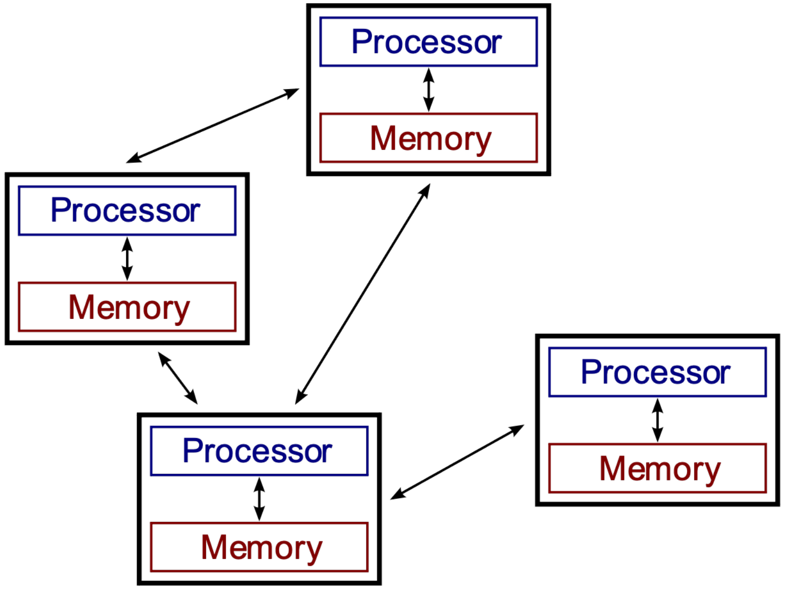
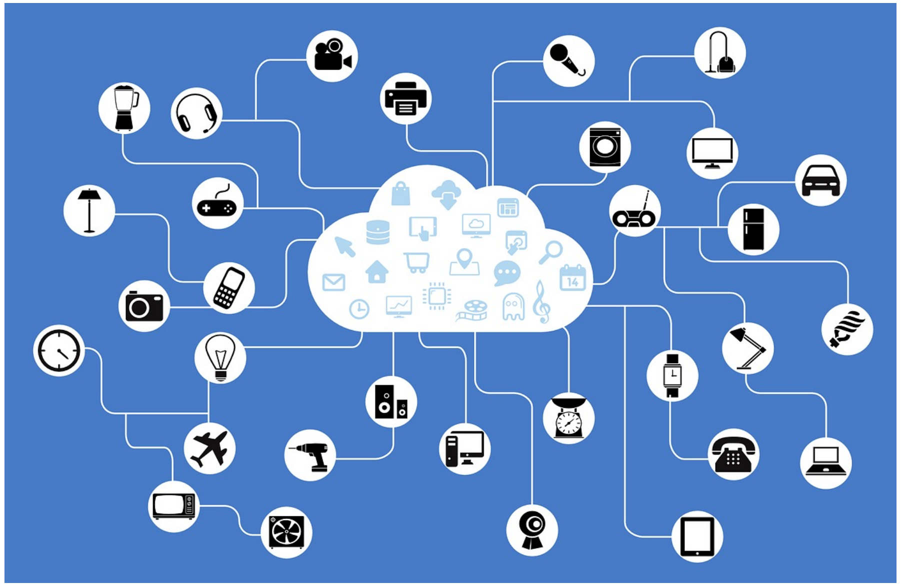
Embedded Operating System Distributed synchronization

Graziano Pravadelli

Distributed systems





Properties of distributed systems

No common clock, no shared memory

Information distributed among many nodes

Processes make decisions based on local information

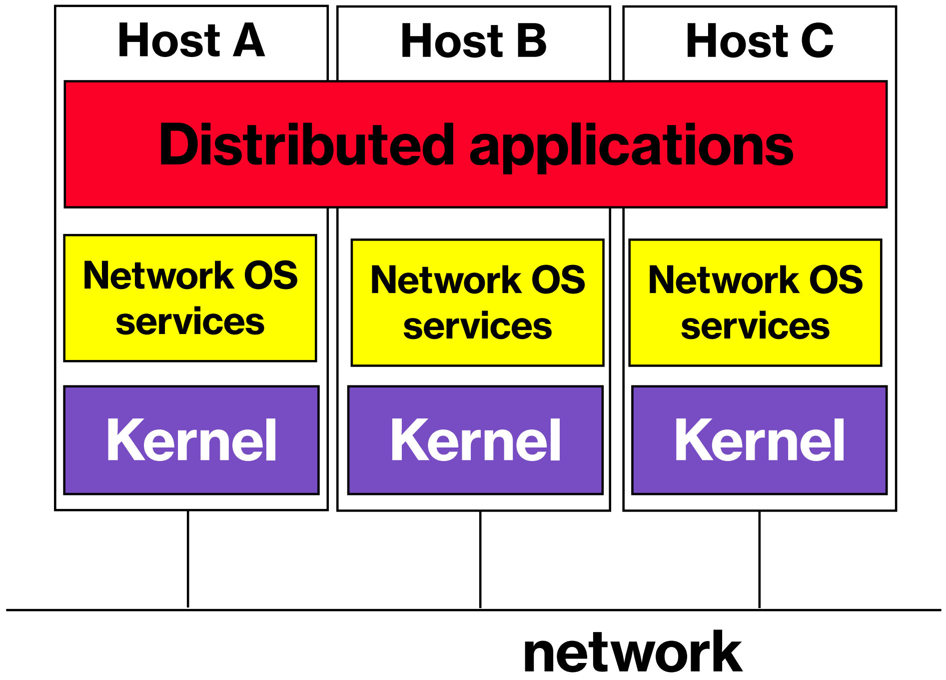
Classification

		HW	
		Loosely coupled	Tightly coupled
SW	Loosely coupled	Network O.S. (NOS)	–
	Tightly coupled	Distributed O.S. (DOS)	Multiprocessor O.S. (MOS)

