Synchronization

Agenda

- Time synchronization
- Logical clocks
- Distributed Mutual exclusion

References

- Chapter 6 of “Distributed Systems” by M. van Steen and A. S. Tanenbaum
- Chapter 9 of “Design and Analysis of Distributed Algorithms” by N. Santoro

Clock Synchronization

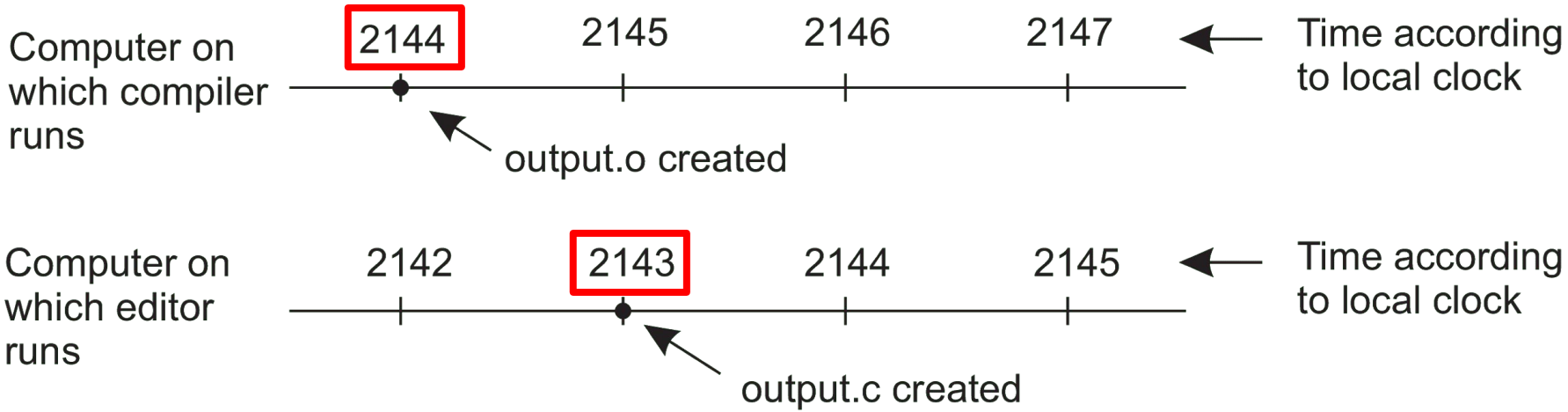
Recall the assumptions of our model

- In a distributed system each machine/entity has its local own clock
- There is no notion of global time, but the time is something that depends on the single entity and on the observer

We may end up in a situation where an event occured after another one is assigned an earlier time

Example: make on Different Machine

Lack of time synchronization: a possible object file mismatch

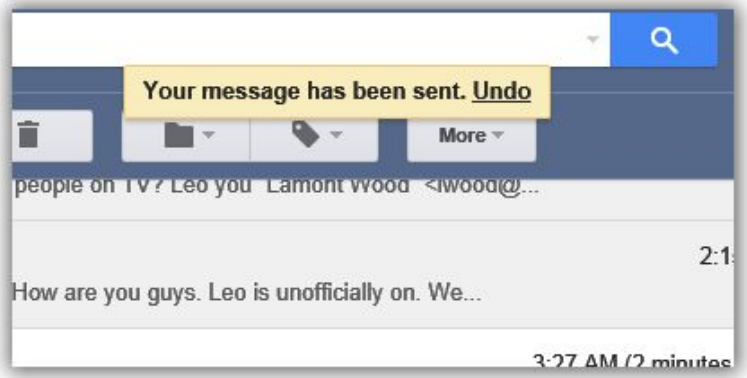


2143 < 2144 make doesn't call the compiler!

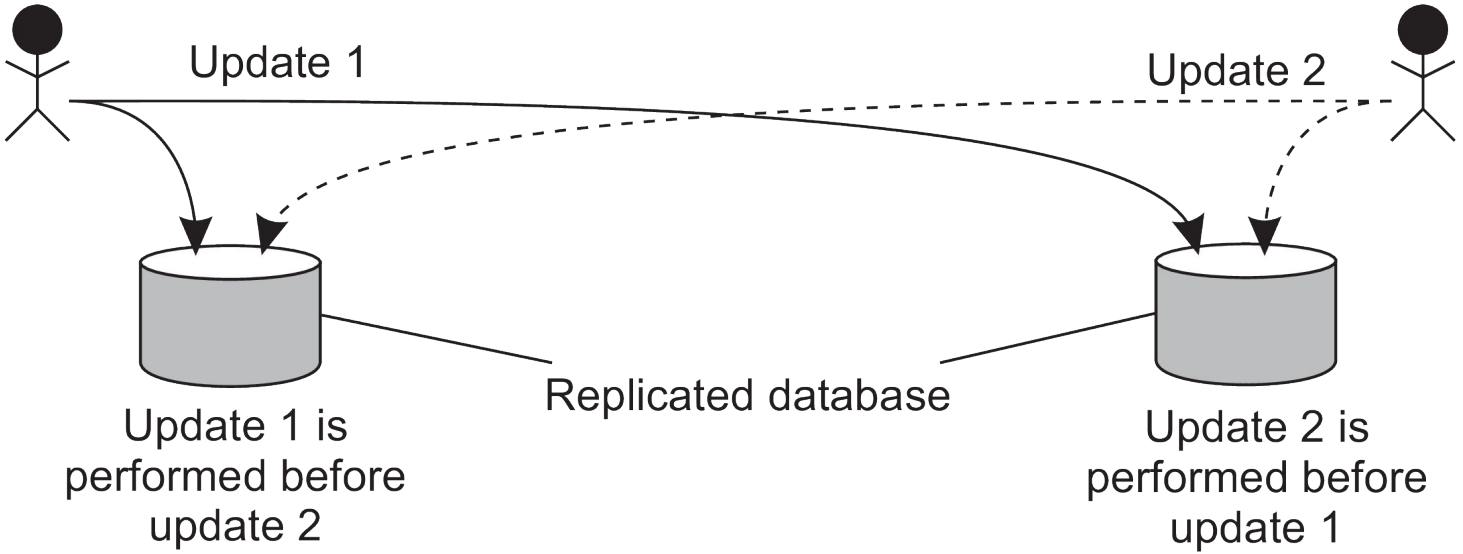
Example: Undoing Actions

The goal: undo the transmission of a message **M** and all messages transmitted because of **M**

The idea: if t(**B**) < t(**A**) then the transmission of **B** is not caused by **A**



Example: Database Replication



Update 1: deposit $100 to the account
Update 2: increase the account with 1% interest

Problem: two updates must be performed in the same order on both replicas

The Need for Clock Synchronization

Many applications rely on an accurate account of the time

- Financial brokerage
- E-commerce (transactions between your bank and shops)
- Environmental sensing

Problem: is it possible to synchronize all clocks in a distributed system?

Measuring Time in a Single Machine

- When a process **A** wants to know the current time **Ta**, it carries out a system call to the operating system
- **Later** the process **B** asks for the time too, it obtains **Tb**
- On a single machine it holds

Ta <= Tb

Physical Clocks

- Timers are electronic devices that count oscillations occurring in a crystal at definite frequency
- A crystal has associated two registers: the counter and the holding register
- The counter is decremented at each oscillation
- When the counter reaches zero an interrupt is triggered (clock tick) and the value of counter is set using the holding register

When We Consider Many Machine...

- Crystals in different machine don’t run all at exactly the same frequency
- **Clock skew** => there are differences among the values read by the different machines
- The network exacerbates the problem
  - Arbitrary message delays
  - Messages may not arrive at all

Synchronizing Physical Clocks

External synchronization: keep all machines synchronized to an **external reference clock**

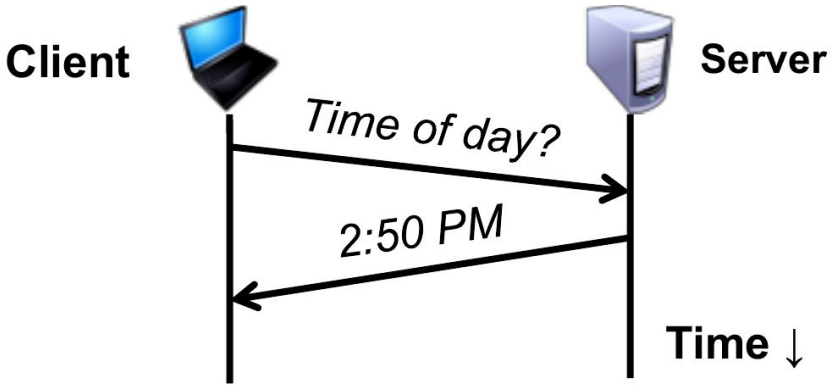
Internal synchronization: keep all machines synchronized with each other

Coordinated Universal Time (UTC)

- International standard for timekeeping
- It is based on atomic time and its signals are broadcast regularly by land-based radio stations and satellites
- UTC can is used as an external reference source to synchronized all nodes of a distributed system

Synchronization to a Time Server (Naive Solution)

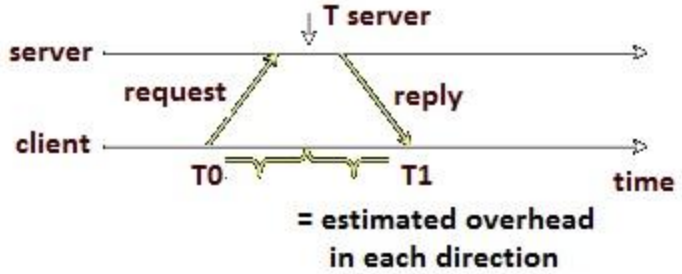
- Assume there is a server with an accurate clock, e.g., it has a UTC receiver
- We can ask the current time to the server



- **Wrong results**: network delays outdate the server answer

Cristian's algorithm

1. The client sends a request to the server together with a timestamp **T0**
2. The server answers sending its time **T**
3. The client timestamp when the message is received **T1**
4. The client can estimate the round trip time as (T1 - T0)/2



$$T_{new} = T_{server} + \frac{T_1 - T_0}{2}$$

Berkeley algorithm

Perform **internal synchronization** among a set of machines

The idea: the time server scans all machines periodically, calculates an average, and informs the others on how to adjust their time

Assumptions

- No machine has access to an external clock
- All machine have accurate local clocks

The Algorithm

1. A master is chosen via an election algorithm
2. It polls the slaves who reply with their time
3. It observes the RTT of the messages and estimates the time of each slave and its own
4. It then averages the clock times
5. Finally, the master sends out the amount (positive or negative) that each slave must adjust its clock

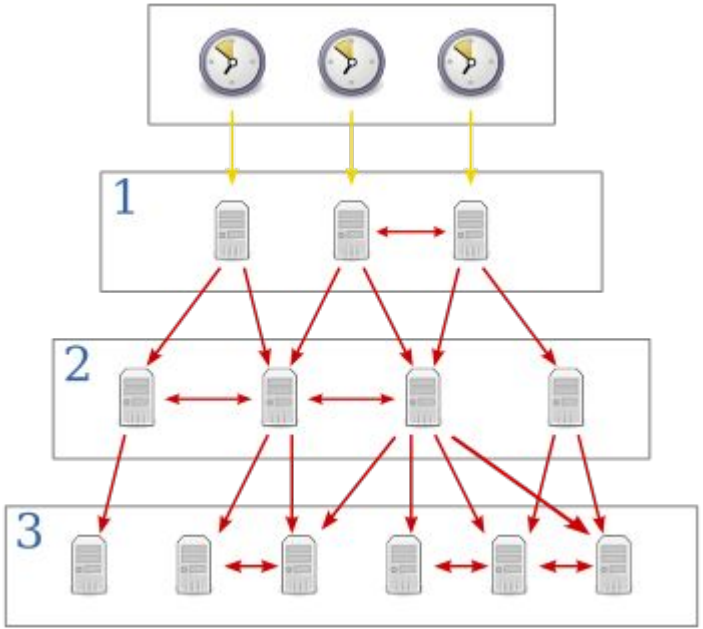
The Network Time Protocol (NTP)

