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

## *Leslie Lamport*

ANDRÉ ALLAVENA

Cornell University

# Motivations

- Time is part of our world
- But time is difficult for computers
- It cannot be send via messages
- Time is closely linked to “before” and “after”, but there are a lot of issues with those words

# Distributed System

A system is distributed if the message transmission delay is not negligible (eg: spatially separated, multiprocessor, etc.)

$\rightarrow$ Sometime, we cannot say which event happened first, if any. This rises a lot of difficulties, for synchronisation among others.

Also, one must be careful that algorithms to create total ordering of events might not always give the same ordering as the one perceived by the user.

# The Partial Ordering

Event $a$ happened before $b$ if $a$ happened at an earlier time than $b$. But

- Must have physical clocks!
- What if clocks not perfectly accurate?

Let's define *happened before* without clocks.

# Definitions

The system is constituted of a collection of processes

**process:** a sequence of event

**event:** Subprogram, single machine instruction, etc.

Events form a sequence in a process (total ordering)

**receiving a message:** is an event in a process

**sending a message:** is another event

# $a$ happened before $b$

$a \rightarrow b$ ($a$ happened before $b$) iff

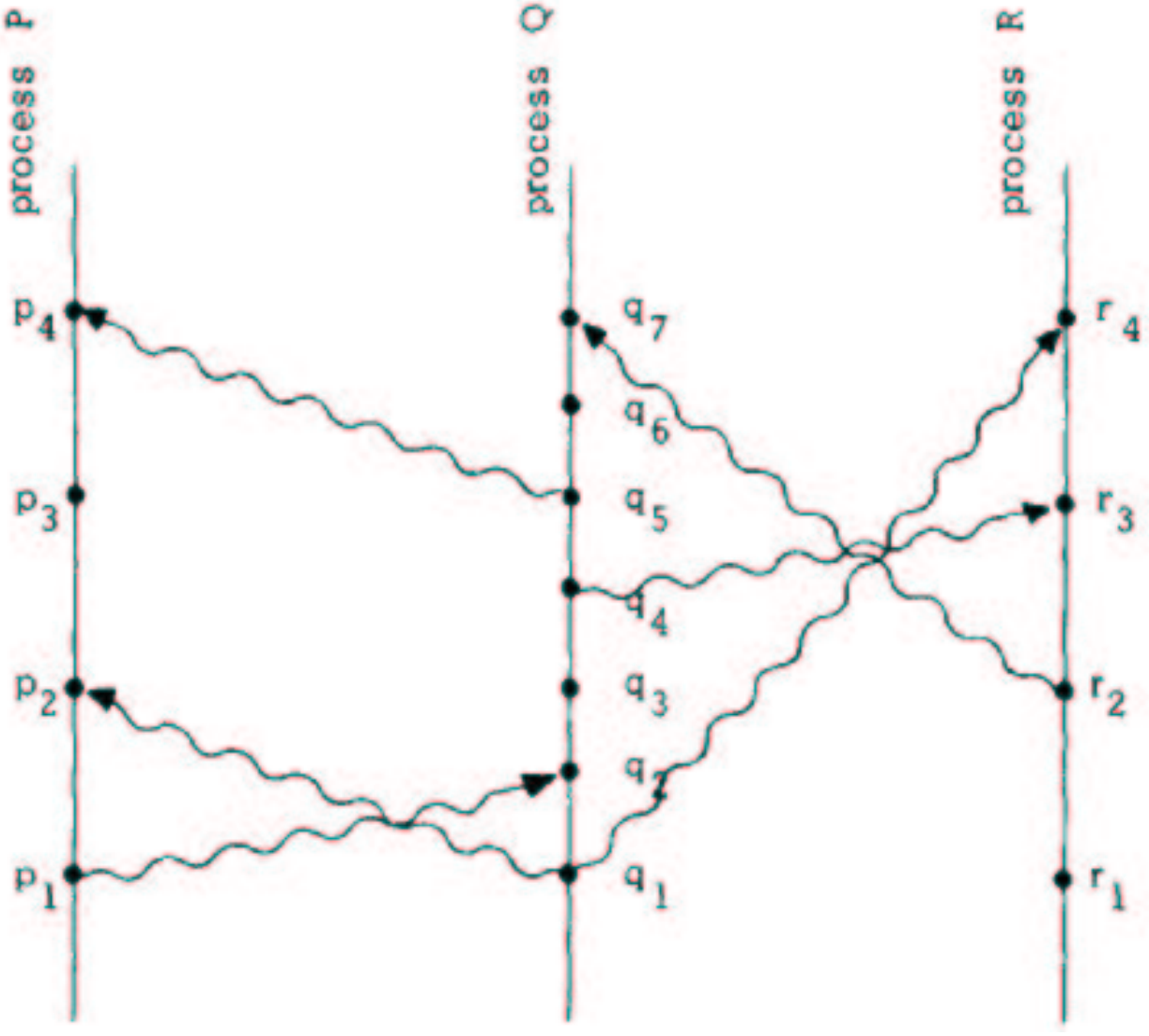
- $a$ and $b$ are events in the same process and $a$ comes before $b$
- $a$ is the sending of message $m$ and $b$ is the receiving of that message $m$
- $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

