Distributed Systems

11 Synchronization

June-15-2009

Gerd Liefländer
System Architecture Group

Outline of Next Lectures

- Motivation for Timing
- Physical Clocks
- Logical Clocks

How to adjust your clock or how to get a precise time stamp?

- Global State
- Election Algorithms
- Mutual Exclusion

How to control concurrent activities?

- Distributed Transactions
- Distributed Deadlocks

How to deal with complicated distributed applications?

Synchronization & Coordination

Synchronize to order actions or events, i.e.

- Sequencing accesses to exclusive resources
- Requires a concept of a global time

Coordinate to agree on states/values, i.e.

- Actions that must occur simultaneously
- Actions that must occur at predefined times
- Agree on environment variables
- Agree on system/process state

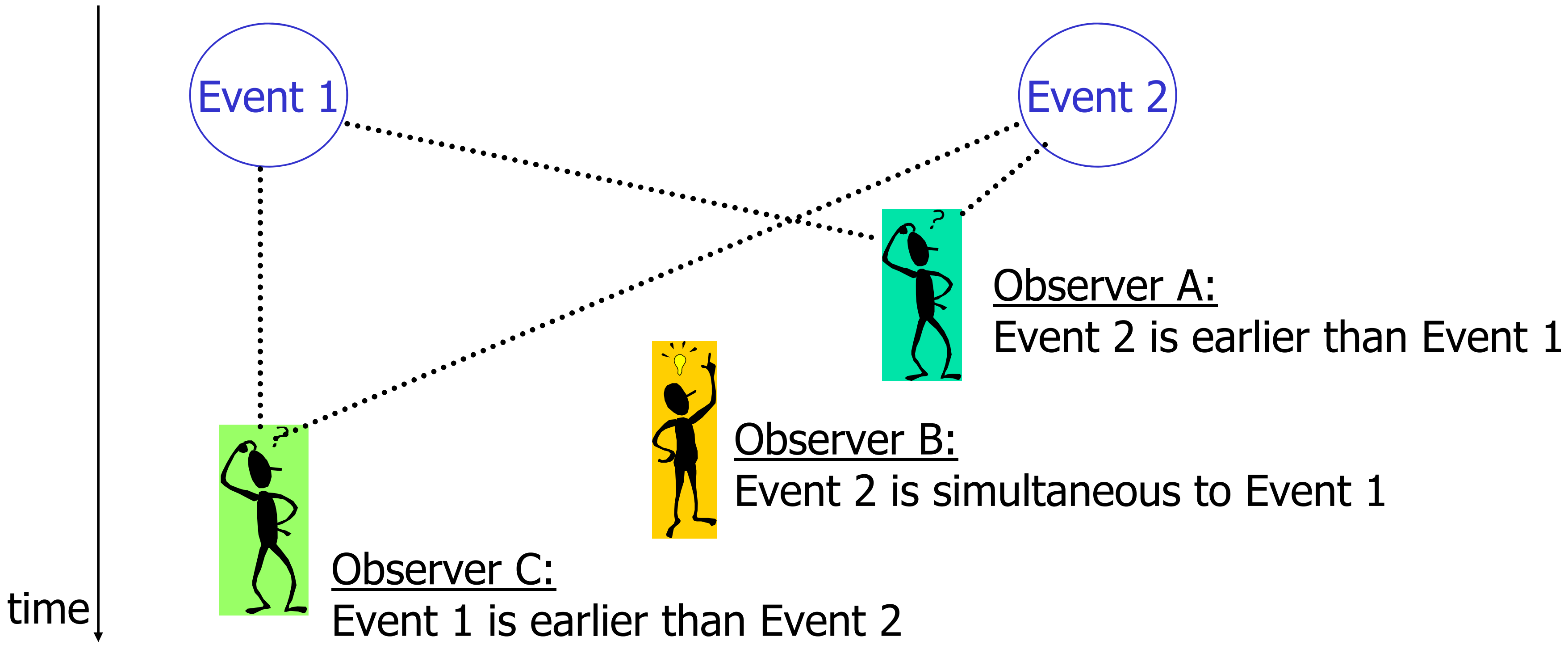
Problems with Time

Principle problems with time in a DS

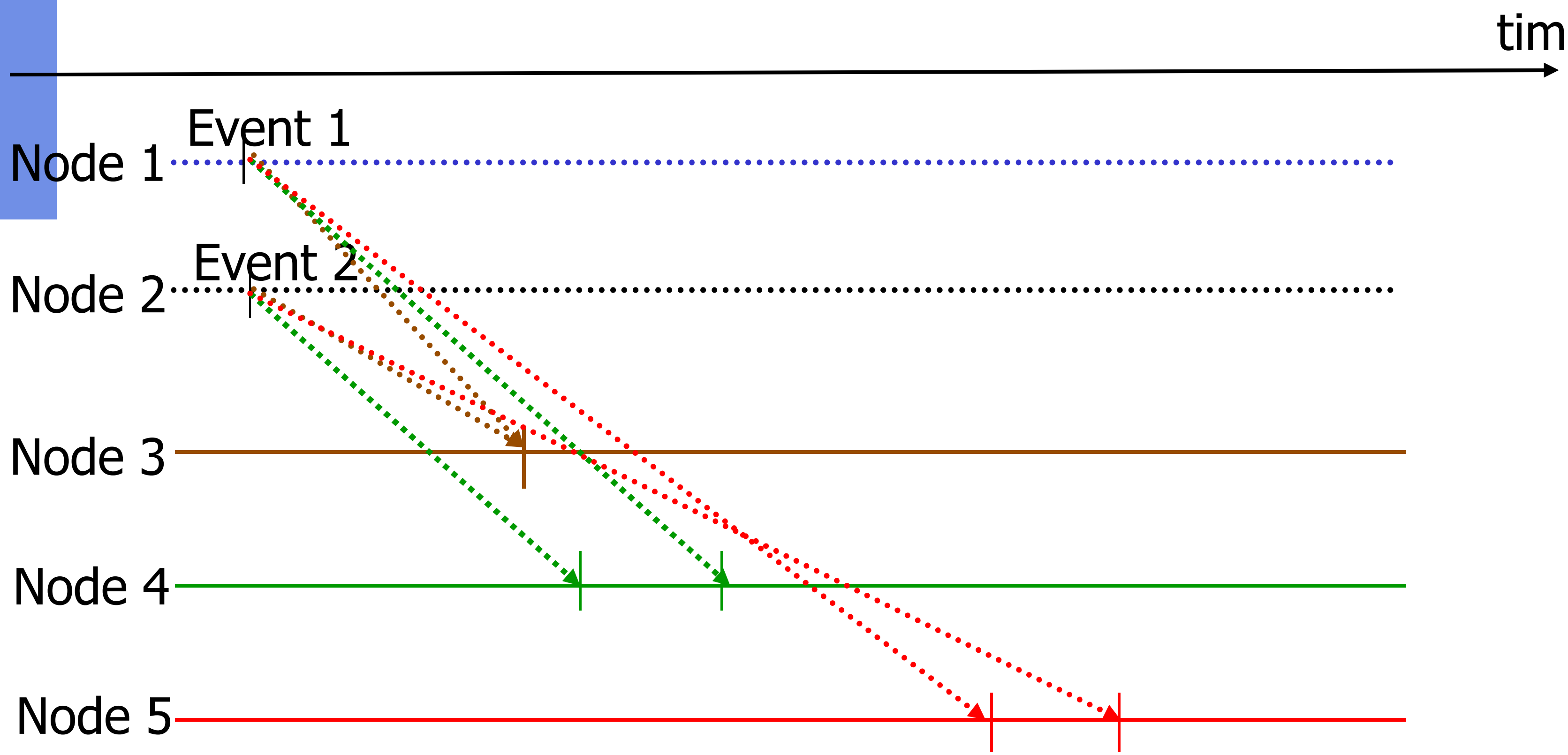
Philosophy

<u>Myth of Simultaneity:</u>

Suppose: Event 1 and event 2 happen at the same time



Event Timelines (Previous Example)



<u>Note:</u> Arrow starts from an event and ends at an observation point.
The slope of arrow depends on the relative speed of propagation
(that can vary according to network congestions)

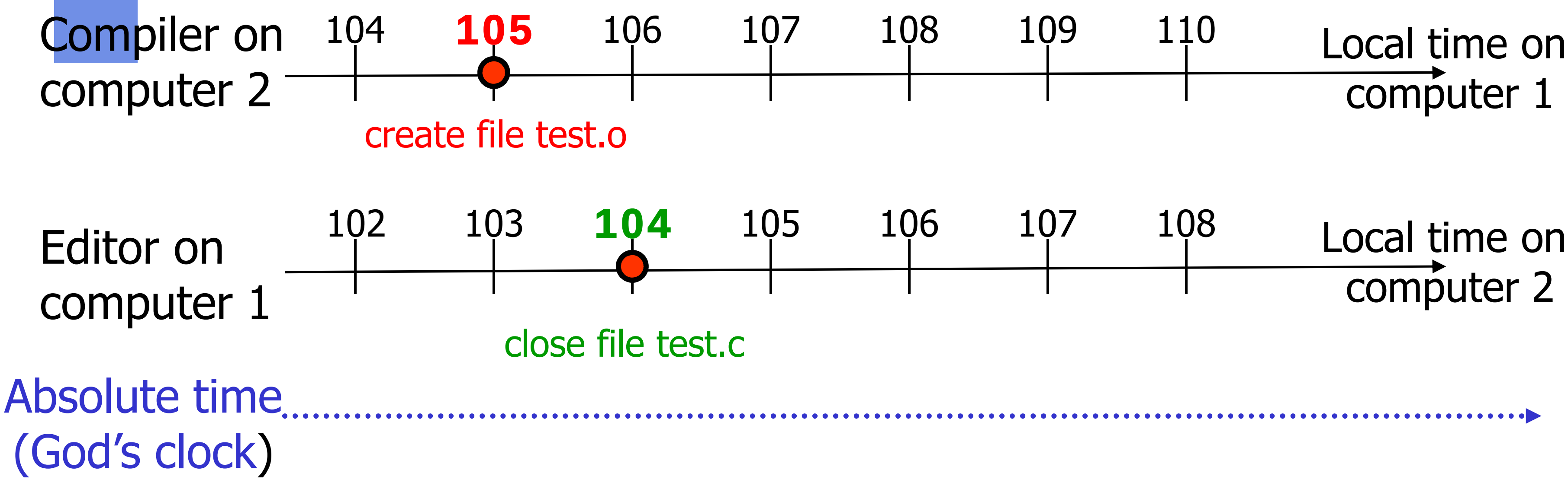
Physical Clocks

Why Precise Timestamps in DS?

In order to:

1. do some "precise" measurements
2. guarantee "up-to-date" data or
3. judge the actuality of data
4. establish a "total ordering" of objects or operations (see distributed transactions)

Inprecise Time Stamps



Assume: Clock(computer2) is faster than clock(computer1)

`test.c` was created after previous `test.o`, but **`make`** doesn‛t recompile, because `test.o` has a newer time stamp

Lack of a Uniform Global Time in DS

- However, due to nature of non precise clocks

  ⇒ ∃ no global, unique time in a DS, if each node has its own physical clock

- Assumption: ∃ central time server being able to deliver exact times via "time-messages" to all nodes of a widely spread DS (e.g. a LAN or a MAN)

  ⇒ due to non deterministic transfer-times of messages

  ⇒ no uniform time on all nodes of the DS

Transfer-time of time-messages from the central time server may vary over *time*

Physical Time

Some systems (e.g. Real-Time-Systems) may need accurate times.

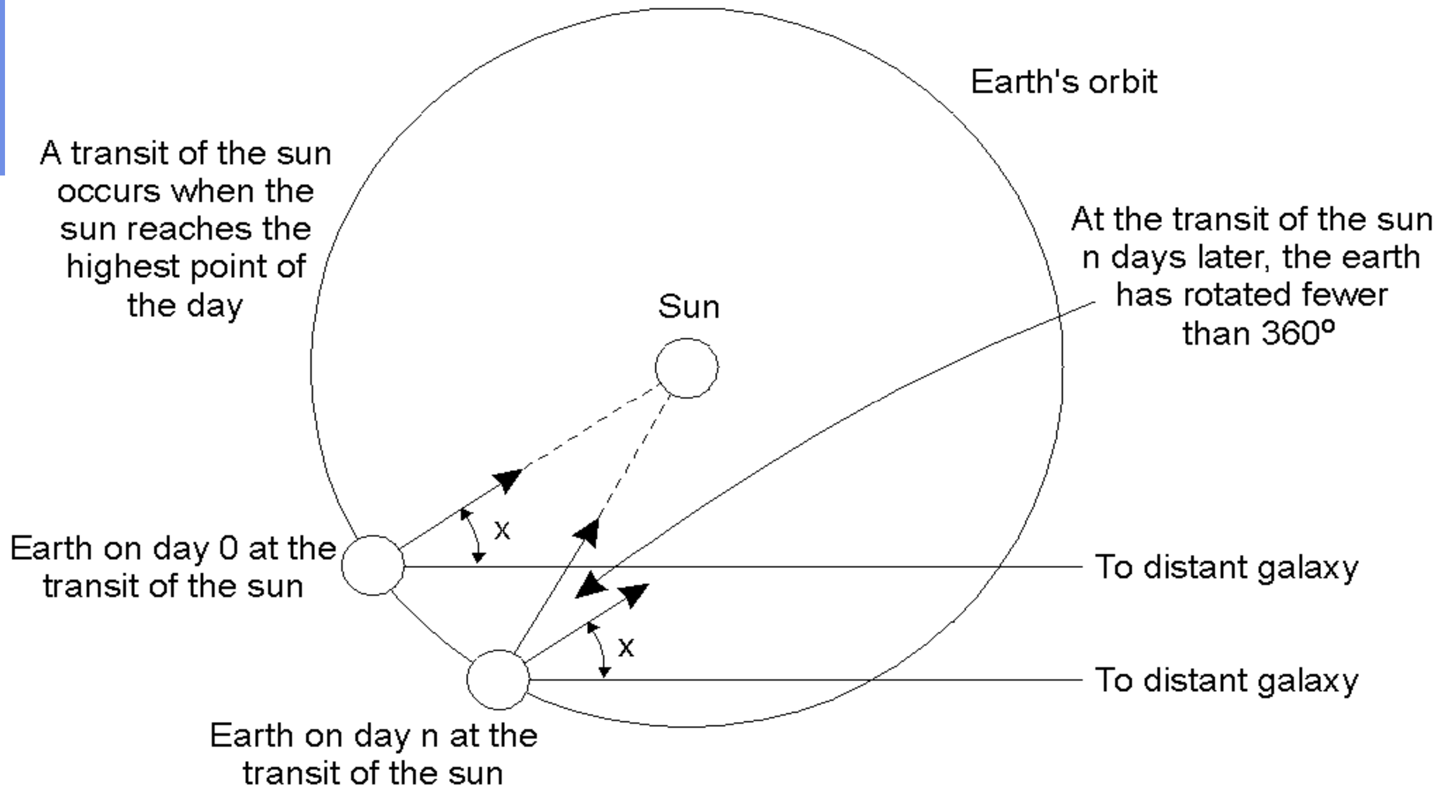
How to achieve a high accuracy?
Which physical entity may deliver precise timing?

1. Earth } Today: 1 sec ~ 1 solar day / 86400, however, rotation of earth slows down due to tidal friction and atmospheric drag (T-Rex had a "~400 day year" However, the year was as long as today)

2. Atom } State transitions in atoms (defined by BIH* in Paris)
1 sec = time a cesium atom needs for 9 192631 770 state transitions, invention 1948, initiate 1958, about 50 labors with cesium 133 clocks

*BIH = Bureau International de l´Heure à Paris

Mean Solar Day



Mean solar day $\neq$ turnaround of 360°

~ 359,xyz °

Problem with Physical Time

- Definition:

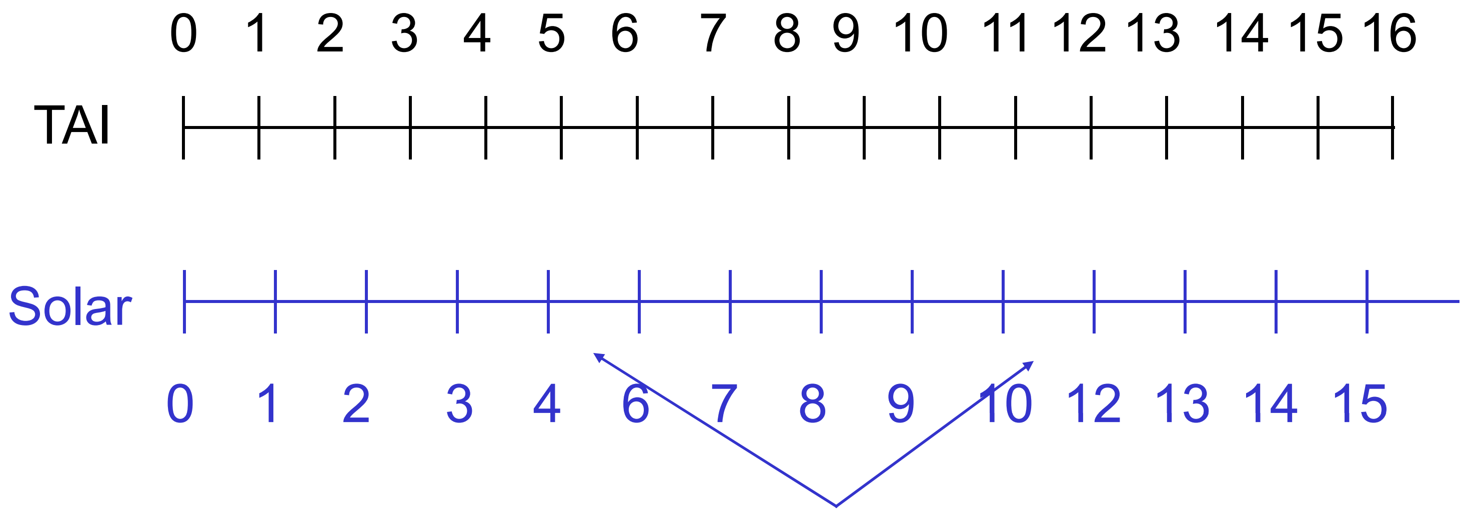
TAI-day = mean day of all cesium 133 clocks,
TAI-day ~ **3 msec shorter** than a solar day

⇒

BIH inserts a **leap sec**, if the difference between
a solar day and a TAI*-day is more than **800 msec**

* TAI (Temps Atomic International)

Leap Second[1]



Introduce a „leap second“

[1]See Gregorian calendar in 1582 when pope Gregory XIII decreed that 10 days had to be cancelled from the calendar. This event caused major problems. Which ones?
(BTW protestant people refused to obey for another 150 years!)