- NTP allows client to be synchronized to UTC
- It is widely deployed throughout the Internet, and it is the state of the art in time synchronization protocols for unreliable networks
- It is designed to provide a reliable service able to survive to lengthy losses of connectivity

System Structure

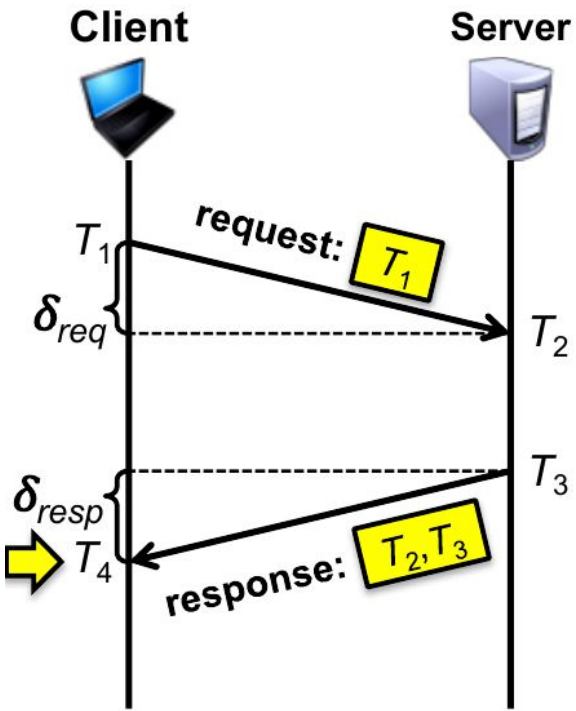
Server and time sources are arranged in layers (strata)

- Stratum 0 => high-precision timekeeping devices such as atomic clocks (reference clocks)
- Stratum 1 => machines whose system time is synchronized to within a few microseconds of their attached stratum 0 devices
- ....



How Does The Procol Work?

1. The client sends a request timestamped with **T1**
2. The server timestamp the receipt of the message with **T2**
3. The server replies with a timestamp **T3**
4. The client compute the timestamp **T4** when the answer arrives



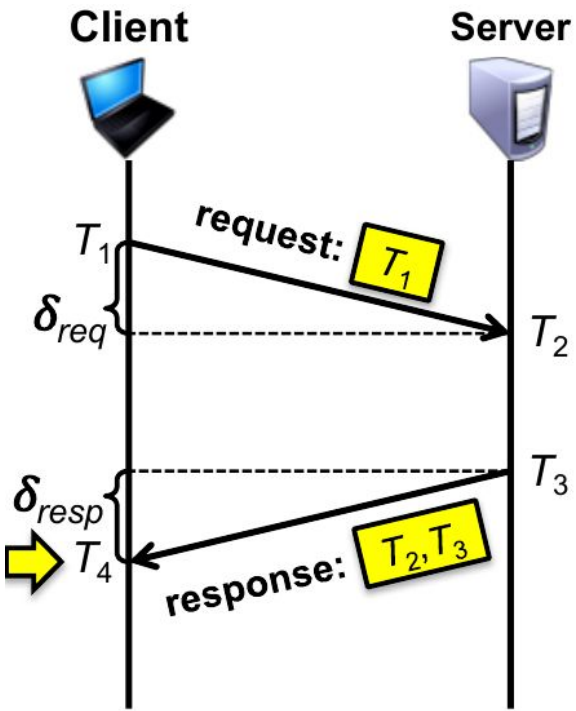
How Does The Procol Work?

The client computes the offset and the RRT

$$\Theta = \frac{(T2 - T1) + (T4 - T3)}{2}$$

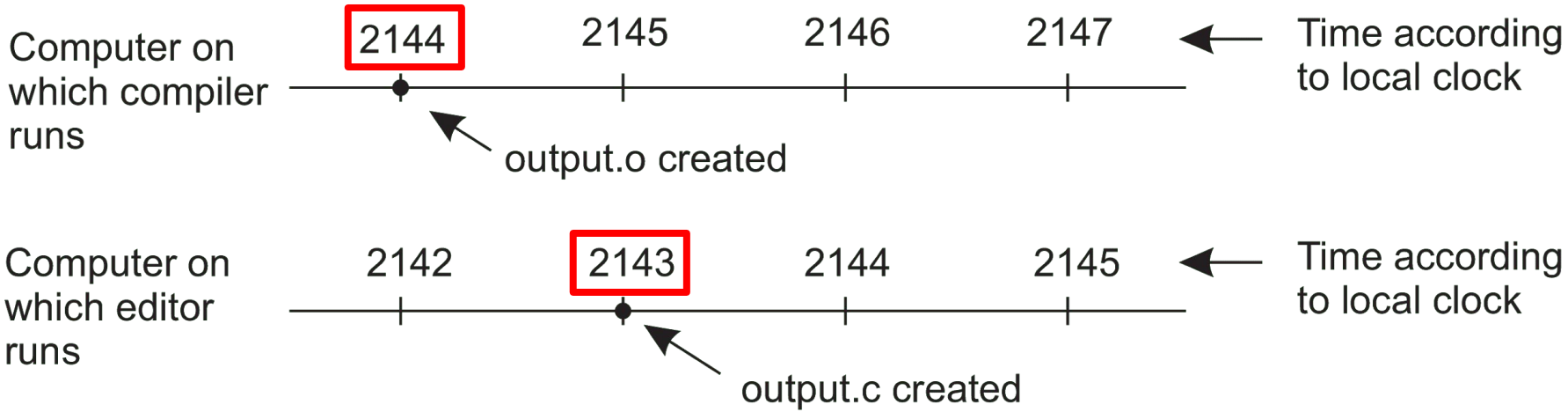
$$\delta = (T4 - T1) - (T3 - T2)$$

It takes many measurements from different servers and then takes the one with the minimum value of δ



Logical Clocks

Observation: the actual time of events doesn't matter, what it is important is the relation between two events



We only need to know that output.c was modified after output.o creation!

Logical Clocks

The idea: ignore the precise clock time and consider the order in which two events occur

Happens-before relation

Strict Partial Order

A strict partial order **<** defined over a set **E** is a binary relation (**<** $\subseteq$ **E**x**E**) that for all a,b,c in **E** satisfies

1. not a **<** a (**irreflexivity**)
2. a **<** b and b **<** c => a **<** c (**transitivity**)
3. if a **<** b then not b **<** a (**asymmetry**)

Happens-before Relation

Given a set of events **E**, the happened-before relation

$$\text{->} \subseteq \mathbf{E} \times \mathbf{E}$$

is the least **strict partial order** on events such that

1. If events a and b occur on the same node, a -> b if the occurrence of event a preceded the one of event b
2. If event a is the sending of a message and event b is the reception of the message sent in the event a, a -> b

Note

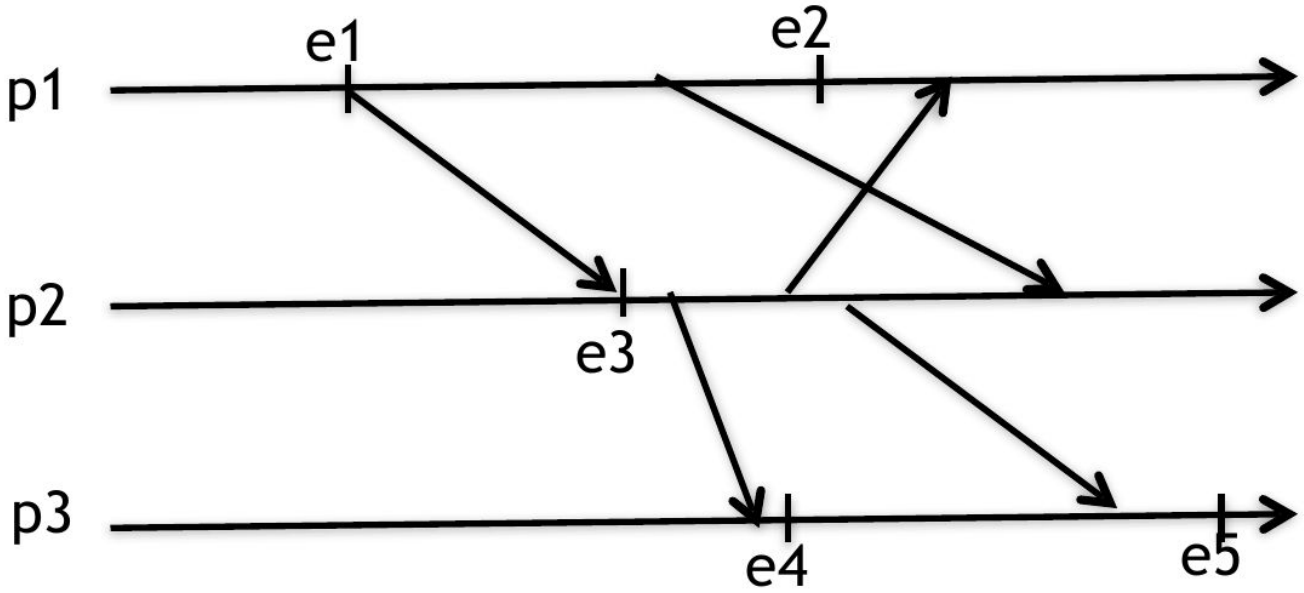
The happens-before relation correlates events occuring on the same node or among nodes that exchange messages

Two events a and b occuring on two nodes that do not exchange messages are not related by ->, i.e.,

a -> b and b -> a are **not** true => a and b are **concurrent**

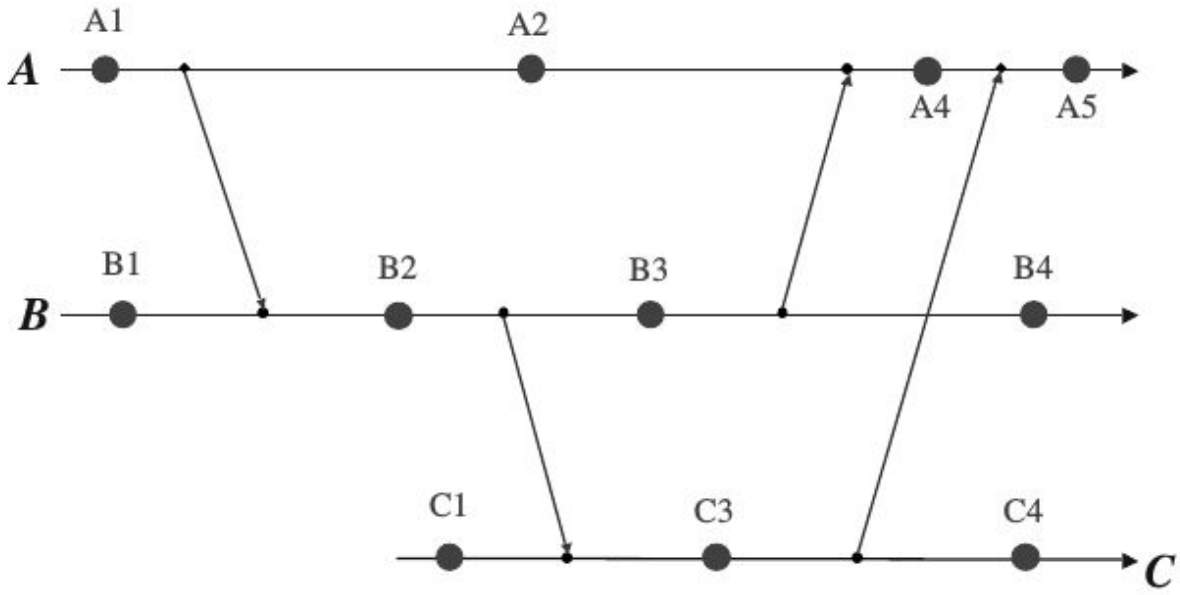
We cannot say anything about which event happened first

Example



e1 -> e3, e1 -> e2, e3 -> e4, e1 -> e5, e2 || e5

Example



A1 -> B2 -> C3, B3 -> A4, C3 -> A5, B3 || C3, C3 || B4, B4 || C4

Virtual Time

Problem: using local clocks and counters, can we create an common notion of time **C** among all entities in a distributed system?

