# Synchronization

Process A

Process B

Process C

Process D

Process E

Process F

Process G

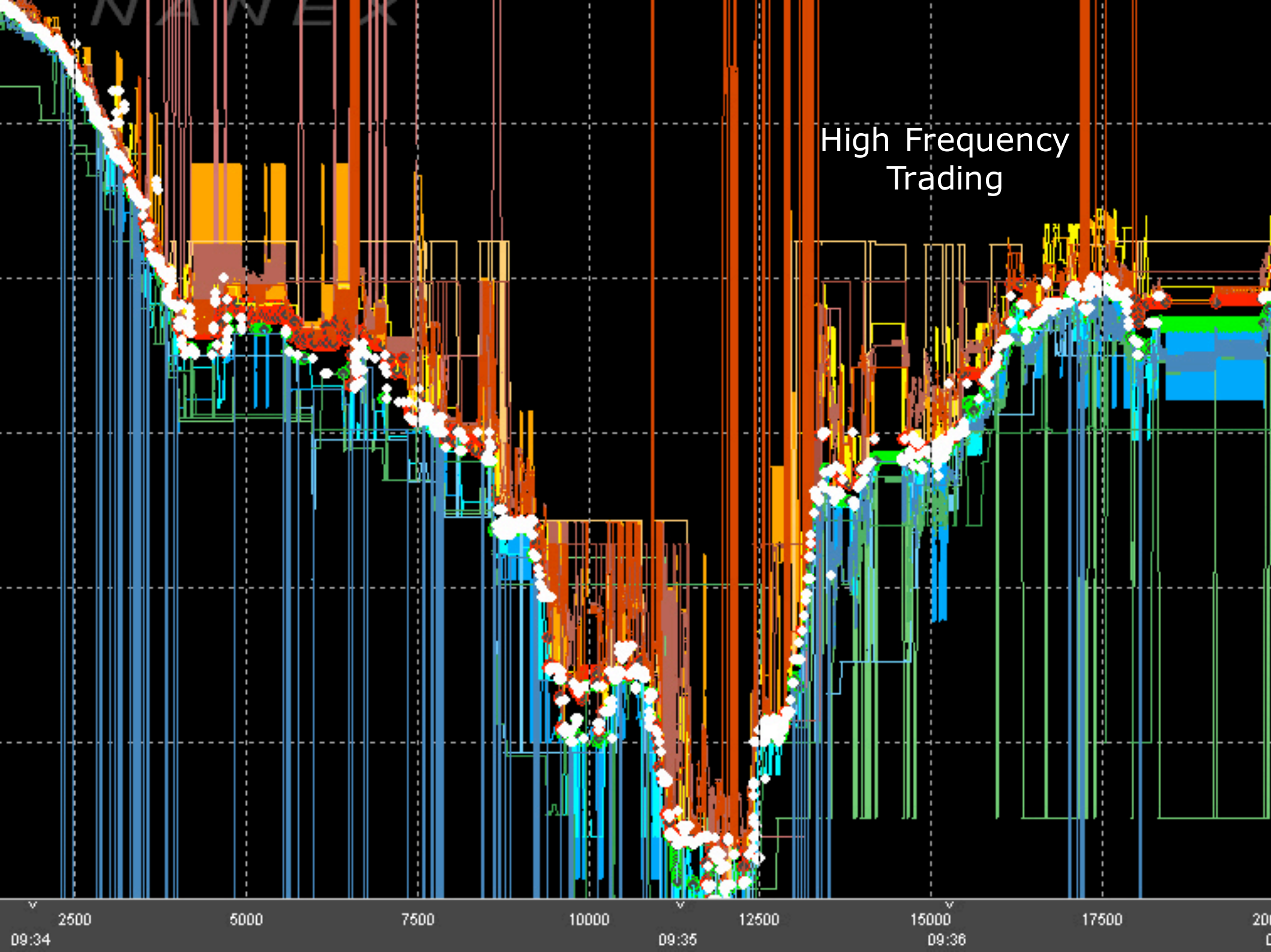
