Unit 9. Distributed algorithms

Concurrency and Distributed Systems

Teaching Unit Objectives

- Understand the **features** of distributed algorithms and relevant examples.
- Understand difficulties of **temporal synchronization**, in the absence of a unique clock that all processes could consult.
- Understand the **wide variety** of algorithmic problems that arise in distributed systems, identifying some of the most relevant: consensus, leader election, mutual exclusion.
- Knowing some **algorithms** at a basic level, studying their operation, their characteristics, their complexity and cost and the fact of whether they tolerate failures or not.
- Identify the failures in the nodes as the greatest source of **complexity** in the design of distributed algorithms.

Distributed Algorithms

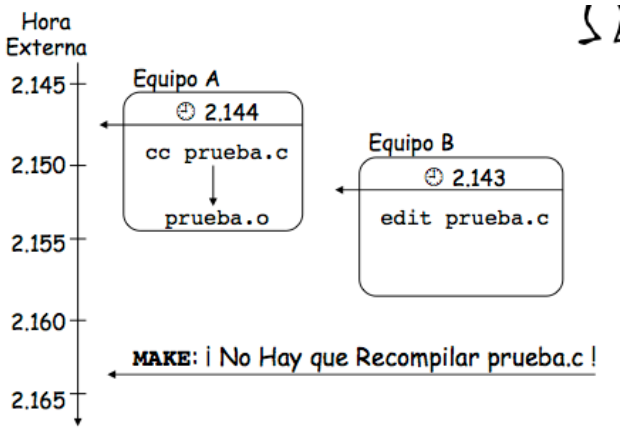
- Fundamental Algorithmic Concepts
  - Independent of particular technologies
  - Characteristics of distributed **decentralized** algorithms:
    1. No node has all the complete system information
    2. They run on independent processors (nodes)
       - The failure of a node does not prevent the algorithm from progressing
    3. They make decisions based on local information
    4. There is no precise source of global time

Content

- Global States and Time
  - Clocks, events and states
  - Clock synchronization algorithms: Cristian, Berkeley
  - Logical clocks and Vector clocks
  - Global states
- Examples of Algorithmic Problems in Distributed Systems
  - Mutual exclusion
  - Leader election
  - Consensus
- Algorithms in the presence of failures

Difficulties in temporal synchronization

- **Synchronization** provides mechanisms for coordinating activities.
- It is more complex in distributed systems than in centralized ones. Some examples:
  - Access to shared resources.
  - Event ordering.



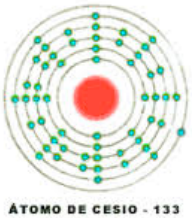
Characterization of the problem

Characterization of the problem → ***Synchronising physical clocks:***

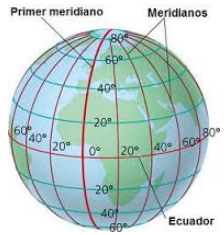
- Each node *i* has a local clock *Ci*
  - It represents a universal temporal coordinated (UTC) value
- Given any real instant t
  - The goal is that all nodes have Ci(t)=t
  - This means that all local clocks have the same time, and this time must be the same as the 'exact true right' time.
- Problem: clock chips in which Ci clocks are based are not absolutely precise.
  - Examples of types of clocks:
    - **quartz crystal** clocks
      - Drift 10^{-6} seconds per each real second
    - "**High-precision**" clocks → Drift 10^{-7} a 10^{-8} seconds
    - **Atomic** clocks (the most precise ones) → Drift 10^{-13}

TAI and UTC times

- **International Atomic Time (TAI)**: Time established by the Bureau Internationale de l'heure (BIH) from all atomic clocks of all laboratories around the world
  - Atomic second: time a Cesium-133 atom needs to do 9.192.631.770 transitions

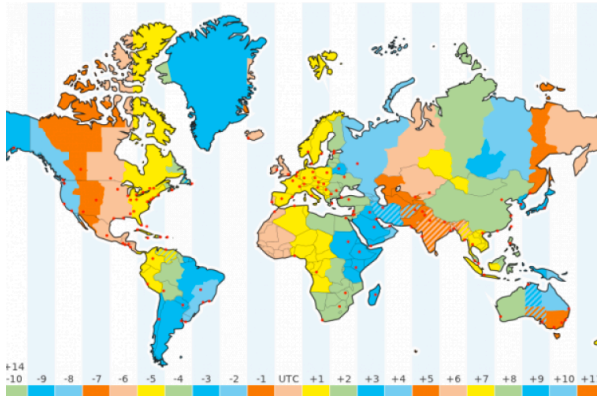


- **Universal Time:** units of time with astronomical origin, depending on the periods of rotation and translation of the Earth around the Sun
  - Important! The rotation period is gradually lengthening
    - The Earth is being held back by the friction of the tides, atmospheric effects and currents of convection in the nucleus



TAI and UTC times

- Problem: **atomic time** and **universal time** tend to desynchronized
  - Currently: one day TAI (86400 sec. TAI) is 3 milliseconds less than one solar day.
  - When the difference reaches 900 ms, BIH adds a second "leap" to the TAI counter, resulting in the **Coordinated Universal Time (UTC).**
  - UTC is the base for all official measures of time (politics, civil and military)
    - All time zones are defined in relation to UTC, the time zone centered on the Greenwhich meridian
    - https://www.utctime.net/



Synchronising physical clocks

- Suppose that each node "i" has a local clock C_i
  - We can synchronize it with the UTC time
    - UTC time available through radio stations (time signals), geostationary satellites, GPS.
- Unfortunately clock chips in which Ci clocks are based are not absolutely precise.
  - They normally have a relative error of 10^{-6}
    - This means that when 1.000.000 milliseconds have passed, the local clock might show that 999.999 or 1.000.0001 milliseconds have passed.
  - Therefore $\mathbf{C_i(t)}$ is more and more different from **t**
- **Solution**: to **periodically synchronize clocks** of all nodes using some reliable time source.

Content

- Global States and Time
  - Clocks, events and states
  - Clock synchronization algorithms: Cristian, Berkeley
  - Logical clocks and Vector clocks
  - Global states
- Examples of Algorithmic Problems in Distributed Systems
  - Mutual exclusion
  - Leader election
  - Consensus
- Algorithms in the presence of failures

Clock synchronization algorithms

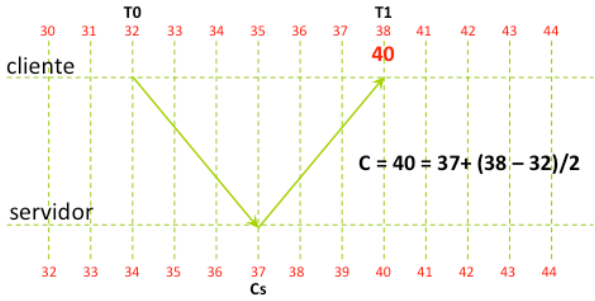
- **Synchronization algorithms of physical clocks:**
  - **Cristian Algorithm** (F. Cristian, 1989)
  - **Berkeley Algorithm** (Gusella & Zatti, 1989)
  - Network Time Protocol – NTP (1995)
    - Cristian and Berkeley designed for low latency networks (e.g. LAN)

Clock Synchronization algorithms: **Cristian Algorithm**

- **Cristian** Algorithm synchronizes the local clock **Cc** of a **client** computer with the local clock **Cs** of a **server** computer
- Considerations:
  - The server computer has a a **very precise clock**, possibly synchronized with others that are still more precise.
    - Eg. Synchronized with an UTC time
  - Clocks must not go back.
  - Synchronization implies message transmission, but transmission of messages through the network takes time.

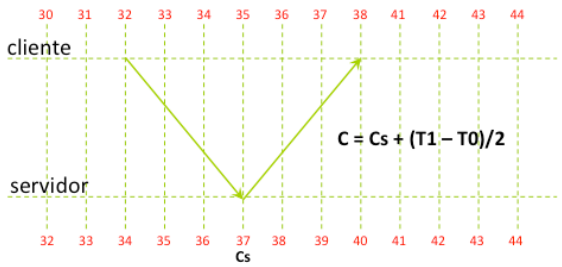
Clock Synchronization algorithms: **Cristian Algorithm**

- **Cristian Algorithm:**
  - The ***client*** asks the value of the clock to the *server* at instant **T0** (this instant is according to the client local clock Cc).
  - The ***server*** receives the request and answers with the value of its own clock: **Cs**



  - The answer arrives to the ***client*** at **T1** (according to the client local clock **Cc**).
  - The client sets its clock to the value **C = Cs + (T1 – T0)/2.**
  - The **client** updates its clock according to:
    - If **C > Cc**, the **client clock is set to Cc=C.**
    - If **C < Cc**, the client stops Cc the next **Cc-C** units of time.
      - It stores LAG = Cc – C and several clock ticks are discarded until a LAG time has been elapsed (this prevents the clock from being put back; instead, the clock is "stopped" some time).

Clock Synchronization algorithms: **Cristian Algorithm**



- **Features:**
  - It assumes that the sending time of both messages (request and answer) is practically the same
    - In T1 the value of Cs will have been incremented in (T1-T0)/2
  - If one of the two messages takes more time to be transmitted, then this adjustment is not right
    - Normally, the duration of both messages is the same.
  - Anyway, a perfect synchrony is nearly impossible (with any algorithm)
  - The right behaviour of a distributed application can be dependent on the values of local clocks only if it can tollerate the error margin inherent to the synchronization algorithm applied

Clock Synchronization algorithms: **Berkeley Algorithm**

- **Berkeley Algorithm -** Features
  - There is a set of nodes composed of:
    - A server, named S
    - N clients, named C_i
      - This algorithm does not assume any computer with a precise clock.
      - But one of the computers of the distributed system will act as a **server** and the others as its **clients**.
  - Each node has its own local clock
  - The goal is to synchronize the local clocks of all nodes between them.
  - Periodically, at server initiative, all nodes synchronize their clocks.

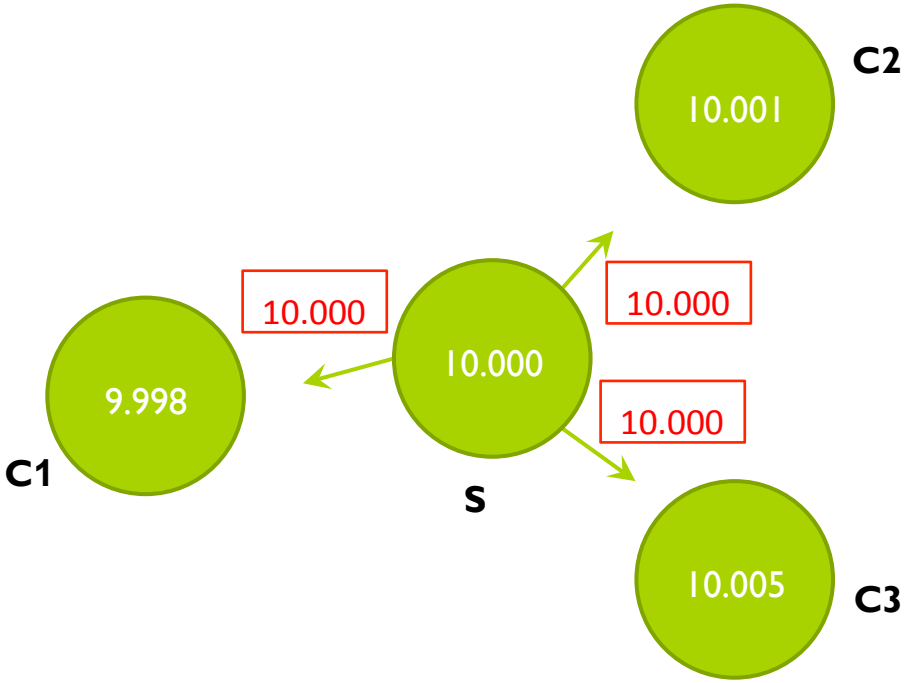
Clock Synchronization algorithms: **Berkeley Algorithm**

- **Berkeley Algorithm**
  - The **server** periodically broadcasts the value of its clock.
  - Each **client** calculates the difference $\mathbf{D_i}$ between its local clock and the clock value notified by the server in its message.
  - Each client notifies the difference D_i to the server
  - Given the replies, the **server**:
    1. computes the average difference (including the server)
    2. updates its own clock
    3. answers to every client with the difference that each of them has to apply in order to update its local clock.

Berkeley Algorithm: Example

- The server broadcasts its clock at **T0**

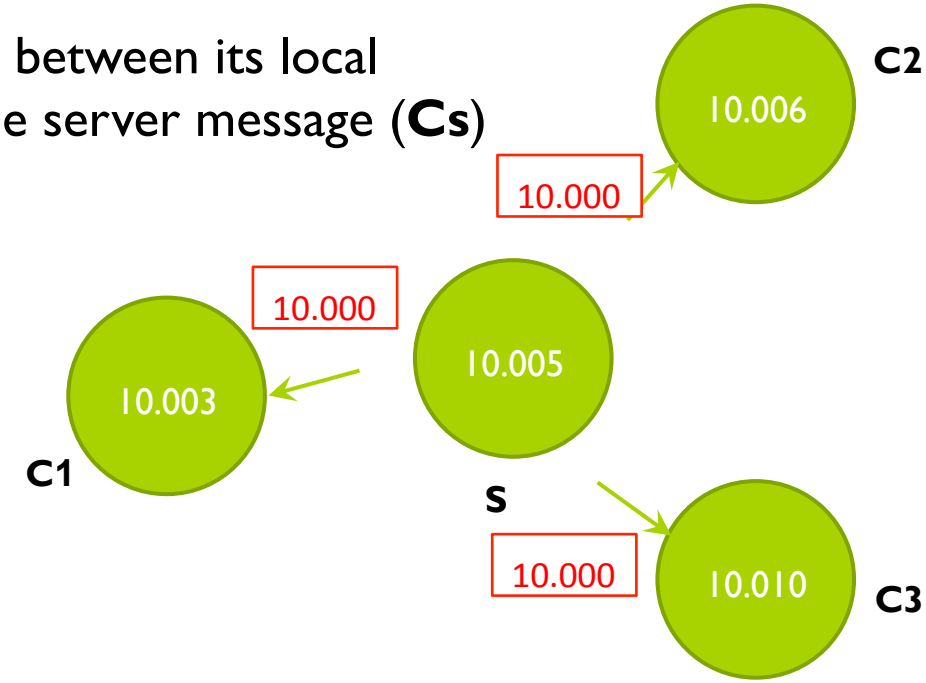
T0	10.000



Berkeley Algorithm: Example

- Messages arrive to clients
  - Example: Let us assume that messages arrive after 5 units of time. *To simplify, we assume that they arrive all together.*
- Each client calculates the difference $\mathbf{D_i}$ between its local clock (Cc) and the value received in the server message (**Cs**)
  - $D_i = Cc - Cs$

