Distributed Systems and Algorithms

Martin Quinson <martin.quinson@loria.fr>
Abdelkader Lahmadi <abdelkader.lahmadi@loria.fr>
Olivier Festor <olivier.festor@inria.fr>

LORIA – INRIA Nancy Grand Est

2016-2017

(compiled on: October 2, 2017)

Chapter 1

Theoretical foundations

- Time and State of a Distributed System
- Ordering of events
- Abstract Clocks
  - Global Observer
  - Logical Clocks
  - Vector Clocks

Time bugs

"On August 12, 1853, two trains on the Providence & Worcester Railroad were headed toward each other on a single track. The conductor of one train thought there was time to reach the switch to a track to Boston before the approaching train was scheduled to pass through. But the conductor's watch was slow. As his speeding train rounded a blind curve, it collided head-on with the other train—fourteen people were killed. The public was outraged. All over New England, railroads ordered more reliable watches for their conductors and issued stricter rules for running on time."

Time bugs

- At midnight on December 31, 2008, many first generation Zune 30 models froze. Microsoft stated that the problem was caused by the internal clock driver written by Freescale and the way the device handles a bissextile year.
- 28/02/2010: PS3 owners worldwide suddenly found themselves locked out of the PlayStation Network, with trophy and game functionality severely compromised. The bug causes the date on the system to be reset to 1st January 2000, and reseting the clock manually makes no difference.
- 10/2010: The iPhone Daylight Savings Time Alarm Clock Bug
- 25/02/1991: The Patriot Missile Failure during the Gulf War, killing 28 soldiers and injuring 100 other people. The cause is inaccurate calculation of the time (https://www.ima.umn.edu/ãrnold/disasters/patriot.html)

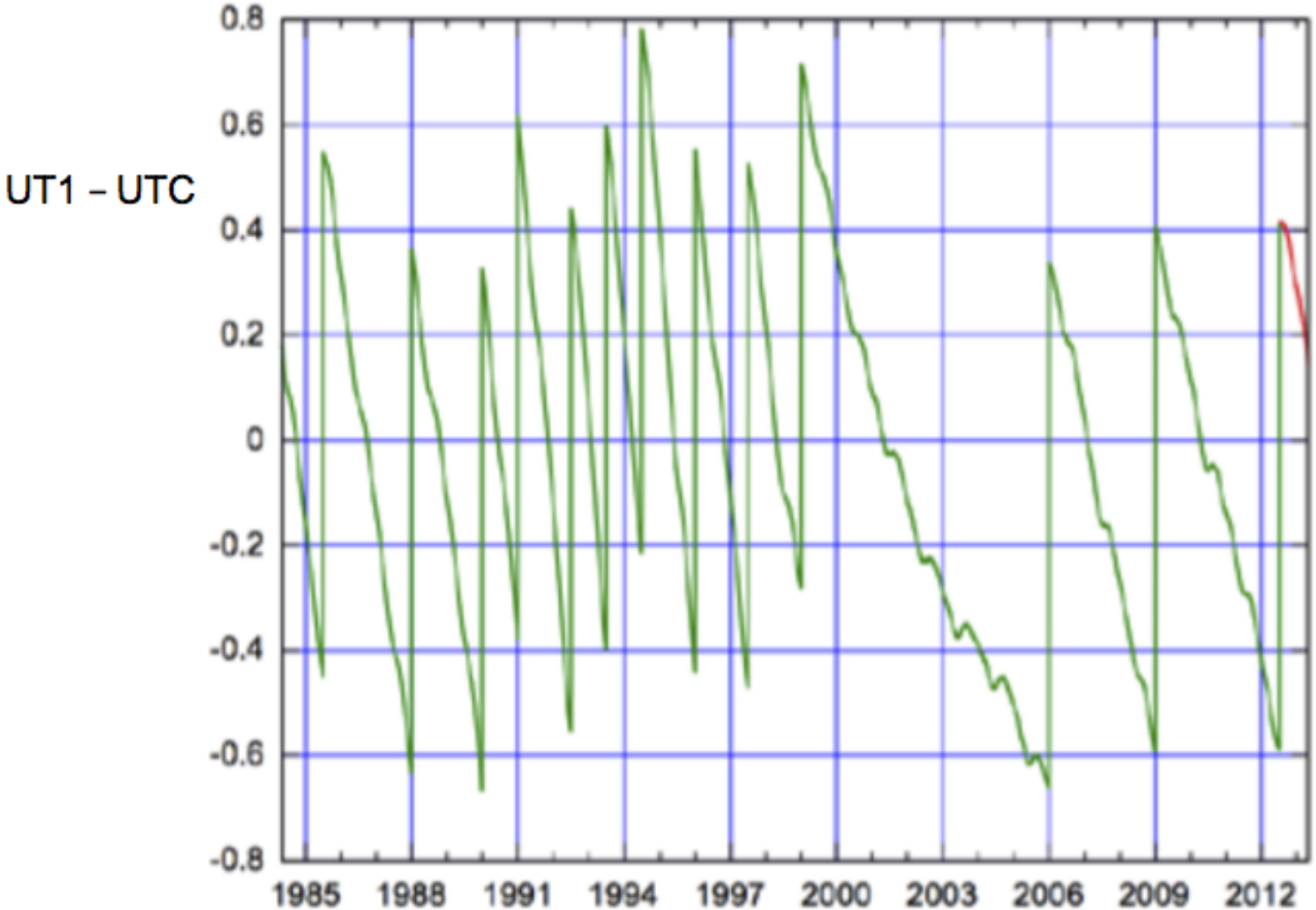
Time standards

- Primary standard: rotation of earth
- De facto primary standard: atomic clock
- 1 atomic second = 9,192,631,770 orbital transitions of $Cesium^{133}$ atom
- 86400 atomic sec = 1 solar day - 3 ms (requires leap second correction each year)
- UT1: based on astronomical observations, Greenwich Mean Time
- International Atomic Time (TAI): the weighted average of the readings of nearly 300 atomic clocks in over fifty national laboratories worldwide
- Coordinated Universal Time (UTC): TAI + leap seconds to be within 800ms of UT1
- UTC is based on TAI since 1972

GPS: Global Positioning System

- System of 32 satellites broadcast accurate spatial coordinates and time maintained by atomic clocks : location and precise time computed by triangulation
- GPS time is nearly 14 sec ahead of UTC: per the theory of relativity, on board clocks run at slower rate than the clocks on the earth (38 ms per day).
- Receivers apply a clock-correction offset in oder to display UTC correctly

Comparing time standards



Atomic clocks

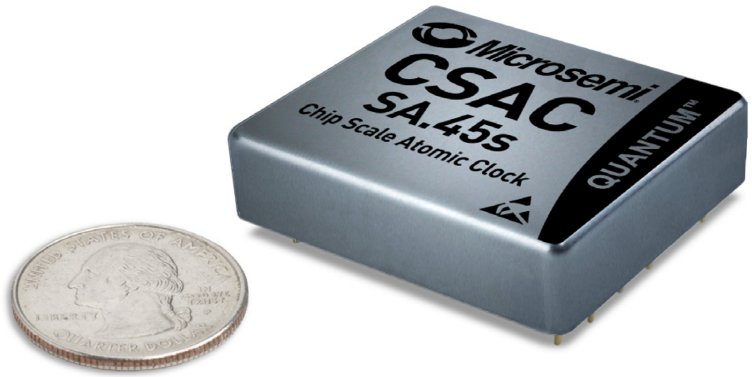
Chip-Scale Atomic Clock

Key Features

- Power consumption <120 mW
- Less than 17 cc volume, 1.6" x 1.39" x 0.45"
- Aging <3.0E-10/month
- 10 MHz CMOS-compatible output
- 1 PPS output and 1 PPS input for synchronization
- RS-232 interface for monitoring and control
- Short term stability (Allan Deviation) of 2.5E-10@ TAU =1 sec

Applications

- Underwater sensor systems
- GPS receivers
- Backpack radios
- Anti-IED jamming systems
- Autonomous sensor networks
- Unmanned vehicles



With an extremely low power consumption of <120 mW and a volume of <17 cc, the Microsemi® SA.45s Chip Scale Atomic Clock (CSAC) brings the accuracy and stability of an atomic clock to portable applications for the first time.

The SA.45s provides 10 MHz and 1 PPS outputs at standard CMOS levels, with short-term stability (Allan Deviation) of 2.5E-10 @ TAU =1 sec, long-term aging of < 3E-10/month, and maximum frequency change of 5E-10 over an operating temperature range of -10°C to +70°C. The unit can also be ordered with a wider temperature range (Option 002) of -40°C to +85°C, with slightly higher power consumption and a wider maximum frequency change over temperature.

The SA.45s CSAC accepts a 1 PPS input that may be used to synchronize the unit's 1 PPS output to an external reference clock with ±100 ns accuracy. The CSAC can also use the 1 PPS input to discipline its phase and frequency to within 1 ns and 1.0E-12, respectively.

A standard CMOS-level RS-232 serial interface is built in to the SA.45s. This is used to control and calibrate the unit and also to provide a comprehensive set of status monitors. The interface is also used to set and read the CSAC's internal time-of-day clock.

The talking clock

- The first telephone speaking clock service was introduced in France, in association with the Paris Observatory, on 14 February 1933
- It is available on 3699 (it costs 50 cents!!)
- http://www.dailymotion.com/video/xxeyyr_le-visage-de-l-horloge-parlante_news

Sequential and Concurrent Events

- Sequential: Totally ordered in time. Feasible in a single process but not true in a distributed system.
- The clocks commonly available at processors distributed across a system do not exactly show the same time
- Built-in atomic clocks are not yet cost-effective
- Certain regions in the world cannot receive such time broadcasts from reliable timekeeping sources
- GPS signals are difficult to receive inside a building
- How to synchronize physical clocks ?
- Can we define sequential and concurrent events ? without using physical clocks, since physical clocks cannot be perfectly synchronized ?

Messages from ACARS (Aircraft Communication Addressing and Reporting System)

```
================================================================================
                    A I R   F R A N C E   A C A R S
                      Liste Des Evenements ACARS
  28 /  256                      **********

MATERIEL  : AV FGZCP   MSG:                          DATES du: 120509 au: 0106
SECT.ENT  :         ATA/PH:       /                  (TA: A330)
--------------------------------------------------------------------------------
  ATA/PH  Typ Sel          Libelle succint du message              Date    Novo
27 23/06  WRN   WN0906010210  272302006F/CTL RUD TRV LIM FAULT 09-06-01 AF 447
22 83/06  WRN   WN0906010210  228301106FLAG ON F/O PFD         09-06-01 AF 447
22 83/06  WRN   WN0906010210  228300106FLAG ON CAPT PFD        09-06-01 AF 447
34 43/06  WRN   WN0906010210  344300506NAV TCAS FAULT          09-06-01 AF 447
22 30/06  WRN   WN0906010210  223002506AUTO FLT A/THR OFF      09-06-01 AF 447
22 83/06  WRN   WN0906010210  228301206FLAG ON F/O PFD        09-06-01 AF 447
22 83/06  WRN   WN0906010210  228300206FLAG ON CAPT PFD       09-06-01 AF 447
27 91/06  WRN   WN0906010210  279100506F/CTL ALTN LAW          09-06-01 AF 447
22 62/06  WRN   WN0906010210  226201006AUTO FLT                09-06-01 AF 447
22 10/06  WRN   WN0906010210  221002006AUTO FLT AP OFF         09-06-01 AF 447
38 31/06  FLR   FR0905312245  38310006VSC    X2,,,,,,,LAV CONF 09-05-31 AF 447
38 31/06  WRN   WN0905312245  383100506MAINTENANCE STATUS      09-05-31 AF 447
    /                                                         09-05-31 AF 444
    /    PFR   DBN013/            090531201300MAY3108522003P  09-05-31 AF 444
PF2/3=Fin   PF4/5=Pag.avion   PF6/7/8/9=Pagination   X=consult.   PF12=Edition
================================================================================
```

```
================================================================================
                    A I R   F R A N C E   A C A R S
                      Liste Des Evenements ACARS
  14 /  256                      **********

MATERIEL  : AV FGZCP   MSG:                          DATES du: 120509 au: 0106
SECT.ENT  :         ATA/PH:       /                  (TA: A330)
--------------------------------------------------------------------------------
  ATA/PH  Typ Sel          Libelle succint du message              Date    Novo
21 31/06  WRN   WN0906010214  213100206ADVISORY                  09-06-01 AF 447
22 83/06  FLR   FR0906010213  22833406AFS    1,,,,,,,FMGEC1(1C 09-06-01 AF 447
34 10/06  WRN   WN0906010214  341036006MAINTENANCE STATUS       09-06-01 AF 447
27 90/06  WRN   WN0906010213  279004006F/CTL SEC 1 FAULT        09-06-01 AF 447
27 90/06  WRN   WN0906010213  279002506F/CTL PRIM 1 FAULT       09-06-01 AF 447
34 12/06  FLR   FR0906010211  34123406IR2    1,EFCS1X,IR1,IR3, 09-06-01 AF 447
34 22/06  FLR   FR0906010211  34220006ISIS   1,,,,,,,ISIS(22FN 09-06-01 AF 447
34 10/06  WRN   WN0906010212  341040006NAV ADR DISAGREE         09-06-01 AF 447
34 12/06  WRN   WN0906010211  341201106FLAG ON F/O PFD          09-06-01 AF 447
34 12/06  WRN   WN0906010211  341200106FLAG ON CAPT PFD         09-06-01 AF 447
27 93/06  FLR   FR0906010210  27933406EFCS1  X2,EFCS2X,,,,,,FC  09-06-01 AF 447
34 11/06  FLR   FR0906010210  34111506EFCS2  1,EFCS1,AFS,,,,,P 09-06-01 AF 447
27 90/06  WRN   WN0906010210  279045006MAINTENANCE STATUS       09-06-01 AF 447
27 90/06  WRN   WN0906010210  279045506MAINTENANCE STATUS       09-06-01 AF 447
PF2/3=Fin   PF4/5=Pag.avion   PF6/7/8/9=Pagination   X=consult.   PF12=Edition
================================================================================
```

Physical Clock Synchronization

Question 1

Why is physical clock synchronization important?

Question 2

With the price of atomic clocks or GPS coming down, should we care about physical clock synchronization?

Types of Synchronization

- Internal: keep the reading of a system of autonomous clocks closely synchronized with one another, despite the failure or malfunction of one or more clocks.
- External: maintain the reading of a clock close to the UTC as possible.
- Phase: clocks are driven by the same source of pulses, they tick at the same rate and are not autonomous

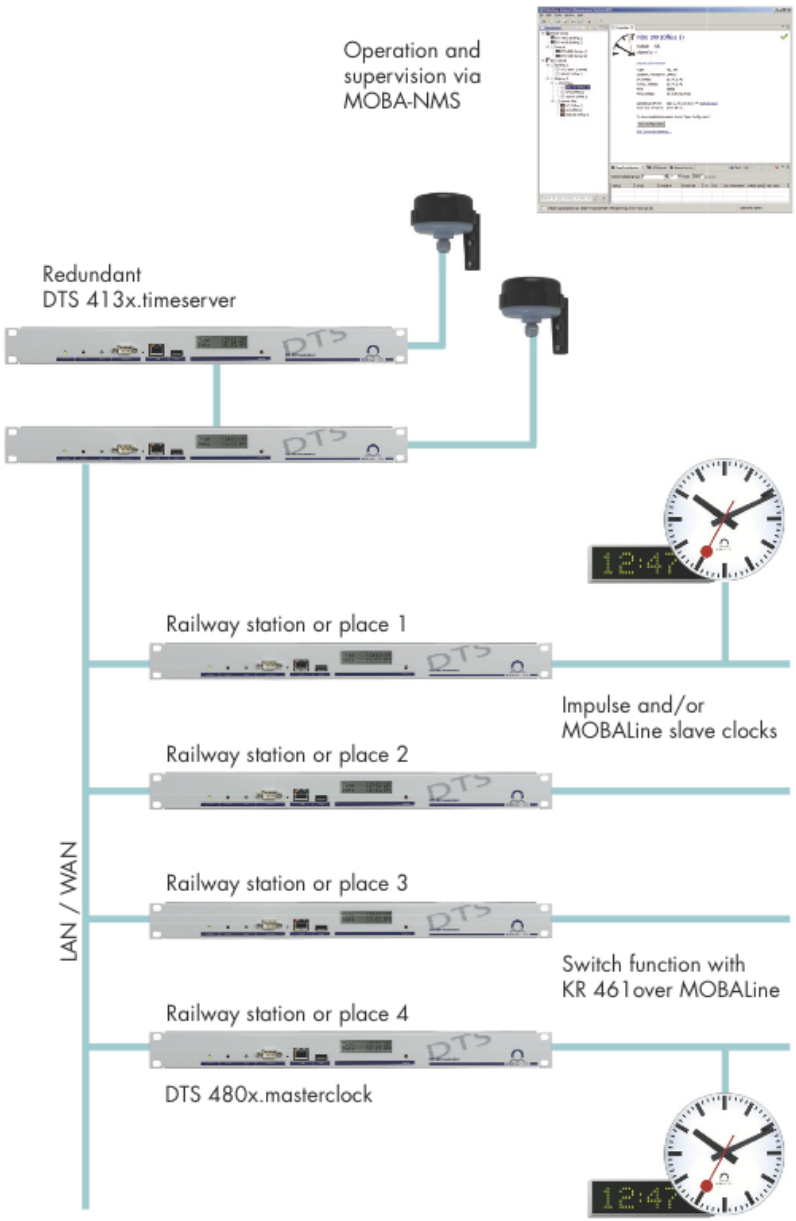
Types of clocks

- Unbounded 0,1,2,3, ...
- Bounded 0,1,2,..,M-1, 0, 1 ...