Process H

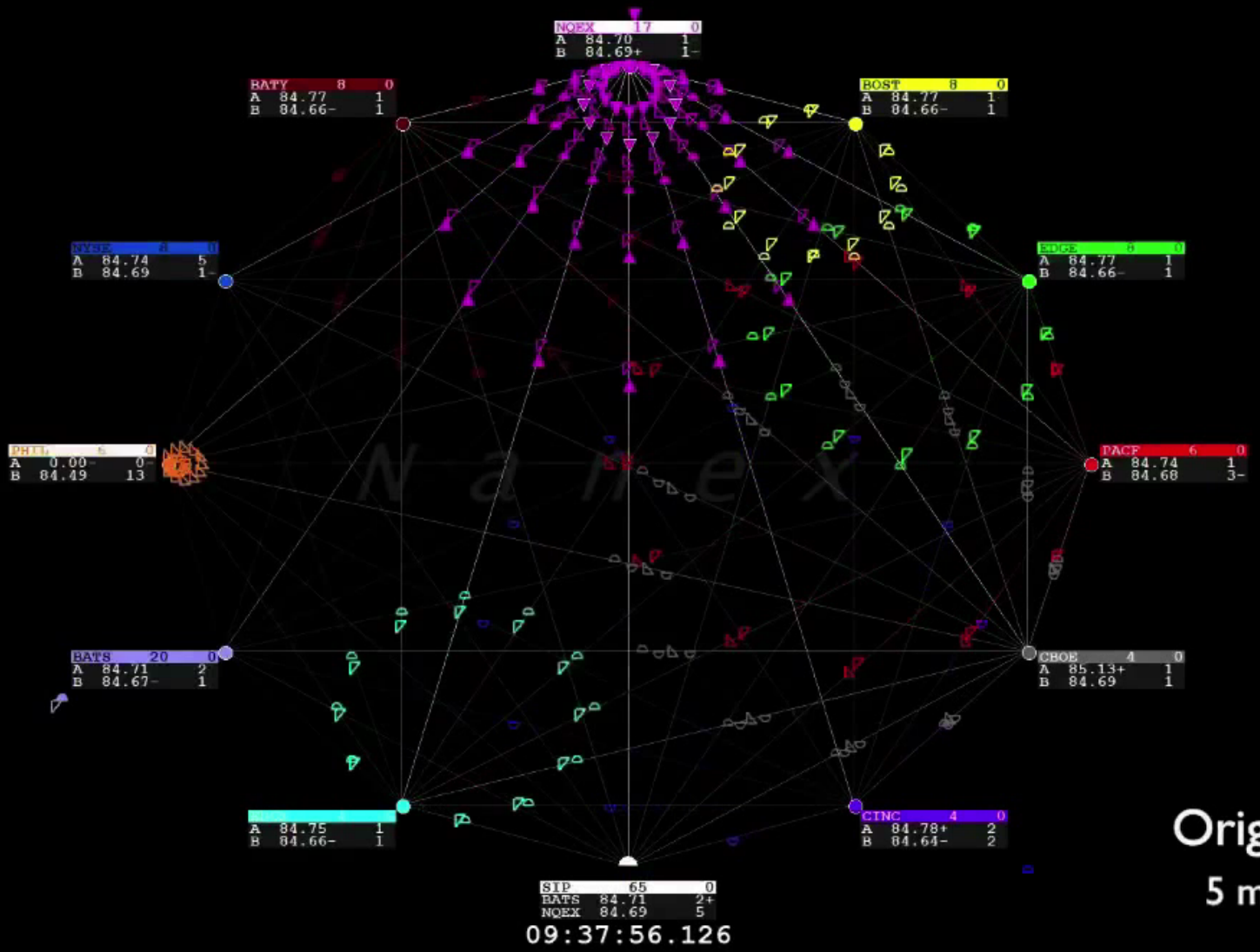
Process I

Process J

Process K

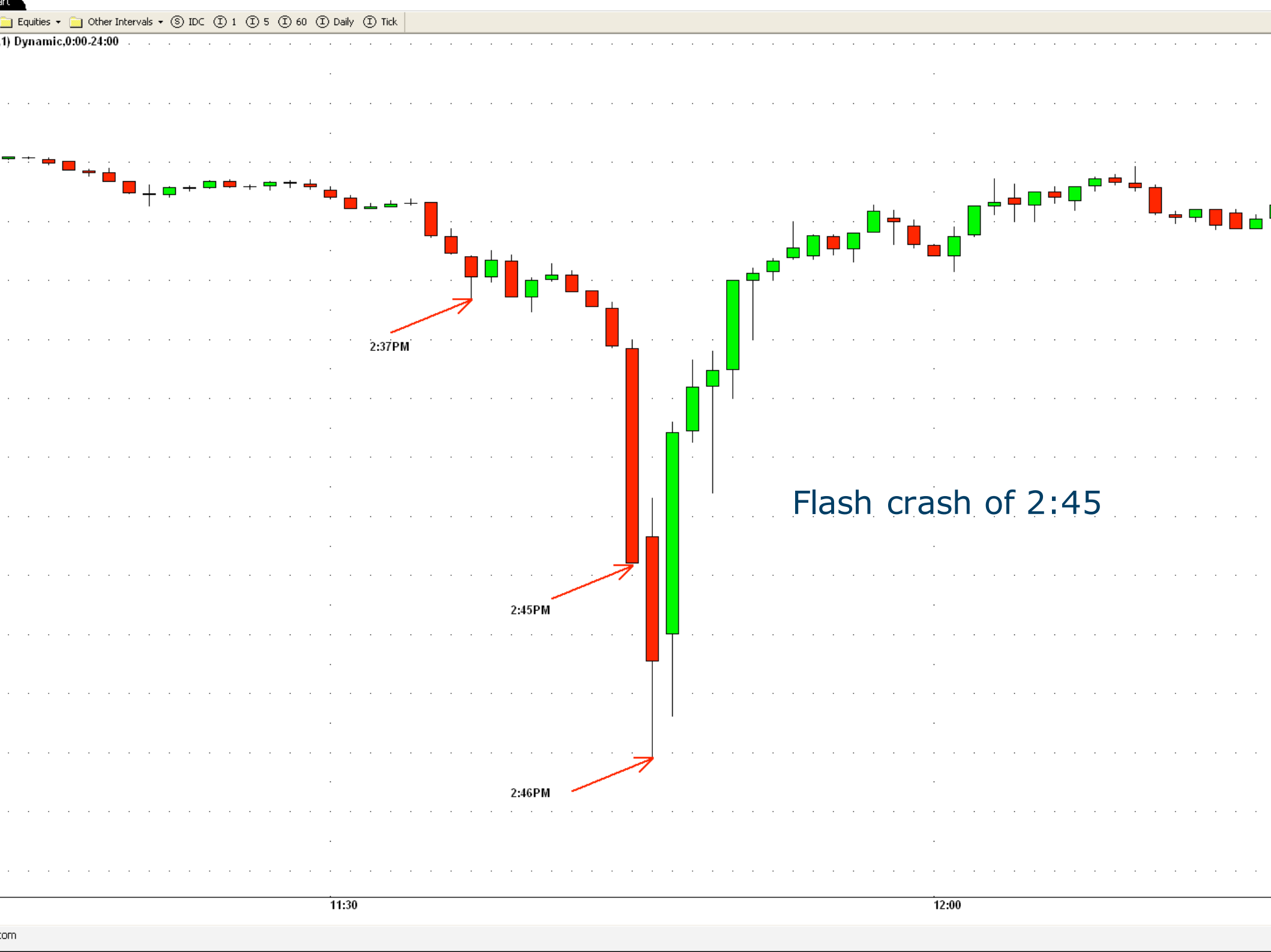
Process :