UTC Time

UTC = universal time coordinated, being the base of any international time measure.

How to implement?

Whenever BIH (or Braunschweig) announces a leap second, power-companies raise the frequency of their to 61 Hz or 51 Hz for 60 s or 50 s to adjust their clocks being based upon 60 Hz or 50 Hz.

UTC Time Broadcasting

UTC-signals come from radio broadcasting stations or from satellites (GEOS, GPS) with an accuracy of:

- 10 msec (broadcasting station)
- 0.5 msec (GEOS)
- 1.0 msec (GPS)

Remark:
Using more than 1 UTC source may improve accuracy

Sources of Precise Times (1)

- DCF77-sender
  - Long wave transmitter
  - Sending time-signals based on the atomic clock at the physical technical institute in Braunschweig
  - Range ~ 2000 km
  - Transmission of a 1-second pulse
  - Modulated bit template (60 Bits)
    - Minute, hour, month, year, day of the week
    - Additional hints when changing from summer to inter time an vice versa + leap seconds, years, …
  - Accuracy $\sim 2*10^{-13}$ averaged on 100 days

Sources of Precise Times (2)

- GPS-receiver
  - Satellites with atomic clocks
  - High precision time signal (necessary for a precise location of a specific location)
  - Accuracy ~ 0.1 sec

Problems with Clock Synchronization

- Interconnection path between local clock and reference clock
  - Uncertain transfer time of messages
  - Different transfer speed according to medium
  - Latency in network components (router, switches)
- Local OS
  - Different copy operations per OS
  - Different latency per interrupt handling

$\Rightarrow$ Exact clock synchronization is impossible

Problems with Clock Synchronization

Adjusting your local clocks:

- local clock behind reference clock
  - could be adjusted in one jump or in n steps
- local clock ahead of reference clock!!!
  - If you adjust in one step 2 different time stamps might get the same value
  - Solution: reduce speed of your local clock until it is synchronized again

Deviations of Local Clocks

Assume:

Each machine has a local timer causing timer interrupt *h times* per second.

If interrupts occurs its handler adds 1 to a software clock, based upon some reference time in the past.

Assumption:

$c(t)$ is the value of this local timer and

t is the exact UTC time.

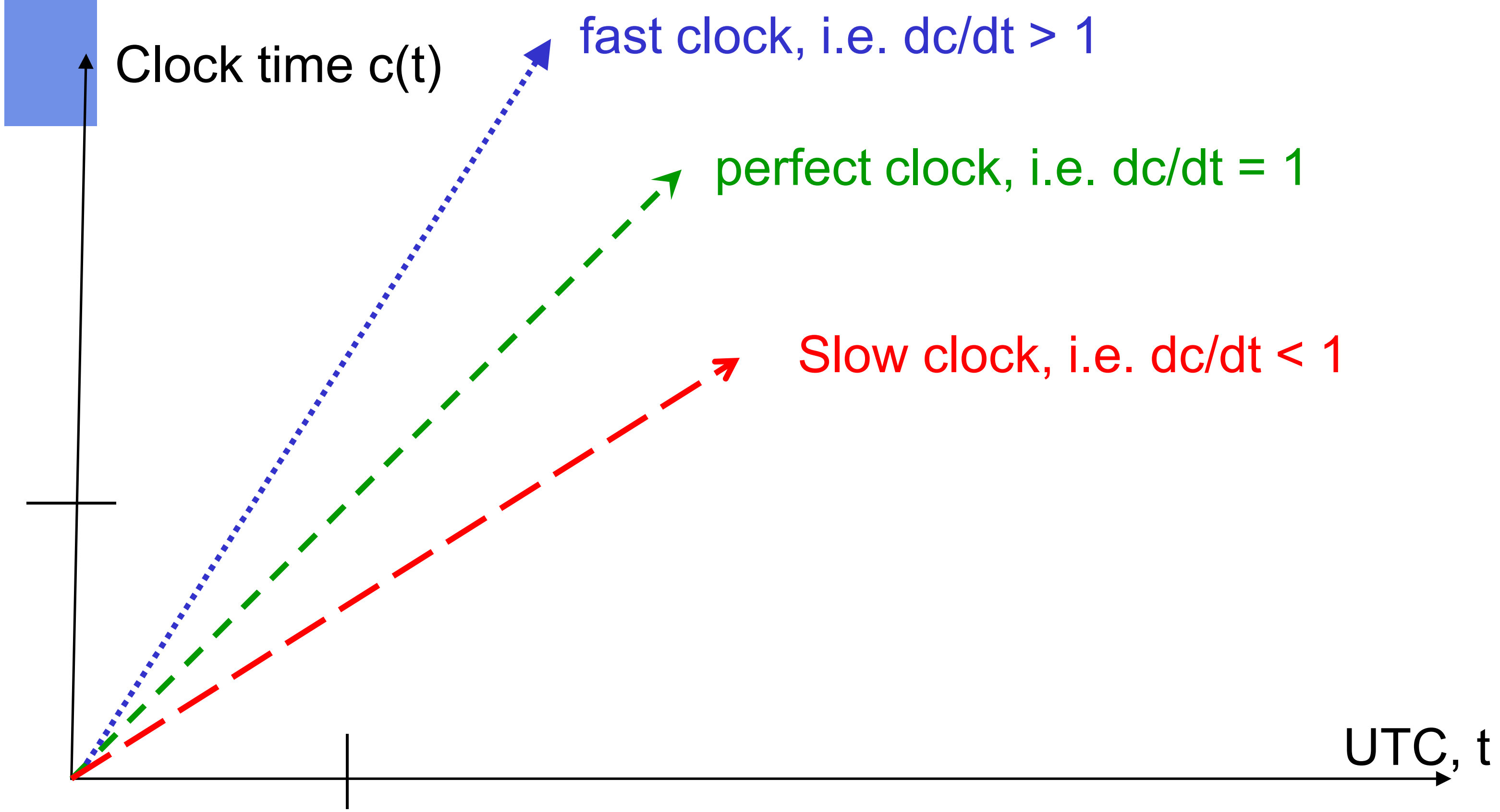
Deviations of Local Clocks

Modern timer chips have an accuracy of about 10^{-5}
i.e. if h = 60 instead of 216 000 ticks per hour,
a timer chip may produce # ticks $\in$ [215 998, 216 002]

More precisely:
A timer chip works within its specification
if there is a constant ρ :

$$1 - \rho \leq dc(t)/dt \leq 1 + \rho$$

Deviations of Local Clocks



Maximal Deviation

If 2 clocks drift in opposite directions from UTC-time, at Δt -after having been synchronized- they may be as much as $2 \rho \Delta t$ apart $\Rightarrow$

If OS has to guarantee that no 2 clocks ever differ by more than δ, then clocks have to be synchronized at least every $\delta/2 \rho$ seconds

Absolute Clock Synchronization

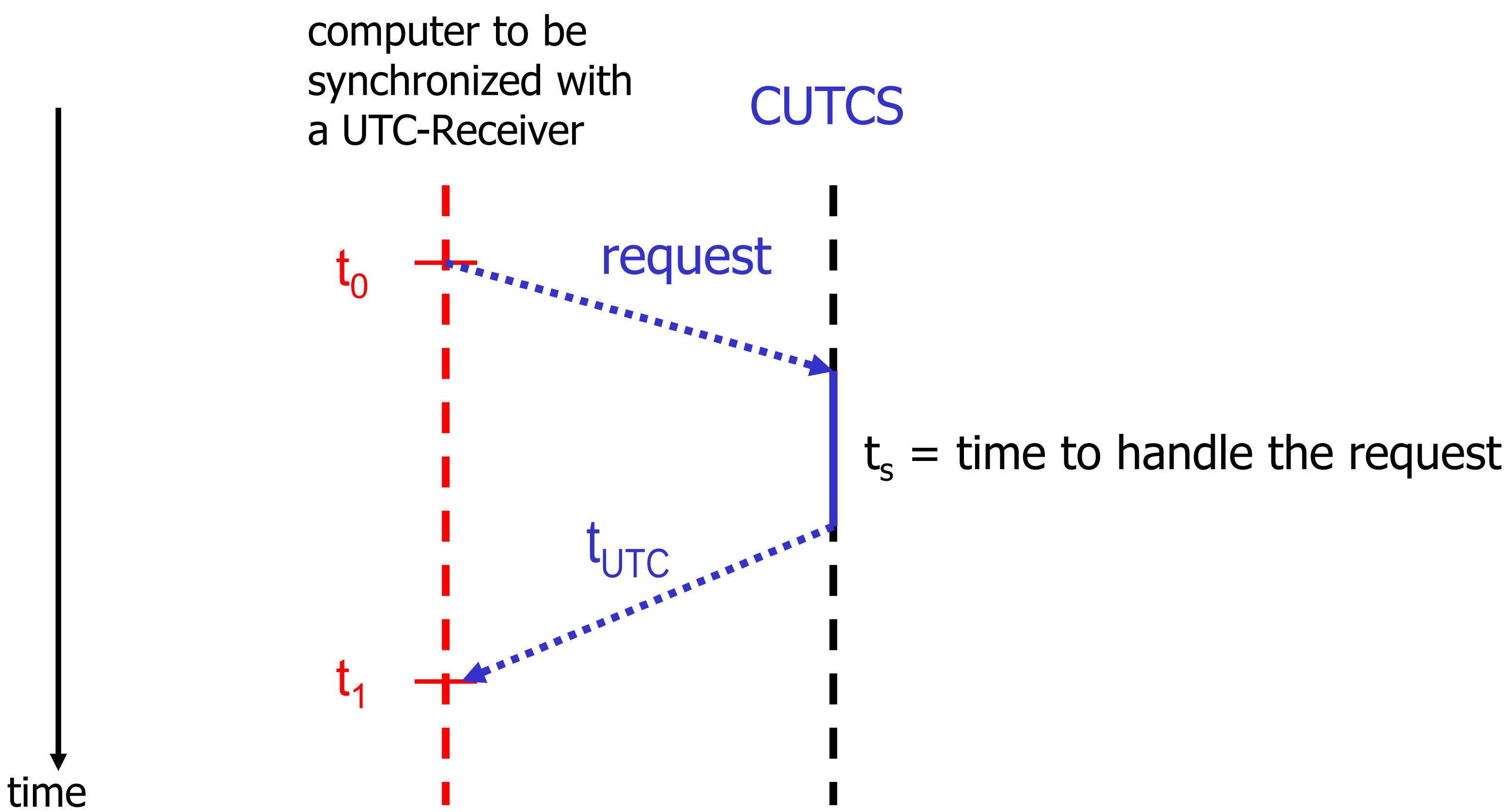
Cristian's[1] Algorithm:

1 wwv-node receiving UTC-signals, serving as the central UTC-time server (CUTCS) for the DS

Periodically (no more than every $\delta/2\,\rho$ seconds) each node in the DS sends a time request to CUTCS, which responds with its current time $\mathbf{t_{UTC}}$

[1]Flaviu Cristian from UoC

Absolute Clock Synchronization



Remark: Both time values (t_0 and t_1) are measured with the same clock

Problems with t_{utc}

1. Suppose node`s local clock is too fast, i.e.

   $t_{UTC} < t_1$

   Just adopting t_{UTC} can cause problems, (i.e. an object file may have an earlier time stamp than its previous changed source.)

   $\Rightarrow$ Adjust incrementally

Problems with t_{utc}

2. How to deal with message propagation time?

Good estimation of MPT is $(t_1 - t_0)/2$, i.e.

$$c(t) = t_{UTC} + (t_1 - t_0)/2$$

Absolute Clock Synchronization

- Initialize local clock: $t := t_{UTC}$
  (Problem: Time-Message Transfer-Time)

- Estimate Message transfer-time,
  $(t_1 - t_0)/2 \Rightarrow t := t_{UTC} + (t_1 - t_0)/2$
  (Problem: Time of the Request Message tr)

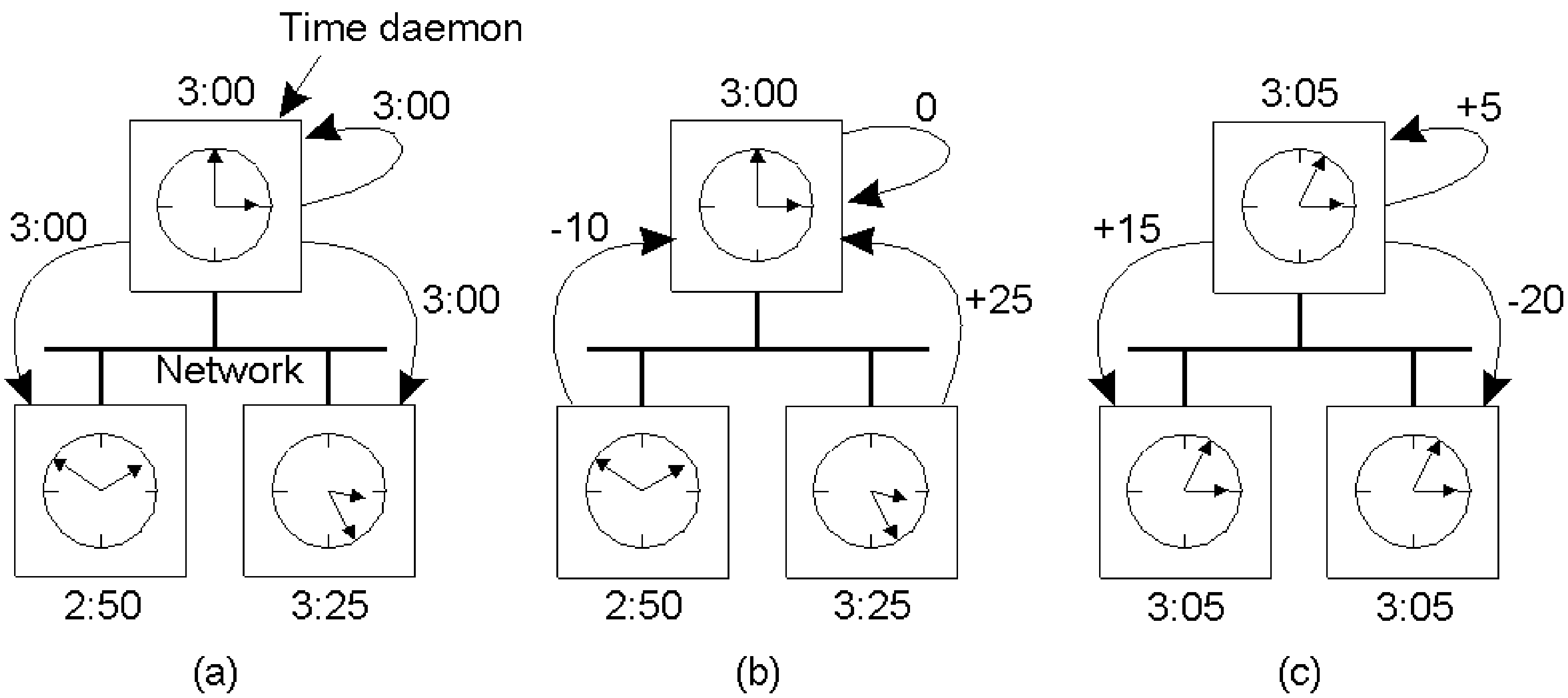
- Suppose: t_s is known, $\Rightarrow t := t_{UTC} + (t_1 - t_0 - t_s)/2$
  (Problem: Message transfer-times are load dependent)

- Multiple measurements ($t1 - t0$):
  - Throw away measurements above a certain threshold value
  - Take all others to get an average

Berkeley Algorithm

- CTS is active, i.e. it periodically polls all nodes to get their current local times $c_i(t)$.
- Based on these answers it calculates a mean and broadcasts this mean to all nodes again.
- Time server can estimate the local times of all nodes regarding the involved message transfer times.
- Time server uses the estimated local times for building the arithmetic mean
- Deviations from this arithmetic mean are sent to nodes enabling them to slow down respectively to speed up.