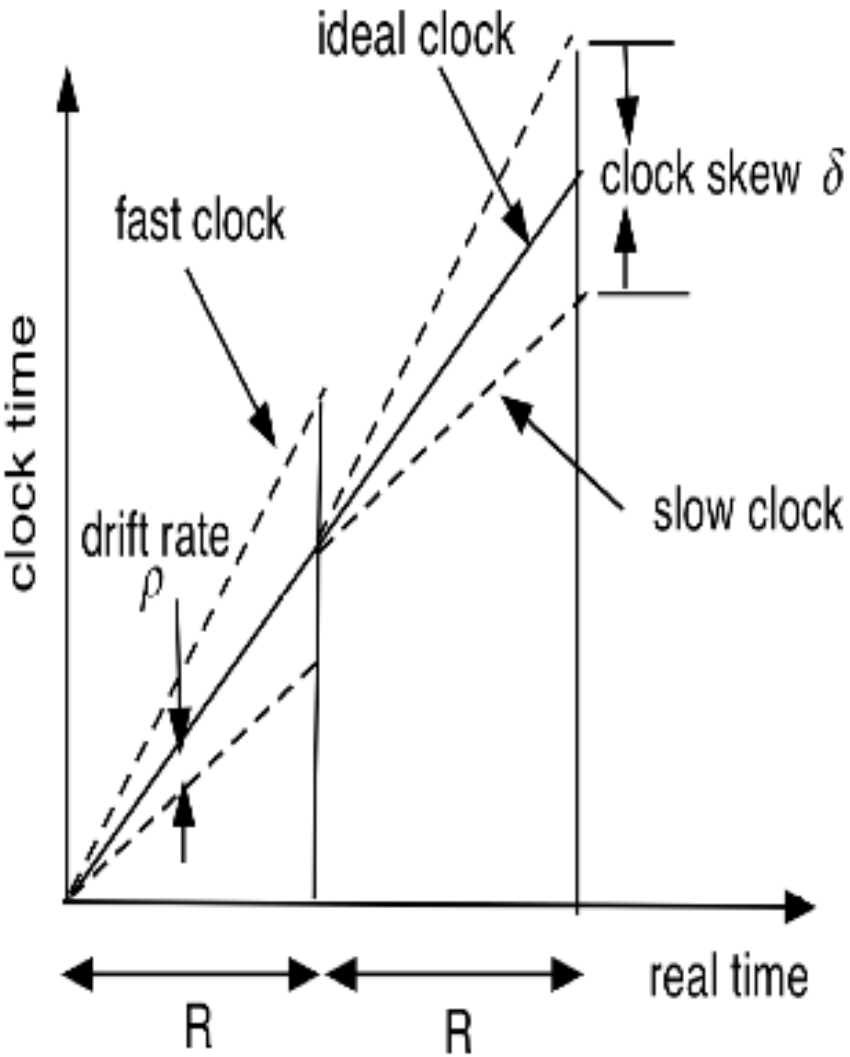
Example: Swiss Time Systems

www.MOBATIME.com

- Example: DTS 4802.masterclock
- Control conventional impulse clocks
- Synchronized by a time signal receiver (DCF 4500 or GPS 4500) and/or NTP time servers
- Application example for railways, airports, schools, hospitals, etc



Clock Synchronization: terminology



- Drift rate (dérive): ρ
- Clock skew (différence entre deux horloges): δ
- Resynchronization interval: R
- Max drift rate ρ implies : $(1-\rho) \leq dC/dt \leq (1+\rho)$
- Drift is unavoidable
- Accounting for propagation delay
- Accounting for processing delay
- Faulty clocks

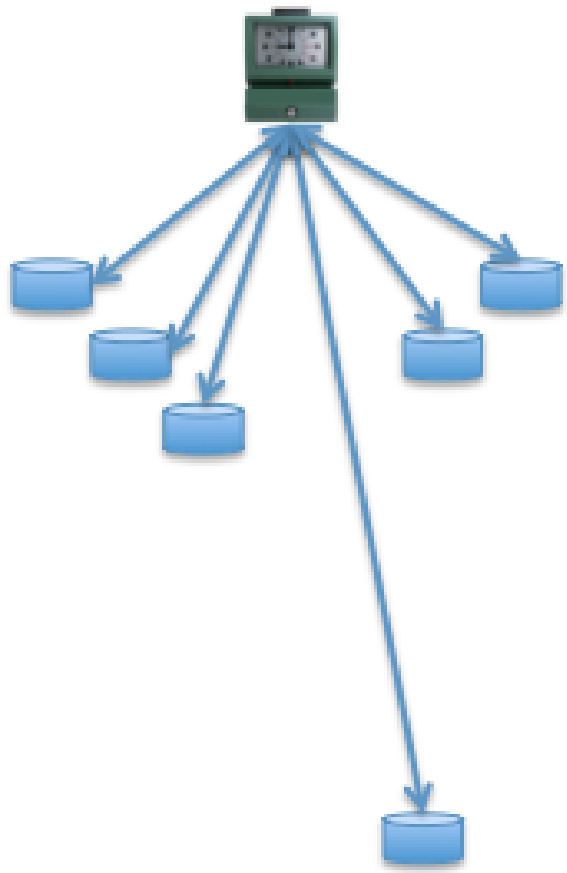
Internal Synchronisation

Berkeley Algorithm

A simple averaging algorithm that guarantees mutual consistency $|c(i) - c(j)| < \delta$.

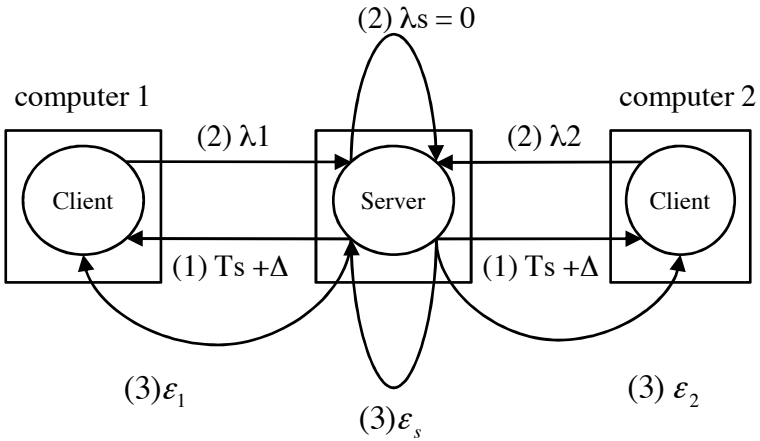
- The participants elect a leader
- The leader coordinates the synchronization

1. A master reads every clock in the system
2. Discard outliers and substitute them by the value of the local clock
3. Computes the average of these values and sends the needed adjustment to the participating clocks



Resynchronize interval will depend on the drift rate

Berkeley algorithm



with

(1)(2)(3) Are events

Ti (i =s,1,2) computer times

λi clock drift between a computer i and server (λi = Ti-Ts)

Δ mean time to transmit a message with $\Delta_{min} \leq \Delta \leq \Delta_{max}$

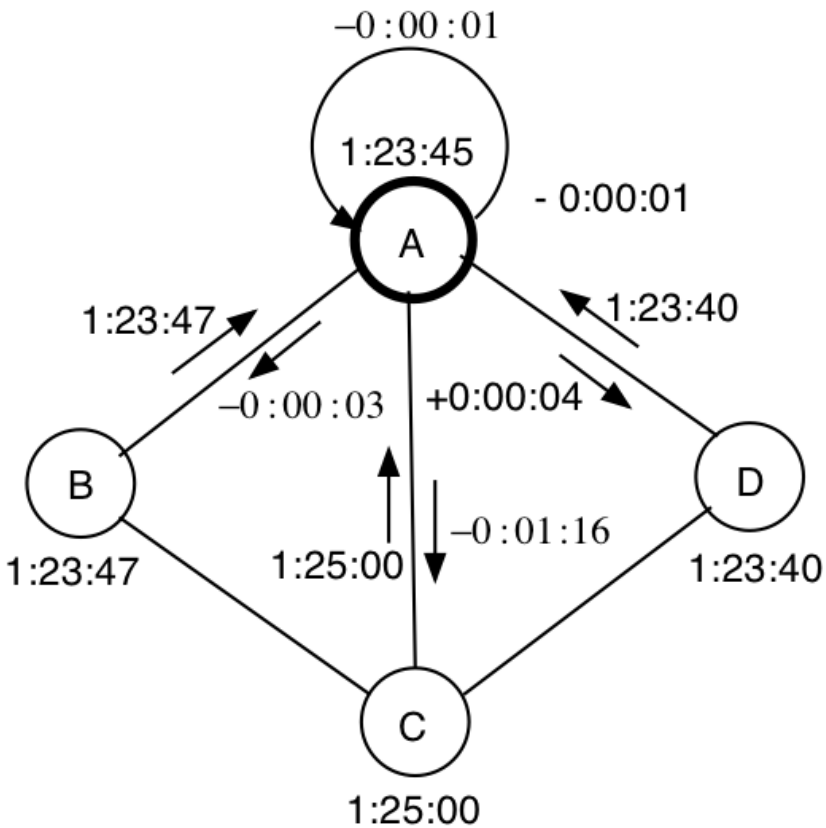
$\overline{\lambda}$ mean clock drift

		(1) Clock	(2) λi = Ti- Ts	(3) $\varepsilon_i = \overline{\lambda} - \lambda_i$	$T_i+\varepsilon_i$
Ts		3:00	0	+5	3:05
Ti	T1	3:25	+25	-20	3:05
	T2	2:50	-10	+15	3:05

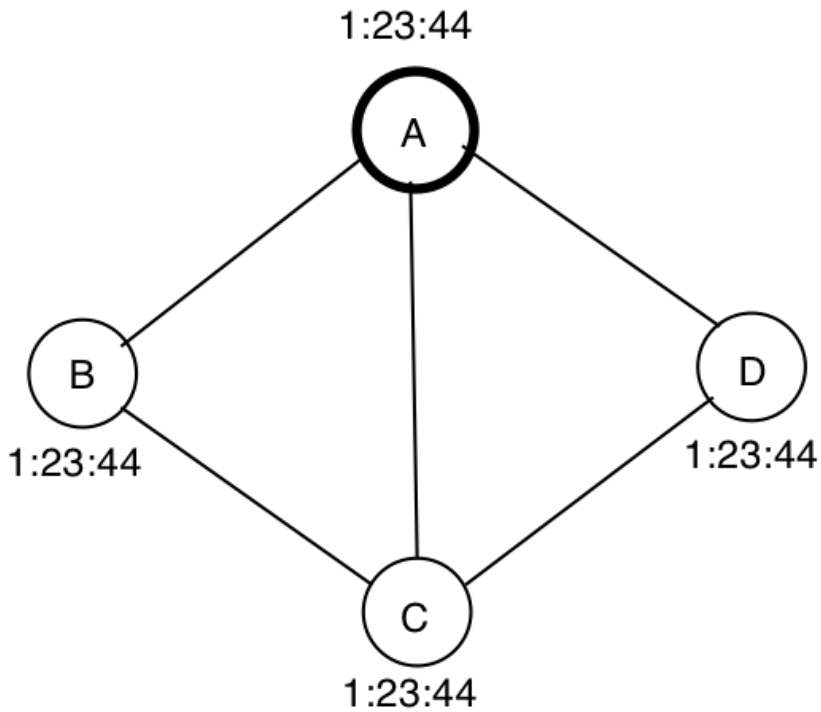
$$\overline{\lambda} = \sum_{i=1}^{n} \frac{\lambda_i}{n} = +5$$

Synchronization error Δ is introduced at (1), when the server sends its clock value. Rest of exchanges (2) (3) are computed from delta parameters, the time delay of messages is then without impact.

Berkeley algorithm



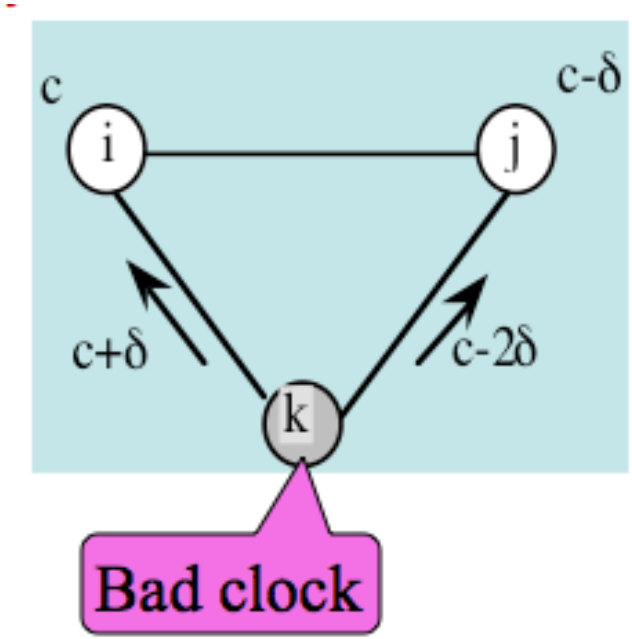
(a) Before



(b) After

Internal Synchronisation

Lamport and Melliar-Smith averaging algorithm handles byzantine clocks too



A faulty clocks exhibits 2-faced or byzantine behavior

Assume n clocks, at most t are faulty

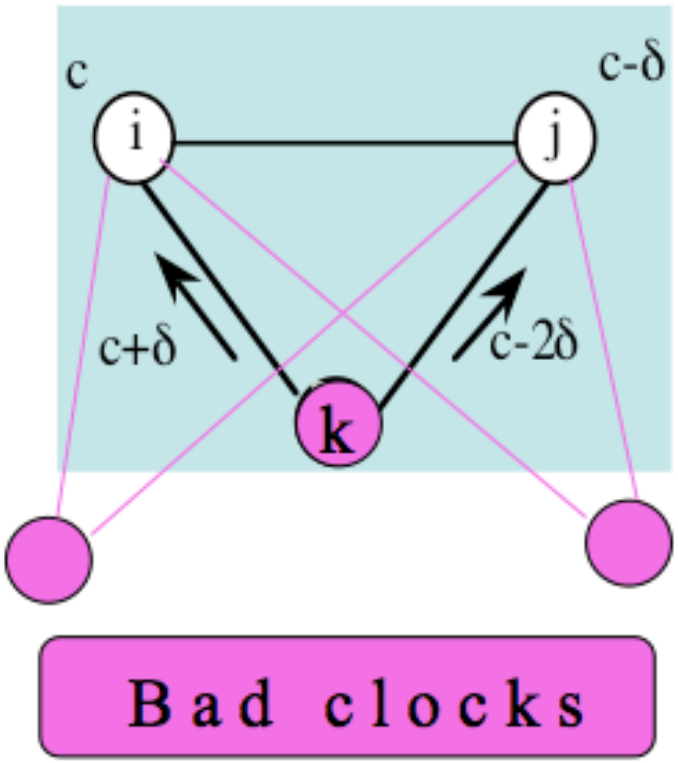
1. Read every clock in the system
2. Discard outliers and substitute them by the value of the local clock
3. Update the clock using the average of these values

Why ?

Synchronization is maintained if $n > 3t$.

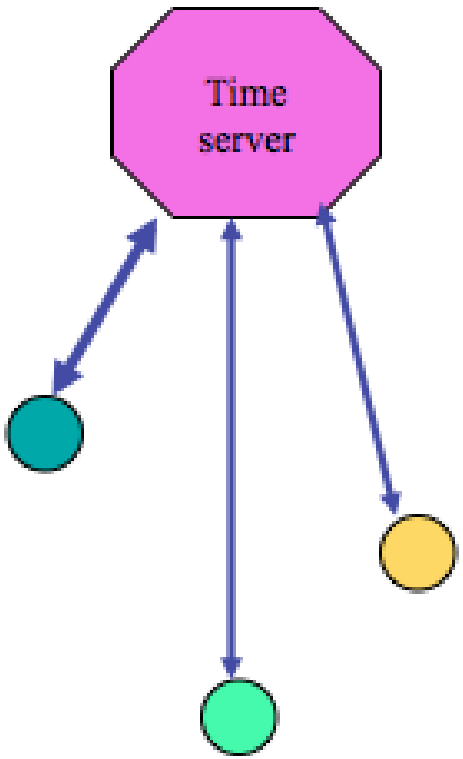
Internal Synchronisation

Lamport and Melliar-Smith algorithm: byzantine clocks



- The maximum difference between the averages computed by two non-faulty nodes is $(3 \times t \times \delta/n)$
- To keep the clocks synchronized : $(3 \times t \times \delta/n) < \delta$
- So, $3 \times t < n$

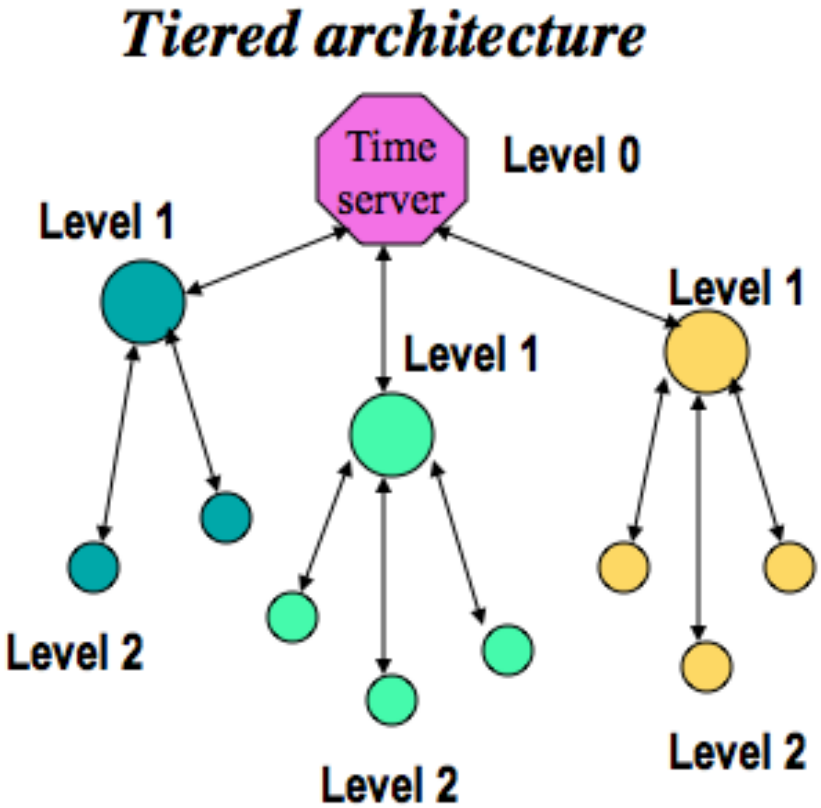
External Synchronisation: Cristian's Method



- Client pulls data from a time server every R unit of time, where $R < \frac{\delta}{2\rho}$
- For accuracy clients must compute the Round Trip Time (RTT), and compensate for this delay while adjusting their own clocks (Too large RTT' are rejected)
- $T_{local} = T_{server} + \frac{RTT}{2}$

Network Time Protocol

Tiered Architecure

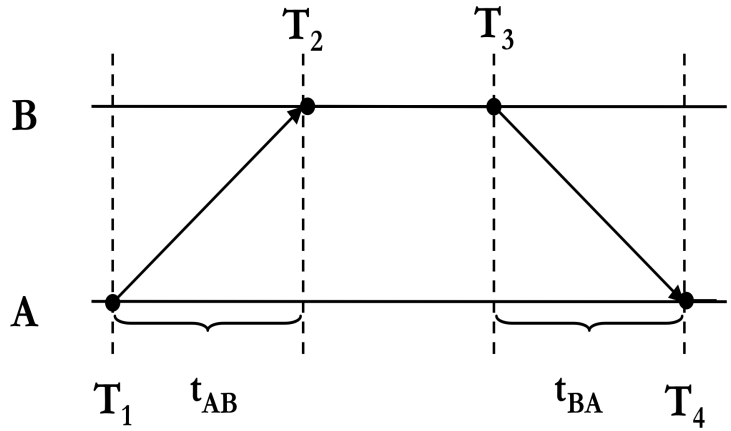


- Broadcast mode: least accurate
- Procedure call: medium accuracy
- Peer-to-Peer: upper level servers use this for max accuracy

The tree can configure itself if some node fails.

