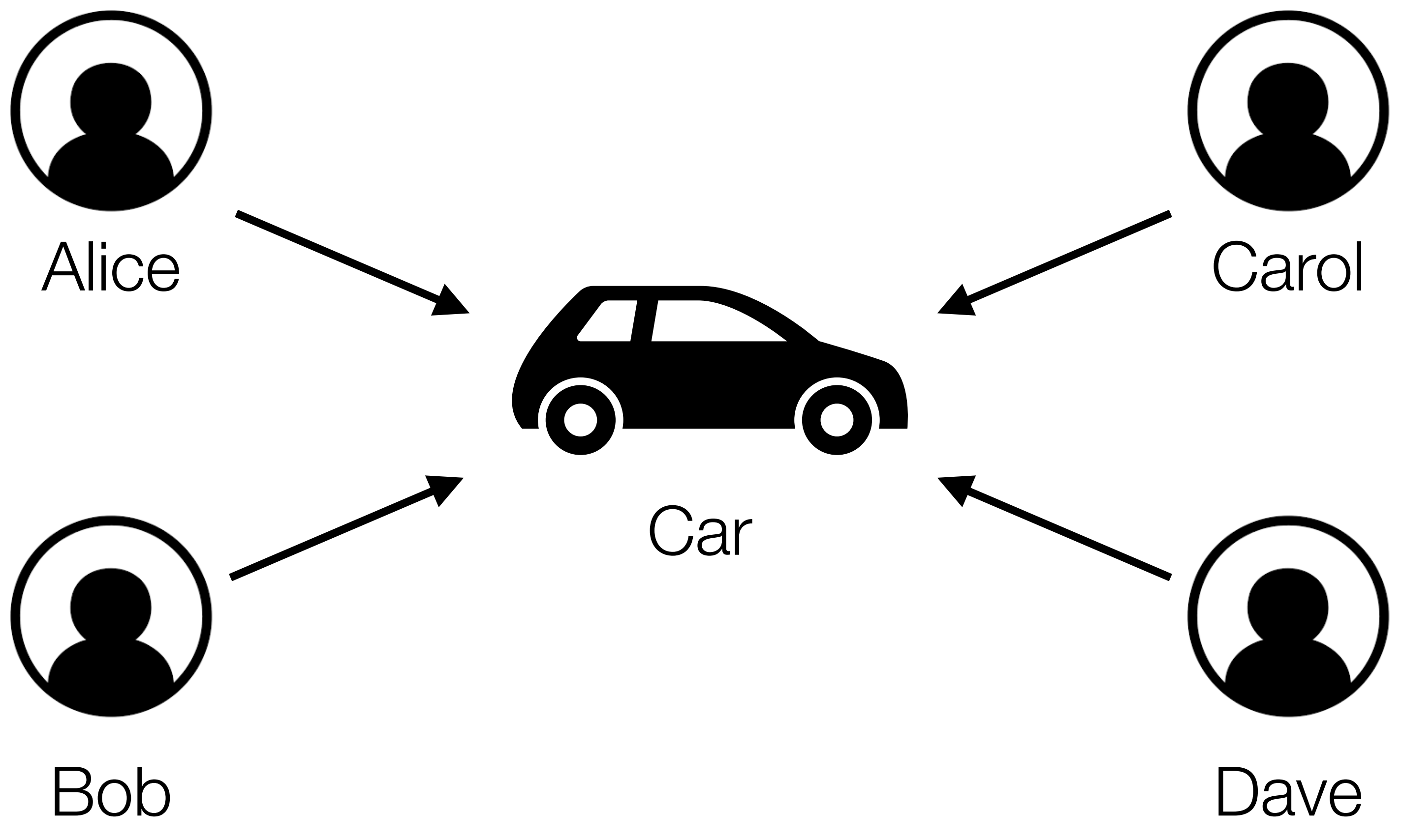
- Processes running on different computers produce events or issue commands
- We are often interested in *ordering* these events/commands in time

- Example: ordering concurrent requests to access a shared resource

Time and clocks

- Processes running on different computers produce events or issue commands
- We are often interested in *ordering* these events/commands in time

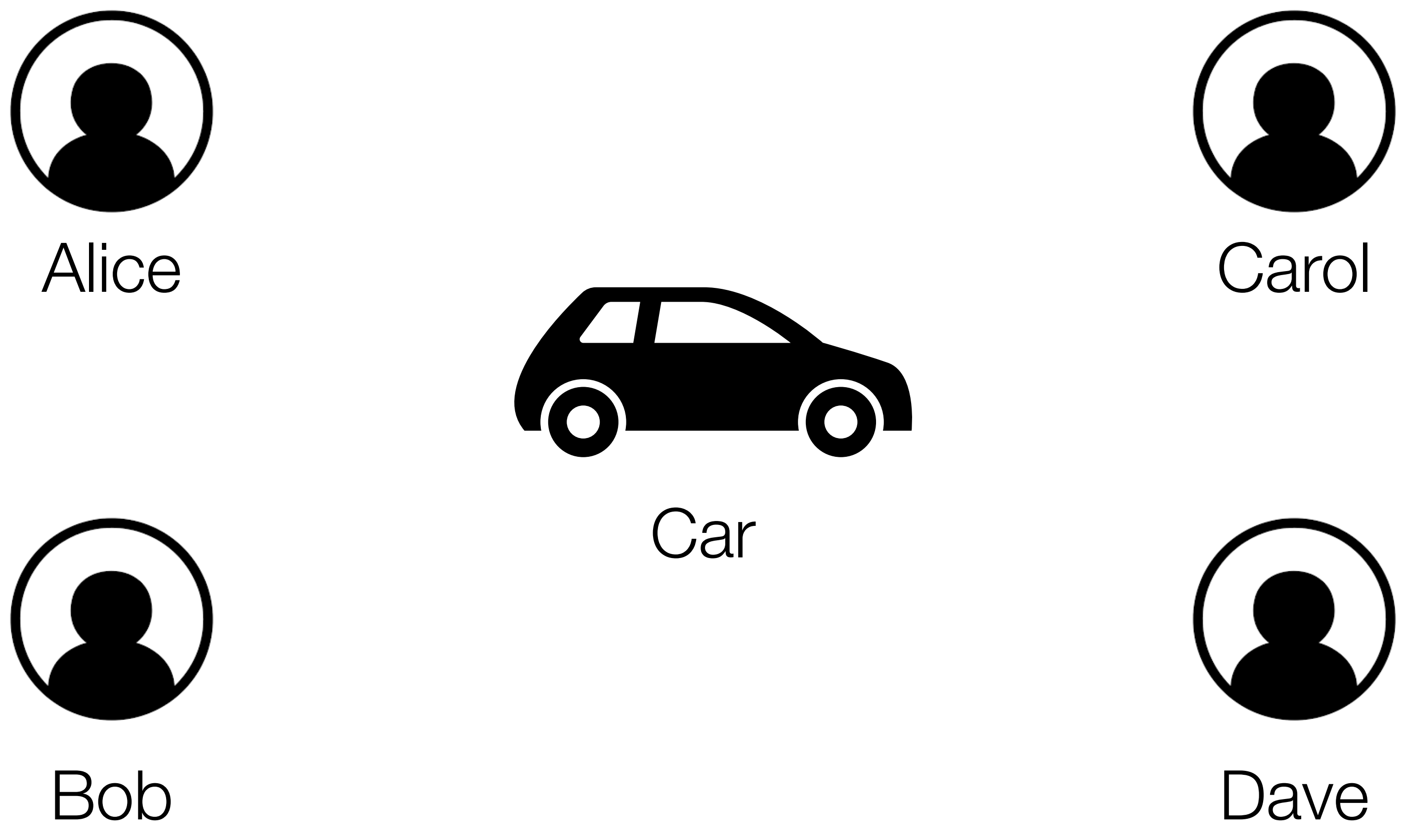
- Example: ordering concurrent requests to access a shared resource

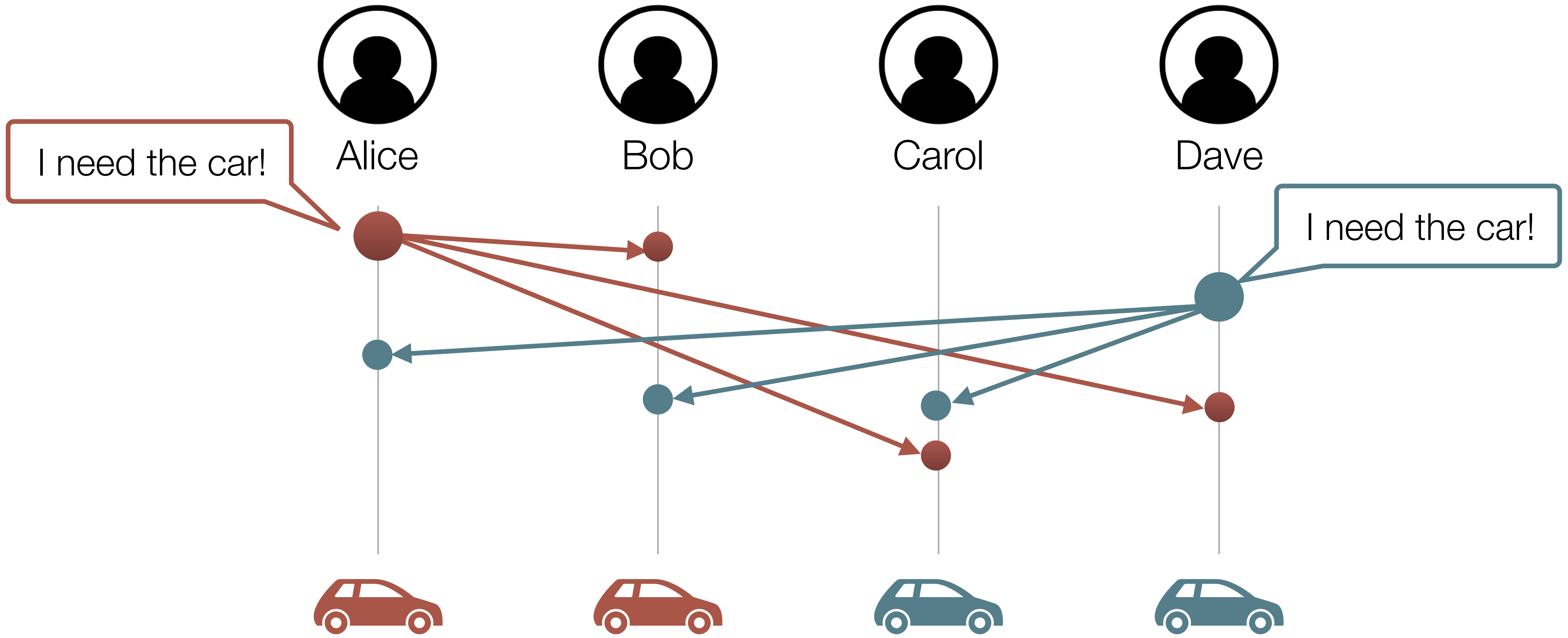


Time and clocks

- Processes running on different computers produce events or issue commands
- We are often interested in *ordering* these events/commands in time

- Example: ordering concurrent requests to access a shared resource





Who was first to reserve the car?

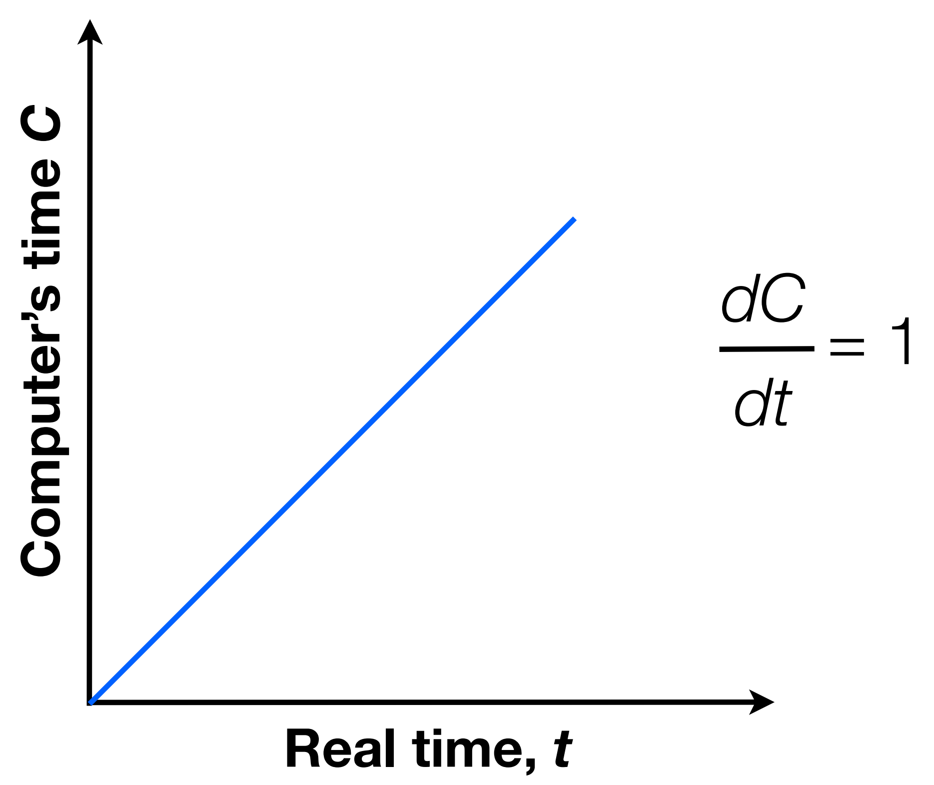
Physical and Logical clocks

- **Physical** clocks keep time of day
  - Count **seconds** (or some other unit) elapsed since a given time
  - Not consistent across systems:
    - clocks tick at different rates (clock drift)
    - difference between two clocks at one point in time (clock skew)
  - Physical time is useful for many things, but may be inconsistent with **causality**
- **Logical** clocks keep track of event ordering among (causally) related events
  - Count **events** that we are interested in
  - no relation to physical time except for their causal order

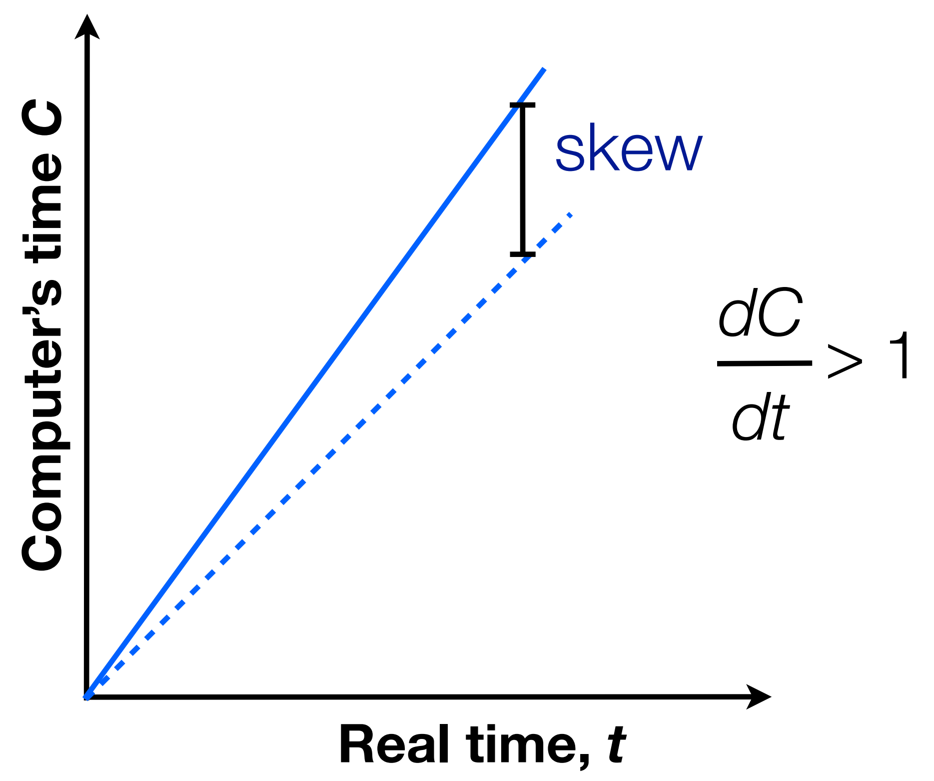
Physical clocks

- clocks tick at different rates (clock drift)
- difference between two clocks at one point in time (clock skew)