NOS architecture

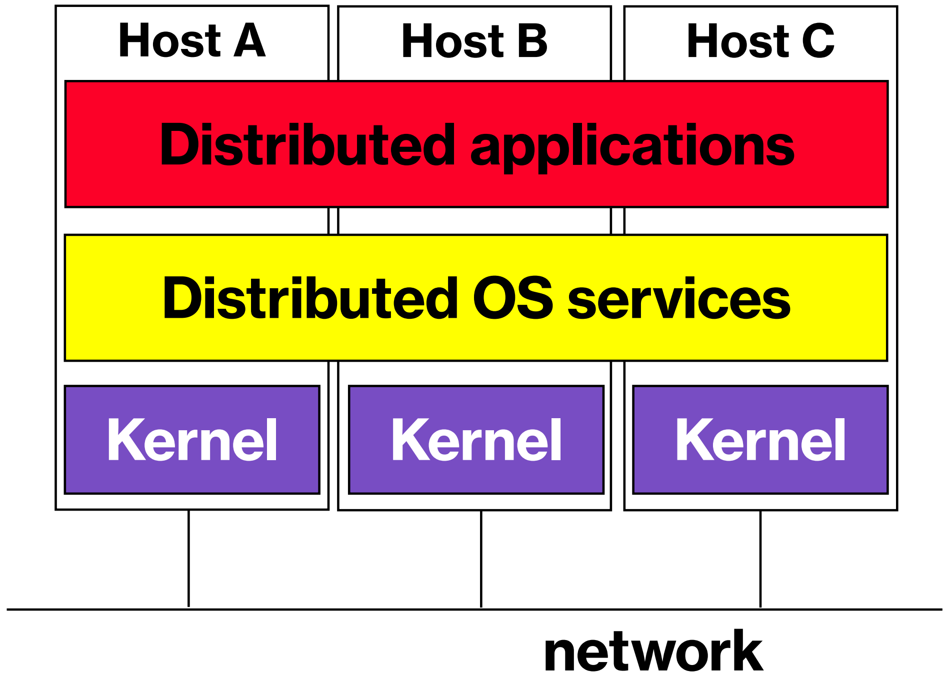
Necessary ad-hoc commands for using the distributed resources in the network

Lack of transparency



DOS architecture

- Shared memory via SW
- Task assignment to processors
- Interprocess communication
- HW faults hiding

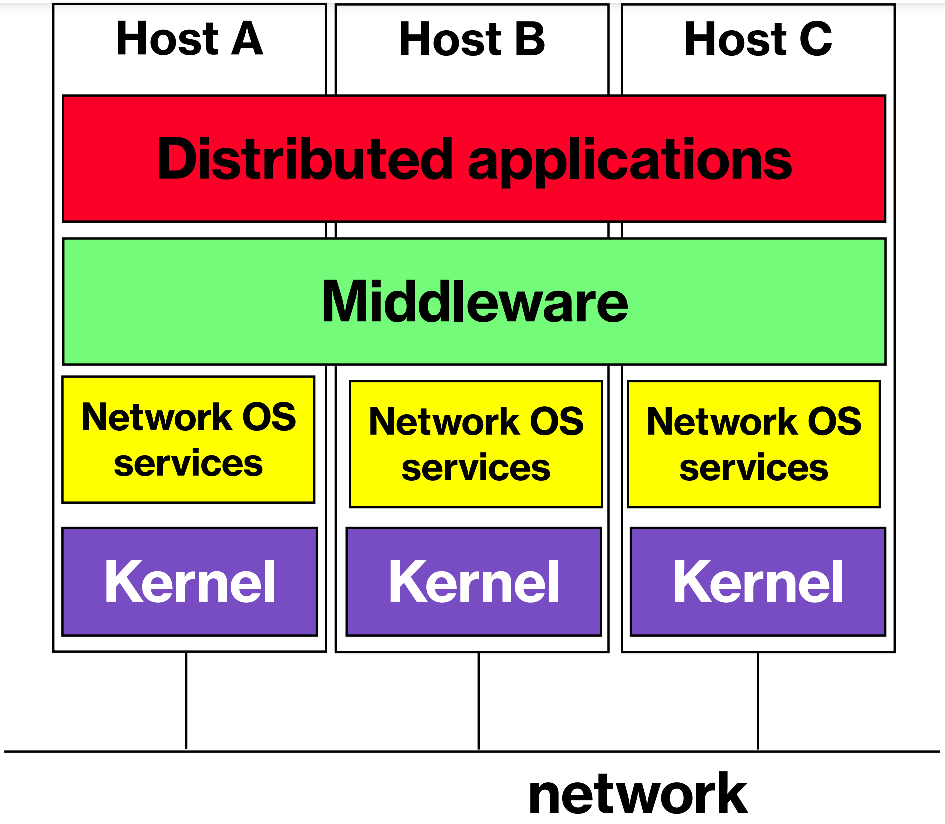


Middleware

DOS → transparency but not much scalability, no heterogeneity

NOS → no transparency but scalability and heterogeneity

- SW layer that merges DOS and NOS benefits
- High-level communication primitives (ex.: RPC)
- Distributed file system
- Naming