Berkeley Algorithm



a) At 3 p.m. daemon asks all the other machines for their clock values

b) The machines answer

c) The time daemon tells everyone how to adjust their clock

Summary

- Cristian's + Berkeley algorithms useful in intranets with only a couple of involved nodes
  - Why not that scalable?
- Both may be improved with fault tolerance methods
  - Instead of **1 Christian's UTC server** use **n time servers** and always take the **first answer**
  - Instead of taking the arithmetic mean from all clients in the Berkeley algorithm take the fault-tolerance mean, i.e. skip deviations with a certain threshold

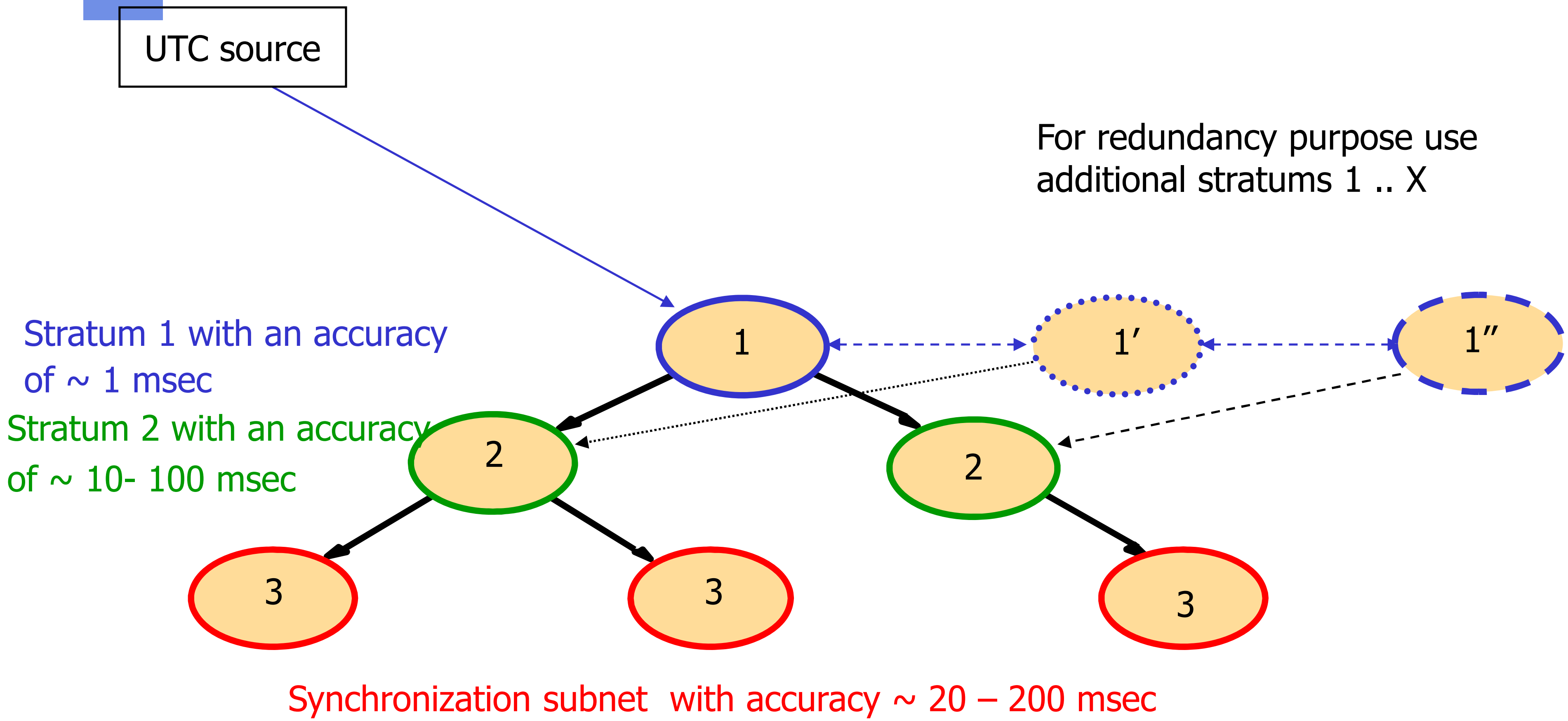
Network Time Protocol (NTP*)

- NTP Architecture
  - Time-servers build up a hierarchical subnet
  - Each primary time server “Stratum 1” has a UTC-receiver
    - Time signals via DCF77, GPS, WWV, CDMA technology
  - Secondary server “Stratum 2” gets its time data via network from one of the Stratum 1 machines
  - Other stations on level 3 synchronized by Stratum 2
    - Accuracy of clocks decreases with increasing level number
    - the net is able to reconfigure

*Mills, D.: “Improved Algorithms for Synchronizing Computer Network Clocks”, IEEE 1995

Network Time Protocol (NTP)

Internet service synchronizing clients to UTC



Logical Clocks

Lamport Time

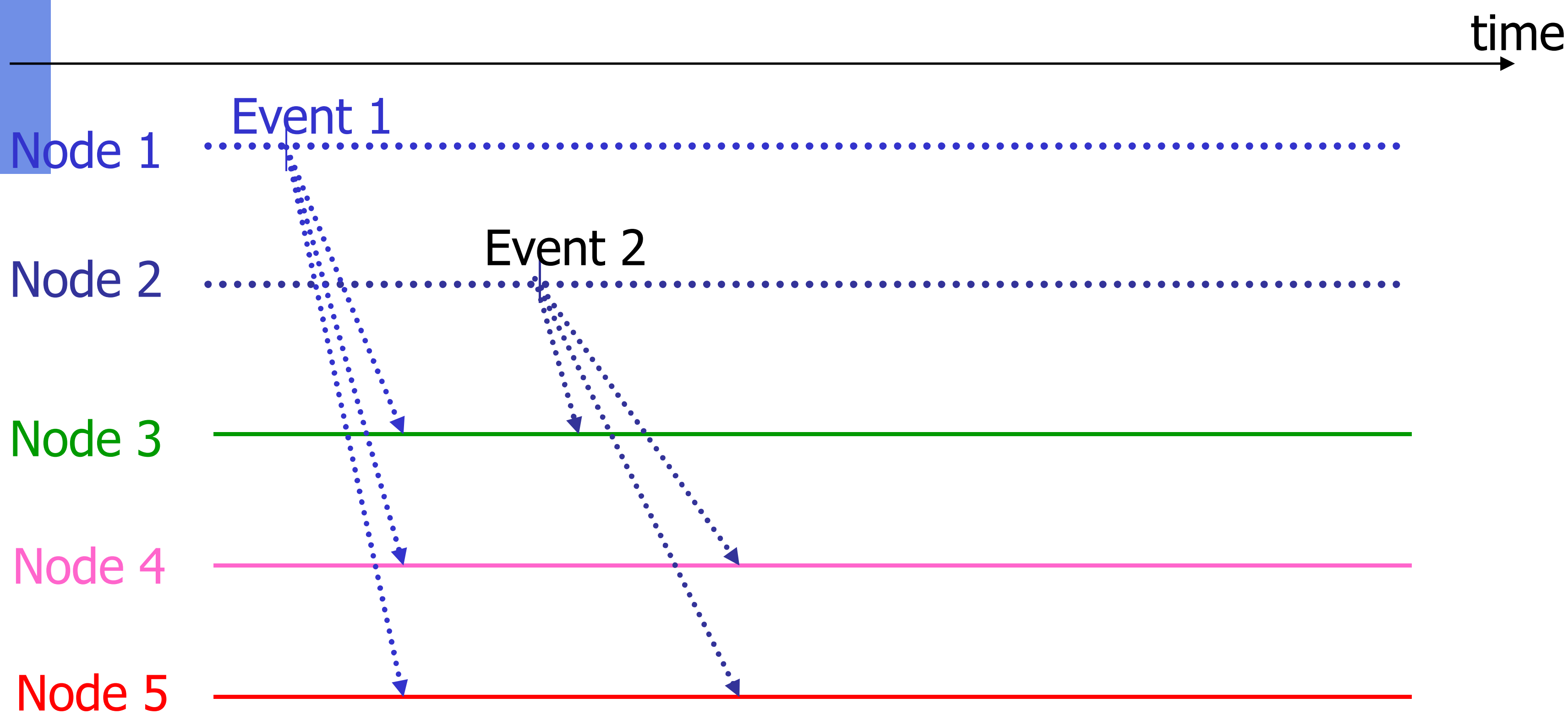
Vector Time

Matrix Time

Motivation for Logical Clocks

- Need not always precise physical time stamps
- However, at least you want to preserve the causality of events
  - Saying that event a has happened before event b is the same as saying that event a could have affected the outcome of event b
  - If the two events a and b happen on activities that do not exchange any data via IPC or shared memory or shared files, the ordering of a and b is irrelevant

Event Timelines (Example of prev. Slide)



Relation "Happened Before"

Smallest relation satisfying the following conditions:

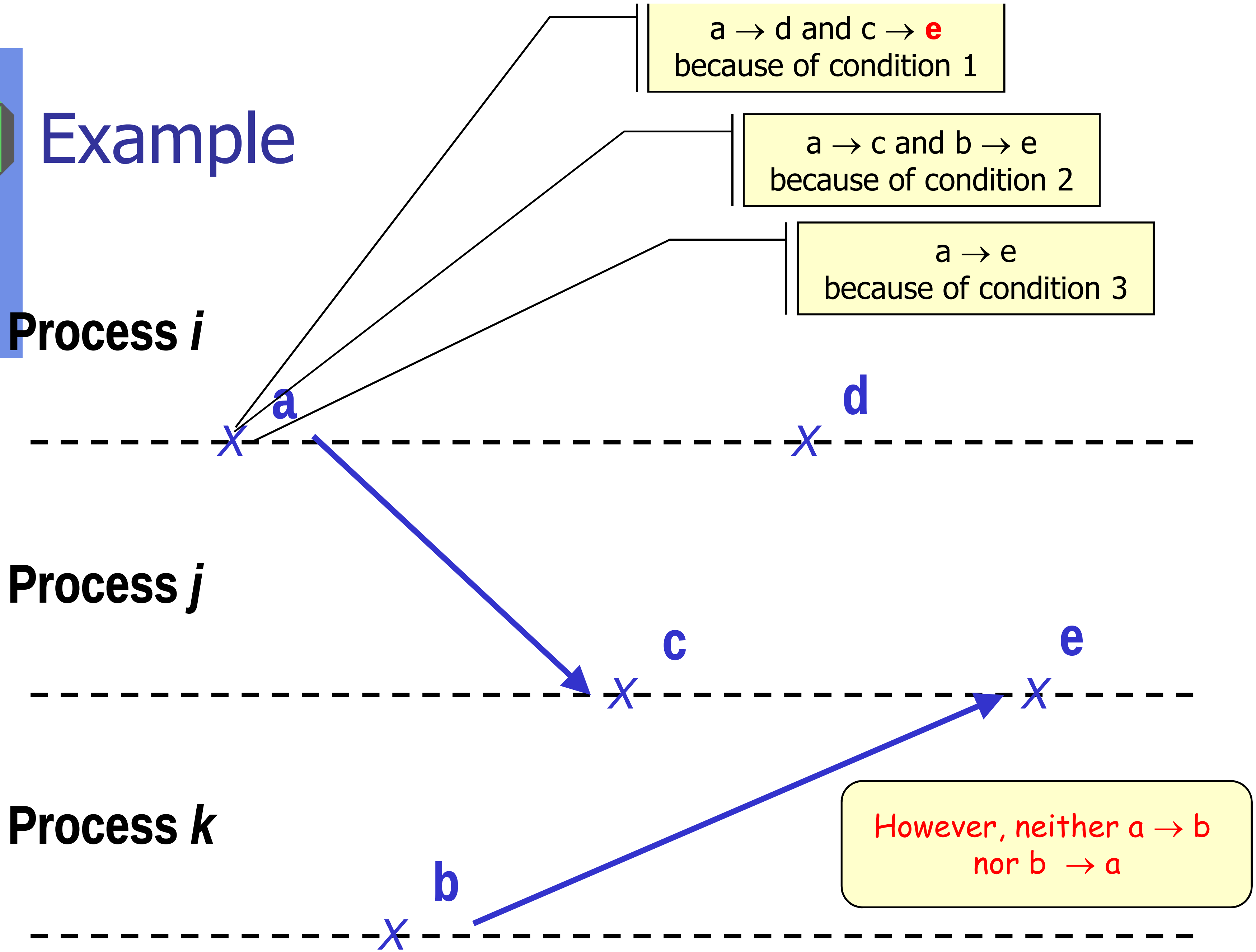
1. If **a** and **b** are events in the same process and **a** happens before **b**, then **a** → **b** (we can also say event b is potentially causal dependent on a)

2. If **a** is a sending event of a message by a sender S and **b** is the receiving event by a receiver R then obviously **a** → **b**

3. If **a** → **b** and **b** → **c** then **a** → **c** (transitivity)

4. If neither **a** → **b** nor **b** → **a**, **a** and **b** are concurrent **||** (not in a "Happened-Before" relation)

Example



a → d and c → e
because of condition 1
a → c and b → e
because of condition 2
a → e
because of condition 3
Process *i*
a
d
Process *j*
c
e
Process *k*
b
However, neither a → b
nor b → a

"Happened Before" (2)

- Let "$\rightarrow_p$" denote the local happened-before relation at node p:
- $a \rightarrow_p b$ iff a and b are both events at p, and a happens before b.

Global happened-before relation "$\rightarrow$" :

$a \rightarrow b$ holds iff

- $\exists$ node p: $a \rightarrow_p b$, or
- $\exists$ message m: a = send(m), b = receive(m)

Note:

The "Happened before" relation reflects potential causality, it does not model real causality

Logical Time

- In many cases it's sufficient just to order the related relevant events, i.e. we want to be able to position these events relatively, but not absolutely.

- Interesting is only the relative position of an event on the time axis
  $\Rightarrow$ no need for any scaling on this time axis

- Simple solution is the ring clock (André Barroso et al. "Synchronization, Coherence, and Event Ordering, 1988):
  - A clock message circulates
  - Incremented at each event (~ HW token ring)

Logical Time

- Characteristics of a logical time:
  - Causal dependencies have to be mapped correctly (e.g. sending a message "happens before" receiving the message)
  - Non related events do not have to be ordered, i.e. can appear in any order on the logical time axis)
- Assumptions:
  - DS := {n single-processor nodes}
  - Activity of each node=sequence of totally ordered events EN
  - 3 types of events
    - local events
    - send events
    - receive events
- The total activity of the system is: $E = \cup EN_{\text{all nodes}}$

Relation: Happened Before

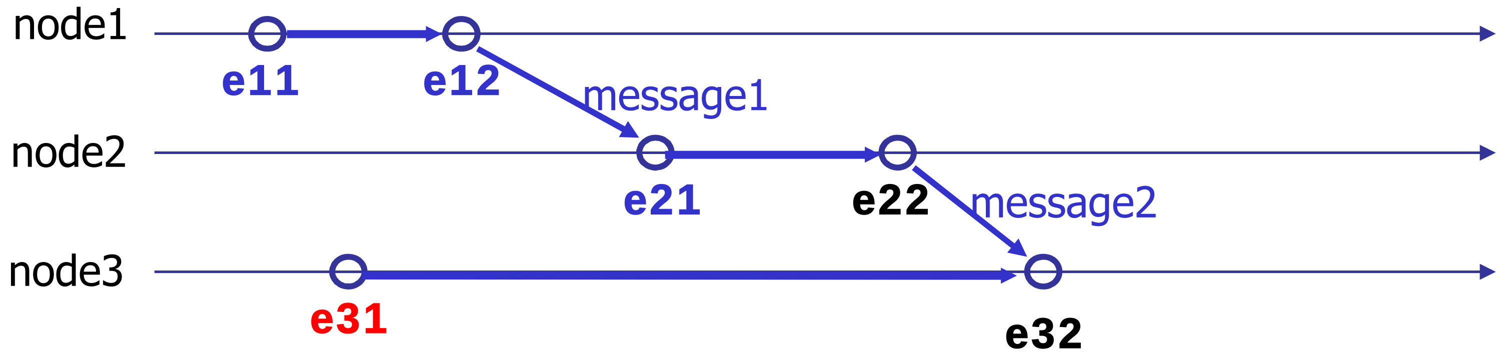
- We cannot always order all events: relation "has happened" before is only a preorder

- If a did not happen before b, it cannot cause b

Concurrent events:

- Two events a and b are concurrent , **a||b** ,
  if neither a → b nor b → a holds.

Example



It holds:

e11 $\rightarrow$ **e12** $\rightarrow$ **e21** $\rightarrow$ **e22** $\rightarrow$ **e32** , furthermore **e31** $\rightarrow$ **e32**, whereas **e31** is neither related "has happened before" to **e11**, nor to **e12**, nor to **e21**, nor to **e22**.

e31 is concurrent to e11, e12, e21, and e22.

Remark: Relation "happened before" $\rightarrow$ is also called causality-relation.

Logical Clock Conditions

- If an event b is potentially causal dependent on another event a (or if a happened before b) then the according logical times LT of both events must satisfy the following condition:

$$a \rightarrow b \Rightarrow LT(a) < LT(b)$$

Logical Times

- Scalar time (~ Lamport)
- Vector time (~ Fidge, Mattern, Schmuck)
- Matrix time (~ Michael, Fischer, Wuu, Bernstein, Lynch, SarinRaynal, Singhal)

Each of these logical clocks obeys 2 major rules:

R1: Describes how the local logical clock is updated when executing one of the 3 major events

- Internal
- Send
- Receive

R2: Describes how the global logical clock is updated

Lamport Time

Lamport Time

Vector Time

Matrix Time

Scalar (Lamport) Clock

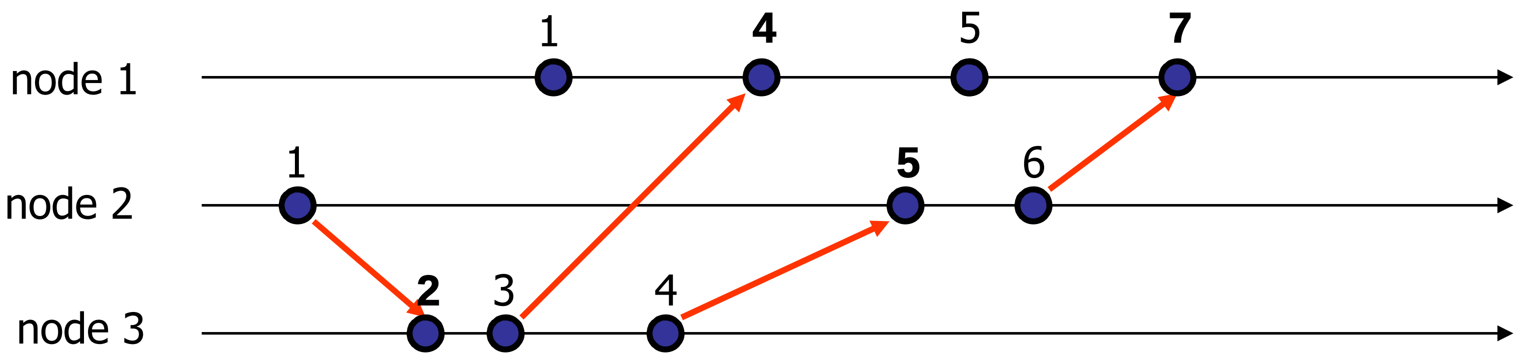
E := {events} and L: E → N defines the Lamport-time L, i.e. each **e ∈ E** gets a **time stamp** **L(e)** as follows:

1. Assume e is either a local event or a sending-event:
   A. If event **e** has no local predecessor, **L(e):= 1**,
   B. Otherwise ∃ a local predecessor **e'**, thus the timestamp of **e**, **L(e):= L(e') + 1**

2. Assume e is a receiving event (with a previous corresponding sending-event s):
   A. If **e** has no local predecessor, **L(e):=L(s)+1**
   B. Otherwise ∃ a local predecessor **e'**, then

   L(e):= max{L(s), L(e')} + 1

Lamport Time



Note: Each local counter is incremented with each local event. In a communication we adjust the involved counters of the two communicating nodes to be consistent with the "happened-before"-relation.