Given an event **e**, assign to it a time value **C(e)** on which all entities agree

Req 1: if **a ->b ⇒ C(a) < C(b)**

Req 2: can we avoid generating additional messages?

Virtual Time

The idea: each entity maintain a virtual clock $\mathbf{C}_x$ that assigns an integer to each event occurring locally

Define a global clock **C** from the virtual clocks of the entities

$$\mathbf{C}(a) = \mathbf{C}_x(a)$$

For each event a occurring at x

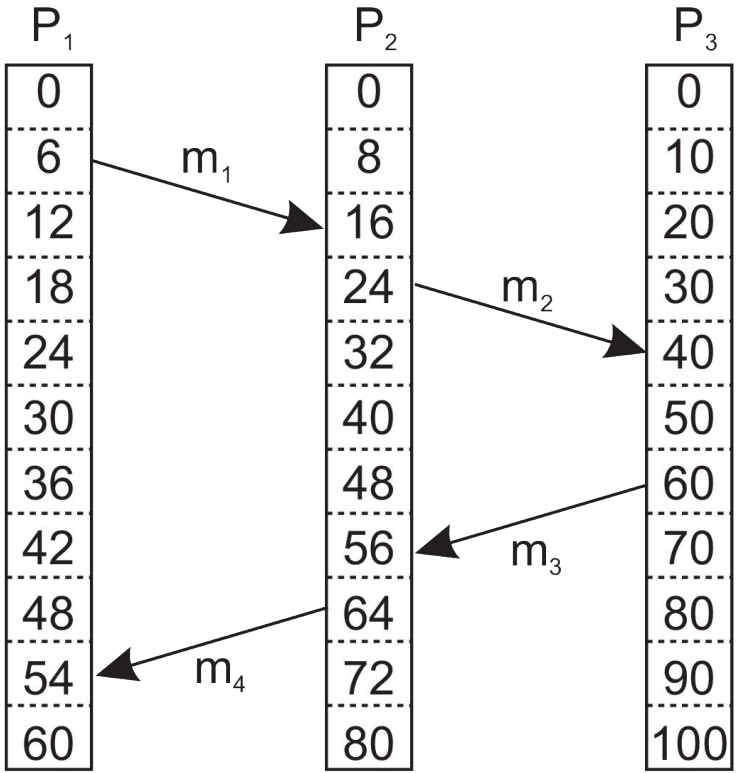
Example

Problem: logical clocks are out of synch

How can we synch them?

m4 leaves process P2 at time 64 and reaches P2 at time 54

No: the message is received before it was sent



m3 leaves process P3 at time 60 and reaches P2 at time 56

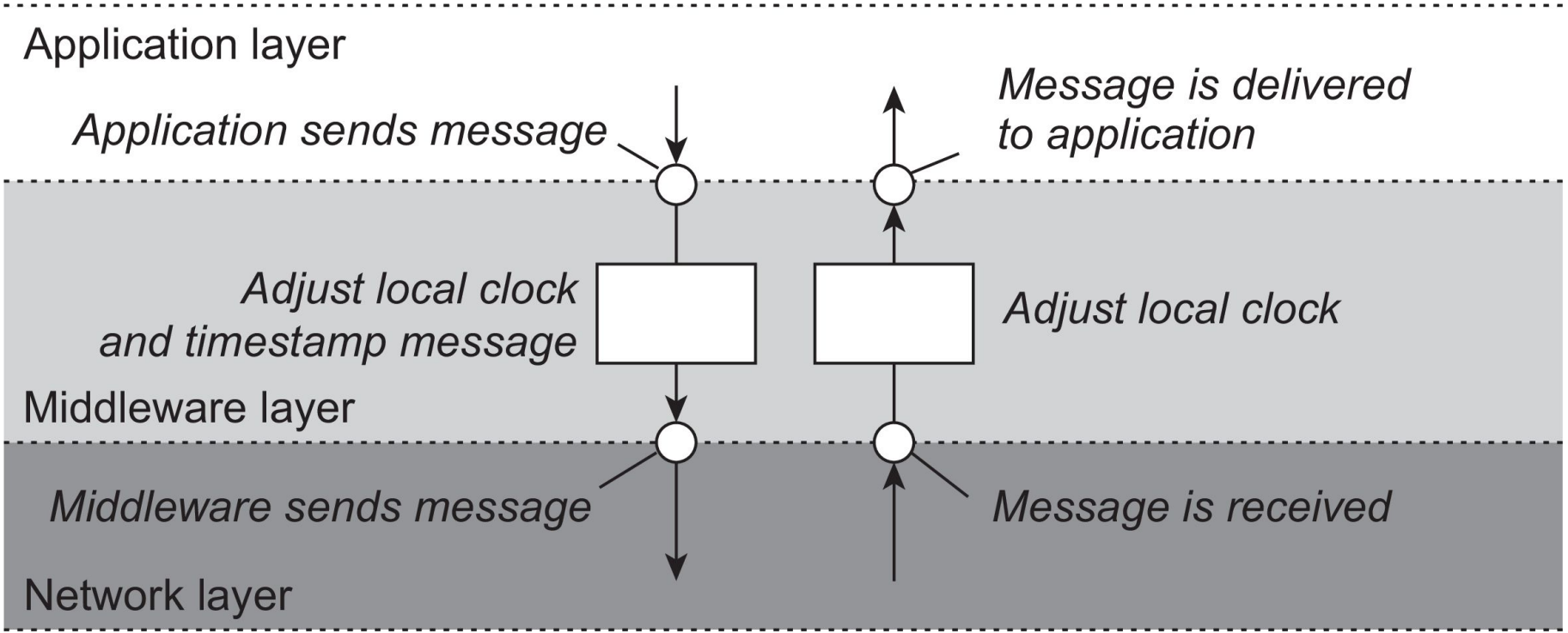
No: the message is received before it was sent

Algorithm Counter Clock

Each entity x $\quad\quad$ $\mathbf{C_x}$ clock of x

1. Before performing an event, $\mathbf{C_x := C_x + 1}$
2. When x sends a message **M**, x sets ts(**M**) with $\mathbf{C_x}$ in **M**
3. When x receives a message **M** from y, it adjusts its local counter $\mathbf{C_x := max\{C_x, ts(M)\}}$, then execute step 1 before processing the **M**

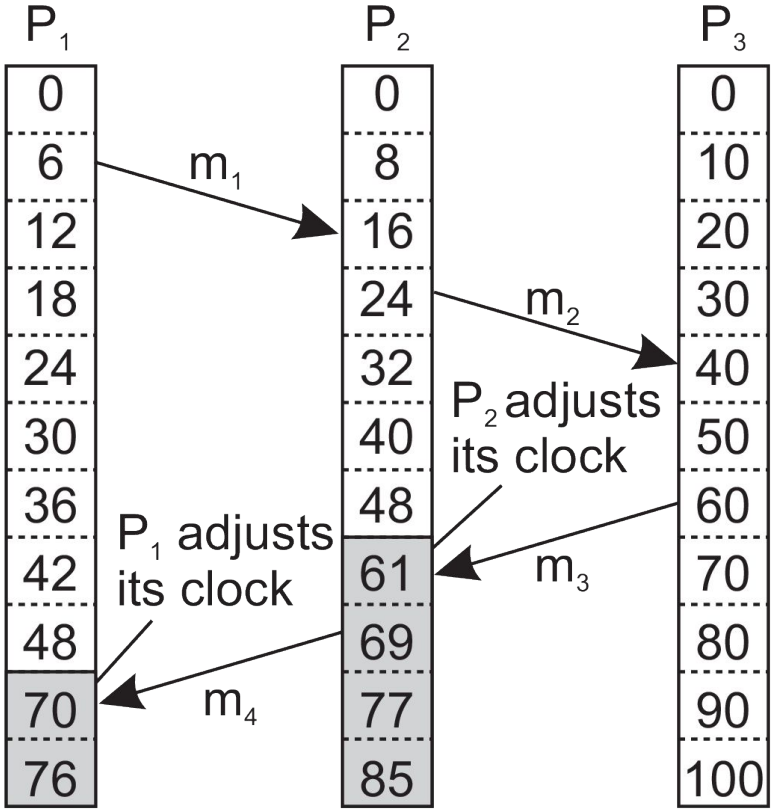
Positioning of Lamport's Logical Clocks



Example

m4 leaves process P2 at time 64 andm4 leaves process P2 at time 64 and reaches P2 at time 54 reaches P2 at time 54