**concurrent:** if $a \not\rightarrow b$ and $b \not\rightarrow a$ then $a$ and $b$ are concurrent

We assume $a \not\rightarrow a$

This is a irreflexive partial ordering on the set of events

# Space Time Diagram



# Other Meaning

$a \not\rightarrow b$ means that it is possible for event $a$ to causally affect event $b$

Events are concurrent if neither can causally affect the other. Only the messages which *have* been sent are considered.

The theory of relativity (and space-time invariant) actually consider the messages that *could* have been sent as well. This is more powerful, but requires know delivery time on the messages. We shall regain this power (good total ordering) later.

# Logical Clocks

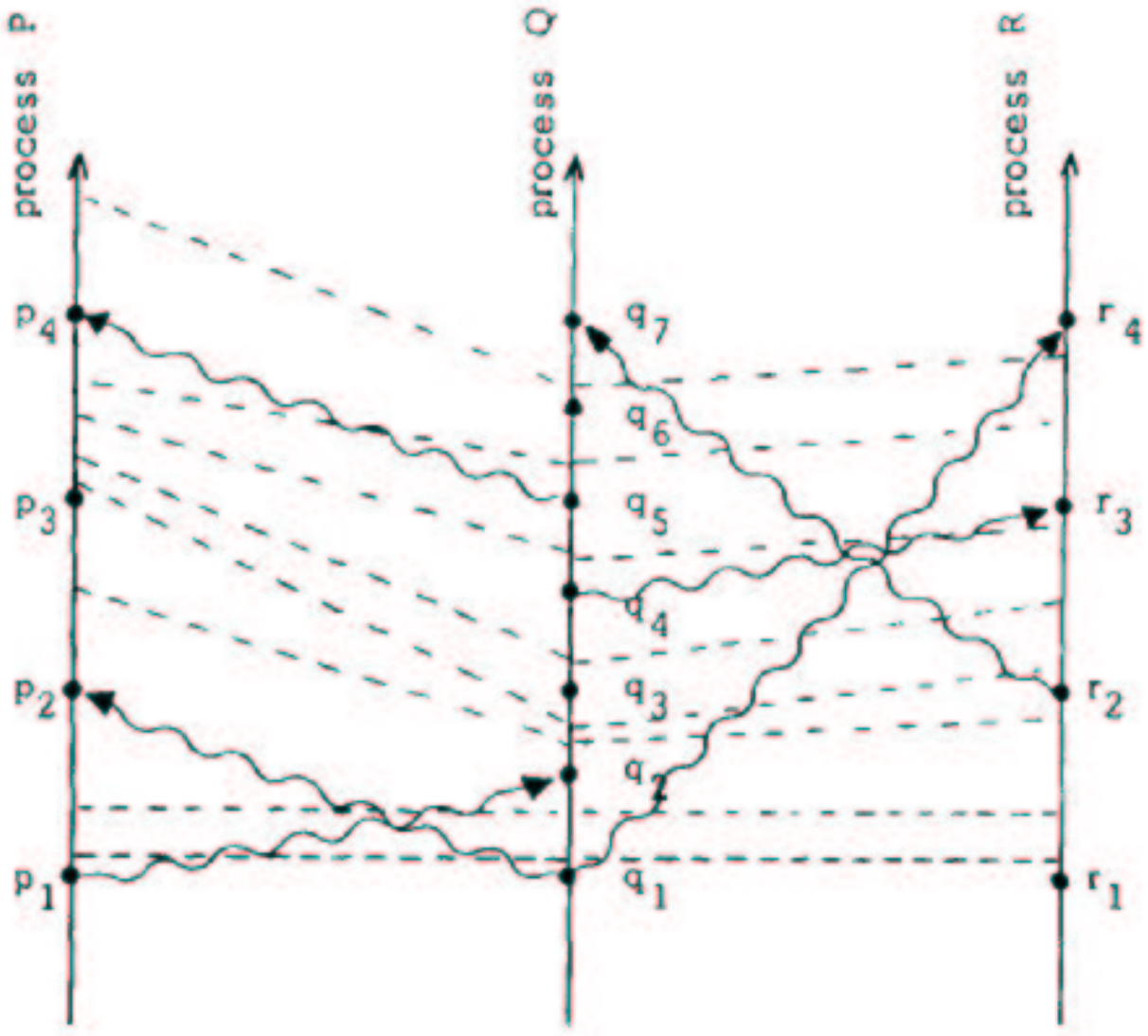
This is just a way of assigning a number to an event

