SYNCHRONIZATION

Distributed Systems

4. Sem BSc Informatics

IMC FH Krems

LECTURE OUTLINE

- Clock synchronization
- Logical Clocks
- Mutual exclusion
- Election algorithms
- Distributed event matching

SYNCHRONIZATION

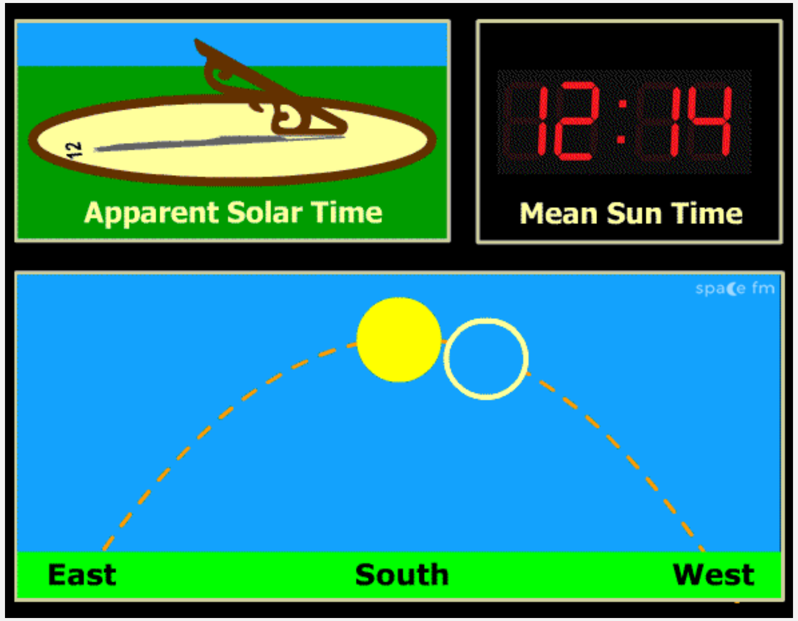
- Remember definition of distributed systems from Lecture 1: *"A distributed system is a collection of independent computers that appears to its users as a single coherent system"*
- Question: **How** do distributed systems manage to appear to its users as a single coherent system?
- Processes in a distributed system **collaborate** to achieve this goal.
- **Communication** is a part answer to this collaboration problem.
- **Coordination** is the next answer to the question.
  - Process coordinate by means of synchronization mechanisms.

CLOCKS

- In a centralized system, time is not ambiguous.
- In a distributed system, there might be small time differences between distributed processes.
- We need agreement on *what happened before/after what.*
- Possible approaches:
    1. Synchronize all clocks in a distributed system.
    2. Use *logical clocks*.

CLOCK SYNCHONIZATION

- Goal: synchronize **physical** clocks.
- Measurement base is **Universal Coordinated Time (UTC)**
  - Laboratories around the world measure time using Cs 133 clocks.
  - Bureau International de l'Heure (BIH - Paris) averages measurements to International Atomic Time (TAI).
  - From time to time, there is a clock skew between TAI and mean solar time.
  - When this happens, BIH announces a leap second.
  - UTC = TAI + leap seconds.
- UTC is broadcasted using shortwave radio stations.
- In combination with satellite services → accuracy ± 50 ns.



Source: space.fm

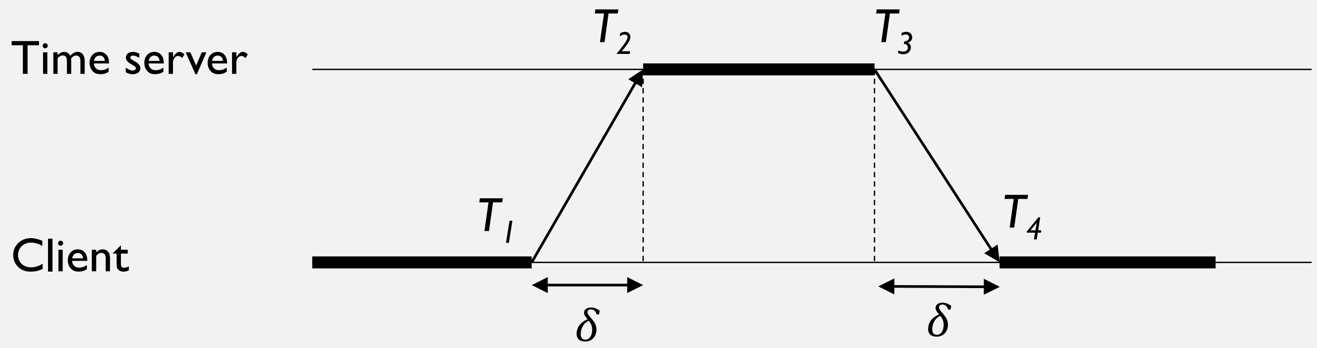
CLOCK SYNCHRONIZATION

Clock synchronization algorithms

- Goal: keep clocks in a distributed system synchronized within a given precision.
- Every hardware clock is subject to a **clock drift** by physical reasons.
  - Standard quartz clocks show drift rate of approx. 30 sec per year.
- **Network Time Protocol** (NTP).
  - Clients contact a time server with UTC receiver and accurate clock.
  - Reported time differs of server time by message delays.

CLOCK SYNCHRONIZATION

Network Time Protocol (NTP)



- Goal: estimate offset $\delta = T_2 - T_1$ (where we assume that $T_2 - T_1 \approx T_4 - T_3$)

$$\delta = \frac{(T_4 - T_1) - (T_3 - T_2)}{2}$$

- What happens if $\delta < 0$?