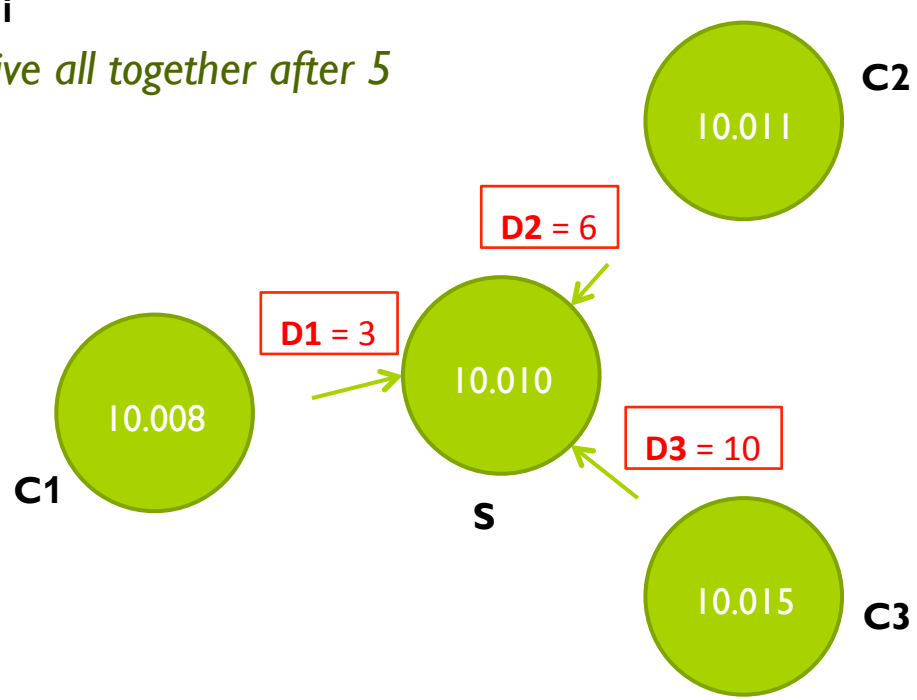
T0	10.000
D1	10.003 - 10.000 = 3
D2	10.006 - 10.000 = 6
D3	10.010 - 10.000 = 10



Berkeley Algorithm: Example

- Each client notifies this difference ($\mathbf{D_i}$) to the server
- The answer of each client arrives at $\mathbf{T1_i}$
  - *In this example, let us assume that they arrive all together after 5 units of time.*

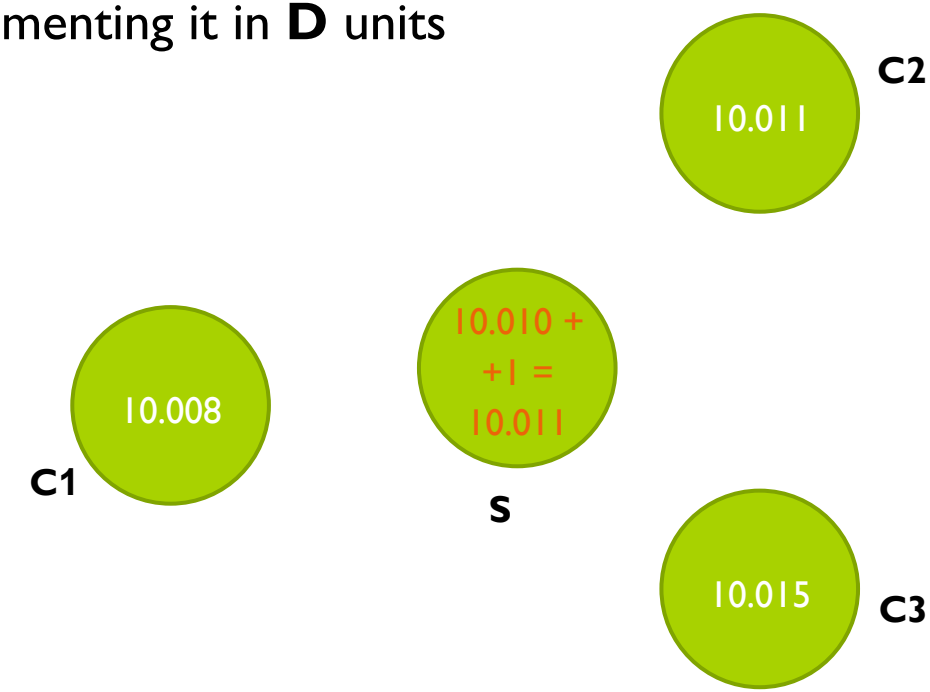
T0	**10.000**
D1	10.003 - 10.000 = **3**
D2	10.006 - 10.000 = **6**
D3	10.010 - 10.000 = **10**
T1i	**10.010**



Berkeley Algorithm: Example

- The server adjusts the difference (taking into account the time that its message took to arrive): $\mathbf{D_i' = D_i - (T1_i - T0)/2}$
- And calculates the mean (including the server)
  $\mathbf{D = \sum D_i' / (N+1)}$
- The server adjusts its own clock, incrementing it in **D** units

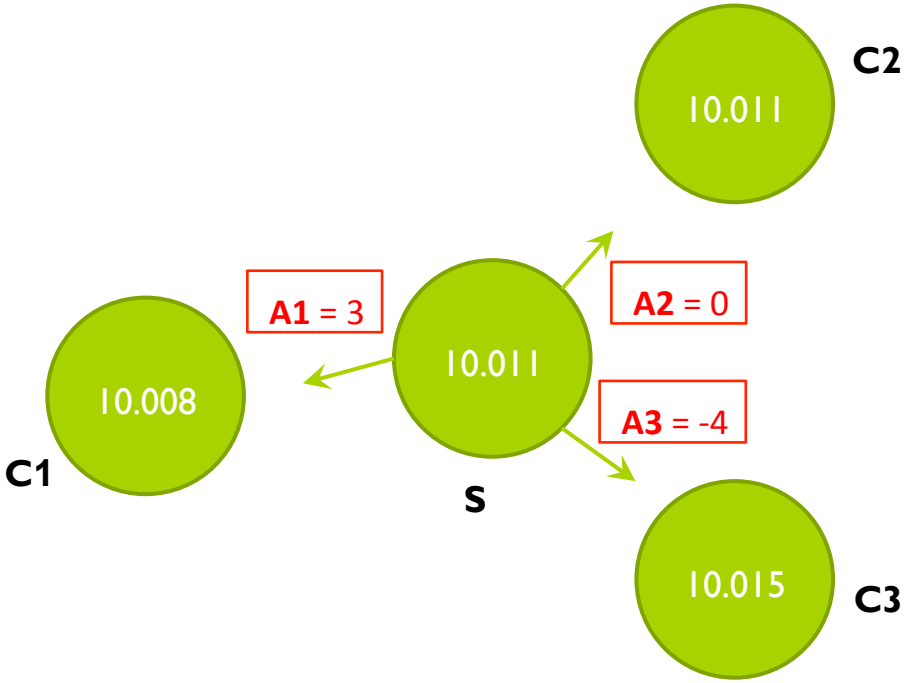
T0	**10.000**
D1	10.003 - 10.000 = **3**
D2	10.006 - 10.000 = **6**
D3	10.010 - 10.000 = **10**
T1i	**10.010**
D1'	3 - (10010 - 10000)/2 = **-2**
D2'	6 - (10010 - 10000)/2 = **1**
D3'	10 - (10010 - 10000)/2 = **5**
D	(-2 + 1 + 5 + 0)/4 = **1**



Berkeley Algorithm: Example

- It notifies this adjustment $A_i = D - D_i'$ to each client

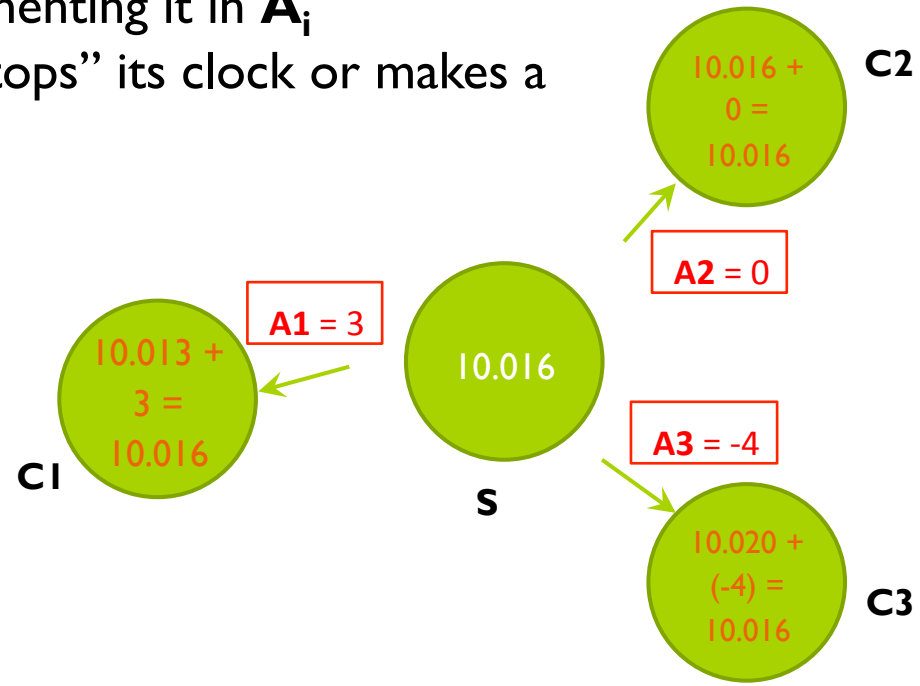
T0	**10.000**
D1	10.003 - 10.000 = **3**
D2	10.006 - 10.000 = **6**
D3	10.010 - 10.000 = **10**
T1i	**10.010**
D1'	3 - (10010 - 10000)/2 = **-2**
D2'	6 - (10010 - 10000)/2 = **1**
D3'	10 - (10010 - 10000)/2 = **5**
D	(-2 + 1 + 5 + 0)/4 = **1**
A1	1 - (-2) = **3**
A2	1 - 1 = **0**
A3	1 - 5 = **-4**



Berkeley Algorithm: Example

- The messages arrive to the clients
  - Let us assume that they arrive after 5 units time, all together.
- Each client adjusts its own clock, incrementing it in $\mathbf{A_i}$ (if the adjustment is negative, then it "stops" its clock or makes a gradual adjustment).

T0	**10.000**
D1	10.003 - 10.000 = **3**
D2	10.006 - 10.000 = **6**
D3	10.010 - 10.000 = **10**
T1i	**10.010**
D1'	3 - (10010 - 10000)/2 = **-2**
D2'	6 - (10010 - 10000)/2 = **1**
D3'	10 - (10010 - 10000)/2 = **5**
D	(-2 + 1 + 5 + 0)/4 = **1**
A1	1 - (-2) = **3**
A2	1 - 1 = **0**
A3	1 - 5 = **-4**



Clock Synchronization algorithms: **Berkeley Algorithm**

- Additional considerations:
  - It does not aim to synchronize all the clocks with the "real" instant, but to reach an agreement between the nodes
  - If any $\mathbf{D_i}$ difference is very different from the others, it is not taken into account
  - If the server fails, a leader election algorithm is started to choose another server
  - Exact synchronization is impossible due to the variability of time in the transmission of messages

Content

- Global States and Time
  - Clocks, events and states
  - Clock synchronization algorithms: Cristian, Berkeley
  - Logical clocks and Vector clocks
  - Global states
- Examples of Algorithmic Problems in Distributed Systems
  - Mutual exclusion
  - Leader election
  - Consensus
- Algorithms in the presence of failures

Logical clocks (Lamport, 1978)

- They indicate the order in which certain events occur, not the actual moment in which they happen.
- They are useful for many types of distributed applications that only require to know if an event has happened before another
- Unlike physical clocks, their timing is perfect but they have certain limitations
- Key ideas:
  - If two nodes do not interact (if they do not exchange messages), it is not necessary that they have the same clock value.
  - In general, it is important the global order in which events happen, but not the actual moment when they happen.

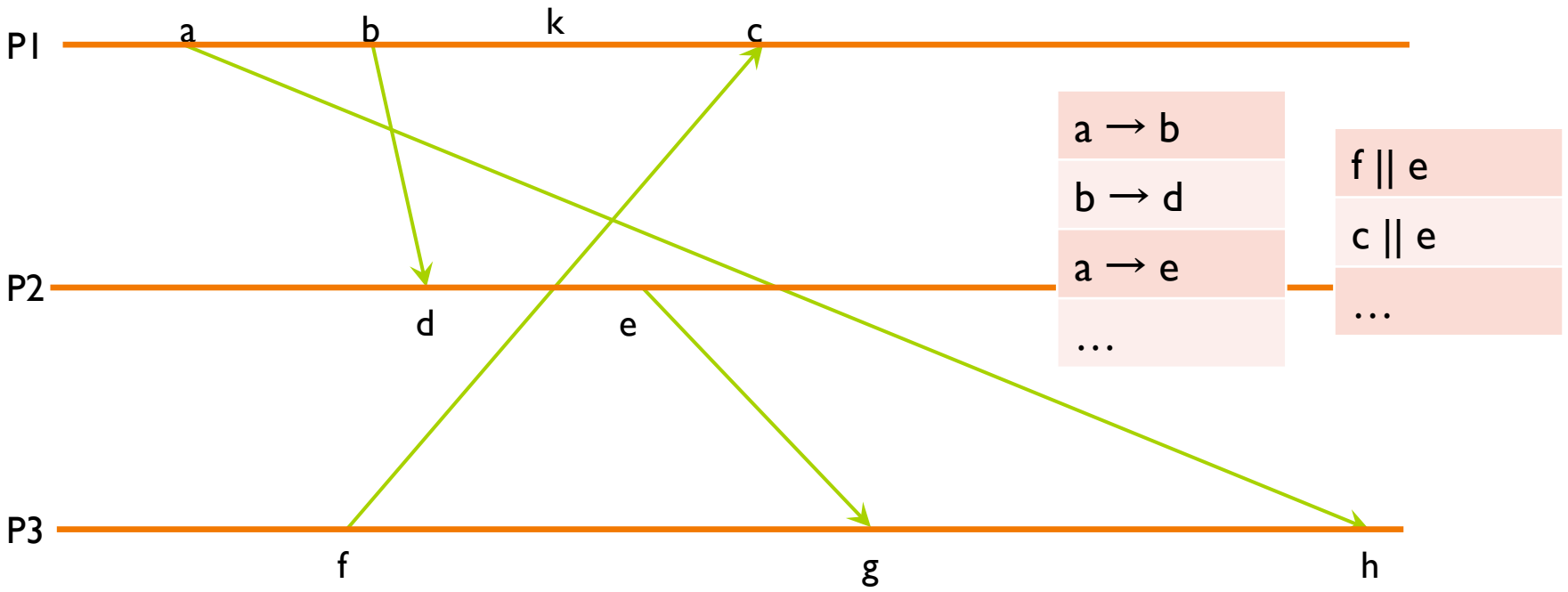
Logical clocks: "happens before" relationship

- To synchronize logical clocks, Lamport defined the relation "happens before"
- **a → b**
  - "*a* happens before *b*"
  - It means that all the nodes agree that first they see event ***a*** and afterwards they see event ***b***.
- The relation *happens-before* can be directly observed in two situations:
  - If ***a*** and ***b*** are events of the same node and ***a*** occurs before ***b***, then **a → b** is true.
  - If ***a*** is the event of sending a message *m* by a node, and ***b*** is the event of reception of *m* by another node, then **a → b** is true.