A computer will try to synchronize its clock with several servers, and accept the best results to set its time. Accordingly, the synchronization subnet is dynamic.

Network Time Protocol



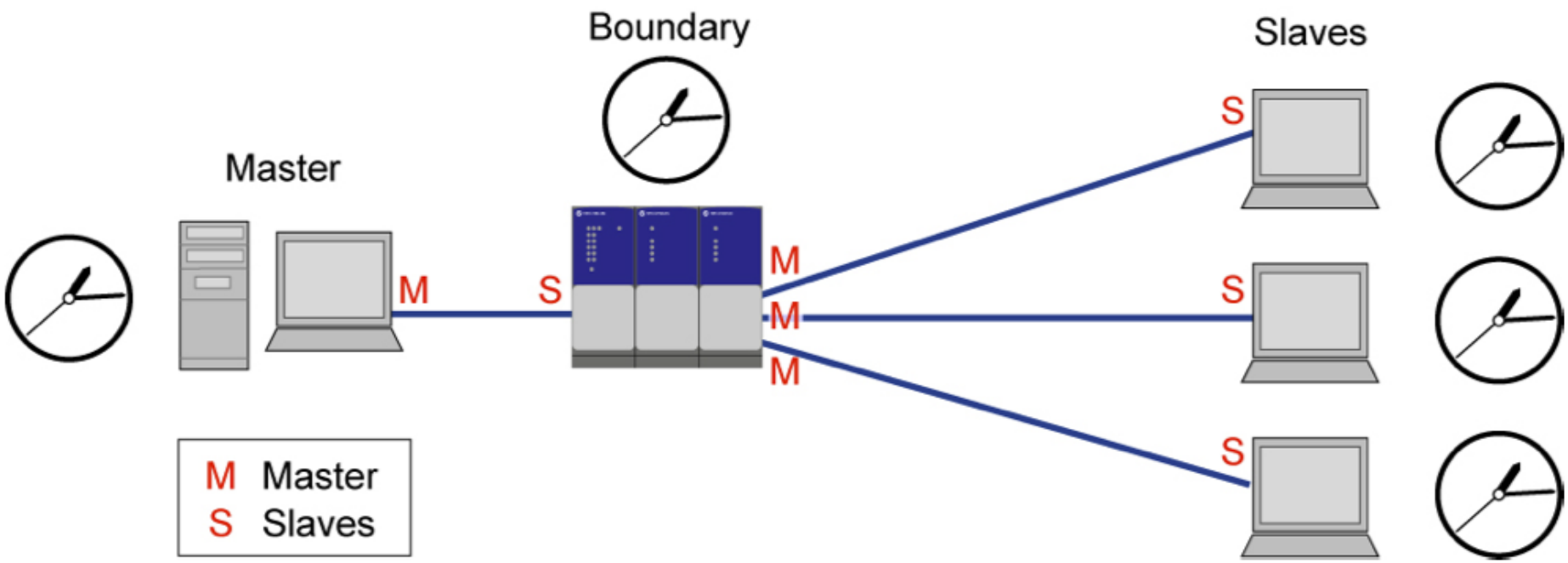
- A: Client B: Time Server
- Transmission time from A to B (t_{AB}) is $t_{AB} = T_2 - T_1$
- Transmission time from A to B (t_{BA}) is $t_{BA} = T_4 - T_3$

Roundtrip delay (δ) and offset (θ) can be computed as follows

- Roundtrip delay (δ) $\delta = (T_4 - T_1) - (T_3 - T_2)$
- Offset (θ) $\theta = [(T_2 - T_1) + (T_3 - T_4)] / 2$

Precision Time Protocol, IEEE 1588

- Synchronisation of distributed clocks
- Two versions: 2002 and 2008
- External source clock: GPS, Atomic clocks
- Multicast over Ethernet networks
- Accuracy of less than 1 microsecond



PTP architecture

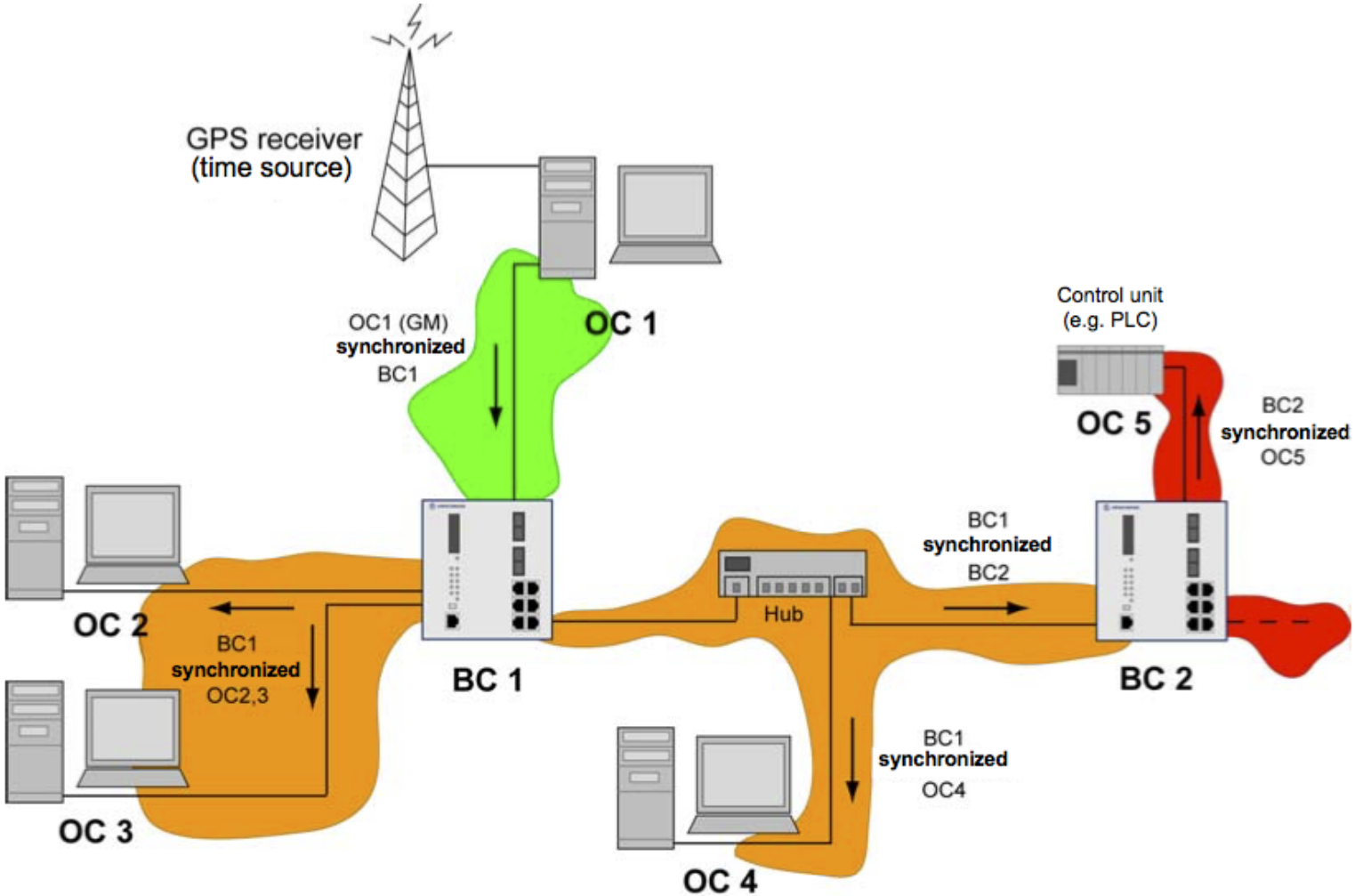
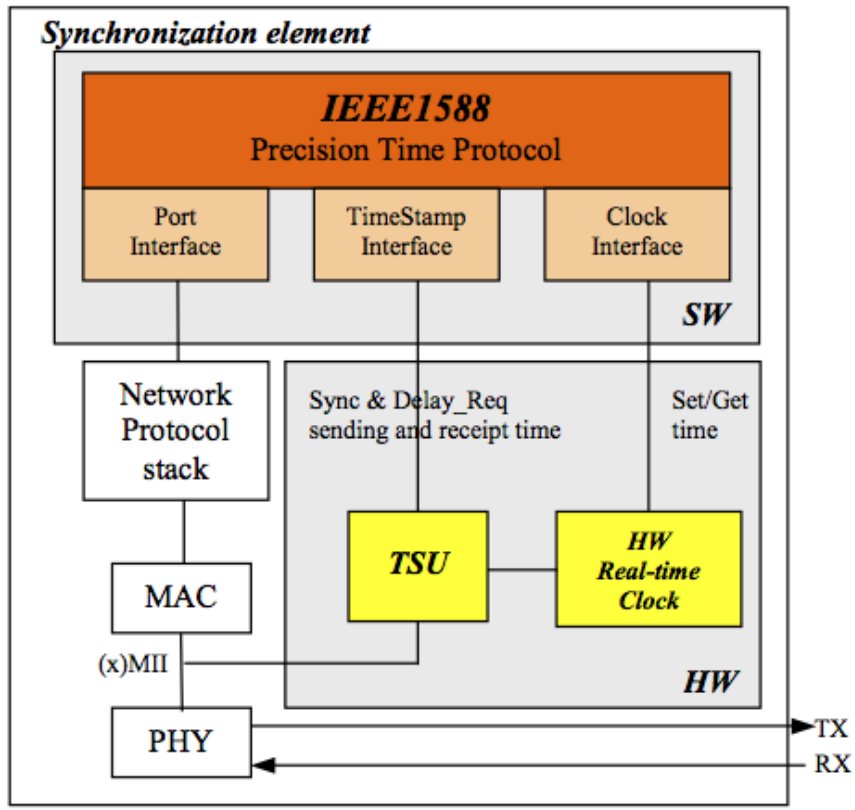


Figure 4: PTP synchronization domain

Architecture of a PTP node

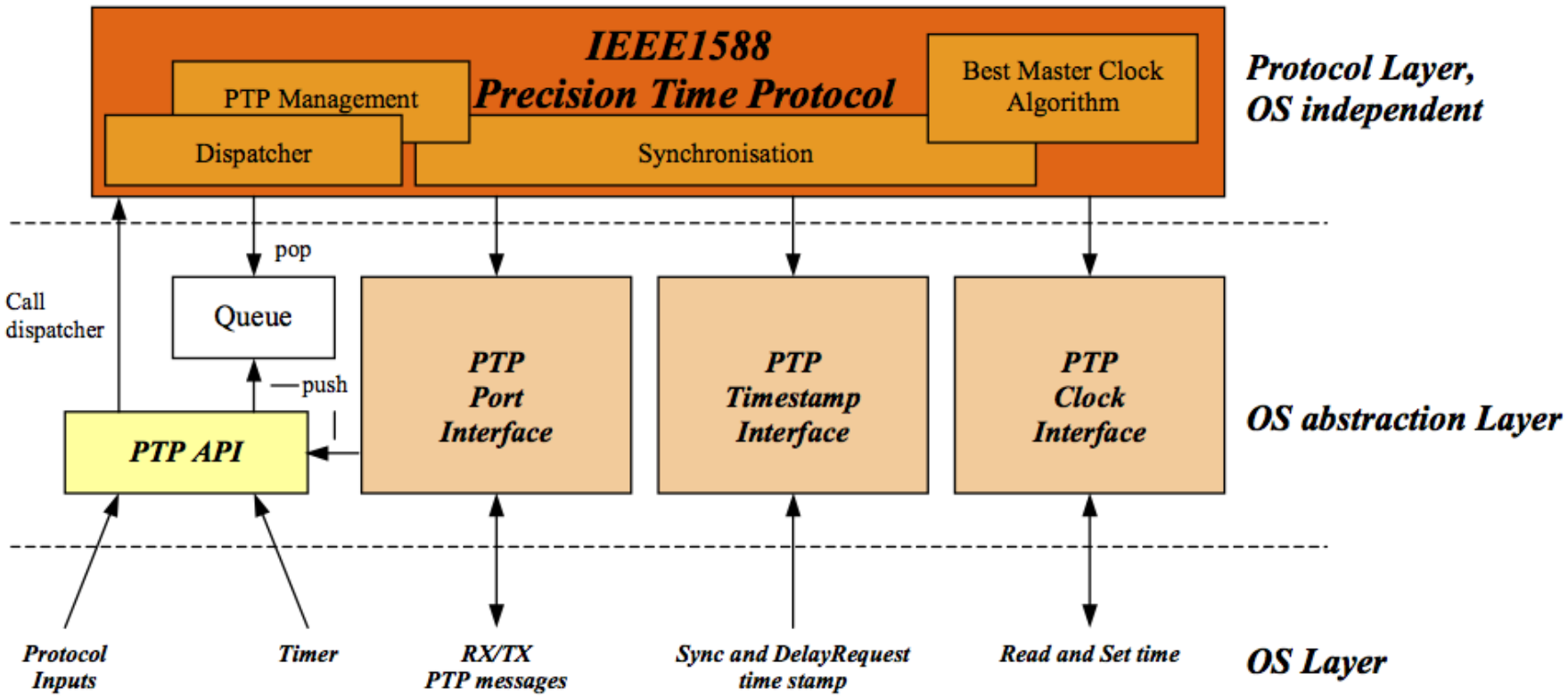
- High-precision real-time clock and a time stamp unit for generating the time stamp: hardware
- Protocol implementation: software



TSU - TimeStamp Unit

PTP: interaction diagram

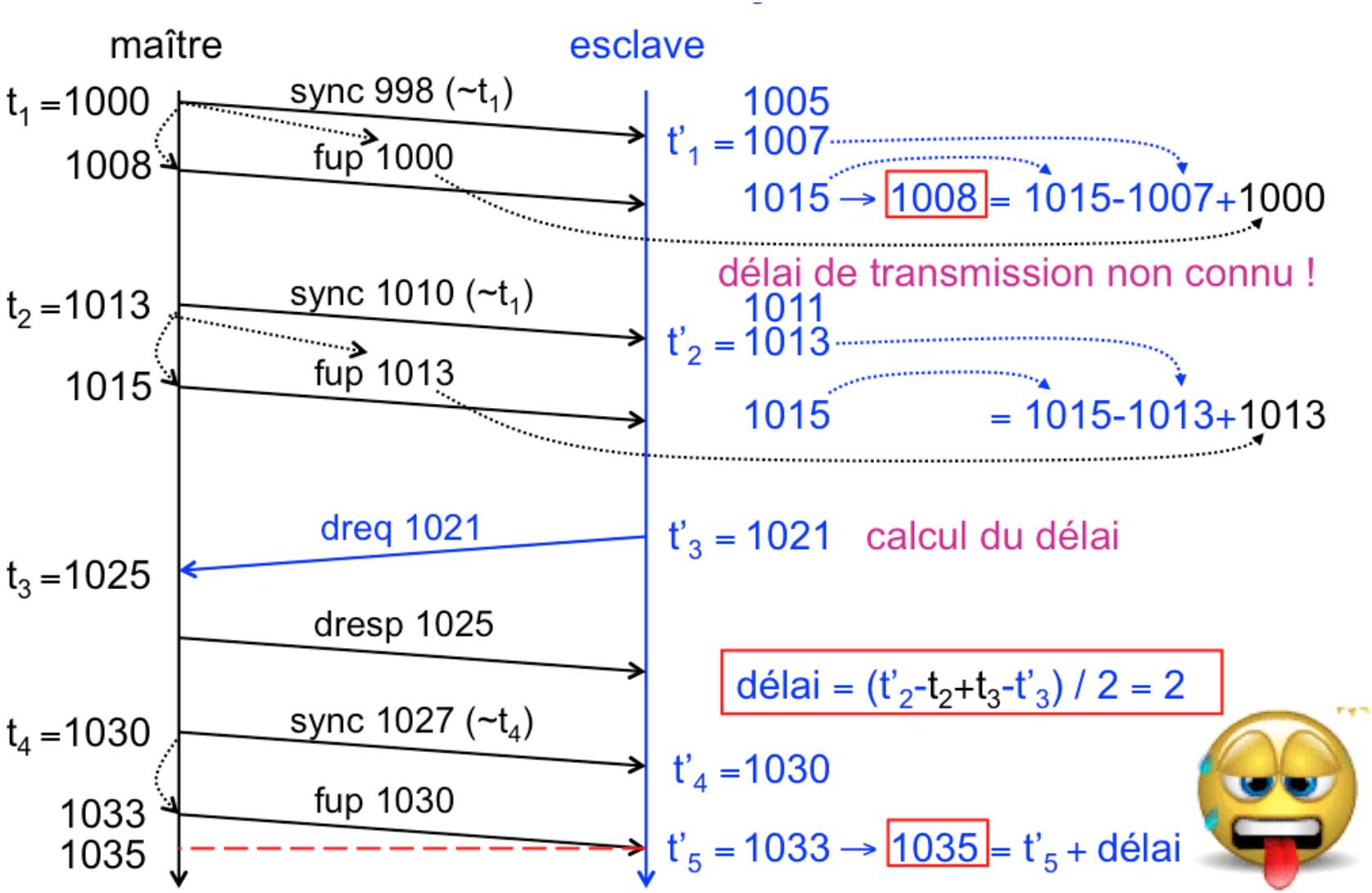
- Best Master Clock: automatic master selection by finding the best available master by comparing the properties (accuracy, stratum, drift, variance,...) of the communicating clocks.