High Frequency Trading
2500
5000
7500
10000
12500
15000
17500
09:34
09:35
09:36




Original Clip

5 min 55 sec

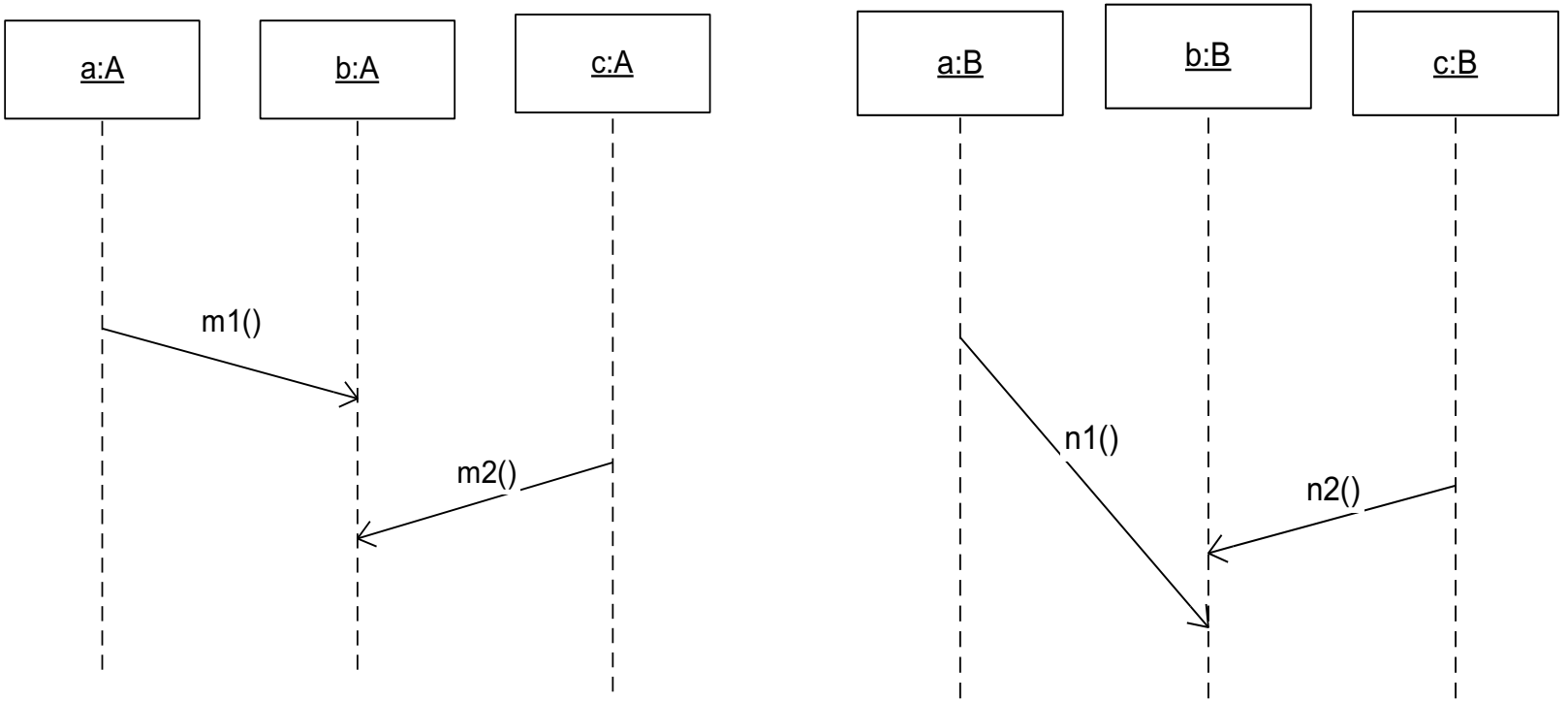

Equities
Other Intervals
IDC
1
5
60
Daily
Tick
1) Dynamic,0:00-24:00
2:37PM
2:45PM
2:46PM
Flash crash of 2:45
11:30
12:00

# Why synchronization?

**NO global state notion in a distributed system**

No global time notion

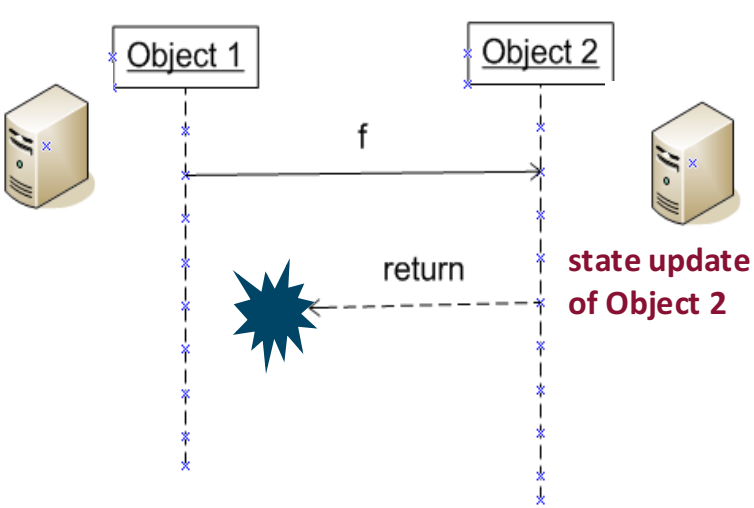