Logical clocks: "happens before" relationship

- It is a **transitive** relationship
  - if **a** → **b** and **b** → **c**, then **a** → **c**.
- Two events **x** and **y** are ***concurrent*** if we can not say which of them happens before the other: **x||y**
- The *happens-before* relation establishes a **partial order** among events in a system.
  - Not all events are related between them, since there can be concurrent events.

Logical clocks: "happens before" relationship



Logical clocks: Lamport's logical clocks

- It is necessary to have a way for measuring the *happens-before* relationship: using *Lamport's logical clock*.
- Logical clocks must mark the instant in which events occur, so that they associate a value to each event.
- We call **C(a)** to the value of the logical clock for event ***a***
- The logical clocks have to satisfy that:
  - If **a → b** , then **C(a) < C(b)**.
- We rewrite the conditions of the relation *happens-before*:
  - For two events in the same node *a* and *b*, if *a* happens before *b*, then C(a) < C(b)
  - For sending (a) and reception (b) events of a given message, then C(a) < C(b).
- Additionally, the clock must never decrease.

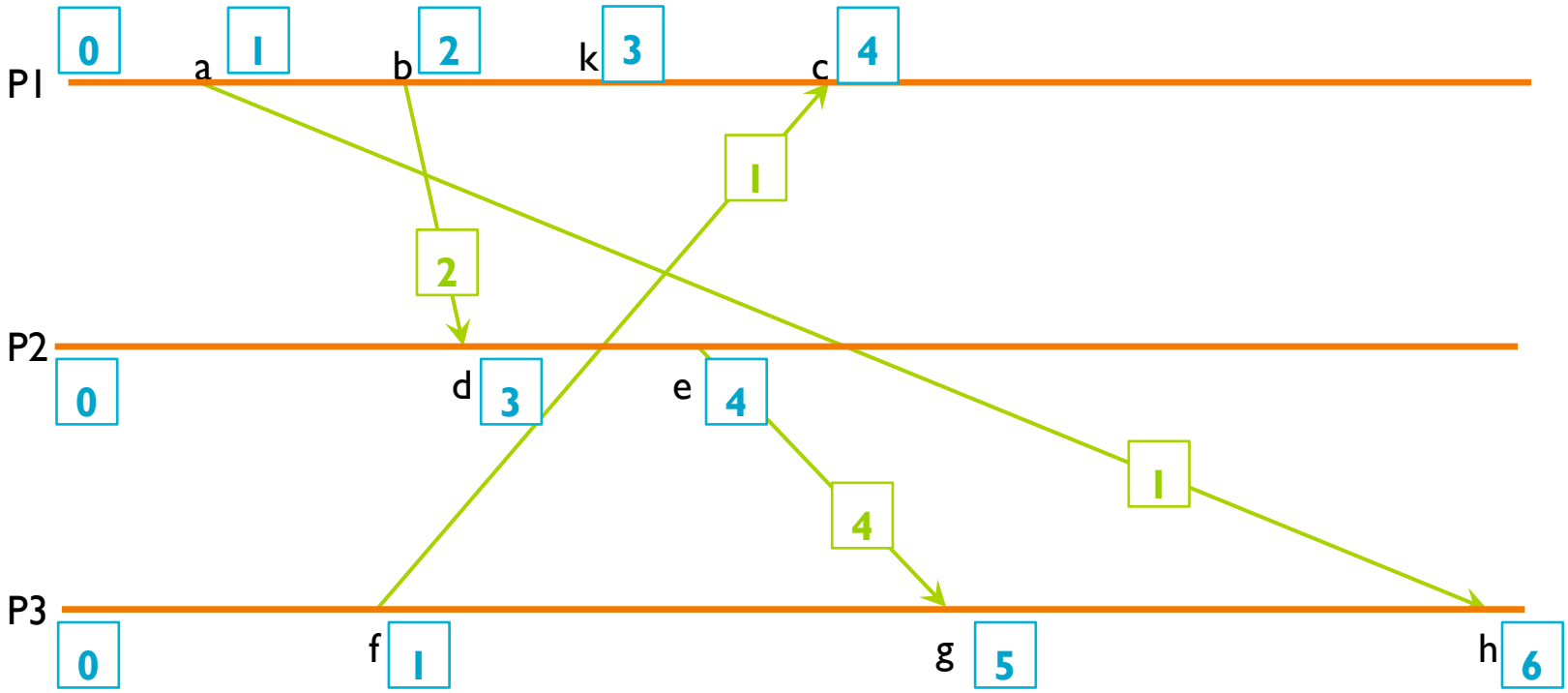
Logical clocks: Lamport's logical clocks

Lamport Algorithm:

- Every node has a counter (logical clock) C_p initialized to 0

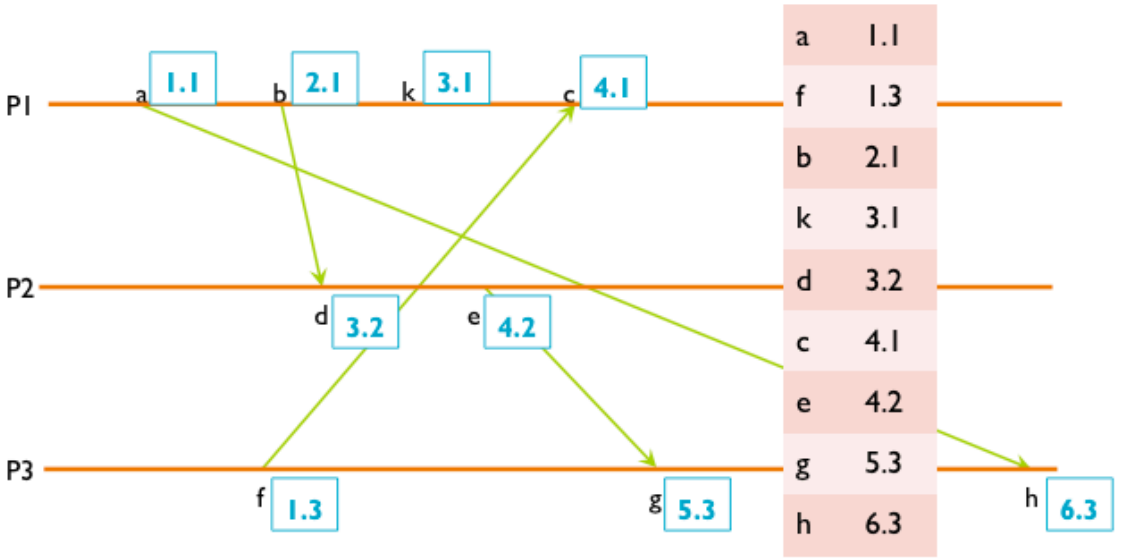
1. Each execution of an event (*sending a message or internal event*) in a node ***p*** increases the value of its counter **Cp** in 1.
2. Each message **m** sent by a node **p** is labeled (C_m) with the value of its counter C_p, i.e. $C_m = C_p$
3. When a node ***p*** receives a message, it updates its clock as follows: $C_p = \max(C_p, C_m) + 1$

Logical clocks: Lamport's logical clocks



Logical clocks: Lamport's logical clocks

- Features:
  - They stablish a parcial order among events.
    - Concurrent events are not ordered between them.
  - But you can obtain a total order by adding the node identifier as sufix



  - If **a → b**, the algorithm ensures that **C(a) < C(b)**
  - But if **C(a) < C(b)** we cannot decide whether **a → b** or **a || b**

Vector clocks

- The Lamport clock ensures that if **a → b** then **C(a) < C(b)**.

- However, if C(a) < C(b) we cannot determine anything. This means that the simple observation of the values of the clocks does not give clues about the ordering of the events.

- In some applications it is relevant to detect when two events are **concurrents**. With logical clocks you cannot know that.

Vector clocks

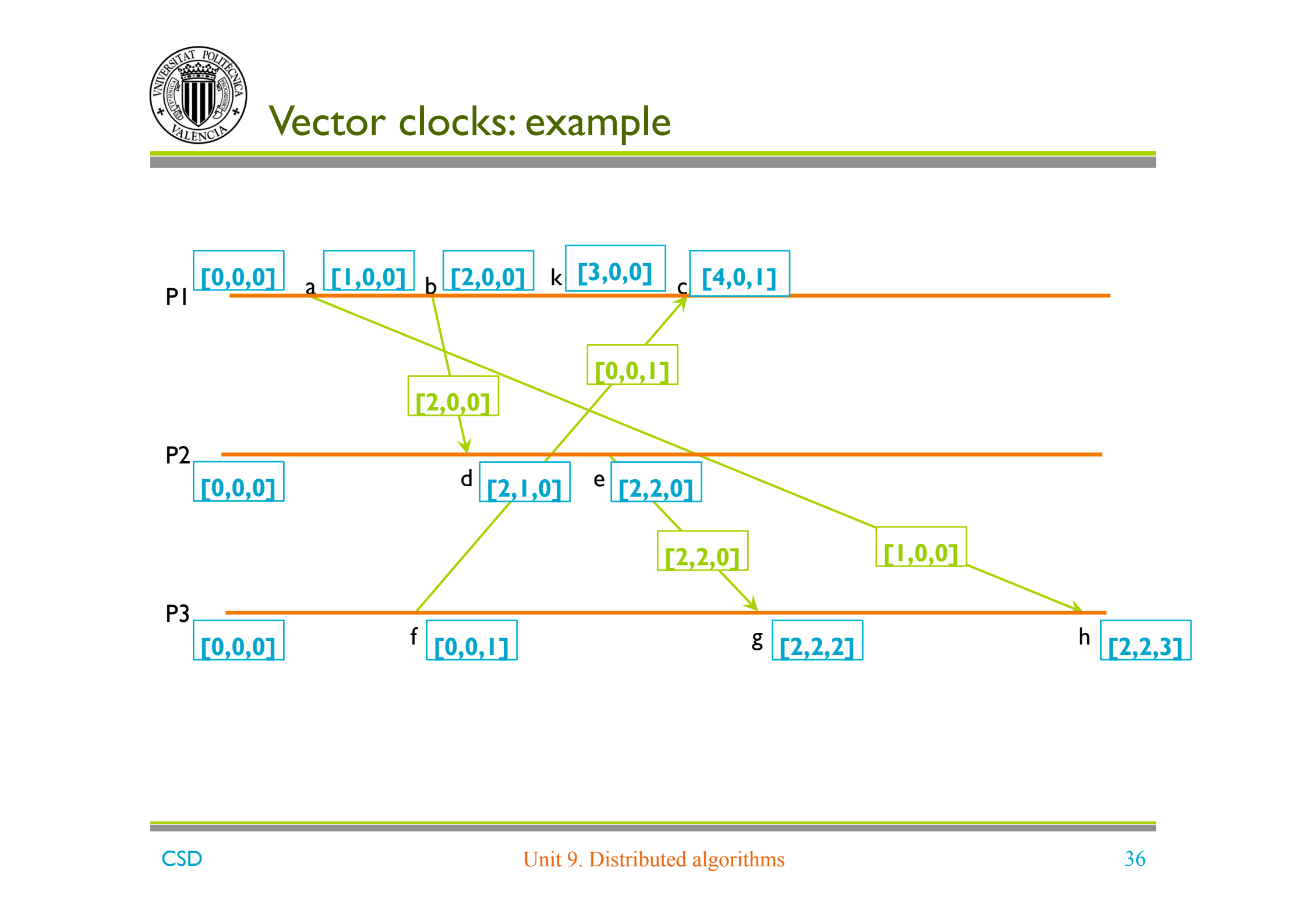
- Solution: using **vector clocks**
  - Vector clocks link an array value **V(x)** to each event x (sending or receiving a message) of a distributed system.
  - They help determining whether an event **happens before** another or if two events are **concurrent.**
  - Given **N** nodes, each node **p** mantains an array of **N** temporal marks $\mathbf{V_p}$ such as:
    - $\mathbf{V_p[p]}$ is the number of events that have occurred in p
    - If $\mathbf{V_p[q]=k}$, then **p** knows there have been at least **k** events in node **q**.

Vector clocks: algorithm

- Each node **p** has a vector clock $\mathbf{V_p}$ initialized with all its elements to 0

1. Node ***p*** increments $\mathbf{V_p[p]}$ in 1 unit each time it sends a message or executes an internal event.
2. A message ***m*** sent by a node ***p*** carries its vector clock. That is, $\mathbf{V_m = V_p}$.
3. When **p** receives a message *m*, it increments $\mathbf{V_p[p]}$ in 1 and it updates its vector clock selecting the maximum value between its local value and the value of the clock at the message, for each of its components:

$$\forall i, 1 \le i \le N, V_p[i] = \max(V_p[i], V_m[i])$$



Vector clocks: example

Vector clocks: features

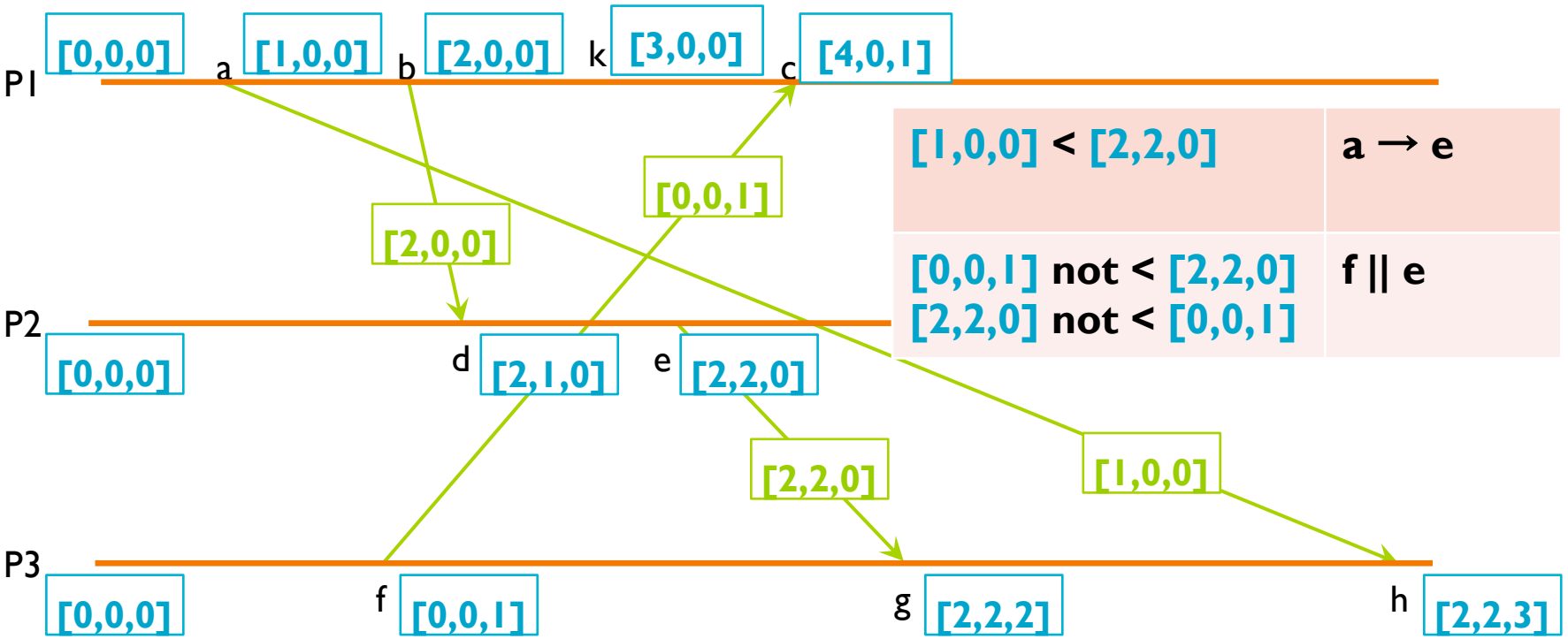
- **V(a) < V (b)** if:
  - all the components of V(a) is lower or equal to the respective component of V (b)
  - and moreover there is at least one component that is stricly **lower**

$$V(a) \leq V(b) \Leftrightarrow \forall i, 1 \leq i \leq N, V(a)[i] \leq V(b)[i]$$

$$V(a) < V(b) \Leftrightarrow V(a) \leq V(b) \wedge \exists i, 1 \leq i \leq N, V(a)[i] < V(b)[i]$$

- If **a → b** then **V(a) < V (b)**
- If **V(a) < V (b)** then **a → b**
- If **V(a) < V (b) does not hold** and **V(b) < V (a) does not hold** then **a||b**

Vector clocks: example



[1,0,0] < [2,2,0]	a → e
[0,0,1] not < [2,2,0] [2,2,0] not < [0,0,1]	f || e

Content

- Global States and Time
  - Clocks, events and states
  - Clock synchronization algorithms: Cristian, Berkeley
  - Logical clocks and Vector clocks
  - Global states
- Examples of Algorithmic Problems in Distributed Systems
  - Mutual exclusion
  - Leader election
  - Consensus
- Algorithms in the presence of failures

Global state: necessity

- **Global state:** local state of each process + state of each communication channel.
  - *State of each process*: state of the variables that interest us.
  - *State of each communication channel*: messages sent and not yet delivered.