- A clock $C_i$ for process $P_i$ is a function from events to (non-negative) integers.
- $C\langle a \rangle$ is defined to be $C_i\langle a \rangle$ when $a$ occurs in $P_i$
- No relation between clocks and physical time
- Think of $C$ as for Counter

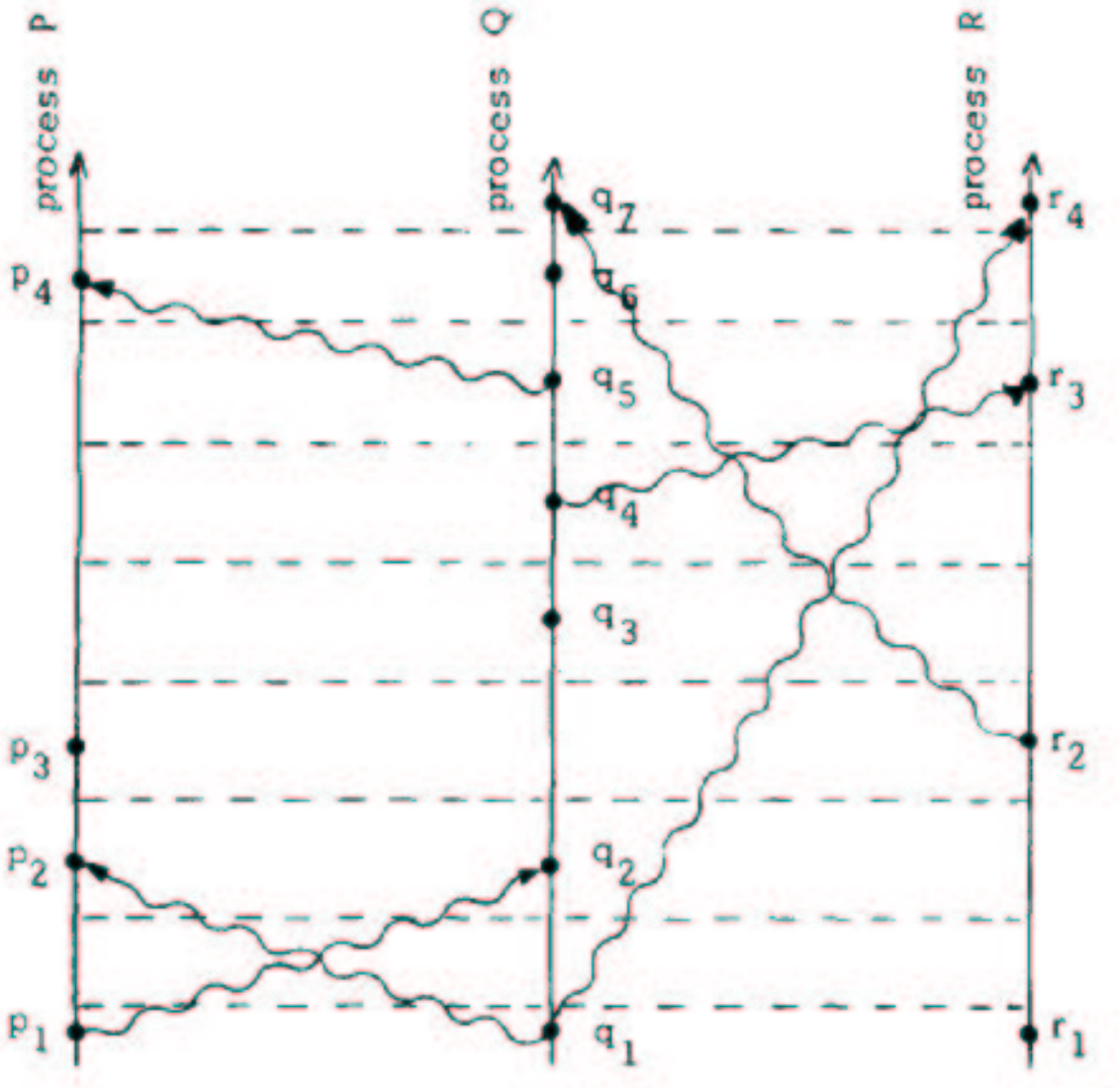
# Clock Condition

- For any event $a$, $b$, if $a \rightarrow b$ then $C\langle a \rangle < C\langle b \rangle$
  The reverse doesn't hold.
- The Clock Condition is satisfied if
  - If $a$ and $b$ events in $P_i$ then $C_i\langle a \rangle < C_i\langle b \rangle$
  - If $a$ is the sending of a message by $P_i$ and $b$ is the receiving of that message by $P_j$, then $C_i\langle a \rangle < C_j\langle b \rangle$
- A clock ticks through every number,
- The clicks happen between the events

# Space Time Diagram



# Space Time Diagram, revisited



# Implementation

Consider the events to be steps in algorithms

- Each process $P_i$ increments $C_i$ between any two consecutive events
- Each message $m$ contains a time-stamp $T_m$ containing the current time of the sender
- If event $a$ is the sending of message $m$ by $P_i$, then $T_m = C_i\langle a \rangle$
- When $P_j$ receives $m$, it increases its clock to be strictly greater than the time-stamp in $m$
- $C_j\langle b \rangle := \max(T_m + 1, C_j\langle b \rangle)$

# Total ordering of the events

All the events have a time, so let's order the events by the time they occur. Break ties by choosing an arbitrary order on the processes.

$a$ is before $b$ ($a \Rightarrow b$) iff

1. $C_i\langle a\rangle < C_i\langle b\rangle$
2. $C_i\langle a\rangle = C_i\langle b\rangle$ and $P_i \prec P_j$

This is an *arbitrary* ordering, and is far from being unique (depends on the choice of the $C_i$).

# Ex: Distributed Mutual Exclusion

Some processes are sharing a single resource.

