Lecture 7
Coordination and distributed synchronization

Distributed Systems
Bachelor In Informatics Engineering
Universidad Carlos III de Madrid

Content

- Synchronization in distributed systems
- Physical and logical clocks
- Distributed mutual exclusion
- Election algorithms
- *Multicast* communication

Synchronization in distributed systems

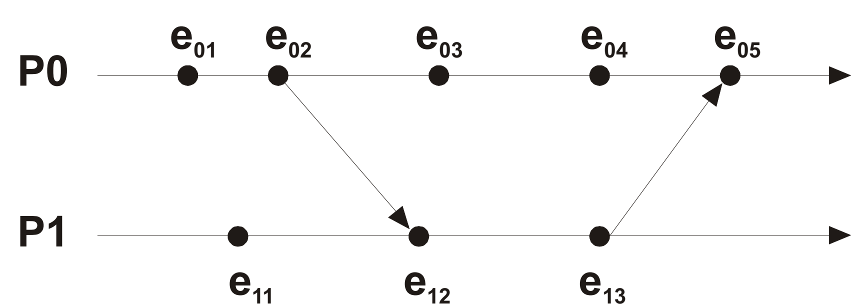
- More complex than in local one, distributed systems uses distributed algorithms.
- Distributed algorithms must have the following properties:
  - Relevant information is distributed among several machines
  - The processes make decisions based only on local information
  - Avoid single point of failure
  - Lack of a centralized clock

Time and distribution

- Difficulties in the design of distributed applications
  - Parallelism between nodes
  - Arbitrary speeds of processors
  - No determinism in the delay of messages. Fails
  - Lack of a global timer

System modelling

- Sequential processes $\{P_1, P_2, \ldots P_n\}$ and communication channels
- Events in P_i
  - $E_i = \{e_{i1}, e_{i2}, \ldots e_{in}\}$
  - $History(P_i) = h_i = \langle e_{i0}, e_{i1}, e_{i2}, \ldots \rangle \quad e_{ik} \rightarrow e_{i(k+1)}$
- Event types
  - Internal (changes in the state of a process)
  - Communication
    - Send
    - Receive
- Timeline diagrams

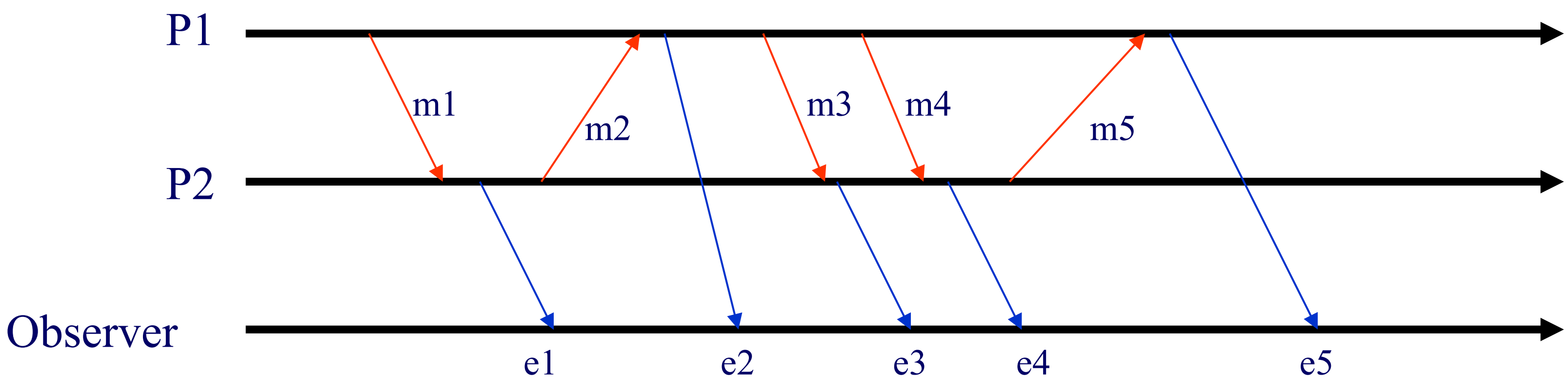


Synchronous and asynchronous models

- **Asynchronous systems**
  - **Lack of a shared clock**
  - Make no assumption on the relative rates of processes
  - The channels are reliable but there is no limit on message delivery
  - Communication between processes is the only solution for synchronization
- **Synchronous systems**
  - There is a perfect synchronization
  - There are limits on communication latencies
  - The real-world systems are not synchronous

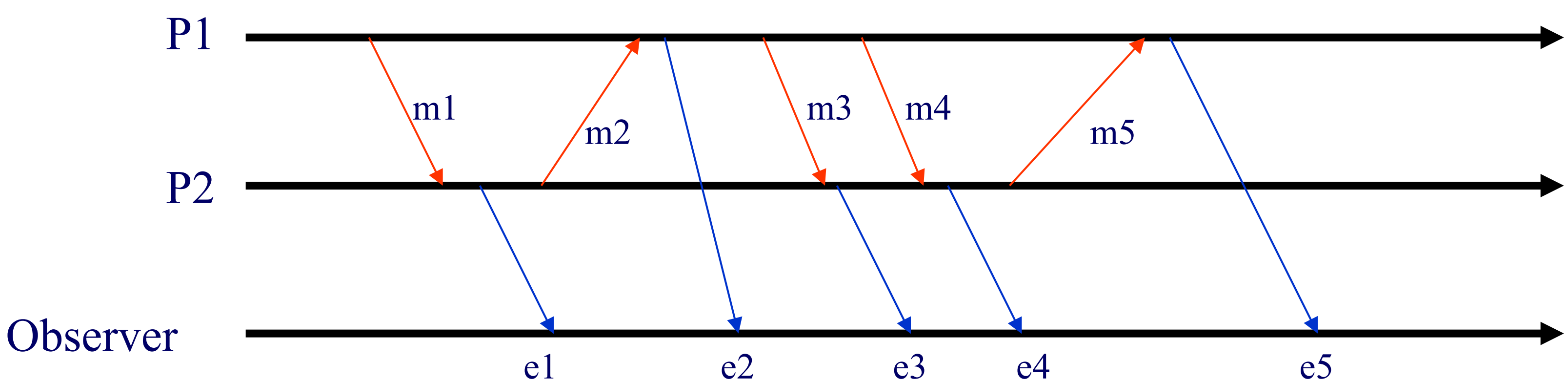
Example

- Monitoring the behavior of a distributed application
  - The observer must order the receiving event messages in processes P1 and P2
    - e1, e2, e3, e4, e5



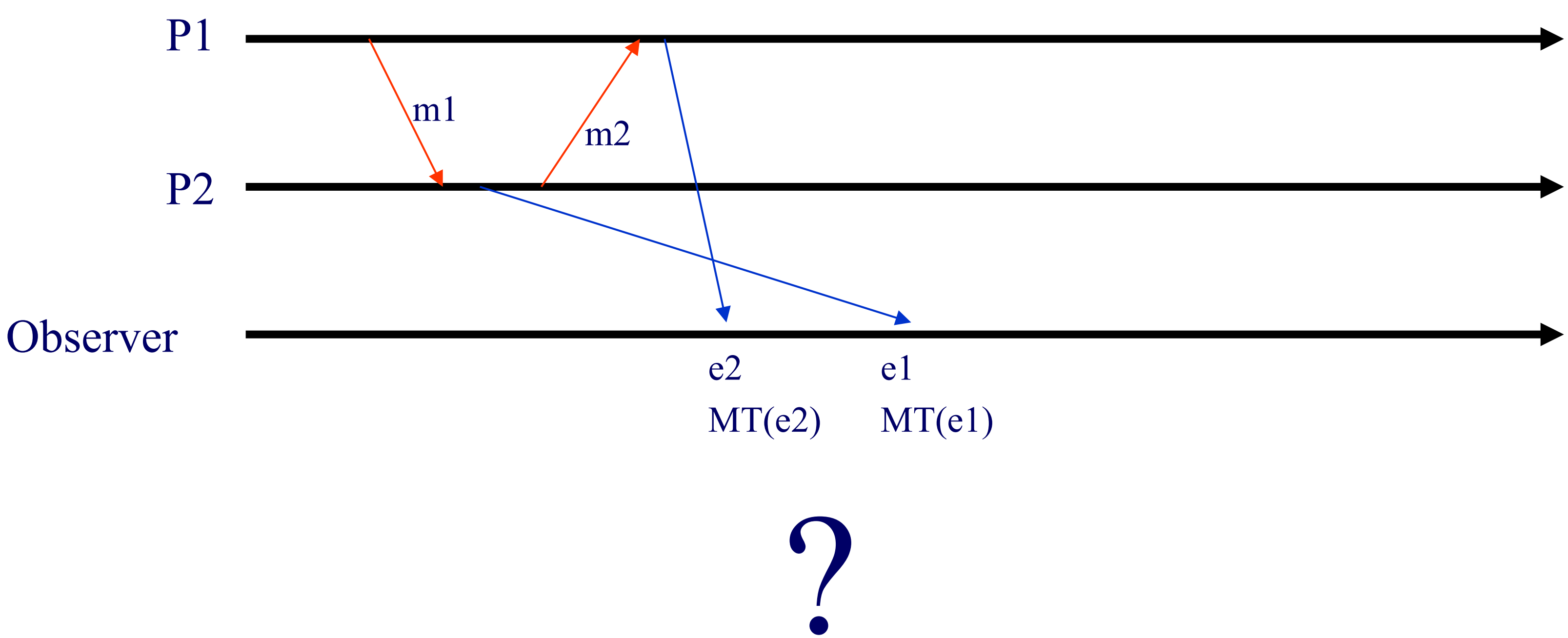
Example

- Monitoring the behavior of a distributed application
  - The observer must order the receiving event messages in processes P1 and P2
    - e1, e2, e3, e4, e5

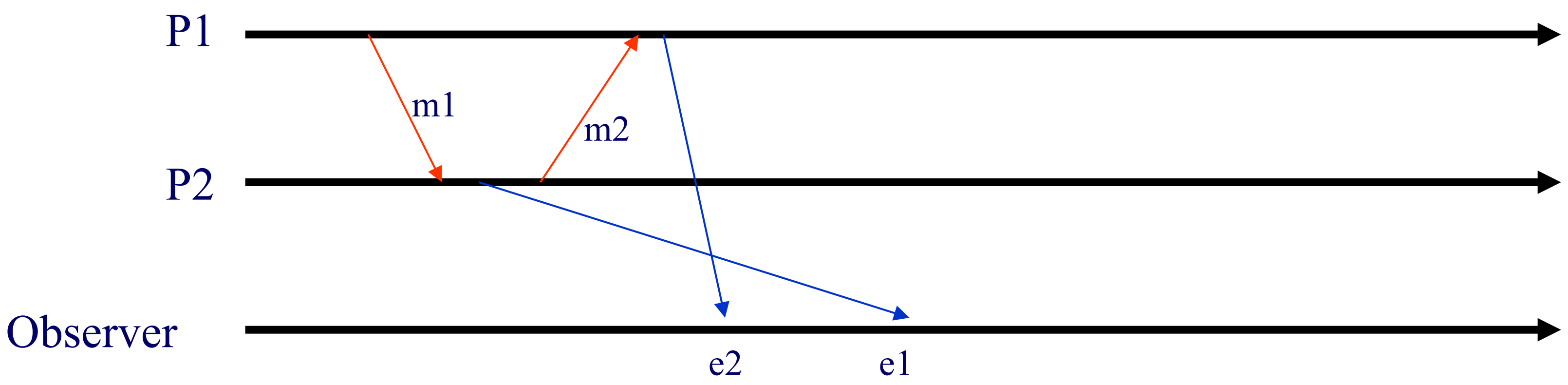


- To sort events we can assign timestamps
  - ei → ek ⇔ MT(ei) < MT (ek)

Timestamps in the observer?