OK: the message is received after it was sent

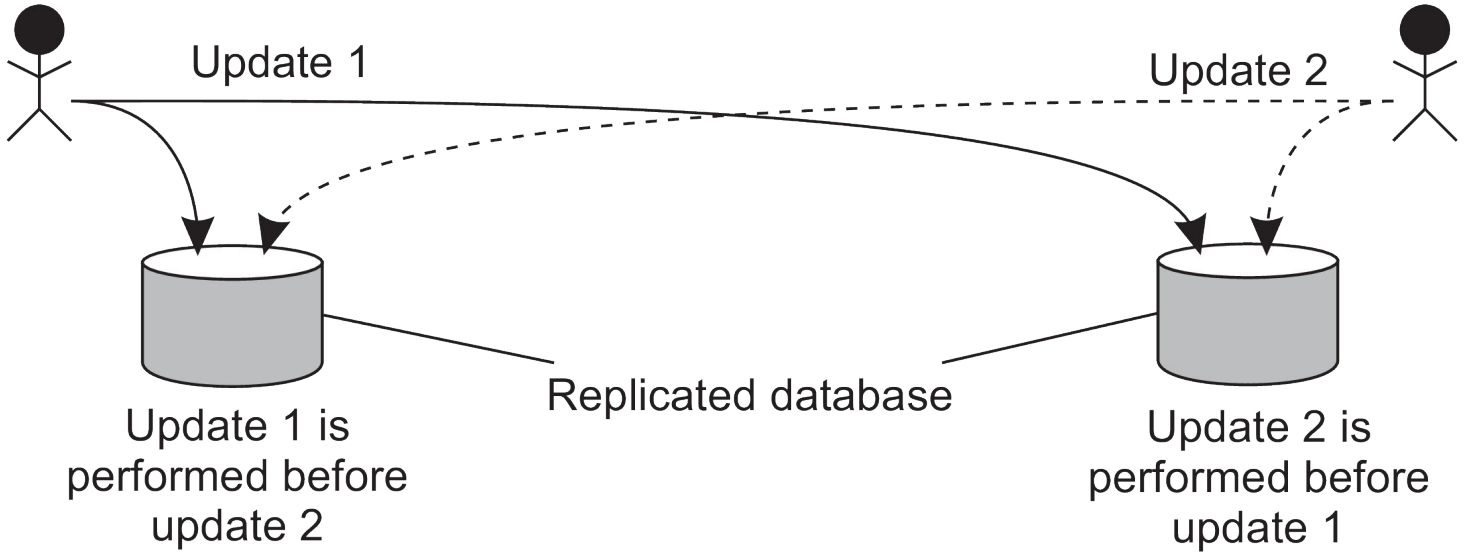


m3 leaves process P3 at time 60 and reaches P2 at time 61

OK: the message is received after it was sent

The relative order of events is preserved

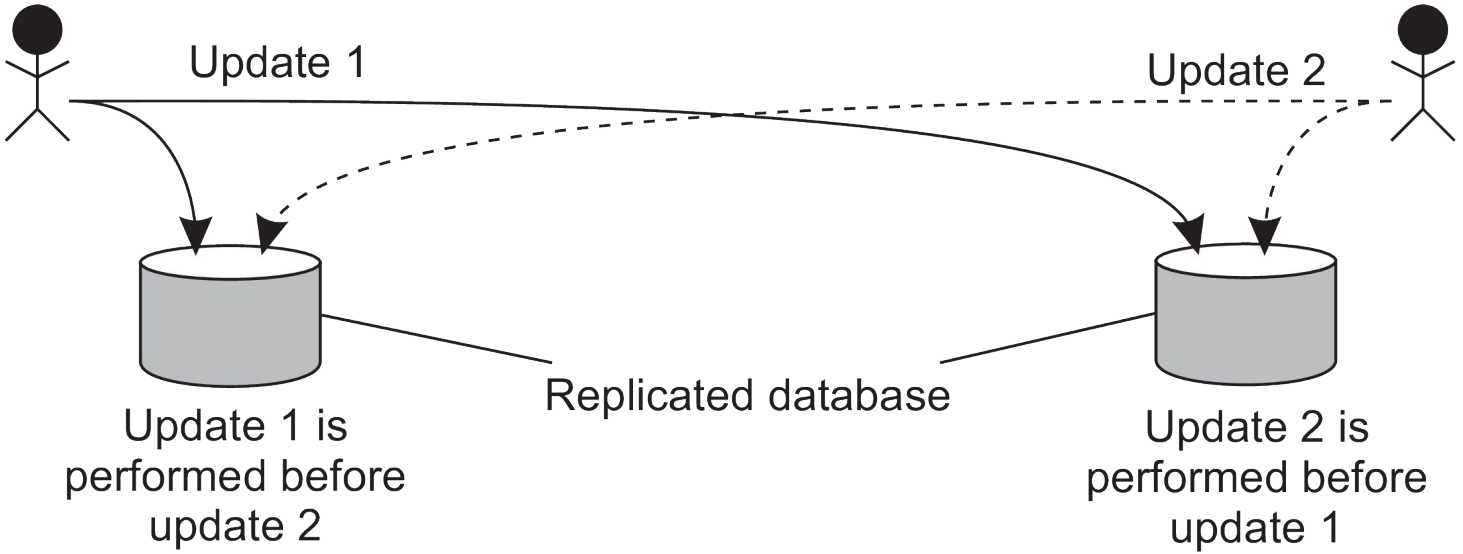
Application: Total-ordered Multicast



Update 1: deposit $100 to the account
Update 2: increase the account with 1% interest

Problem: two updates must be performed in the same order on both replicas

Application: Total-ordered Multicast



Problem: two updates must be performed in the same order on both replicas

Solution: **Total-ordered multicast** allows all messages to be **delivered** in the same order to each receiver

Assumptions

- Group of entities multicasting messages to each other (complete network)
- Each message is timestamped with the current logical time by the sender
- Messages from the same sender are received in the order they were sent
- Each entity maintains a queue of messages
- A message is conceptually sent also to the sender

The algorithm

- When an entity **x** receives a message **M**, it uses Lamport's algorithm to adjusts its local clock
- And then **x** stores **M** inside its local queue
- The receiver **x** sends an ACK message to all other entities in the group
- A message is delivered to an application when it is in front of the queue and has been acknowledged by other entities in the group

Observations

- Logical clocks ensures that ACK messages have a higher timestamps than the received message
- If a message is removed from the queue, we also remove all corresponding ACK messages
- All entities in the group will have identical local queues, i.e., same messages and in the same order,

Messages are delivered in the same order

Recap on Logical Clocks

What we have

- All events in a distributed systems are totally ordered
- If a -> b (a happens before b) then **C**(a) < **C**(b)

Question: if **C**(a) < **C**(b), can we say anything on the relation between a and b?

Let's consider

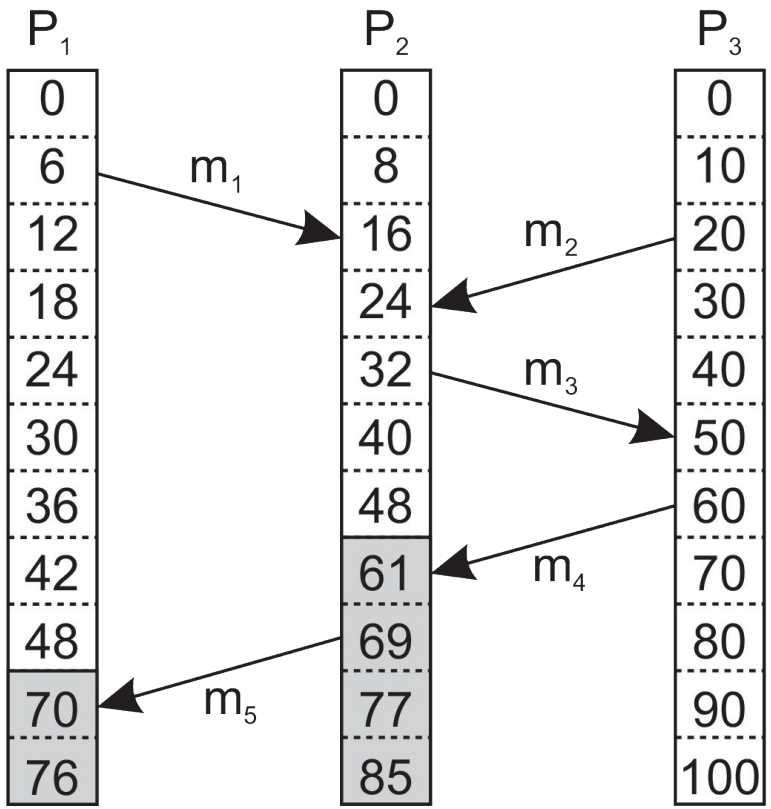
$T_{snd}(m1) = 6$ $T_{rcv}(m1) = 16$
$T_{snd}(m3) = 32$ $T_{rcv}(m3) = 50$

Logical clocks ensure
$T_{snd}(m1) < T_{rcv}(m1) \Rightarrow a \rightarrow b$
$T_{rcv}(m1) < T_{snd}(m3) \Rightarrow b \rightarrow c$

a, b concern m1
b, c are local event of P2

What about $T_{rcv}(m1) < T_{snd}(m2) \Rightarrow ???$?

We cannot say anything: sending m2 has nothing to do with receiving m1 (m1 || m2)



Logical Clocks Limitations

The problem with Lamport clocks is that they do not capture causality

We cannot infer relationships among events using timestamps

Vector Clocks

- Vectors clocks are more detailed representation of what a node knows about events in other entities
- Assume there are **n** entities in our system
- Each entity **x** maintains a vector $\mathbf{VC_x}$
  a. $\mathbf{VC_x[x]}$ is the number of events that occurred at **x** (the logical clock of **x**)
  b. $\mathbf{VC_x[y] = k}$ means that **x** knows that **k** events have occurred at **y** (**x**'s knowledge of the local time of **y**)

Algorithm VectorClock

Each entity **x** $\quad\quad$ $\mathbf{VC_x}$ vector clock of x

1. Before performing an event, $\mathbf{VC_x[x] := VC_x[x] + 1}$
2. When x sends a message **M**, **x** sets the *vector timestamp* **ts(M)** to $\mathbf{VC_x}$ after executed 1
3. When **x** receives a message **M** from **y**, it adjusts its local vector $\mathbf{VC_x[i] := max\{VC_x[i], ts(M)[i]\}}$, for each **k** then execute step 1 before processing the message **M**

Observations

1. When **y** receives **M** from **x** ⇒ ts(**M**)[**x**] - 1 is the number of events occurred at **x** that precede the sending of **M**
2. Also, **x** tells **y** on how many events occurred at other entities before sending **M** (**x**'s view)

ts(**M**) tells **y** how many events in other entities have preceded the sending of **M**

How Can We Use Vector Clocks?

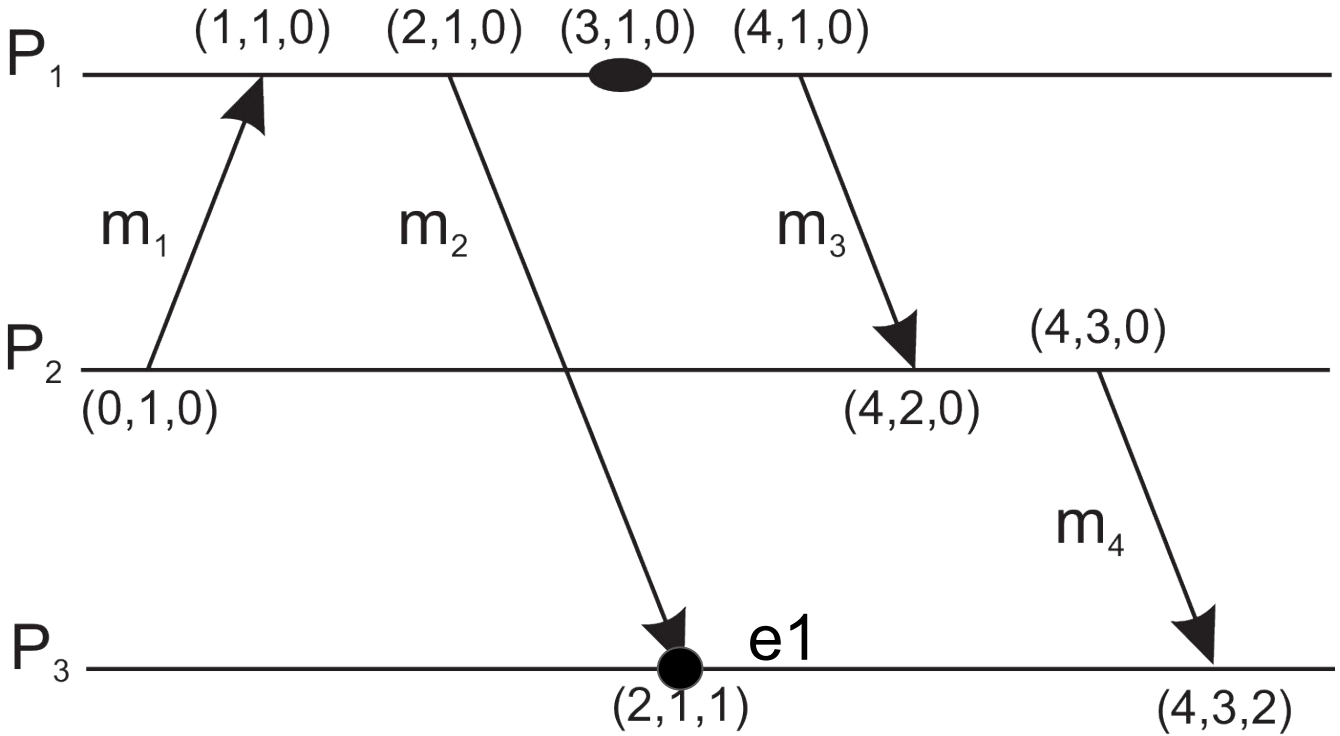
- **ts(M)** is vector timestamp of message **M**
- **ts(M1)** < **ts(M2)** if and only if

for all k= 1,...,n **ts(M1)[k] <= ts(M2)[k]** and

there exists **k'** such that **ts(M1)[k'] < ts(M2)[k']**

- **M1** and **M2** are *concurrent* if neither **ts(M1)** < **ts(M2)** nor **ts(M1)** > **ts(M2)**