Summary

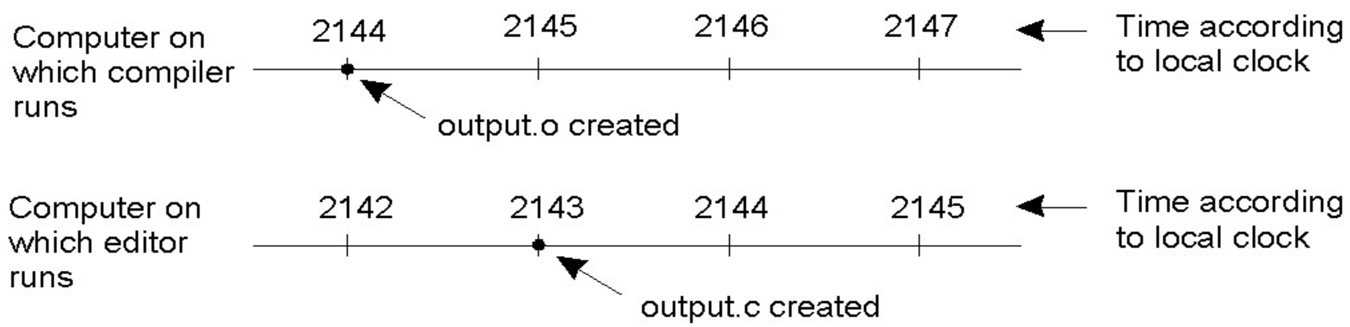
- Clock synchronization
- Mutual exclusion
- Deadlock

- Techniques used for traditional O.S. are not suitable as they suppose the presence of a shared memory

Clock synchronization

Clocks synchronization – Motivation

- When each host has its own clock, the order of events can be inverted
- Example: Iteration editing/compilation + make



Goal

A distributed system is a set of processors $P = \{P_1, P_2, \dots, P_n\}$

- P_i reacts to events
  - External events: sending, receiving messages
  - Internal events: local I/O, signals arrival, …

E = set of possible events in the system

E_{P_i} = set of possible events in P_i

Goal: to define an order for E

Clocks synchronization

Can all the clocks of a system be synchronized? No, in principle		
Global time concept required, but not achievable	No common clock	Communication delay (non-deterministic) between nodes

A "relaxed" synchronization version is possible	
Based on the concept of *logical time*	i.e., based on relative time (not absolute)

Logical (virtual) clocks

IDEA: the order of events is more important than the exact time in which events occur

- Synchronization of logical clocks is simpler
- Not necessarily linked to actual time

System model

- Execution of processes = sequence of events
- Granularity = event (instruction, procedure call, sending a message …)

Relation "$\rightarrow$" (happened-before)

Definition

- If A and B are events in the same processor, and A was executed before B, then $A \rightarrow B$

The relation captures causal dependencies between events

- Event A has a causal effect on B if $A \rightarrow B$

Meaning of “→”

- In the same processor, events are totally ordered
- Among different processors, if
  - A is the event corresponding to sending a message from a processor, and
  - B is the event corresponding to receiving the message from another processor
- then
  - $A \rightarrow B$

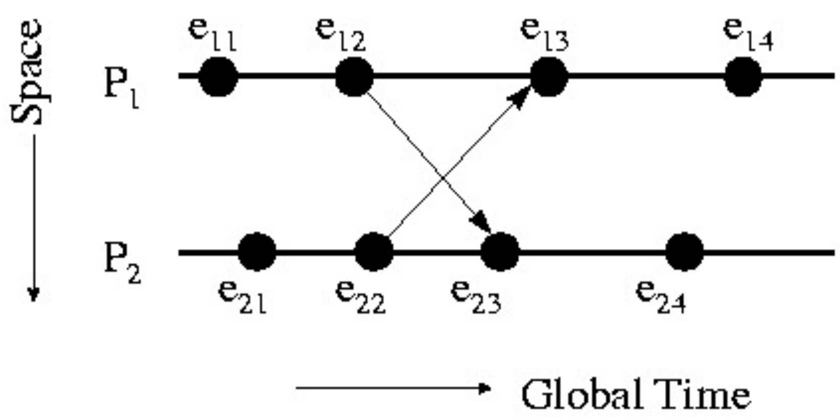
Properties of "$\rightarrow$"

- It is an anti-reflexive partial order relation
  - Anti-reflexive
    - $\neg$ (A $\rightarrow$ A)
  - Asymmetric
    - If A $\rightarrow$ B then $\neg$ (B $\rightarrow$ A)
  - Transitive
    - If A $\rightarrow$ B and B $\rightarrow$ C and ... and V $\rightarrow$ Z then A $\rightarrow$ Z

Characteristics of "$\rightarrow$"

- Concurrent events are allowed
  - $A \parallel B$ if $\neg (A \rightarrow B)$ and $\neg (B \rightarrow A)$
  - When two processes on different processors do not exchange messages
- For each event pair A and B in a system
  - $A \parallel B$ or $A \rightarrow B$ or $B \rightarrow A$

Example of "→"



- $e_{12} \rightarrow e_{23}$
- $e_{22} \rightarrow e_{13}$
- $e_{23} \rightarrow e_{24}$
- $e_{12} \rightarrow e_{24}$
- $e_{11} \parallel e_{22}$

Implementation of “$\rightarrow$”

Goal

- for each event *a*, defining a time value C(*a*) on which each processor agrees

Solutions

- System of logical clocks [Lamport 78]
- Vector clock [Fidge 91 / Mattern 88 / Raynal&Singhal 96]

Lamport's logical clocks

Each processor P_i has a clock C_i

C_i assigns values $C_i(a)$ (*timestamp*) for each event *a* of P_i

Values of C_i

No relation with actual time	Monotonically increasing

Implementation by using local counters

Lamport's logical clocks – How?