CLOCK SYNCHRONIZATION

Network Time Protocol (NTP)

- One of the oldest protocols in the Internet.
- UPD port 123.
- Can achieve accuracy in range 1-50 ms.
- Client (which can be another NTP server) synchronizes its clock.
  - Either $\delta > 0$: clock must be set forward.
  - Or $\delta < 0$: client's clock is set to a smaller frequence, until both clocks are in sync.
- Servers/clocks organized in *strata*.
  - Stratum 0: atomic clocks.
  - Stratum 1: those servers directly synchronized with an atomic lock (stratum 0).
  - In general, if server A synchronizes with a stratum k server, then A becomes a stratus $k+1$ server.

LOGICAL CLOCKS

- For most applications, we don't need all clocks to be sychronized with true UTC time.
- It is enough for all machines in a (closed) system to agree about what events preceeded other, and vice-versa.
- Logical clocks, introduced by Lamport (1978) achieve this.
- Let a, b be two events in a distributed system.
- We define a relation $\mathrm{Happens{-}before}(a, b)$ denoted by $a \rightarrow b$ such that:
  1. If a and b originated in the same process, and a happened before b, then $a \rightarrow b$
  2. If a is the event of sending a message and b the event of receiving that message, then $a \rightarrow b$
- Events that happened before another event (according to this relation) are said to be *causally* related.

LOGICAL CLOCKS

- What we look for is a logical clock C such that all process agree that

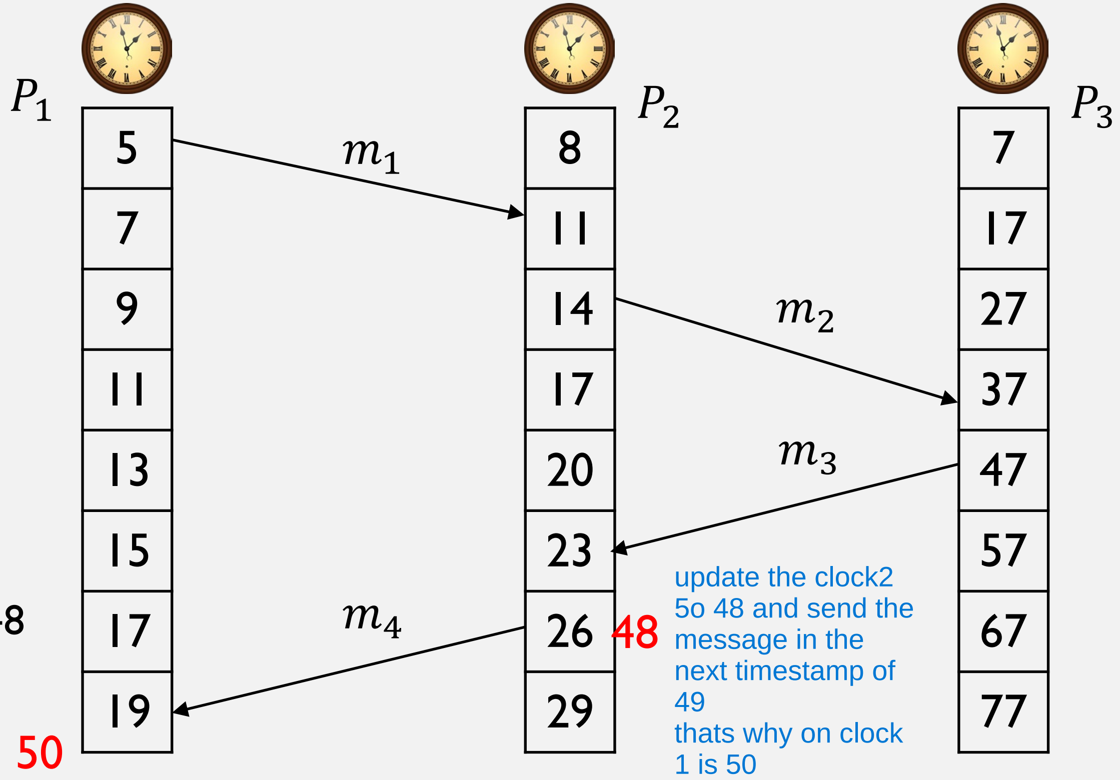
$$\text{If } a \to b \text{ then } C(a) < C(b)$$

- The relation is transitive: if $a \to b$ and $b \to c$ then $a \to c$
- If two events x and y happen in processes that do not exchange messages, then neither $x \to y$ nor $y \to x$ (x and y are concurrent).
  - C induces a partial order in the set of events.
- Corrections to a logical clock C can be made by adding values, never by substracting.

this clocks are more like counters rather than clock in classical way

LOGICAL CLOCKS

- Each process P_i has a clock C_i running at a given frequency



1. P_2 receives message m_1 from P_1 with timestamp $t_s(m_1) = 5$
2. P_3 receives message m_2 from P_2 with timestamp $t_s(m_2) = 14$
3. P_2 receives message m_3 from P_3 with timestamp $t_s(m_3) = 47 > C_2(t)$
4. P_2 updates its clock to 48
5. P_1 reacts analogously