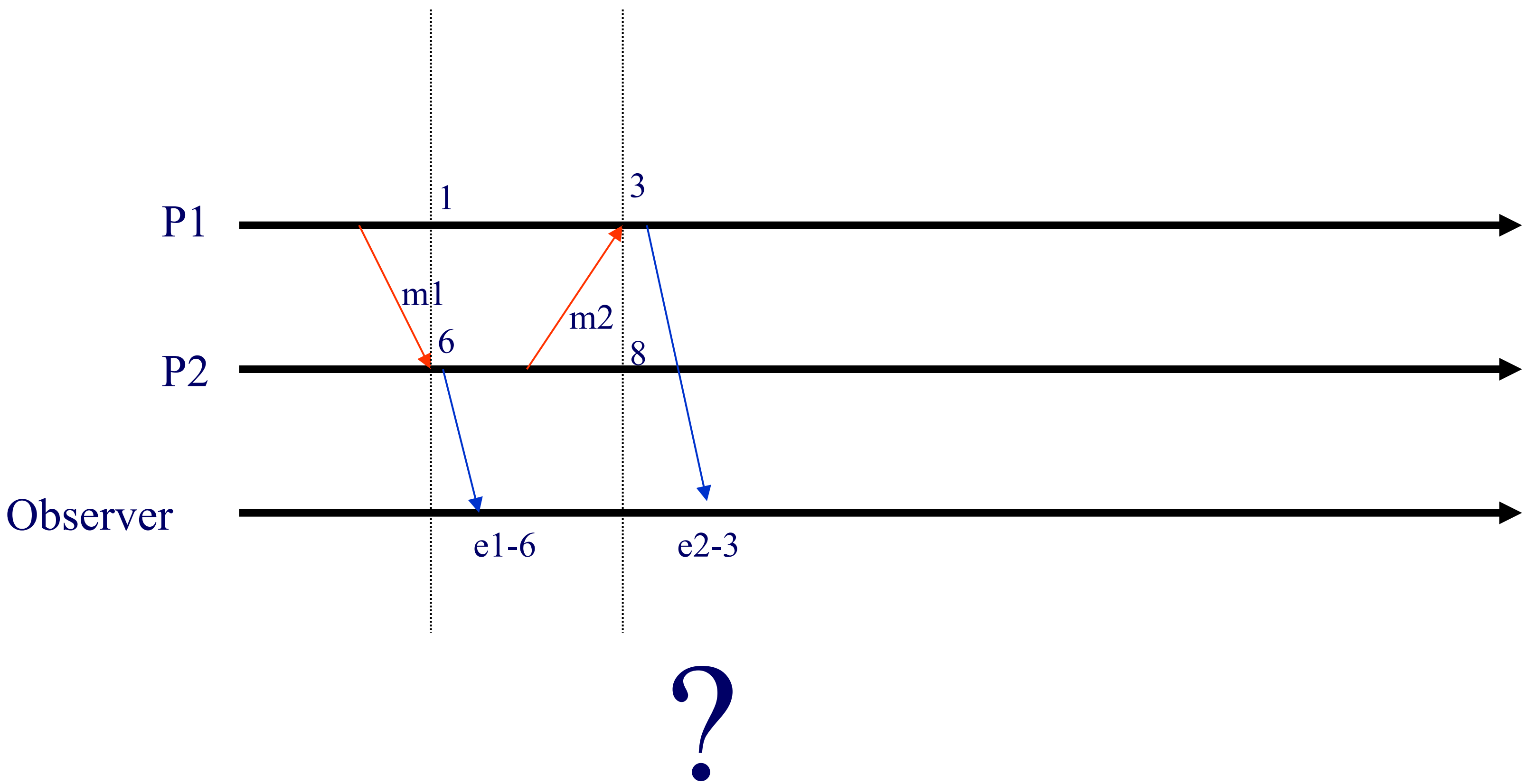


Timestamps in the observer?

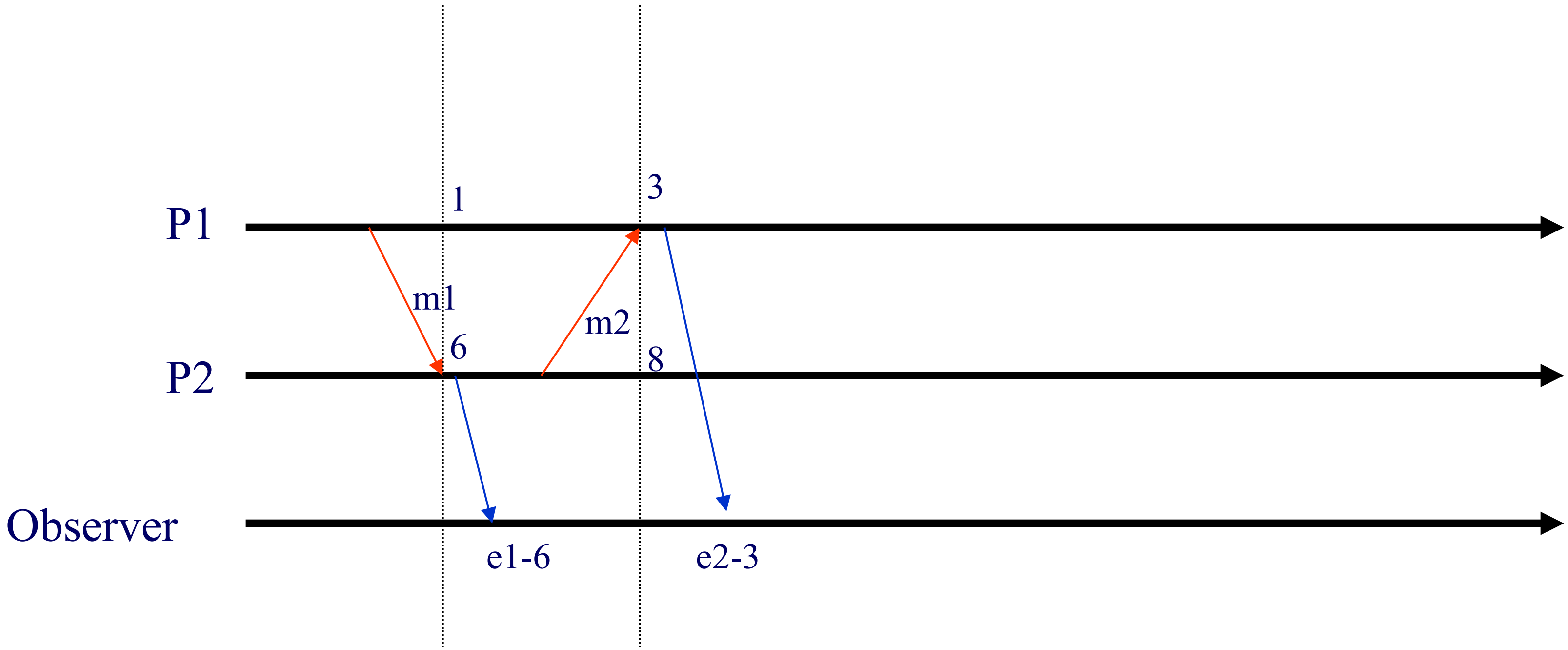


MT(e2) < MT (e1)   but   e1 → e2

Timestamps in the observer?

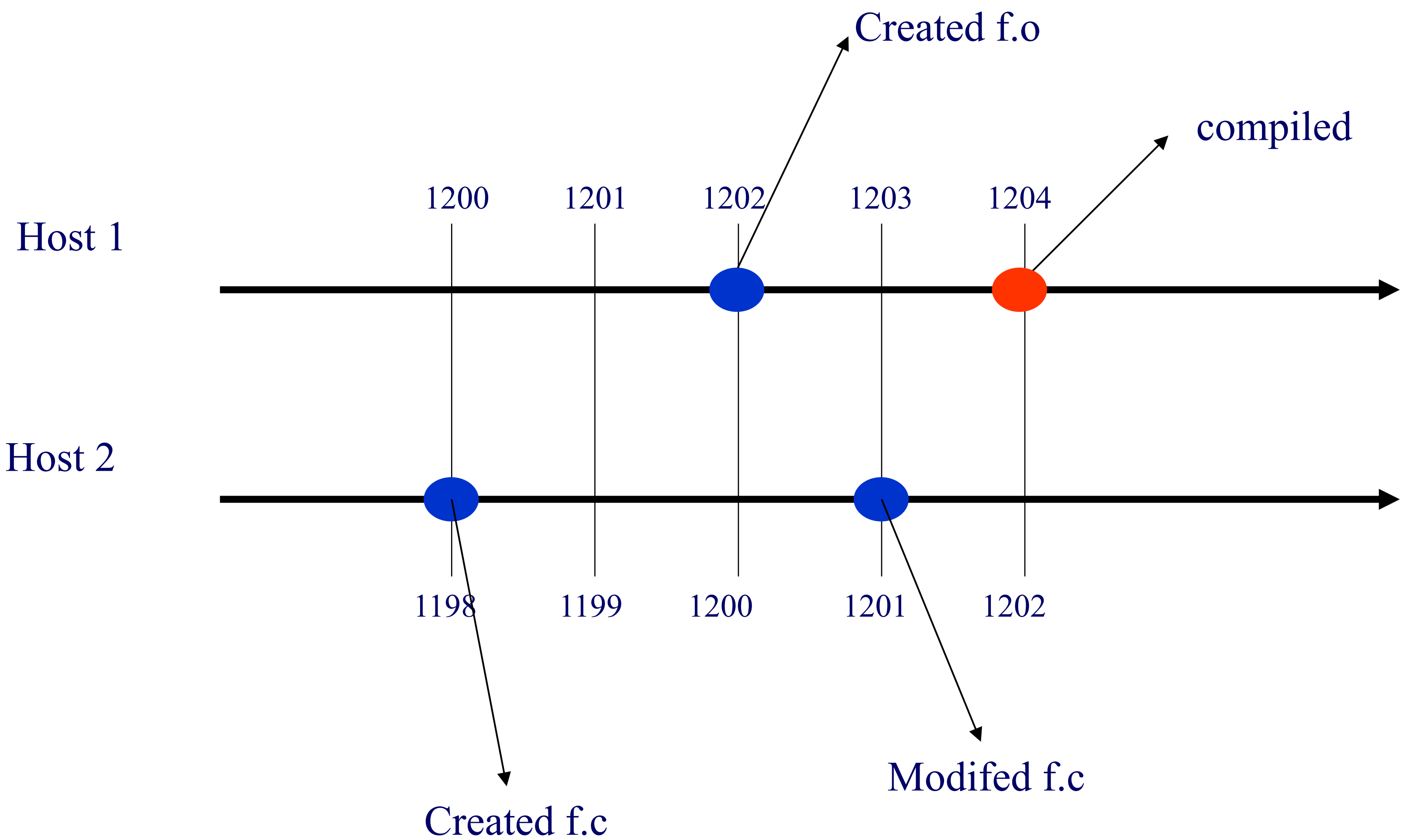


Timestamps in the observer?