- Condition for a logical clock system to be "correct":
  - If $a \rightarrow b$ then $C(a) < C(b)$ (regardless of where *a* and *b* are located)

- To be realized:
  - Conditions of correctness C1, C2
  - Implementation rules IR1, IR2

Conditions of correctness

[C1]

- For each pair of events *a* and *b* in P_i, if *a* happens before *b* then $C_i(a) < C_i(b)$

[C2]

- If *a* corresponds to sending a message *m* by processor P_i, and *b* corresponds to receiving the message *m* at processor P_j, then $C_i(a) < C_j(b)$

Implementation rules

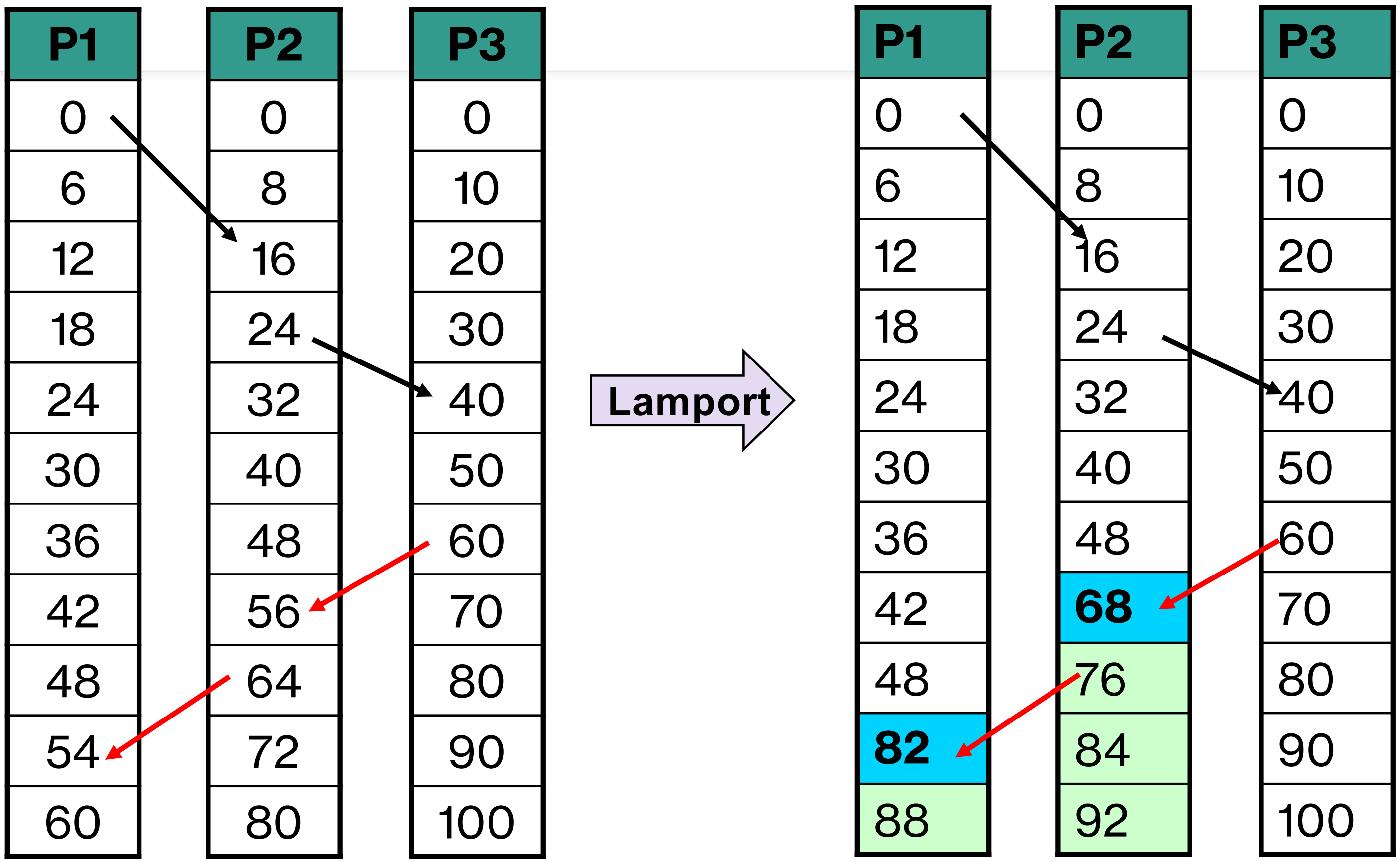
[IR1]

- The clock C_i is incremented between two successive events in the processor P_i such that $C_i = C_i + d$ ($d > 0$)

[IR2]

- If *a* corresponds to sending a message *m* by P_i, then *m* is associated with a timestamp $t_m = C_i(a)$
- When *m* is received by P_j, $C_j = \max(C_j, t_m) + d$

Example



P1
0
6
12
18
24
30
36
42
48
54
60
P2
0
8
16
24
32
40
48
56
64
72
80
P3
0
10
20
30
40
50
60
70
80
90
100
Lamport
P1
0
6
12
18
24
30
36
42
48
82
88
P2
0
8
16
24
32
40
48
68
76
84
92
P3
0
10
20
30
40
50
60
70
80
90
100

Can we obtain a total order?