Perfect clock



Drift with fast clock



Physical clocks

- Key point: there is no default notion of “**global** time” in a distributed system
- There are only **local** clocks on each computer
- Protocols exist to synchronise computer clocks (e.g. Network Time Protocol, **NTP**)
- But such protocols can only synchronise clocks to within certain time **bounds** (e.g. within 10s of milliseconds). The synchronisation is **not absolute.**

Logical clocks

- Assign **sequence numbers** to events
- Allows cooperating processes to agree on the order of events, even if their physical clocks are not synchronized
- Assume no central time source
- Each system maintains its own, local, logical clock
- No notion of "happened *when*", only "happened *before* or *after*"
- No total event order, but a partial order

Happened-before

- Lamport's "happened-before" notation

$$a \rightarrow b$$

event a happened before event b

e.g. a is message being sent; b is receipt of the message

Transitive relationship:

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

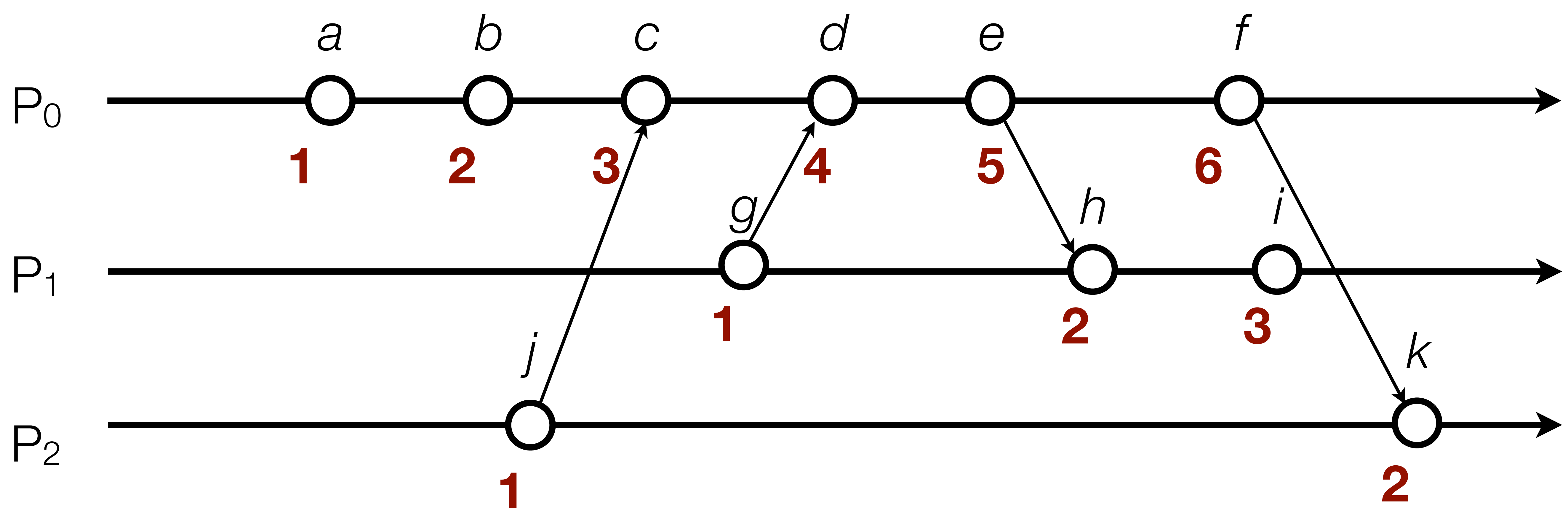
Logical clocks & concurrency

- Assign a "clock" value to each event (just a number)
  - If $a \rightarrow b$ then clock(a) < clock(b)
  - Time cannot run backwards
- If a and b occur on different processes that do not exchange messages, then neither $a \rightarrow b$ nor $b \rightarrow a$ are true
  - These events are said to be **concurrent**
  - Written as $a \parallel b$

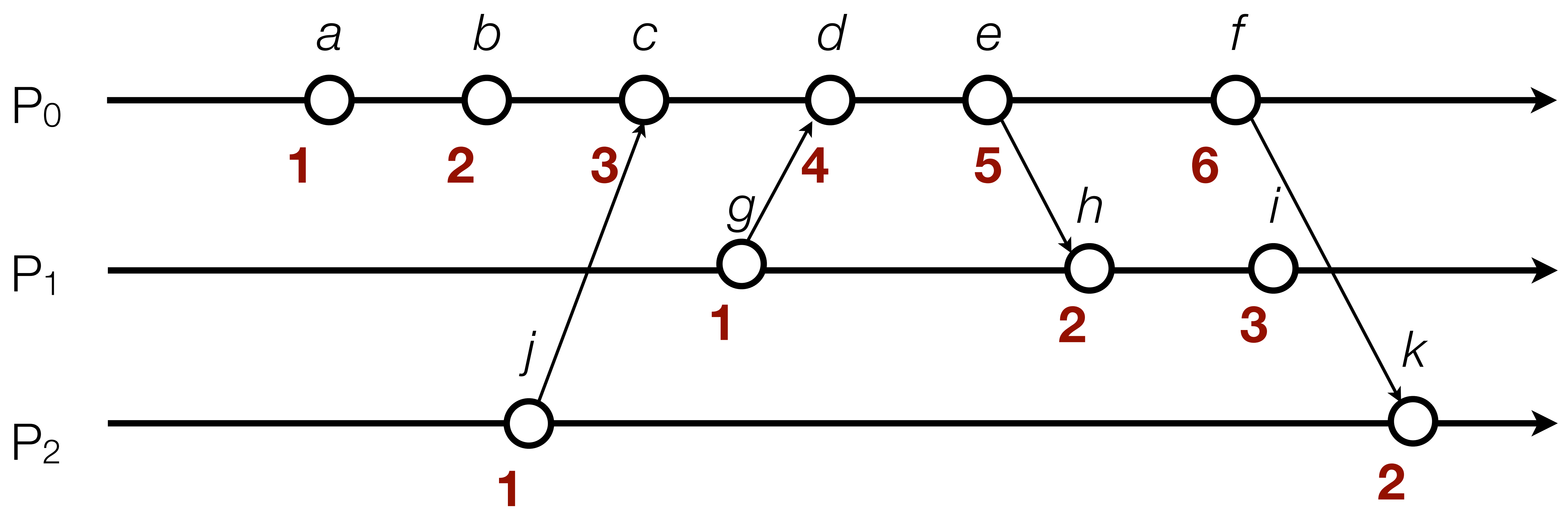
Event counting example

- Three processes P_0, P_1, P_2
- Events labeled *a*, *b*, *c*, ...
- Local event counter in each process
- Processes occasionally communicate (exchange messages)

Event counting example



Event counting example



Local counters are not consistent with causality:

$e \rightarrow h$ but $5 \nless 2$
$f \rightarrow k$ but $6 \nless 2$

Lamport clocks (in code)

on initialisation **do**
    $t := 0$ ▷ each node has its own local variable t
end on

on any event occurring at the local node **do**
    $t := t + 1$
end on

on request to send message m **do**
    $t := t + 1$; send (t, m) via the underlying network link
end on

on receiving (t', m) via the underlying network link **do**
    $t := \max(t, t') + 1$
    deliver m to the application
end on

Lamport clocks (in words)

- Each process has a local clock *t*, which is just a monotonically increasing integer *counter*

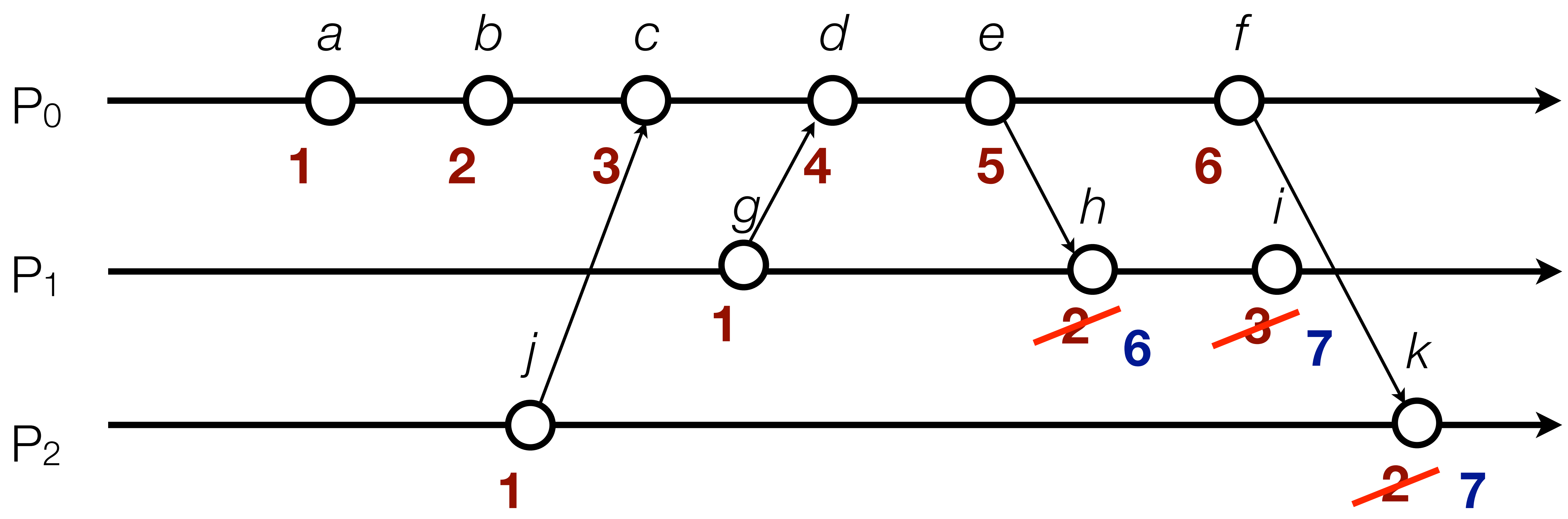
- The clock is ticked between any two local *events* occurring in the local process (events are the activities that we would like to order in time)

- When a process sends a message, it includes *t* as a logical timestamp in the message

- When a process receives a timestamped message, if its own clock < the message timestamp, the process advances its local clock to the message timestamp, then ticks the clock (increments by 1) to count the event of “receiving” the message.

Event counting example

- Applying Lamport's algorithm:



Lamport's algorithm

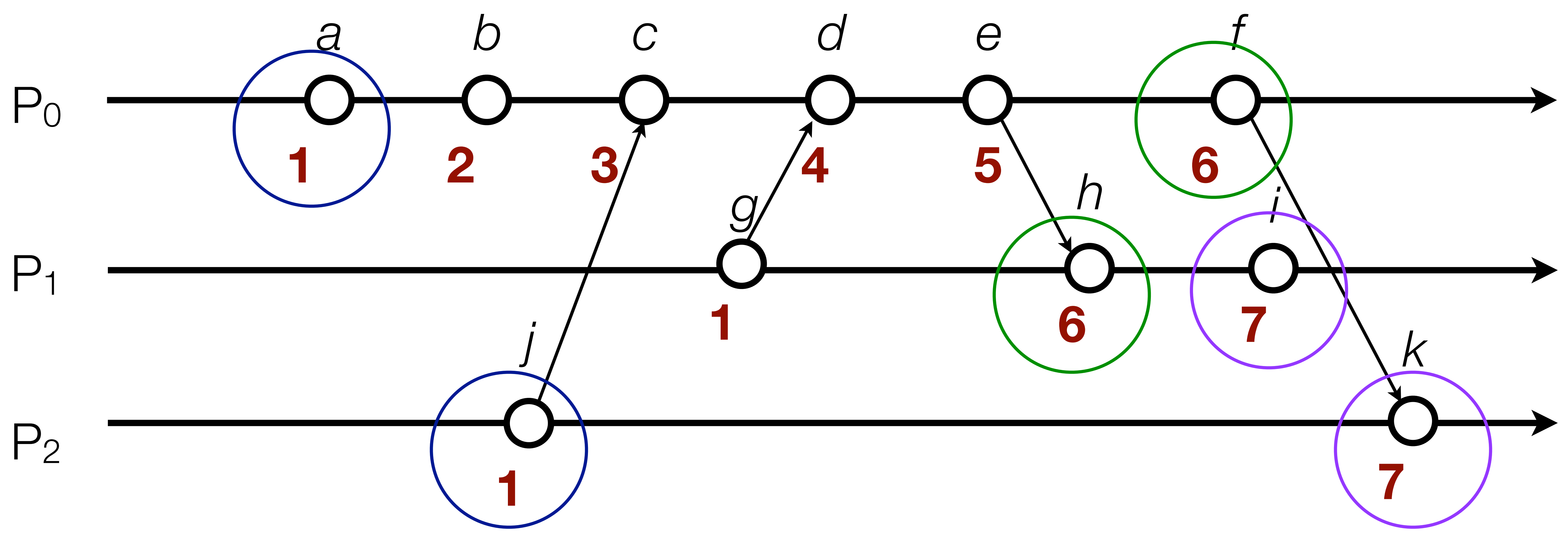
- Algorithm allows us to maintain time ordering among related events
  - **Partial ordering**

Summary: Lamport clocks

- Algorithm needs monotonically increasing software counter
- Incremented (at least) whenever an event needs to be timestamped
- Each event e has a Lamport timestamp $L(e)$ attached to it
- For any two events, if $a \rightarrow b$ then $L(a) < L(b)$

Problem: identical timestamps

- may cause confusion if processes all need to make consistent decisions based on the timestamps of two events



a → *b, b* → *c*, ...: local events sequenced

i ↛ *c, f* ↛ *d*, *d* ↛ *g*, ...: Lamport imposes a send → receive relationship

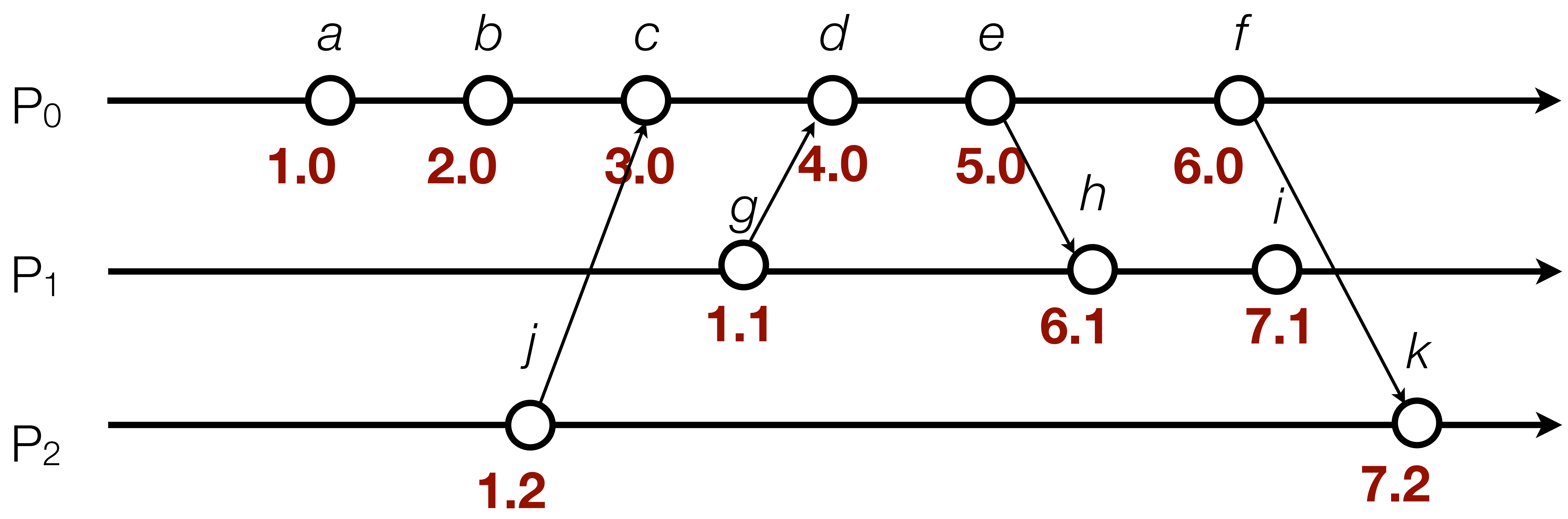
Concurrent events (e.g. *b* & *g*, *i* & *k*) <u>*may*</u> have the same timestamp... or not