Clocks must be synchronized

Example 2: make



Timestamps

- Physical clocks
- Logical clocks

Physical clocks

- To order two events of a process simply assign a **timestamp**
- Given physical instant **t**
  - $H_i(t)$: value of the HW based clock
  - $C_i(t)$: value of the SW based clock (generated by the OS)
    - $C_i(t) = a\, H_i(t) + b$
      - Example: # of ns or ms elapsed since a reference date
    - Clock resolution : period between updates of $C_i(t)$
      - Determines the events order
- Two clocks on two different computers provide different measures
  - We need to synchronize the physical clocks in a distributed system

Example

- `int gettimeofday (struct timeval *tp,`
  `struct timezone *tzp)`
  - Returns the amoung of seconds and miliseconds from January 1, 1970

Synchronizing physical clocks

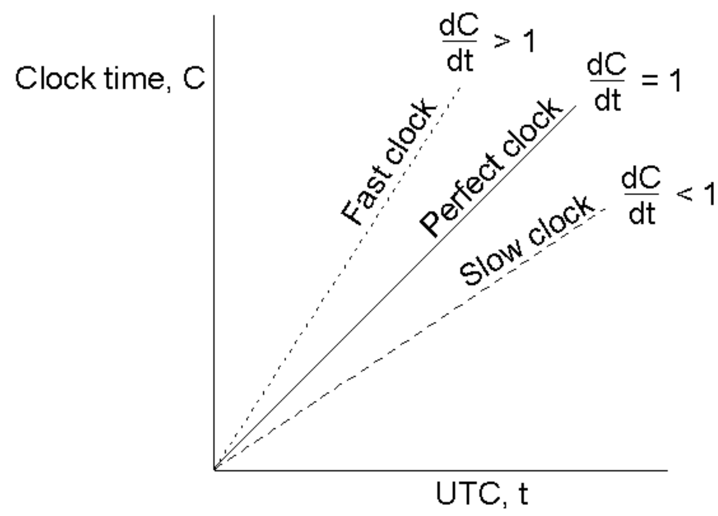
- Computers in a distributed system have clocks that are **not** synchronized
- Important to ensure proper synchronization
  - In real-time applications
  - Natural management of distributed events (dates files)
  - Performance analysis
- Traditionally used synchronization protocols that exchange messages
- Currently it can be improved by GPS
  - Computers counts with a GPS
  - One or two computers use a GPS and the rest are synchronized by standard protocols

Synchronizing physical clocks

- **D**: maximum deadline
- **S**: time source UTC, t
- **External synchronization**:
  - The clocks are synchronized if $|S(t) - C_i(t)| < D$
  - The clocks are considered synchronized within D
- **Internal synchronization** between the computers clocks of a distributed system
  - The clocks are synchronized if $|C_i(t) - C_j(t)| < D$
  - Given two events from two computers, it can establish an order according to their clocks if they are synchronized
- External synchronization $\not\Leftarrow$ $\Rightarrow$ Internal synchronization

Clocks synchronization methods

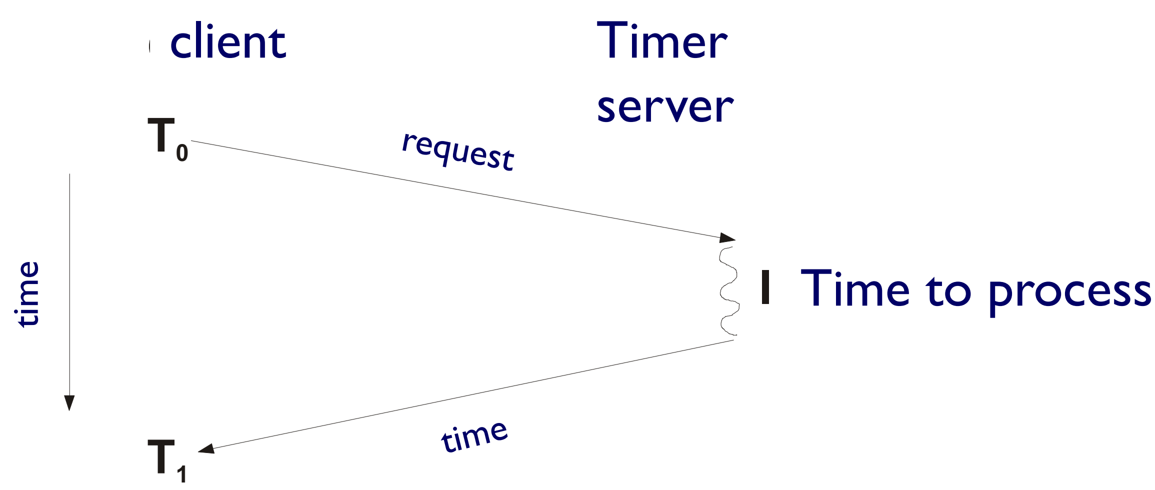
- Synchronization in a synchronous system
- Cristian algorithm
- Berkeley algorithm
- *Network time protocol (ntp)*



Synchronization in a synchronous system

- **P1** sends its local clock **t** to **P2**
  - P2 can update its local clock to $\mathbf{t + T_{transmit}}$ if $T_{transmit}$ is the time for transmiting the message
  - However, $T_{transmit}$ can vary or been unknown
    - Competing for the use of the network
    - Network congestion
- In a **synchronous system** the minimum and maximum transmission time of a message is known
- u = (max - min)
  - If P2 sets its clock to the value t + (max-min)/2, then the maximun drift is $\leq$ u/2
- The problem is that in an asynchronous system $T_{transmit}$ is not bounded

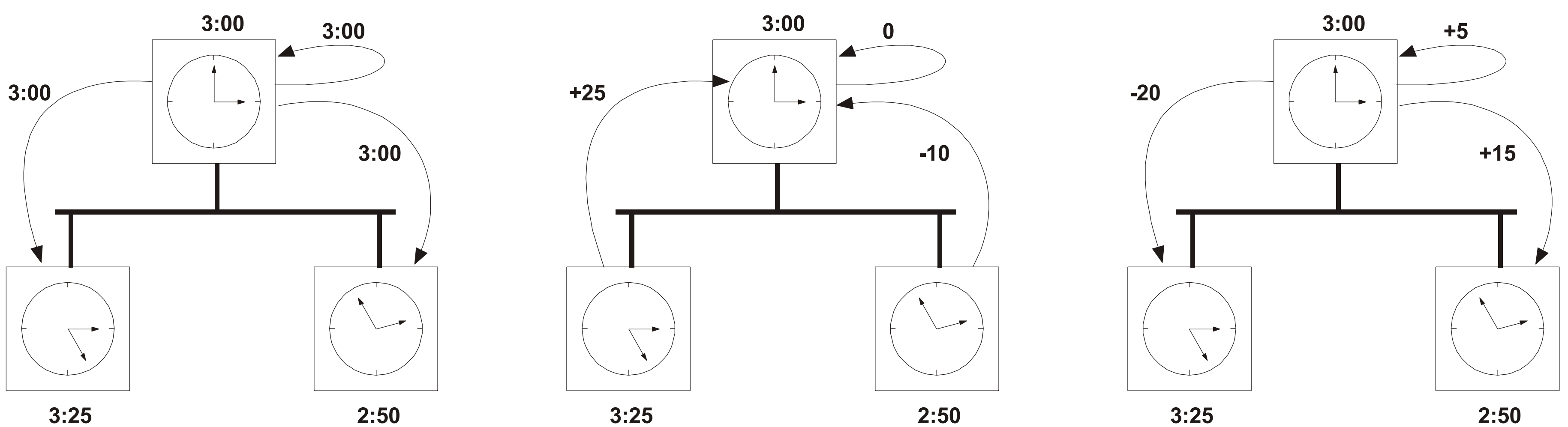
Cristian algorithm



- Message transmission time (T1 -T0) / 2
- Message difusion time: (T1 -T0 -I) / 2
- The value T returned by the server can be increased at (T1 - T0 -I) / 2. The value in the client will be t + (T1 -T0 -I) / 2
- To improve accuracy, we can make several measurements and discard any where T1 -T0 exceeds a limit

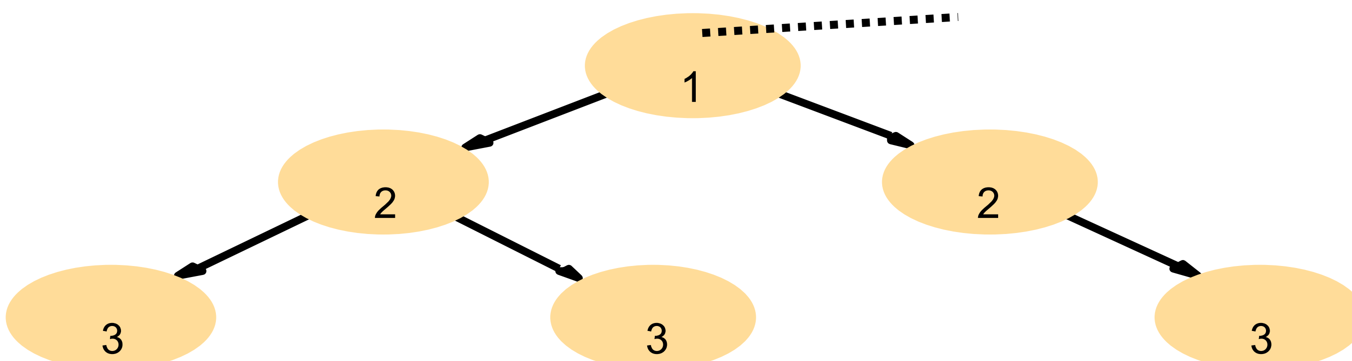
Berkeley algorithm

- The time server performs periodic sampling of all machines to ask for time
- Calculate the average time and tells all machines that update their clock to the new time or to decrease the refresh rate