PTP: the algorithm

- The time of the master is reported to the slave as accurately as possible
- Compensate all the processing times and run times
- The synchronisation is divided into two phases
- Clean synchronisation message SYNC: sent by the master at defined intervals (every 2 seconds for example) and contains the current time of the master
- However out of date when it leaves the master: procesing time, reading the clock, ...
- Second message Follow-up is sent to slave
- Upon reception, the slave computes the correction value (the offset)
- The slave corrects its clock
- However, transmission line delays are not considered
- The salve sends a Delay Request packet to the master
- The master generates a time stamp on receiving the packet and sends Delay Response packet
- The slave determines the transmission delay and adjusts its clock

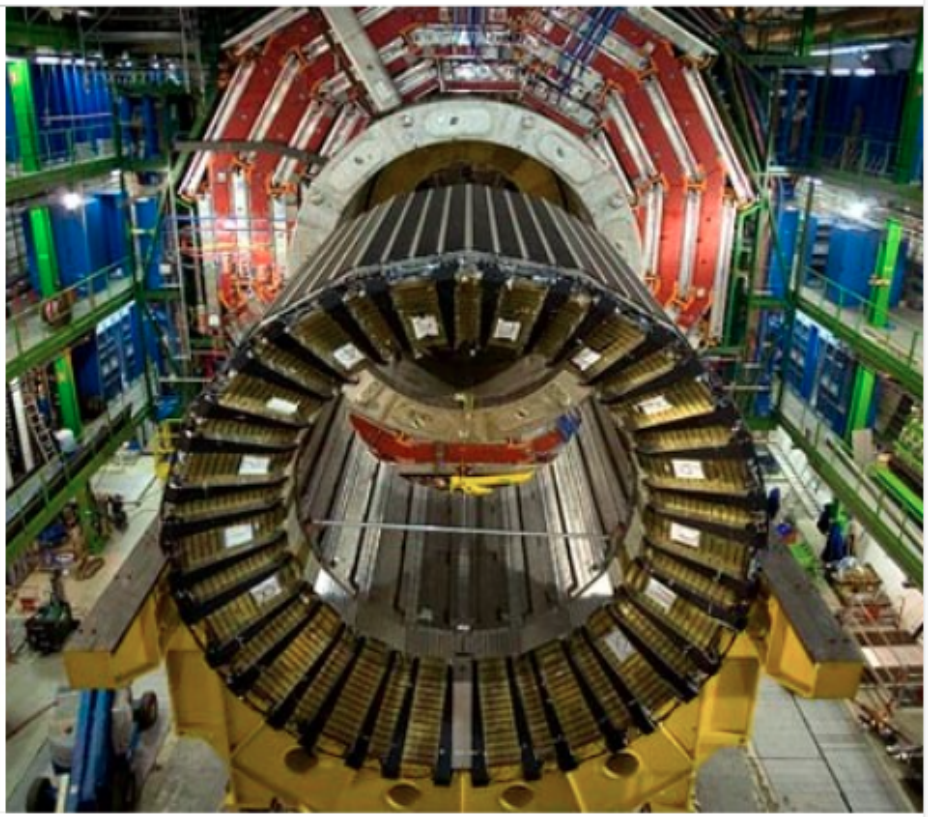
PTP: the algorithm



source: slides Gerard Berry, collège de France

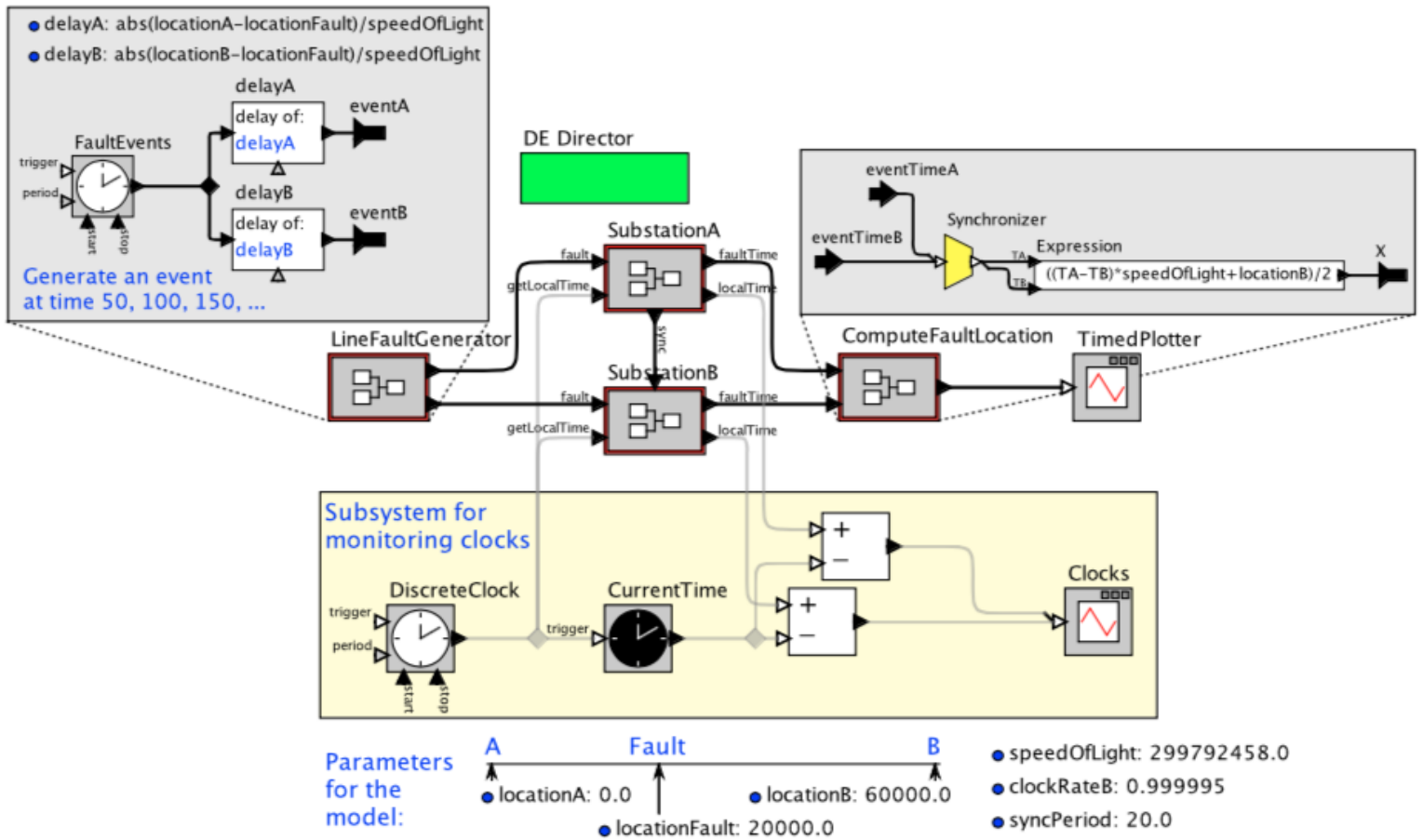
PTP: use cas

- The White Rabbit project at CERN claims to be able synchronize clocks on a network spanning several kilometers to under 100 picoseconds using IEEE 1588 PTP over Ethernet



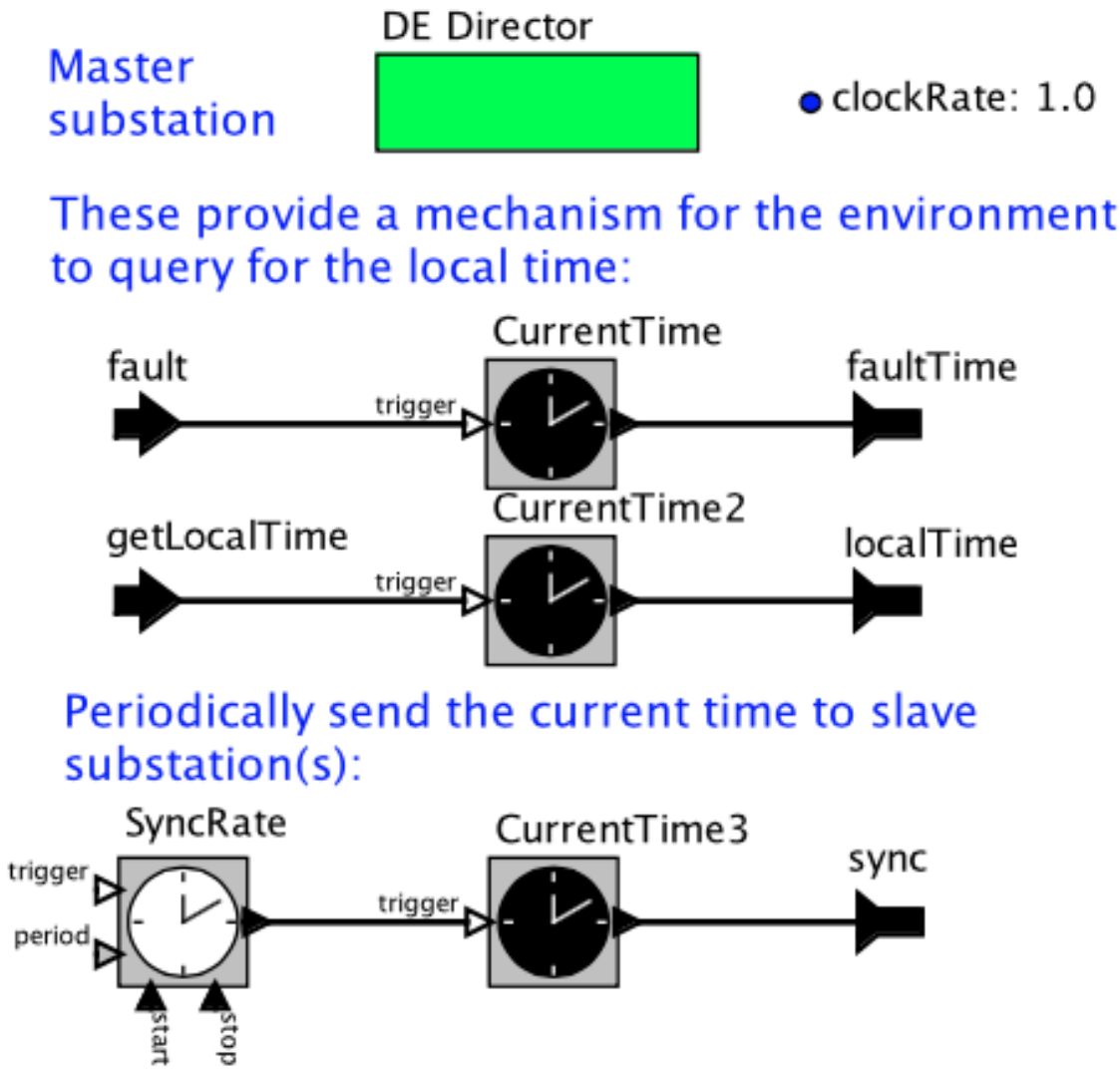
Clock Synchronisation: line fault detection

- Electric power systems: transmission lines may span many kilometers
- Fault occurs: finding the fault may be very expensive
- Estimate the location of the fault based on the time that the fault is observed at each end of the transmission line



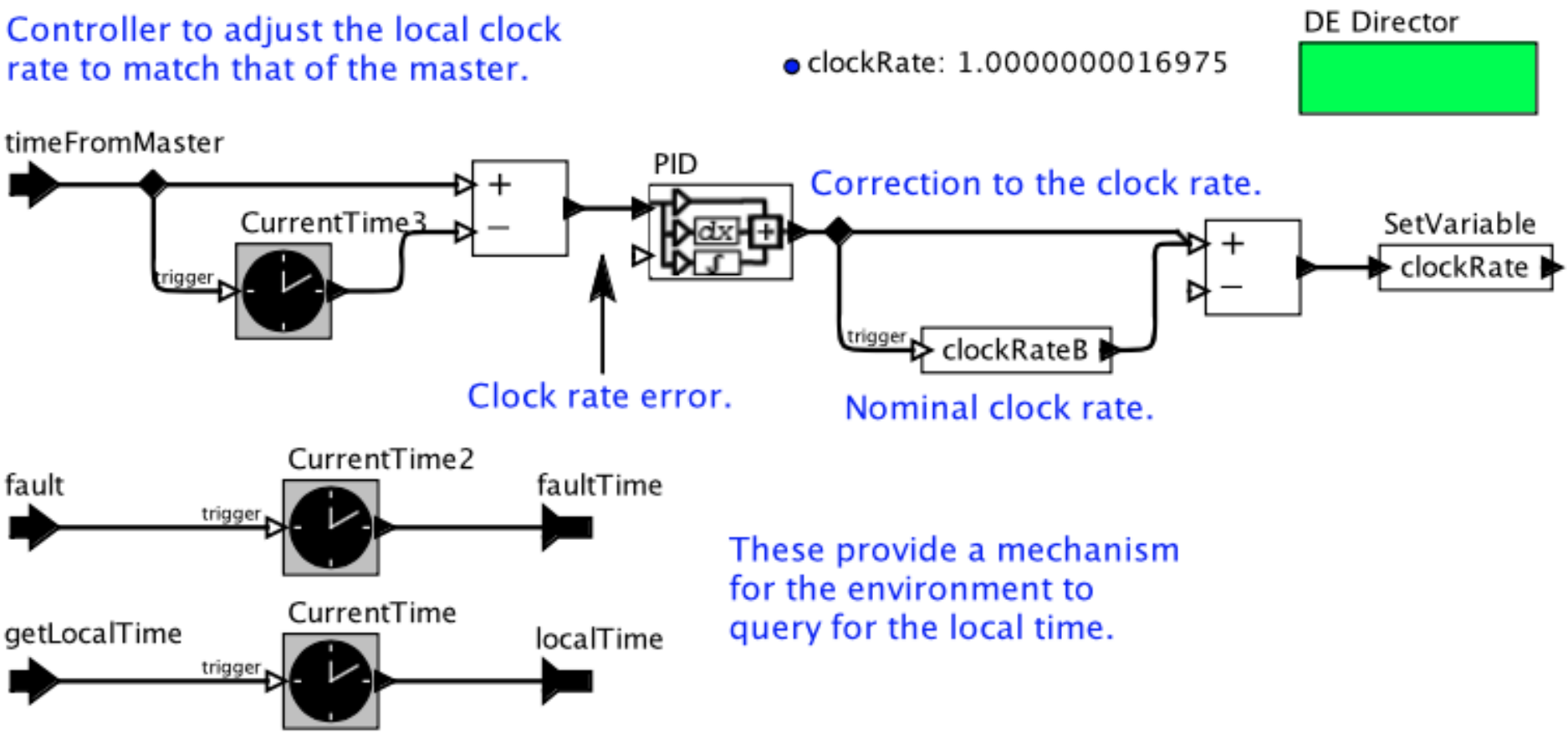
Clock Synchronisation: line fault detection

Substation A, the clock master



Clock Synchronisation: line fault detection

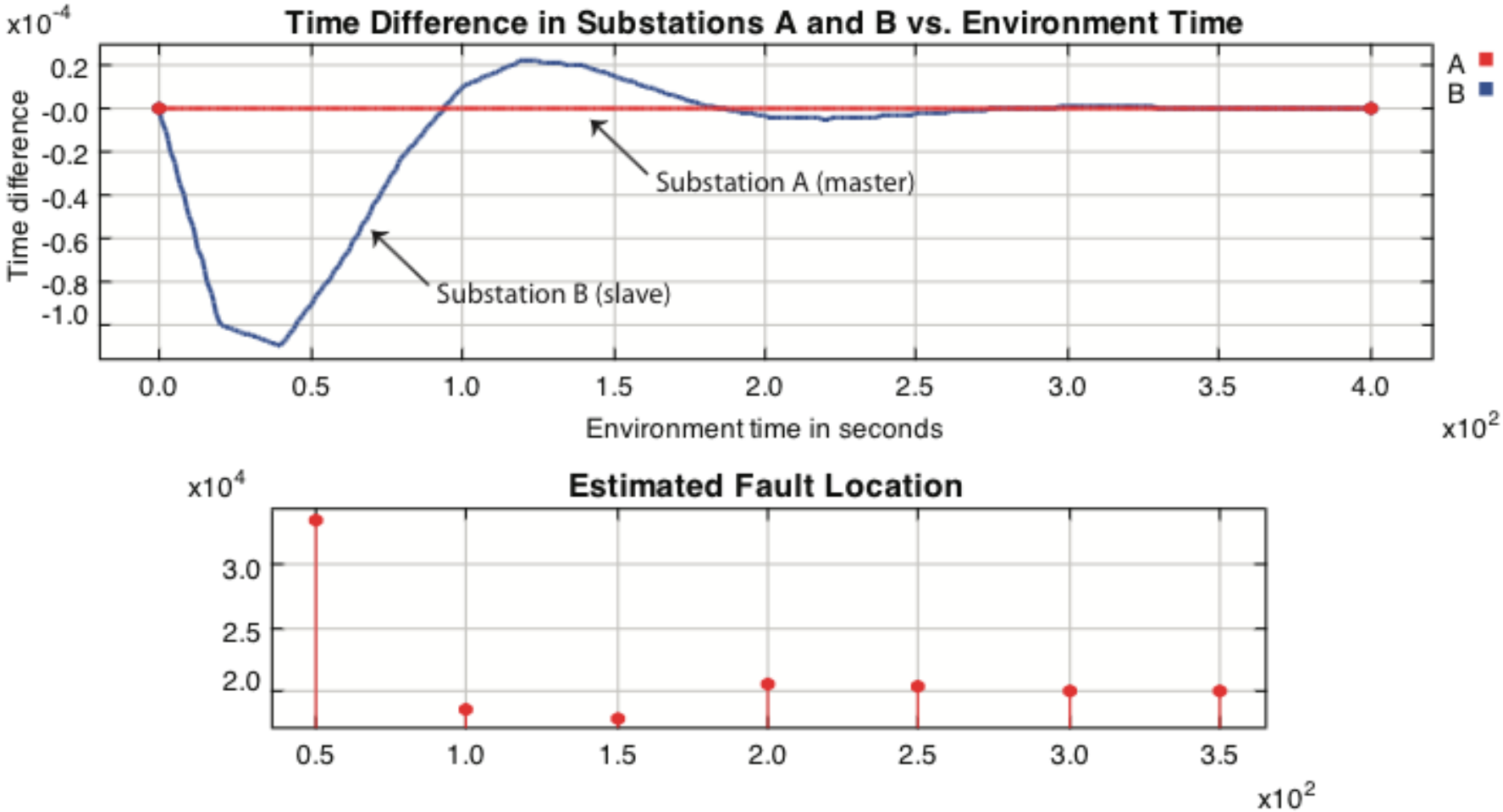
Substation B, the clock slave



Clock Synchronisation: line fault detection

- Let X denote the location of the fault event along the transmission line (the distance from the substation A). Then X satisfies the following equations
- $s \times (T_A - T_0) = X$
- $s \times (T_B - T_0) = D - X$
- T_0 is the time of fault (unknown)
- T_B the time of fault detection at substation B
- T_A is the time of fault detection at substation A
- s is the speed of propagation along the transmission line
- D is the distance from A to B
- $D - X$ is the distance from B to the fault
- $X = ((T_A - T_B) \times s + D)/2$
- The calculation is correct is the clocks of A and B are perfectly synchronized

Clock Synchronisation: line fault detection



Clock Synchronisation: Time-Triggered Protocol

- TTP was originally designed at the Vienna University of Technology in the early 1980s
- TTP is a dual-channel 25 Mbit/s time-triggered field bus. It can operate using one or both channels with maximum data rate of 2x 25 Mbit/s
- TTP offers fault-tolerant clock synchronization that establishes the global time base without relying on a central time serve
- TTP is often used in mission critical data communications applications: aircraft engine (FADEC), Thales Rail Signalling Solutions, cabin pressure control system (Airbus A380), electric and environmental control system of the Boeing 787 Dreamliner

Problems with Clock adjustment

- What problems can occur when a clock value is advanced from 171 to 174 ?
- What problems can occur when a clock value is moved back from 180 to 175 ?