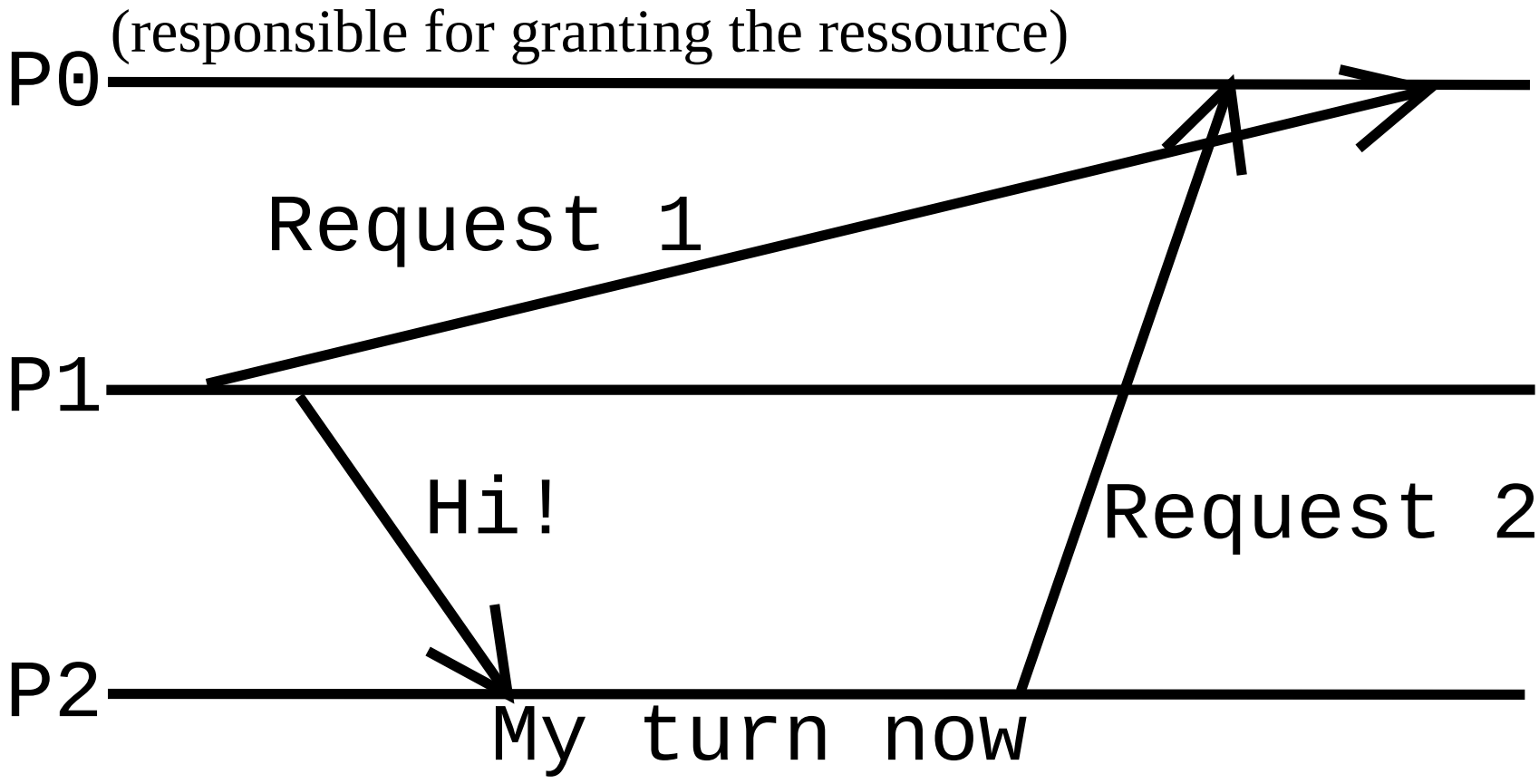
Only one process at a time can use the resource

**exclusion:** A process which has been granted the resource must release it before another one can grab it

**no lock:** If every process which is granted the resource eventually release it, then every request is eventually granted

**ordering:** Different requests must be serviced in the order they are made

# A central scheduling system?




A central scheduling system doesn't work!

# Solution: Idea

FIFO, but

- Distributed
- Broadcast the request
- Force a synchronisation of the clocks
- Broadcast the release

# Distributed Algorithm

Assumptions

- messages sent from $i$ to $j$ are received in the order they were sent
- all messages are eventually received
  - This can be enforced by acks
- each process has its own request queue, initialised to $T_0 : P_0$
- $P_0$ has the resource at the beginning
- $T_0$ is smaller than the initial value of the clocks

# Solution: Algorithm

- To request, $P_i$ broadcasts $T_m : P_i$ (aka, my current time is $T_m$, I am $P_i$), and put it in its own queue
- When $P_j$ receives the request, it acknowledges it and puts it in its queue
- To release, $P_i$ broadcasts a release (time-stamped?) message, and erases the request from its queue
- When $P_j$ receives the release message, it erases the request from its queue

# Algorithm, continued

$P_i$ is granted the resource when

- There is a $T_m : P_i$ at the top of the request queue (ordering is $\Rightarrow$ on request messages)
- $P_i$ has received a message from every other process, time-stamped later than $T_m$

This guarantees exclusion, no locking and ordering.