Total order used in many synchronization algorithms

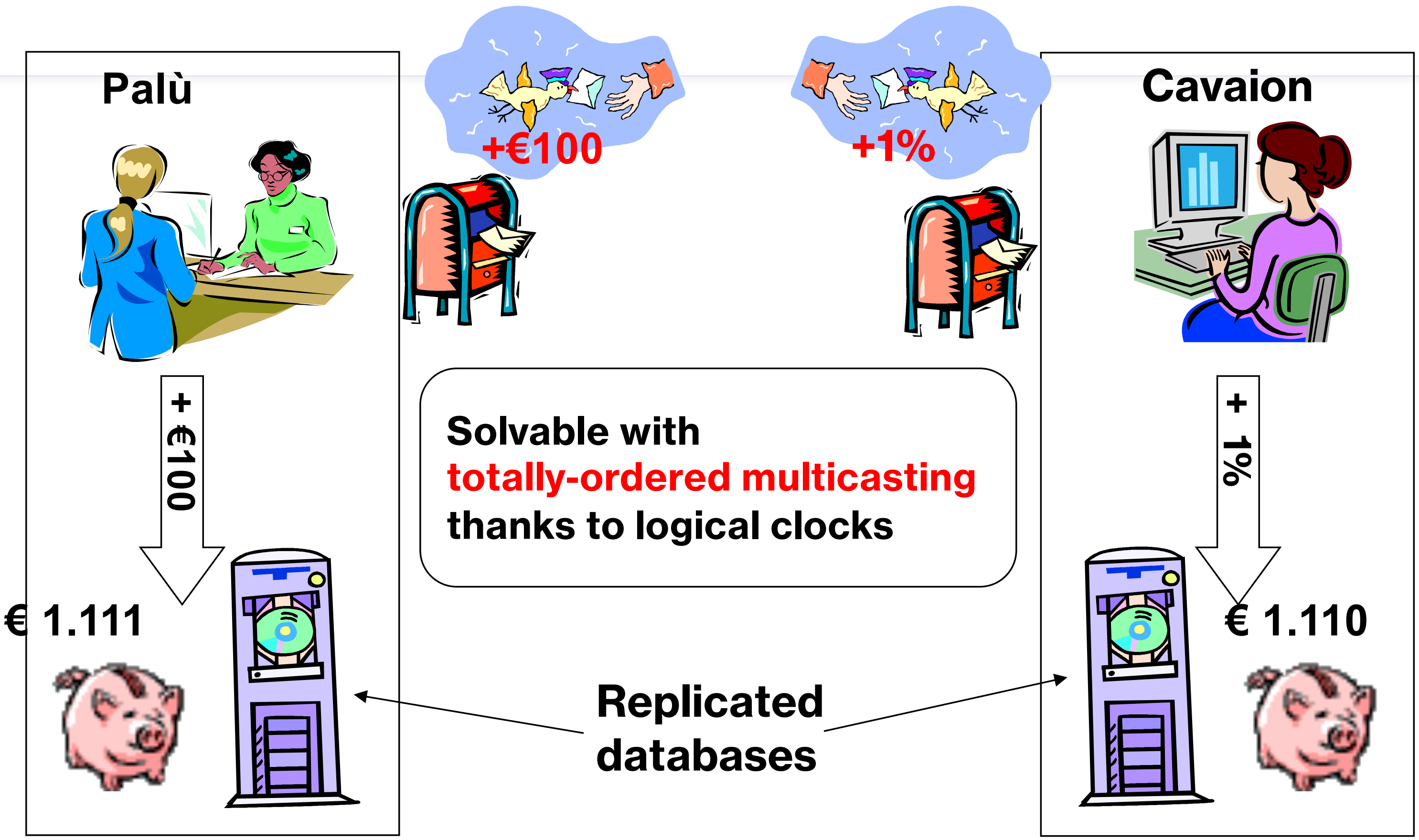
$\rightarrow$ is a partial order

To get total order, add the ID of the corresponding processor to the timestamp of an event

Example

If A happens at time 10 on P_1, and B at time 10 on P_2, $A \rightarrow B$ because $P_1 < P_2$

Synchronization problem



Totally-ordered multicasting

- Each process sends messages to other processes in the group (even to itself)
- Each message *m* has a timestamp t_m corresponding to the sender's logical clock
- Messages arrive in order (as sent) and there is no loss
  - Condition obtainable through ack and numbering of messages
- When a process P_i receives *m*
  - *m* is stored in a totally-ordered queue based on t_m
  - P_i sends ack to all others ($t_{ack} > t_m$)
- *m* is processed by P_i only when it is on the top of the queue and P_i received its ack from all other processes

N.B. The queue is the same for all processes

Limitations of "Lamport"

- Lamport ensures
  - if $a \rightarrow b$, then $C(a) < C(b)$
- The algorithm is
  - simple
  - completely distributed
  - fault tolerant

- Lamport does not ensure
  - if $C(a) < C(b)$, then $a \rightarrow b$
    - Due to events on remote processes
- How to understand if there is a cause-effect relation looking at the timestamps?