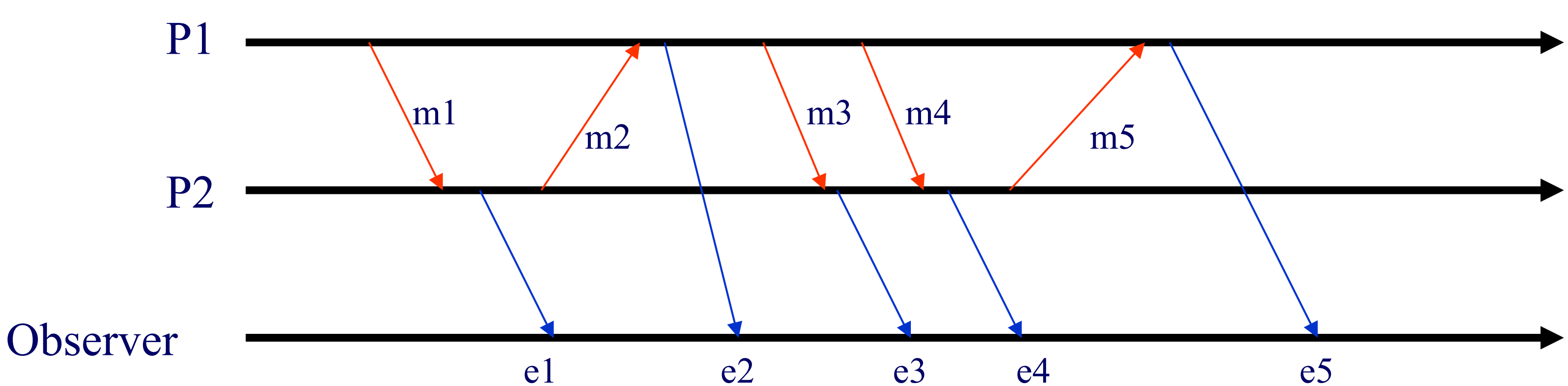
Network time protocol (NTP)

- Service to synchronize machines on the Internet to UTC
- Three types of synchronization
  - multicast: for high-speed LAN networks
  - RPC: similar to Cristian's algorithm
  - symetric: between peers
- Used servers located throughout the Internet with UDP messages



Logical clocks

- Since there can not perfectly synchronize physical clocks in a distributed system, you can not use physical clocks to sort events
- Can we order events in a different way?



Potential causality

- In the absence of a global clock, the cause/effect is the only possibility to sort events
- Potential causality (Lamport , 1978) is based on two observations:
  1. If two events occur in the same process (pi (i = 1..N)) then occurred in the same order they were observed
  2. If a process sends (m) and one receives (m), then send occurr before the event receive event
- Then, Lamport defines the potential causal relationship
  - **Before–than** (→) between any two events
    - Ej: a→b
- **Partial order**: reflexive , anti-symmetric, and transitive
  - Two events are concurrent (a || b) if it can not be deducted a list of potential causality

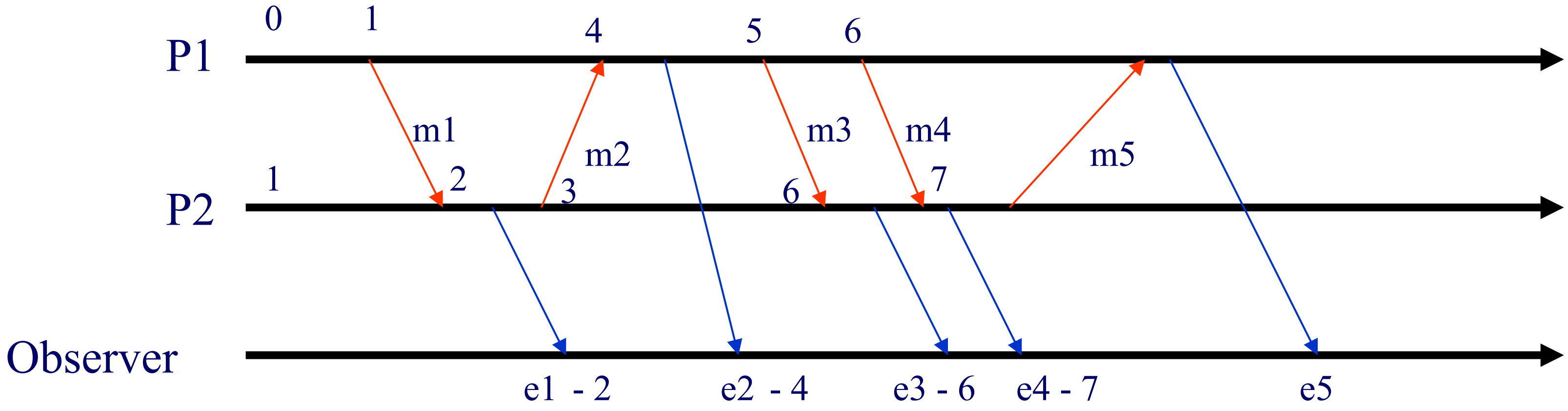
Importance of potential causality

- Synchronization of logical clocks
- Distributed debuging
- Logging of global states
- Monitorization
- Causal delivery
- Distributed replica management

Logical clocks (Lamport algorithm)

- Useful for sorting events in the absence of a common clock
- Lamport's algorithm (1978)
- Each process P maintains an integer variable RL_P (logical clock)
- When a process P generates an event, $\boldsymbol{RL_p=RL_p+1}$
- When a process sends a message m to another, it includes the value of the logic clock (private value)
- When a process Q receives a message m with a time value t, the process updates its clock, $\boldsymbol{RL_q=max(RL_q,t)+1}$
- The algorithm ensures that if **a → b then RL(a) < RL(b)**
  - The opposite can not be proved

Example



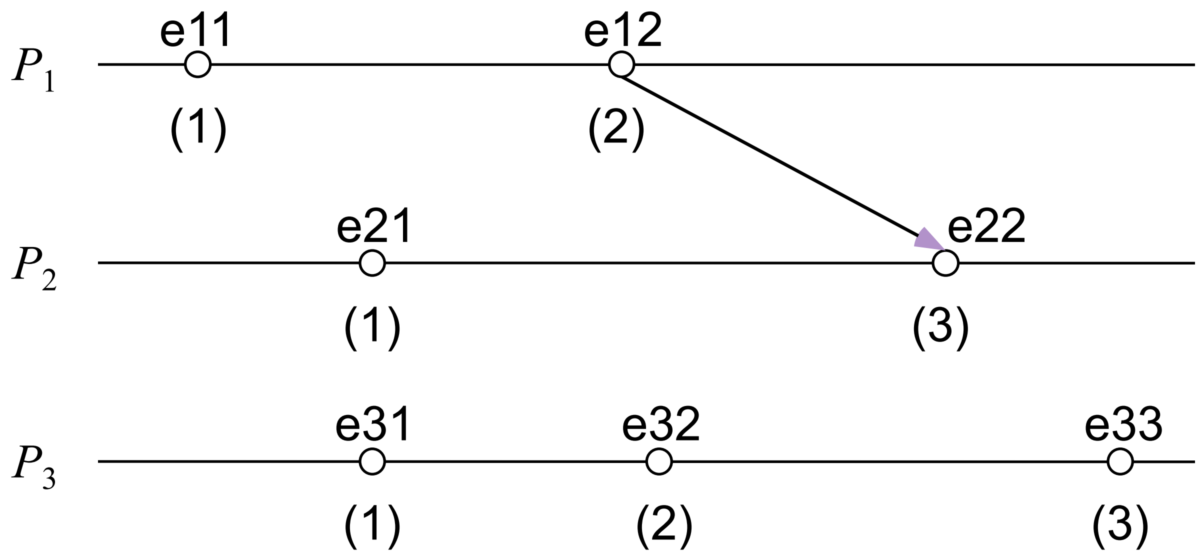
Totally ordered logical clocks

- Lamport 's logical clocks impose only a partial order relation:
  - Events of different processes can have associated the same timestamp
- You can extend the order relation to get a total order relation adding the **process id**
  - **(T_a, P_a):** timestamp of the event a in the process *P*
- $(T_a, P_a) < (T_b, P_b)$ if
  - $T_a < T_b$ **or**
  - $T_a = T_b$ y $P_a < P_b$

Problems in logical clocks

- Not sufficient to identify causality
  - Given RL (a) and RL (b) can not know:
    - if *a* occurs before *b*
    - if *b* occurs before *a*
    - if *a* and *b* are concurrents
- We need a relation (*F(e)*, <) that:
  - $a \rightarrow b$ only if $F(a) < F(b)$
  - **Vector clocks** can represent accurately the potential causal relationship