This is a distributed algorithm, each process follows the rules on it own.

# Generalisation

The previous method can be used to implement any kind of synchronisation for a distributed system.
Synchronisation is specified in terms of a State Machine (wait for lecture 2/21)

- Set of possible commands (request and release in the previous example),
- Set of possible sates
- Transition function

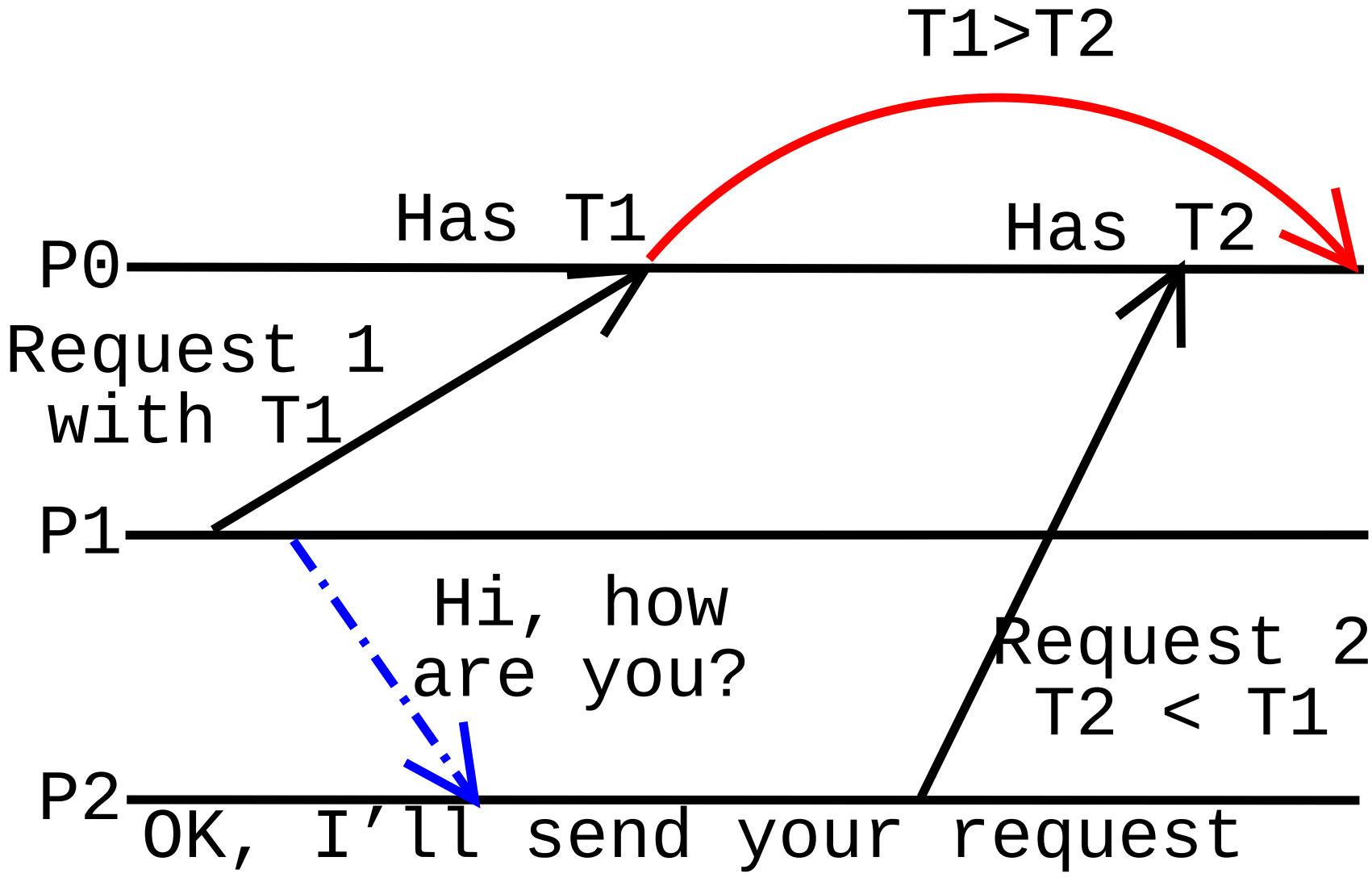
# Generalisation continued

Each process independently simulates the State Machine, using the broadcasted commands.

There is synchronisation because of the time-stamps on all the messages and the wait to be sure everybody has a later time than the request.

Note that this requires all processes to be alive and part of the algorithm...

# Anomalous Behaviour



# Anomalous Behaviour Continued

- Let $\mathscr{S}$ be the set of system events
- Let $\underline{\mathscr{S}}$ be the set of all events
- Let $\underline{\rightarrow}$ be the *happened before* in $\underline{\mathscr{S}}$