How to be sure about event ordering ?



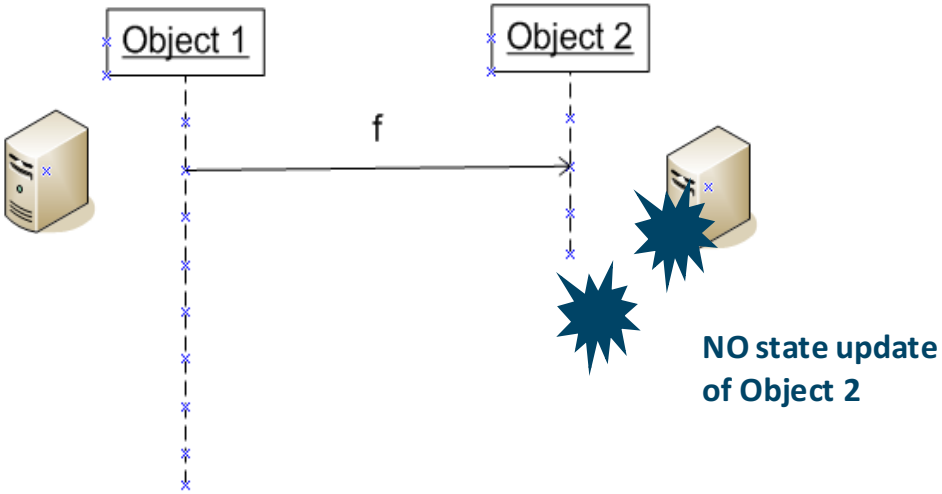
# Why synchronization?

## The origin of the problem

- Concurrency
- Inter-process communication with uncertain delay
- Impossible to distinguish between failing process and failing network link




communication
failure [1]




process
failure [2]

How can Object 1 distinguish between [1] and [2] ?