Unique timestamps (total ordering of events)

- We can force each timestamp to be unique
- Define global logical timestamp (T_i, *i*)
  - T_i represents local Lamport timestamp
  - *i* represents process number (globally unique)
    - e.g. *i* = host address + process ID
  - To compare timestamps:

```
(Ti, i) < (Tj, j) <=>
 Ti < Tj or
 Ti = Tj and i < j
```

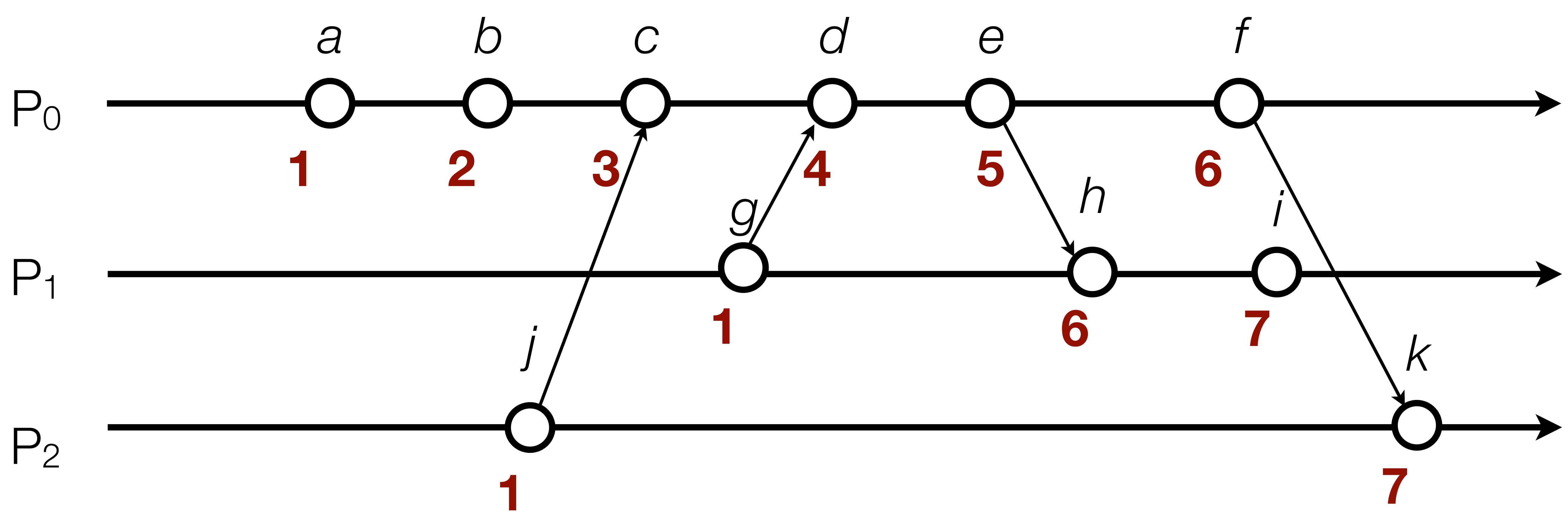
Unique (totally ordered) timestamps



(Notation: **1.0** *means* T_i *= 1, i = 0)*

Problem: detecting causal relations

- Lamport clocks: if $e \rightarrow e'$ then $L(e) < L(e')$
- But, if $L(e) < L(e')$, we cannot conclude that $e \rightarrow e'$
  - E.g. $L(j) < L(b)$ but $j \nrightarrow b$



Problem: detecting causal relations

- By looking only at Lamport timestamps, we cannot conclude which events are causally related and which are not

- Solution: use a vector clock (see later)

Summary so far: 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

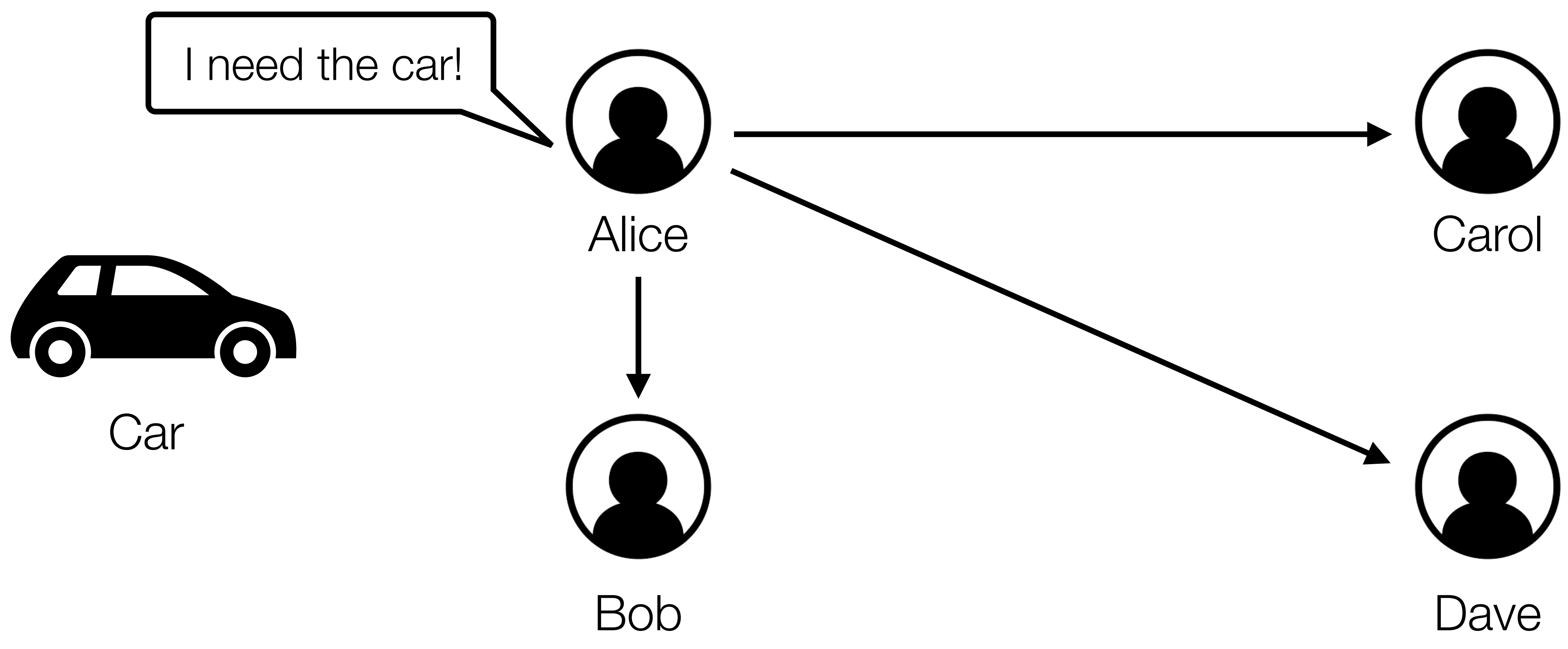
Why is establishing a total order on events useful?

- One example: guarantee fairness in Distributed Mutual Exclusion
- Distributed algorithm proposed by Leslie Lamport in 1978

Distributed Mutual Exclusion

What is distributed mutual exclusion?

- A set of distributed processes (clients) request exclusive access to a resource
- Like multiple threads trying to acquire a lock in a multi-threaded program
- But in a fully distributed setting: assume no shared memory, no shared clock
- Must use explicit message passing

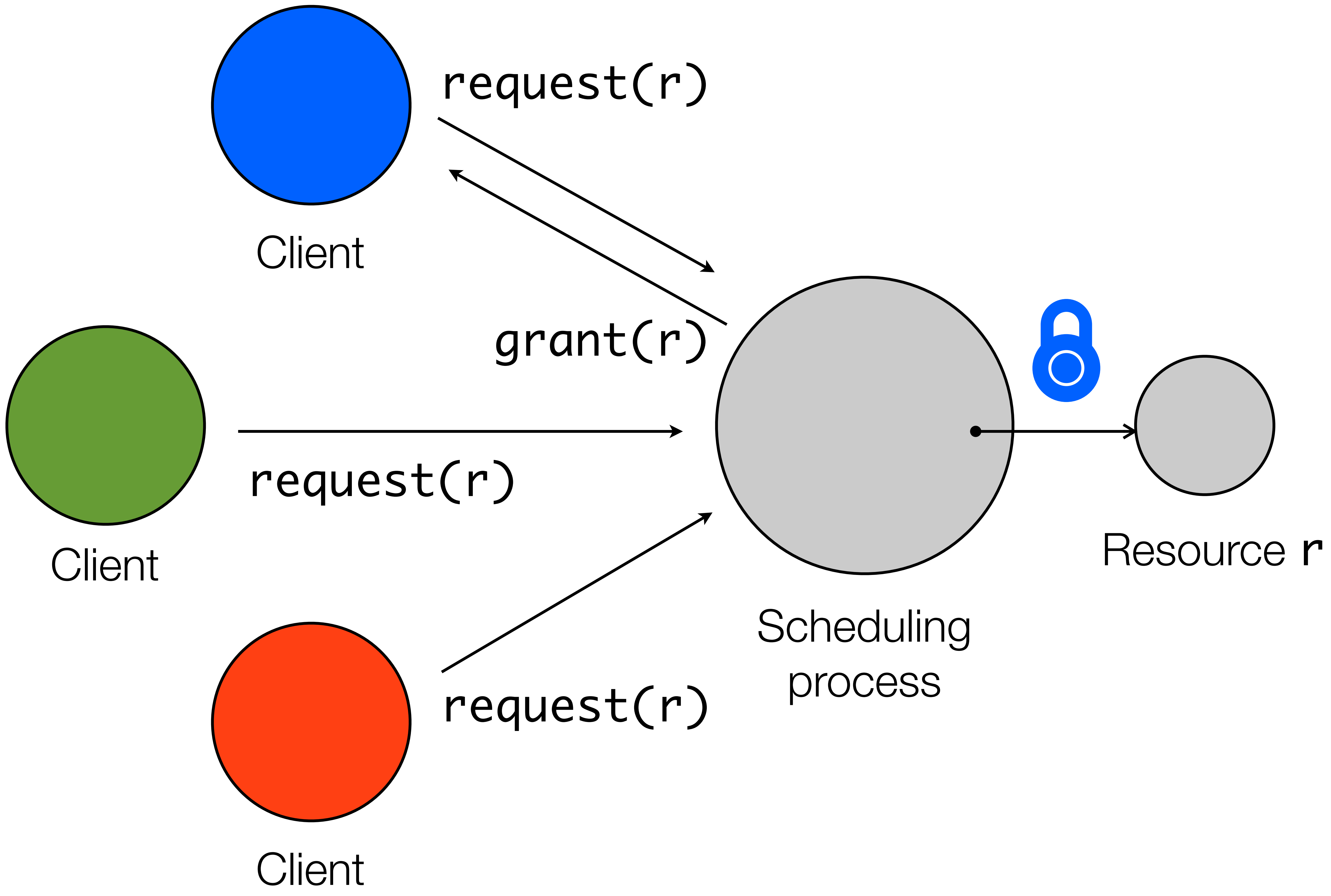


Distributed Mutual Exclusion: desirable properties

- **Safety**: at most one process may access the shared resource at a time
  - No concurrent access allowed in the “critical section”
- **Liveness**: requests to acquire or release the resource eventually succeed
  - No livelocks (retry forever) or deadlocks (wait forever)
- **Fairness**: requests to acquire the resource are granted in happened-before order
  - This is where Lamport clocks will be useful

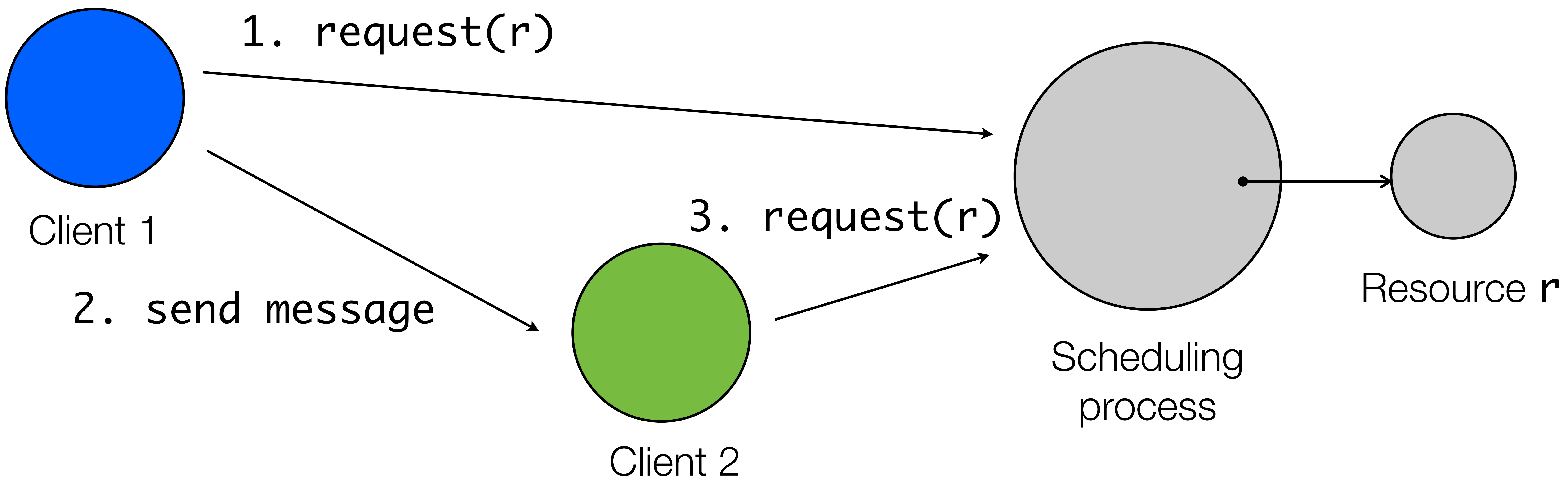
Distributed Mutual Exclusion: Central Coordinator

- One process protects resource and ensures only one client can use the resource at a time
- Benefit: simple



Central Coordinator: Problems

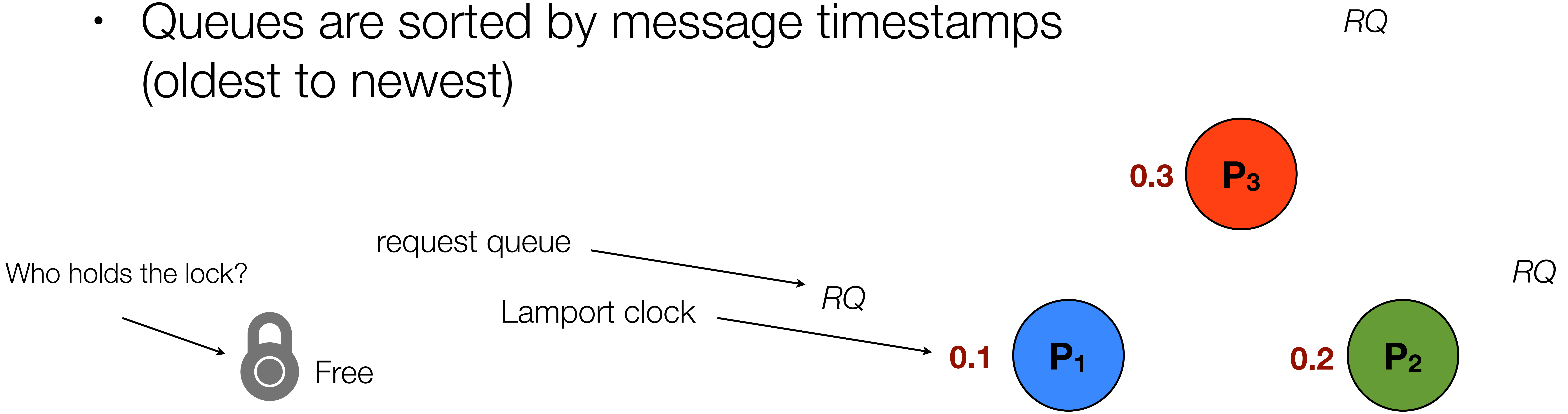
- Single scheduler process: bottleneck
- Grants requests in the order in which they *arrived* (FIFO), not necessarily in the order in which they were initially *sent*. For example:



Client 2's request may arrive before Client 1's request

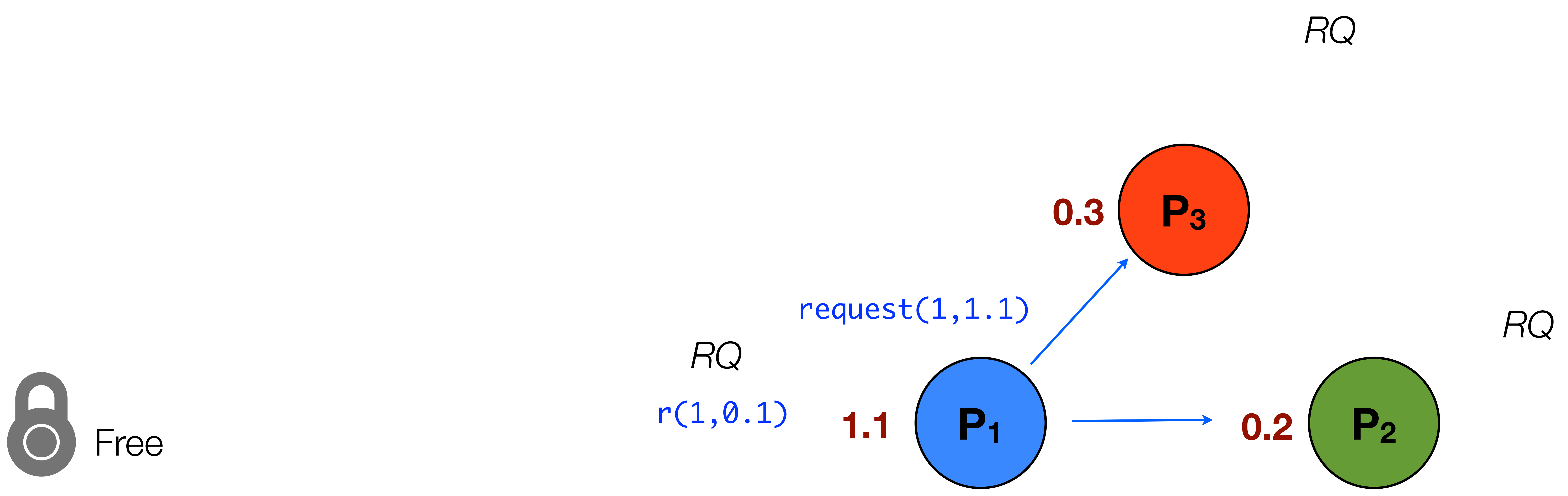
Lamport's mutual exclusion algorithm

- Distributed algorithm, no central scheduling process, only clients
- Each process maintains a local request queue *RQ*
  - *RQ* contains mutual exclusion requests
  - Queues are sorted by message timestamps (oldest to newest)



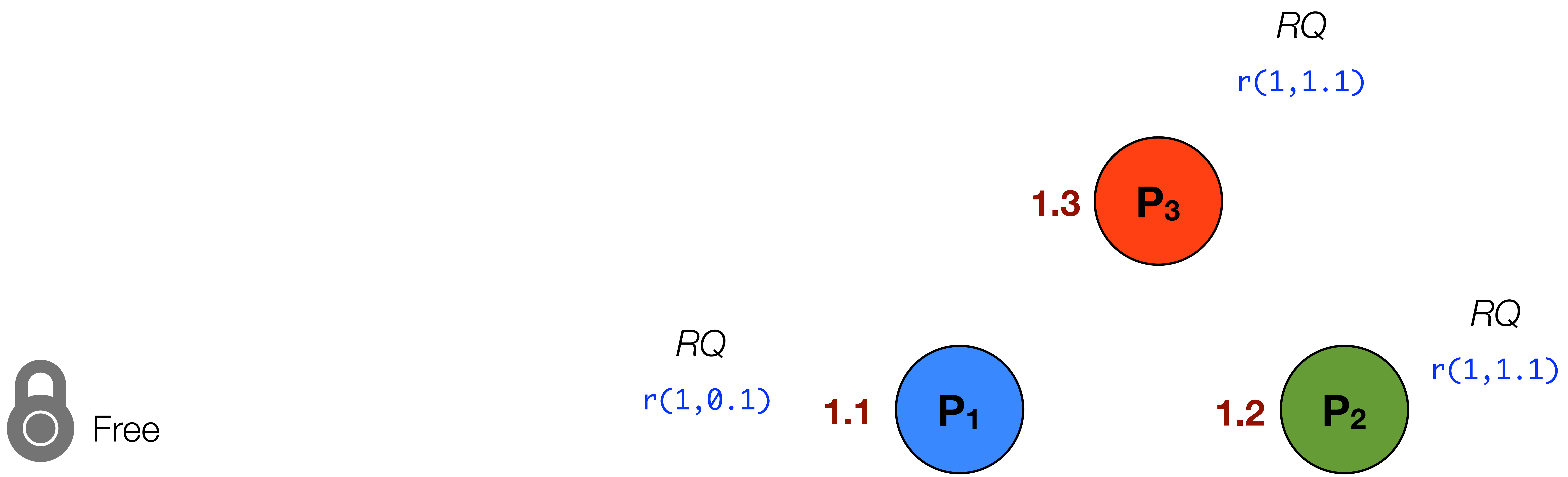
Lamport's mutual exclusion algorithm

- For P_i to request access to the resource:
  - P_i sends a `request(i, Ti)` message to all nodes, and places the request in its own queue (T_i = lamport timestamp)



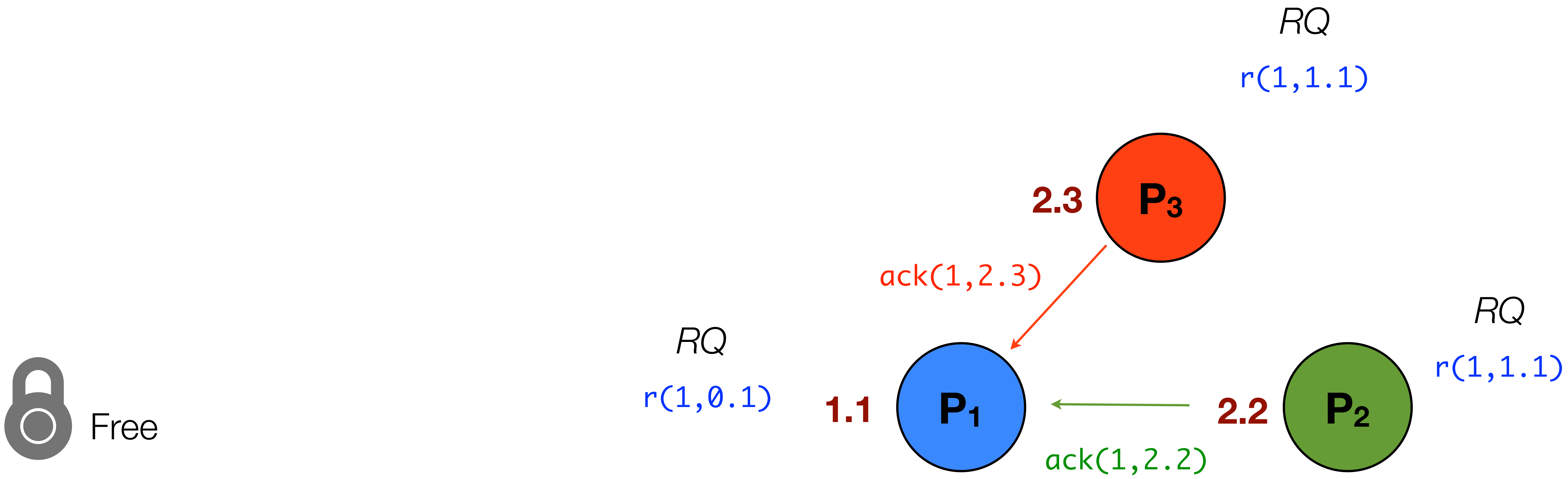
Lamport’s mutual exclusion algorithm

- When a process P_j receives such a request:
  - It replies immediately with a timestamped ack message and places the request in its queue



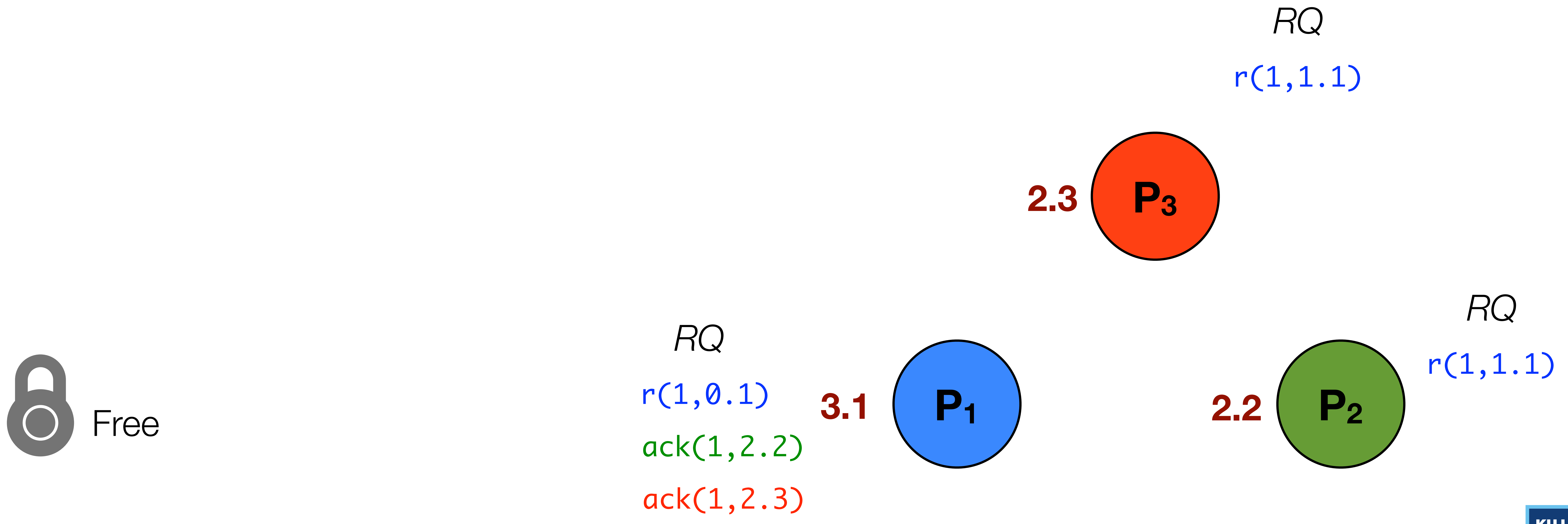
Lamport's mutual exclusion algorithm

- When a process P_j receives such a request:
  - It replies immediately with a timestamped ack message and places the request in its queue



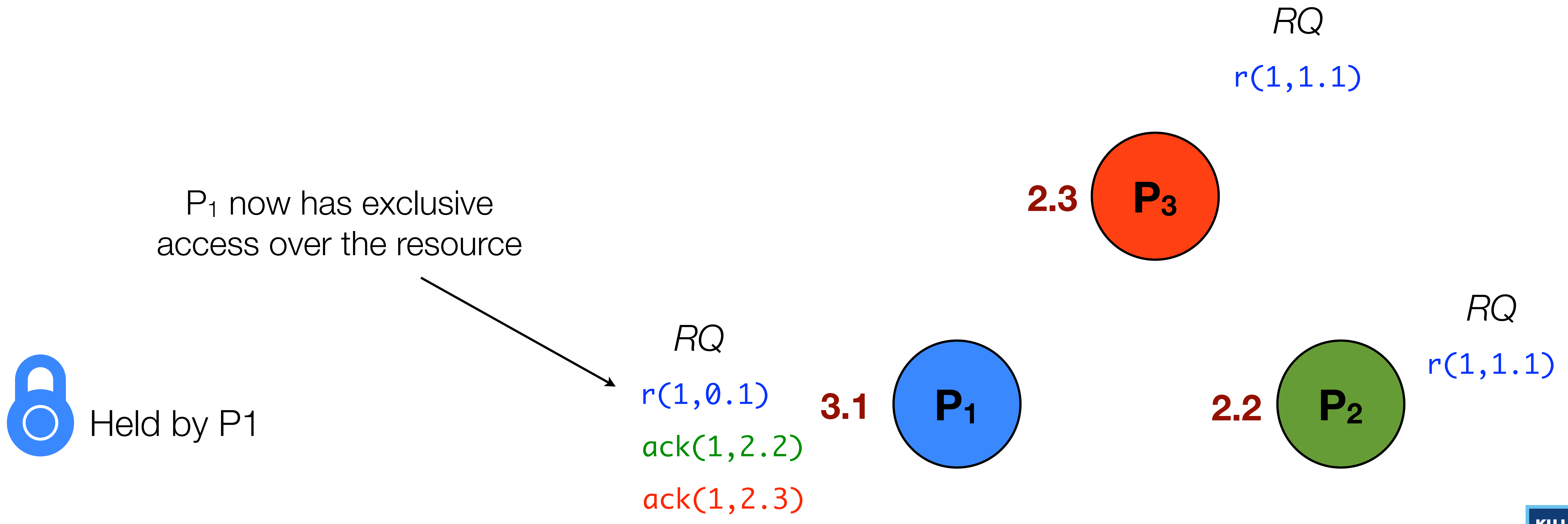
Lamport’s mutual exclusion algorithm

- When a process P_j receives such a request:
  - It replies immediately with a timestamped ack message and places the request in its queue