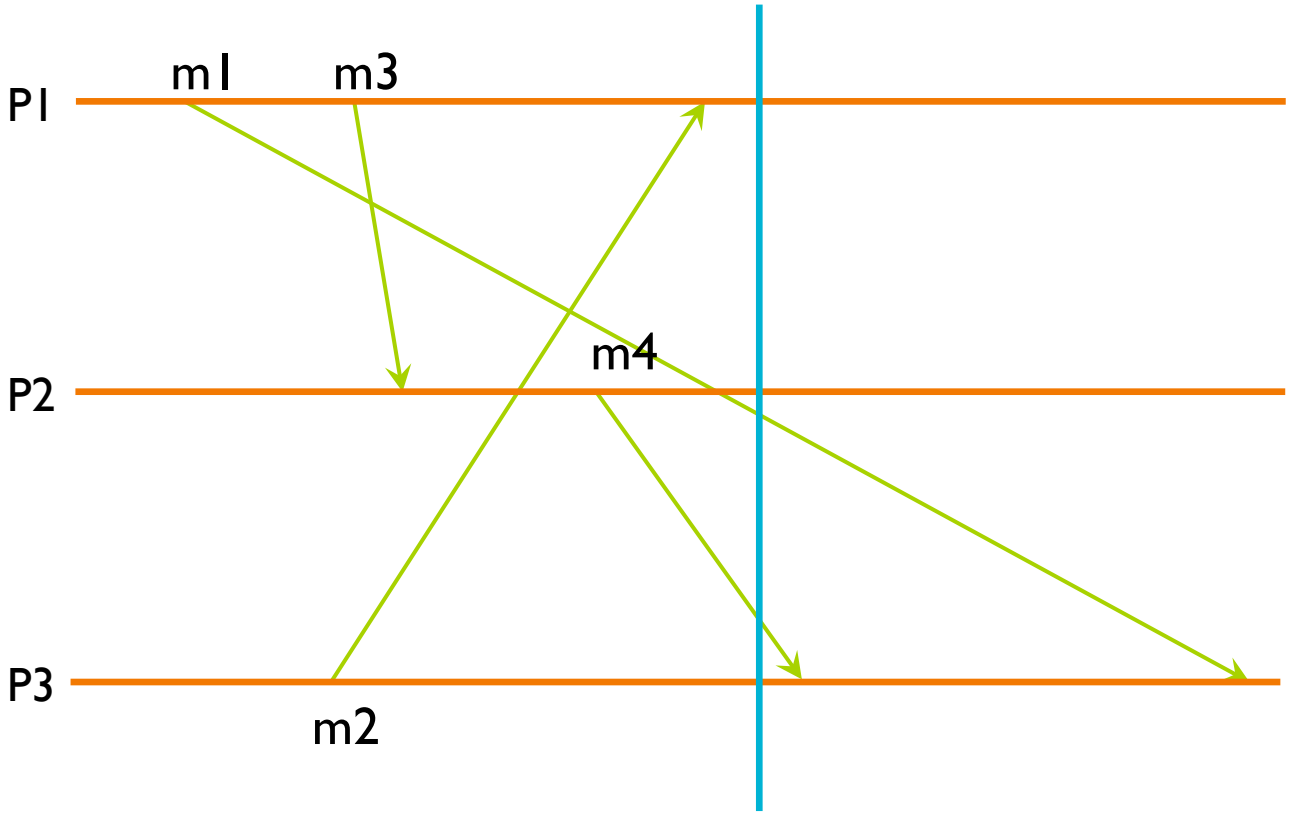
- Examples of usage of the global state:
  - Unused remote object collector (Garbage collection)
    - It is determined whether any node keeps any reference to the remote object
    - And if in any message in transit there is any reference to this remote object
    - If no reference is found, the object is deleted
  - Detection of distributed deadlock
    - Is the system in a state of mutual lock?
  - Distributed termination detection
    - Is the distributed algorithm finished? All processes may be stopped ... but there may be a message on the way.

Global state: necessity

- Preliminary concepts
  - A **distributed snapshot** reflects a state that could have occurred in the system.
  - The snapshots reflect only consistent states:
    - If P has received a message from Q in a snapshot, Q had sent before that message to P.
  - The notion of "global state" is reflected by the concept of **cut**.

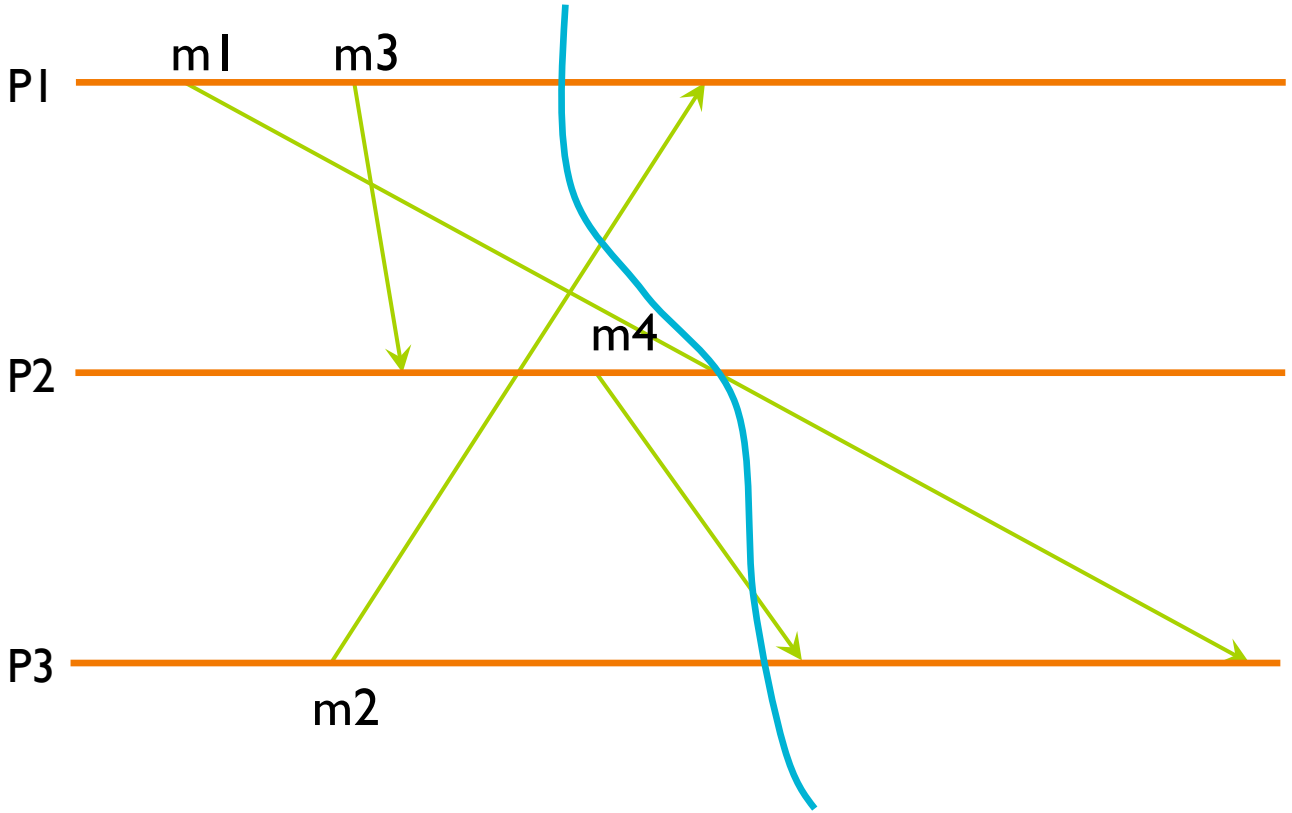
Global state
Snapshot: **precise snapshot**

- The precise snapshot is not factible. For a precise snapshot it is required that all nodes have their clocks perfectly synchronized.



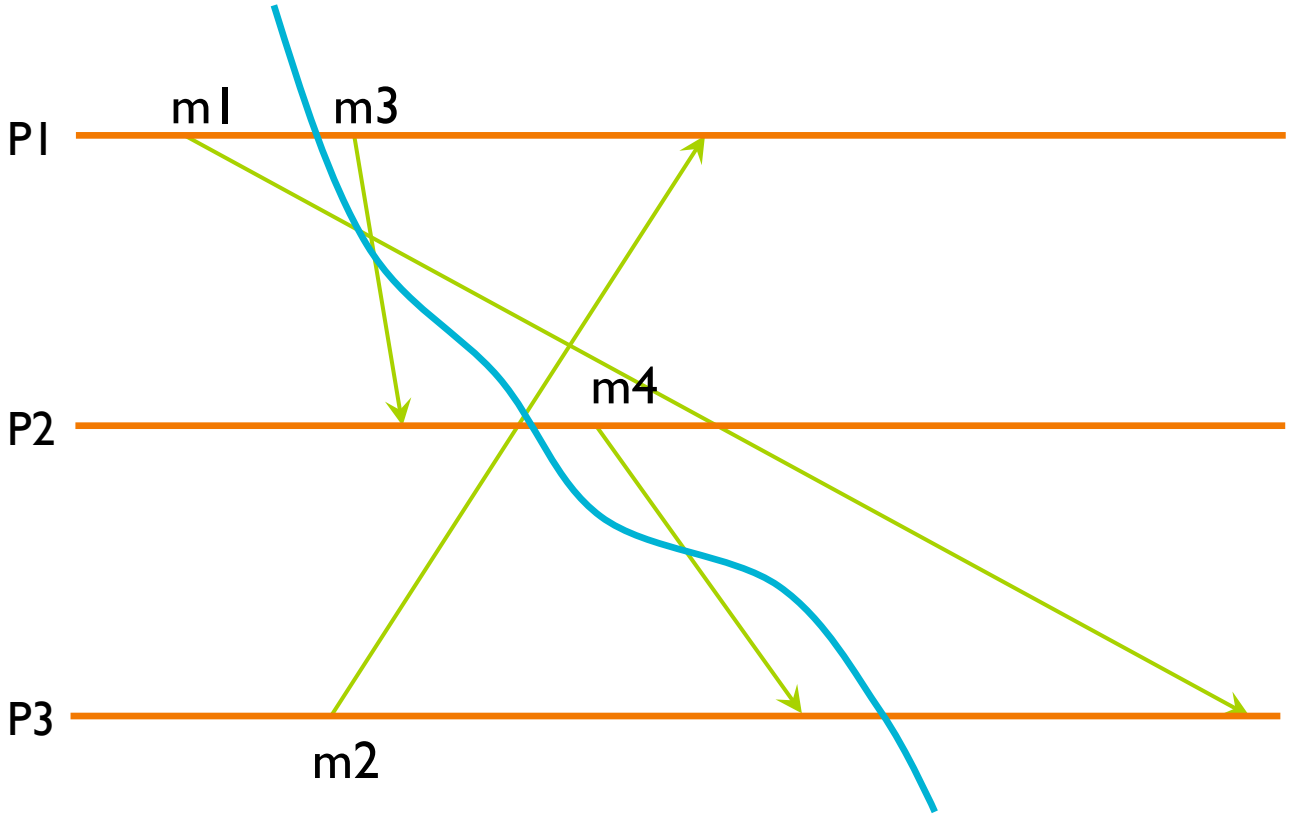
Global state
Snapshot: **consistent** snapshot

- The snapshot is **consistent** when it does not show any message delivery without its respective sending.



Global state
Snapshot: **inconsistent** snapshot

- The snapshot is **inconsistent** when it shows the delivery of some messages (e.g. m3 and m4) but not its sending.