LOGICAL CLOCKS

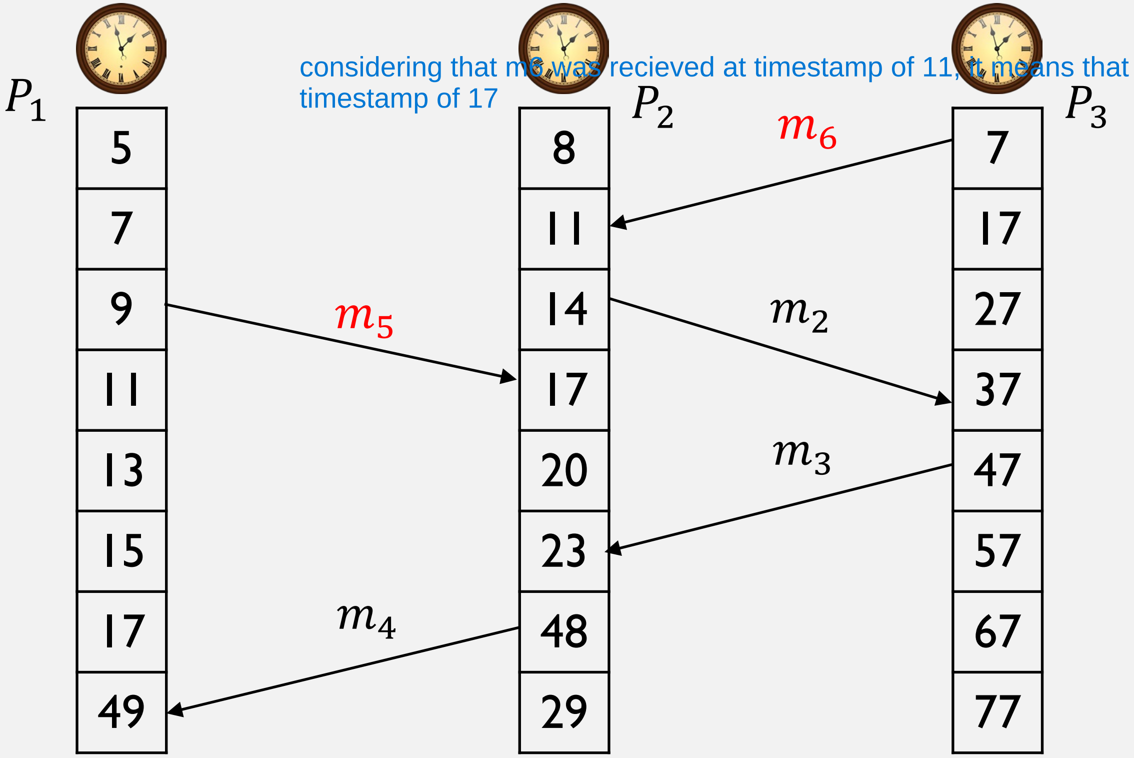
General algorithm for Lamport clocks:

- Process P_i has clock C_i which is incremented by 1 with a given frequency.
  - C_i can be seen as a general event counter.
- If process P_i sends a message m to process P_j, it sends as timestamp $t_s(m) = C_i$
- Upon receiving m, P_j updates its clock by $C_j \rightarrow \max(C_j, t_s(m))$.
- P_j increments its clock by 1 and forwards the message to the receiving application.
- Lamport clocks are implemented in the middleware layer.

LOGICAL CLOCKS

- Can we conclude from the counters that m_6 was **sent** before m_5?

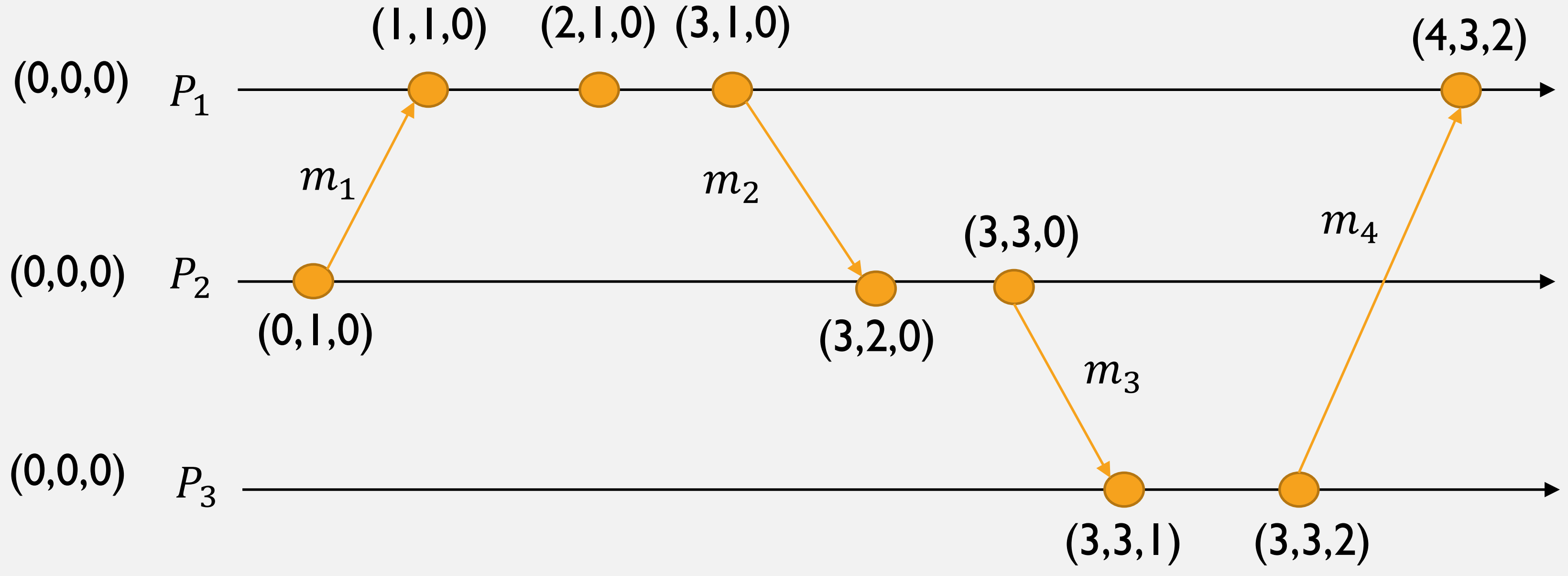
considering that m6 was recieved at timestamp of 11, it means that it was sent earlier thatn m5 whihc was recieved at timestamp of 17