Time and State of a Distributed System

Fundamental Goal: think about a system or an application

What do we need

- Define a state: for example to define predicates
- Define an order: to coordinate the activities

Why is it harder for Distributed Systems? (Inherent Limitations)

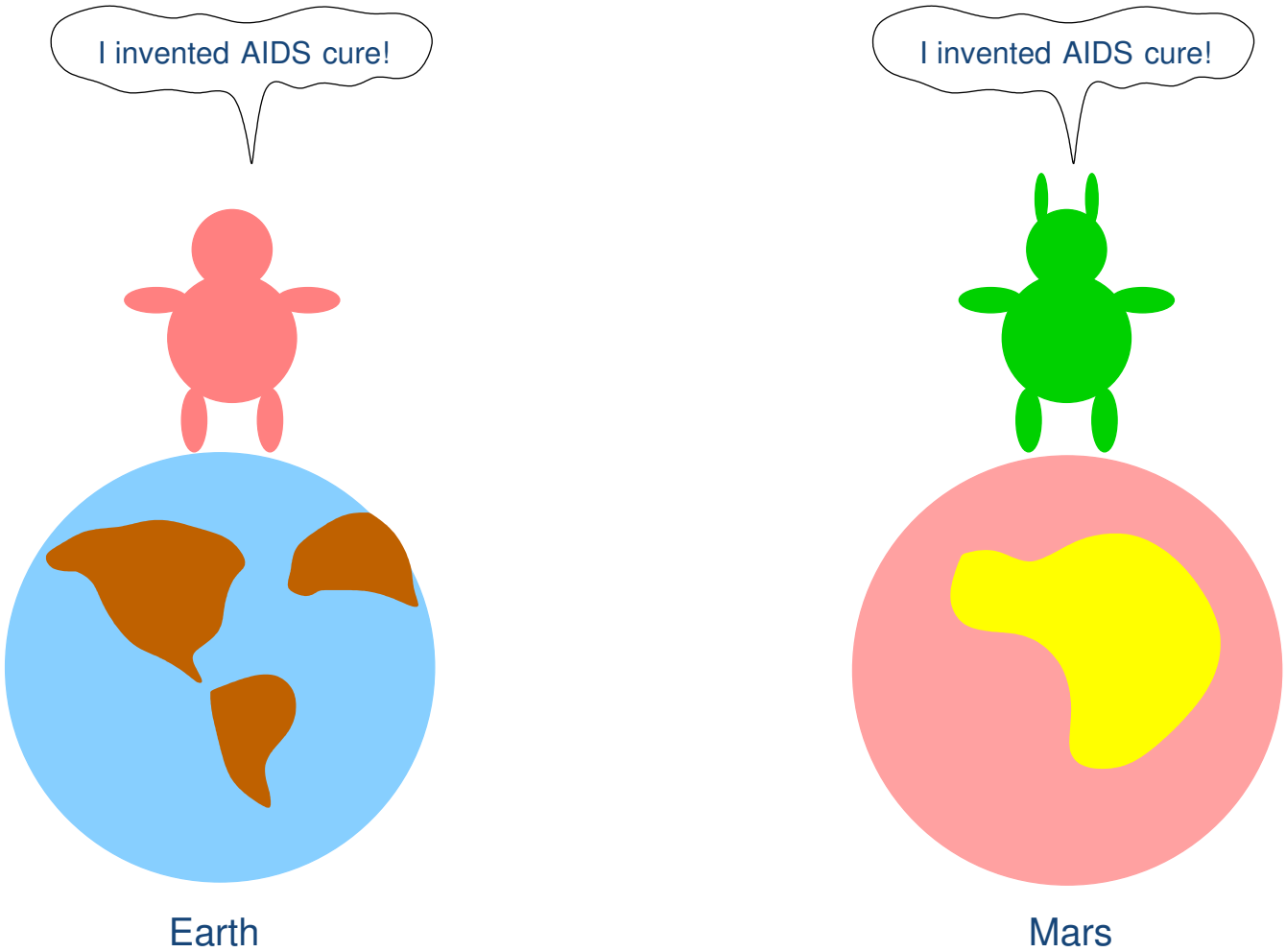
- Absence of Global Clock: There is no common notion of time
- Absence of Shared Memory: No process has up-to-date global knowledge
- Asynchronous communications and computations (generally speaking)
  - Ie, comm/comp time has no maximum
  - Because dynamically changing load and resources not exclusively allocated
  - Synchronous systems (real time, phone) more rare because more expansive

Goal now

- Define an order relation (used later for global state)
- At the end, that's quite simple, but it needed several years of research

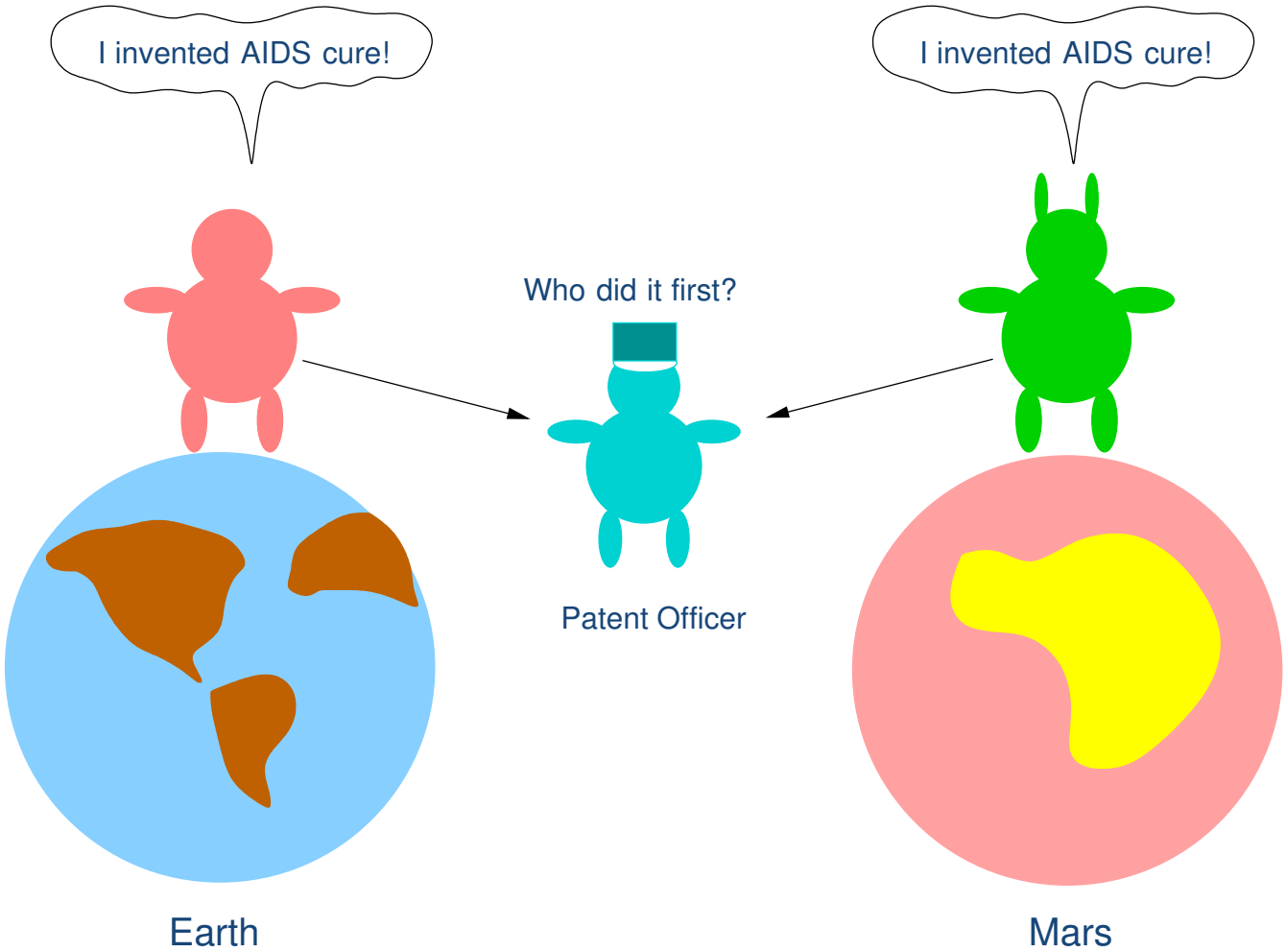
Absence of Global Clock

Different processes may have different notions of time



Absence of Global Clock

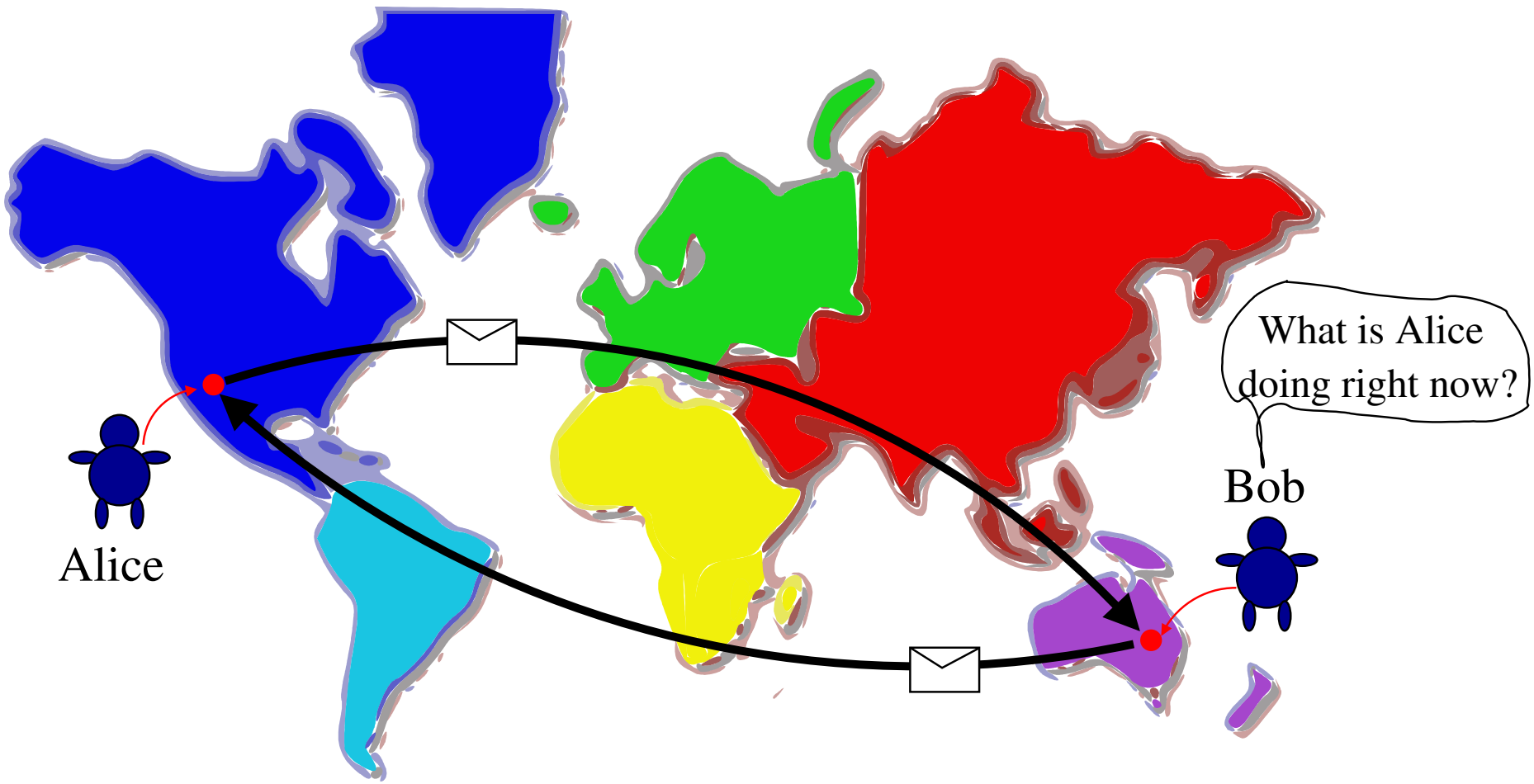
Different processes may have different notions of time



- Problem: How do we order events on different processes?

Absence of Shared Memory

A process does not know current state of other processes

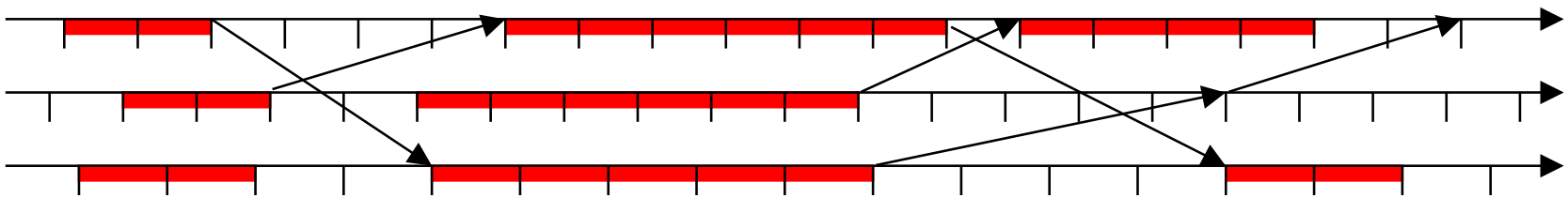


- Problem: How do we obtain a coherent view of the system?

The Reliable Asynchronous Model

That's the weaker (reliable) model

- Very strong constraints from the system
  - No upper bound on communication or computation
  - Algorithms working here work also in more friendly models
  - Models made more friendly by removing constraints (setup upper bounds)
  - (that's not the worst model: it is reliable)
- This model is often used for Bounding costs or Impossibility results



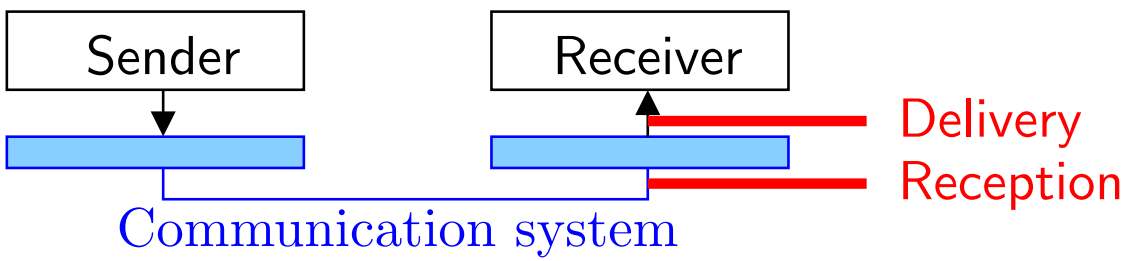
- Each site has a clock (not synchronized, with relative drifts)
- Processes only communicate by message exchanges
- Possible events:
  - Local (process internal state change)
  - Emission or Reception of messages

About messages

Properties of the communication system

1. No loss: Every sent message arrives (no upper bound on transit time)
   - How to achieve this: failure detection (with timeout) and resending
2. Messages are not altered
   - How to achieve this: Mechanisms for detection and correction of errors
3. FIFO channel between processes
   - How: message numbering
   - Assumption sometimes removed ($\Rightarrow$ even harder)

Distinguish message reception and delivering



Distributed Execution

Definition: Distributed algorithm

A distributed algorithm is a collection of distributed automata, one per process

Definition: Distributed execution

The execution of a distributed algorithm is a sequence of events executed by the processes

- Partial execution: a finite sequence of events
- Infinite execution: an infinite sequence of events

Possible events

- send(m,p): sends a message m to process p
- receive(m): receives a message m
- local events that change the local state

Process Execution and Synchronization

Process Execution

- That's a suite of events (its history, its trace)

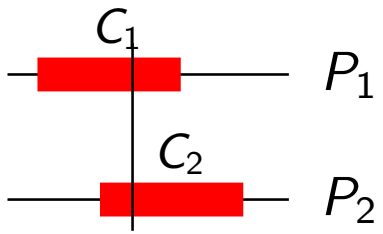
  Recall: kind of possible events = {local, sending, receiving}
- Suite ordered by the local clock
- For P_1: e_1^1, e_1^2, e_1^3, e_1^4, $\ldots e_1^k$, $\ldots$

"Synchronizing processes" ?!

$\rightsquigarrow$ force an order to the events of these processes

- Example: mutual exclusion

$$\text{either} \begin{cases} end(C_2) \textbf{ precedes } begin(C_1) \\ end(C_1) \textbf{ precedes } begin(C_2) \end{cases}$$



When is it possible to order two events?