Remark:
Lamport time is consistent with the "happened-before"-relation, i.e. if $x \rightarrow y$, then $L(x) < L(y)$, but not vice versa.

Properties of Lamport Time

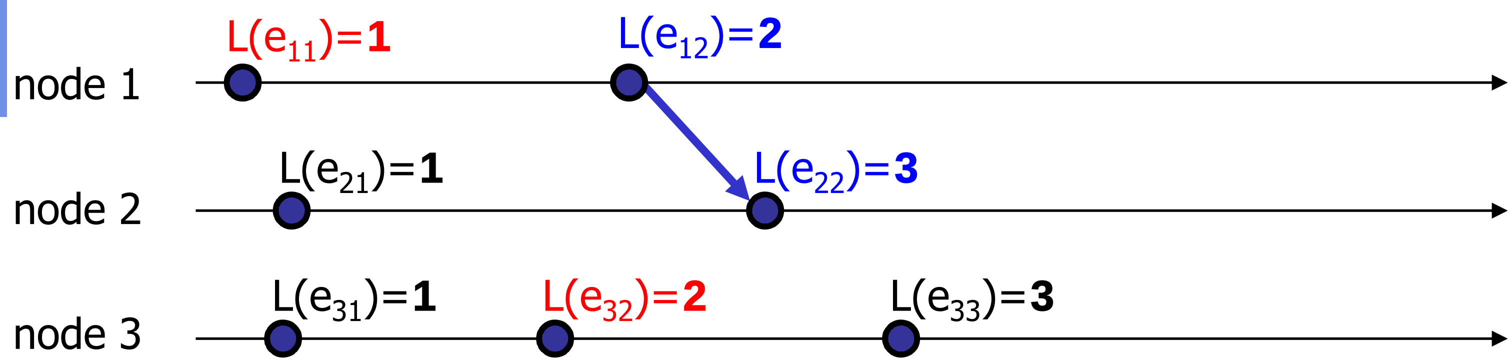
The Lamport time is consistent with the causality, but it does not characterize causality, i.e. it is not strongly consistent

A: If “x causes y”, then *x* has a smaller Lamport-time stamp than *y*,

$$x \rightarrow y \Rightarrow L(x) < L(y)$$

B: However: $L(x) < L(y) \not\Rightarrow$ “x causes y” !!!

Limitation on Lamport Clocks



From „Lamport time" values you cannot conclude whether two events are in any causal relationship, e.g. $e_{12} \rightarrow e_{22}$, because $L(e_{12}) < L(e_{22})$, but

$$e_{11} \not\rightarrow e_{32}\text{, even though } L(e_{11}) < L(e_{32})$$

Total Ordering of Events

A Lamport-time gives us a partial-ordering of distributed events which is sufficient for many problems.

However, if we add the unambiguous node number[1], we can establish a total-ordering:

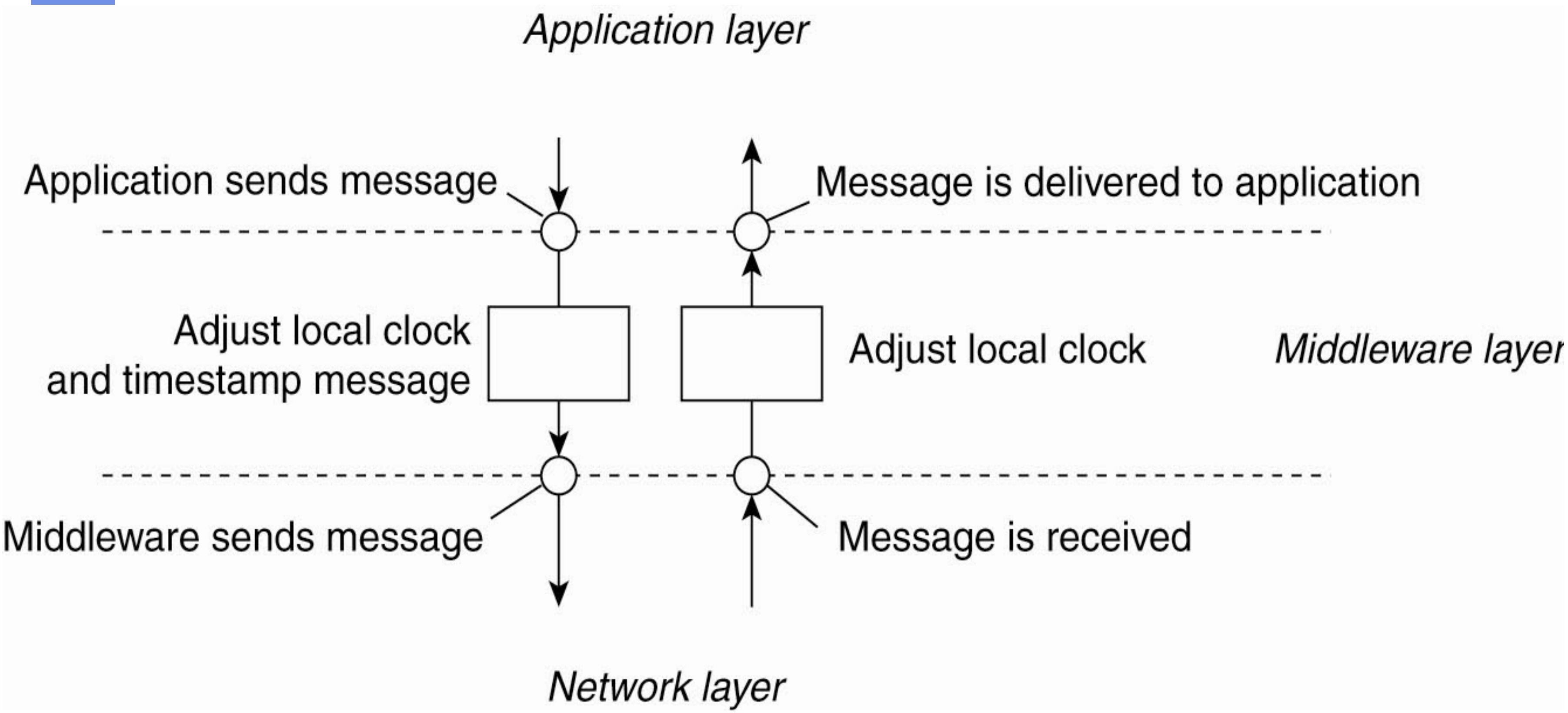
An event e at node a gets the global time stamp:

$$LT(e) := (L(e), a).$$

$$(L(e),a) < (L(e'),b) \Leftrightarrow L(e) < L(e') \text{ or } L(e) = L(e') \text{ and } a < b$$

[1]In Coulouris they use the PID instead of the node ID

Lamport's Logical Clocks (4)

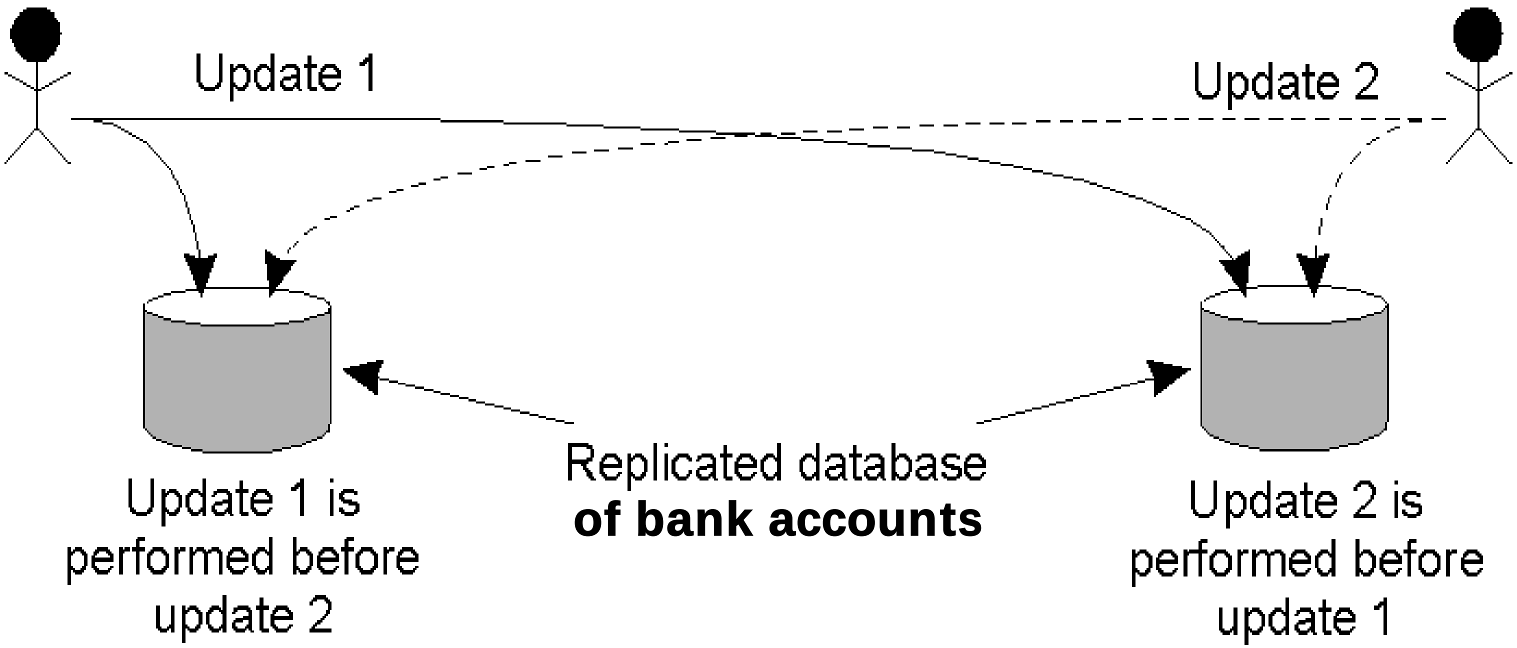


- The positioning of Lamport's logical clocks in DSs

Applications of Total Ordering

- To ensure liveliness properties in distributed algorithms, e.g.
  - Requests are time stamped and served according to the total order in these timestamps, e.g. to ensure fair mutual exclusion
  - Completely sorted multicast

Unsynchronized Update of 2 Replicas

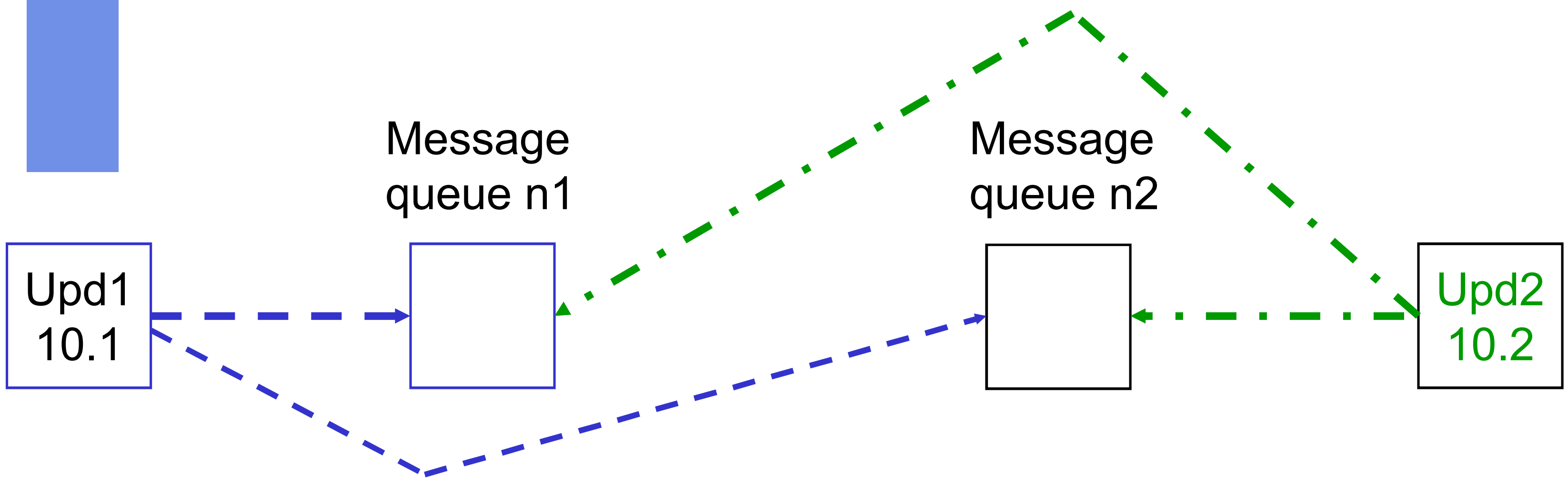


Update 1 = add 100 $

Update 2 = add 5% interest

Solution: Sorted multi cast for account transactions

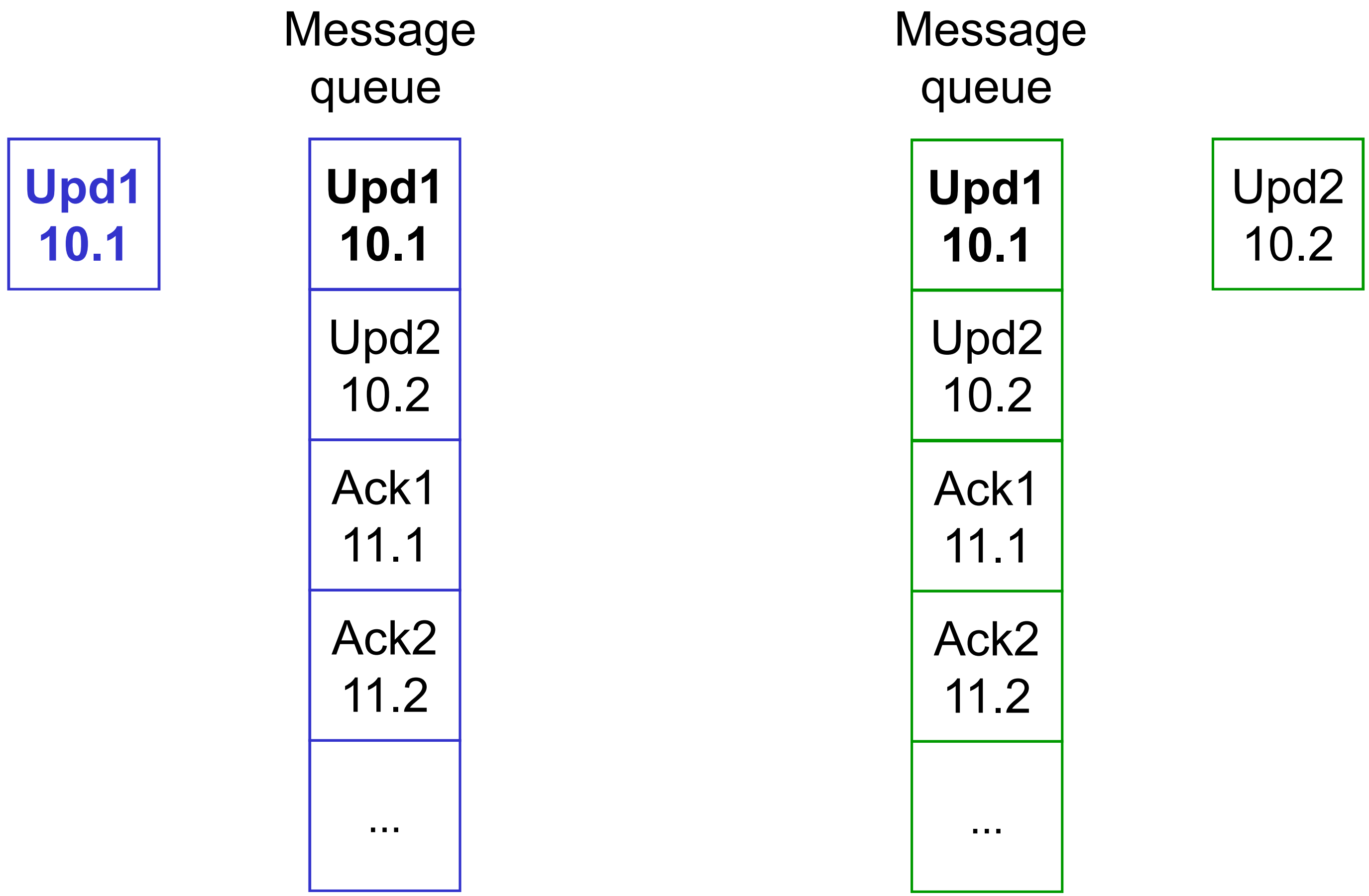
Using Lamport Time-Stamps



Goal: Deliver all multicast messages in a way that all sites receive them in the same order

Solution: Install identical local receive message queues
Use total ordered Lamport-Time for each update- and each acknowledge message