VECTOR CLOCKS

- If $C(a) < C(b)$ we cannot always infer that a actually happened before b.
- For that, we need to account for the *causal histories* of the events that happened in the system.
- Main idea: extend Lamport clocks with the clocks of all the other processes in the system.
- Resulting construction: ***vector clock.***
- This vector clock is sent along with each message.

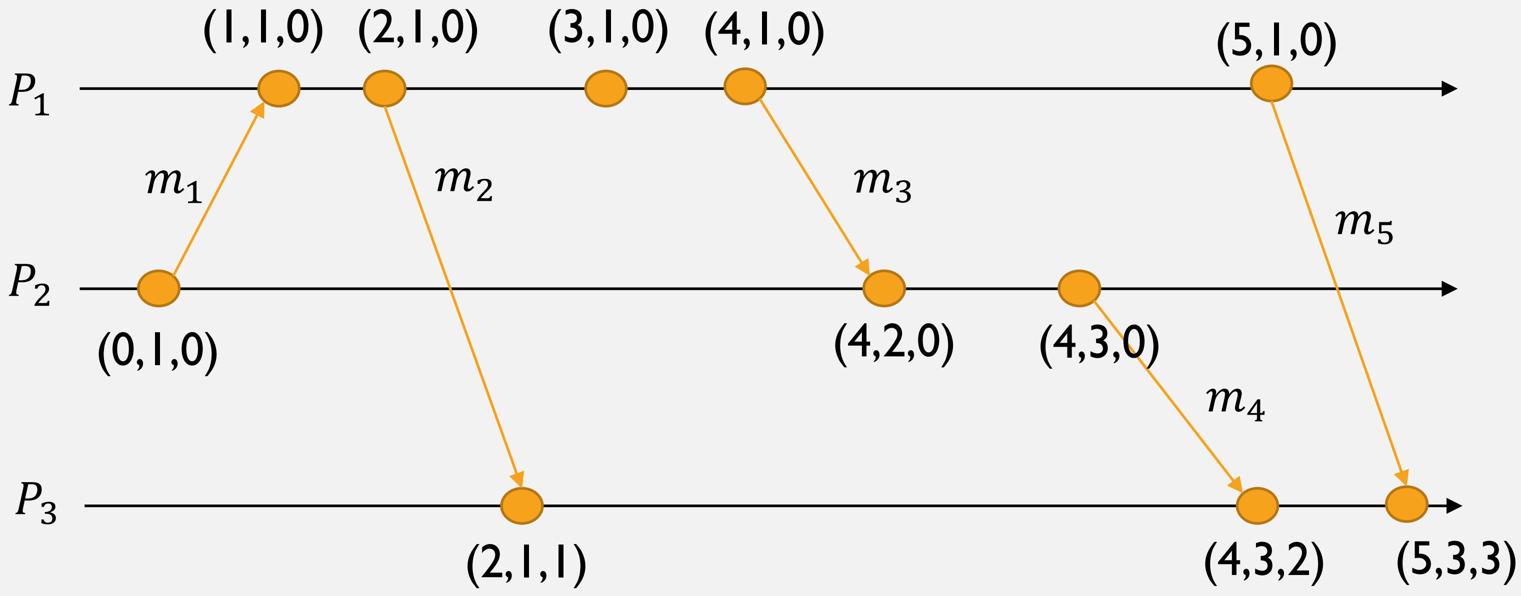
VECTOR CLOCKS



VECTOR CLOCKS

- $\mathrm{VC}_i = [C_1, C_2, \dots, C_i, \dots, C_n]$ is a vector of logical clocks, where $\mathrm{VC}_i[i]$ is the current logical clock (= event counter) of process P_i.
- Upon an event occurring (i.e. before sending a message) $\mathrm{VC}[i] \leftarrow \mathrm{VC}[i] + 1$
- If P_i sends a message to P_j then P_i sends its vector clock along with the message.
- Upon receiving the message, P_j adjusts its own clock by $\mathrm{VC}[k] \leftarrow \max(\mathrm{VC[k]}, t_s(m))$ for all processes k and then increments it .
- Comparing timestamps:
  - $a \rightarrow b$ if and only if $\mathrm{VC}_a[k] \leq \mathrm{VC}_b[k]$ for all k and there is at least one j such that $\mathrm{VC}_a[j] < \mathrm{VC}_b[j]$.

VECTOR CLOCKS



Are sending of m_4 and m_5 causally related?