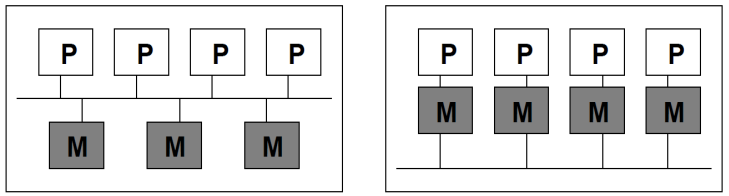
# Synchronization and coordination

## Distributed algorithms require time

- Algorithms designed to work in a distributed environment
  - Require synchronization and coordination
  - Implement synchronization and coordination

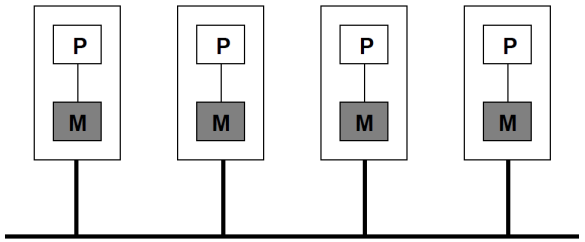
## Coordination and synchronization

- Uni-/multi-processors
  - All processes can coordinate and synchronize using a shared clock / memory




Uniform Nonuniform

- Multi-computer
  - Dangerous to pick a single master to coordinate! (reliability, scalability)

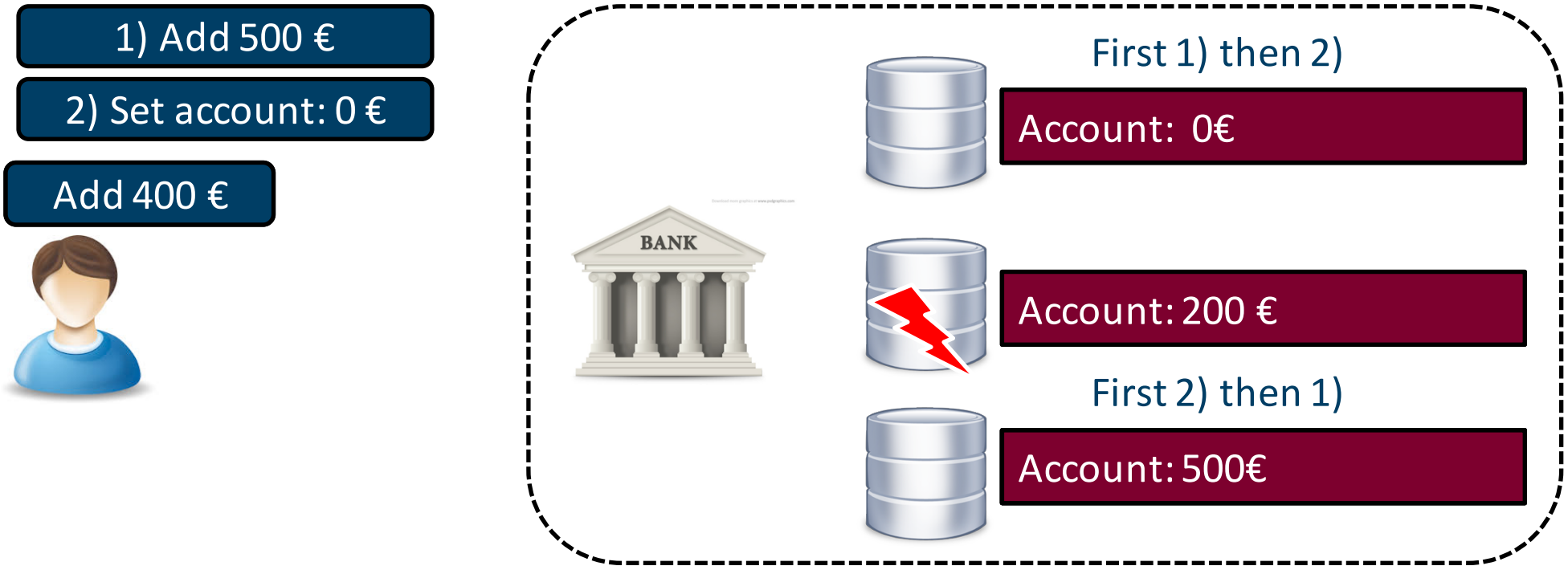


# Remember last week? Atomic multicast

**Multicasting in presence of process failures within a process group and dynamic membership within that group**

- Message is delivered to all **non-faulty** processes or to none
- All members should agree on the current group membership
- Totally ordered: all processes receive the same order