Problems in logical clocks



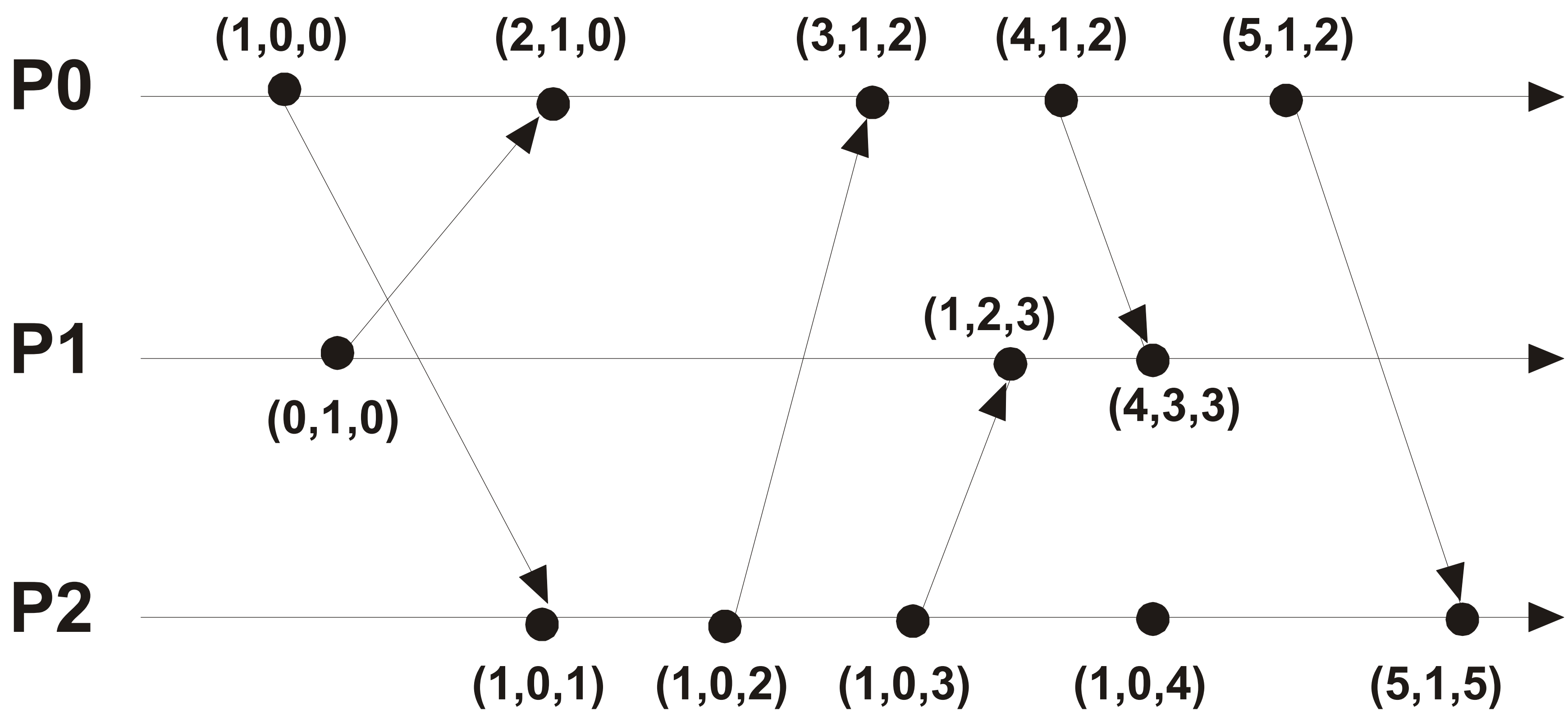
C(e11) < C(e22), and e11→e22 is true

C(e11) < C(e32), but e11→ e32 is false

Vectorial clocks

- Developed independently by **Fidge**, **Mattern** and **Schmuck**
- Every process has an associated vector of integers **RV**
- **$VC_i[a]$** is the value of the clock vector for the process *i* when you run the event *a*
- Maintenance of vector clocks
  - Initially $VC_i = 0 \ \forall i$
  - When a process generates an event *i*
    - $VC_i[i] = VC_i[i] + 1$
  - All messages transmit the VC
  - When a process *j* receives a message with VCi
    - $VC_j = max(VC_j, VCi)$ (element by element)
    - $VC_j[j] = VC_j[j] + 1$ (reception event)

Vectorial clocks



Properties of vectorial clocks

- $RV < RV´$ if
  - $RV \neq RV´$ and
  - $RV[i] \leq RV´[i], \forall i$
- Given two events a and b
  - $a \rightarrow b$ and $RV(a) < RV(b)$
  - a and b and concurrent when
    - $RV(a) \leq RV(b)$ neither $RV(b) \leq RV(a)$

Distributed coordination and quorum

- Distributed mutual exclusion
- Election algortihms
- Multicast communication

Distributed mutual exclusion

- The processes execute the following code

  in()

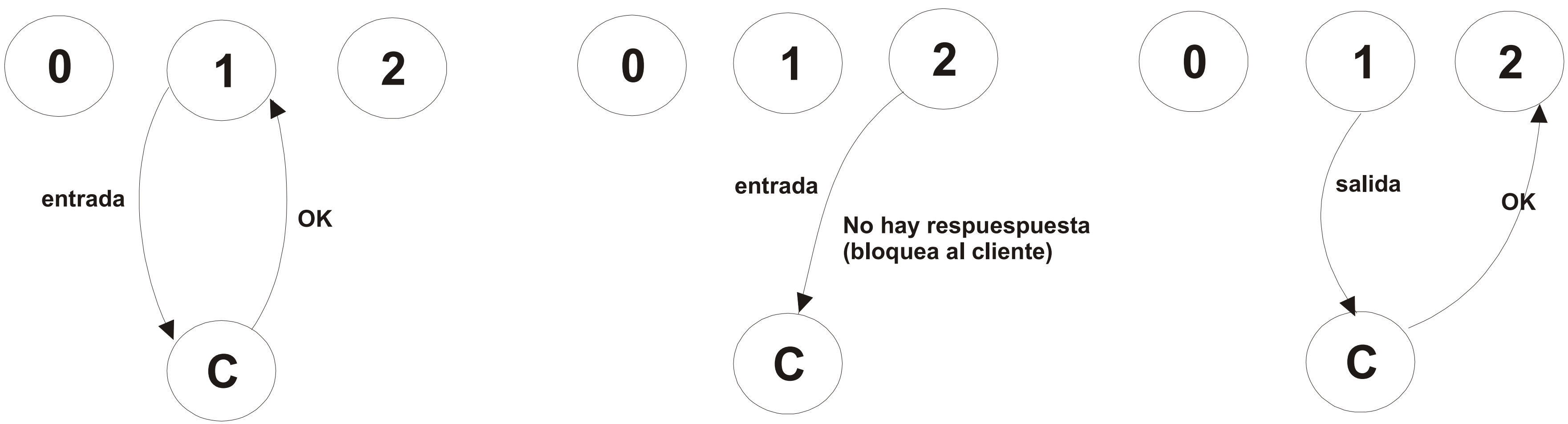
  CRITICAL SECTION

  out()

- **Requirements** for resolving a critical section
  - Mutual exclusion
  - Progress
  - Bounded waiting
- **Algorithms**
  - Centralized algorithm
  - Distributed algorithm
  - Token-ring

Centralized

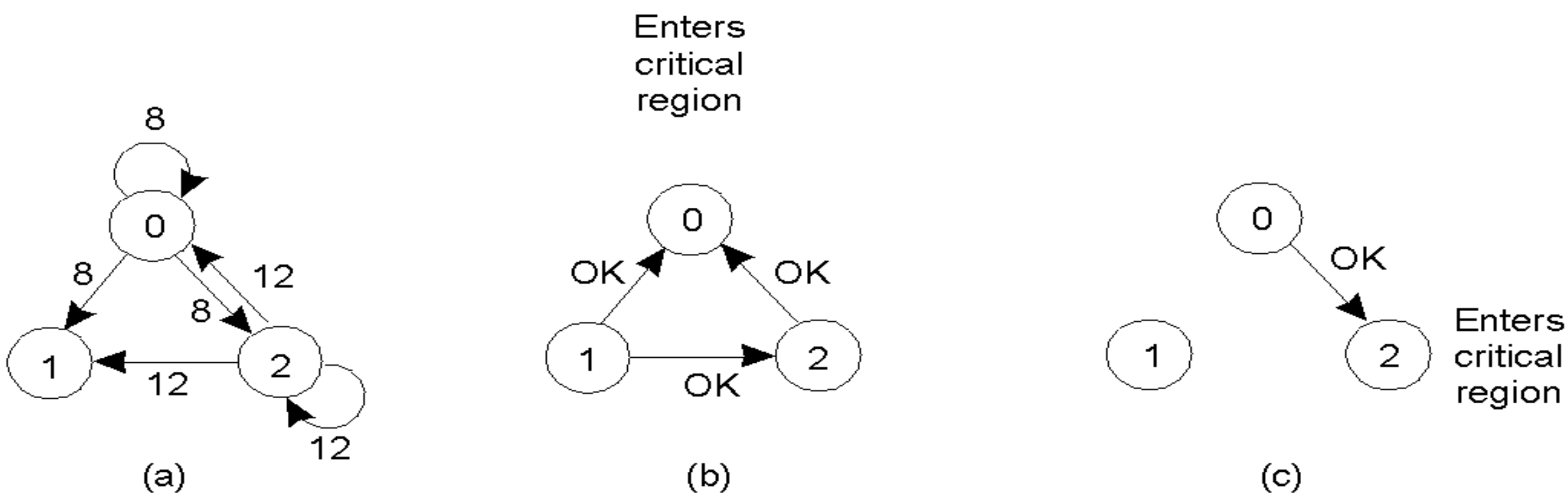
- Exists a centralized coordinator



@Source: *Jesús Carretero, Félix García, Pedro de Miguel y Fernando Pérez.* **Mc Graw Hill**

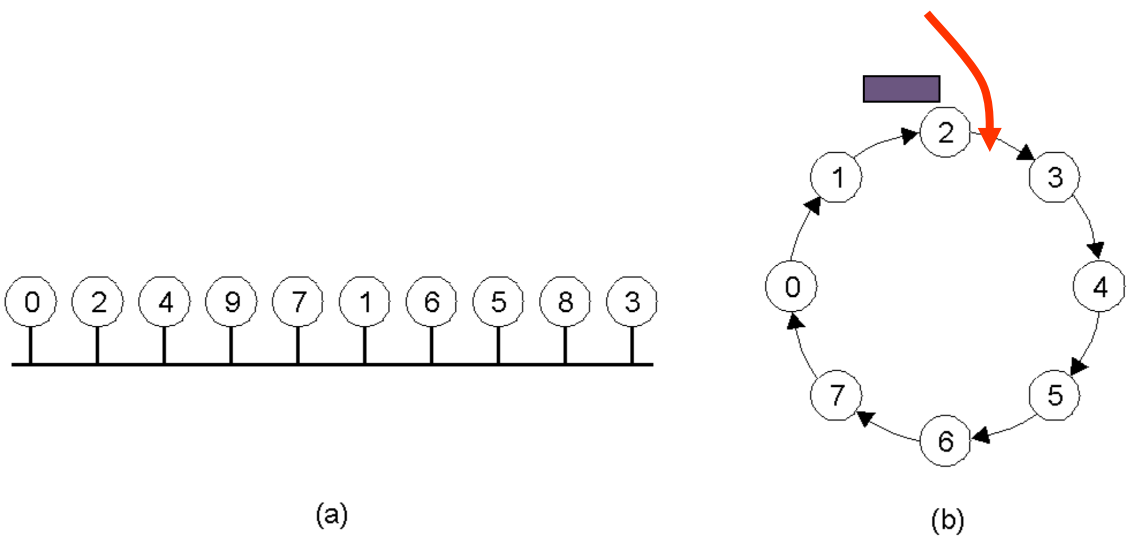
Example of distributed algorithm

a) Two processes (P0 , P2) want to enter the critical region at the same time.
b) The process 0 has the lowest time mark, so it enters.
c) When the process 0 ends, it sends an OK message, and in this way, process 2 enters in the critical section.