LECTURE OUTLINE

- Clock synchronization
- Logical Clocks
- **Mutual exclusion**
- Election algorithms
- Distributed event matching

MUTUAL EXCLUSION

- How to ensure that processes that access the same resource simultaneously leave it in a consistent state?
- **Mutual exclusion:** ensure that only one process can access a given resource at a time.
- We need to prevent two common problems:
  1. **Starvation**: A process that needs to access a resource waits forever to get access granted.
  2. **Deadlock**: A situation where a number of processes wait for another process to finish using a resource, but that process is waiting as well.

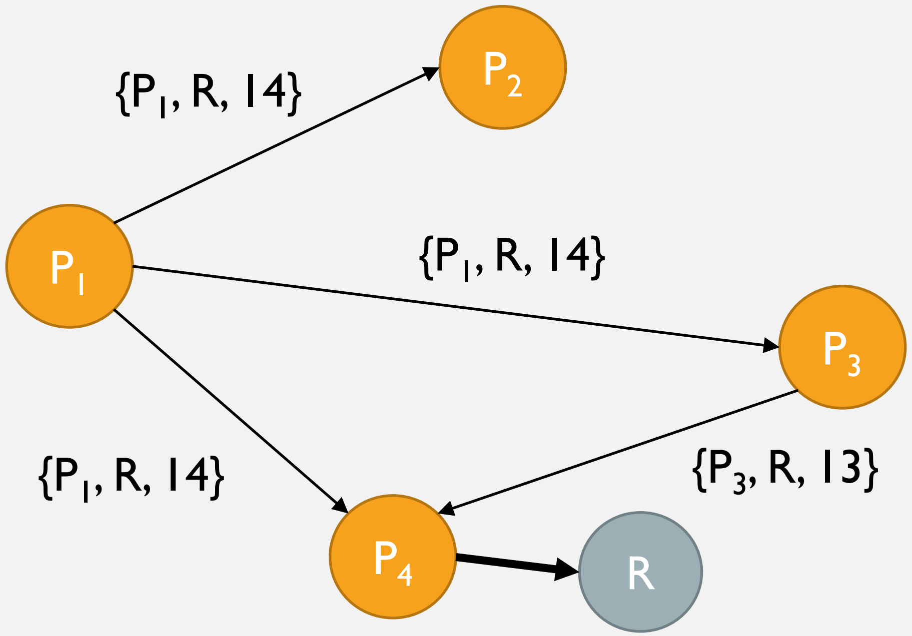
MUTUAL EXCLUSION

1. **Permission-based approach**

- Based on logical clocks.
- Each process sends a request for a given resource as (process name, resource name, logical clock) to *all* other processes.
- When a process receives the request:
  - If it is not interested in the resource → do nothing.
  - If it is currently accessing the resource → queue the request and send a response.
  - If it is currently **not** accesing the resource, but wants to → compare its clock with the timestamp in the requests.
    - The lowest value wins and the request who "lost" gets queued.

MUTUAL EXCLUSION

1. **Permission-based approach**

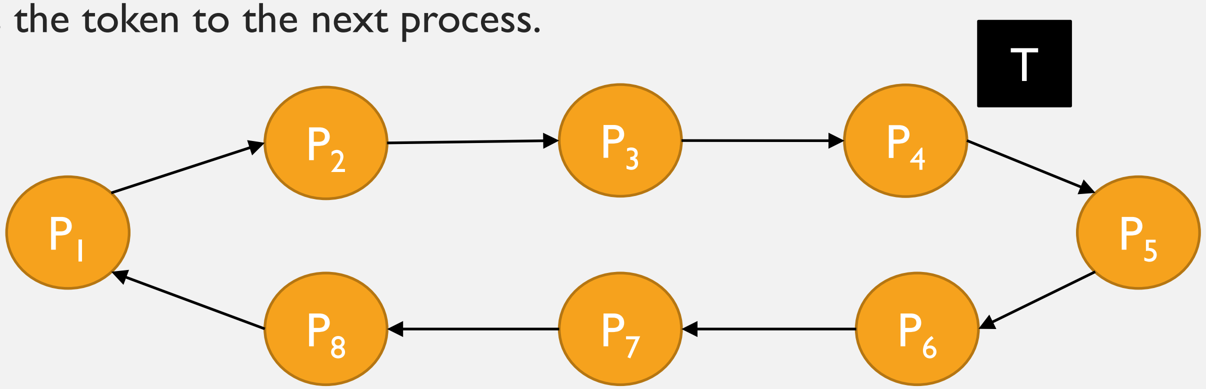


1. P_1 and P_3 are interested in accessing resource R
2. P_2 is not interested, so it ignores the request
3. P_4 is using the resource now
4. P_4 queues P_3 next
5. P_3 compares its clock to that of P_1 and queues it
6. When P_4 is done, it will handle the request to P_3
7. When P_3 is done, it will handle the request to P_1

MUTUAL EXCLUSION

2. **Token-based approach**

- Processes form an overlay network with the form of a ring.
- A token is generated and passed along the ring. The token is associated with a given resource.
- When a processes receives the token, it might
  - Use the associated resource.
  - Pass the token to the next process.



LECTURE OUTLINE

- Clock synchronization
- Logical Clocks
- Mutual exclusion
- **Election algorithms**
- Distributed event matching

ELECTION ALGORITHMS

- In some situations, one process has to act with a special role, for instance, as a coordinator
  - For instance, only one process has rights to change some data.
  - Coordinator role for replicating data.
  - If coordinator fails, a new coordinator must be "appointed".
- This process can be elected by an administrator, or automatically by means of an **election algorithm.**
- Main goal: after the algorithm, all processes should agree on which process has the coordinator role.