Causality Principle

- The Cause comes before the Effect

Three Cases:

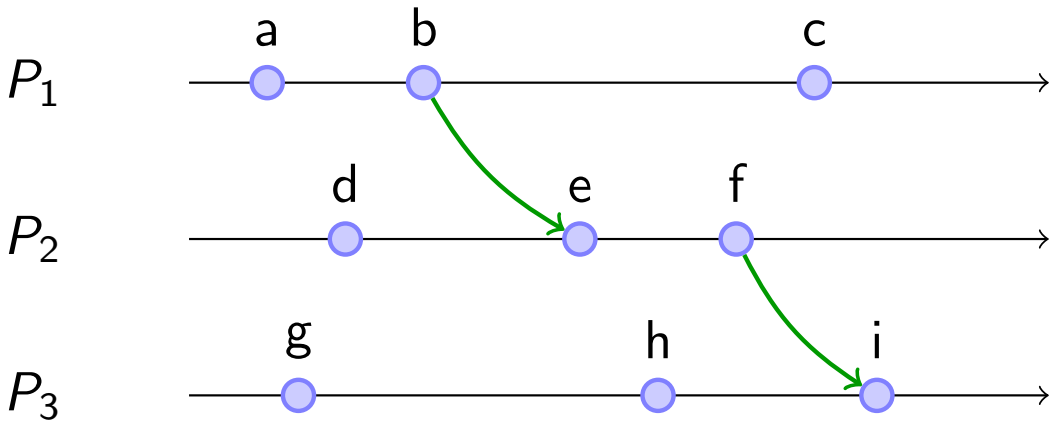
1. Events executed on the same process:
   - if e and f are events on the same process and e occurred before f, then e *happened-before* f
2. Communication events of the same message:
   - if e is the send event of a message and f is the receive event of the same message, then e *happened-before* f
3. Events related by transitivity:
   - if event e happened-before event g and event g happened-before event f, then e *happened-before* f

Happened-Before Relation

Notation

- Happened-before relation is denoted by $\rightarrow$
- $\rightarrow$ is only partial-order: some events are unrelated

Illustration



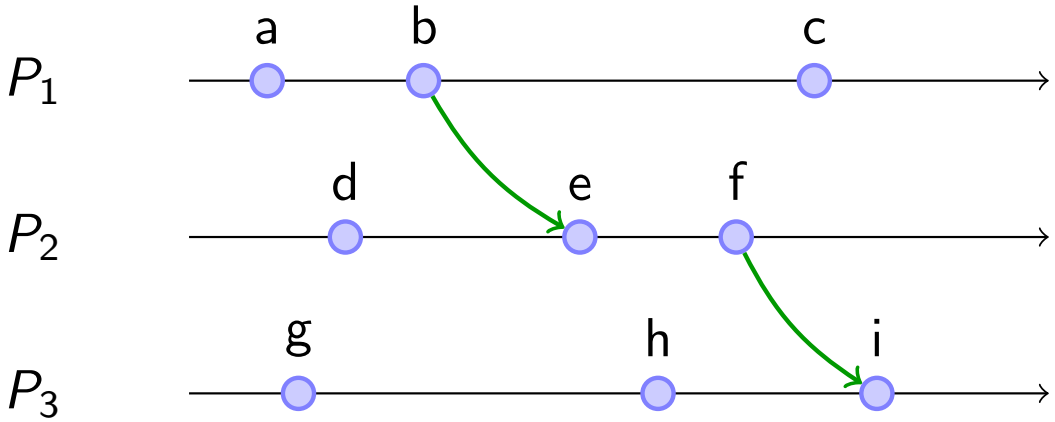
- Events on the same process
  $a \rightarrow b$, $b \rightarrow c$, $d \rightarrow f$
- Events of the same message
  $b \rightarrow e$, $f \rightarrow i$
- Transitivity
  $a \rightarrow c$, $a \rightarrow e$, $a \rightarrow i$

Happened-Before Relation

Notation

- Happened-before relation is denoted by $\rightarrow$
- $\rightarrow$ is only partial-order: some events are unrelated

Illustration



- Events on the same process
  $a \rightarrow b$, $b \rightarrow c$, $d \rightarrow f$
- Events of the same message
  $b \rightarrow e$, $f \rightarrow i$
- Transitivity
  $a \rightarrow c$, $a \rightarrow e$, $a \rightarrow i$

Concurrent events

- Events not related by the happened-before relation
- Concurrency relation is denoted by $\|$
- Examples: $a \| d$, $e \| h$, $c \| i$,
- Concurrency is not transitive: $a \| d$ and $d \| c$ but $a \not\| c$

Meaning of Happened-Before Relation

If $e \rightarrow e^{'}$, this means that we can find a series of events e^1, e^2, e^3, e^4, ... e^n, where $e^1 = e$ and $e^n = e^{'}$, such that for each pair of consecutive events e^i and e^{i+1}:

- e^i and e^{i+1} are executed on the same process, in this order
- e^i = send(m,*) and e^{i+1}= receive(m)

Notes

- happen-before captures the concept of potential causal ordering
- happend-before capture a flow of data between two events
- Two events, e and $e^{'}$ that are not related by the happen-before relation ($e \not\rightarrow e^{'} \wedge e^{'} \not\rightarrow e$) are concurrent, and we write $e \parallel e$

Reality check

Memory model of Java

The multi-threaded memory model of Java is based on the happen-before relation, where communication between threads is based on the acquisition and release of locks. Chapter 17 of the Java Language Specification : `http://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4.5`

Premier chapitre

Theoretical foundations

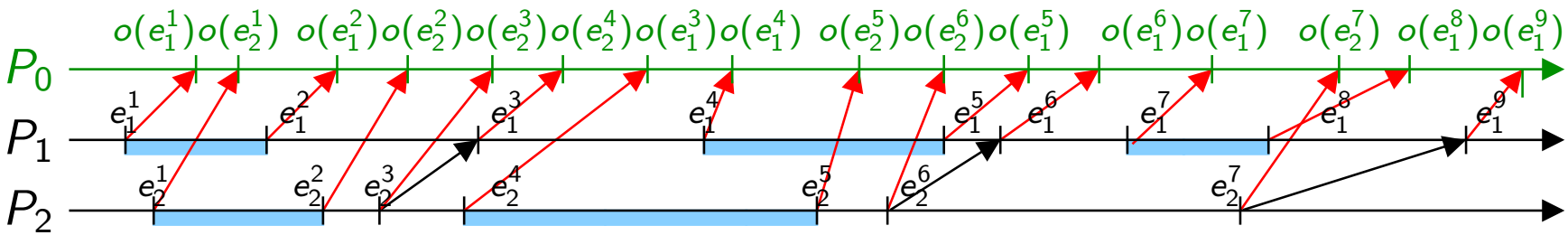
- Time and State of a Distributed System
- Ordering of events
- Abstract Clocks
  - Global Observer
  - Logical Clocks
  - Vector Clocks

Dating System (for sake of global ordering)

Goal: Dating System compatible with Causality

First Approach: notion of observation

- A "observer" process P_0 is informed by message of every event
- The suite of events as observed by P_0 is a global observation
- Later: each process is observer, and observations match



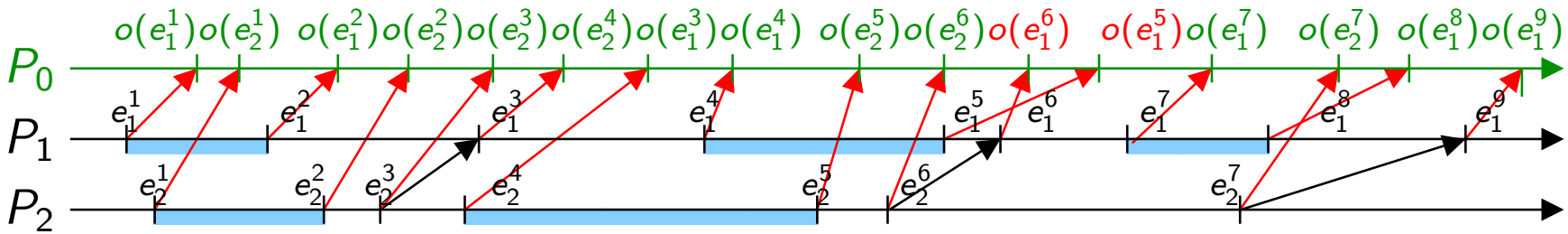
Validity of Observations

Definition

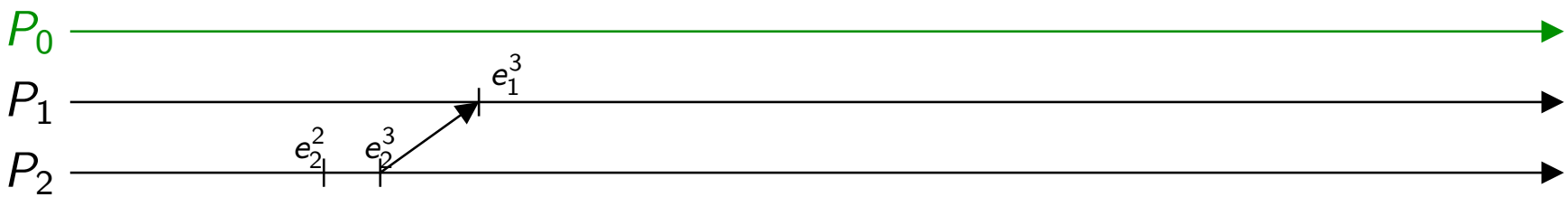
- Observation said valid iff $(e \rightarrow f) \Rightarrow (o(e) \rightarrow o(f))$

Examples

1. $(e_1^5 \rightarrow e_1^6)$ but $o(e_1^6)$ precedes $o(e_1^5)$



2. $(e_2^2 \rightarrow e_1^3)$ because $(e_2^2 \rightarrow e_2^3)$ and $(e_2^3 \rightarrow e_1^3)$



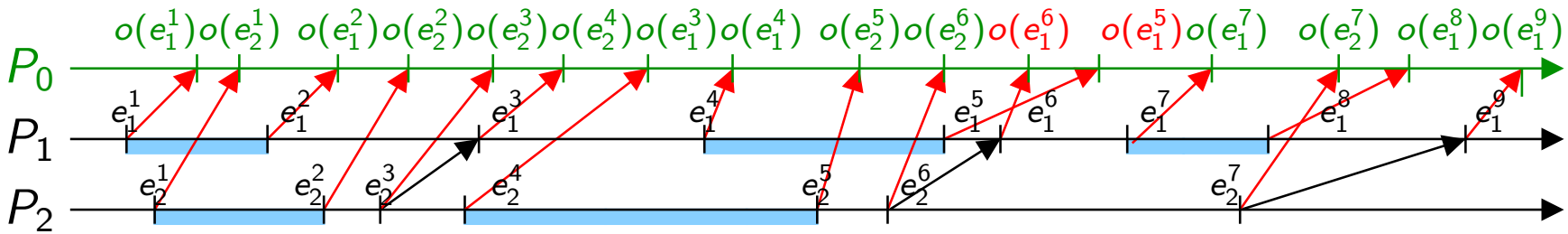
Validity of Observations

Definition

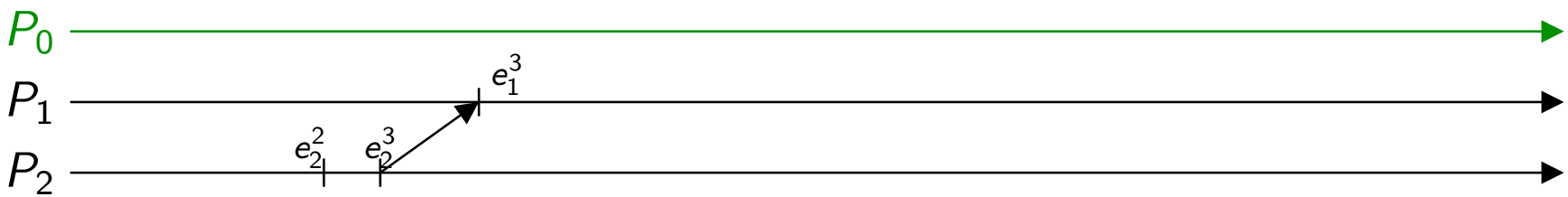
- Observation said valid iff $(e \rightarrow f) \Rightarrow (o(e) \rightarrow o(f))$

Examples

1. $(e_1^5 \rightarrow e_1^6)$ but $o(e_1^6)$ precedes $o(e_1^5)$ ($P_1 \rightarrow P_0$ is not FIFO)



2. $(e_2^2 \rightarrow e_1^3)$ because $(e_2^2 \rightarrow e_2^3)$ and $(e_2^3 \rightarrow e_1^3)$



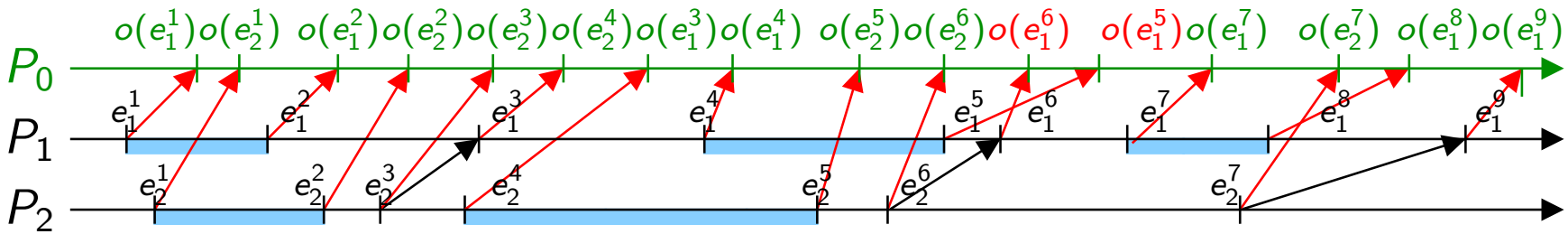
Validity of Observations

Definition

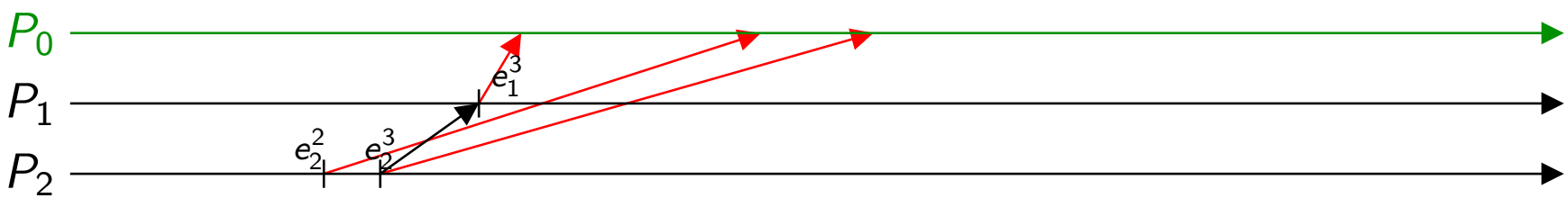
- Observation said valid iff $(e \rightarrow f) \Rightarrow (o(e) \rightarrow o(f))$

Examples

1. $(e_1^5 \rightarrow e_1^6)$ but $o(e_1^6)$ precedes $o(e_1^5)$ ($P_1 \rightarrow P_0$ is not FIFO)



2. $(e_2^2 \rightarrow e_1^3)$ because $(e_2^2 \rightarrow e_2^3)$ and $(e_2^3 \rightarrow e_1^3)$
   but $o(e_1^3)$ can not precede $o(e_2^2)$ even if channels are fifo



Abstract Clocks

Setting up an observer is suboptimal

- Expensive: A huge amount of messages must be sent to the observer
- Not robust: What if the observer fails?
- Not reliable: invalid observations are still possible

Abstract Clocks

- Why: (try to) solve absence of global clock
- How: processes timestamp events locally so that they get globally ordered

Different kind of abstract clocks

- Each offers differing abilities, associated to differing complexities
- Logical clock: used to totally order all events
- Vector Clocks: used to track happened-before relation
- Matrix Clocks: used to track what other processes know about other processes
- Direct Dependency Clocks: used to track direct causal dependencies

Logical Clocks (or Lamport's Clock)

General idea

- Implements the notion of virtual time
- Can be used to totally order all events
- Assigns timestamp $C(e)$ to each event e
- Compute $C(e)$ in a way that is consistent with the happened-before relation:

$$\boxed{e \rightarrow f \Rightarrow C(e) < C(f)}$$

- (Note that this is $\Rightarrow$, not $\Leftrightarrow$)

Time, Clocks and the Ordering of Events in a Distributed System, Leslie Lamport, 1978.

Implementing Logical Clocks

- Each process i has a local scalar counter C_i ($\in \mathbb{N}$)
- Each even e local to i is dated by the current value of C_i
- Each message m sent from i is also annoted with C_i (sending time)

Computation rules on process i

Initialization :	$C_i \leftarrow 0$
Local event :	$C_i += 1$
Sending message (m) :	$C_i += 1$ then send (m, C_i)
Receiving message (m, E_m):	$C_i \leftarrow max(C_i, E_m) + 1$

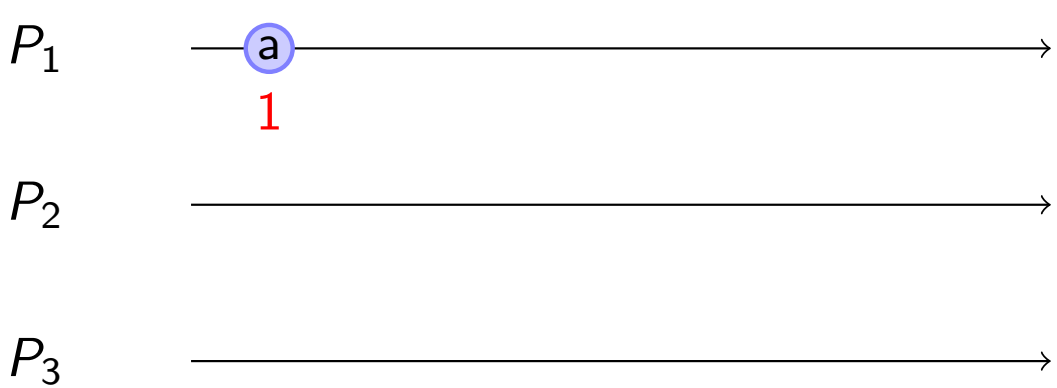
Implementing Logical Clocks

- Each process i has a local scalar counter C_i ($\in \mathbb{N}$)
- Each even e local to i is dated by the current value of C_i
- Each message m sent from i is also annoted with C_i (sending time)

Computation rules on process i

Initialization :	$C_i \leftarrow 0$
Local event :	$C_i += 1$
Sending message (m) :	$C_i += 1$ then send (m, C_i)
Receiving message (m, E_m):	$C_i \leftarrow max(C_i, E_m) + 1$

Example



- a first event of $P_1 \rightsquigarrow C(a) = 1$

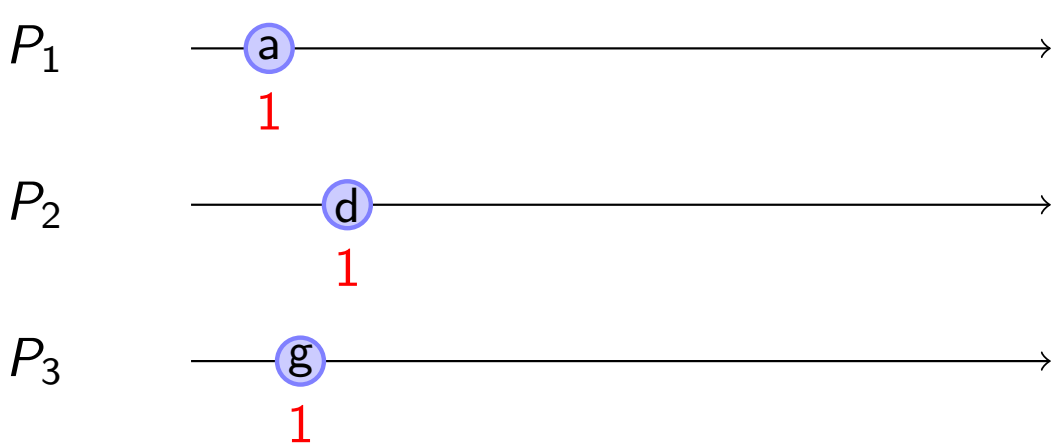
Implementing Logical Clocks

- Each process i has a local scalar counter C_i ($\in \mathbb{N}$)
- Each even e local to i is dated by the current value of C_i
- Each message m sent from i is also annoted with C_i (sending time)