Example

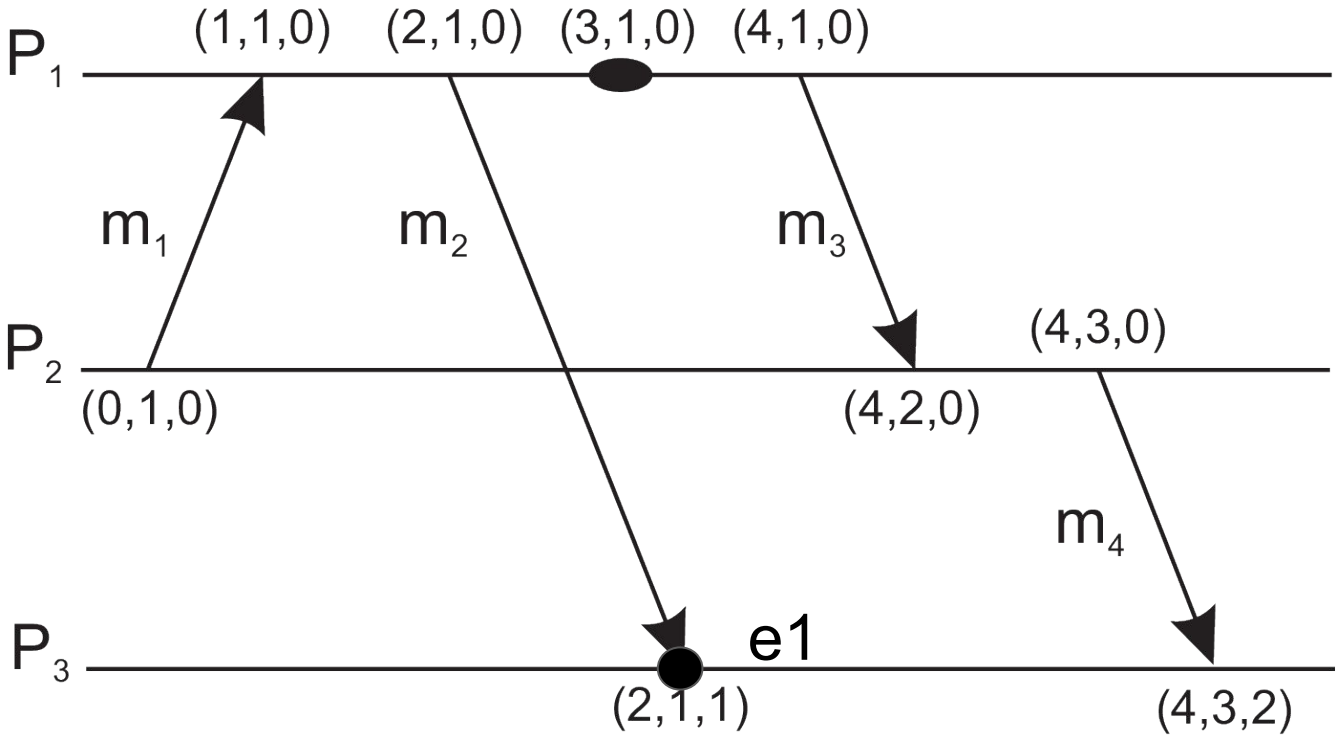


ts(m2) = (2, 1, 0)
ts(m4) = (4, 3, 0)
ts(e1) = (2, 1, 1)
ts(m3) = (4,1,0)

ts(m2) < ts(m4) then m2 may casually precede m4

Question
ts(e1) < ts(m3)?
ts(m3) < ts(e1)?

Example

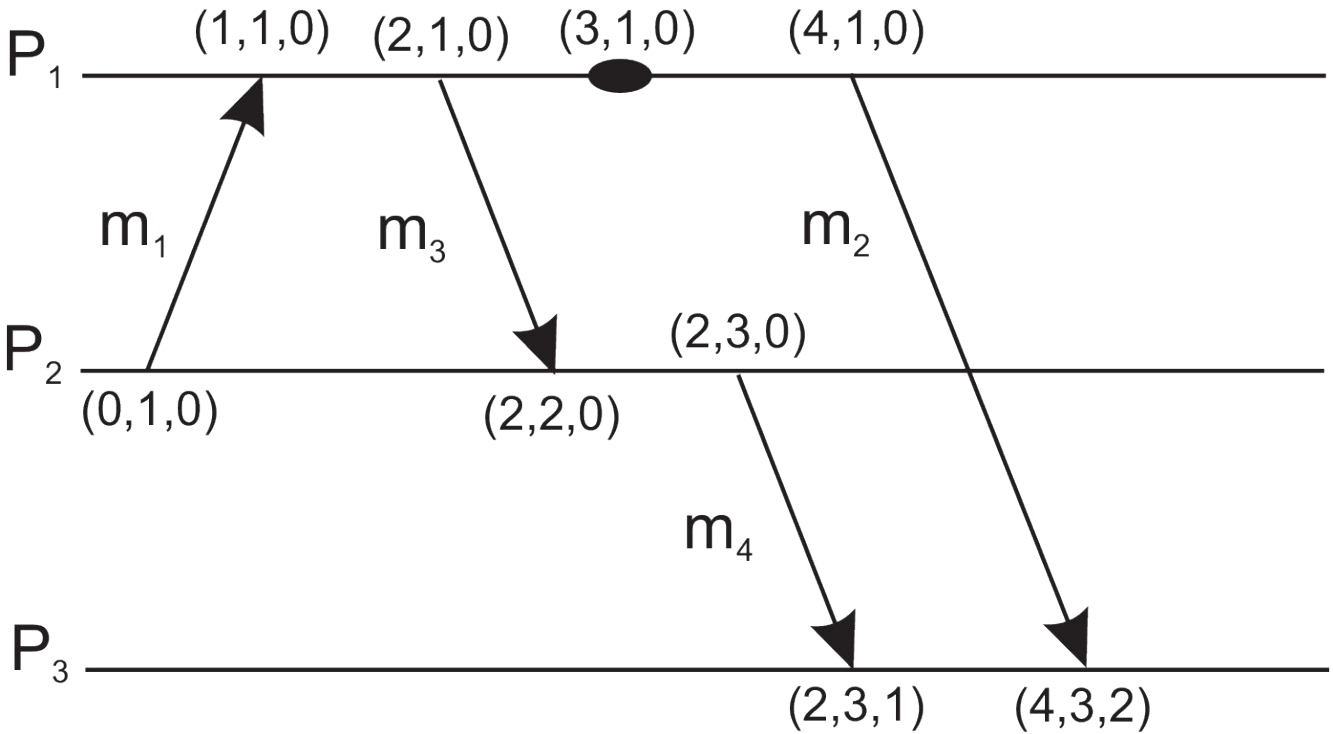


ts(m2) = (2, 1, 0)
ts(m4) = (4, 3, 0)
ts(e1) = (2, 1, 1)
ts(m3) = (4,1,0)

ts(m2) < ts(m4) then m2 may casually precede m4

Question
ts(e1) < ts(m3)? **No**
ts(m3) < ts(e1)? **No**

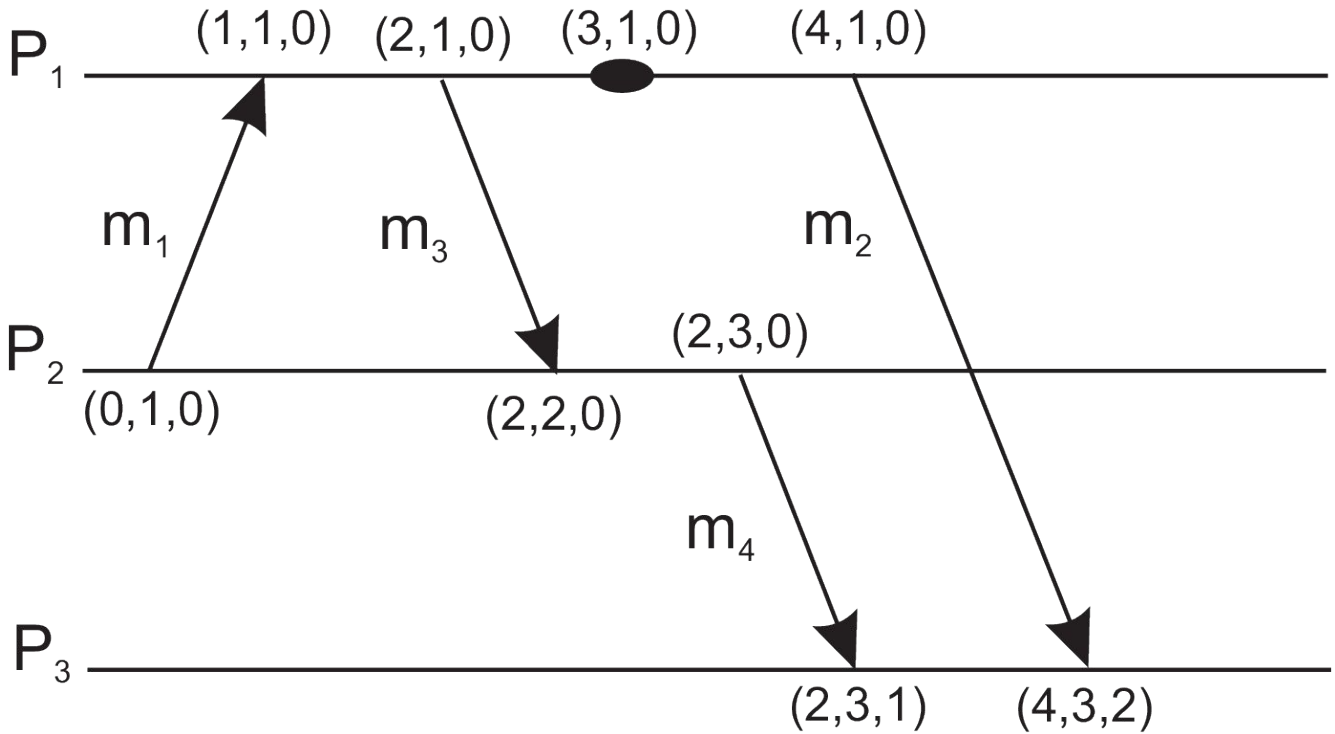
Example



ts(m2) = (4, 1, 0)
ts(m4) = (2, 3, 1)
ts(m3) = (2, 1, 0)

Question
ts(m2) < ts(m4)?
ts(m4) < ts(m2)?