To avoid anomalous behaviour, one can either

- ask the user to input a correct time-stamp
- have stronger clock guarantees

# Strong Clock Condition

- For any events $a$, $b$ in $\mathscr{S}$:
  if $a \underline{\rightarrow} b$ then $C\langle a \rangle < C\langle b \rangle$
- This is stronger than before when we only had $a \rightarrow b$ ie $a$ happened before $b$ in $\mathscr{S}$ (now $\underline{\mathscr{S}}$)

Let $\underline{\mathscr{S}}$ be the set of events in the real world

One can construct physical clocks, running quite independently, and having the Strong Clock Condition, therefore eliminating anomalous behaviour.

# Physical Clocks

Assumptions:

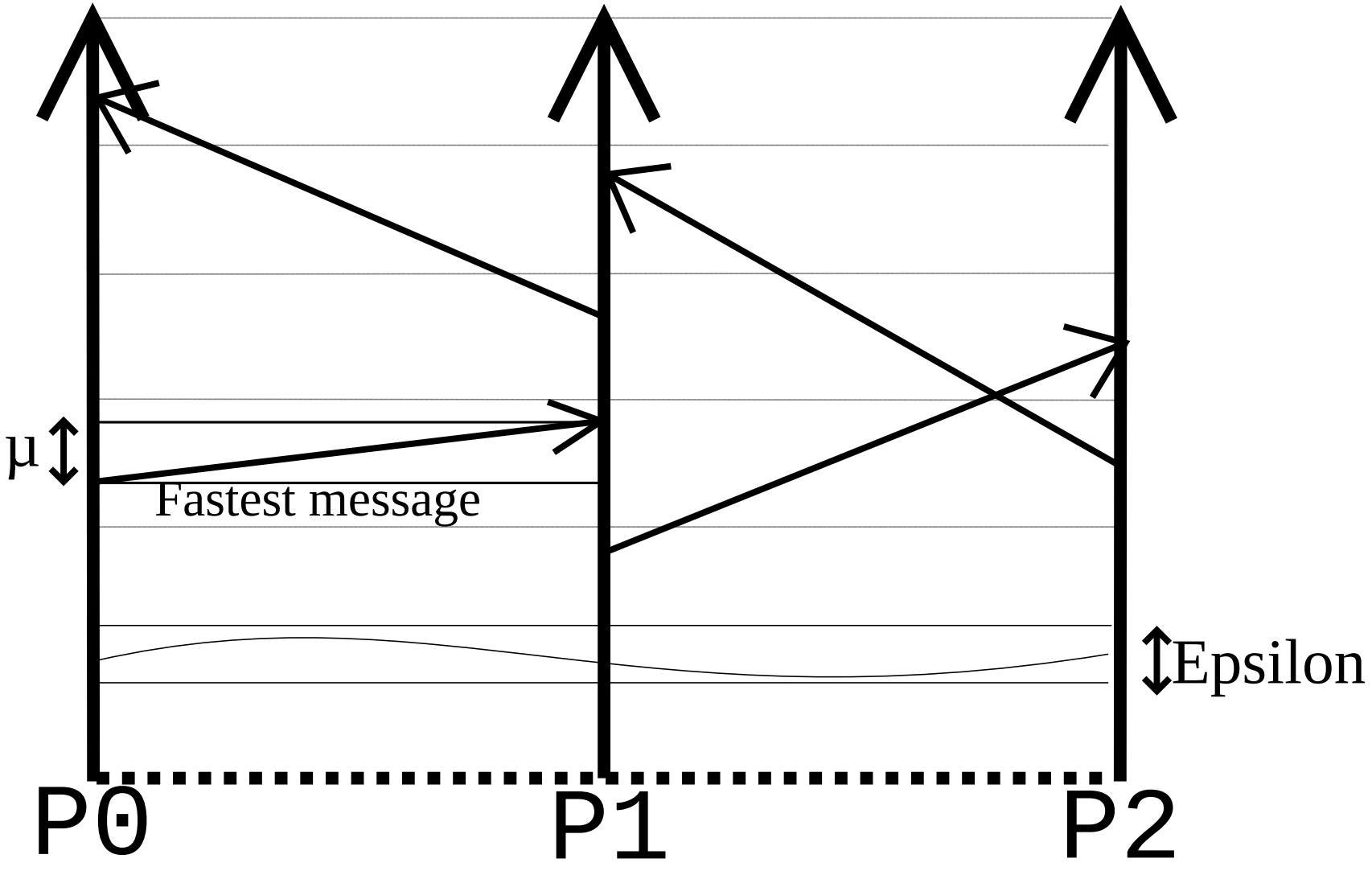
- They run continuously
- approximatively at speed 1:
  $\forall i \quad \exists \kappa \quad$ such that $|\frac{dC_i(t)}{dt} - 1| < \kappa$
- and are approximatively synchronised
  $\forall i, j : |C_i(t) - C_j(t)| < \varepsilon$
- and verify the ordinary Clock Condition