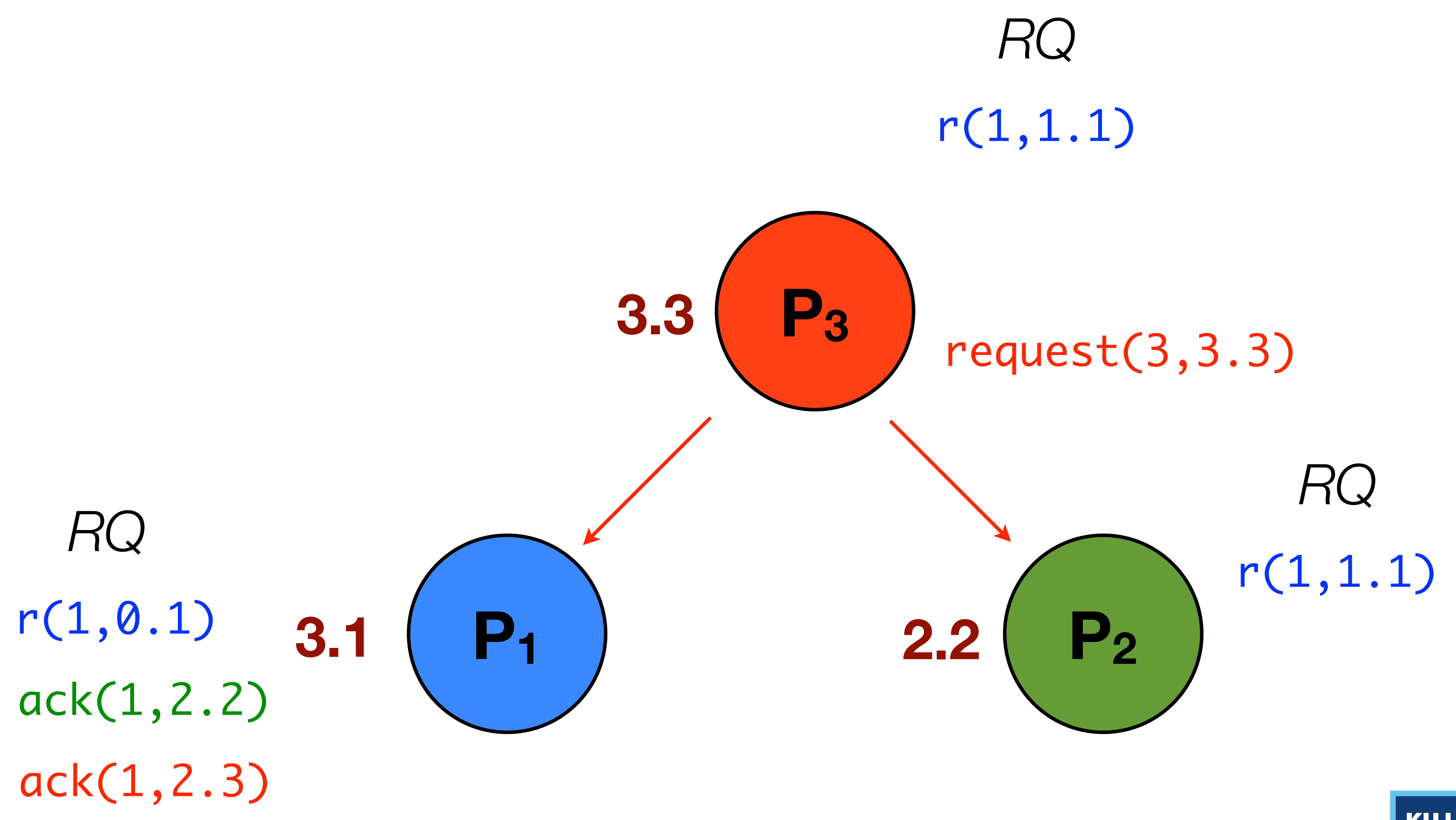
Lamport's mutual exclusion algorithm

- For P_i to acquire the resource (enter the critical section), two conditions must hold:
  - P_i has received **all** replies to its `request` message
  - P_i's own `request` message has **the earliest** timestamp in its queue



Lamport’s mutual exclusion algorithm

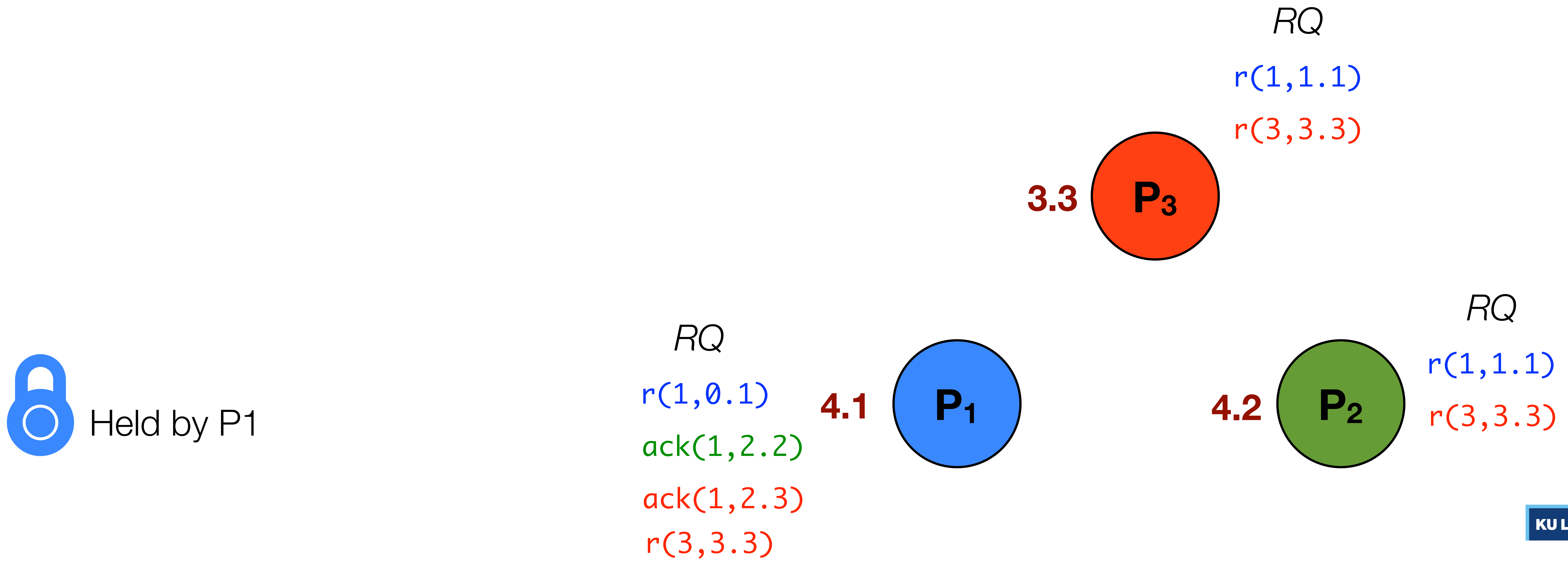
- Now P_3 wants to acquire the resource while P_1 still holds the lock:



Held by P1

Lamport's mutual exclusion algorithm

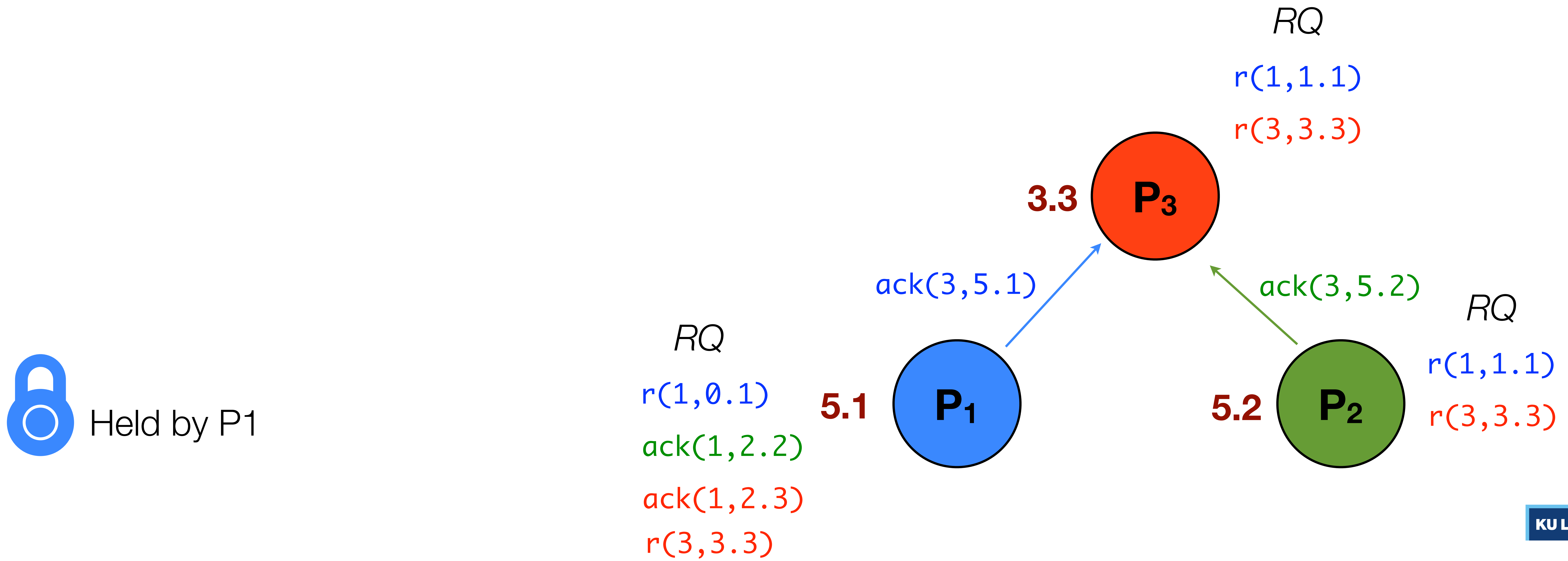
- P_1 and P_2 queue the request:



KU LEUVEN DistriNet

Lamport's mutual exclusion algorithm

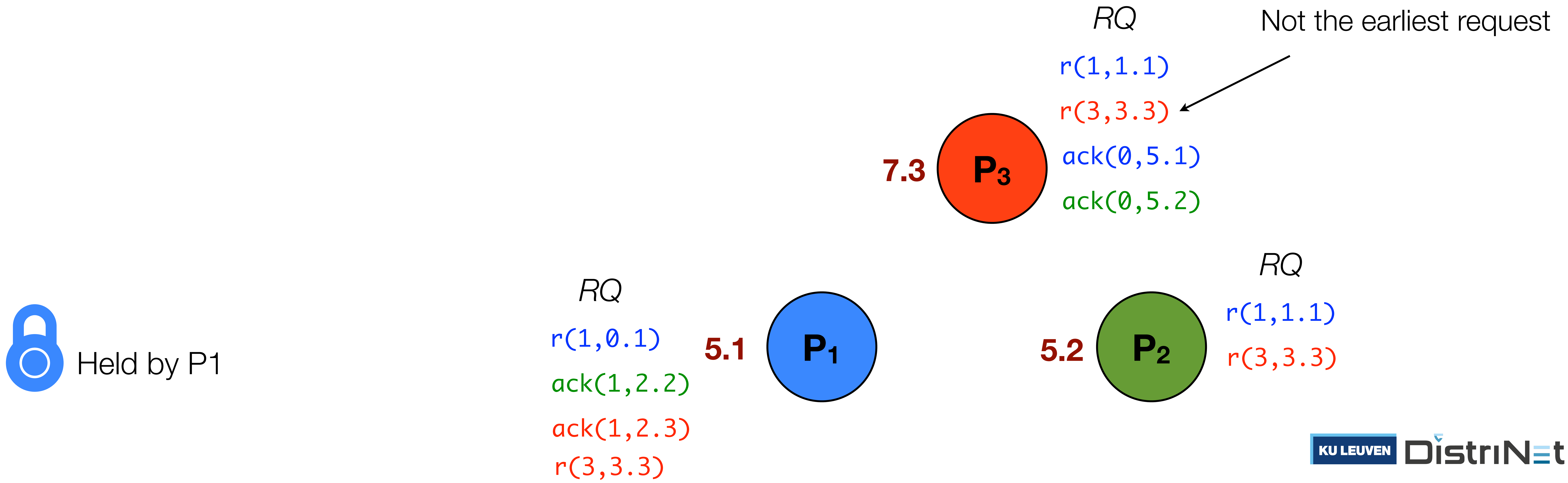
- P_1 and P_2 reply with acknowledgement:



Lamport's mutual exclusion algorithm

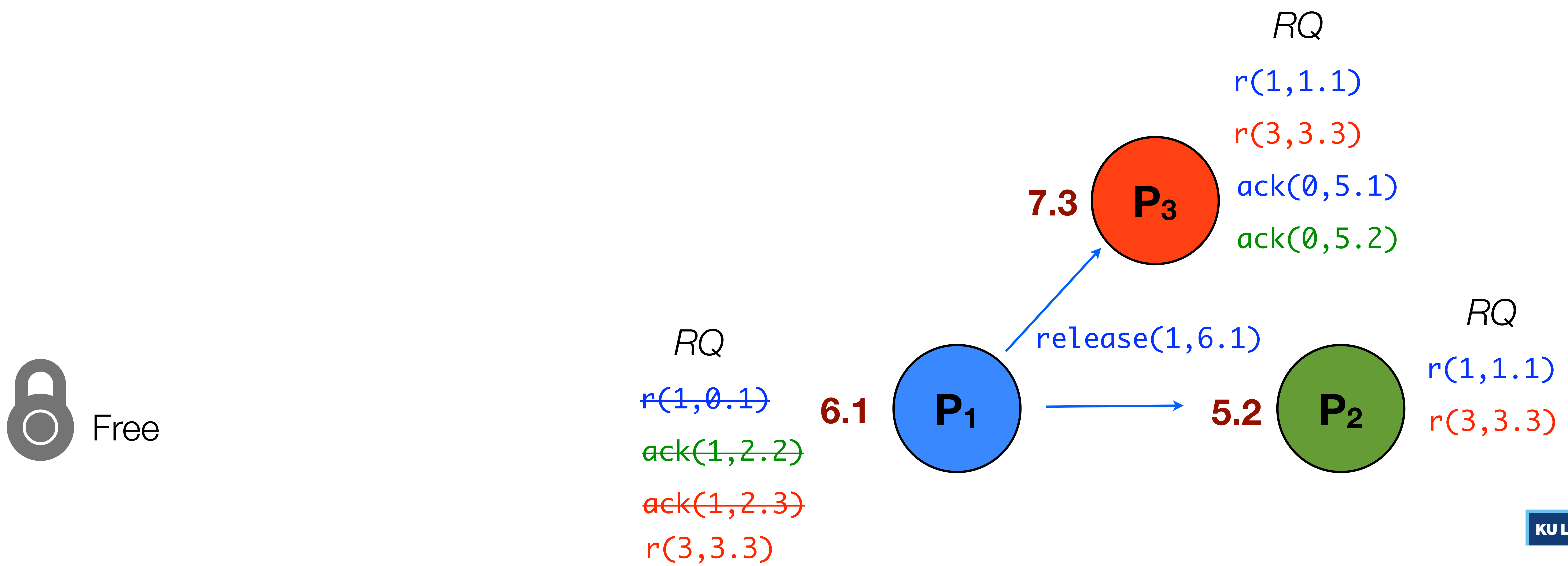
- P_3 received all replies, but must still wait for P_1 to release the resource (its request is *not* the earliest in the queue)

- P_1 retains exclusive access until it explicitly releases the resource:



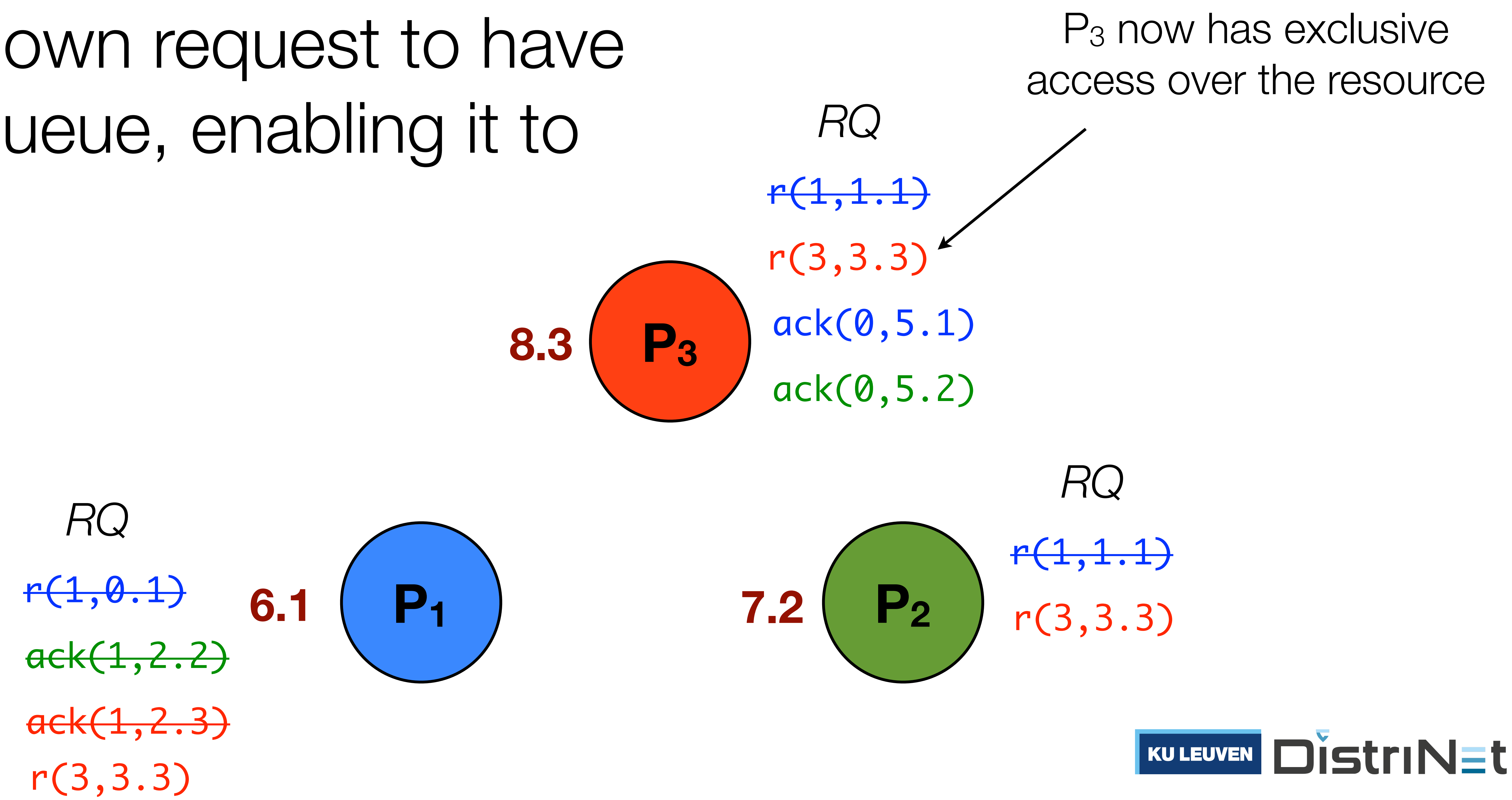
Lamport's mutual exclusion algorithm

- To release the resource (exit the critical section):
  - Remove own request from own request queue
  - Send a timestamped `release(i,Tᵢ)` message