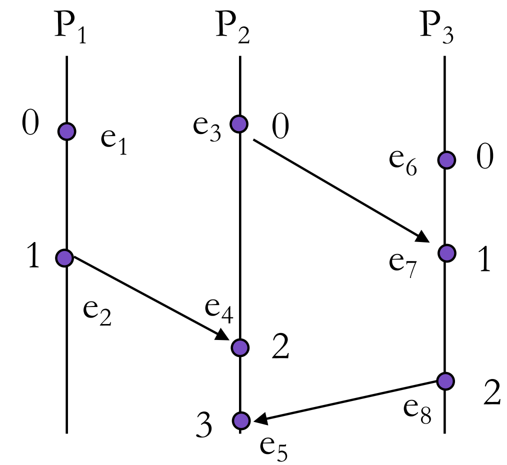
Limitations of "Lamport” – Example

Lamport does not capture the following causal relation

If $C(e_i) < C(e_j)$ then $e_i \rightarrow e_j$

$C(e_3) < C(e_2)$
but $e_3 \not\rightarrow e_2$

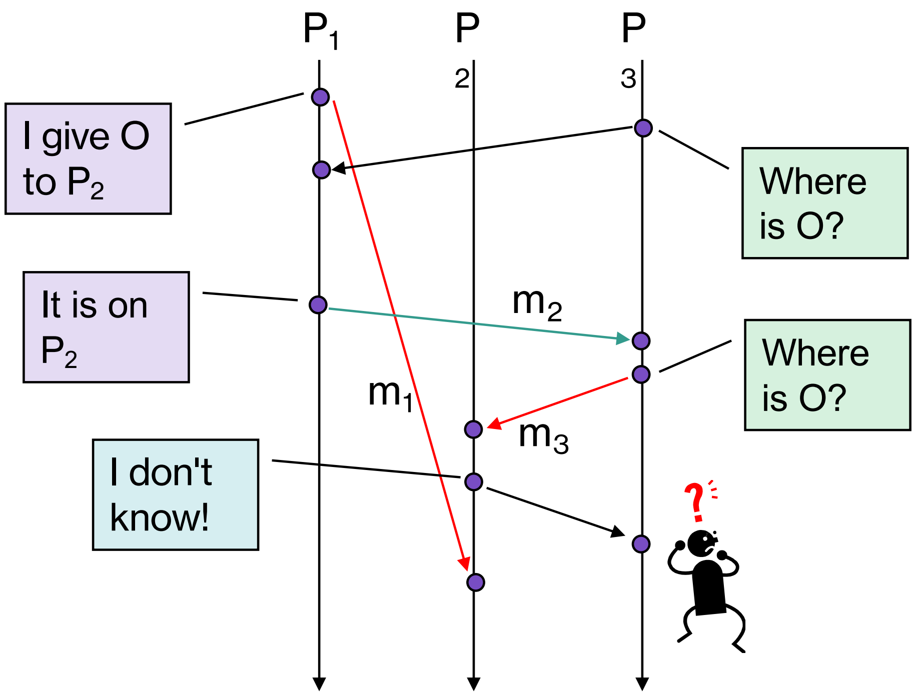
e_3 and e_2 are concurrent



Is causal relation necessary?

- It allows
  - debugging
  - liveness and fairness for mutual exclusion
  - consistency for replicated databases
  - deadlock detection
  - construction of consistent global states for crash recovery
  - determination of the degree of parallelism between processes
  - ...

Is causal relation necessary? – Example



- $S(m)$ = m is sent
- $R(m)$ = m is received
- **Violation of causality**
  - m_1 arrives after m_3
  - $S(m_1) \rightarrow S(m_3)$ but $R(m_3) \rightarrow R(m_1)$

Violation of causality – Solutions?

Typical problem

- A processor P_i sends two messages m_1, m_2
- A processor P_j receives two messages m_1, m_2
- $S(m_1) \rightarrow S(m_2)$
- $R(m_2) \rightarrow R(m_1)$

Solution

- To avoid violation of causality we must know if 2 events are causally related to each other
- By vectors of logical clocks

Vector clock [Fidge 91 / Mattern 88 / Raynal&Singhal 96]

- Similar to logical clock, but more values (vector) rather than just one
  - *n* processors → *n* elements in the vector
  - $C_i[i]$: logical time of P_i
  - $C_i[j]$: the best estimation of the logical time of P_j by P_i
    - C_i [j] = time of the last event in P_j that "happened before" the current time of P_i

Implementation rules

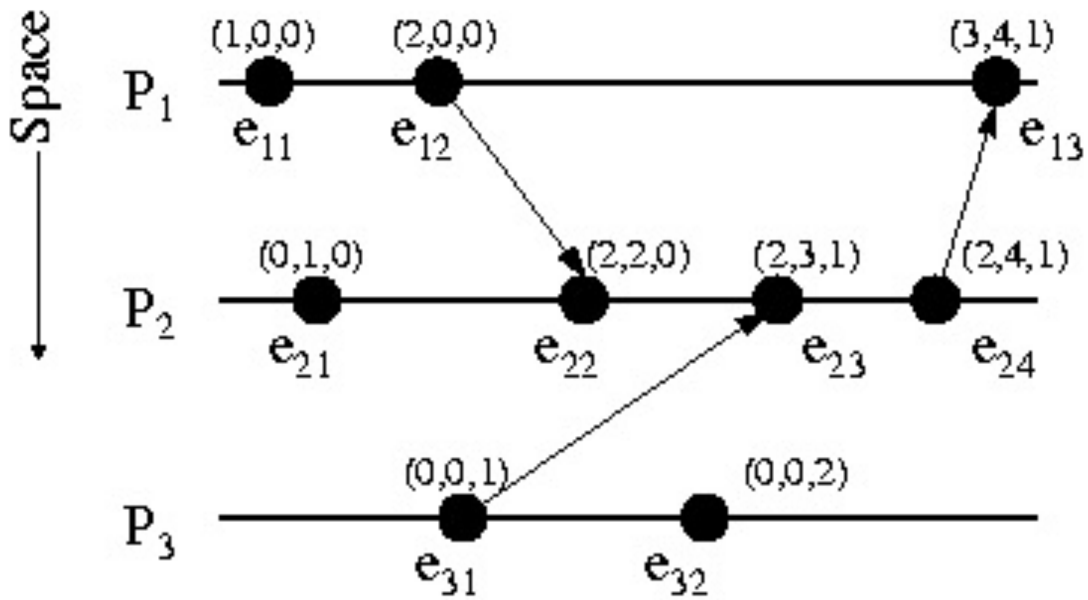
[IR1]