Global state
Chandy-Lamport algorithm (1985)

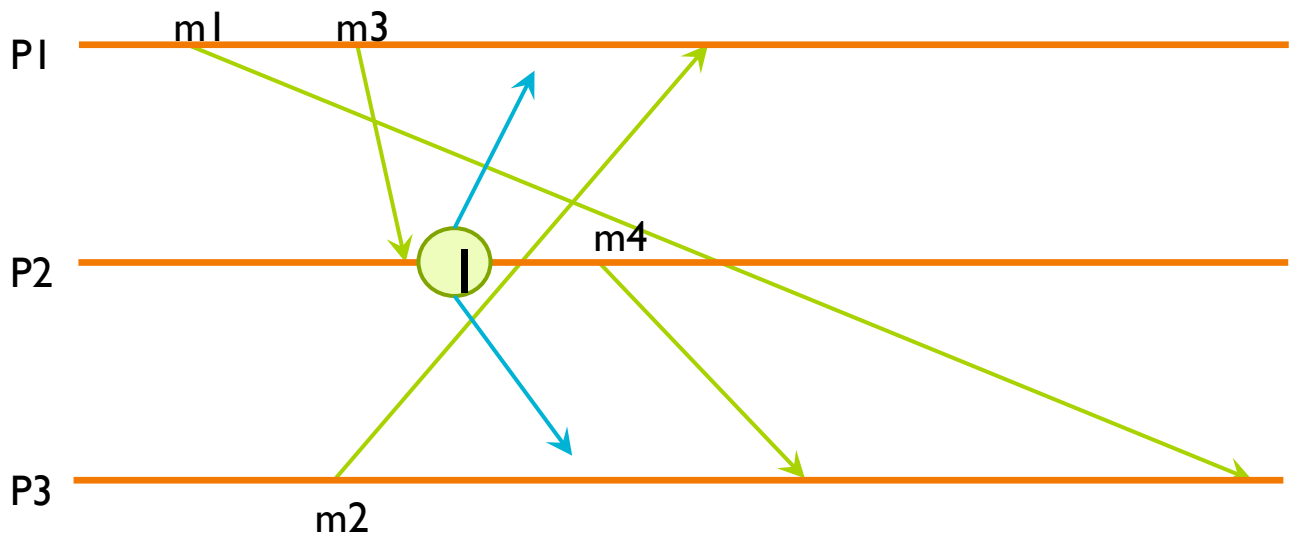
- This algorithm creates a consistent snapshot of the global state of a distributed system
- We assume that:
  - The distributed system is formed by several nodes
  - Network with a complete topology, there is a channel between every pair of nodes
  - All channels are:
    - Reliable
    - They transmit their messages in FIFO order
    - They are unidirectionals, between two nodes **p** and **q** there are two channels (**p**,**q**) and (**q**,**p**)

Global state
Chandy-Lamport algorithm

- **Chandy-Lamport Algorithm:**
  1. Initiator node *p* stores its local state, then it sends a MARK message to the rest of nodes.
  2. When a node ***p*** receives the MARK message through channel **c**:
     a) If it has not stored its local state yet:
        - It stores its local state and sets this channel **c** as empty.
        - Then, previous to any other message, it sends **MARK** to all the other nodes and starts registering all the messages that arrive from the other channels different to **c**.
     b) If it has already stored its local state:
        - It stores all messages received by this channel c (maybe there are no messages) and stops registering the activity of that channel.
  3. When a node **p** has received MARK by all its incoming channels
     - It sends its local state previously stored and the state of its incoming channels to the initiator node (except if he is the initiator) and finishes its participation in the algorithm.

Global state
Chandy-Lamport algorithm

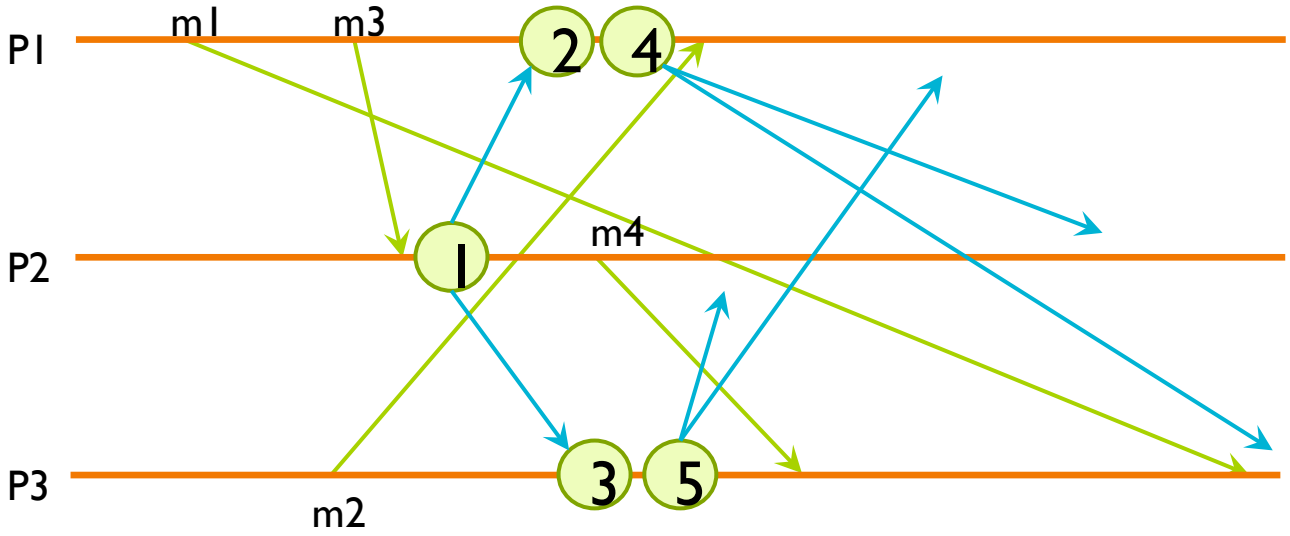
- The initiator node
  - Stores its local state and sends a MARK message to the rest of nodes (Step 1)



Local state	Step	Value
P1		
P2	1	St. P2
P3		
Channel State	**Step**	**Value**
P1,P2		
P2,P1		
P1,P3		
P3,P1		
P2,P3		
P3,P2		

Global state
Chandy-Lamport algorithm

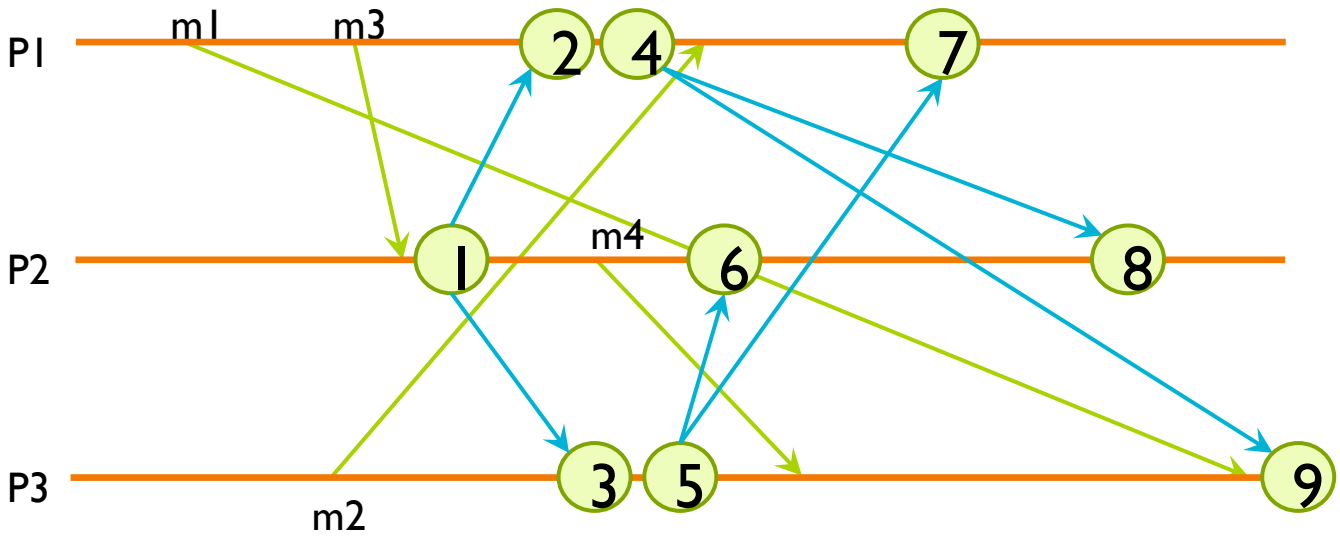
- When a node **p** receives MARK by channel **c**, if it has not registered its state yet
  - It registers its local state and marks channel **c** as empty (steps 2,3)
  - Then, before sending any other message, it sends MARK to the other nodes and it starts registering the messages that come by other channels different to **c** (steps 4,5)



Local state	Step	Value
P1	2	St. P1
P2	1	St. P2
P3	3	St. P3
Channel State	**Step**	**Value**
P1,P2		
P2,P1	2	Empty
P1,P3		
P3,P1		
P2,P3	3	Empty
P3,P2		

Global state
Chandy-Lamport algorithm

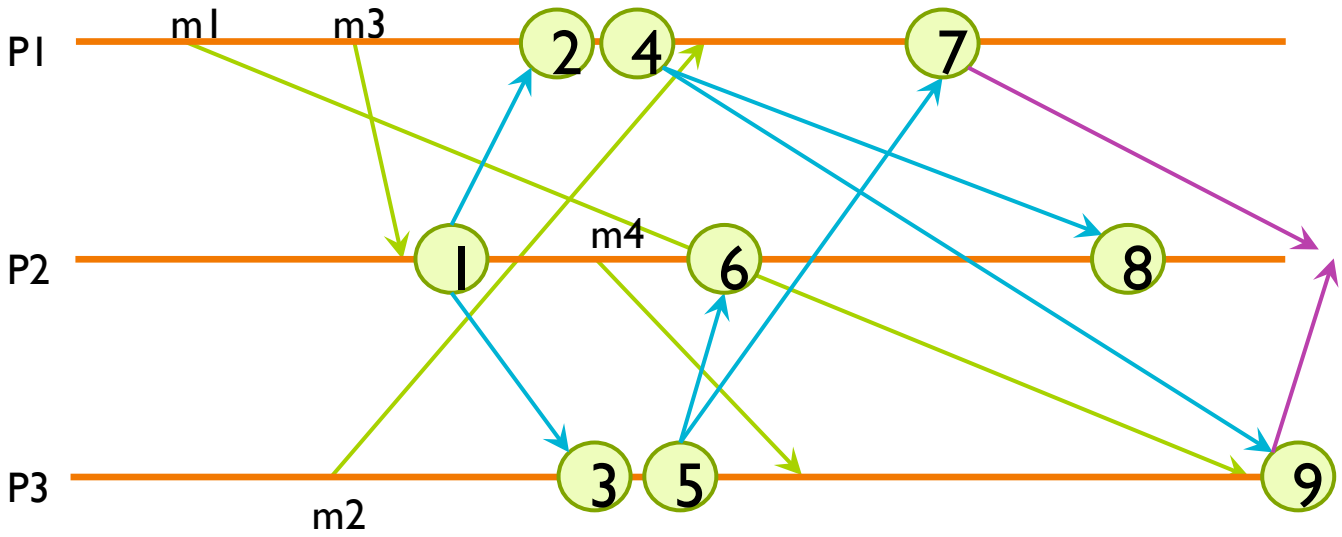
- When a node **p** receives MARK by channel **c**, and it had previously registered its state
  - It stores all messages received by this channel c (there might be none) and stops registering the activity of this channel (steps 6,7,8,9)



Local state	Step	Value
P1	2	St. P1
P2	1	St. P2
P3	3	St. P3
Channel State	**Step**	**Value**
P1,P2	8	Empty
P2,P1	2	Empty
P1,P3	9	m1
P3,P1	7	m2
P2,P3	3	Empty
P3,P2	6	Empty

Global state
Chandy-Lamport algorithm

- When a node **p** has received MARK by all its incomming channels
  - It sends its local state (previously stored) and the state of its incomming channels to the initiator node (unless he is the initiator) and it finishes the algorithm (Steps 7,8,9)



Local state	Step	Value
P1	2	St. P1
P2	1	St. P2
P3	3	St. P3
Channel State	**Step**	**Value**
P1,P2	8	Empty
P2,P1	2	Empty
P1,P3	9	m1
P3,P1	7	m2
P2,P3	3	Empty
P3,P2	6	Empty

Global state
Chandy-Lamport algorithm

- Additional considerations:
  - The algorithm can work as described here if you add the identifier of the initiator node to the MARK message.
    - Then nodes that receive MARK can know who is the initiator and can later send it the information with its registered state and received messages.
  - In the algorithm described by Chandy & Lamport, authors do not specify how the recorded information by each node has to be collected
    - You can use the technique explained here
    - Or you can use other existing algorithms for collecting information
      - For example, each node can broadcast its recorded information to all outgoing channels, so then all nodes can finally know which is the global state of the system.

Content

- Global States and Time
  - Clocks, events and states
  - Clock synchronization algorithms: Cristian, Berkeley
  - Logical clocks and Vector clocks
  - Global states
- Examples of Algorithmic Problems in Distributed Systems
  - Mutual exclusion
  - Leader election
  - Consensus
- Algorithms in the presence of failures

Examples of Algorithmic Problems in DS

- Examples of Algorithmic Problems in Distributed Systems:
  - Routing
  - Broadcast messages to groups
  - Garbage collection
  - Replication protocols
  - Group membership
  - Failure detection
  - Distributed commitment
  - Leader election
  - Consensus
  - Mutual exclusion
  - Deadlock detection
  - Distributed termination detection
  - ….

Examples of Algorithmic Problems in DS

- In this unit we will focus on:
  - Mutual exclusion
  - Leader election
  - Consensus

Content

- Global States and Time
  - Clocks, events and states
  - Clock synchronization algorithms: Cristian, Berkeley
  - Logical clocks and Vector clocks
  - Global states
- Examples of Algorithmic Problems in Distributed Systems
  - Mutual exclusion
  - Leader election
  - Consensus
- Algorithms in the presence of failures

Distributed Mutual exclusion

- Problem to solve: access to a critical section (e.g. a shared resource) of several processes that are in different nodes
  - Avoid inconsistences
  - Solutions need to ensure access with mutual exclusion by processes

- Proposed solutions:
  a) Centralized algorithm
  b) Distributed algorithm
  c) Algorithm for rings

Correction conditions for mutual exclusion

- **Security**: at most, only one process can be running within the critical section at a given time (i.e. mutual exclusion)
- **Liveness**: any process that wants to enter the critical section achieves it at some point.
  - **Progress**: if the critical section is free and there are processes that wish to enter, one of them is selected in finite time.
  - **Bounded waiting**: if a process wants to enter the critical section, it should only wait a finite number of times for others to enter before it enters.

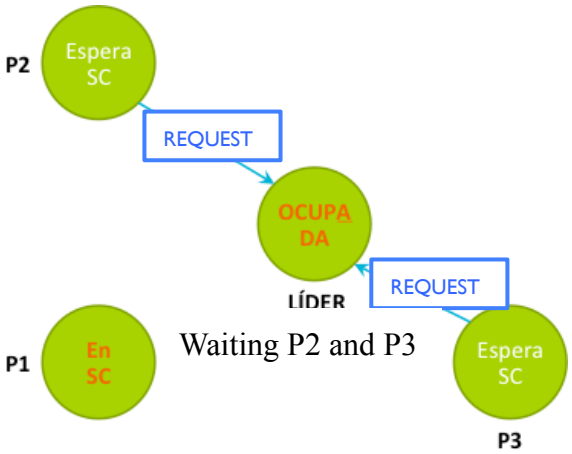
Mutual exclusion
a) Centralized algorithm (Lamport 1978)

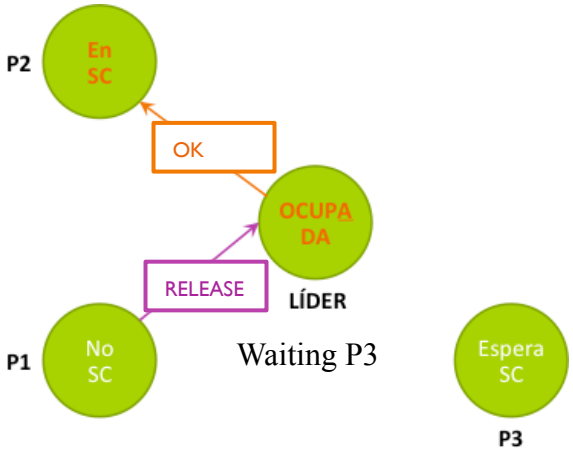
- A node is selected as coordinator (leader).
  - This leader will control the access to the critical section

1. When a node wants to enter into the critical section, it sends a **REQUEST** message to the leader, asking for permission.
2. If no other node is in the critical section, the leader replies **OK**.
   - The leader registers that now the critical section is occupied
   - The requesting node begins using the critical section
3. If the critical section is occupied, and another node requests the leader to use it…
   - The leader registers the identifier of the node requesting to access the critical section and does not answer him.
   - The requesting node remains blocked waiting for the answer.