Lamport’s mutual exclusion algorithm

- When a process receives a `release(i,Ti)` message:
  - Remove the previous `request(i,_)` message for that process from local request queue
  - This may cause the process’s own request to have the earliest timestamp in the queue, enabling it to gain exclusive access



Lamport's mutual exclusion algorithm

- Illustrates that a fully distributed algorithm is possible. No central coordinator.
- Fair scheduling (requests are granted in-order)
- But... *N* points of failure (if one process crashes, no other process can acquire access to the resource anymore) (Why?)
- A lot of messaging traffic:
  - Requests: (N-1) requests + (N-1) ack replies = 2(N-1) messages
  - Releases: (N-1) release messages

Ricart and Agrawala's algorithm

- Another distributed algorithm for distributed mutual exclusion.
- See handbook chapter 15.2 for a full description of this algorithm.
- Very similar to Lamport's algorithm (also uses lamport timestamps), but:
  - **Lamport**: processes *always* reply to *all* requests. A process has the lock when its request is earliest in the queue.
    N-1 requests + *N-1* replies + *N-1* releases = 3(*N*-1) messages
  - **Ricart & Agrawala**: process that has the lock *does not reply* to a request until it releases the lock. A process has the lock when it received a reply from *all* other processes. No explicit release messages are sent.
    N-1 requests + *N-1* replies + *0* releases = 2(*N*-1) messages

Vector Clocks

Recall: causal relations cannot be determined from Lamport clocks

- Lamport clocks: if $e \rightarrow e'$ then $L(e) < L(e')$
- But, if $L(e) < L(e')$, we cannot conclude that $e \rightarrow e'$
  - E.g. $L(j) < L(b)$ but $j \nrightarrow b$
- By looking only at Lamport timestamps, we cannot conclude which events are causally related and which are not
- Solution: use a vector clock

Vector Clocks

- A vector clock for n processes is an array of n integers
- Each process P_i has its own local vector clock V_i
- For a vector clock V_i,
  - $V_i[i]$ is the number of events that process P_i has timestamped
  - $V_i[j]$ $(j \mathrel{!}= i)$ is the number of events that have occurred at P_j that have potentially affected P_i

Vector Clocks (in code)

on initialisation at process P_i **do**
  $V := \langle 0, 0, \ldots, 0 \rangle$ // *V is a local variable at process P_i*
end on

on any event occurring at process P_i **do**
  $V[i] := V[i] + 1$
end on

on request to send message *m* at process P_i **do**
  $V[i] := V[i] + 1$;
  send (V, m) via network
end on

on receiving (V', m) at process P_i via the network **do**
  $V[j] := \max(V[j], V'[j])$ for every $j \in \{1, \ldots, n\}$
  $V[i] := V[i] + 1$;
  deliver *m* to the application
end on

Vector Clocks (in words)

on initialisation at process P_i **do**
  $V := \langle 0, 0, \ldots, 0 \rangle$ // *V is a local variable at process P_i*
end on

on any event occurring at process P_i **do**
  $V[i] := V[i] + 1$
end on

on request to send message *m* at process P_i **do**
  $V[i] := V[i] + 1$;
  send (V, m) via network
end on

on receiving (V', m) at process P_i via the network **do**
  $V[j] := \max(V[j], V'[j])$ for every $j \in \{1, \ldots, n\}$
  $V[i] := V[i] + 1$;
  deliver *m* to the application
end on

Four update rules:

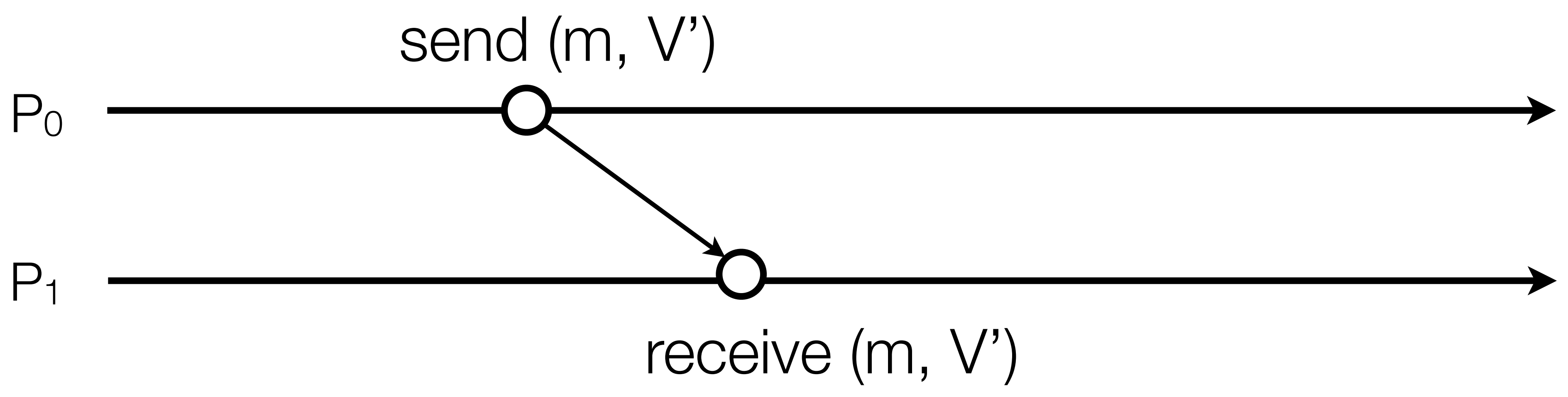
1. Each Process `Pi` initializes its clock `V` to `0` at each of `n` processes
2. Before timestamping an event, a process `Pi` increments element `i` of their local clock
3. When Process `Pi` sends a message to process `Pj`, it attaches a copy of its local clock `V`
4. When Process `Pi` receives a message from process `Pj` with attached clock `V'` it compares the elements of `V` and `V'` and sets its local clock to the highest of the two values

Vector Clocks (in words)

on receiving (V' , m) at process P_i via the network **do**
  $V[j] := \max(V[j], V'[j])$ for every $j \in \{1, \ldots, n\}$
  $V[i] := V[i] + 1;$
  deliver m to the application
end on

4. When Process `Pi` receives a message from process `Pj` with attached clock `V'` it compares the elements of `V` and `V'` and sets its local clock to the highest of the two values

Example
(assume n = 4):



P_0's clock when it sends m:	V' = [0,5,12,1]
P_1's clock before it receives m:	V_{old} = [2,8,10,1]
P_1's new clock after line 2:	V_{new} = [2,8,12,1]
P_1's new clock after line 3:	V_{new} = [2,9,12,1]

Comparing vector timestamps

- Define:

  $V = V' \Leftrightarrow V[i] = V'[i]$ for i = 1 ... N

  $V \leq V' \Leftrightarrow V[i] \leq V'[i]$ for i = 1 ... N

  $V < V' \Leftrightarrow V \leq V'$ and $V \neq V'$

- For any two events e, e':

  if $e \rightarrow e'$ then $V(e) < V(e')$

  ... just like Lamport's algorithm, but also:

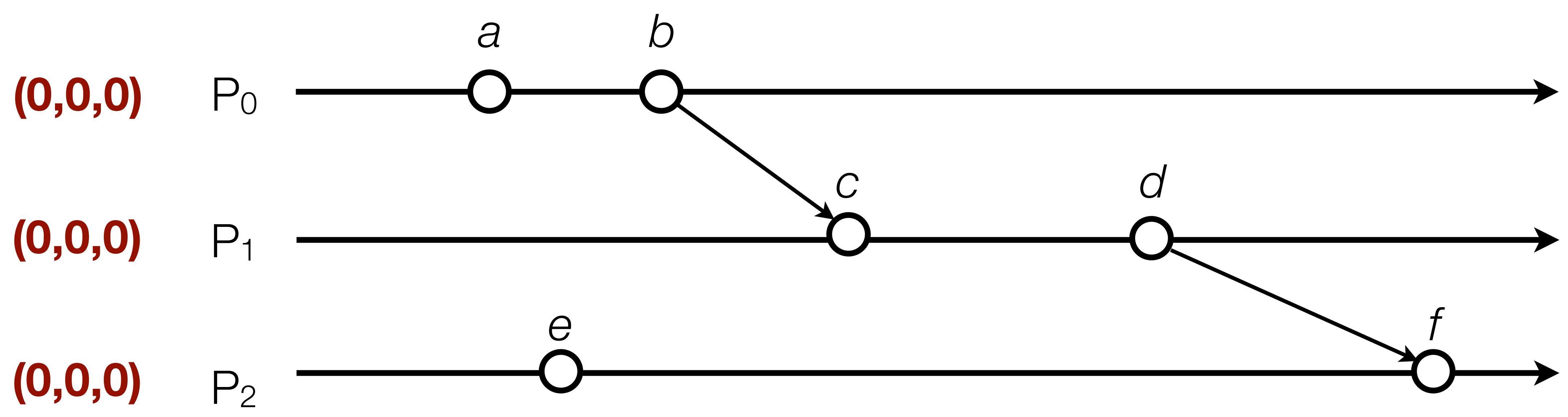
  if $V(e) < V(e')$ then $e \rightarrow e'$

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

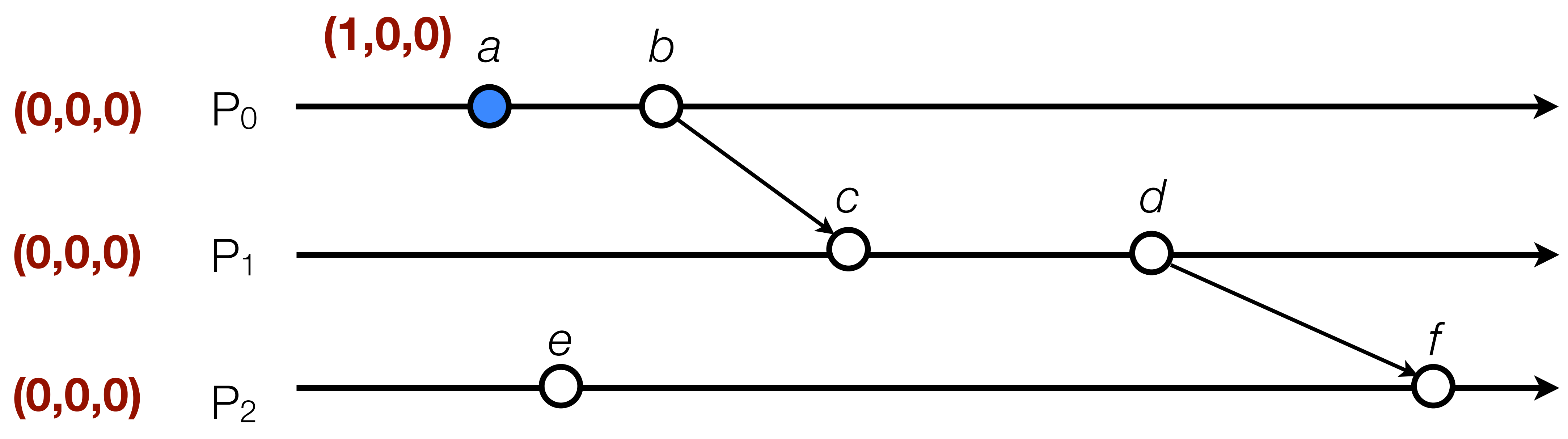
  $e \,||\, e'$

Vector timestamps

Example (assume n = 3):