Token-ring

- Processes are conceptually arranged as a ring
- A **token** is transmitted in a circular way
- When a process wants to enter in the SC, it should wait in order to collect the token
- When the node leaves the SC, the token is sent to the next node



Algorithms comparison

- Centralized
  - Messages: 3
  - Lag: 2
  - Problems: fail in the coordinator
- Distributed
  - Message: 2(n-1)
  - Lag: 2(n-1)
  - Problems: fail in any process
- Token ring
  - Message: 1 to n-1
  - Lag: 1 a n-1
  - Problems: token lost, fail in any process

Election algorithms

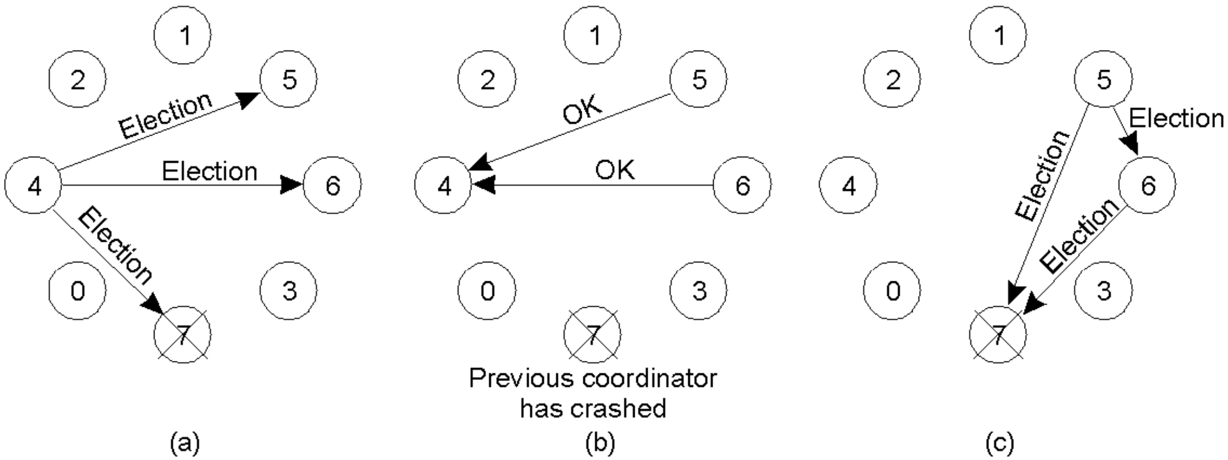
- Useful in applications where the existence of a coordinator is required
- The algorithm must be executed when **the coordinator fails**
- Election **algorithms**
  - Bully algorithm
  - Token-ring algorithm
- The objective of the algorithms is to obtain a unique although the algorithm starts concurrently in several processes

Bully algorithm. Example

- Uses **timeouts** (T) for detecting fails
- Assume that each process knows which processes have greater ID
- Three types of messages:
  - **coordinator**: message to all processes with lower IDs
  - **quorum**: sent to processes with greater IDs
  - **OK**: response to the election
    - If not received within T, the transmitter sends election coordinator message.
    - Otherwise, the process waits to receive a coordinator T message. If it does not arrive, the process starts a new election

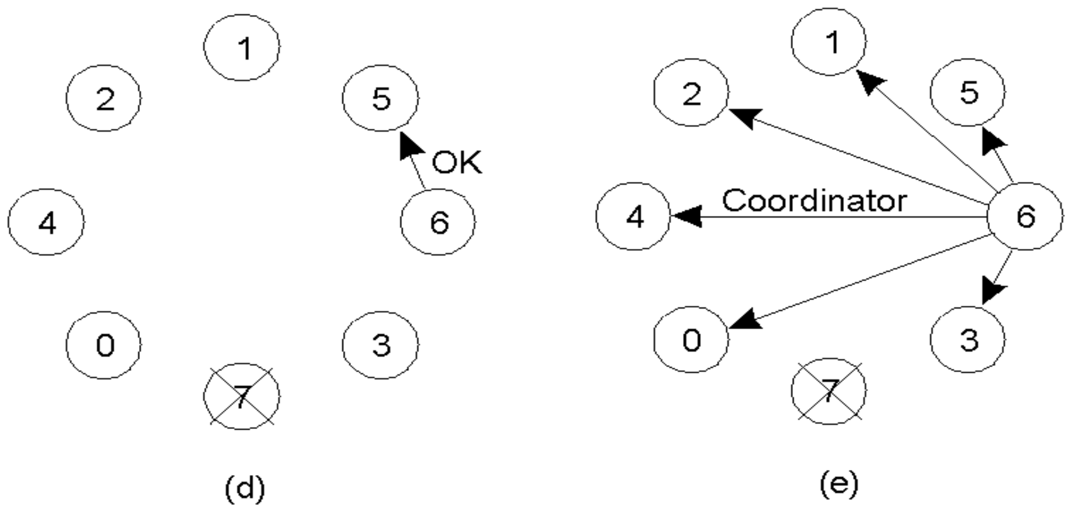
Bully algorithm. Example

- When a process P notes that the coordinator does not respond initiates an election:



a) Process 4 sends election
b) Process 5 and 6 respond , telling him to stop
c) Now 5 and 6 begin choosing…

Bully algorithm. Example



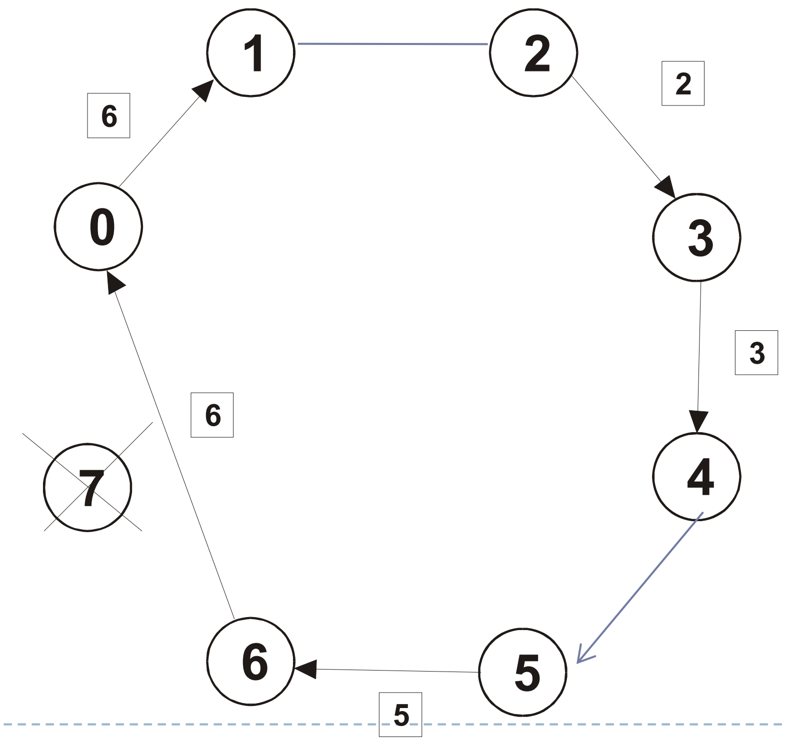
d) Process 6 indicates 5 to stop
e) Process 6 tells everyone that he is the coordinator

Token-ring algorithm

- Any process can begin the election and election sends a message to its neighbor with its identifier and marked as participant
- When a process receives an election message, it compares the message ID with yours:
  - If greater forwards the message to the next
  - If it is smaller and is not a substitute participant identifier his message and forwards.
  - If it is smaller and is a participant fails forwards
  - When a message is forwarded, the process is marked as participant
- When a process receives an identifier number and it is the largest, this is chosen as coordinator

Token-ring algorithm

- The 2 and 5 processes generate an election message and send it to the following process
- Coordinator is chosen as the process that receives a message with smaller value
- This process then sends messages to all reporting that is the coordinator



Distributed deadlock

- **Deadlocks** due to resource allocation. There deadlock when the following conditions are met
  - Mutual exclusion
  - Starvation and Livelock
  - No expulsion
  - Circular wait conditions
- **Deadlocks** due to misuse of synchronization operations
- **Deadlocks** due to communication channel
  - All processes are waiting for a message from another member of the group and there is no messages in the channel

Multicast communication

- **Broadcast**: the sender sends a message to all nodes in the system
- **Multicast**: the sender sends a message to a subset of all nodes
- These operations are normally implemented by point-to-point message passing calls

Usage

- Replicated **servers**:
  - A replicated service consists of a server set.
  - The client requests are sent to all group members . Although a group member fails the operation will be performed.

- Better performance:
  - Data replication.
  - When data is changed, the new value is sent to all processes that manage replicas.

Multicast types

- **Non-reliable multicast**: no guarantee that the message is delivered to all nodes.
- **Reliable multicast**: the message is received by all running nodes.
- **Atomic multicast**: protocol ensures that all group members receive messages from different nodes in the same order.
- **Causal multicast**: ensures that the message is delivered according to causal relationships.

Motivation for atomic multicast

- Given a bank with a replicated database.
- Consider a bank account with a balance of 1,000 euros.
  - A user enters 200 euros sending a multicast to both databases.
  - At the same time, a user pays 10 % interest by sending a multicast to both databases..
  - What if messages arrive in different order in the two databases?

Multicast implementation

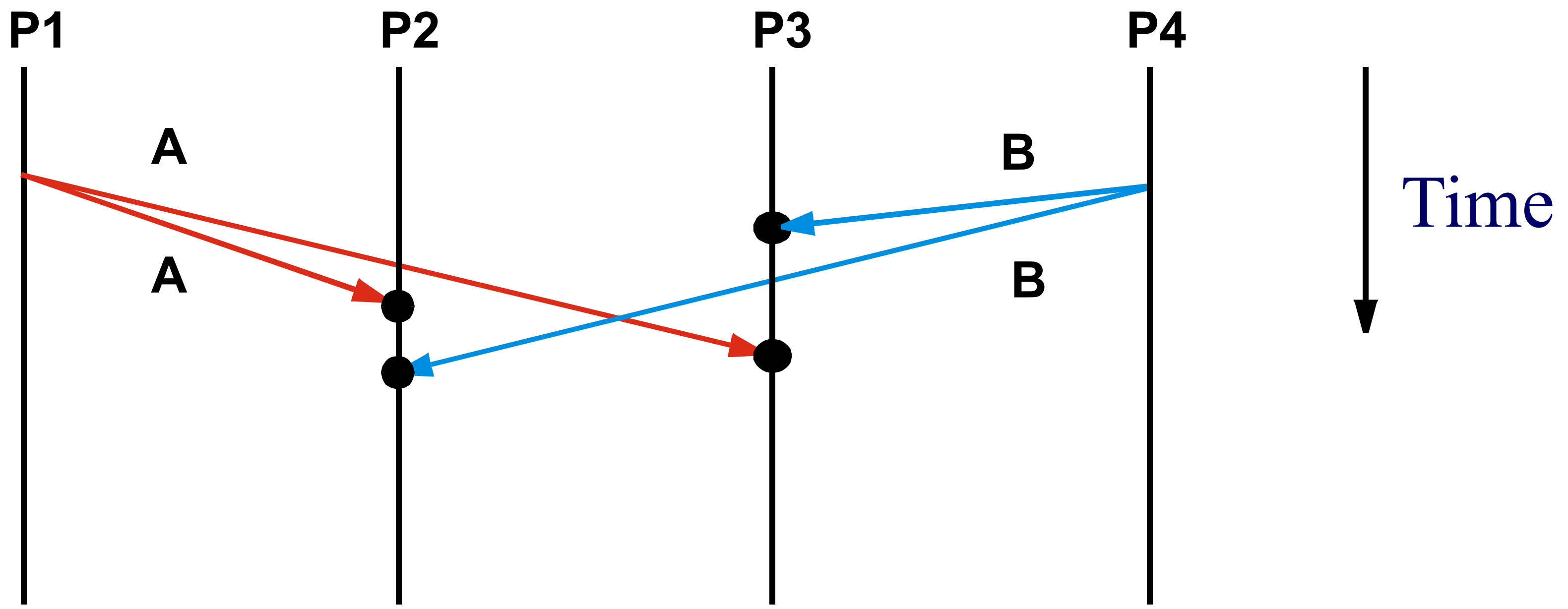
- Implementations of multicast operations:
  - By **point-to-point** communication
  - Unreliable mechanism