Using Lamport Time-Stamps



Updates are done according to the order in the queue after acknowledges from all sites have arrived

Vector Time

Vector Time[1]

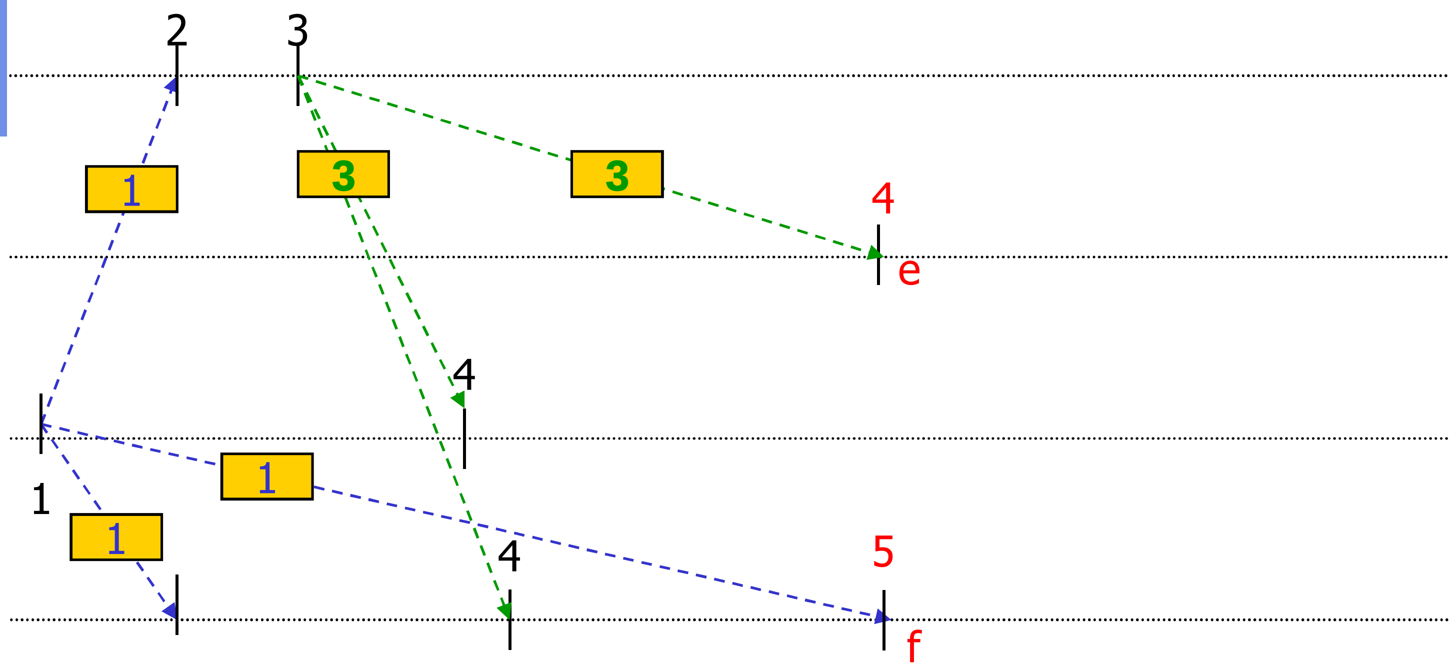
- Introduced to overcome limitations of Lamport times
- If L(a) < L(b) $\not\Rightarrow$ event a causally precedes event b
- With Lamport-time-stamps no sufficient support for causality
- Example: In a newsgroup every entry is multicasted to all subscribers, however any comment should follow the original article

Requirement for vector time:

If VT(a) < VT(b) $\Rightarrow$ event a causally precedes event b

[1]Mattern, F.: “Virtual Time and Global Sates in DS”, Proceedings of Parallel and Distr. Alg., 1989
Fidge, C.: “Logical Time in Distr. Computing Systems”, IEEE Computer, 1991

Limitations of Lamport Time



Result: Event e $\not\rightarrow$ f although L(e) < L(f)

Vector Time

- Assumption: ∃ n tasks (processes) P_i in DS
- Each P_i has its own local "vector clock" being a n-dimensional time-vector (initially zeroed)
- $VT_i(a)$ is timestamp of event a in process P_i
- $VT_i[i]$ number of events that have happened or are known in in P_i
- If $VT_i[j] = k$ means that this is P_i currently best guess that at least k events have happened in P_j or are known to P_j

Only an estimation

Vector Time

DS with n distributed processes. Every process p has its VT_p reflecting the current vector-time of p, if it is built according to the following rules:

(1) Initially, $VT_i := (0,\ldots0)$ for all $i \in [1, n]$

(2) For each event e in P_i the local time component of VT_i is incremented, i.e. $VT_i[i] += 1$

(3) Whenever P_i sends a message m, P_i adds its current vector-time $t=VT_i$ to this message m

(4) When P_j receives a message m with timestamp t it sets $VT_j := \max\{ t[k], VT_j[k] \}$*

*Build the maximum component wise

Properties of Vector Time

The following relations for the vector-time hold:

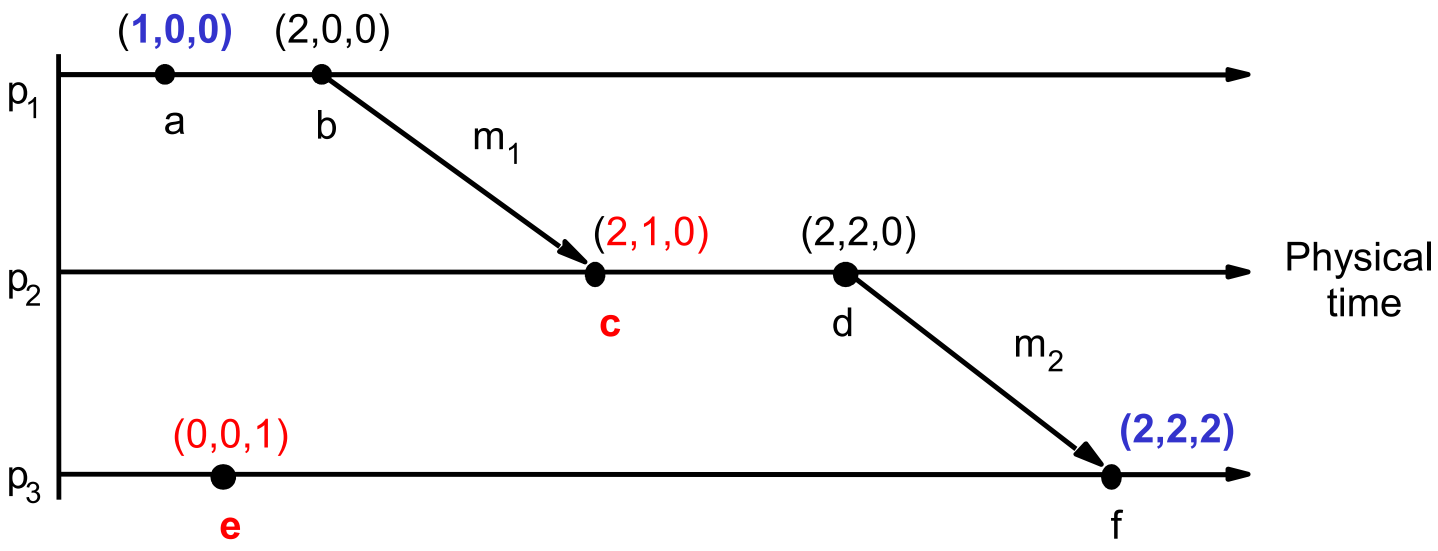
Suppose u, v are two vector times of dimension n :

1. $u \leq v \Leftrightarrow u[k] \leq v[k] \; \forall k = 1, \ldots n$

2. $u < v \Leftrightarrow u \leq v$ and $u \neq v$

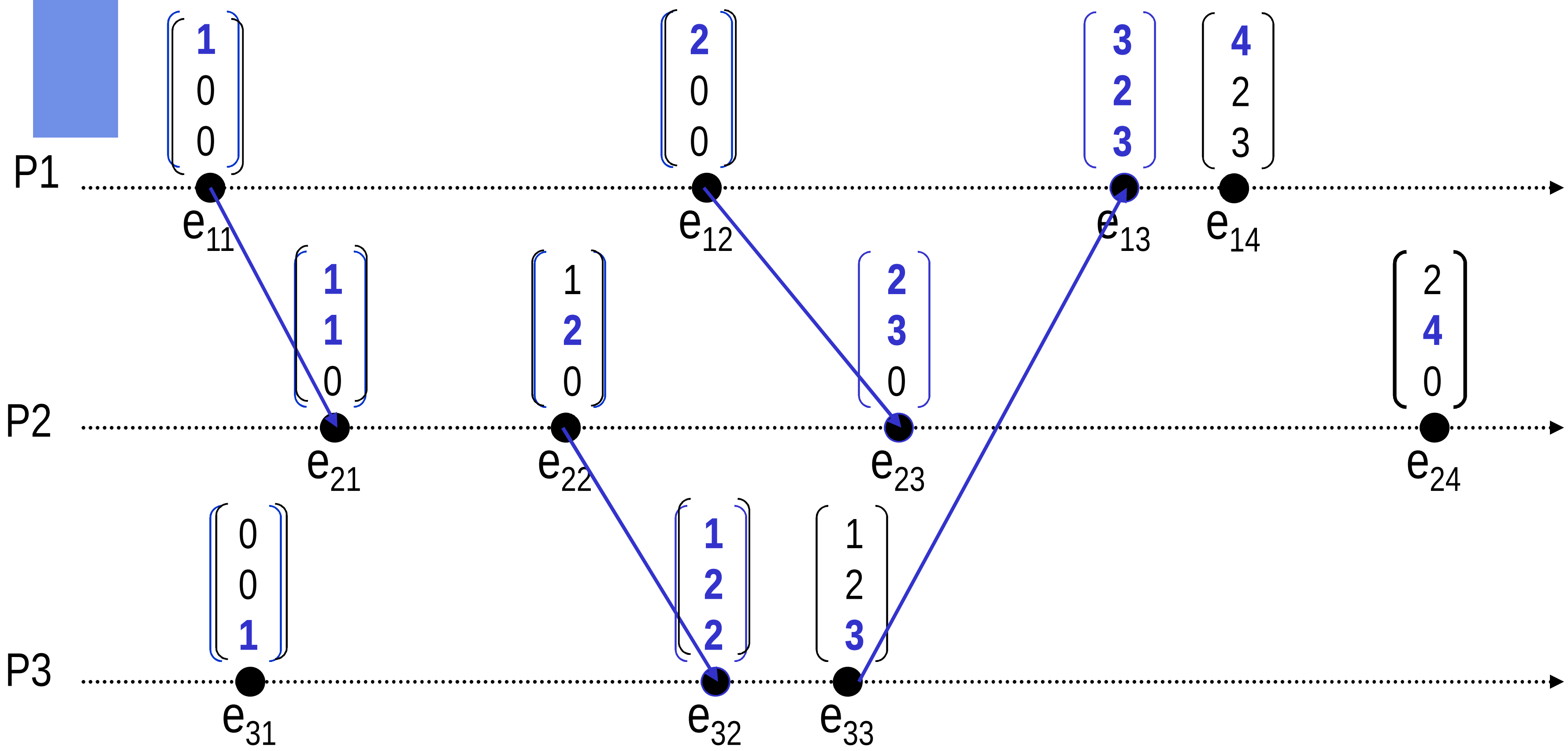
3. $u \,||\, v \Leftrightarrow \neg(u < v)$ and $\neg(v < u)$

Characteristics of Vector Time



- **VT(a) < VT(f)** $\Rightarrow$ **a** $\rightarrow$ **f** event a has happened before event f (thus ***a*** might have caused ***f***)
- Events c || e, because neither VT(c) < VT(e) nor VT(e) < VT(c)

Example: Vector Time



Characteristics of Vector Time

The following inter relationships between causality or the "happened before" relation and vector-time hold:

A.) $e \rightarrow e' \Leftrightarrow VT(e) < VT(e')$

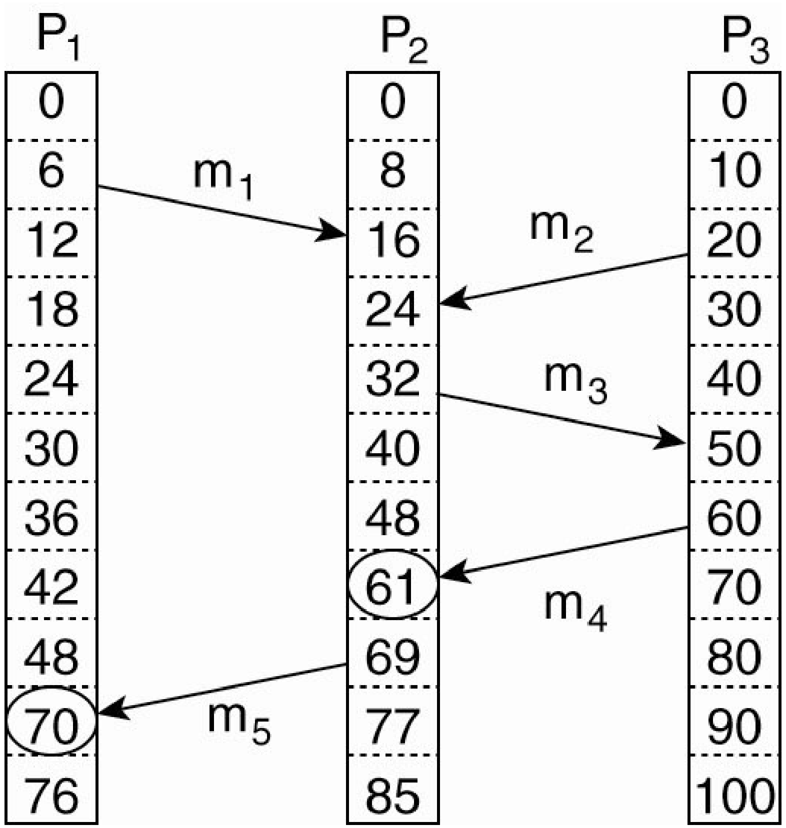
B.) $e \,||\, e' \Leftrightarrow \neg(VT(e) < VT(e'))$ and
$\neg\,(VT(e') < VT(e))$

Vector-time is the best known estimation for global sequencing that is based only on local information.

Applications of Vector Time

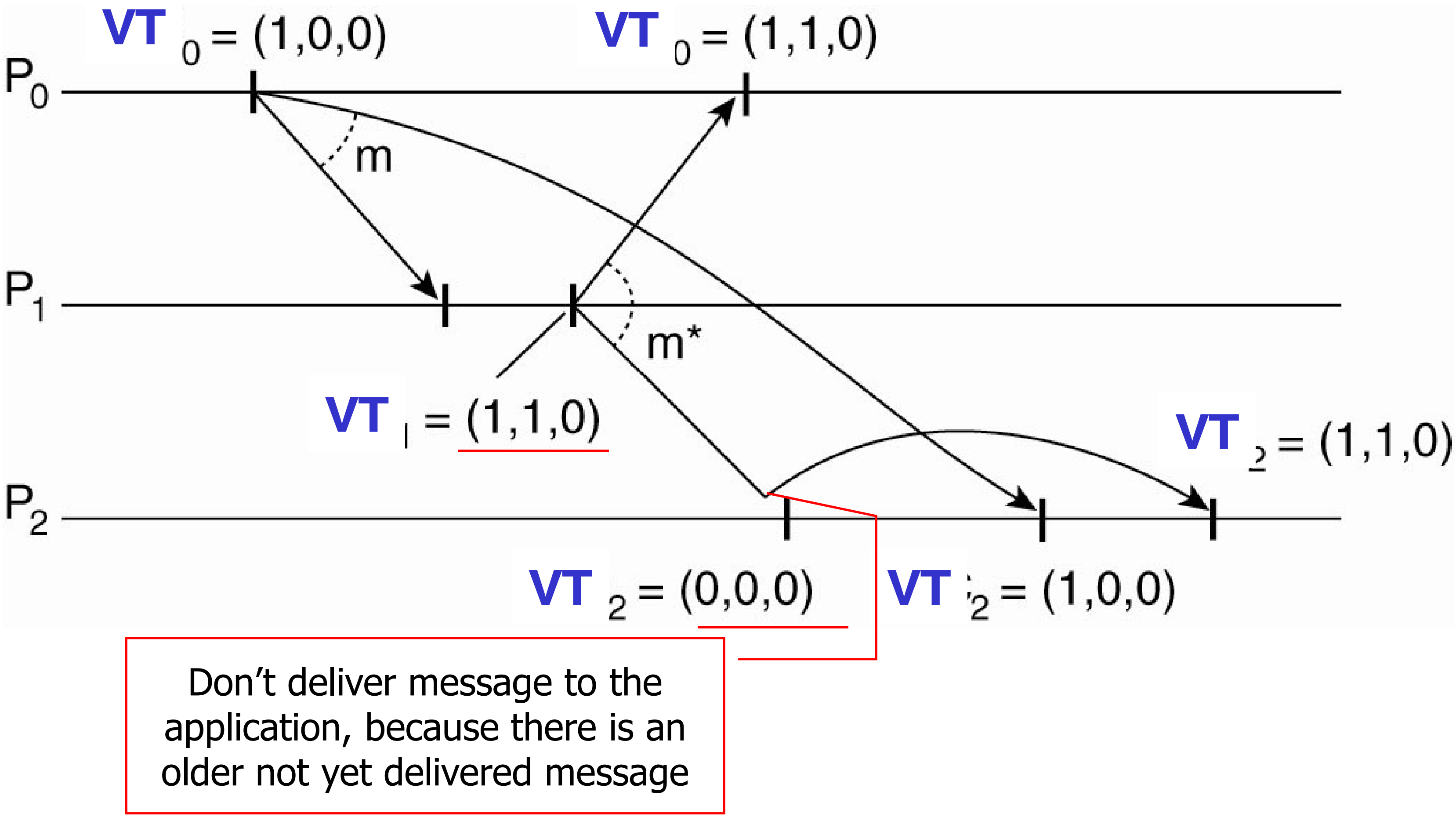
- Vector time stamps reflect potential causal ordering $\Rightarrow$ used for
  - Distributed debugging
  - Causal ordered communication
  - Causal distributed shared memory
  - Establishing global breakpoints
  - Determining consistency of checkpoints in optimistic recovery