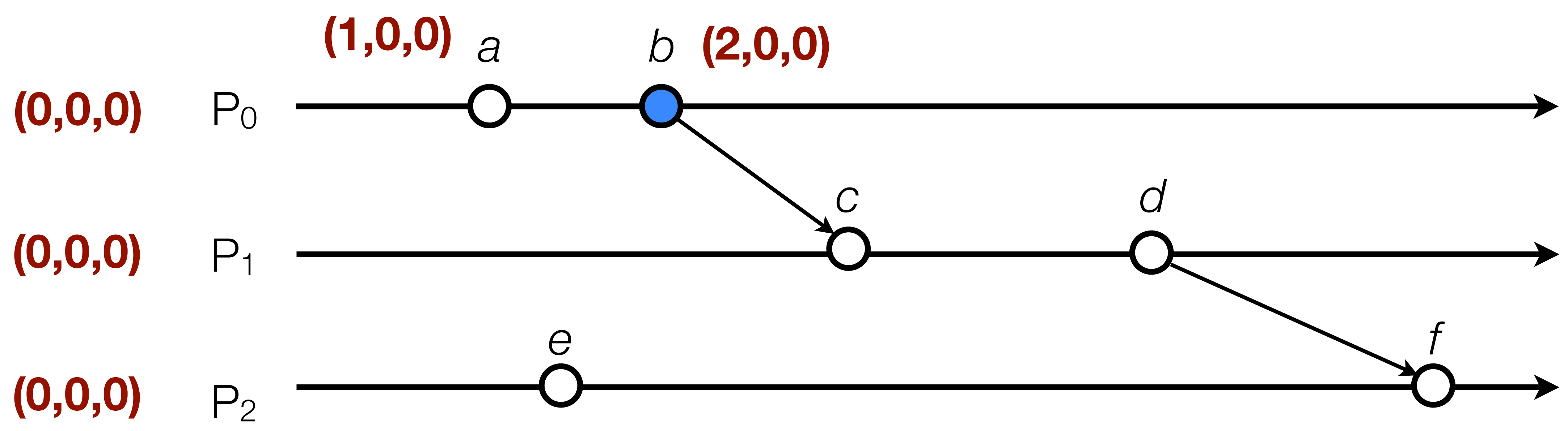


Vector timestamps



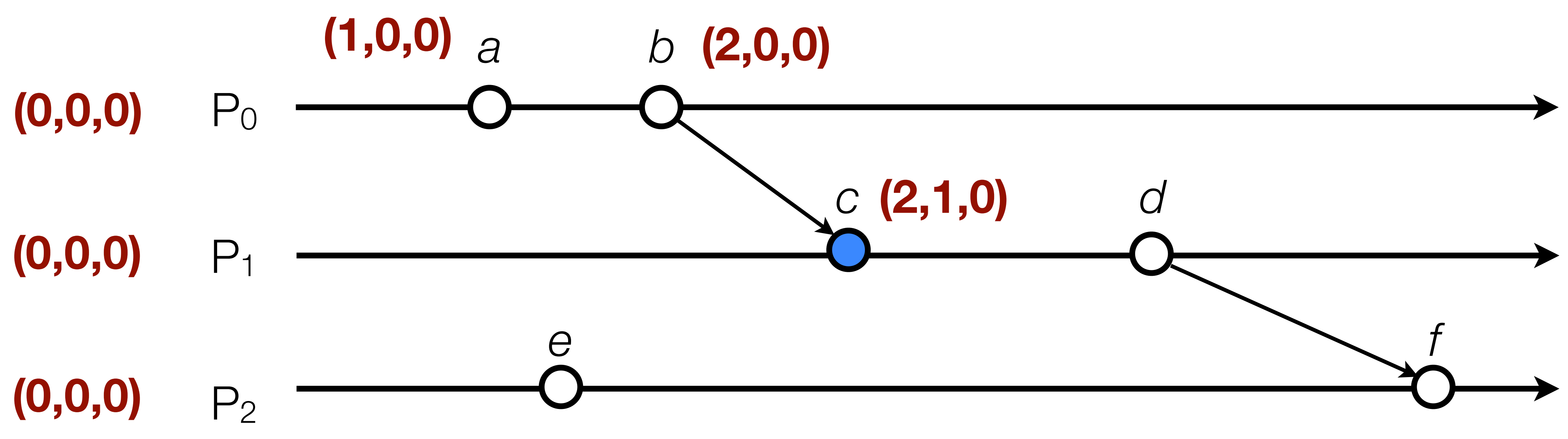
Event	Timestamp
a	**(1,0,0)**

Vector timestamps



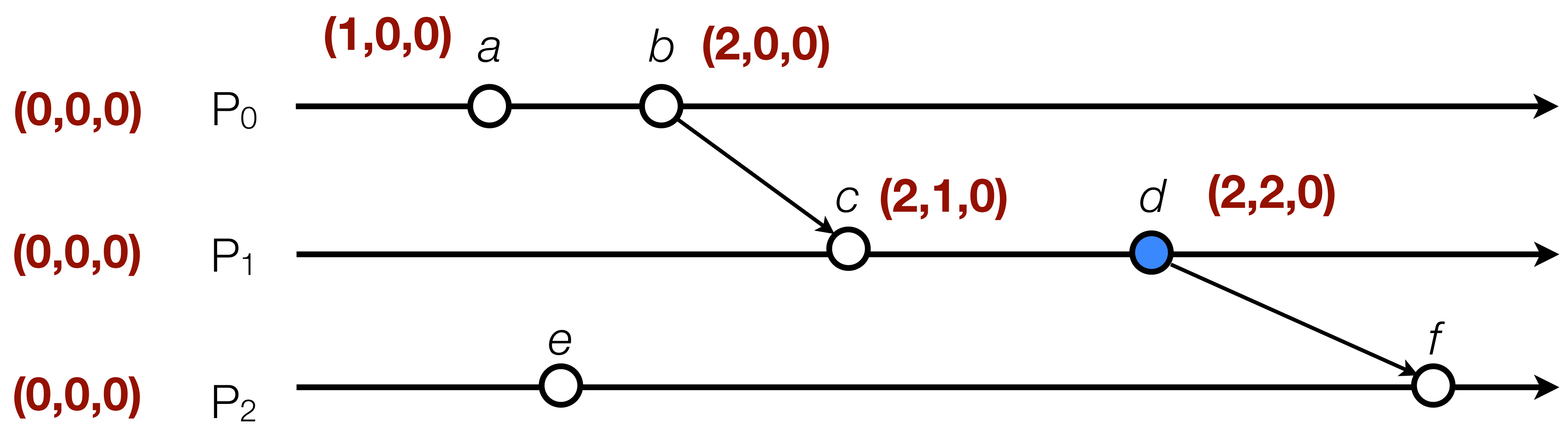
Event	Timestamp
a	(1,0,0)
b	(2,0,0)

Vector timestamps



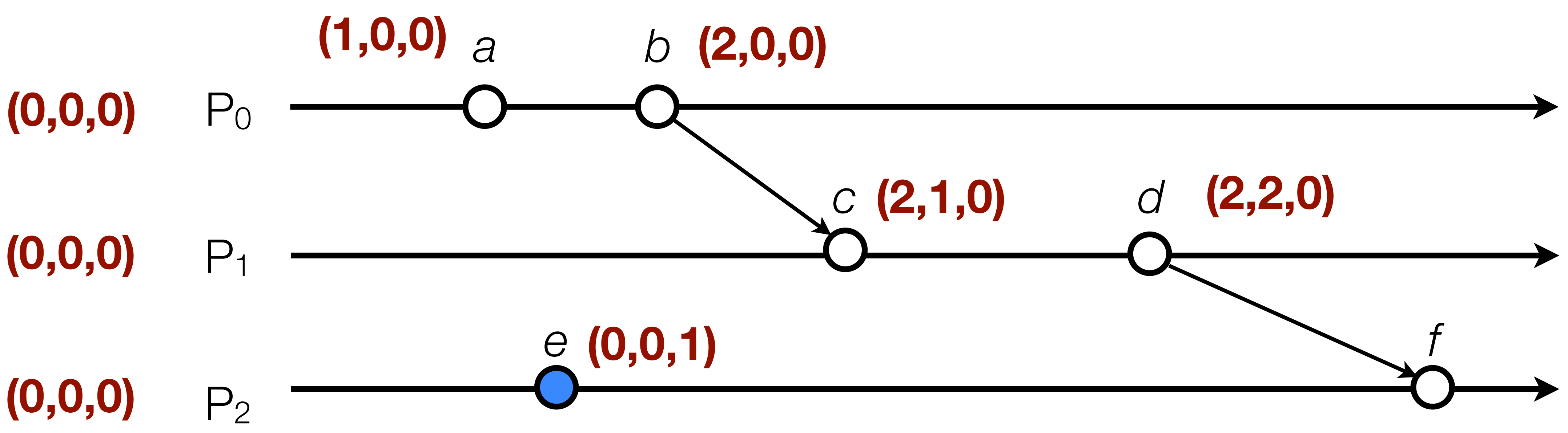
Event	Timestamp
a	**(1,0,0)**
b	**(2,0,0)**
c	**(2,1,0)**

Vector timestamps



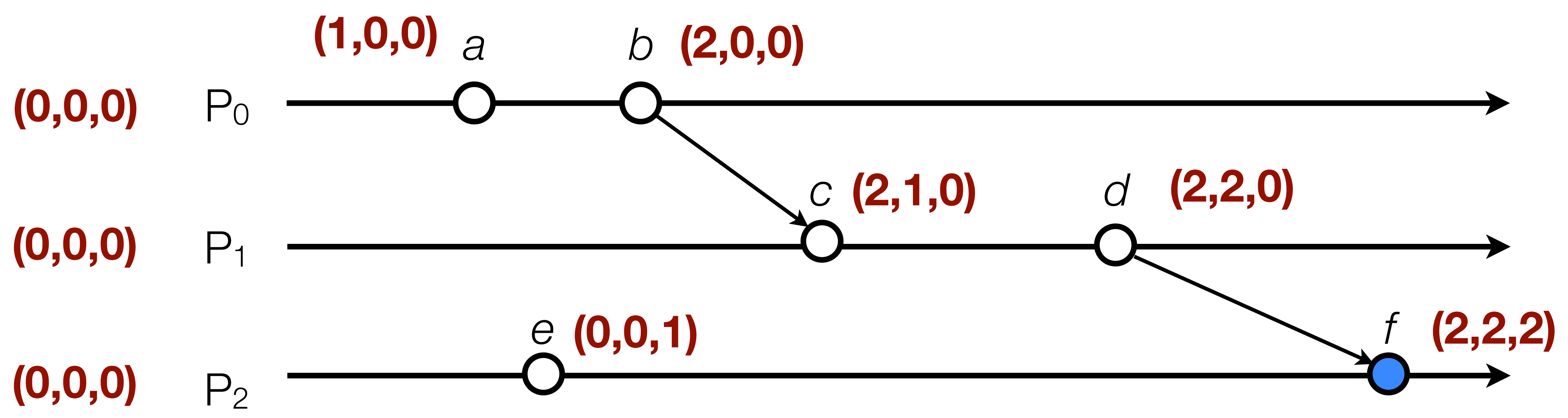
Event	Timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)

Vector timestamps



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

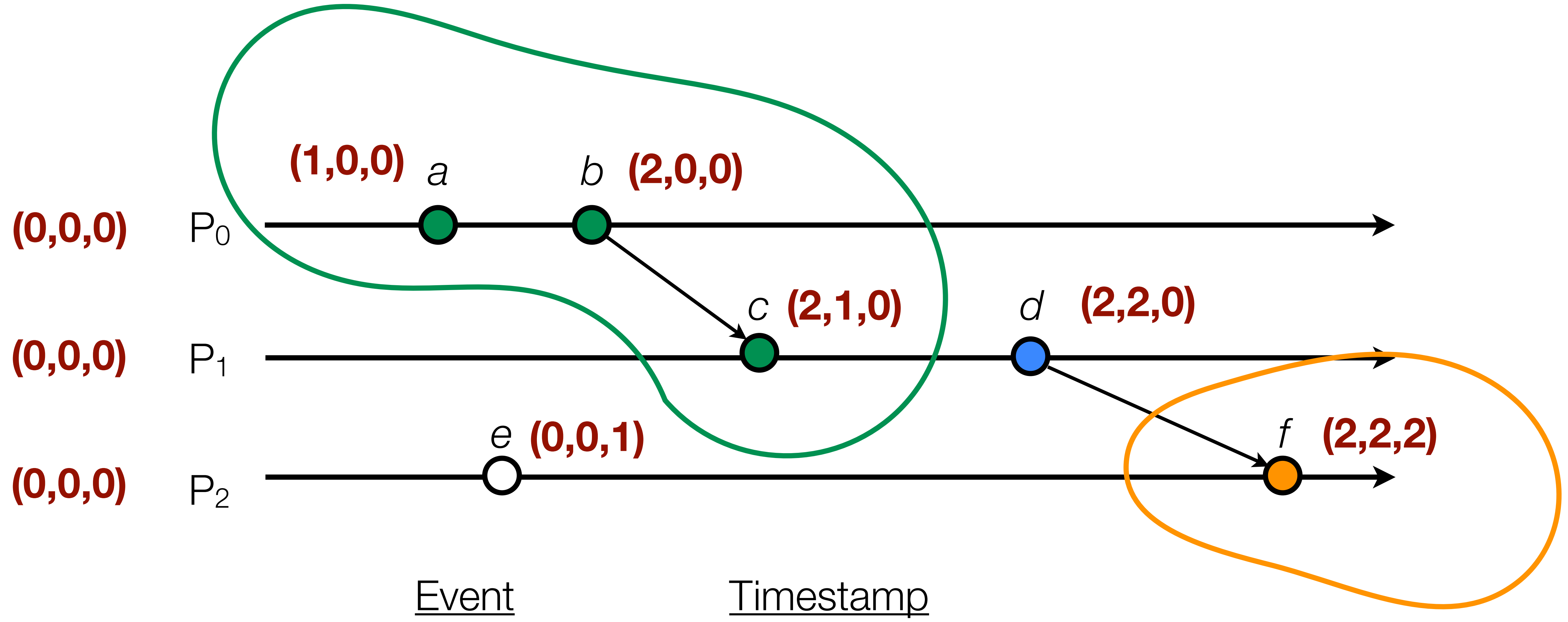
Vector timestamps



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

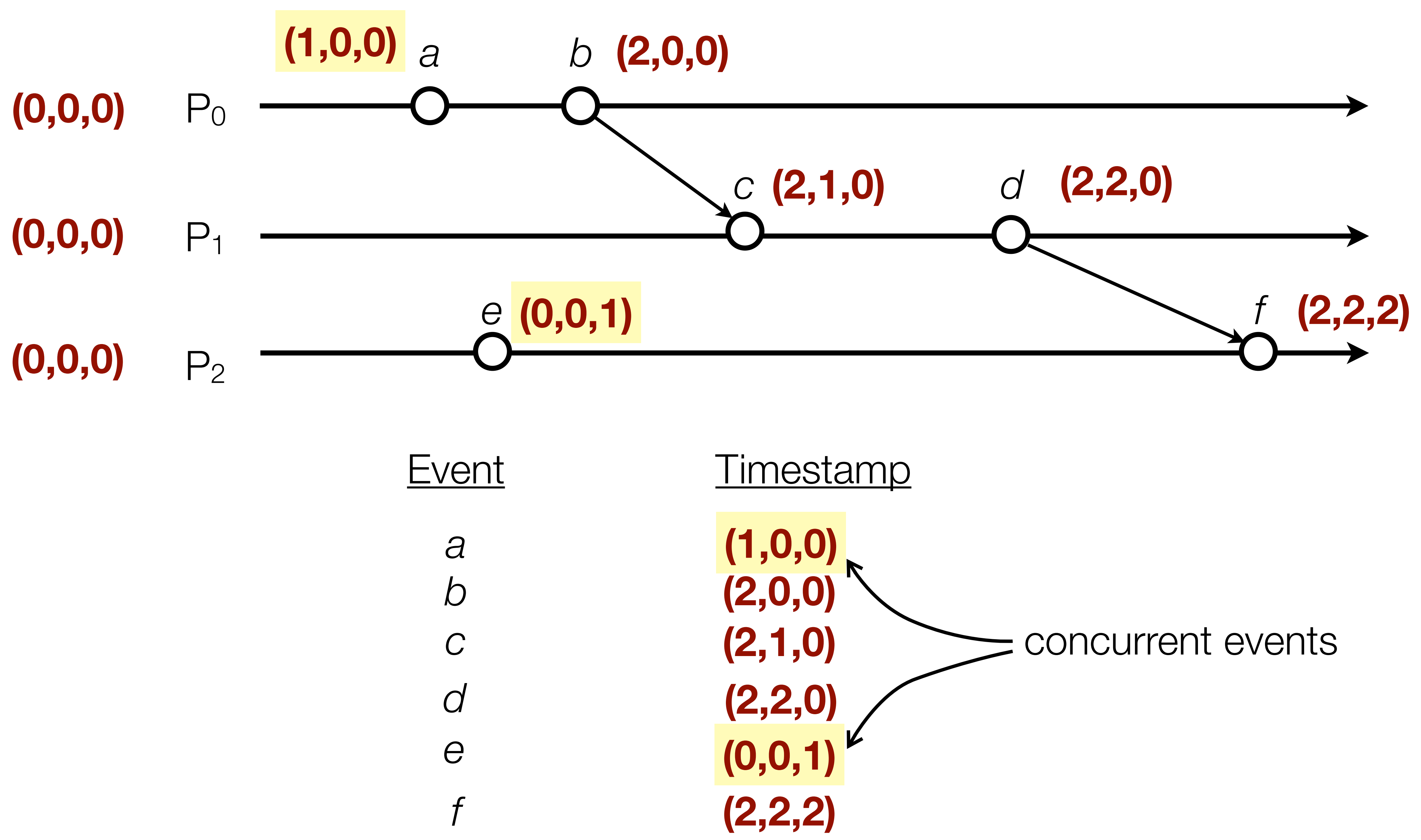
events e_{cause} for which $V(e_{cause}) < V(d)$



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)