- Reliability issues :
  - Some of the messages may be lost
  - The sender process may fail. In this case, some processes will not receive the messages

Reliable multicast

- A message is sent to all processes and it is expected confirmation of all
  - If all confirm, the **multicast** is completed
  - If it is not confirmed, it is retransmitted. If we do not recieve confirmation, we can assume that the process has failed and it is removed from the group
- If the communication fails during operation, the multicast will not be atomic
  - For having an atomic operation, if the sender fails, any of the recipients must complete operation to all other
  - When a process receives a message, it sends an acknowledgment to the sender and monitors to see if it fails. In presence of failure, the process completes the multicast

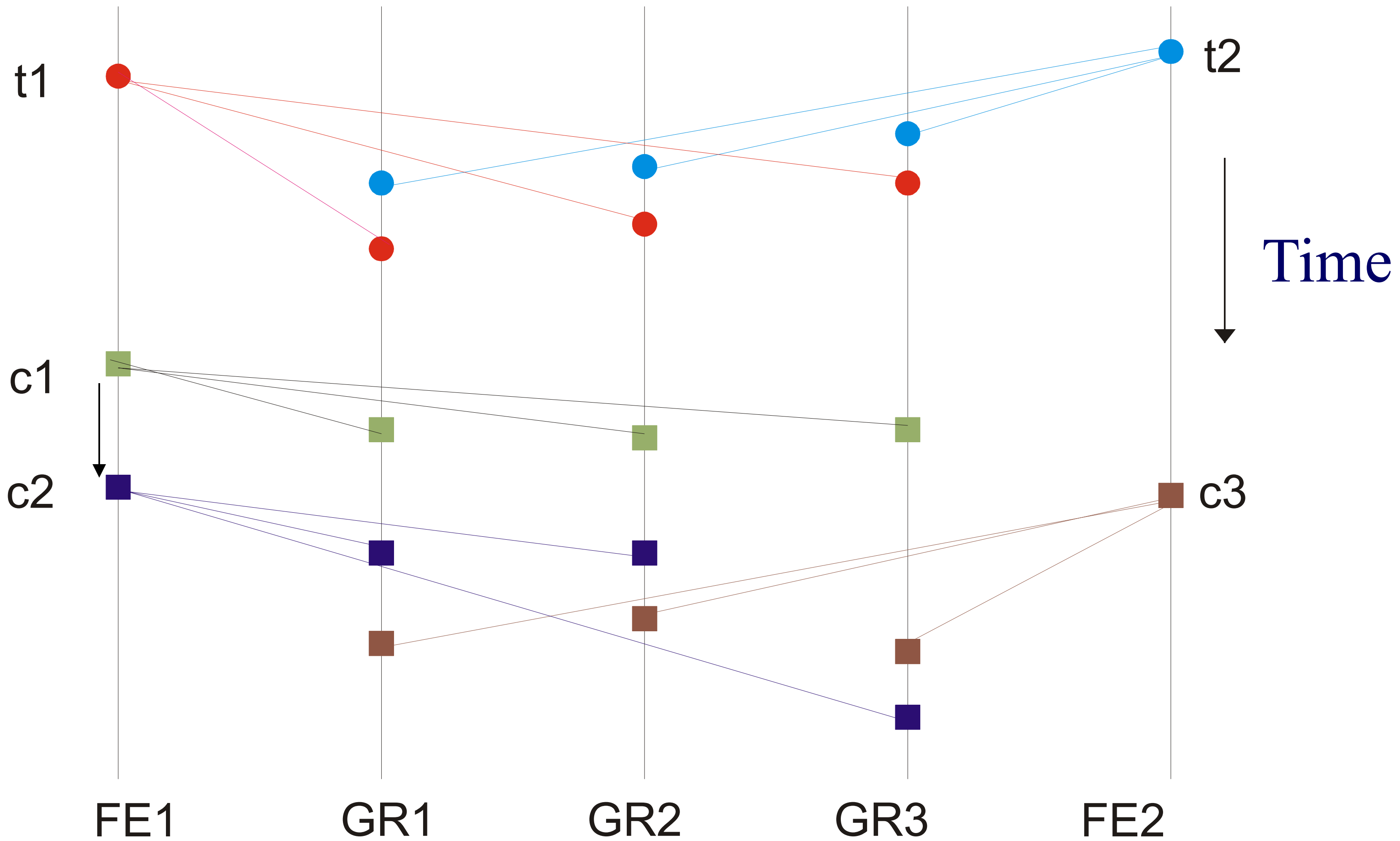
Example of unordered multicast



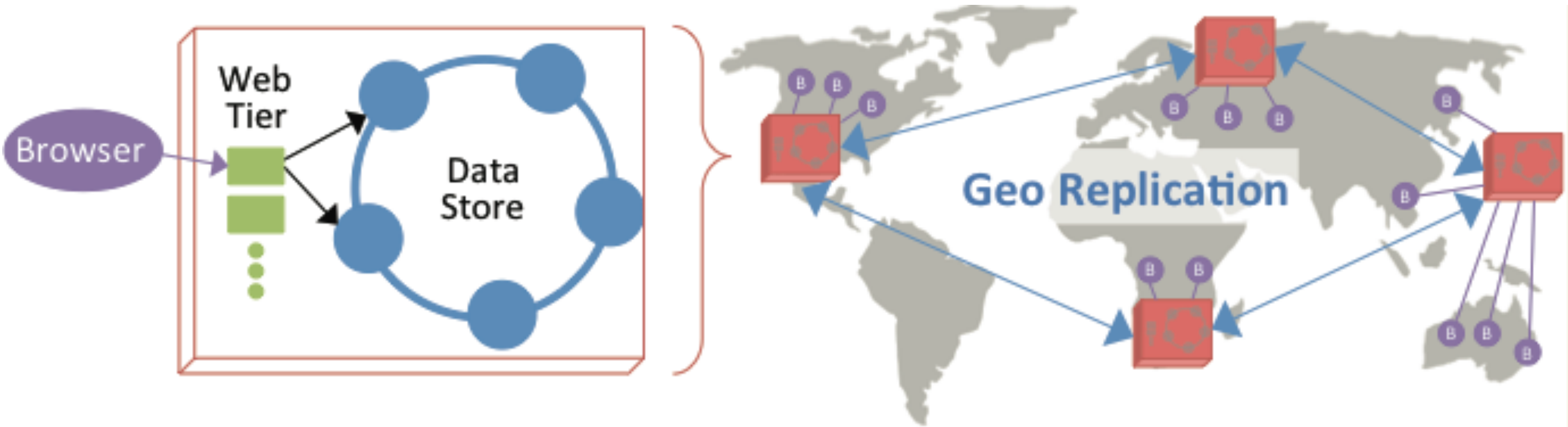
Sorting requests

- The order of request is important in distributed systems
  What happens in an asynchronous system when a client modifies a data and later another client request this information?
- Some applications require an order in making requests
- **Total ordering**: given two requests r_1 and r_2, r_1 is then processed in all proceedings before r2 or r_2 is processes before r_1
- **Causal ordering**: is based on potential causal relationships . If r_1 precedes r_2 then r_1 is processed before r_2 in all the processes

Total and causal ordering

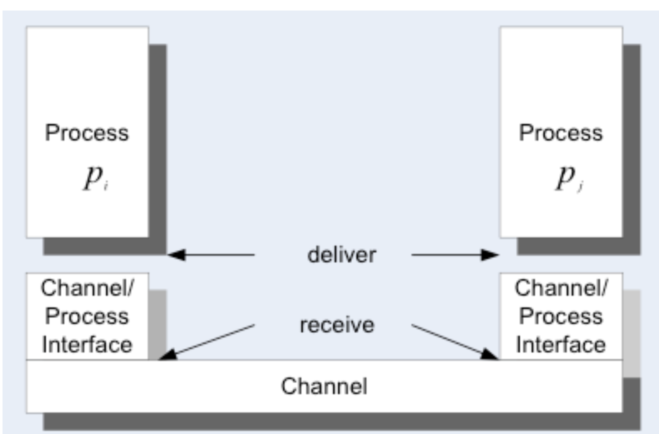


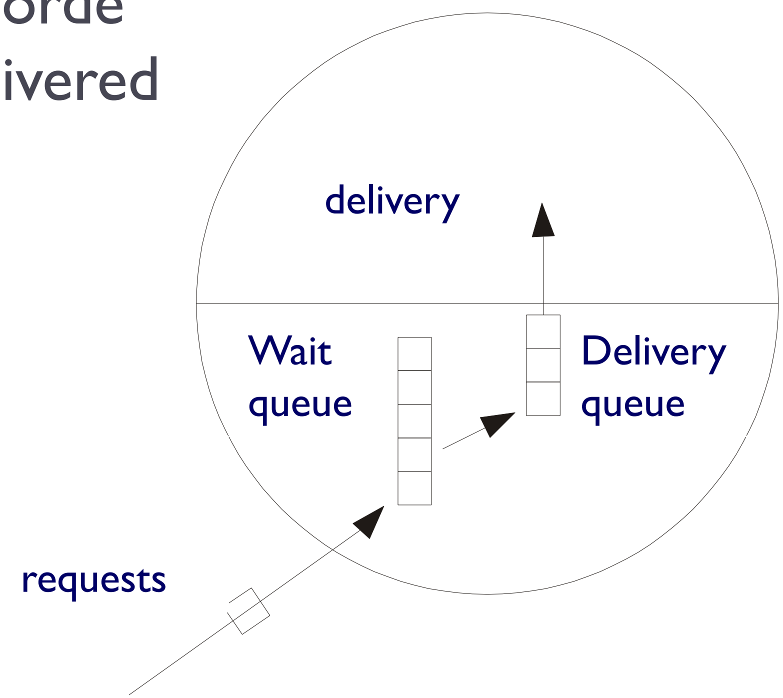
Problems in georeplicated servers



Implementation

- A request received is not delivered until the order restrictions can be met
- A stable message is queued for delivery
- Must be ensured that
  - **Segurity**: any message out of orde
  - **Progress**: tall mesages are delivered