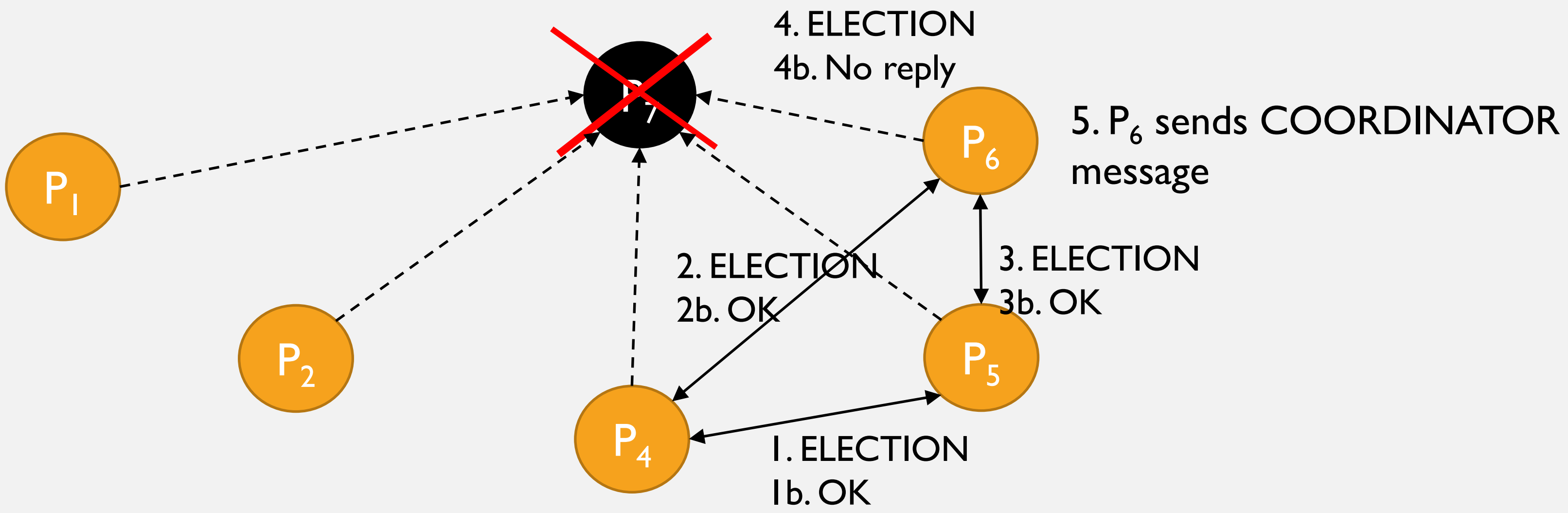
ELECTION ALGORITHMS

Bully algorithm

- A group of processes $\{P_1, P_2, \dots, P_N\}$ needs to elect a coordinator. We assume that processes are ordered in ascending order by their IDs $\mathrm{id}(P_k) = k$
- A process k announces that it will hold an election by means of a ELECTION message and sends it to all processes above it $P_{k+1}, \dots, P_N$
- If some process $j > k$ answers, then it takes over and P_k waits for the result.
- If no process replies, then P_k has won. It will be the new coordinator and announces it by means of a COORDINATOR message to all other processes.

ELECTION ALGORITHMS

Bully algorithm



What should happen if P_7 recovers?

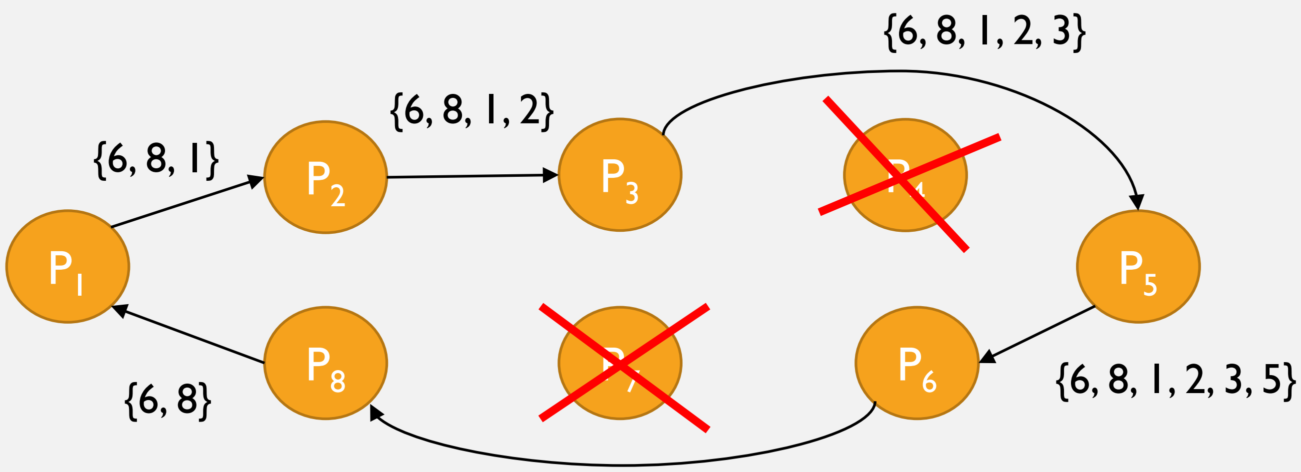
ELECTION ALGORITHMS

Ring algorithm

- Processes are arranged in a ring-like logical network.
- When coordinator fails, a process sends an ELECTION message, containing a list of process Ids.
  - The process includes itself in the list.
  - It sends the list to the next running process.
- Each process in the ring adds itself to the list.
- As soon as the message gets back, it changes into an COORDINATOR message to announce the new coordinator.
  - The new coordinator is the process with the highest ID in the list.