- $C_i[i]$ is incremented between two successive events in the processor such that $P_i C_i[i] = C_i[i] + d$ $(d > 0)$

[IR2]

- If *a* corresponds to sending a message *m* by P_i, then
  - a vector timestamp $t_m = C_i(a) = (C_i[1], \ldots, C_i[n])$ is assigned to *m*
  - when *m* is received by P_j, C_j is computed as $\forall\, k,\ C_j[k] = \max\,(C_j[k], t_m[k]) + d$

Vector clock - Example



Vector clock – Theorem

Theorem

- if $t_a < t_b$, then $a \rightarrow b$ and viceversa

Definition

- $t_a < t_b$ if and only if
  - $t_a[i] \leq t_b[i]$ for each i
  - There exists j such that $t_a[j] \neq t_b[j]$

Examples

$(1,0,3) \rightarrow (2,0,5)$

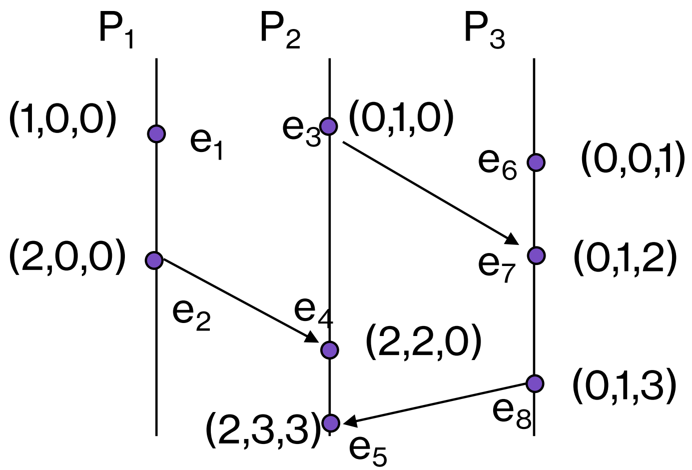
$(1,1,3) \not\rightarrow (1,0,3)$

$(1,1,3) \not\rightarrow (1,1,3)$

It allows identifying cause-effect relations between events

- Not possible with Lamport's clocks

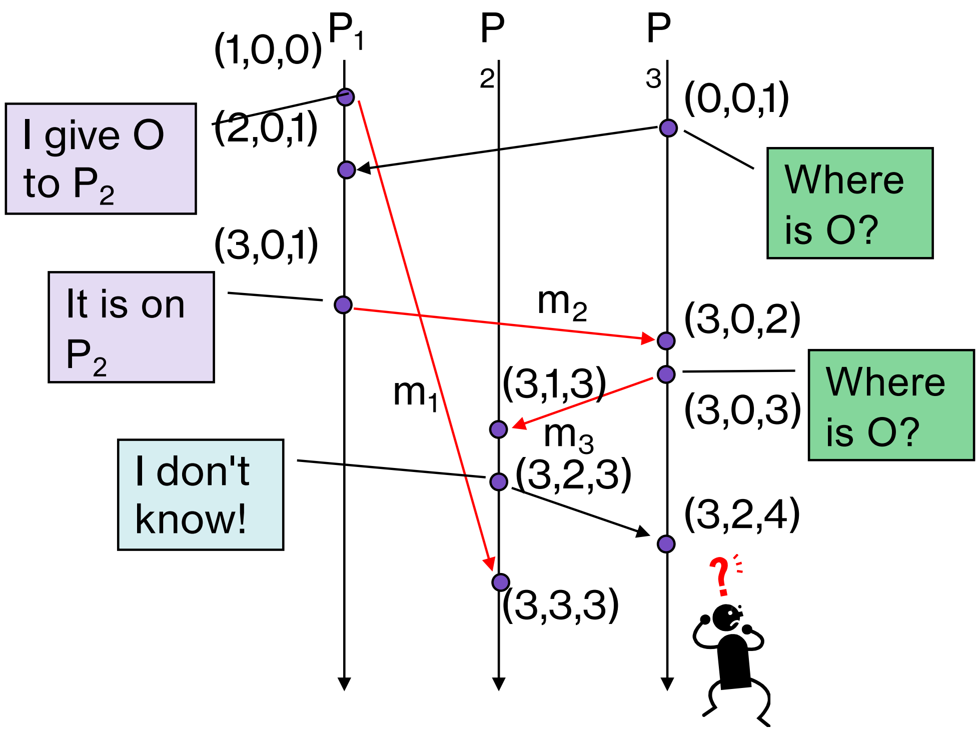
If $t_a < t_b$, then $a \rightarrow b$? – Example



$C(e_3) < C(e_2)$?