Synchronizing Vector Clocks



- Concurrent message transmission using logical clocks

Enforcing Causal Communication



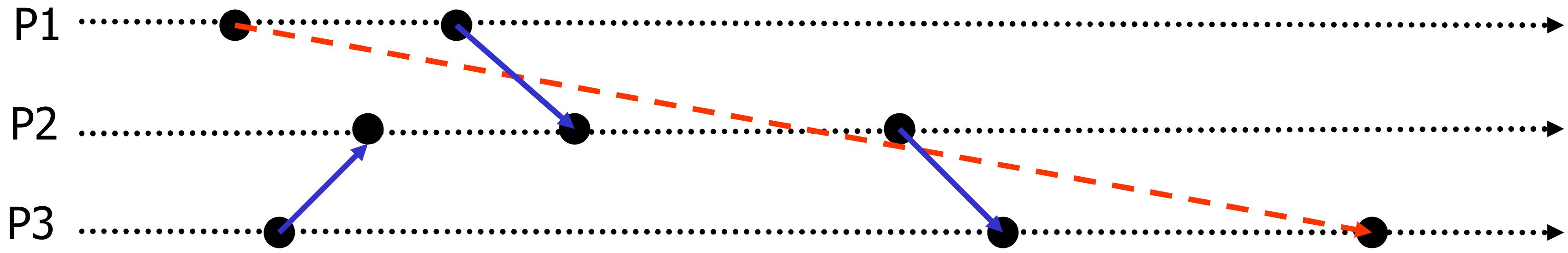
- Assumption: Each IPC is a broadcast & only send/receive events

Causal Ordering of Messages

Definition:
Suppose m_1 and m_2 are two messages being received at the same node i.
A set of messages is causally ordered if for all pairs <m1, m2> the following holds:

$$send(m1) \rightarrow send(m2) \Rightarrow receive(m1) \rightarrow receive(m2)$$

Example of non causally ordered messages:



Matrix Clocks by Raynal & Singhal

- Each node i maintains a n x n matrix M_i, initialized to 0, (i.e. no message was sent up to now).
- Before sending a message M from node i to node j, process P_i **increments M_i [i,j]**

$$M_i = \begin{pmatrix} M_i[1,1] & M_i[1,2] & \dots & M_i[1,j] & \dots & M_i[1,n] \\ M_i[2,1] & M_i[2,2] & \dots & M_i[2,j] & \dots & M_i[2,n] \\ \dots & \dots & \dots & \dots & \dots & \dots \\ M_i[i,1] & M_i[i,2] & \dots & \mathbf{M_i[i,j] + 1} & \dots & M_i[i,n] \\ \dots & & & & & \\ M_i[n,1] & M_i[n,2] & \dots & M_i[n,j] & \dots & M_i[n,n] \end{pmatrix}$$

Protocol: Causal Ordering of Messages

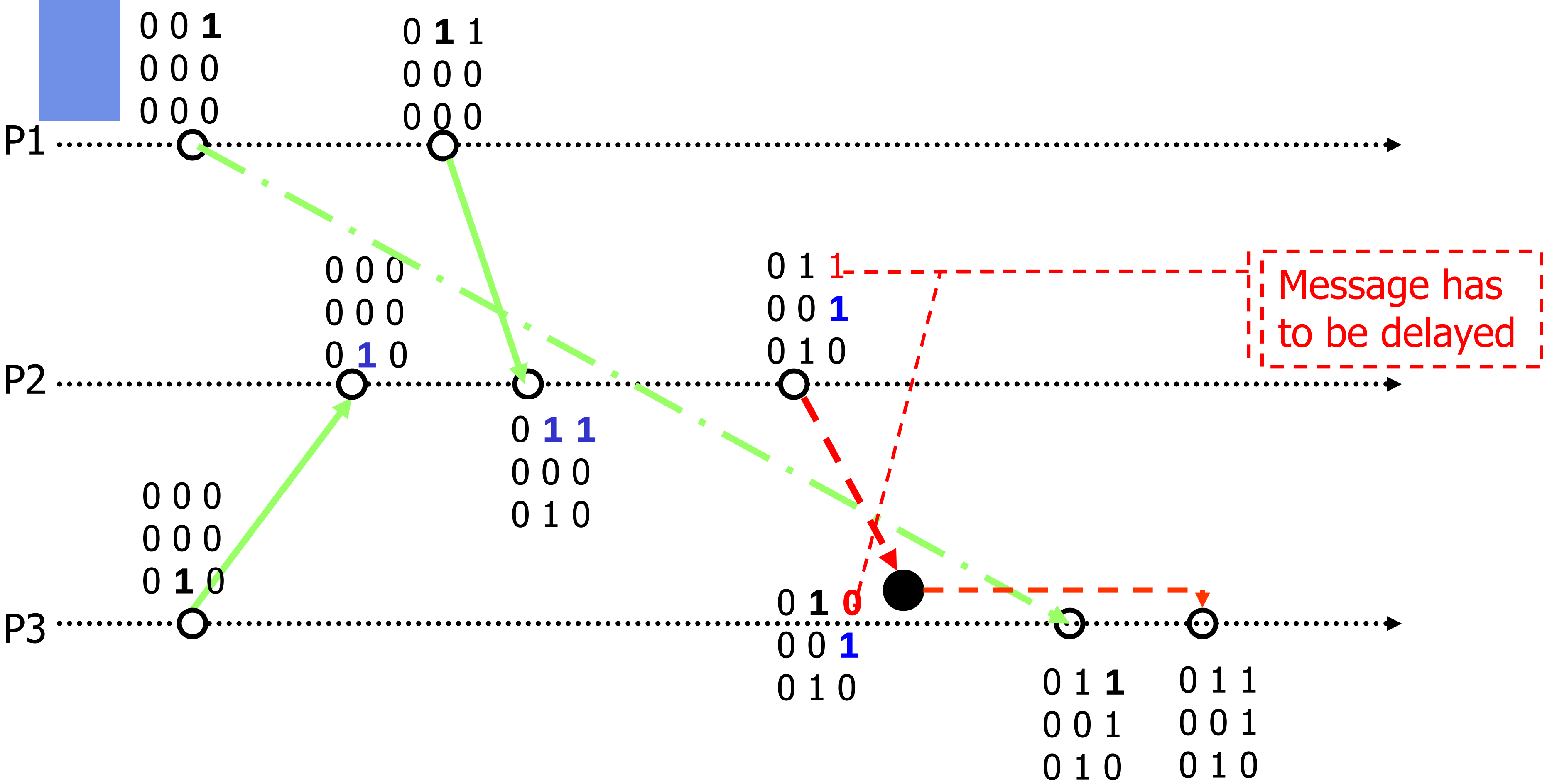
The incremented matrix M_i and sender number i are appended to the message, i.e. < i, M_i, message M> is sent to node j

Upon receiving a message (with Matrix M_i) at node P_j first this P_j updates its matrix M_j as follows:

1. $\forall$ k,l $\in$[1,n], l $\neq$ j: $M_j[k,l] = \max\{ M_j[k,l] , M_i [k,l] \}$
2. Increment $M_j[i,j]$, i.e. regard the current message
3. Delay this message, i.e. queue it before delivering to the application, until the following holds:

   $\exists$ $k \in [1,n]$, $M_j[k,j] < M_i[k,j]$

Example


0 0 1
0 0 0
0 0 0
0 1 1
0 0 0
0 0 0
P1
0 0 0
0 0 0
0 1 0
0 1 1
0 0 1
0 1 0
Message has
to be delayed
P2
0 1 1
0 0 0
0 1 0
0 0 0
0 0 0
0 1 0
P3
0 1 0
0 0 1
0 1 0
0 1 1
0 0 1
0 1 0
0 1 1
0 0 1
0 1 0

Global State

Chandy/Lamport: Distributed Snapshots: Determining Global States of DS

http://research.microsoft.com/users/lamport/pubs/chandy.pdf

Dijkstra: Comments on Chandy/Lamport/Misra Algorithm

http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD864.html

Global State

<u>Global state of a DS?</u>

Consists of:

- Local state of each node (process of an application system)
- Messages in transit since recording each local state

<u>Local state?</u>

- Dependent on what we are interested in, e.g.
  - references in use for distributed garbage collection
  - wait conditions in case of distributed deadlock detection

<u>Problem:</u>

- Due to lack of a unique global clock $\Rightarrow$ it is hard to get a time consistent snapshot of all local states, i.e. locals states are recorded in some unpredictable fashion

Snapshot Problem

Why interested in a global state?

Suppose computation of a distributed application has stopped on each involved node

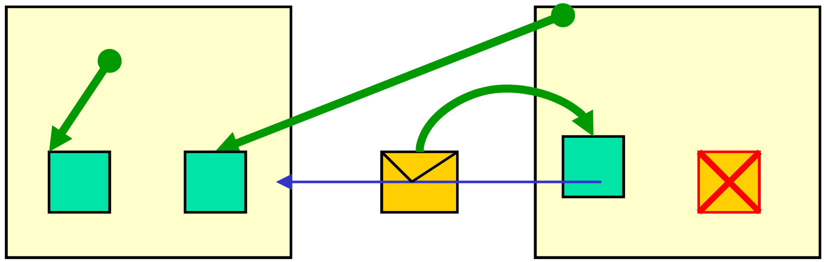
Distinguish whether

⇒ a distributed application

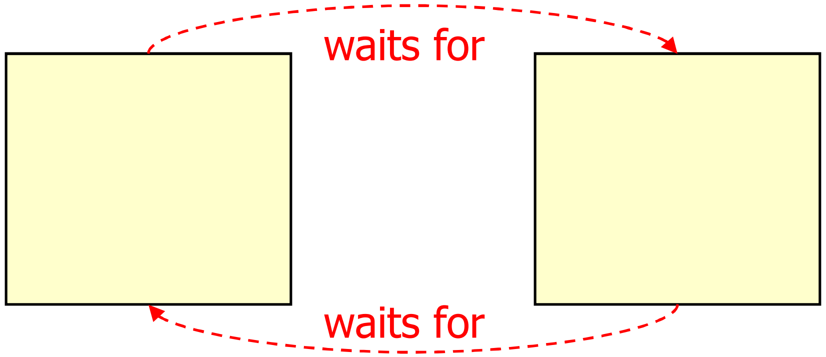
1. is **blocked** due to I/O
2. has **terminated** or
3. is **deadlocked**

Snapshot Problem

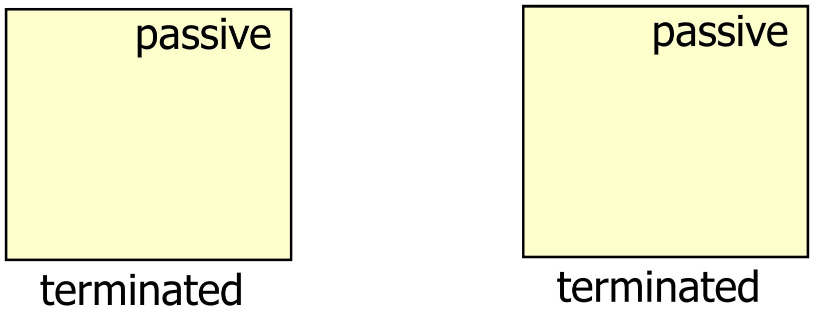
- Garbage collection



- Deadlock



- Termination problem



Why Consistent Global State?

How to combine information from multiple nodes, that the sampling reflects a global consistent state?

Problem:

- Local view is not sufficient
- Global view:
  - We need messages transfers to the other nodes in order to collect their local states
  - Meanwhile these local states can change again

Local History

- N processes P_i, $P := \{P_1, P_2, \ldots P_n\}$, for each P_i:
  - On a separate node n_i
  - Event series = history $h_i := \langle e_{i,1}, e_{i,2}, \ldots \rangle$
  - May be finite or not
- Observing a local history h_i up to event $e_{i,k}$ you get: prefix of history $h_{i,k} := \langle e_{i,1}, e_{i,2}, \ldots , e_{i,k} \rangle$
- Each $e_{i,k}$ is either a local or a communication event
- Process state:
  - State of P_i immediately before $e_{i,k}$ denoted $s_{i,k}$
  - State $s_{i,k}$ records all events included in history $h_{i,k-1}$
  - Hence, $s_{i,0}$ refers to P_i 's initial state

Global History and Global State

- Global history $h := h_1 \cup h_2 \cup \ldots \cup h_{n-1} \cup h_n$
- Similarly we can combine a set of local states to form a global state $S := (s_1, s_2, \ldots s_n)$
- However, which combination of local states is consistent?

Cuts

- Similar to the global state, we can define cuts based on k-prefixes:
- $C := h_{1,c1} \cup h_{2\,,c2} \cup \ldots \cup h_{n-1,cn-1} \cup h_{n,cn}$
- $h_{1,c1}$ is history up to and including event $e_{1,c1}$
- The cut C corresponds to the state

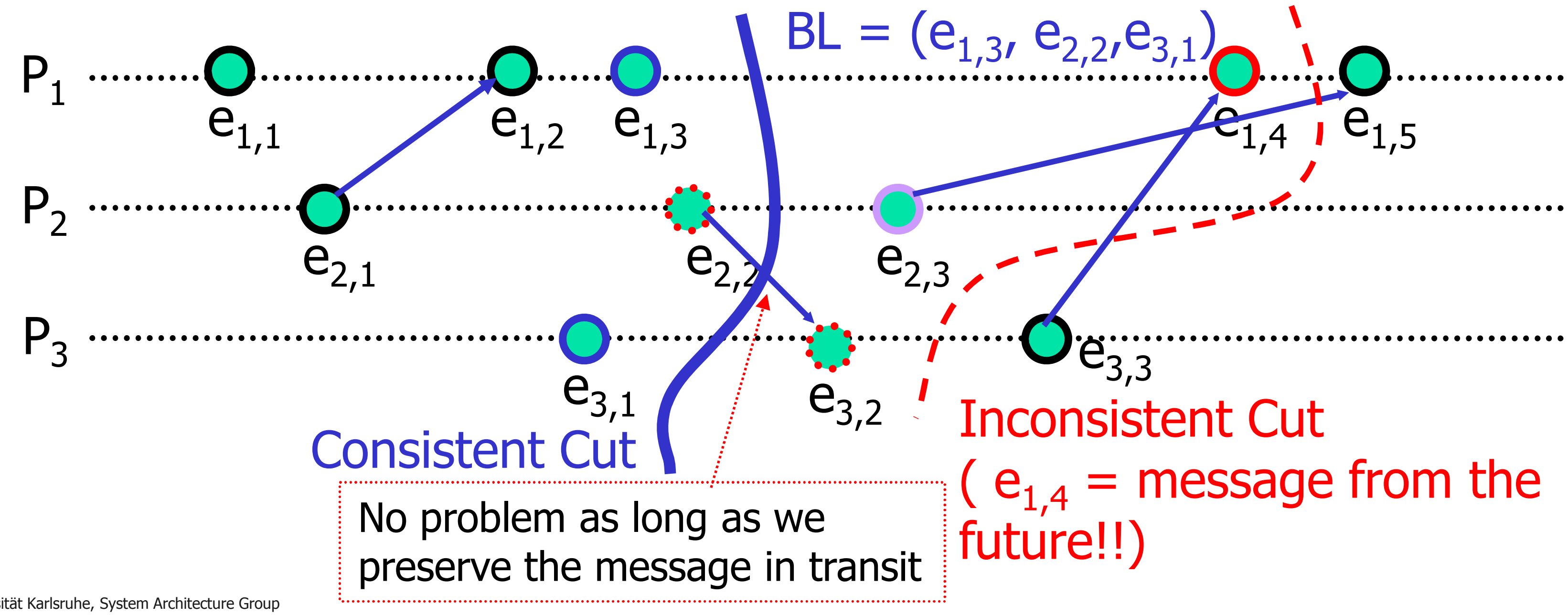
  $S = (s_{1,c1+1}, s_{2,c2+1}, \ldots s_{n,cn+1})$
- The final events in a cut are its frontier or its border line :

  $BL = \{e_{i,ci} \mid i \in \{1,2, \ldots n\}\}$

Distributed Snapshots

- Global state of system S:
  $S := (s_{1,c1}, s_{2,c2}, \ldots, s_{n,cn})$
  with the border line:
- $BL := (e_{1,c1}, e_{2,c2}, \ldots, e_{n,cn})$

Events have already happened



Consistent Cuts

- We call a cut C consistent iff for all events $e' \in C: e \rightarrow e'$ implies $e \in C$

- A global state is consistent if it corresponds to a consistent cut

Remark:

- We can characterize the execution of a system as a sequence of consistent global states

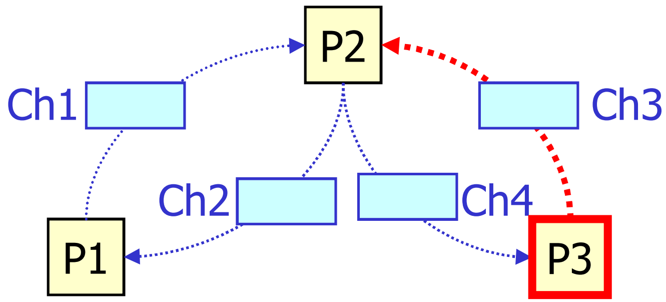
Linearization

- A global history that is consistent with the “happened before” relation is also called a linearization or consistent run
- A linearization only passes through consistent global states
- A state S’ is reachable from state S’ if ∃ a linearization that passes through S and S’

Chandy/Lamport Algorithm[1]

Assumptions:

1. No process failures, no message losses
2. Sequence of received messages is the same as sequence of sent messages
3. Bidirectional channels with FCFS property
4. Network is a strongly connected graph
   - From each process there is a connection path to each other process



[1]published 1985

Chandy Lamport Algorithm (2)

- Each process can initiate CLA to get a new global state
- CLA also regards states of the communication channels
- 2 types of messages
  - **Marker messages**
  - Application messages
- First marker message is for saving process state
- Second marker message is for saving channel state

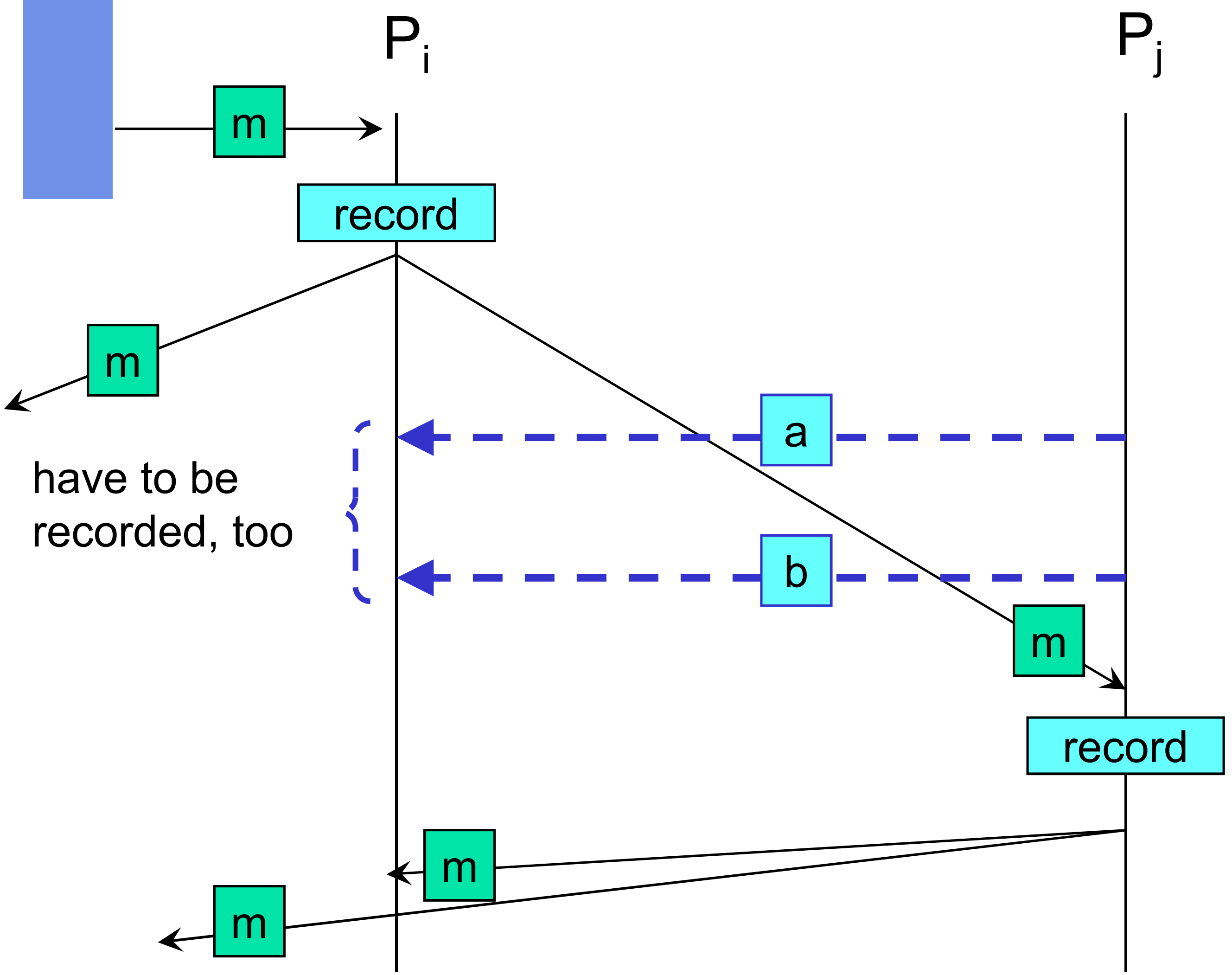
Principle of Operation

- Initially broadcast a marker message that contains a unique snapshot id (e.g. initiator id + sequence #) in order to differ from concurrent snapshot initializations
- Process p receiving a marker message for the first time:
  - If not yet done, save and record local state of receiver and install per input channel an empty message queue
  - Having recorded its local state, process p sends the marker message to all its other output channels
  - Continue with the local application process p
  - Each received application message is queued in the corresponding message queue

Principle of Operation

- Process p receiving the marker message at another input channel CH_i
  - Terminate the collection of messages at message queue MQ_i
  - Save channel state CH_i and record it to local state of p
  - If all incoming channels of p have been saved and recorded, send the aggregated local state of p to the initiator of the CLA

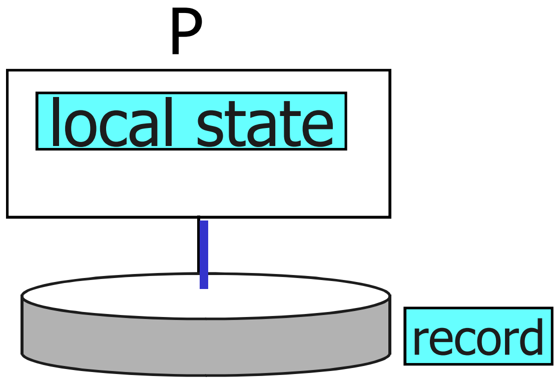
Principle of Chandy/Lamport



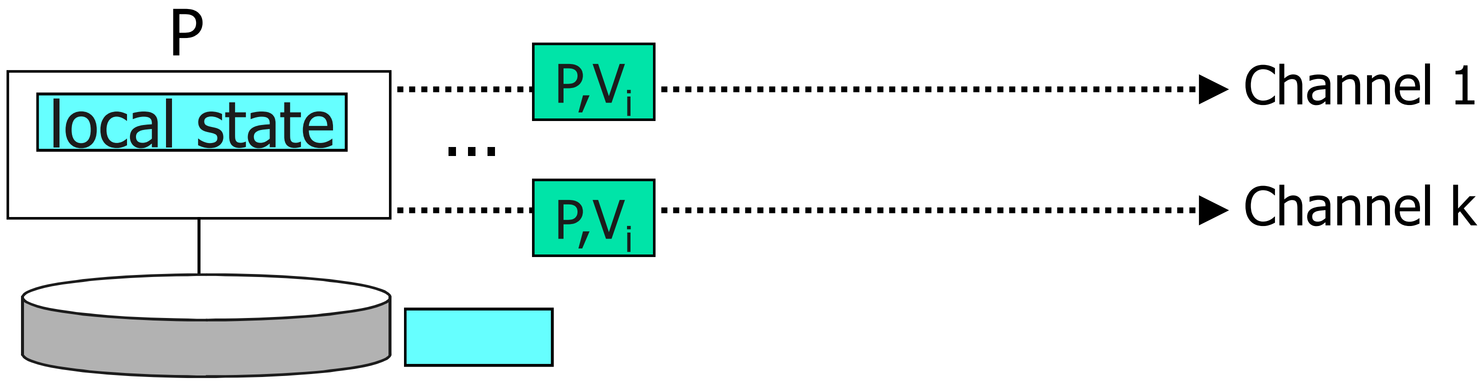
Distributed Snapshot

Task P wants to get a global view

1.

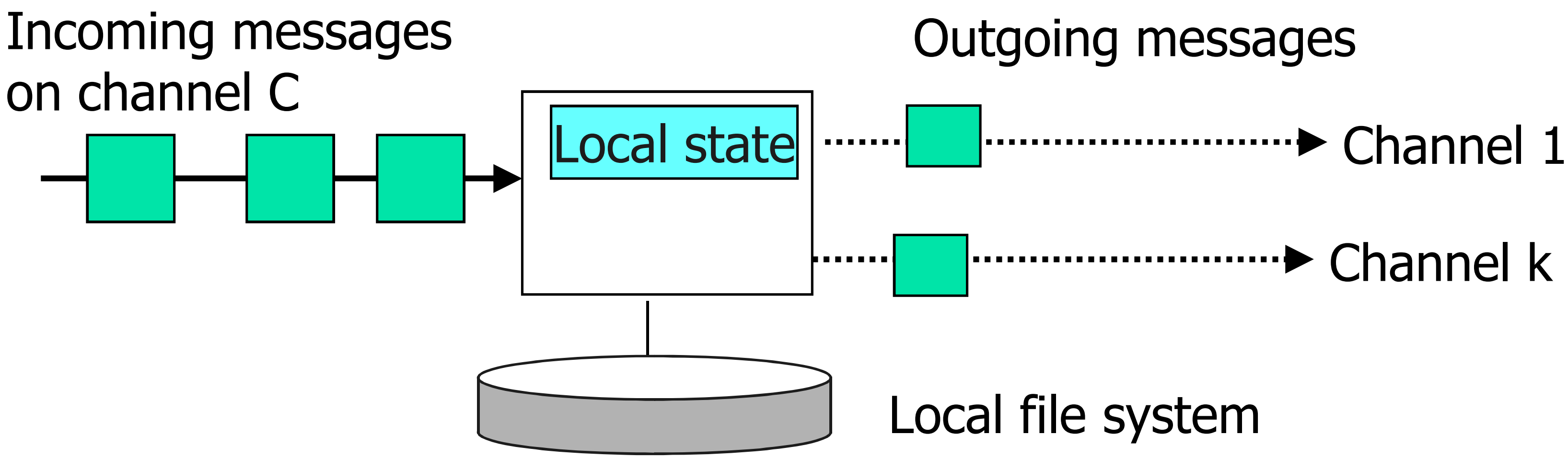


2.



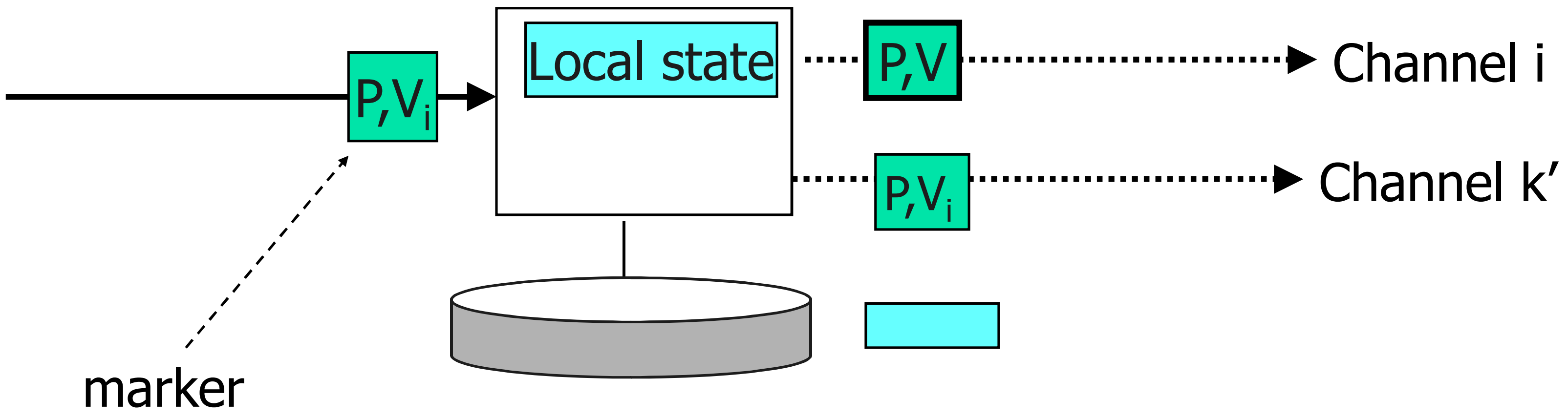
Distributed Snapshot

Receiver Task Q



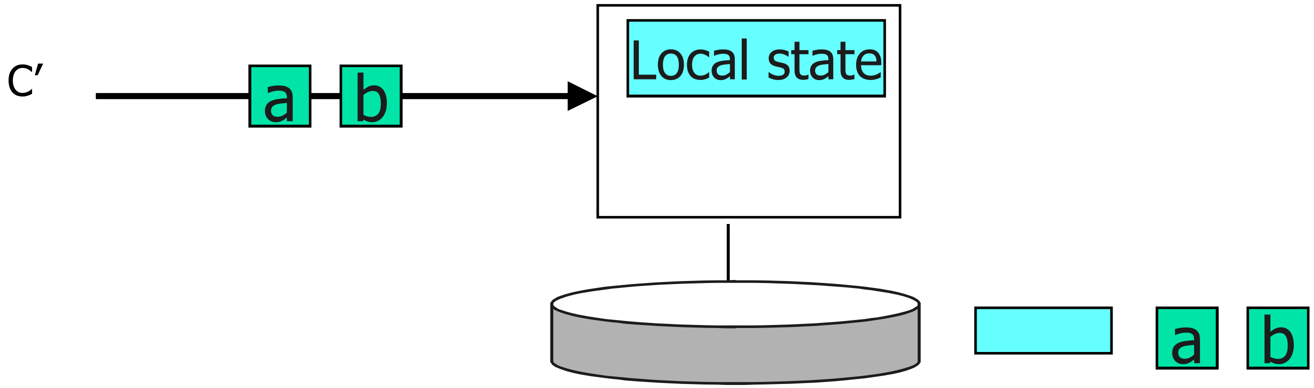
Distributed Snapshot

Case 3a: Q receives a marker for the very first time
on one of its channels ⇒
records its current local state,
sends marker messages on all output channels



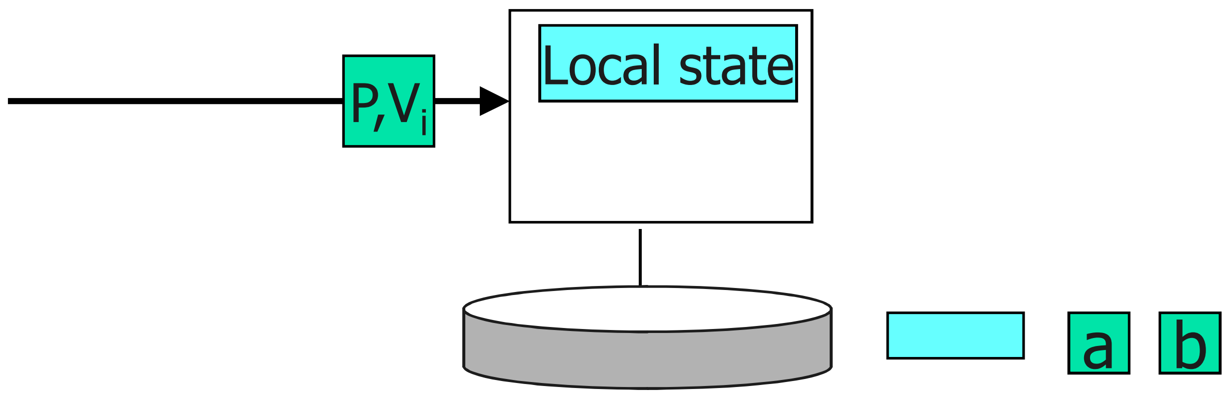
Distributed Snapshot

Case 3b: Q records all incoming messages on those channels C′ without a marker message up to now



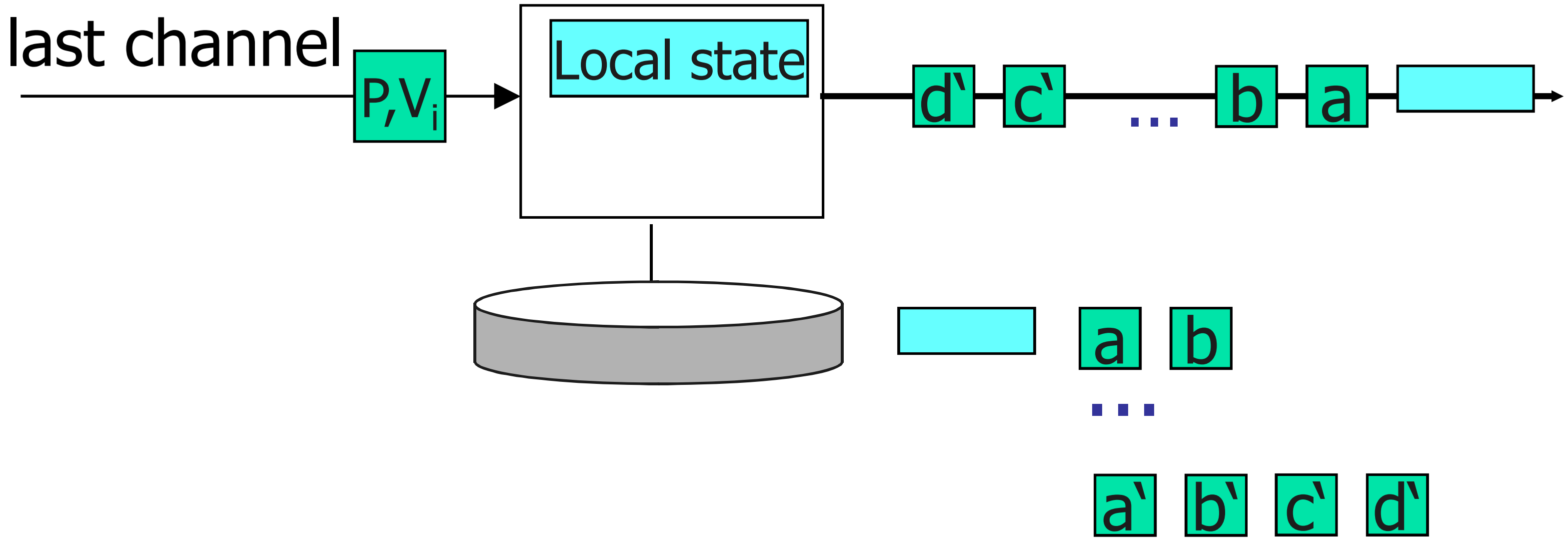
Distributed Snapshot

Case 3b: Q records all incoming messages on a channel C until marker message on channel C has arrived, then finishes recording at channel C