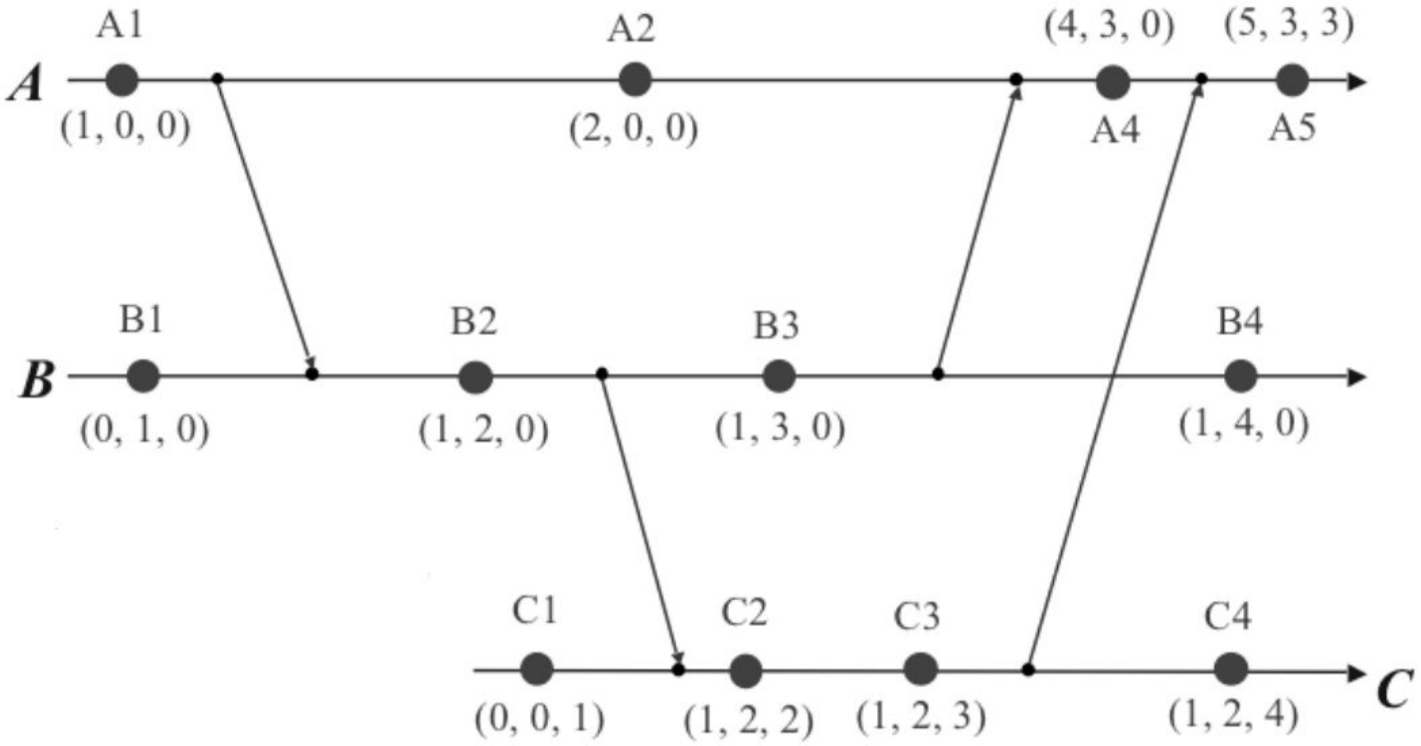
Example



ts(m2) = (4, 1, 0)
ts(m4) = (2, 3, 0)
ts(m3) = (2, 1, 0)

Question
ts(m2) < ts(m4)? **No**
ts(m4) < ts(m2)? **Yes**

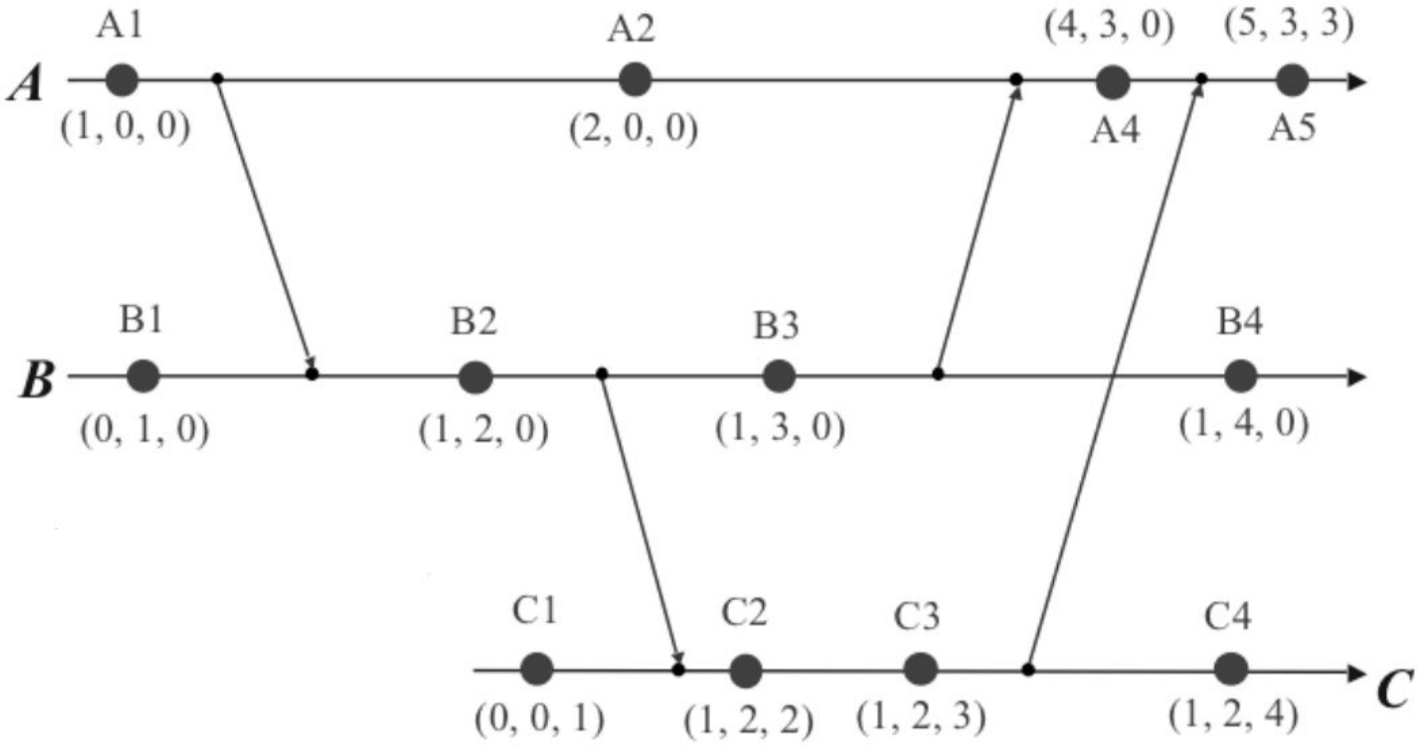
Example



Questions

ts(A5) < ts(B4)?
ts(A1) < ts(C4)?
ts(A2) < ts(B3)?
ts(B2) < ts(A5)?

Example



Questions
ts(A5) < ts(B4)? **No**
ts(A1) < ts(C4)? **Yes**
ts(A2) < ts(B3)? **No**
ts(B2) < ts(A5)? **Yes**

Recall: Totally-ordered Multicast

- All messages are delivered in the **same order** to each receiver (at application level)
- Can be implemented at middleware level using Lamport’s Logical Clocks

Causally-ordered Multicast

The idea: related messages are delivered and are processed in the same order on different nodes of the multicast group, whereas unrelated messages may be delivered in any order

We can use vector clocks, but clocks are adjusted only when

1. A message is sent or delivered to the application
2. Reception does not cause clock synch (only the delivery)

How Does it Work?

Assumptions: 1, ..., n entities

Initialization

$VC_x[j] := 0$ for all $j = 1, ..., n$

On sending message M

$VC_x[x] := VC_x[x] + 1$

sendToAll(ts(M))