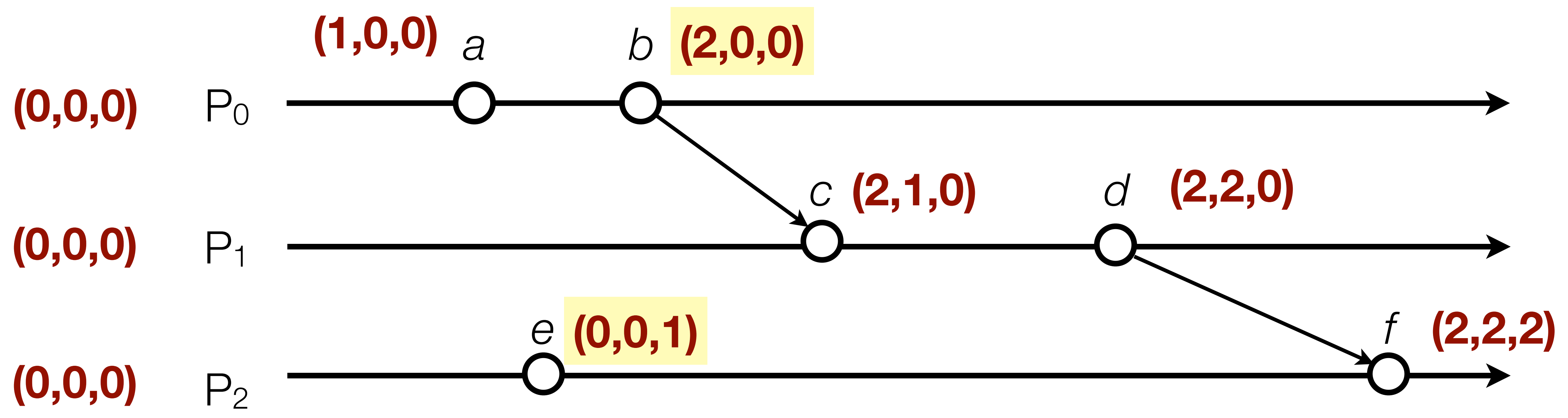
events e_{effect} for which $V(d) < V(e_{effect})$

Vector timestamps



Two events e and e' are **concurrent** if neither V(e) ≤ V(e') nor V(e') ≤ V(e)

Vector timestamps

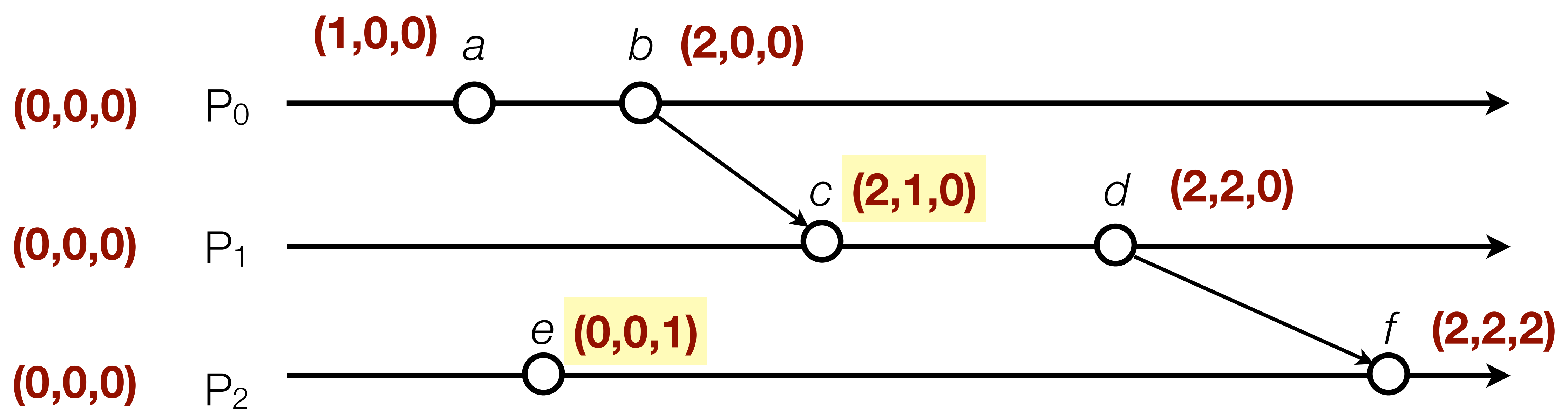


Event	Timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

Two events e and e’ are **concurrent** if neither V(e) ≤ V(e’) nor V(e’) ≤ V(e)

Vector timestamps

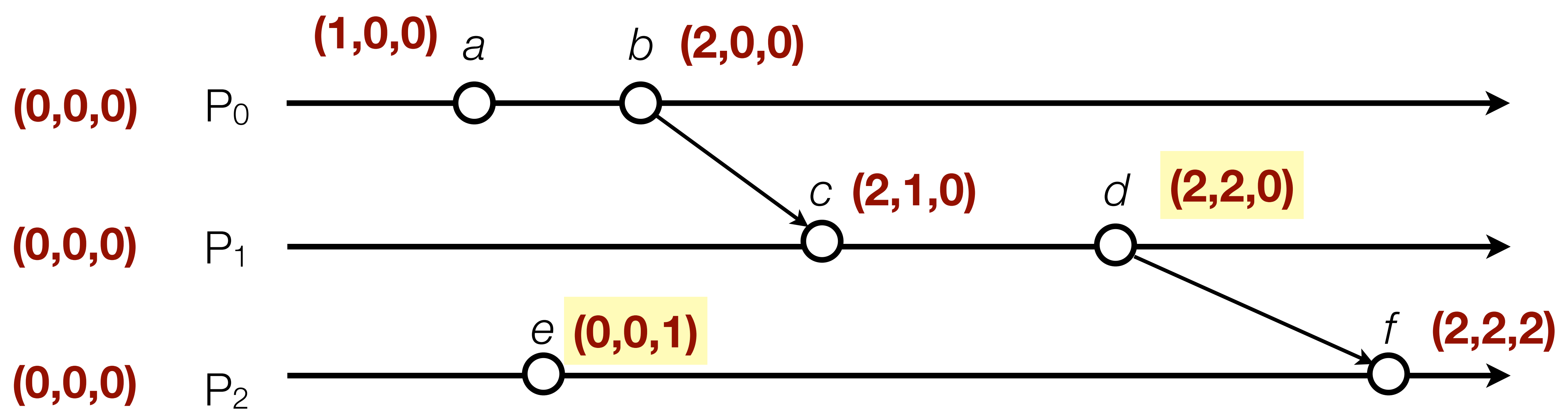


Event	Timestamp
a	**(1,0,0)**
b	**(2,0,0)**
c	**(2,1,0)**
d	**(2,2,0)**
e	**(0,0,1)**
f	**(2,2,2)**

concurrent events

Two events e and e' are **concurrent** if neither V(e) ≤ V(e') nor V(e') ≤ V(e)

Vector timestamps



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

Two events e and e’ are **concurrent** if neither V(e) ≤ V(e’) nor V(e’) ≤ V(e)

Vector Clocks: applications

- Used in replicated databases
- Key-value store API:
  - `get(key)`: returns a tuple `(value, clock)` or *multiple* tuples in case of write conflicts
  - `put(key, value, clock)`: client should pass the last-read vector clock for `key`
- The vector clock timestamps allow the database to causally relate the `get` and `put` events:
  - If two or more `put` operations have concurrent vector timestamps, then the database can detect that the updates are concurrent and cause a **write conflict.** The database then stores *all* values and their clocks, so the client can resolve the conflict on the next `get`.
  - A client that `get`s multiple values can choose a reconciled value and store it using a new `put` operation using a vector timestamp that is more recent than all the previously recorded vector timestamps. The database uses the timestamp to detect that this new `put` operation can safely overwrite the old values and resolve the conflict.
- Real-world systems that use this mechanism: Riak, Amazon DynamoDB

Vector Clocks: applications (example)

- Example adapted from Riak (original at: http://basho.com/why-vector-clocks-are-easy/)

__Alice__, __Ben__, __Cathy__, and __Dave__ are planning to meet next week for dinner.

The planning starts with __Alice__ suggesting they meet on Wednesday.

__Dave__ exchanges email with __Ben__, and they decide on Tuesday: __Ben__ proposes the new value, and then __Dave__ confirms __Ben__'s proposal.

At the same time, __Cathy,__ unaware of __Dave__ and __Ben__'s suggested alternative meeting time, decides on Thursday instead.

When __Dave__ later checks the current proposal, he discovers that his and __Ben__'s suggestion for Tuesday now conflicts with __Cathy__'s suggestion for Thursday.

__Dave__ picks Thursday and updates the proposal, thus resolving the conflict.

When __Alice__ next checks the proposed date, she learns that the group has settled on Thursday.

Vector Clocks: applications (example)

- Example adapted from Riak (original at: http://basho.com/why-vector-clocks-are-easy/)

***Alice**, **Ben**, **Cathy**, and **Dave** are planning to meet next week for dinner.*

*The planning starts with **Alice** suggesting they meet on Wednesday.*

***Dave** exchanges email with **Ben**, and they decide on Tuesday: **Ben** proposes the new value, and then **Dave** confirms **Ben**'s proposal.*

*At the same time, **Cathy,** unaware of **Dave** and **Ben**'s suggested alternative meeting time, decides on Thursday instead.*

*When **Dave** later checks the current proposal, he discovers that his and **Ben**'s suggestion for Tuesday now conflicts with **Cathy**'s suggestion for Thursday.*

***Dave** picks Thursday and updates the proposal, thus resolving the conflict.*

*When **Alice** next checks the proposed date, she learns that the group has settled on Thursday.*

4 processes: {A, B, C, D}

1 key-value pair: (dinnerDay, string)

Vector Clocks: applications (example)

- Example adapted from Riak (original at: http://basho.com/why-vector-clocks-are-easy/)

__Alice__, __Ben__, __Cathy__, and __Dave__ are planning to meet next week for dinner.

The planning starts with __Alice__ suggesting they meet on Wednesday.

__Dave__ exchanges email with __Ben__, and they decide on Tuesday: __Ben__ proposes the new value, and then __Dave__ confirms __Ben__'s proposal.

At the same time, __Cathy,__ unaware of __Dave__ and __Ben__'s suggested alternative meeting time, decides on Thursday instead.

When __Dave__ later checks the current proposal, he discovers that his and __Ben__'s suggestion for Tuesday now conflicts with __Cathy__'s suggestion for Thursday.

__Dave__ picks Thursday and updates the proposal, thus resolving the conflict.

When __Alice__ next checks the proposed date, she learns that the group has settled on Thursday.

A: put(dinnerDay, "wed", (1,0,0,0))

B: v, c = get(dinnerDay) // "wed", (1,0,0,0)
C: v, c = get(dinnerDay) // "wed", (1,0,0,0)
D: v, c = get(dinnerDay) // "wed", (1,0,0,0)

Vector Clocks: applications (example)

- Example adapted from Riak (original at: http://basho.com/why-vector-clocks-are-easy/)

***Alice**, **Ben**, **Cathy**, and **Dave** are planning to meet next week for dinner.*

*The planning starts with **Alice** suggesting they meet on Wednesday.*

***Dave** exchanges email with **Ben**, and they decide on Tuesday: **Ben** proposes the new value, and then **Dave** confirms **Ben**'s proposal.*

*At the same time, **Cathy,** unaware of **Dave** and **Ben**'s suggested alternative meeting time, decides on Thursday instead.*

*When **Dave** later checks the current proposal, he discovers that his and **Ben**'s suggestion for Tuesday now conflicts with **Cathy**'s suggestion for Thursday.*

***Dave** picks Thursday and updates the proposal, thus resolving the conflict.*

*When **Alice** next checks the proposed date, she learns that the group has settled on Thursday.*

B: v, c = get(dinnerDay) // "wed", (1,0,0,0)
B: put(dinnerDay, "tue", c)

D: v, c = get(dinnerDay) // "tue", (1,1,0,0)
D: put(dinnerDay, "tue", c)

Vector Clocks: applications (example)

- Example adapted from Riak (original at: http://basho.com/why-vector-clocks-are-easy/)

Alice, ***Ben***, ***Cathy***, and ***Dave*** *are planning to meet next week for dinner.*

The planning starts with ***Alice*** *suggesting they meet on Wednesday.*

C: v, c = get(dinnerDay) // “wed”, (1,0,0,0)

Dave *exchanges email with* ***Ben***, *and they decide on Tuesday:* ***Ben*** *proposes the new value, and then* ***Dave*** *confirms* ***Ben***’*s proposal.*

At the same time, ***Cathy,*** *unaware of* ***Dave*** *and* ***Ben***’*s suggested alternative meeting time, decides on Thursday instead.*

C: put(dinnerDay, “thu”, c)

When ***Dave*** *later checks the current proposal, he discovers that his and* ***Ben***’*s suggestion for Tuesday now conflicts with* ***Cathy***’*s suggestion for Thursday.*

Dave *picks Thursday and updates the proposal, thus resolving the conflict.*

When ***Alice*** *next checks the proposed date, she learns that the group has settled on Thursday.*

Vector Clocks: applications (example)

- Example adapted from Riak (original at: http://basho.com/why-vector-clocks-are-easy/)

***Alice**, **Ben**, **Cathy**, and **Dave** are planning to meet next week for dinner.*

*The planning starts with **Alice** suggesting they meet on Wednesday.*

***Dave** exchanges email with **Ben**, and they decide on Tuesday: **Ben** proposes the new value, and then **Dave** confirms **Ben**'s proposal.*

*At the same time, **Cathy,** unaware of **Dave** and **Ben**'s suggested alternative meeting time, decides on Thursday instead.*

*When **Dave** later checks the current proposal, he discovers that his and **Ben**'s suggestion for Tuesday now conflicts with **Cathy**'s suggestion for Thursday.*

D: v, c = get(dinnerDay) // **conflict!**
// v = ["tue", "thu"]
// c = [(1,1,0,1), (1,0,1,0)]

***Dave** picks Thursday and updates the proposal, thus resolving the conflict.*

*When **Alice** next checks the proposed date, she learns that the group has settled on Thursday.*

Vector Clocks: applications (example)

- Example adapted from Riak (original at: http://basho.com/why-vector-clocks-are-easy/)

__Alice__, __Ben__, __Cathy__, and __Dave__ are planning to meet next week for dinner.

The planning starts with __Alice__ suggesting they meet on Wednesday.

__Dave__ exchanges email with __Ben__, and they decide on Tuesday: __Ben__ proposes the new value, and then __Dave__ confirms __Ben__'s proposal.

At the same time, __Cathy,__ unaware of __Dave__ and __Ben__'s suggested alternative meeting time, decides on Thursday instead.

When __Dave__ later checks the current proposal, he discovers that his and __Ben__'s suggestion for Tuesday now conflicts with __Cathy__'s suggestion for Thursday.

__Dave__ picks Thursday and updates the proposal, thus resolving the conflict.

When __Alice__ next checks the proposed date, she learns that the group has settled on Thursday.

D: v, c = get(dinnerDay) // **conflict!**
// v = ["tue", "thu"]
// c = [(1,1,0,1), (1,0,1,0)]

D: put(dinnerDay, "thu", merge(c[0], c[1]))

Vector Clocks: applications (example)

- Example adapted from Riak (original at: http://basho.com/why-vector-clocks-are-easy/)

__Alice__, __Ben__, __Cathy__, and __Dave__ are planning to meet next week for dinner.

The planning starts with __Alice__ suggesting they meet on Wednesday.

__Dave__ exchanges email with __Ben__, and they decide on Tuesday: __Ben__ proposes the new value, and then __Dave__ confirms __Ben__'s proposal.

At the same time, __Cathy,__ unaware of __Dave__ and __Ben__'s suggested alternative meeting time, decides on Thursday instead.

When __Dave__ later checks the current proposal, he discovers that his and __Ben__'s suggestion for Tuesday now conflicts with __Cathy__'s suggestion for Thursday.

__Dave__ picks Thursday and updates the proposal, thus resolving the conflict.

When __Alice__ next checks the proposed date, she learns that the group has settled on Thursday.

A: v, c = get(dinnerDay) // "thu", (1,1,1,2)

Logical Clocks: summary

- **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
- **Lamport clocks** $L(e)$: if $e \rightarrow e'$ then $L(e) < L(e')$
  - But, if $L(e) < L(e')$, we cannot conclude that $e \rightarrow e'$
- **Vector clocks** $V(e)$: if $e \rightarrow e'$ then $V(e) < V(e')$
  - And also, if $V(e) < V(e')$ then $e \rightarrow e'$
  - Two events are **concurrent** if neither $V(e) \leq V(e')$ nor $V(e') \leq V(e)$

Summary: time, coordination and agreement

- **Time, clocks, event ordering**: processes must agree on the order of events, even if there is no shared global clock.

- **Distributed mutual exclusion**: processes must agree on who has exclusive access to a resource, even in the absence of a central coordinator.

- Next: coordination and agreement in **Group communication**: processes must agree on the set of messages to be delivered, even in the face of unreliable or slow network links.