**Why so important? Guarantees consistency of the system**



# What is time?

**Physical time**

= Real time, related to solar time

Important when connection to "real" time is needed

Example: UTC, UNIX time,…

## 1 Physical clock synchronization

## 2 Logical clocks

**Logical time**

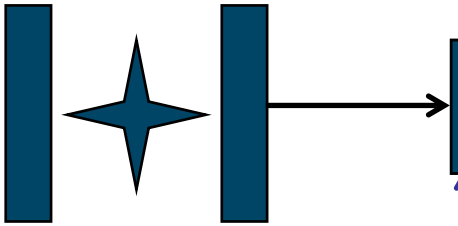
- = not related to solar time
- value only important for event ordering
- easier to realize (no specific hardware needed)

# 1 Physical clock synchronization

1. How do computers measure time?
2. Clock skew and clock drift
3. Time standards
4. Clock synchronization

# From crystal to time

Crystal with stable
oscillating frequency f



6 8 7 5 5 2 0 1

Digital counter

interrupt
when counter==0

clock
tick

interupt frequency
**H**=60/s

8 7 7 4 4 2 1 7

Digital counter
Start value

interupt service
routine

**100987650100019**

incrementing counter
clock value in memory
= #ms elapsed after **β**
→ **C(t)**

Clock chip

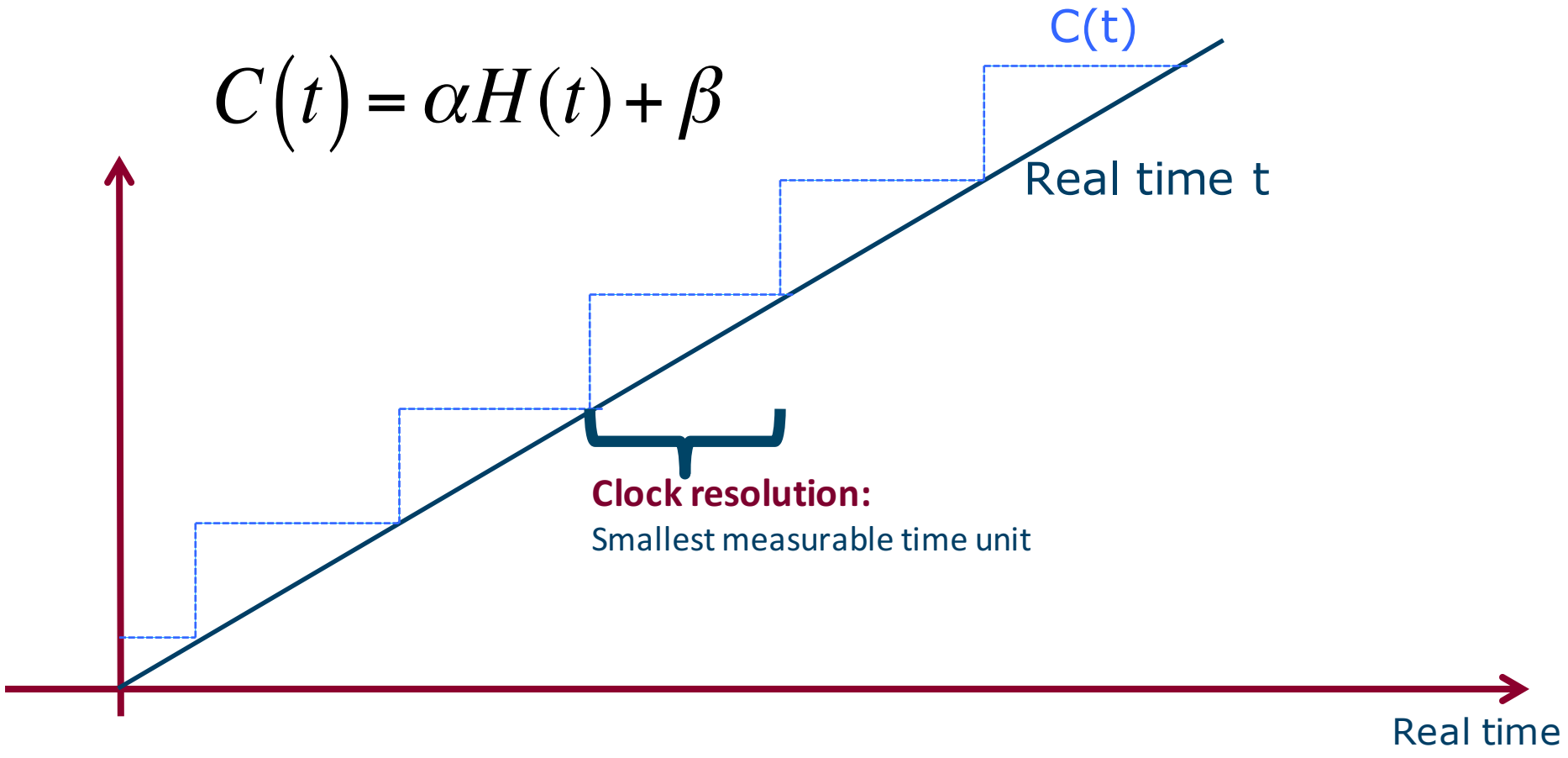
# From crystal to time

## Real time ?

- "Real" time = **t**
- Hardware time H(t)
- Ideally : **C(t) = t**

**C(t) can be computed from hardware time H(t), scaling factor $\alpha$ and $\beta$**

$$C(t) = \alpha H(t) + \beta$$



# Drift : clocks get out of pace …

**Crystal oscillation frequency NOT stable**

- ambient temperature
- fabrication tolerances
- packaging issues (strain on the crystal)
- remaining battery power

**This leads to**

- changing oscillation frequency
- changing interrupt frequency

  $\Delta H/H \approx 10^{-5}$

**Drift rate = rate clock C(t) looses track with t**

drift rate specification ($\rho \geq 0$)

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

# Skew: clocks keep different times

## Drift rate values

- "Ordinary" quartz clock : 1 s in 11-12 days -> $\rho=10^{-6}$ s/s
- "High precision" quartz clock: $\rho=10^{-7}$ a $10^{-8}$ s/s