Computation rules on process i

Initialization :	$C_i \leftarrow 0$
Local event :	$C_i += 1$
Sending message (m) :	$C_i += 1$ then send (m, C_i)
Receiving message (m, E_m):	$C_i \leftarrow max(C_i, E_m) + 1$

Example



- a first event of $P_1 \rightsquigarrow C(a) = 1$

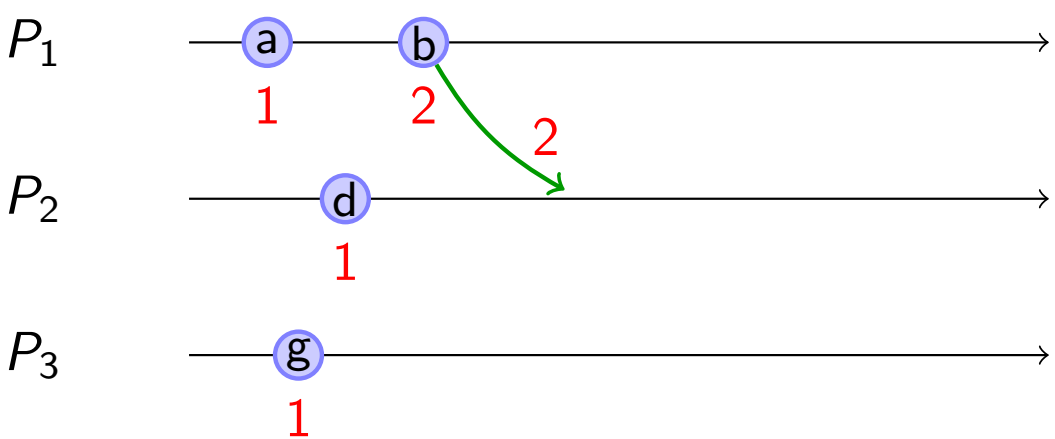
Implementing Logical Clocks

- Each process i has a local scalar counter C_i ($\in \mathbb{N}$)
- Each even e local to i is dated by the current value of C_i
- Each message m sent from i is also annoted with C_i (sending time)

Computation rules on process i

Initialization :	$C_i \leftarrow 0$
Local event :	$C_i += 1$
Sending message (m) :	$C_i += 1$ then send (m, C_i)
Receiving message (m, E_m):	$C_i \leftarrow max(C_i, E_m) + 1$

Example



- a first event of $P_1 \rightsquigarrow C(a) = 1$
- b: send $\rightsquigarrow C_1 := C_1 + 1$; send 2

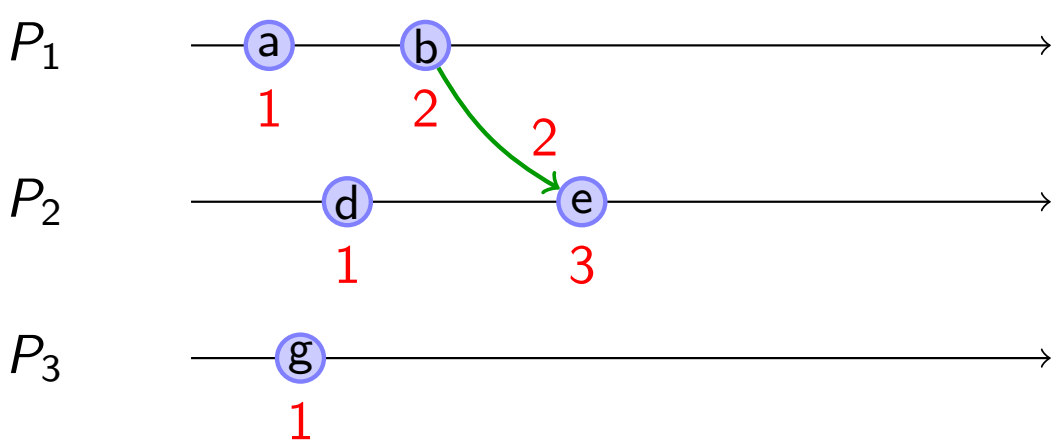
Implementing Logical Clocks

- Each process i has a local scalar counter C_i ($\in \mathbb{N}$)
- Each even e local to i is dated by the current value of C_i
- Each message m sent from i is also annoted with C_i (sending time)

Computation rules on process i

Initialization :	$C_i \leftarrow 0$
Local event :	$C_i += 1$
Sending message (m) :	$C_i += 1$ then send (m, C_i)
Receiving message (m, E_m):	$C_i \leftarrow max(C_i, E_m) + 1$

Example



- a first event of $P_1 \rightsquigarrow C(a) = 1$
- b: send $\rightsquigarrow C_1 := C_1 + 1$; send 2
- e:recv $C(e) = max(1, 2) + 1 = 3$

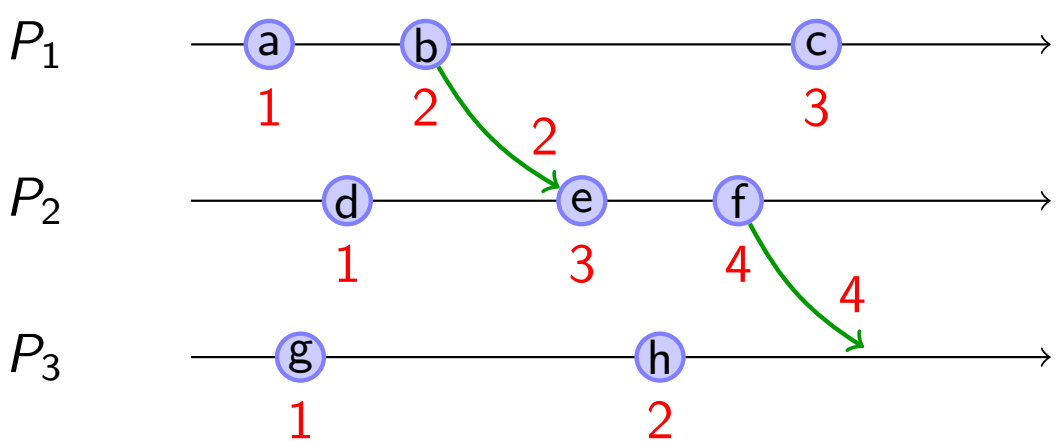
Implementing Logical Clocks

- Each process i has a local scalar counter C_i ($\in \mathbb{N}$)
- Each even e local to i is dated by the current value of C_i
- Each message m sent from i is also annoted with C_i (sending time)

Computation rules on process i

Initialization :	$C_i \leftarrow 0$
Local event :	$C_i += 1$
Sending message (m) :	$C_i += 1$ then send (m, C_i)
Receiving message (m, E_m):	$C_i \leftarrow max(C_i, E_m) + 1$

Example



- a first event of $P_1 \rightsquigarrow C(a) = 1$
- b: send $\rightsquigarrow C_1 := C_1 + 1$; send 2
- e:recv $C(e) = max(1, 2) + 1 = 3$
- c, h: local events; f: send

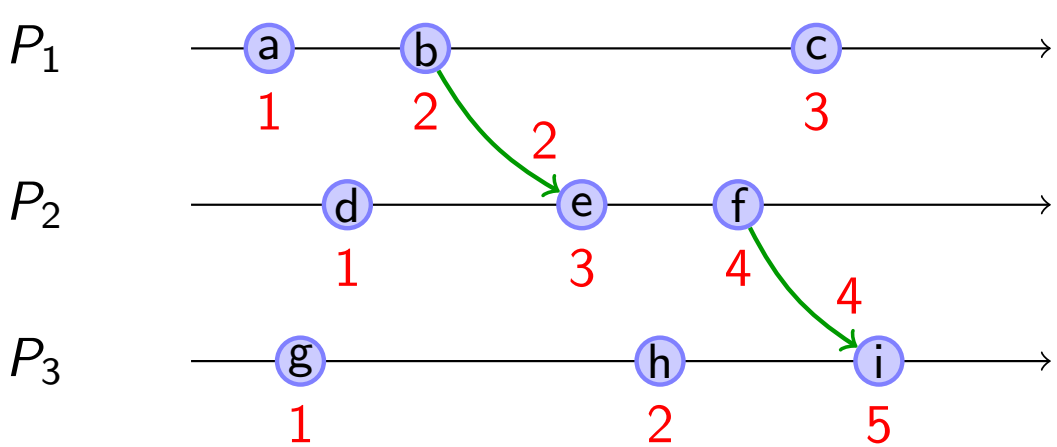
Implementing Logical Clocks

- Each process i has a local scalar counter C_i ($\in \mathbb{N}$)
- Each even e local to i is dated by the current value of C_i
- Each message m sent from i is also annoted with C_i (sending time)

Computation rules on process i

Initialization :	$C_i \leftarrow 0$
Local event :	$C_i += 1$
Sending message (m) :	$C_i += 1$ then send (m, C_i)
Receiving message (m, E_m):	$C_i \leftarrow max(C_i, E_m) + 1$

Example



- a first event of $P_1 \rightsquigarrow C(a) = 1$
- b: send $\rightsquigarrow C_1 := C_1 + 1$; send 2
- e:recv $C(e) = max(1, 2) + 1 = 3$
- c, h: local events; f: send
- i:recv; $C(i) = max(4, 2) + 1 = 5$

Conclusion on Logical Clocks

Possible Applications

- Distributed waiting queue (mutual exclusion; replicas update)
- Determine least access (cache coherence, DSM)

Conclusion on Logical Clocks

Possible Applications

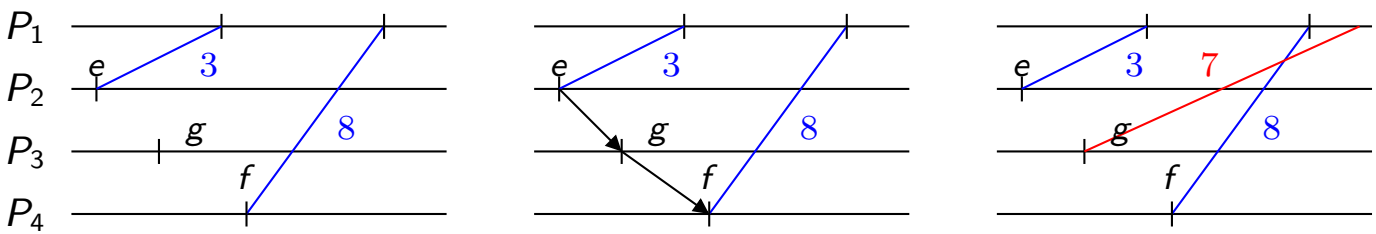
- Distributed waiting queue (mutual exclusion; replicas update)
- Determine least access (cache coherence, DSM)

Limits of the Logical Clocks

- Cannot be used to determine events concurrency

$$\boxed{(e \parallel f) \text{ does not imply } (C(e) = C(f))}$$

- Some missing events may go undetected:
  - If $C(e) < C(f)$, is there any g so that $e \rightarrow g \rightarrow f$?
  - Impossible to answer with logical clocks only



P_3 will synchronize later | P_3 synchronized | P_3 desynchronized!

Conclusion on Logical Clocks

Possible Applications

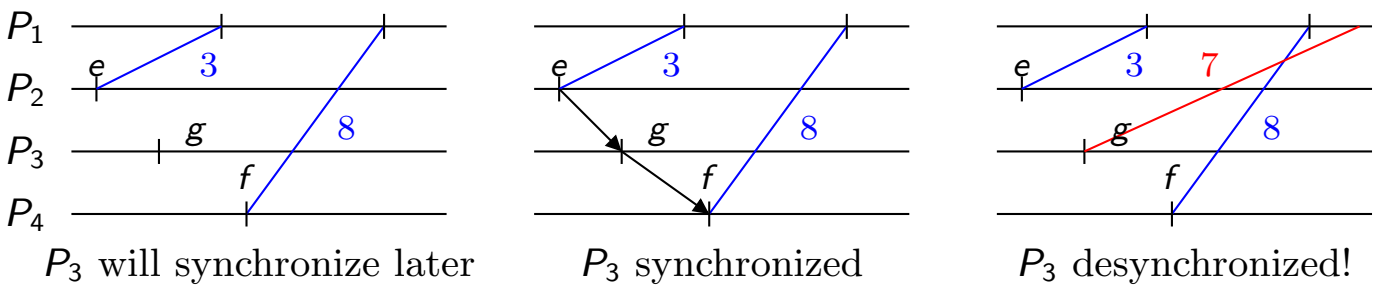
- Distributed waiting queue (mutual exclusion; replicas update)
- Determine least access (cache coherence, DSM)

Limits of the Logical Clocks

- Cannot be used to determine events concurrency

$$\boxed{(e \parallel f) \text{ does not imply } (C(e) = C(f))}$$

- Some missing events may go undetected:
  - If $C(e) < C(f)$, is there any g so that $e \rightarrow g \rightarrow f$?
  - Impossible to answer with logical clocks only



P_3 will synchronize later — P_3 synchronized — P_3 desynchronized!

- All is because $\boxed{e \rightarrow f \Rightarrow C(e) < C(f)}$ is no $\Leftrightarrow$

Vector Clocks

General idea

- Captures the happened-before relation
- Assigns timestamp to each events such that

$$\boxed{e \rightarrow f \Leftrightarrow C(e) < C(f)}$$

- Like the name says, values $C(e)$ are not scalars but vectors ($\in \mathbb{N}^{\#processes}$)
  $V_i[j]$: What i knows of the clock of j

Vector Clocks

General idea

- Captures the happened-before relation
- Assigns timestamp to each events such that

$$\boxed{e \rightarrow f \Leftrightarrow C(e) < C(f)}$$

- Like the name says, values $C(e)$ are not scalars but vectors ($\in \mathbb{N}^{\#processes}$)
  $V_i[j]$: What i knows of the clock of j

Comparing two vectors: component-wise

- Equality: $V = W$ iff $\forall i, V_i = W_i$
- Comparison: $V < W$ iff $\forall i, V_i \leq W_i$ and $\exists i, V_i < W_i$
- Examples: $\begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} < \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}$ and $\begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} < \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$ but $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \not< \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$

Implementing Vector Clocks

- Each process i has a local scalar vector C_i $(\in \mathbb{N}^{\#processes})$

Computation rules on process i

Initialization :	$C_i \leftarrow \{0, ..., 0\}$
Local event :	$C_i[i] += 1$
Sending message (m) :	$C_i[i] += 1$ then send (m, C_i)
Receiving message (m, E_m):	$\forall k, C_i[k] \leftarrow max(C_i[k], E_m[k])$
	$C_i[i] += 1$

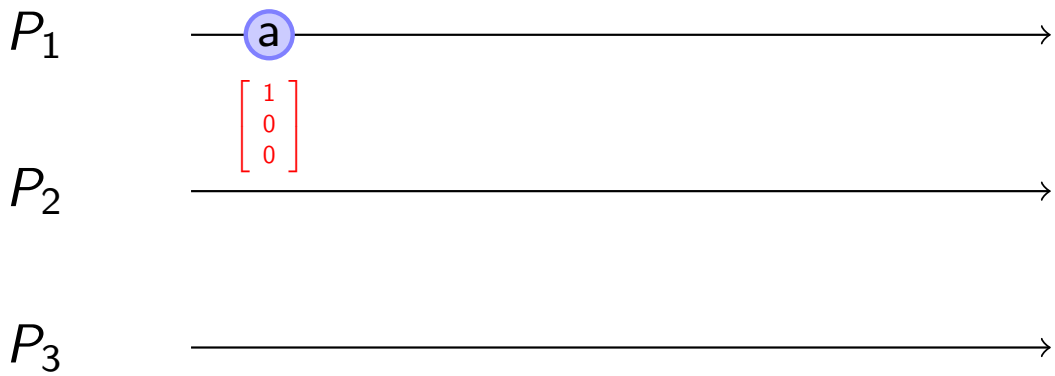
Implementing Vector Clocks

- Each process i has a local scalar vector C_i $(\in \mathbb{N}^{\#processes})$

Computation rules on process i

Initialization :	$C_i \leftarrow \{0, ..., 0\}$
Local event :	$C_i[i] += 1$
Sending message (m) :	$C_i[i] += 1$ then send (m, C_i)
Receiving message (m, E_m):	$\forall k, C_i[k] \leftarrow max(C_i[k], E_m[k])$
	$C_i[i] += 1$

Example



- a first event of $P_1 \rightsquigarrow C(a) = \begin{bmatrix}1\\0\\0\end{bmatrix}$

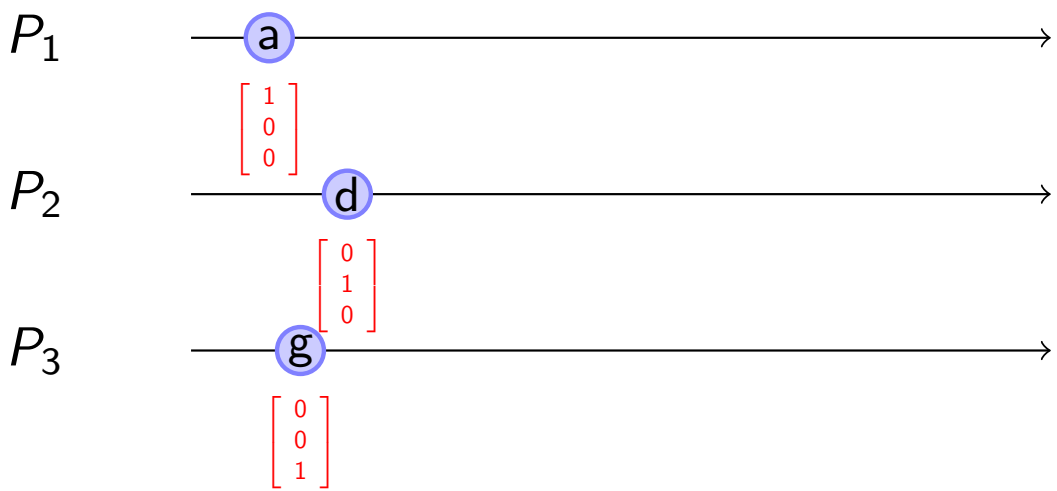
Implementing Vector Clocks

- Each process i has a local scalar vector C_i ($\in \mathbb{N}^{\#processes}$)

Computation rules on process i

Initialization :	$C_i \leftarrow \{0, ..., 0\}$
Local event :	$C_i[i] += 1$
Sending message (m) :	$C_i[i] += 1$ then send (m, C_i)
Receiving message (m, E_m):	$\forall k, C_i[k] \leftarrow max(C_i[k], E_m[k])$
	$C_i[i] += 1$

Example



- a first event of $P_1 \rightsquigarrow C(a) = \begin{bmatrix}1\\0\\0\end{bmatrix}$