Mutual exclusion
a) Centralized algorithm

4. When a node leaves its critical section, notifies the leader with a **RELEASE** message.
   - If there is any node blocked waiting for a permit to go inside the critical section, the leader sends **OK** to this node.
   - If there are several nodes waiting, the leader can select, for example, the first node that requested the critical section.
   - If there is not any node waiting for accessing the critical section, then the leader registers that the critical section is now free





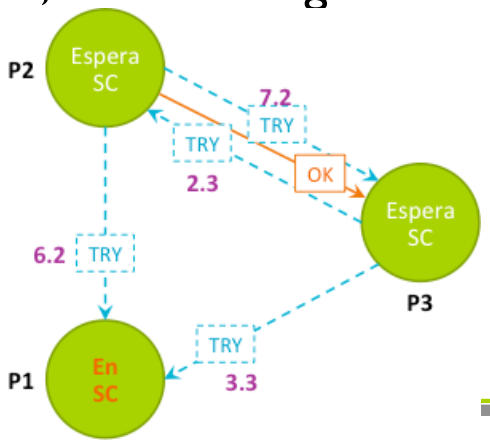
Mutual exclusion
b) Distributed algorithm (Ricart-Agrawala 1981)

- We suppose that all events are ordered (e.g, using Lamport logical clocks with their node identifiers).

1. When a process wants to enter the critical section, it broadcasts a **TRY** message to all processes of the system.
2. When a process receives a **TRY** message:
   a) If it is not in its critical section neither wants to enter it, then it replies with **OK**.
   b) If it is in its critical section, it does not answer and queues the request.
   c) If it is not in its critical section, but it wants to enter, it compares the mark of the incoming message with the one it sent to the rest of processes. The lowest number wins:
      - If the incoming message is lower, it answers **OK**.
      - If it is higher, it does not answer and queues the message.
3. A process enters the critical section when it receives **OK** from all.
4. When it leaves the critical section, it sends **OK** to all the processes that sent the messages retained in its queue.

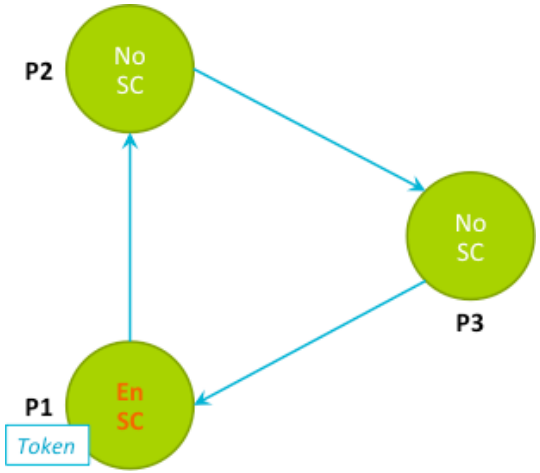
Mutual exclusion
b) Distributed algorithm

- **Features:**
  - Messages are labeled with the Lamport's logical clock plus the ID of the sender node.
  - When a node receives a TRY message, and he is not in its critical section but he wants to enter it:
    - If the label of the incoming message is lower than the label of the message he sent, then he answers OK
    - If it is higher, then he does not answer and stores the identity of the sender of the TRY message
  - Then, when a critical section gets free, the "winning" node will be the first one to go inside



Mutual exclusion
c) Algorithm for rings (Le Lann 1977)

- There is not any coordinator node
- Coordination is solved by ring communication and the use of a *token* that circulates through the ring
  - Requires reliable communication
  - If a node falls down with the token, then a new *token* must be built.
- Only the node with the *token* can be inside the critical section.



Mutual exclusion
c) Algorithm for rings

- **Algorithm for rings:**
  - Initial situation: the critical section is free and there is one single *token* in the ring, in one of the nodes.
1. If the node with the token does not want to go inside its critical section, then it sends the *token* to next node.
2. A node waits to go inside its critical section if it does not have the *token*.
3. A node goes inside its critical section when it gets the *token*
   - And it does not pass the token to next node until it finishes its critical section.
- Therefore, if a node wants to enter its critical section, it has to wait that the critical section gets free and that it obtains the token.

Mutual exclusion Comparative

Algorithm	Messages to enter/leave	Delay before entry (in message rounds)	Problems
Centralized	3	2	Coordinator crash
Distributed	2(n-1)	2(n-1)	Crash of any process
Ring	1 to infinite	0 to n-1	Lost of token

Content

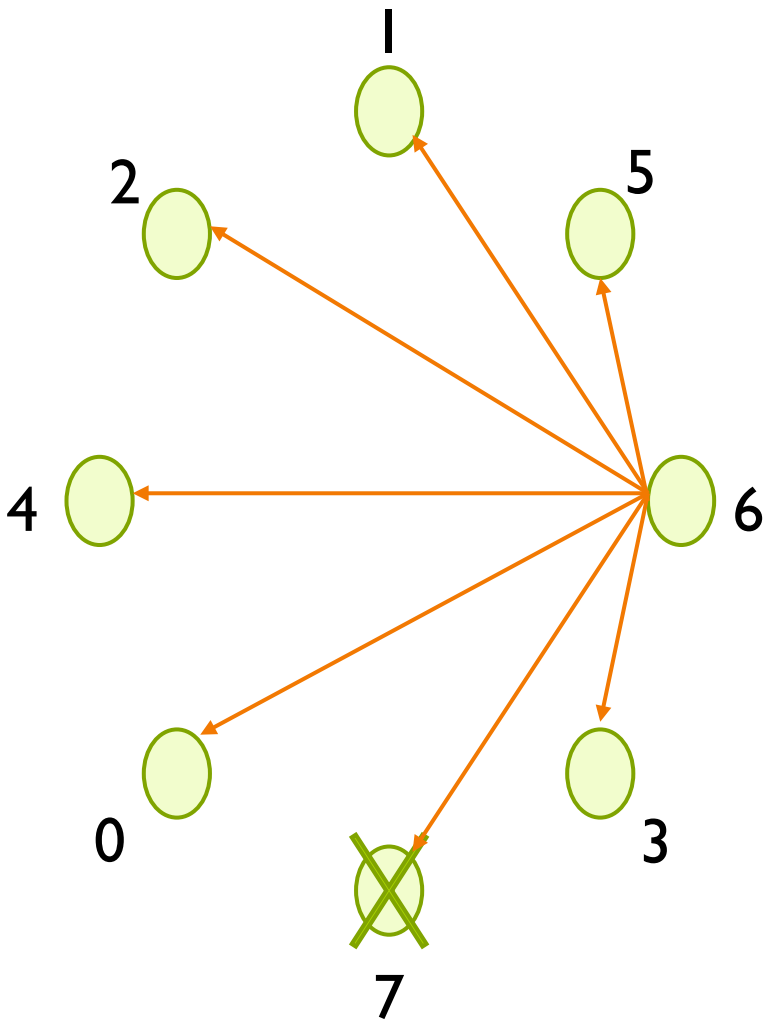
- Global States and Time
  - Clocks, events and states
  - Clock synchronization algorithms: Cristian, Berkeley
  - Logical clocks and Vector clocks
  - Global states
- Examples of Algorithmic Problems in Distributed Systems
  - Mutual exclusion
  - Leader election
  - Consensus
- Algorithms in the presence of failures

Leader election: concept

- Many distributed algorithms require that a node act as a leader or coordinator
  - For simplification
  - For reducing the number of messages needed to reach an agreement
- The leader election is done at the beginning or when the nodes detect that the leader does not answer any more (it might have fallen down)
- It is assumed that all nodes know the identifiers of the other nodes of the distributed system.
- Leader election is a particular case of consensus, but we study it separately given its importance.
- **Examples of leader election algorithms:**
  - Bully algorithm
  - Algorithm for rings

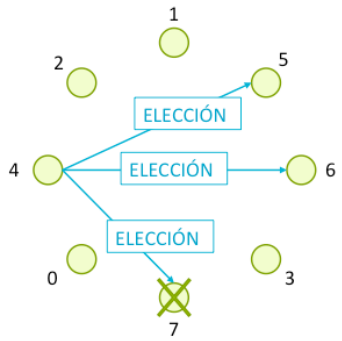
Leader election:
a) **Bully algorithm** (García-Molina 1982)

- It is used to select a new leader
- It is started by one of the nodes of the distributed system
- The new leader will be the active node with the highest identifier
- Once selected, the new leader will notify the others that he is the leader

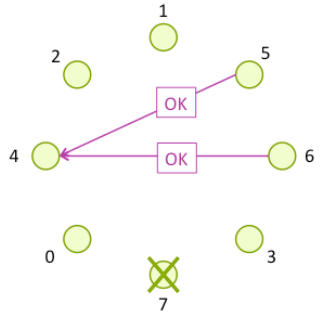


Leader election:
a) **Bully algorithm**

1. When a node wants to start an election (e.g., because the current leader does not answer), he sends **ELECTION** to all the nodes with identifier (ID) higher than its own.



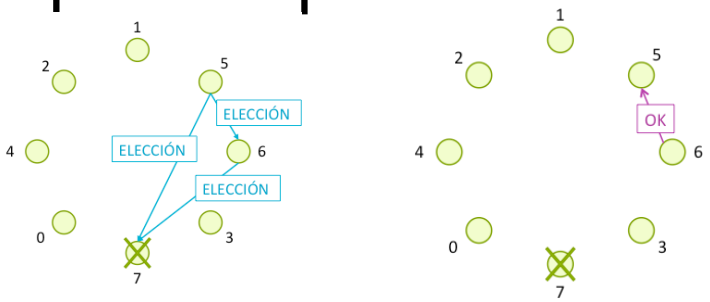
2. When a node receives **ELECTION**, sends **OK** to whom sent it (to let this node know that he also participates in the election and that he will win this other node).



3. When a node receives at least one **OK** message, he stops participating in the process.

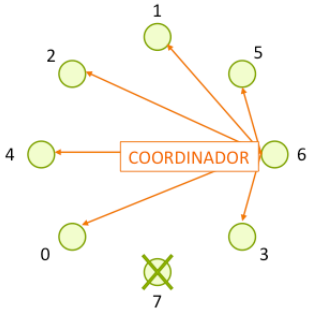
Leader election:
a) **Bully algorithm**

The previous steps are repeated ….



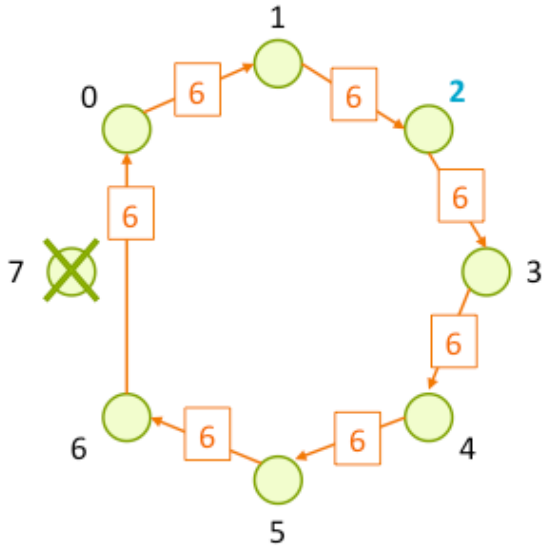
4. Until a node **does not receive any answer**. Then he is the new leader.

5. The new leader broadcast a **COORDINATOR** message to all the other nodes, to let them know that he is now the new leader.



Leader election:
b) **Algorithm for rings**

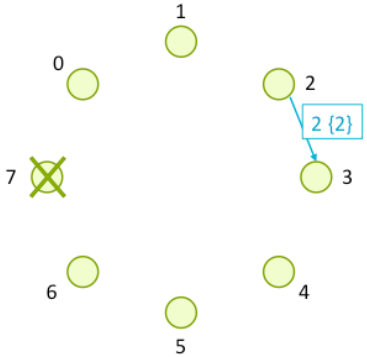
- The nodes are located in a logical ring and they send messages through the channels of this ring
- This algorithm is used to select a new leader, as initiative of one of the nodes of the ring
- Once selected, the identity of the new leader will be propagated through the ring



Leader election:
b) **Algorithm for rings**

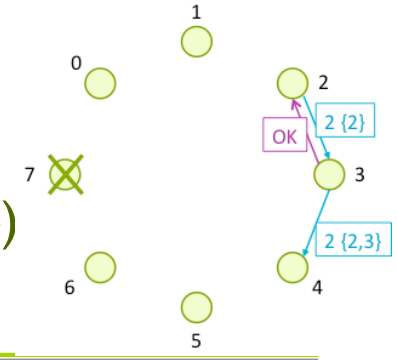
1. When a node want to begin a leader election:
   - he builds an ELECTION message with two fields:
     - Initiator
     - List of participant nodes
   - It assigns its identifier to both fields and sends

   the message to the next node in the ring



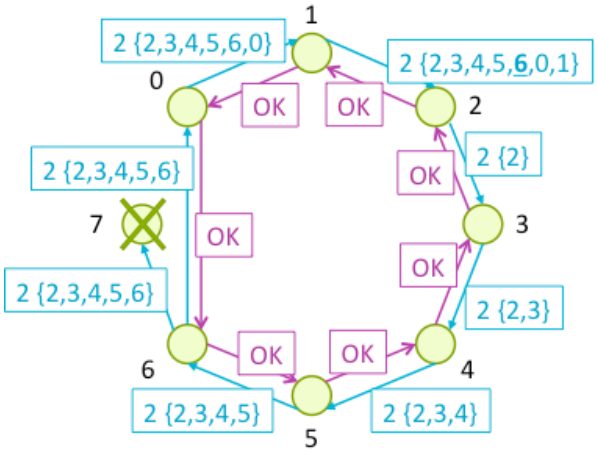
2. When a node receives an ELECTION message, if he is not the initiator:
   - It answers the sender node with an OK message
   - He inserts itself n the list of participant nodes
   - He sends the message to next node
   - If no confirmation arrives (i.e. there is not any OK message)

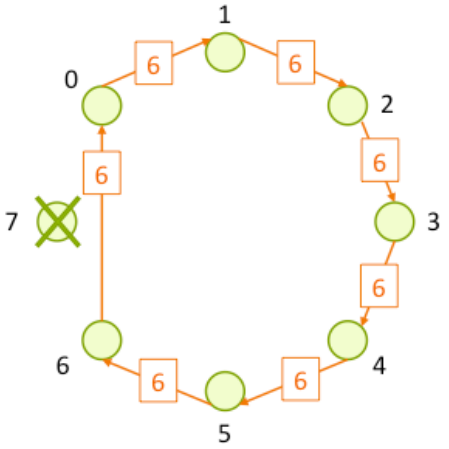
   then it resends the message to the following node of the ring



Leader election:
b) **Algorithm for rings**

3. When a node receives an ELECTION message and he is the **initiator**:
   - He selects as leader the highest identifier of list of participant nodes
   - He builds a COORDINATOR message including there the identifier of the new leader
   - And he broadcasts the message through the ring





Content

- Global States and Time
  - Clocks, events and states
  - Clock synchronization algorithms: Cristian, Berkeley
  - Logical clocks and Vector clocks
  - Global states
- Examples of Algorithmic Problems in Distributed Systems
  - Mutual exclusion
  - Leader election
  - Consensus
- Algorithms in the presence of failures

Consensus

- **Consensus**: we define the problem as the **agreement** that the participating nodes must reach in the value of a **variable**.
- **Particular cases of consensus:**
  - Leader election→ agree on the value of the variable "*leader node*"
  - Group membership→ agree on the value of the variable "*list of active nodes*".
  - Distributed commitment→ agree on the value of the boolean variable "*commit/rollback*"
  - Orderly dissemination of messages→ agree on the value of the variable(s) "sequence number of the message".
  - etc….