On receiving message M from y

Locally enqueue **M**

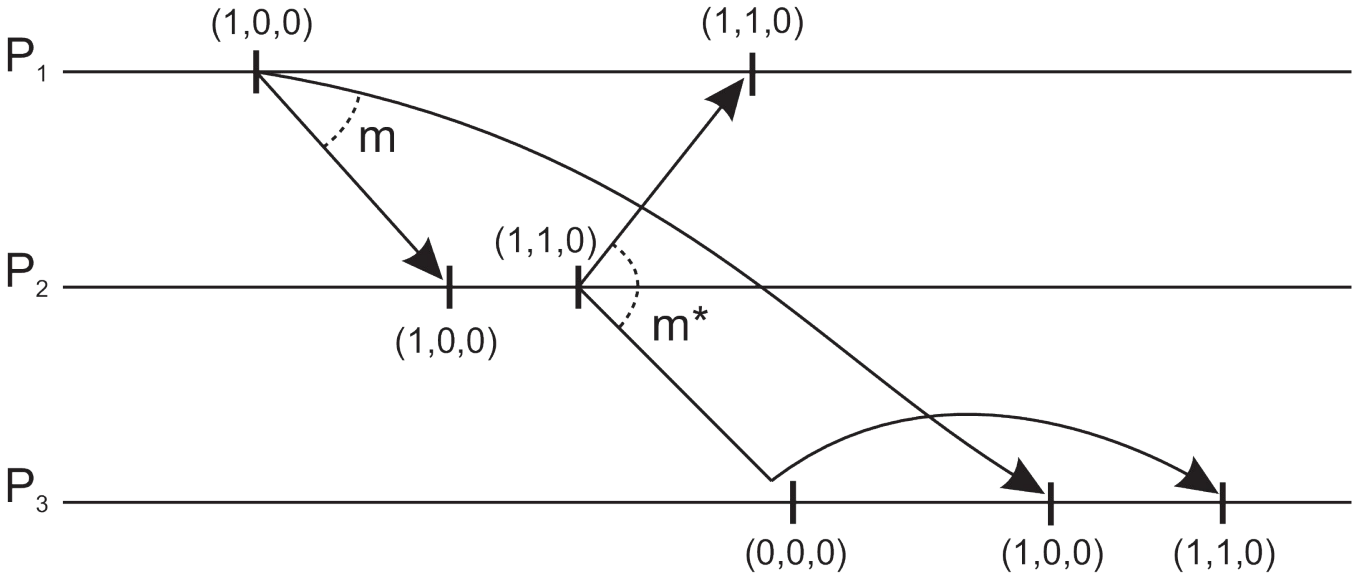
Deliver M to an application when

- $ts(M)[y] = VC_x[y] + 1$
- $ts(M)[k] <= VC_x[k]$ for $k <> y$

And then update $VC_x[k]$ for all k

$VC_x[k] = \max(VC_x[k], ts(M)[k])$

Example



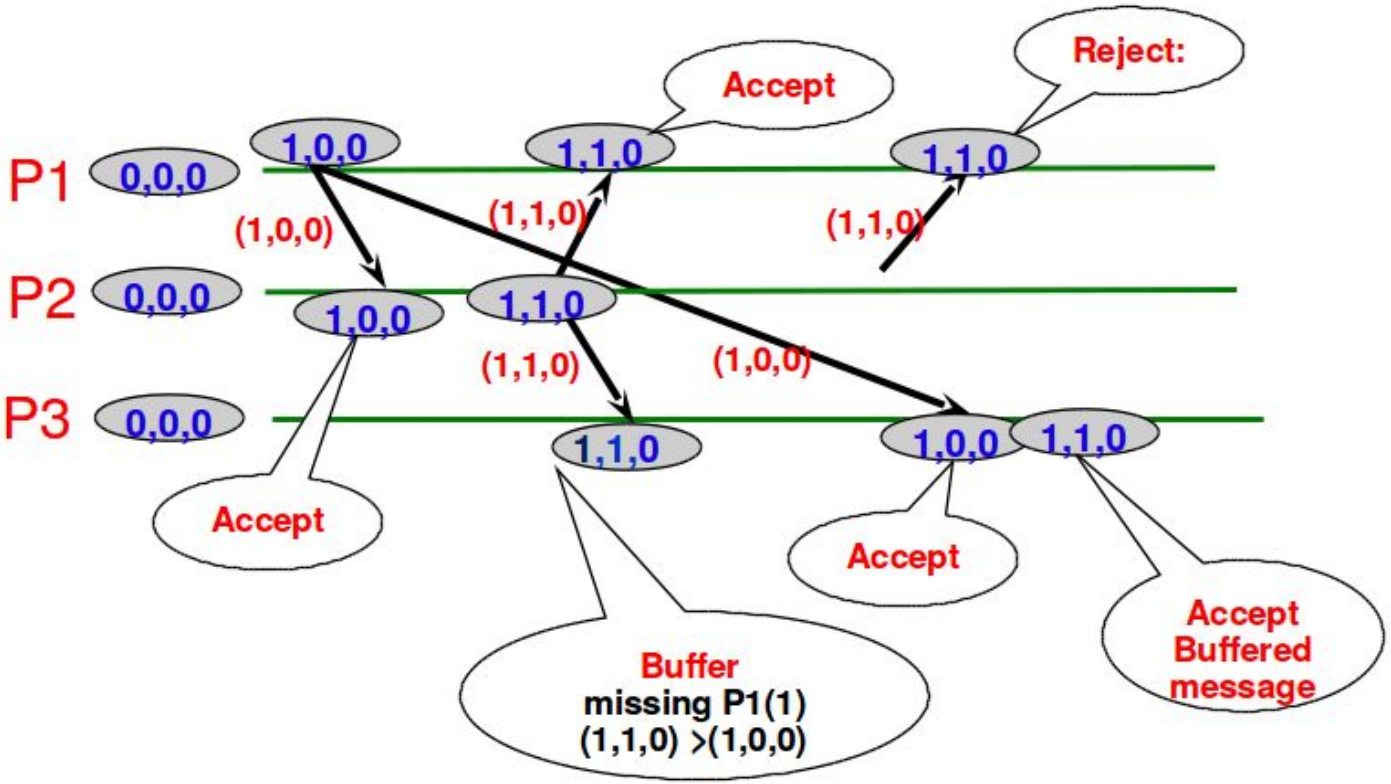
P2 receives m and delivers it to its application

P2 sends m* that is faster than m

P3 receives m* but it cannot deliver it because understands that it is missing a message from P1 (m* is enqueued)

When m arrives, P3 delivers it and then delivers m*

Example


P1
P2
P3
0,0,0
0,0,0
0,0,0
1,0,0
1,1,0
1,1,0
1,0,0
1,1,0
1,0,0
1,1,0
(1,0,0)
(1,1,0)
(1,1,0)
(1,1,0)
(1,0,0)
Accept
Reject:
Accept
Accept
Buffer
missing P1(1)
(1,1,0) >(1,0,0)
Accept
Buffered
message

Distributed Mutual Exclusion

Scenario

a system with n entities

1 shared resource

more than one entity wants to access the shared resource

What we want

only one entity at the time should be allowed to access the resource

Distributed Mutual Exclusion

A distributed mutual exclusion mechanism is a **protocol** that ensures the following properties:

1. **Mutual exclusion** ⇒ if an entity is performing a critical operation, no other entity is doing so

2. **Fairness** ⇒ if an entity wants to perform a critical operation, it will do so within finite time

Distributed Mutual Exclusion

There are two kinds of algorithms for mutual exclusion

1. **Token-based** ⇒ the token grants access to the resource

   - There is only one token that travels across the network

   - no deadlock, no starvation but token can be lost

2. **Permission-based** ⇒ a process asks permission to other processes

Algorithms

- Centralized algorithm
- Traversal-based algorithm
- Token ring
- AskAll
- Quorum-based algorithm

Centralized Algorithm

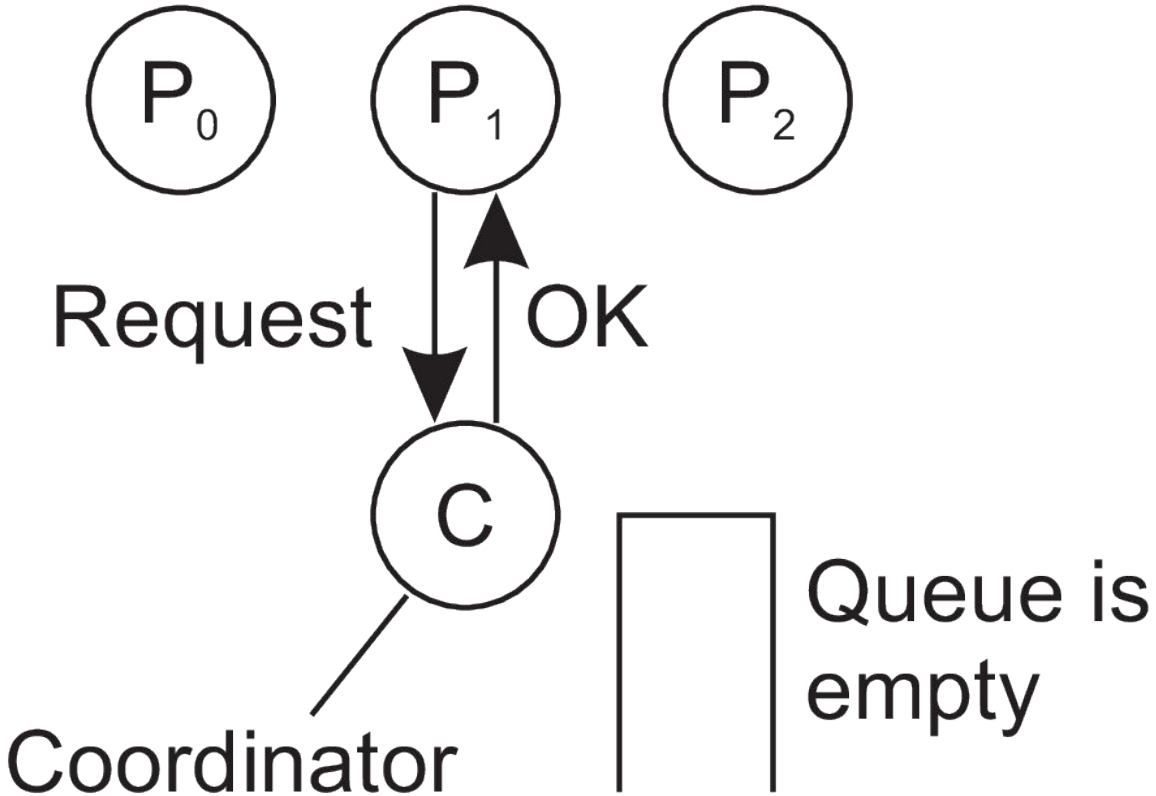
The idea: there is a leader in the system that grants permissions to access a shared resource

How does it work?

An entity **x** asks for permission to the leader

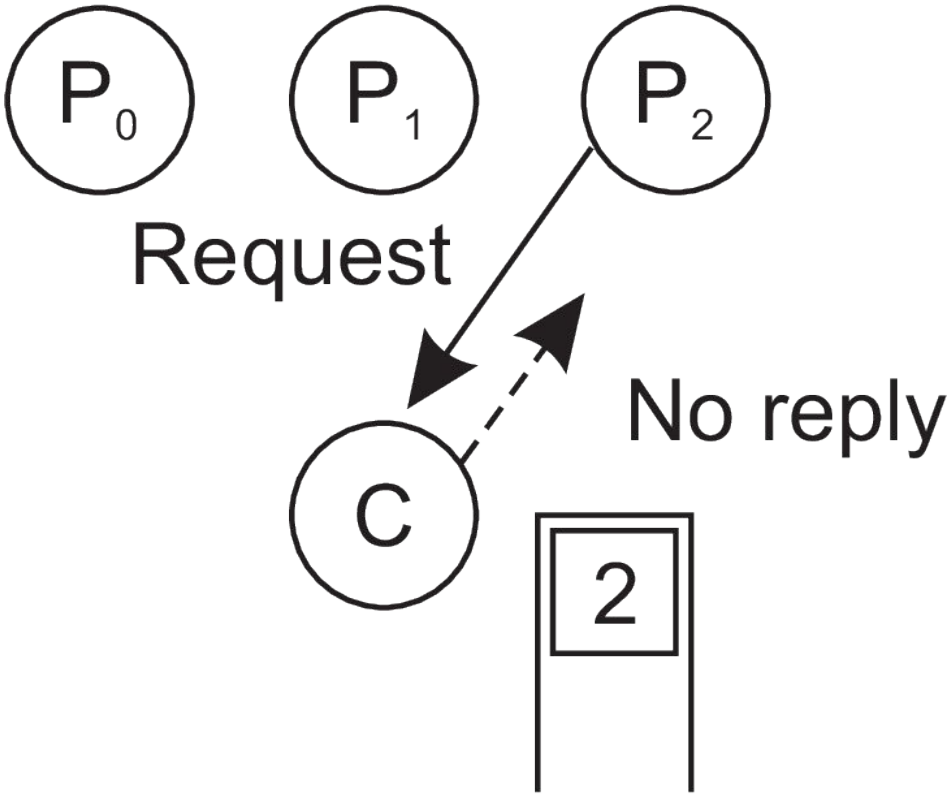
When the leader answers, **x** can access the resource