ELECTION ALGORITHMS

Ring algorithm



1. {6}
2. COORDINATOR{8}

LECTURE OUTLINE

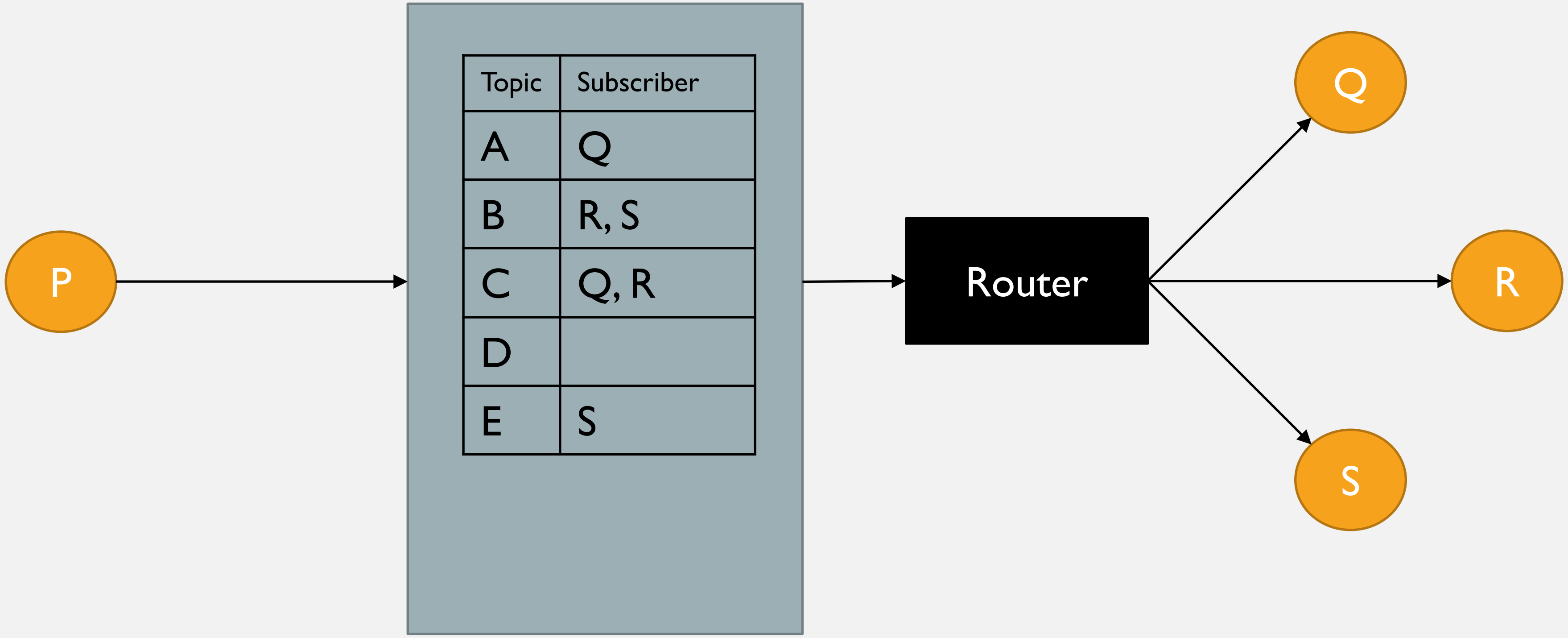
- Clock synchronization
- Logical Clocks
- Mutual exclusion
- Election algorithms
- **Distributed event matching**

EVENT MATCHING

- Remember **publish/subscribe** systems: a subscriber process P subscribes to a topic to get an event notification as soon as an event happens that is sent to that topic.
- Two main (coordination) issues:
  1. How to match subscriptions to events.
  2. How to notify subscribers in case of a match.
- Assume there is a function $\text{match}(S, N)$ that returns true if a subscription matches a notification, or false otherwise.