Consensus Algorithms

- There are currently very relevant and well-known implementations of consensus applied to replication and leader election in cloud systems
- **Examples**
  - **Paxos** [Lamport, 1998] [Lamport, 2001]
    - Dominant during last decades
    - Used in Amazon…
  - **Raft** [Ongaro 2014] → used in Facebook
    - Developed to be easy to understand.
    - Equivalent to Paxos in fault tolerance and performance.
    - List of more than 50 open source Raft implementations

    (https://raft.github.io/)

Consensus

Consensus Problem Definition:

- N nodes try to agree on the value of certain **Variable V.**
- All nodes receive the command **START** at approximately the same time.
  - In other words, we start from a situation in which we assume that all nodes want to participate in the consensus when starting to execute their algorithms.
- Each node "i" has an initial estimation of the variable: **estimate (Vi)**
- After executing the consensus algorithm, all must provide **decision (Vj)** as output, with "j" being the node that proposed the estimate value.

- It is not necessary to know "j", but it is necessary that "j" exists
  - The agreement must have arisen by adopting the estimate of some node
  - It is not a consensus to decide a value "derived" or generated from the estimates → in these cases they are problems different from consensus.

Consensus

- **Correction conditions for consensus** → a correct solution must fulfill these four properties:
  - (liveness) **Termination:** every correct node eventually decides some value.
  - (security) **Uniform integrity:** every node decides at most once.
  - (security) **Agreement:** no two correct nodes decide differently.
  - (security) **Uniform validity:** if a node decides **v**, then **v** was proposed by some node.

Consensus

- **Distributed algorithm (simplified)**
  - We assume N nodes in a totally connected network.
  - Of the N nodes, some nodes might be "turned off", because they have previously failed, but it is unknown which ones.
  - We assume that during the execution of the algorithm, no node fails.
  - **Algorithm example:**
    - All nodes broadcast their estimate(V)
    - All nodes decide as decision(V) the estimate value proposed by the node with lowest identifier.

Content

- Global States and Time
  - Clocks, events and states
  - Clock synchronization algorithms: Cristian, Berkeley
  - Logical clocks and Vector clocks
  - Global states
- Examples of Algorithmic Problems in Distributed Systems
  - Mutual exclusion
  - Leader election
  - Consensus
- Algorithms in the presence of failures

Algorithms in the presence of failures

- It is the greatest source of complexity in the design of algorithms.
  - In most cases, we have assumed that there are no failures (Chandy-Lamport, Berkely, Cristian, Mutual Exclusion, etc.) ... or that there are nodes that have previously failed (leader election, consensus)
- It is an unrealistic assumption.
  - The nodes fail and may fail during the execution of the algorithms.
- Nodes can fail in various ways.
  - The simplest type of failure, and still quite realistic is the **stop failure** (or crash failure)
    - The nodes fail at a certain moment and when they fail, they no longer run.
  - We do not consider "arbitrary" failures (byzantine), where nodes can behave erroneously and unpredictably → this leads to another range of algorithms

Distributed Consensus Algorithms considering failures

- We assume N nodes in a totally connected network.
- Of the N nodes, some nodes may be "turned off", because they have previously failed, but it is unknown which ones.
- We assume that **during the execution of the algorithm**, the **nodes can fail** with **stop failure.**
- We suppose that the nodes have a "failure detector" (or fault detector), based on "timeouts", to recognize when a node has failed
  - Note that a very tight *timeout* can make (wrongly) believe a certain node that another node has failed.
- To solve most problems in distributed algorithms in the presence of failures, we assume that timers are "fine" adjusted
  - Sooner or later, all the "right" nodes will see that the other correct nodes are correct
  - That is, sooner or later the timers are well adjusted.
  - We call these well-adjusted timers as "**eventually perfect failure detectors**".

- **Basic algorithm** (scheme used in Paxos)
  - The nodes are numbered and ordered, from 0 to N-1.
  - The nodes may be running or being "off", or fail during the algorithm.
  - We can tolerate $\lfloor (N-1)/2 \rfloor$ failures
    - That is, the "ground" of (N-1)/2 …. so with 3 nodes we tolerate 1 failure; with 4 nodes, 1 failure; with 5 nodes, 2 failures….
  - Therefore, it **only works** if there is at least $\lceil (N+1)/2 \rceil$ **correct nodes**, where $\lceil X \rceil$ is the ceiling of the real number x.
    - Example: with 5 nodes, the algorithm works if there are at least 3 correct nodes.

- **Basic algorithm** (scheme used in Paxos)
  - Rounds are executed until there is a round where at least $\lceil (N+1)/2 \rceil$ nodes successfully participate
    - In fact, the algorithm only supports that $\lfloor (N-1)/2 \rfloor$ nodes can really fail, but during certain rounds it is possible that the different failure detectors are not properly adjusted ... until eventually the failure detectors are properly adjusted.
  - If more nodes than $\lfloor (N-1)/2 \rfloor$ fail, the algorithm does NOT work, because the coordinating nodes get blocked.

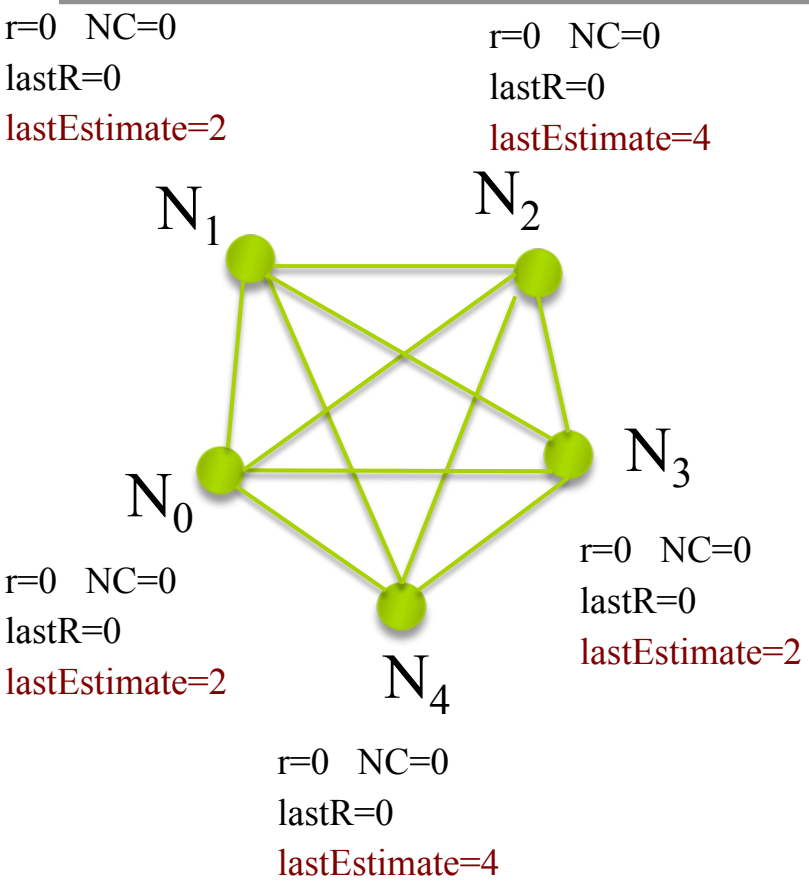
- **General Behaviour**
  - The nodes will execute rounds until they issue "decision(V)".
  - In each round they select as coordinator the next node
    - That is, round 0 coordinator 0, round 1 coordinator 1 ... round 2 coordinator 2 ... in round K coordinator (K mod N)
  - In each round we distinguish what the **coordinator** of the round does and the other nodes (**ordinary nodes**).
  - All nodes maintain these variables:
    - current round (**r**)
    - current coordinator (**NC**)
    - value that the node proposes to the variable (**lastEstimate**)
    - most recent round that made the node change the proposal (**lastR**)

Distributed Consensus Algorithms considering failures

- **Start**
  - All nodes start **r=0, NC = 0, lastR = 0, lastEstimate=** (value proposed by each node)
- **In each round "r":**
  - **Phase 1:** All nodes send to the coordinator of the NC round: "**estimate (r, lastEstimate, lastR)**"
  - And ordinary nodes wait in response a "propose" message.
    - They wait for the response message a maximum time.
    - If the time is exceeded, we say that its "failure detector" believes that the coordinator has failed.
    - Notice again that perhaps it has not failed, but the *timeout* expired.

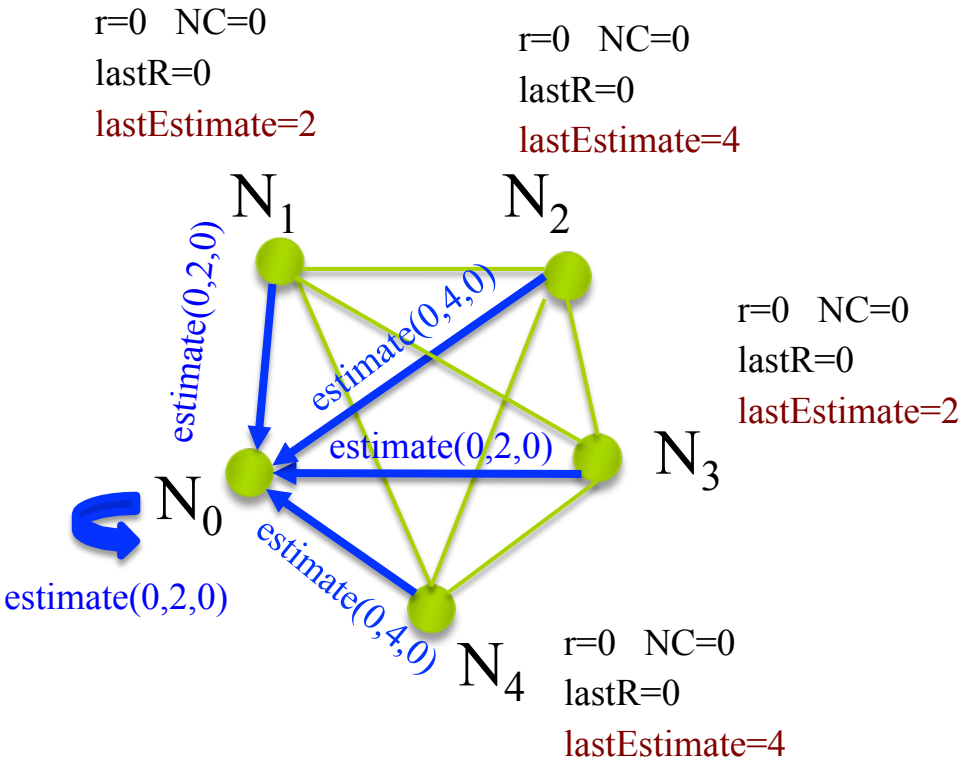
- **In each round "r":**
  - **Phase 2:** The coordinator waits to receive $\lceil (N+1)/2 \rceil$ "estimate" messages.
    - Note that timeouts are not used, nor failure detectors.
    - The coordinator knows that he will receive at least that number of messages, since we do not admit more number of possible failures.
    - Until it does not receive this quantity of messages, the coordinator is waiting
      - And the algorithm does not progress
    - Example: If there are 5 nodes, the coordinator waits to receive 3 estimate messages, including its own message.

Distributed Consensus Algorithms considering failures Example



START

Works if there are at least
|(5+1)/2|=3 correct nodes

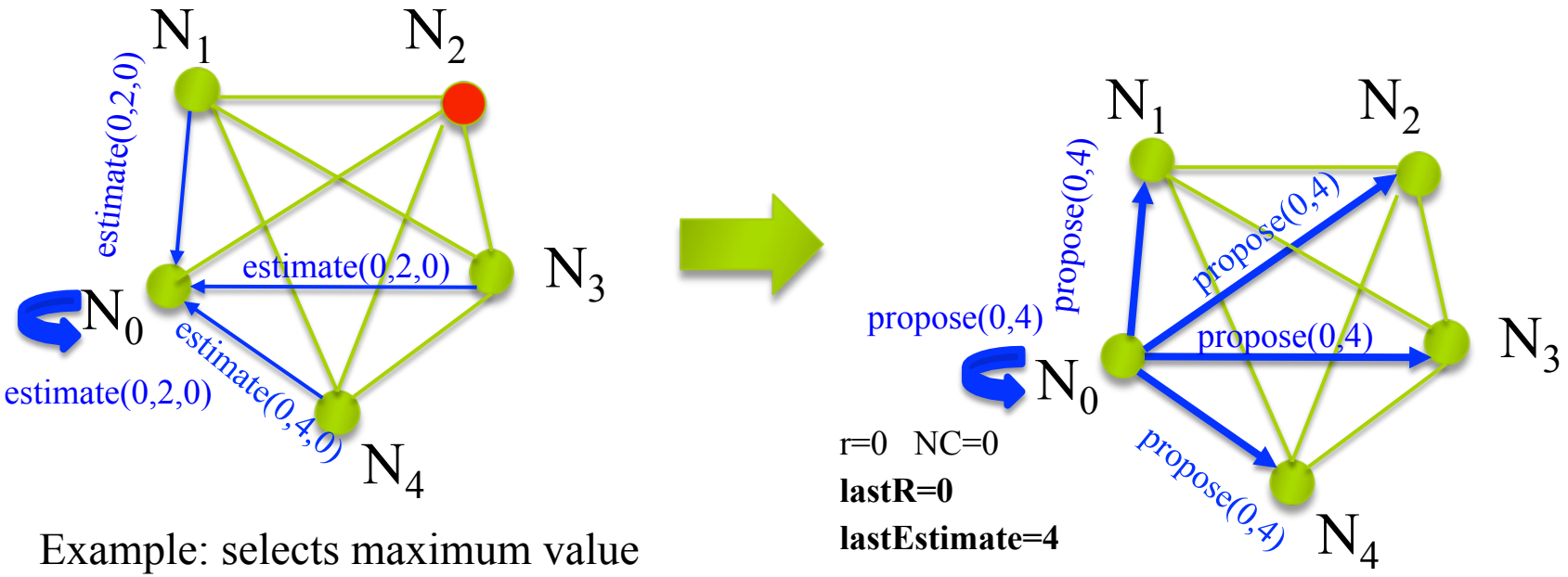


ROUND 0

All nodes send
estimate(r,lastEstimate,lastR) to NC

Distributed Consensus Algorithms considering failures

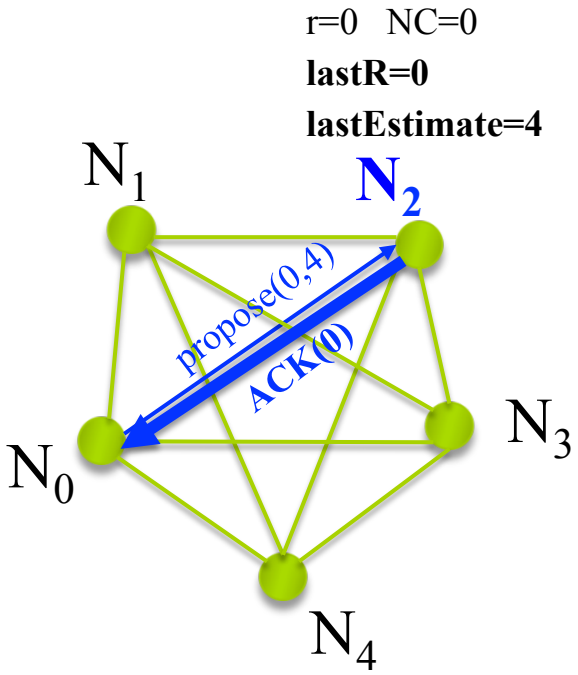
- The coordinator chooses one of the "lastEstimate" values that he receives, from among those with the highest "lastR" value, and this will be the proposal of the round
  1. Assigns its "**lastEstimate**" value to this chosen value
  2. Assigns **lastR = r** (its own round, not the "lastR received")
  3. The coordinator broadcasts "**propose (r, lastEstimate)**".