Example



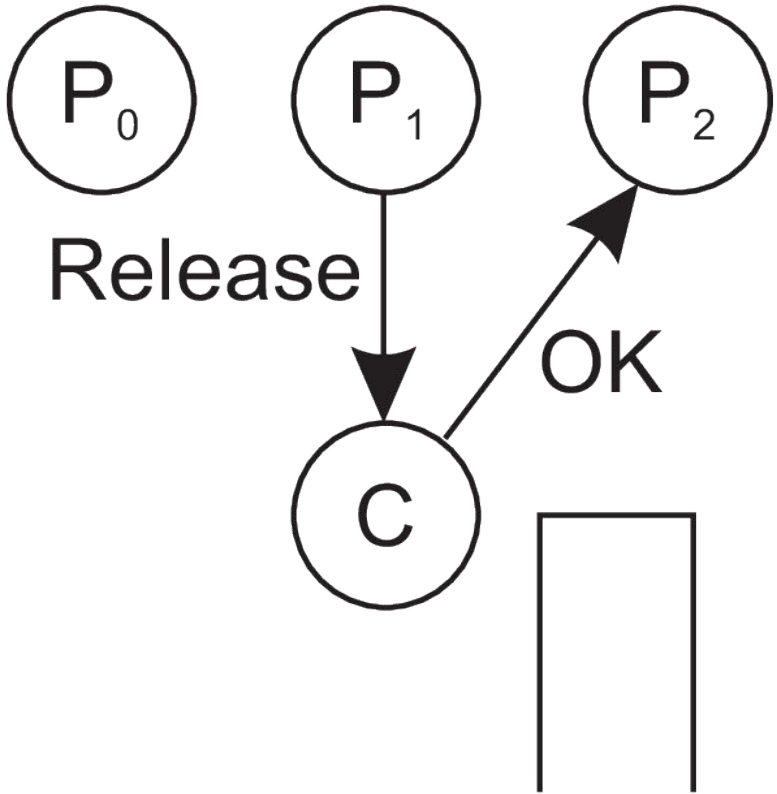
The queue maintain pending requests

Example



The queue maintain pending requests

Example



The queue maintain pending requests

Message Complexity

To access and release a resource we need

- A request from the entity **x** to the leader
- The response of the leader **r** to the request
- A message of the entity to the leader for realising the resource

n. messages: $3d(\mathbf{x}, \mathbf{r}) < 3\ \mathrm{diam}(G)$

In a complete network: 3 messages

Centralized Algorithm

Pros:

- The algorithm is correct and ensures fairness
- Few messages required to access the critical section

Cons:

- The leader is the single point of failure

Traversal-based Algorithm

The idea: there is a token traversing continuously the network; the token grants access to the resource

How does it work? An entity waits for the token, before accessing the resource

When an entity **x** receives the token

- If **x** wants access the resource, it can
- Otherwise, **x** forwards the token to another entity

How Can be Implement?

- Algorithms for network traversal, e.g., DF or DF+, DF++

  O(m) messages - m number of edges

- Using a spanning tree

  O(m) to build the spanning tree

  O(n) per traversal

Traversal-based Algorithm

Pros:

- The algorithm is correct and ensures fairness
- Starvation cannot occur

Cons:

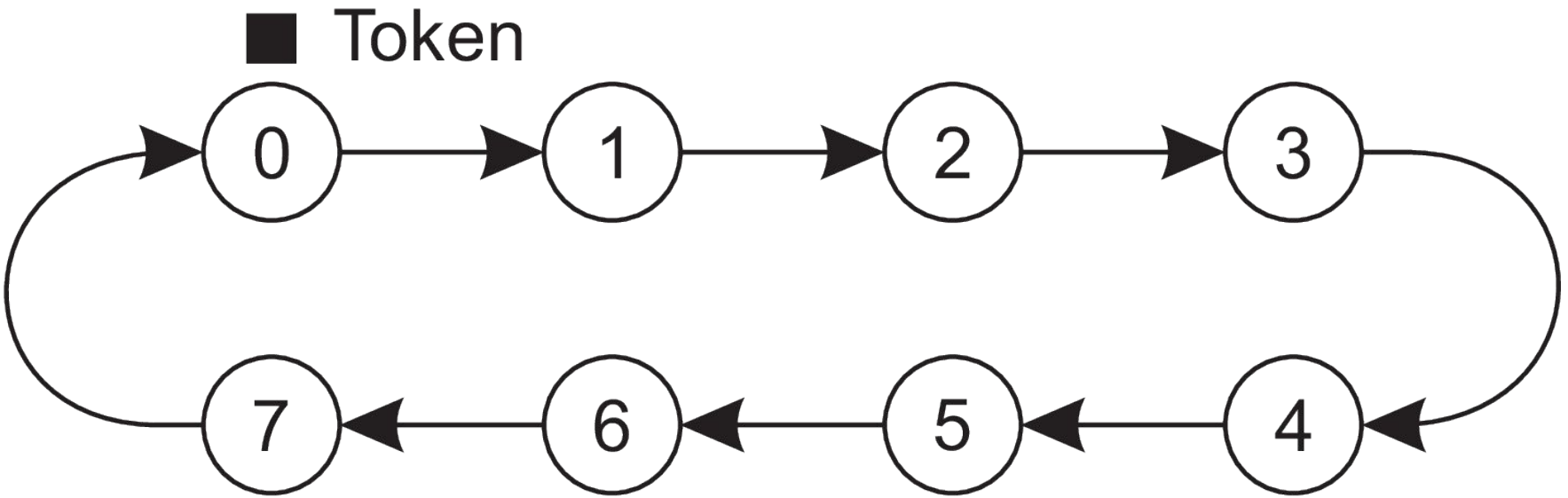
- The token can be lost

Token Ring

The idea

It is similar to the traversal-based one, except that the network topology is assumed to be a ring and the token travel across the ring

Example



AskAll Algorithm

The idea: if an entity **x** needs to access a shared resource, it asks all the other entity in the network for their permission; if all grant the permission, **x** can access the resource

Requirements: a total ordered of all events in the system ⇒ use Lamport's logical clocks

How Does It Work?

An entity **x** sends a message to all other entities containing the request and its logical clock, and waits for answers

When **y** receives a request

- If **y** isn’t using the resource and doesn’t want to, it sends back an OK message.
- If **y** is using the resource, it enqueues the request by **x**

How Does It Work?

When **y** receives a request

- Otherwise, if **y** wants to access the resource, it compares the timestamp of **x**'s message with the one of the message it sent (the lowest wins):

  If **x**'s message has a lowest one ⇒ **y** sends OK

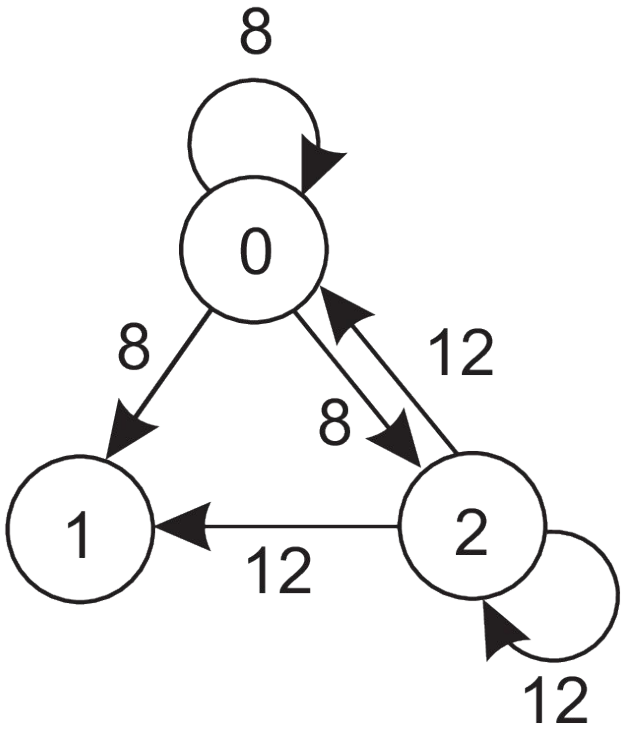
  Otherwise, **y** enqueues the **x**'s request, sending nothing

How Does It Work?

When **x** receives all other permissions, it accesses the resource

When **x** is finished, it sends OK to all entities in its queue and dequeues them

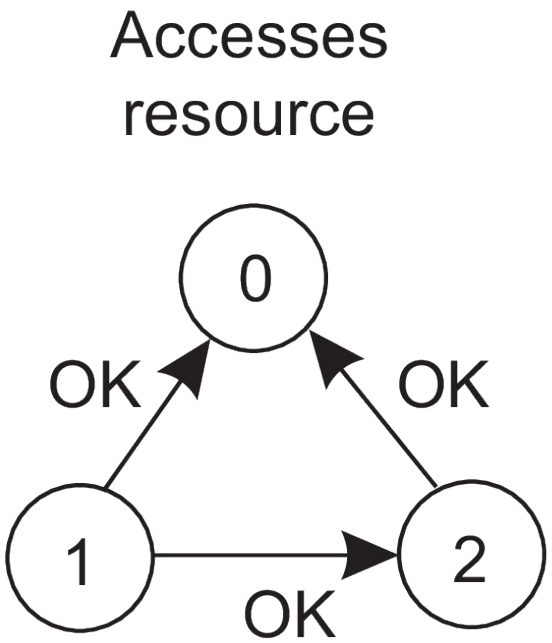
Example



0 sends m0
2 sends m2

ts(m0) = 8
ts(m2) = 12

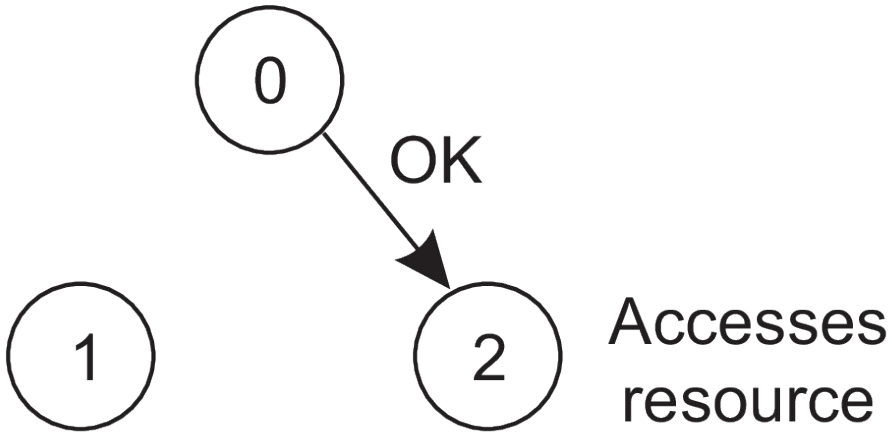
Example



0 sends m0
2 sends m2

ts(m0) = 8
ts(m2) = 12

Example



0 sends m0
2 sends m2

ts(m0) = 8
ts(m2) = 12

Message Complexity

n - 1 ⇒ request messages +

n - 1 ⇒ OK messages

---

2 (n - 1) messages are needed to grant 1 access to a resource

AskAll Algorithm Algorithm

Pros:

- The algorithm is correct and ensures fairness
- Starvation and deadlock do not occur

Cons:

- If a node fails, it blocks all the entities
- We need multicast primitives
- Getting permissions from all other nodes can be a burden for nodes with small capabilities

Quorum-based algorithm

The idea: using a voting schema, an entity can access a resource if obtains enough votes

Assumptions

- Each resource **R** is replicated n times
- Each replica has its own coordinator that regulates the access
- When an entity wants access **R**, it must get a majority of votes from **m > n / 2** coordinators

Message Complexity

m: the number of coordinators for a resource **R**

An entity **x** needs to send **m** request messages

To have access **x** receives at most **m** response messages

The number of message is at most **2 m**

Variant: when an entity **x** is not granted access, it waits for a random time and retries ⇒

at most **2 m k** messages if **x** tries **k** times

Conclusions

- Time synchronization
- Logical clocks
- Distributed Mutual exclusion

References

- Chapter 6 of “Distributed Systems” by M. van Steen and A. S. Tanenbaum
- Chapter 9 of “Design and Analysis of Distributed Algorithms” by N. Santoro