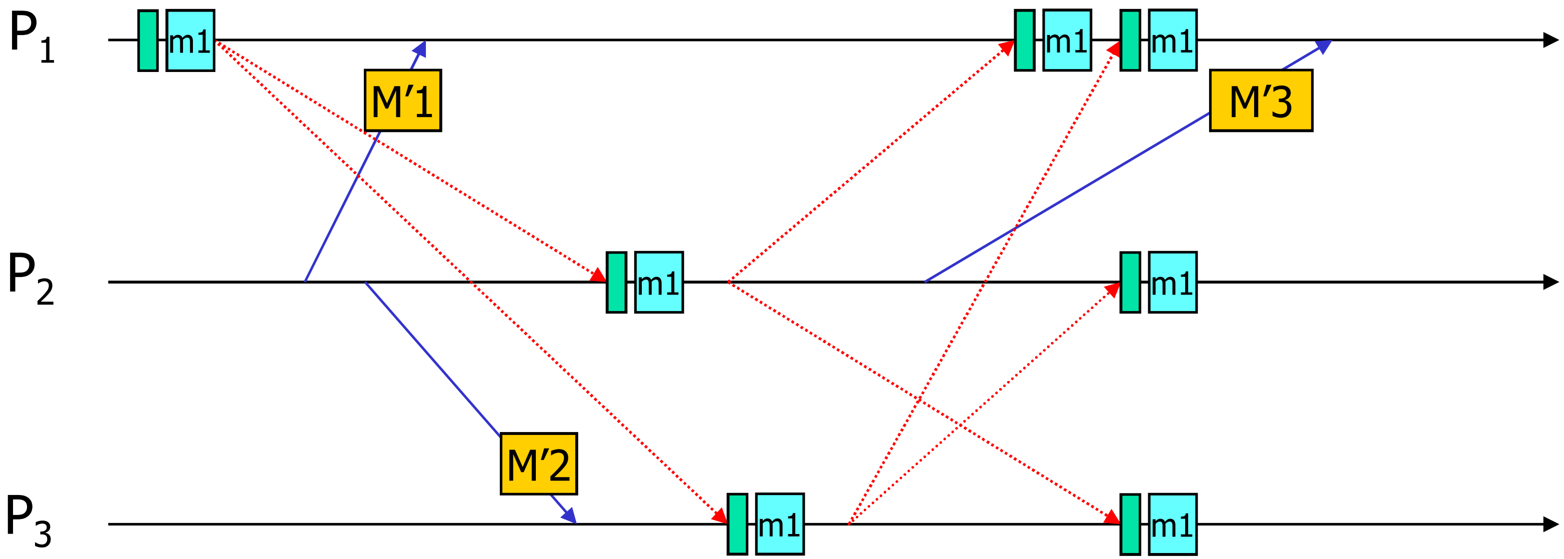


Distributed Snapshot

Case 3c:
Having received a marker message on each incoming channel, Q sends its accumulated state to the snapshot-initiator P



Example of Chandy & Lamport

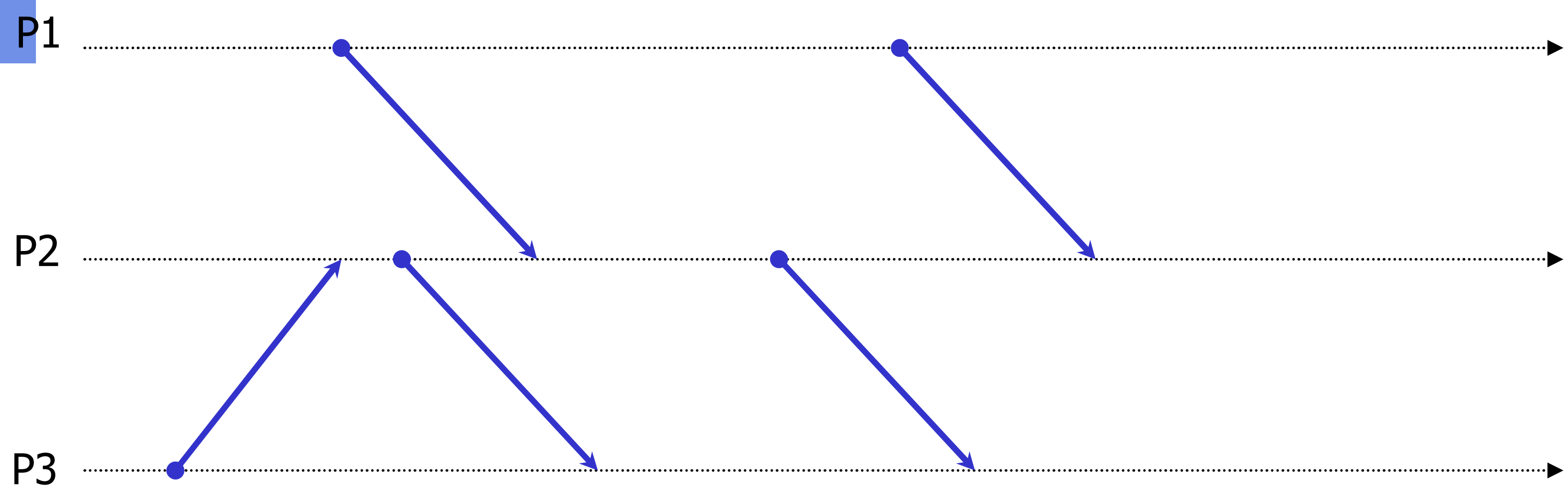


mi Marker message

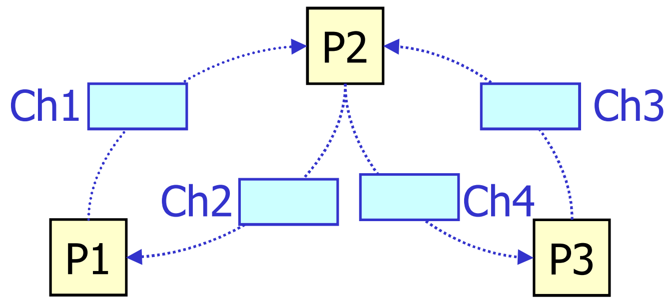
M'i Ordinary message

M'3 does not belong to the snapshot, each P_i has a channel to each other P_j, $i \neq j$

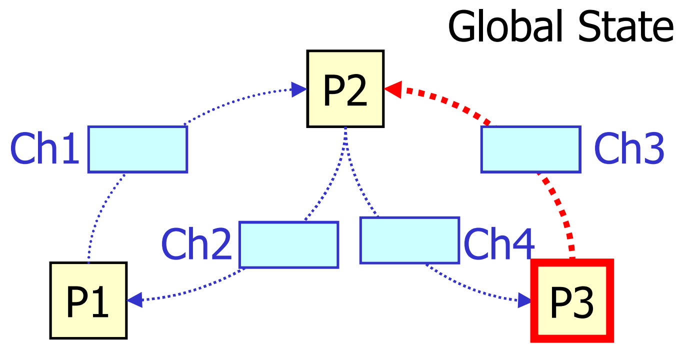
Example: Chandy-Lamport Algorithm

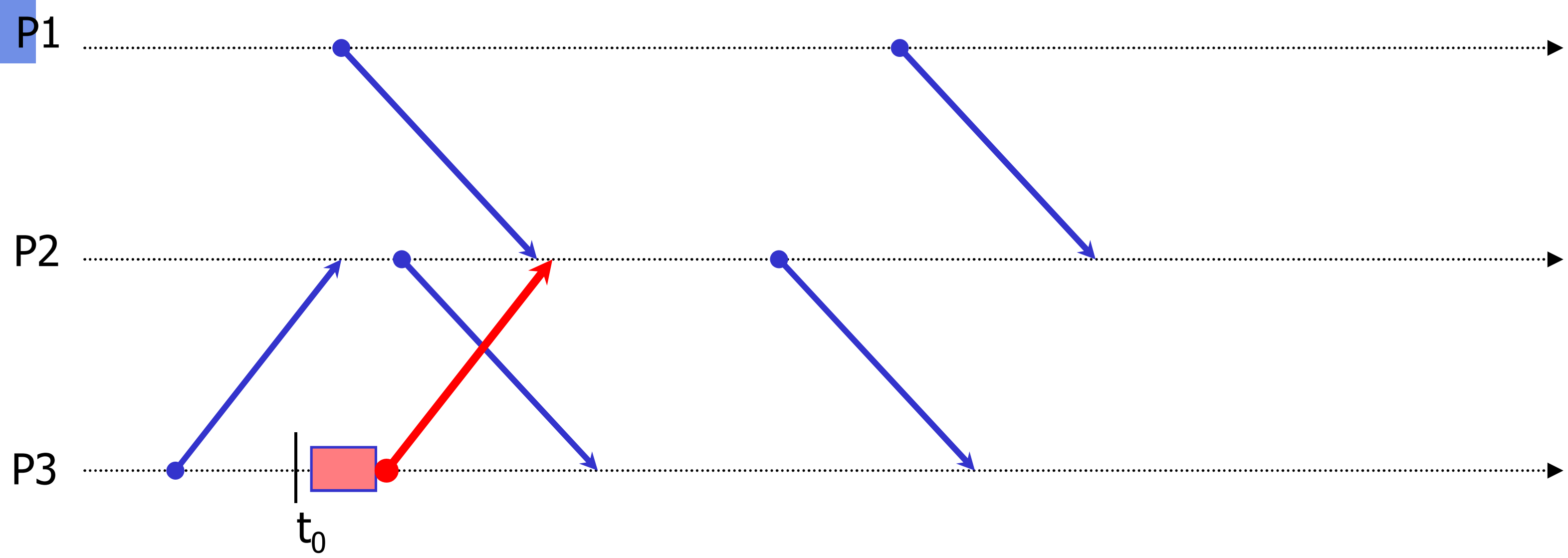


Application Messages



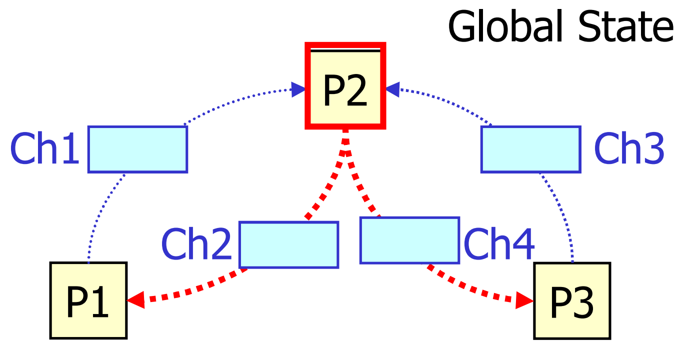
Example

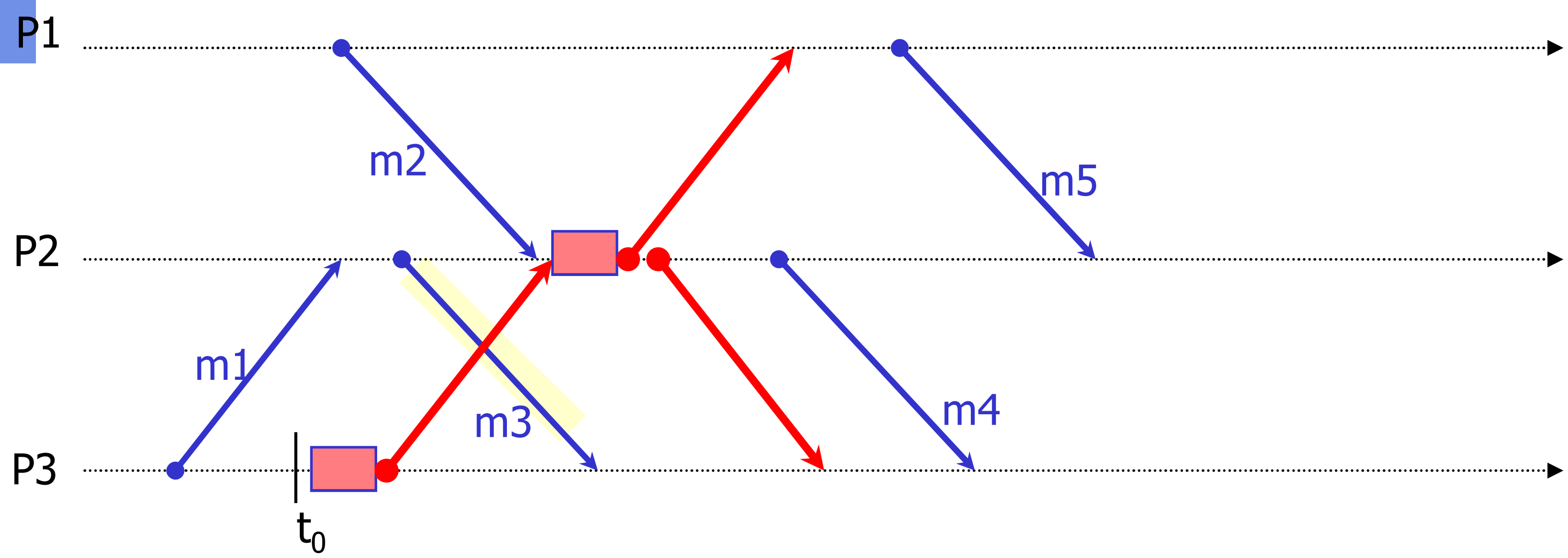




- P3 wants to start the algorithm at t_0 to get a global view it constructs an empty message queue to collect future application messages from P2 via Ch4
- P3 saves its local state and sends a marker message to P2 via channel Ch3

Example

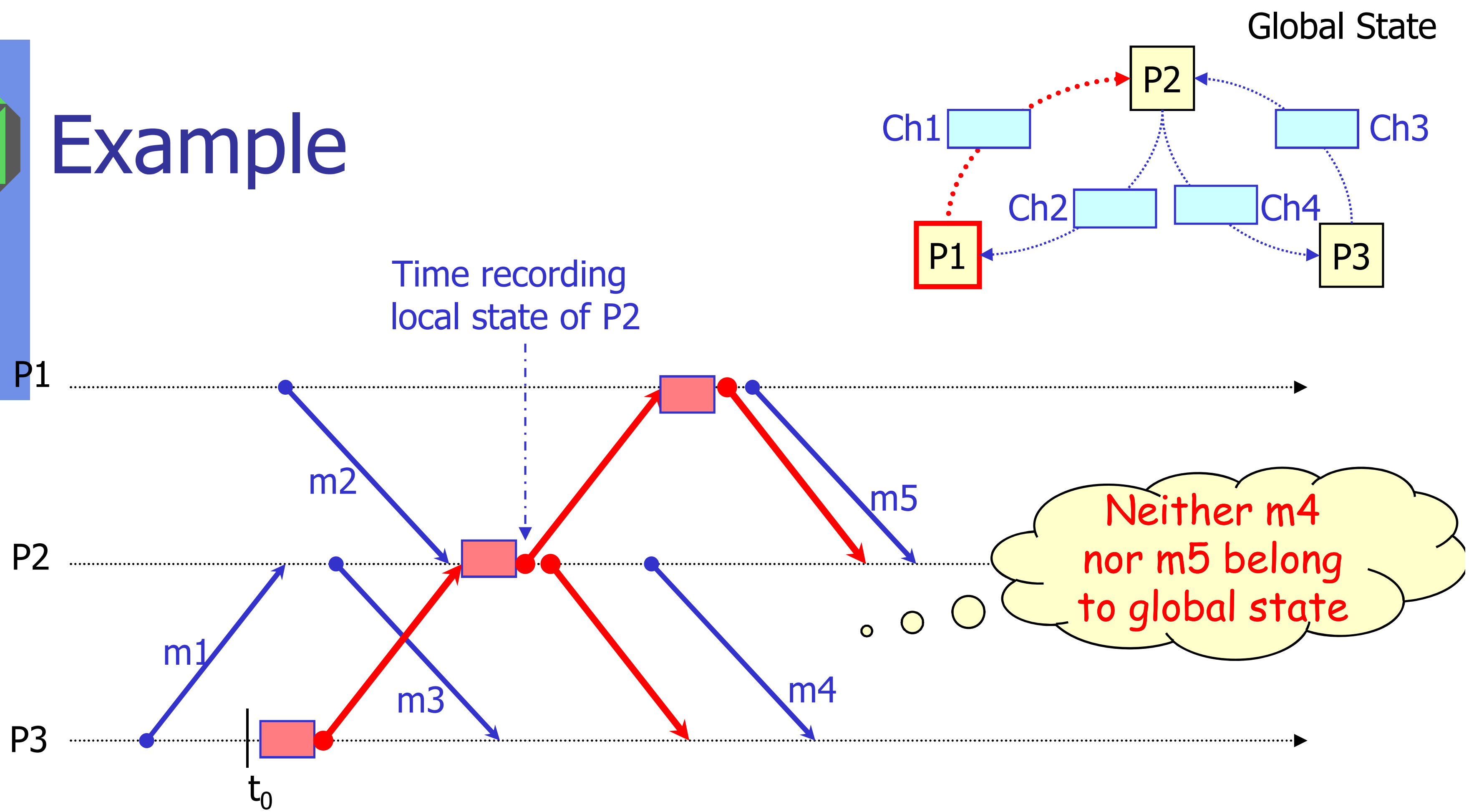




- P2 saves its local state and sends marker messages to P1 via Ch2 and to P3 via Ch4

- P3 adds to its local state that transient application message **m3** was received

Example



- P1 saves local state and sends a marker message to P2 & its local state to P2 (for forwarding it to P3)
- P2 receives last marker at one of its input channels Ch2 and forwards recorded local states of P1 and P2 to P3