Example: selects maximum value received → Value 4

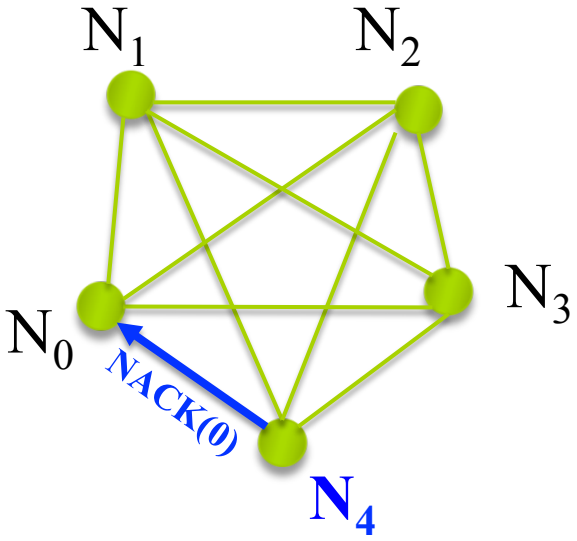
- **Phase 3**: Ordinary nodes wait to receive "propose (r, proposeR)" from the coordinator or their maximum waiting timeout expires.
- a) If they receive "propose" then:
  1. They answer with "**ACK (r)**" to NC.
  2. They update **lastEstimate = proposeR**,

  and **lastR = r.**



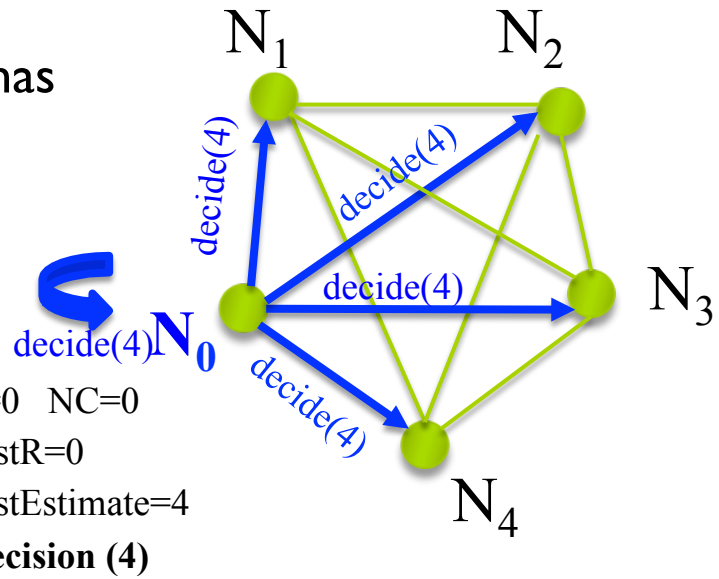
- With this the ordinary node knows that the coordinator proposed a value because he received the sufficient number of *estimates* ....
- The **node adopts it as its new estimate**, remembering the round in which it happened.

- Phase 3: Ordinary nodes wait to receive "propose (r, proposeR)" from the coordinator or their maximum waiting timeout expires.
- b) If its *timeout* expires when waiting for proposal, they answer with "**NACK (r)**" to NC.
  - Note that they send NACK to the coordinator although their "failure detector" believes that the coordinator has failed.
  - This guarantees that the coordinator will receive $\lceil (N+1)/2 \rceil$ answers of type ACK or NACK



- **Phase 4:** The coordinator waits **ACK** or **NACK** responses from the ordinary nodes.
  - Waits $\lceil (N+1)/2 \rceil$ messages, again without using timeouts, because he knows that he should received at least that amount of messages.
- If the coordinator receives $\lceil (N+1)/2 \rceil$ **ACK** messages:
  - It broadcasts "**decide(lastEstimate)**" and generates "**decision(lastEstimate)**"
  - This node already knows that consensus has been reached. → this is the final value.

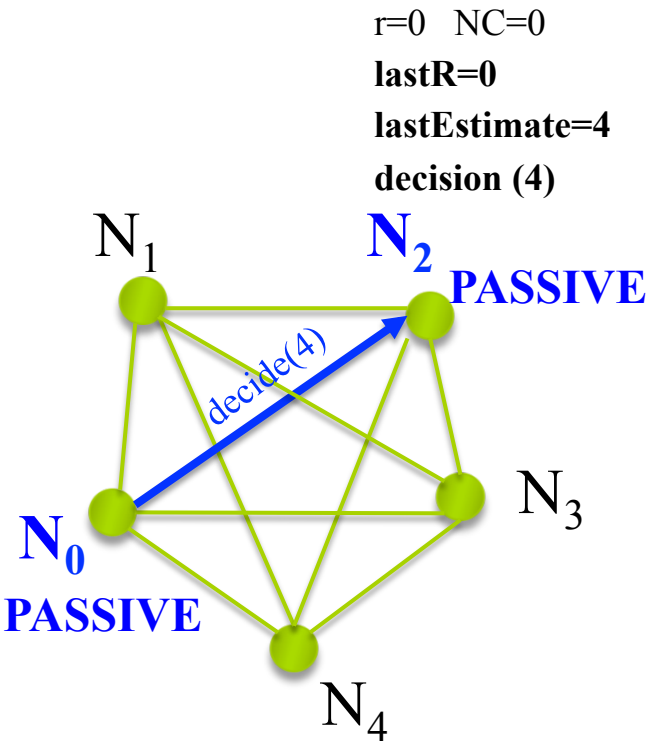
- **Example: NC has received 3 ACK (including its own message)**
- **Needs to receive: |(5+1)/2|=3 ACK messages**



Distributed Consensus Algorithms considering failures

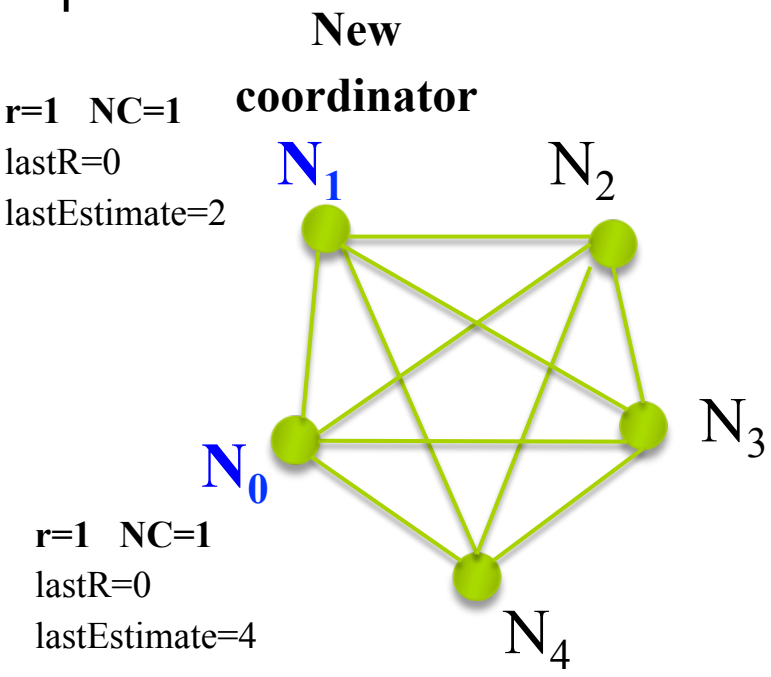
- If an ordinary node receives "**decide(lastEstimate)**", it generates "**decision(lastEstimate)**" → this node now knows that consensus has been reached.

- All the nodes that have generated "decision" continue to participate in the algorithm as **PASSIVE** nodes.
  - When it receives a propose(r,proposeR) message, it answers **ACK** iff **proposeR==lastEstimate** and **r>=lastR**.
  - i.e. whenever the proposeR matches its decision and the propose round is equal or higher than the round when it decided.
  - In this way they help other correct nodes to decide on their future rounds.

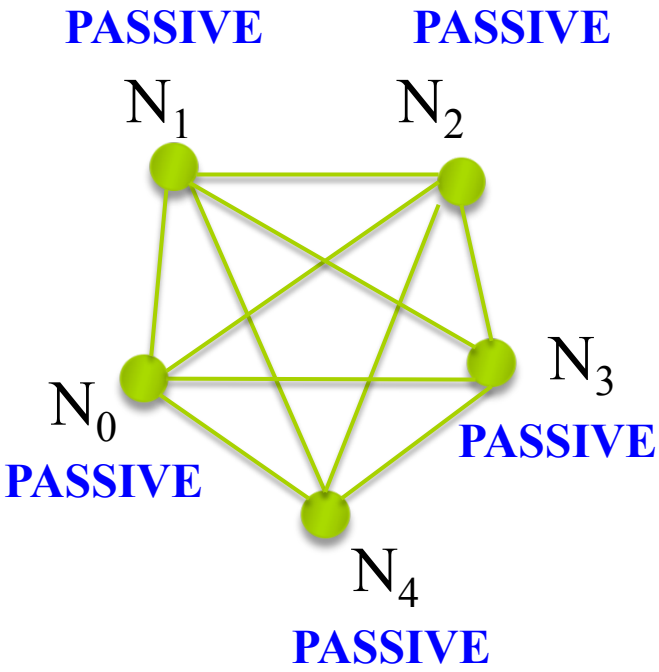


Round changes

- **Coordinator:** after Phase 4, if the coordinator does not receive a sufficient number of **ACK**, then he increases the round number and will no longer be a coordinator.
- **Ordinary node:** after Phase 3, the ordinary node increases round and becomes the new coordinator if it corresponds.



- **Ending**
  - The algorithm ends when all nodes are in **PASSIVE** mode.
    - Note that it is not known when the algorithm ends, but sooner or later every correct node will have generated the same decision.
    - And sooner or later every correct node will be in **PASSIVE** mode, without generating new messages or executing more code.
    - Sooner or later, (eventually) all nodes generate "decision" on the same value, as soon as the failure detectors of $\lceil (N+1)/2 \rceil$ nodes observe that the coordinator of that round is operative.



- **Conclusion**
  - To reach consensus in the presence of failures, we need to have well-adjusted timeouts, or at least, eventually well-adjusted timeouts.
  - **Consensus**→ leader election, atomic commitment, message order, group membership, etc.
  - For the vast majority of realistic algorithms and realistic systems we need **failure detectors** that "sooner or later" are quite "good"
    - They detect as failures the nodes that have really failed and do not detect the correct nodes as failed.
  - The treatment of failures is complex and its complete treatment is out of the scope of this subject.

Learning results of this Teaching Unit

- At the end of this unit, the student should be able to:
  - Describe the problems involved in managing time in a distributed environment.
    - Identify the advantages introduced by a logical ordering of events in a distributed application.
- Illustrate the classic solutions for some synchronization problems in a distributed environment:
  - Image of the global state
  - Leader election
  - Mutual exclusion
- Describe the concept of consensus and illustrate solutions of consensus algorithms.
- Describe distributed consensus algorithms considering failures.

Bibliography

- Atomic clocks
  - **"El hombre que ajusta la hora en España"**. El Mundo, Martes 30 de Junio de 2015. Sección Ciencia, páginas 29 – 30.
- Cristian Algorithm:
  - Flaviu Cristian. **Probabilistic clock synchronization**. Distributed Computing, 3(3):146–158, 1989.
- Berkeley Algorithm:
  - Riccardo Gusella & Stefano Zatti. **The accuracy of the clock synchronization achieved by TEMPO in Berkeley** UNIX 4.3BSD. IEEE Trans. Software Eng., 15(7):847–853, 1989
- Lamport's logical clocks:
  - Leslie Lamport. **Time, clocks, and the ordering of events in a distributed system**. Commun. ACM, 21(7):558–565, 1978.
- Vector clocks:
  - D. Stott Parker Jr., G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser & C. Kline. **Detection of mutual inconsistency in distributed systems**. IEEE Trans. Software Eng., 9(3):240–247, 1983.

Bibliography

- Chandy & Lamport algorithm:
  - K. Mani Chandy & Leslie Lamport. **Distributed snapshots: Determining global states of distributed systems.** ACM Trans. Comput. Syst., 3(1):63–75, 1985.
- Bully algorithm:
  - *classical algorithm*: H. Garcia-Molina. **Elections in a distributed computing system**. IEEE Trans. Comput., 1982, C-13, 48-59.
  - *Improvements to classical algorithm:* G. Murshed, A. R. Allen. **Enhanced Bully Algorithm for Leader Node Election in Synchronous Distributed Systems**. Computers, Vol. 1, pp. 3-23, 2012.
- Ring algorithm for leader election:
  - E. Chang, R. Roberts. **An improved algorithm for decentralized extrema-finding in circular configurations of processes**. Communications of the ACM, 22(5): 281-283, ACM, 1979.
- Distributed algorithm for mutual exclusion:
  - Glenn Ricart, Ashok K. Agrawala. **An optimal algorithm for mutual exclusion in computer networks**. Commun. ACM, 24(1):9–17, 1981.

Bibliography

- Consensus algorithms for distributed systems:
  - M. Bakhoff**. Consensus algorithms for distributed systems**.
  - D. Ongaro, J. Ousterhout. **In Search of an Understandable Consensus Algorithm**. Proc. USENIX ATC'14, Philadelphia, 2014.
  - L. Lamport, R. Shostak, M. Pease. **The Byzantine Generals Problem**. ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, pp. 382-401, 1982.
  - T. D. Chandra, S. Toueg. **Unreliable Failure Detectors for Reliable Distributed Systems**. Journal of the ACM, Vol. 43, no. 2, pp. 225-267, 1996.