Implementation of the total ordering

- Each request is assigned an identifier of total order (TOI)
- TOI is used to deliver messages in the same order to all processes
- **Centralized method**:
  - A sequencer process is on charge of assigning a TOI to each message
  - Each message is sent to the sequencer
    - The sequencer increments the TOI
    - The sequencer assigns a TOI and sends the message to the processes
  - When a process receives a message with a higher TOI of expected requests, it asks the sequencer to send the message again
  - Possible bottleneck and critical failure point

Distributed method

- Birman and Joseph 1987
- Each process *q* in the group stores:
  - A_q: the bigger aggreged sequence number observed
  - P_q: the bigger proposed sequence number
  - Identifiers must include the number of process to ensure a total order
- When a process *p* executes a BCAST, it sends the message to the rest
- Each process *q* recieves a message from *p*
  - Proposes $P_q = Max(A_q, P_q) + 1$
  - Stores (m, P_q) in the queue and marks the message as undeliverable
  - Sends P_q to the message source (p)
- The process *q* receives all the sequence numbers proposed and selects the highest A as the next sequence number and it sends this number to all
  - In process q, $A_q = Max(A_q, A)$ and the message is marked as deliverable
  - The delivery queue is ordered, and the first message is sent

Example

Node 1
A1 = 14
Multicast (M1)

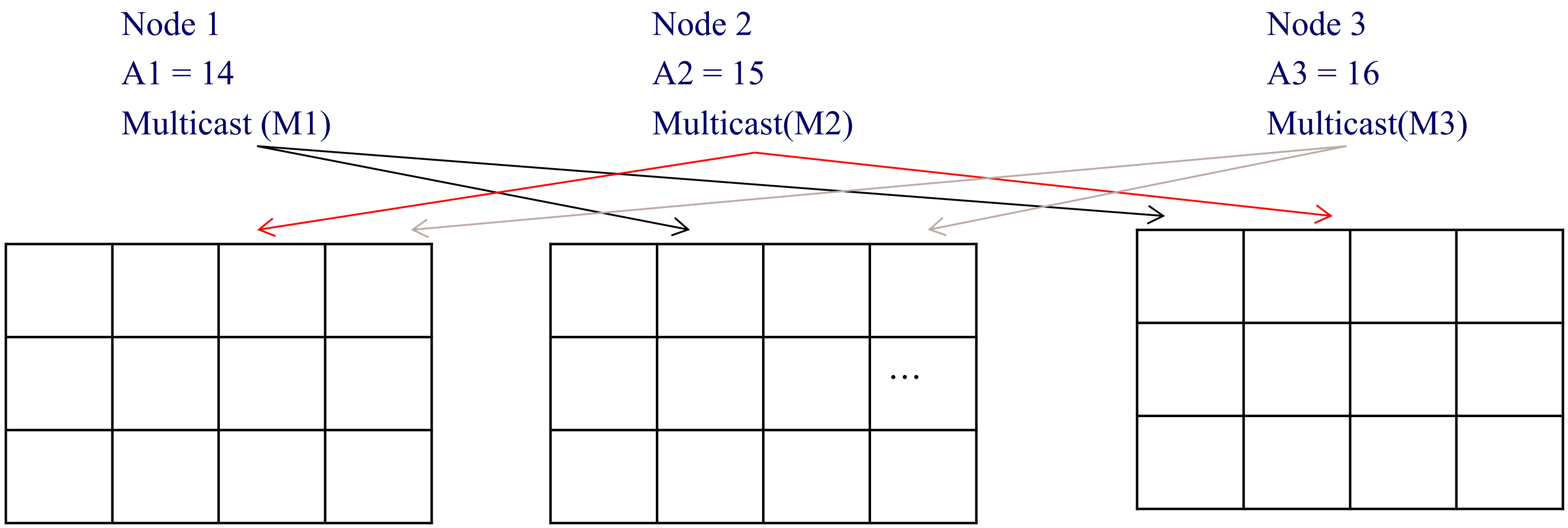
Node 2
A2 = 15
Multicast(M2)

			…

Node 3
A3 = 16
Multicast(M3)

Initially:
The three nodes perform a simultaneous multicast

Example



Initially:

The three nodes perform a simultaneous multicast

Example

Node 1
A1 = 14
Multicast (M1)

M3	M1	M2	
15.1	16.1	17.1	…
U	U	U	

Node 2
A2 = 15
Multicast(M2)

M2	M1	M3	
16.2	17.2	18.2	…
U	U	U	

Node 3
A3 = 16
Multicast(M3)

M1	M3	M2	
17.3	18.3	19.3	…
U	U	U	

Step 1:

- Messages arrive at the receivers in different orders
- They proposed a sequence number, P_q=Max(A_q, P_q) + 1
- (Process Identifier is added)
- Insert in queues and mark as undeliverable (U)

Example

Node 1
A1 = 14
Multicast (M1)

M3	M1	M2	
15.1	17.3	17.1	…
U	U	U	

Node 2
A2 = 15
Multicast(M2)

M2	M1	M3	
16.2	17.3	18.2	…
U	U	U	

Node 3
A3 = 16
Multicast(M3)

M1	M3	M2	
17.3	18.3	19.3	…
U	U	U	

Step 2:

- Node 1 receives the associated marks M1 sent by node 2 (17.2) and 3 (17.3)
- and calculates the maximum of the three , and sends the rest (17.3)

Example

Node 1
A1 = 14
Multicast (M1)

Node 2
A2 = 15
Multicast(M2)

Node 3
A3 = 16
Multicast(M3)

Node 1:

M3	M2	M1	
15.1	17.1	17.3	…
U	U	D	

Node 2:

M2	M1	M3	
16.2	17.3	18.2	…
U	D	U	

Node 3:

M1	M3	M2	
17.3	18.3	19.3	…
D	U	U	

Step 2:

- M1 is marked as undeliverable and queues are reordered
- M1 can be delivered in node 3 because being the top of the queue

Example

Node 1

A1 = 14

Multicast (M1)

Node 2

A2 = 15

Multicast(M2)

Node 3

A3 = 16

Multicast(M3)

M3	M2	M1	
15.1	17.1	17.3	…
U	U	D	

M2	M1	M3	
16.2	17.3	18.2	…
U	D	U	

M3	M2	
18.3	19.3	…
U	U	

Step 2:

- M1 is delivered to the node 3

Example

Node 1

A1 = 14

Multicast (M1)

M3	M2	M1	
15.1	17.1	17.3	…
U	U	D	

Node 2

A2 = 15

Multicast(M2)

M2	M1	M3	
16.2	17.3	18.2	…
U	D	U	

Node 3

A3 = 16

Multicast(M3)

M3	M2	
18.3	19.3	…
U	U	

Step 3:

- The Node 2 receives the brands associated with M2 sent by node 1 (17.1) and 3 (19.3) ,
- Calculates the maximum (19.3)

Example

Node 1
A1 = 14
Multicast (M1)

M3	M2	M1	
15.1	19.3	17.3	…
U	U	D	

Node 2
A2 = 15
Multicast(M2)

M2	M1	M3	
19.3	17.3	18.2	…
U	D	U	

Node 3
A3 = 16
Multicast(M3)

M3	M2	
18.3	19.3	…
U	U	

Step 3:

- The Node 2 receives the brands associated with M2 sent by node 1 (17.1) and 3 (19.3) ,
- Calculates the maximum (9.3)
- It sends the rest

Example

Node 1
A1 = 14
Multicast (M1)

Node 2
A2 = 15
Multicast(M2)

Node 3
A3 = 16
Multicast(M3)

M3	M1	M2	
15.1	17.3	19.3	…
U	D	D	

M1	M3	M2	
17.3	18.2	19.3	…
D	U	D	

M3	M2	
18.3	19.3	…
U	D	

Step 3:

- M2 is marked as undeliverable and queues are reordered

Example

Node 1
A1 = 14
Multicast (M1)

Node 2
A2 = 15
Multicast(M2)

Node 3
A3 = 16
Multicast(M3)

M3	M1	M2	
15.1	17.3	19.2	…
U	D	D	

M3	M2	
18.2	19.3	…
U	D	

M3	M2	
18.3	19.3	…
U	D	

Step 3:

- M2 is marked as undeliverable and queues are reordered
- M1 is delivered to the node 2

Example

Node 1
A1 = 14
Multicast (M1)

Node 2
A2 = 15
Multicast(M2)

Node 3
A3 = 16
Multicast(M3)

M3	M1	M2	
15.1	17.3	19.2	…
U	D	D	

M3	M2	
18.2	19.3	…
U	D	

M3	M2	
18.3	19.3	…
U	D	

Step 4:

- Node 3 receives the associated marks M3 sent by node 1 (15.1) and 3 (18.2)
- Calculates the maximum of all (18.3)

Example

Node 1
A1 = 14
Multicast (M1)

Node 2
A2 = 15
Multicast(M2)

Node 3
A3 = 16
Multicast(M3)

M3	M1	M2	
18.3	17.3	19.2	…
U	D	D	

M3	M2	
18.3	19.3	…
U	D	

M3	M2	
18.3	19.3	…
U	D	

Step 4:

- Node 3 receives the associated marks M3 sent by node 1 (15.1) and 3 (18.2)
- Calculates the maximum of all (18.3)
- It sends the rest

Example

Node 1
A1 = 14
Multicast (M1)

M1	M3	M2	
17.3	18.3	19.2	…
D	D	D	

Node 2
A2 = 15
Multicast(M2)

M3	M2	
18.3	19.3	…
D	D	

Node 3
A3 = 16
Multicast(M3)

M3	M2	
18.3	19.3	…
D	D	

Step 4:

- M3 is marked as undeliverable and queues are reordered
- We can deliver all messages to all nodes
- The delivery order : M1 , M3 and M2 (the order ensures delivery on all nodes)

Implementation of causal ordering

- Each process p_i stores a vector **VT** with n components
- In the process p_j, the **i** component indicates the last message received from **i**
- Algorithm to update the vector
  - All the processes set the vector with 0s
  - When p_i sends a new message, it increases $VT_i(i)$ by 1 and adds VT to the message
- When p_j get a message from with VT, it is delivered if:
  - $vt(i) = VT_j(i) + 1$ (next in the sequence of p_i)
  - $vt(k) \leq VT_j(k)$ for all $k \neq i$ (all previous messages have been delivered to i)
- When a message with VT is delivered, we update the table of p_j:
  - $VT_j = max(VT_j, VT)$, for k=1, 2, ... , n

Example