Since they tend to drift away, we need to re-synchronise them.

# Physical Clocks

- We have to insure that the system $\underline{\mathscr{S}}$ of relevant physical events satisfies our Strong Clock Condition
- For that we only need consider events $a$ and $b$ from $\underline{\mathscr{S}}$ where $a \not\rightarrow b$. So $a$ and $b$ occur in 2 different processes
- Let $\mu$ be a number such that if $a \underline{\rightarrow} b$ and $a$ occurs at $t$ in $P_i$ and $b$ in $P_{j \neq i}$ then $b$ occurs later than $t + \mu$.

# Physical Clocks



# Physical Clocks

- In other words $\mu$ is less than the shortest transmission time.
- Or else the time granularity of the physical world is larger than $\mu$.

To avoid anomalous behaviour we need $C_i(t+\mu) - C_j(t) > 0$, guaranteed if $\frac{\varepsilon}{1-\kappa} \leq \mu$

# Algorithm

$\mu_m$ is the minimum delay of a message, and is known by the processes (it is really needed?)

- Only move clocks forward, and that only upon reception of a message
- When a process receives a message with time-stamp $T_m$, it updates its clock via $C_i(t) = \max C_i(t), T_m + \mu_m$
- This is the same as before, you move the clock forward, and know the current time is at least the time-stamp + the minimum delay of receiving a message.

# Main Result

If there are enough messages with a sufficiently small delivery delay, then there are good bounds on the de-synchronisation of the clocks.

- $\forall m \quad \mu > \mu_m$ an upper bound on the minimum delay
- $\exists \tau, \xi$ so that every $\tau$ second, a message with an unpredictable delay less than $\xi$ is sent over every arc
- then $\varepsilon \approx d(2\kappa\tau + \xi)$ assuming $\mu + \xi \ll \tau$

# Conclusion

- “Happening before” only defines a partial ordering
- This partial ordering can be made total, but in several arbitrary manner, and can lead to anomalous behaviour because it is not conform to the users perception
- But the use of synchronised clocks can overcome this problem