No, $e_3 \not\rightarrow e_2$
e_3 and e_2 are concurrent

Violation of causality



- When m_1 arrives, P_2 understands that there has been a violation of causality because:

$$t_{send}(m_1) < t_{send}(m_3)$$

but

$$t_{receive}(m_3) < t_{receive}(m_1)$$

Physical clocks

In real-time cases we need individual clocks linked to actual time

How to guarantee synchronization of physical clocks?

Coordinated Universal Time (UTC)

Based on International Atomic Time (TAI)

- 1s = time necessary by 133 cesium atom to accomplish 9.192.631.770 transitions
- 50 laboratories calculate the time → average = TAI
- 86400s TAI last 3ms less than the average solar day → compensation required

Atomic clock output broadcasted by radio stations and satellites to the Earth

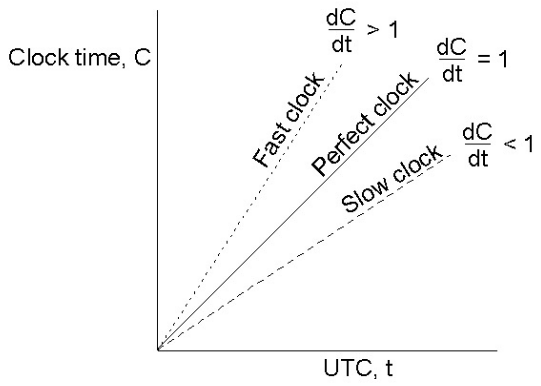
- Accuracy of radio signals
  - from radio stations on earth about 0.1 - 10 ms
  - GPS signals about 1 μs

Computers with receivers can synchronize with these signals

- Other machines must be synchronized

Synchronization of physical clocks

- Model of the system
  - Each host has a timer that causes an interrupt H times per second (tick)
  - When the timer is triggered, 1 is added to a software clock C(t)
- If t is the UTC, ideally $C(t) = t$
  - In pratice $dC/dt \neq 1$
  - Tolerance r
    - Maximum drift rate
  - $1-r < dC/dt < 1+r$



Synchronization – How often?

Assumption

There is a process (time server) that is able to provide a reference time	Periodically, hosts communicate with the time server to synchronize

How often?

After time d*t*, two clocks may differ from 2*r*d*t* with *r* = clock tolerance	If *e* = max error allowed with respect to the universal time *t*, 2rdt < *e*

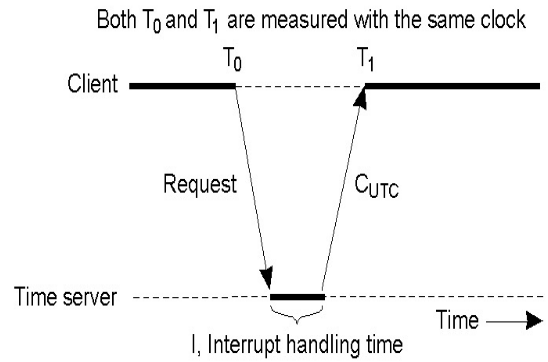
d*t* = distance between two synchronizations = *e*/2*r* seconds

Synchronization of physical clocks

- Centralized methods
  - Cristian's algorithm
  - Berkeley algorithm
- Distributed method

Cristian's algorithm

- Time-server has a copy of the UTC
- Periodically, processes query the time server
  - Propagation time of the request must be considered
  - Estimation: (T1 – T0)/2

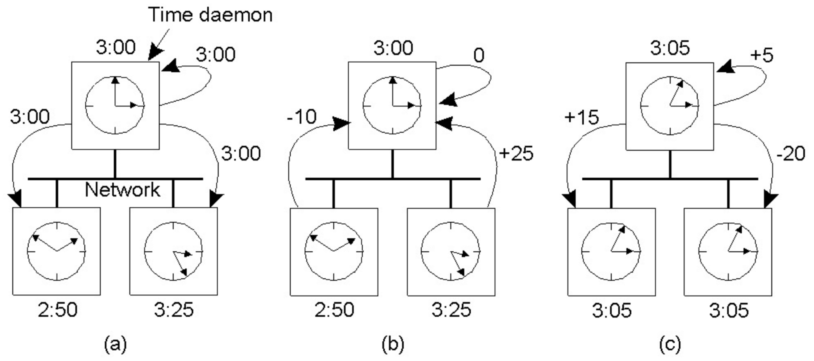


- Difficulty
  - Process to be synchronized can have "fast" clock
- Solution
  - Progressive slowdown of the clock to be synchronized

Berkeley algorithm

1. Time server (daemon) asks time to each host by providing its time
2. Hosts provide the difference between their time and that of the server
3. The daemon tells hosts how to adjust the clocks
    - Es: Average collected time

- Suitable for manually adjusted server, without UTC reference
- Used in Berkeley Unix



Distributed method

1. Time divided into periods of fixed length
2. Hosts send their time in broadcast at the beginning of each period
   - Possibly non-simultaneous submissions due to differences in the various clocks
     - Therefore the broadcast reception remains active for a certain time window S
3. When S expires, the local clock is updated based on
   - Average of the received values
   - Average of received values, excluding N extreme values (outliers)

- It is possible to take into account the estimated propagation time of the broadcast from a given source