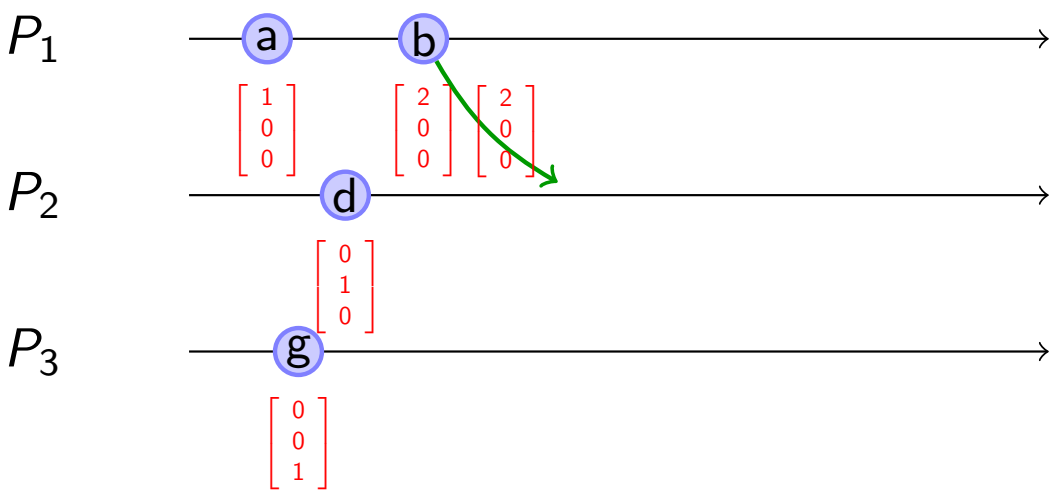
Implementing Vector Clocks

- Each process i has a local scalar vector C_i $(\in \mathbb{N}^{\#processes})$

Computation rules on process i

Initialization :	$C_i \leftarrow \{0, ..., 0\}$
Local event :	$C_i[i] += 1$
Sending message (m) :	$C_i[i] += 1$ then send (m, C_i)
Receiving message (m, E_m):	$\forall k, C_i[k] \leftarrow max(C_i[k], E_m[k])$
	$C_i[i] += 1$

Example



- a first event of $P_1 \rightsquigarrow C(a) = \begin{bmatrix}1\\0\\0\end{bmatrix}$
- b: send $\rightsquigarrow C_1[1] += 1$; send C_1

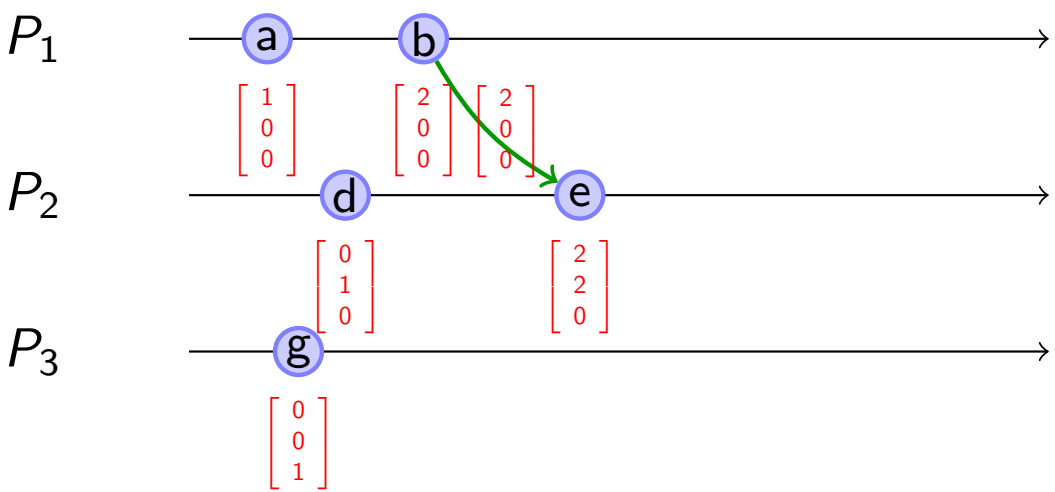
Implementing Vector Clocks

- Each process i has a local scalar vector C_i $(\in \mathbb{N}^{\#processes})$

Computation rules on process i

Initialization :	$C_i \leftarrow \{0, ..., 0\}$
Local event :	$C_i[i] += 1$
Sending message (m) :	$C_i[i] += 1$ then send (m, C_i)
Receiving message (m, E_m):	$\forall k, C_i[k] \leftarrow max(C_i[k], E_m[k])$
	$C_i[i] += 1$

Example



- a first event of $P_1 \rightsquigarrow C(a) = \begin{bmatrix}1\\0\\0\end{bmatrix}$
- b: send $\rightsquigarrow C_1[1] += 1$; send C_1
- e: recv

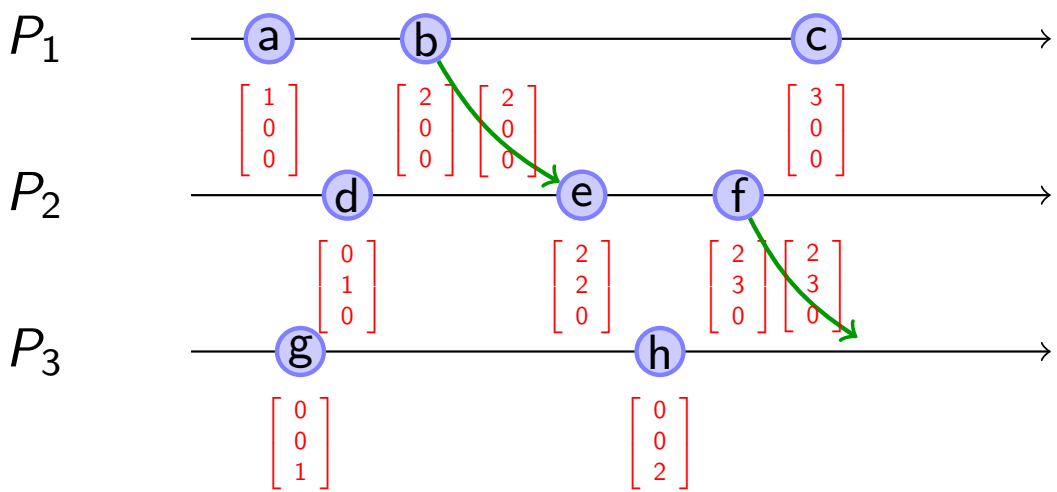
Implementing Vector Clocks

- Each process i has a local scalar vector C_i ($\in \mathbb{N}^{\#processes}$)

Computation rules on process i

Initialization :	$C_i \leftarrow \{0, ..., 0\}$
Local event :	$C_i[i] += 1$
Sending message (m) :	$C_i[i] += 1$ then send (m, C_i)
Receiving message (m, E_m):	$\forall k, C_i[k] \leftarrow max(C_i[k], E_m[k])$
	$C_i[i] += 1$

Example



- a first event of $P_1 \rightsquigarrow C(a) = \begin{bmatrix}1\\0\\0\end{bmatrix}$
- b: send $\rightsquigarrow C_1[1] += 1$; send C_1
- e: recv
- c, h: local events; f: send

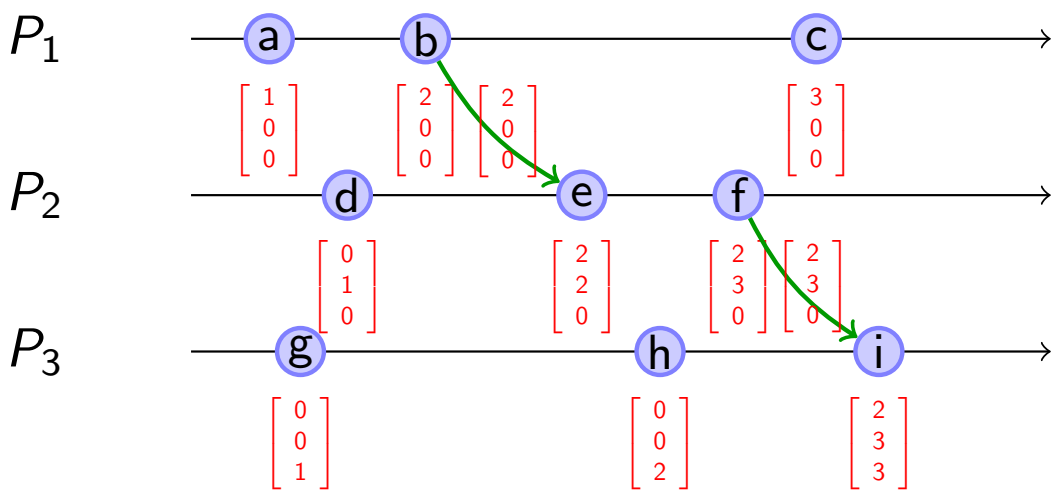
Implementing Vector Clocks

- Each process i has a local scalar vector C_i ($\in \mathbb{N}^{\#processes}$)

Computation rules on process i

Initialization :	$C_i \leftarrow \{0, ..., 0\}$
Local event :	$C_i[i] += 1$
Sending message (m) :	$C_i[i] += 1$ then send (m, C_i)
Receiving message (m, E_m):	$\forall k, C_i[k] \leftarrow max(C_i[k], E_m[k])$
	$C_i[i] += 1$

Example



- a first event of $P_1 \rightsquigarrow C(a) = \begin{bmatrix}1\\0\\0\end{bmatrix}$
- b: send $\rightsquigarrow C_1[1] += 1$; send C_1
- e: recv
- c, h: local events; f: send
- i: recv

Conclusion on Vector Clocks

Possible Applications

- Distributed system monitoring (event dating, distributed debugging)
- Computation of global state; Distributed simulation

Limits of Vector Clocks

- Comparing two vectors can require up to N comparison
- Processes don't know whether the others are up-to-date or lag behind
  - Matrix clocks solve that issue
  - $MC_i[j,k]$: what i knows of the knowledge of j about k's clock
  - This allows causal delivery
  - But matrix clocks are even more expensive ($O(n^2)$)

Global States

Definition: Local state

- The local state of process p_i after the execution of the event e_i^k is denoted σ_i^k
- The local state contains all data items accessible by that process
- Local state is completely private to the process
- σ_i^0 is the initial state of process p_i

Definition: Global state

The global state of a distributed computation is an n-tuple of local states $\sum = (\sigma_1, \ldots, \sigma_n)$ one for each process

Cut

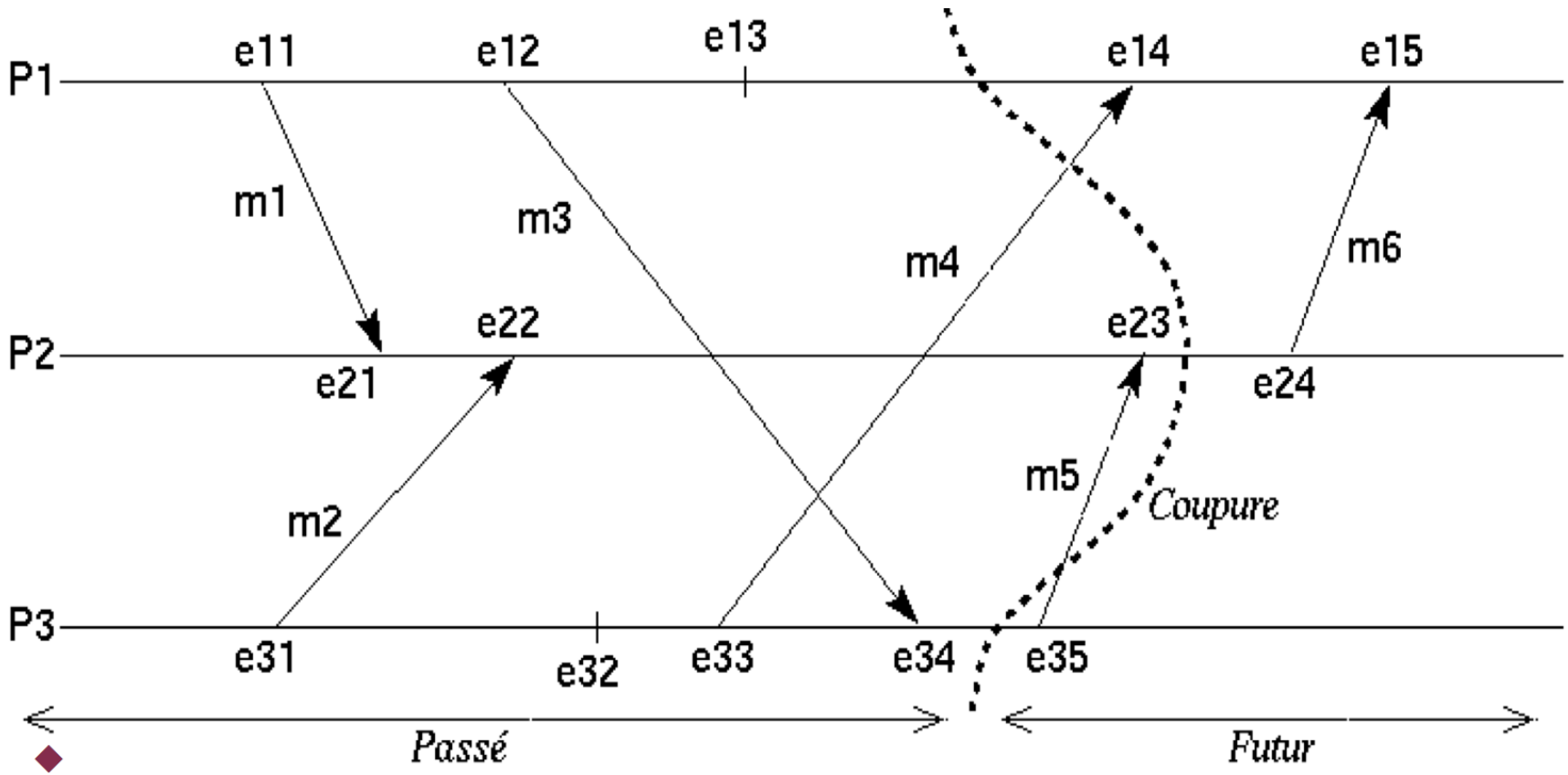
Definition: cut

A cut of a distributed computation is the union of n partial histories, one for each process:

$C = h_1^{c1} \cup h_2^{c2} \cup \ldots \cup h_n^{cn}$

- A cut may be described by a tuple $(c_1, c_2, \ldots, c_n)$ identifying the frontier of the cut, i.e, the set of last events, one per process
- Each cut $(c_1, \ldots, c_n)$ has a corresponding global state $(\sigma_1^{c1}, \sigma_2^{c2}, \ldots, \sigma_n^{cn})$

cuts



- Cut = { e11, e12, e13, e21, e22, e23, e31, e32, e33, e34 }
- State = {e13, e23, e34}

Consistent cut

Definition: Consistent cut

A cut C is consistent, if for all events e and $e^{'}$

$(e \in C) \wedge (e^{'} \rightarrow e) \Rightarrow e^{'} \in C$

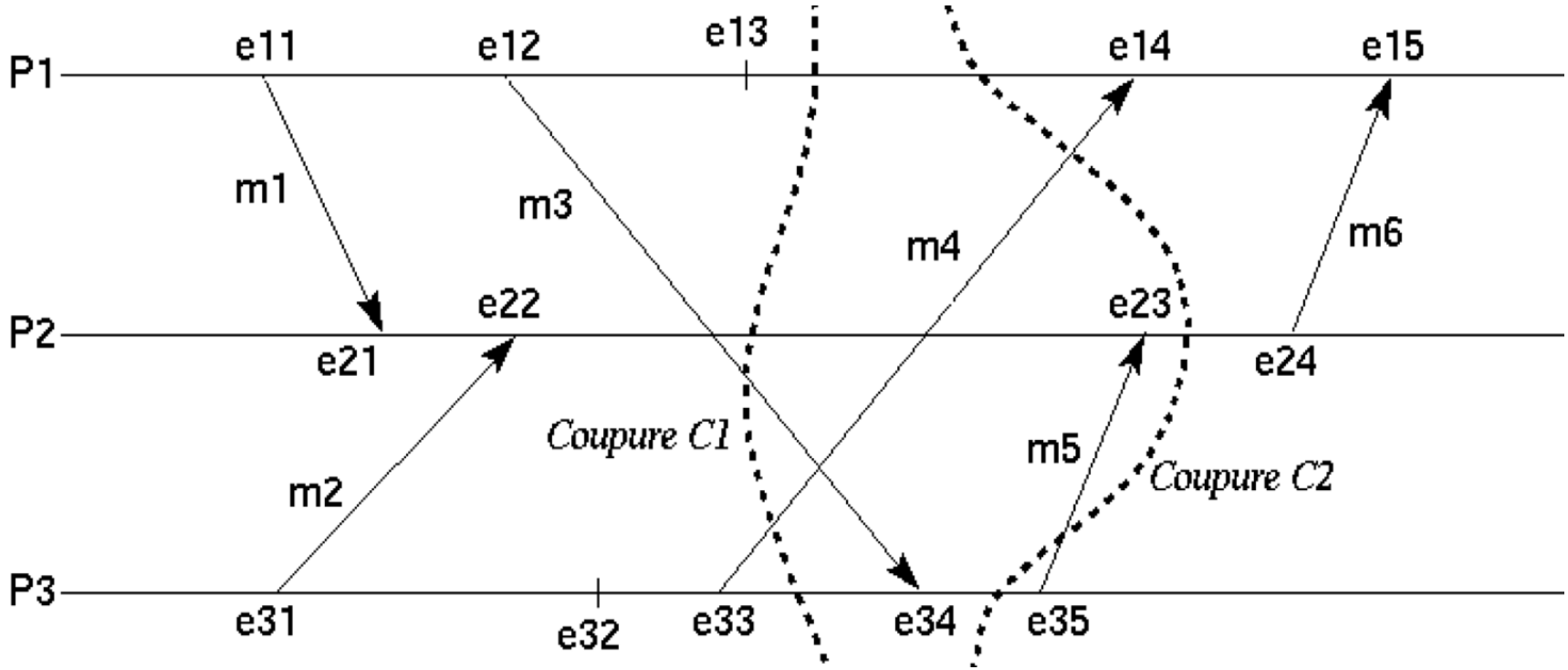
Definition: Consistent global state

A global state is consistent if the corresponding cut is consistent

In other words:

- A consistent cut is left-closed, w.r.t the happen-before relation
- All messages that have been received must have been sent before

cuts



- Cut C1 is consistent and C2 is not ($e35 \rightarrow e23$ and $e35 \notin C2$)
- In the space-time diagram, a cut C is consistent if all the arrows start on the left of the cut and finish on the right of the cut.