EVENT MATCHING

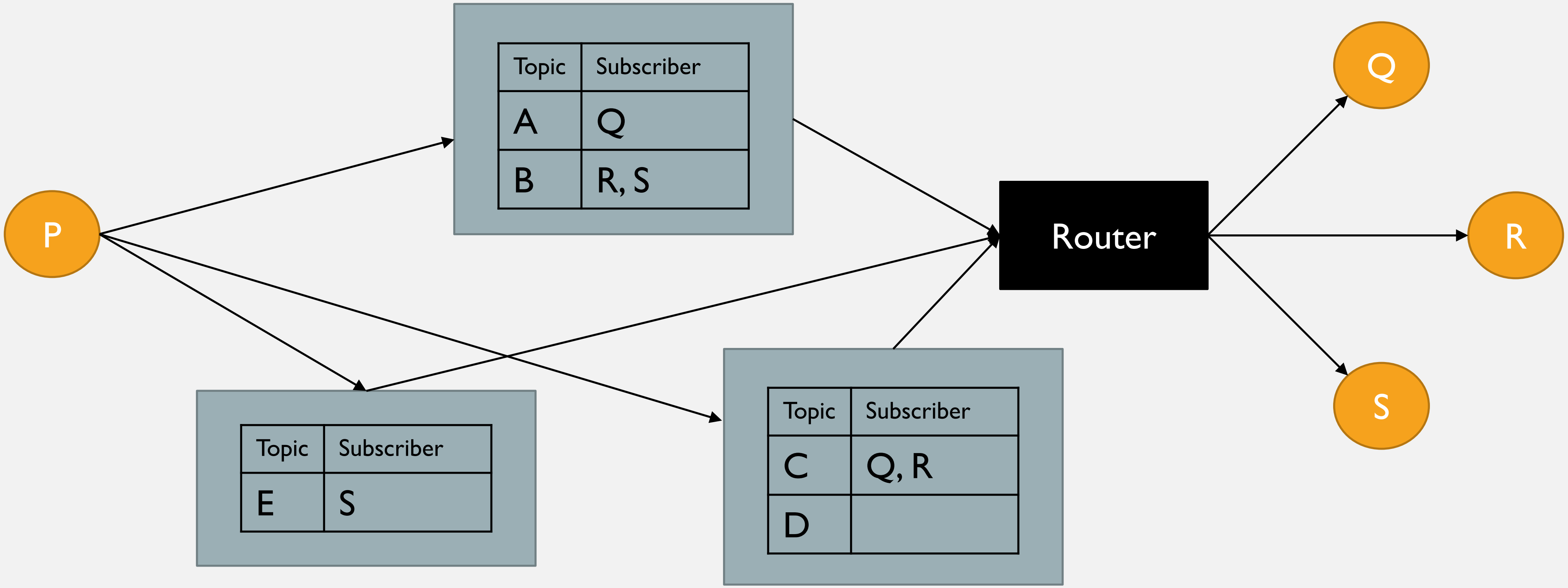
- Centralized implementation



Topic	Subscriber
A	Q
B	R, S
C	Q, R
D	
E	S

EVENT MATCHING

- Distributed implementation: we divide the work among several servers



EVENT MATCHING

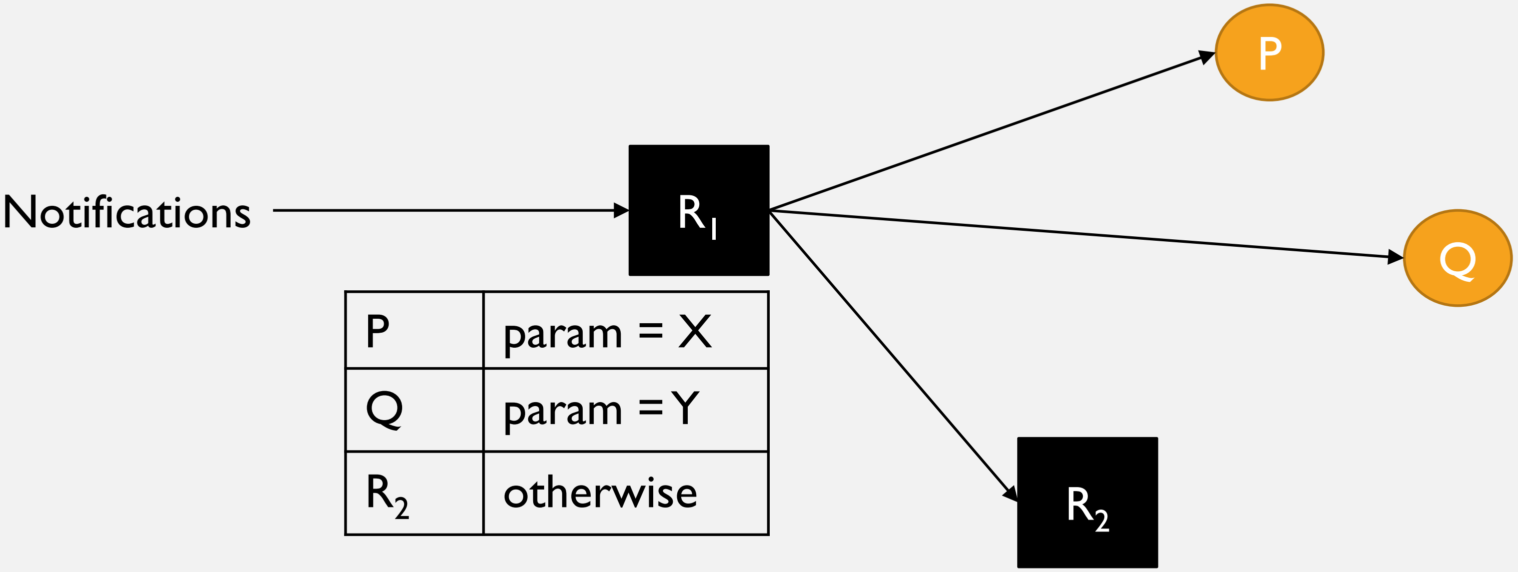
- Distributed implementation: we divide the work among several servers
- Subscriptions distributed among a set of servers (**brokers**)
  - Which server can handle which notification → can be solved by DHT.
  - Routing servers **(routers)** forward notification to subscribers.
- How to forward notifications to subscribers
  - Flooding (i.e. multicast).
  - Selective-notification routing.

EVENT MATCHING

- Distributed implementation: we divide the work among several servers
- Selective notification routing:
  - Server routes a notification based on (part of) its contents.
  - Depending on this information, some routes are discarded.
  - Each broker broadcasts its subscription over the (overlay) network.
  - Routers compose **routing filters.**

EVENT MATCHING

- Distributed implementation: selective notification routing



P	param = X
Q	param = Y
R_2	otherwise