## Clock skew

= difference in time reading between two clocks

skew

$C_1(t)$ $C_2(t)$

$$skew_{C_1,C_2}(t) = |C_1(t) - C_2(t)|$$


8:00:00


8:00:00

Sept 18, 2013
8:00:00


8:01:24

Skew = +84 seconds

+84 seconds/35 days

Drift = +2.4 sec/day

Oct 23, 2013

8:00:00

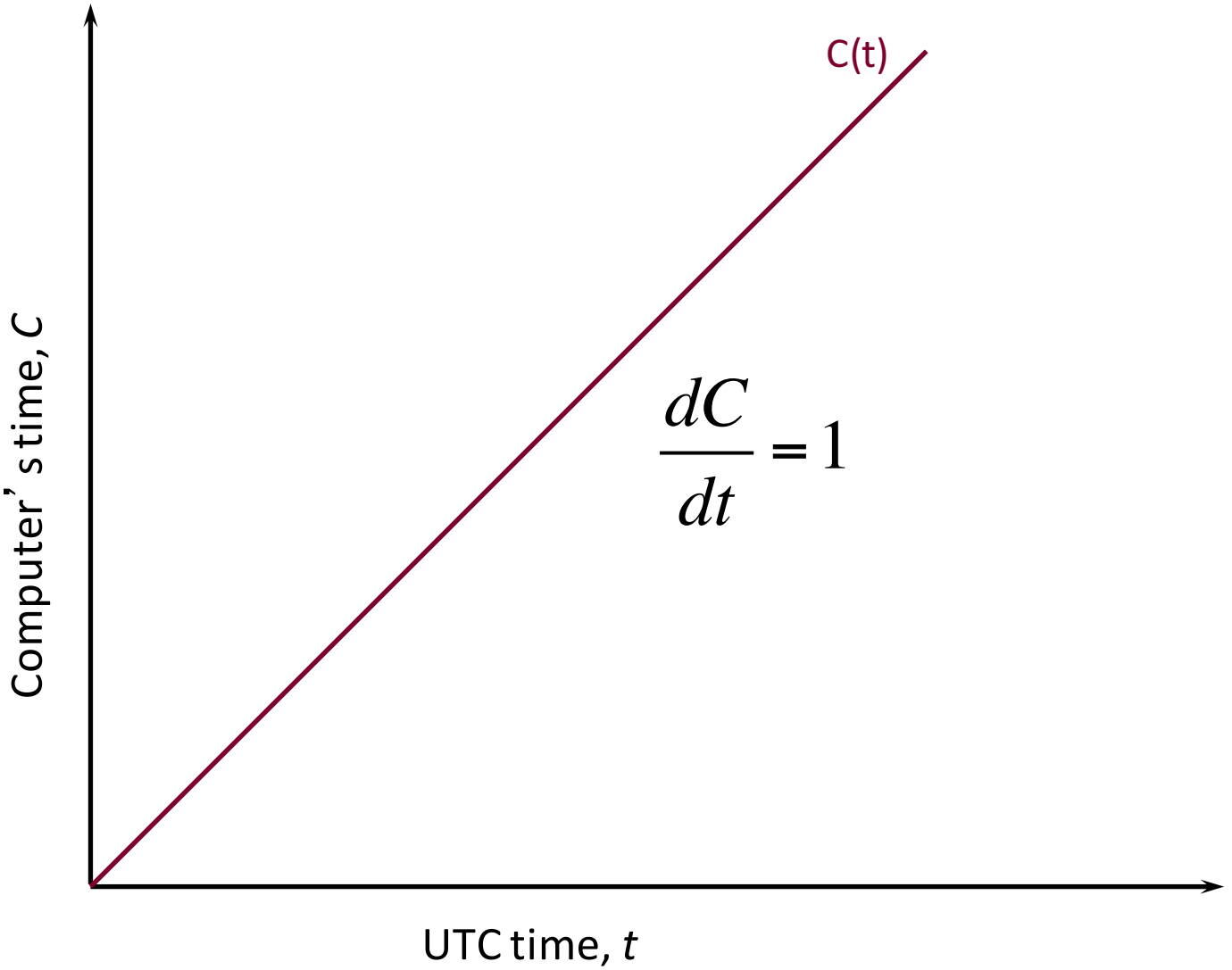

8:01:48

Skew = +108 seconds

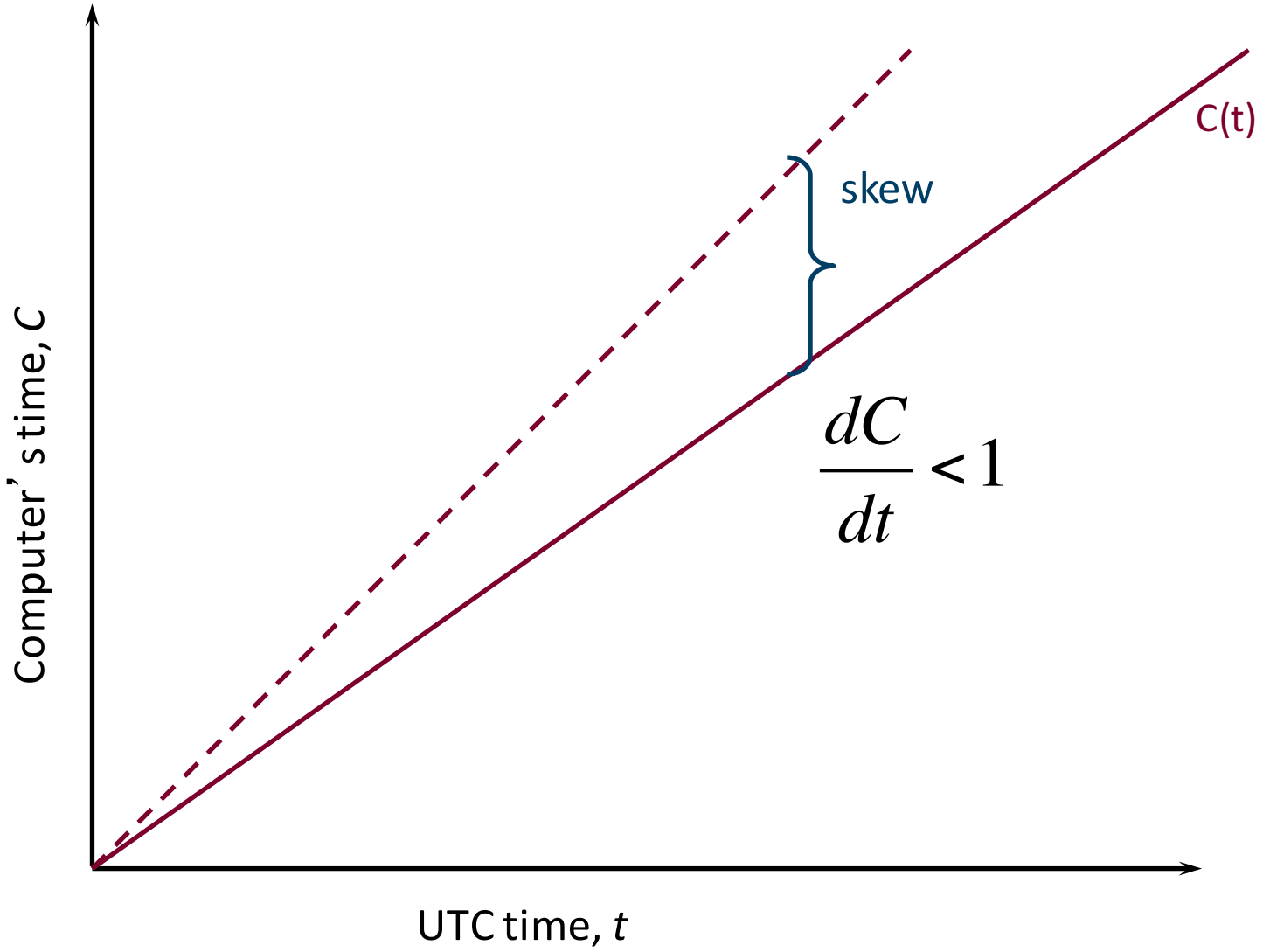
+108 seconds/35 days

Drift = +3.1 sec/day

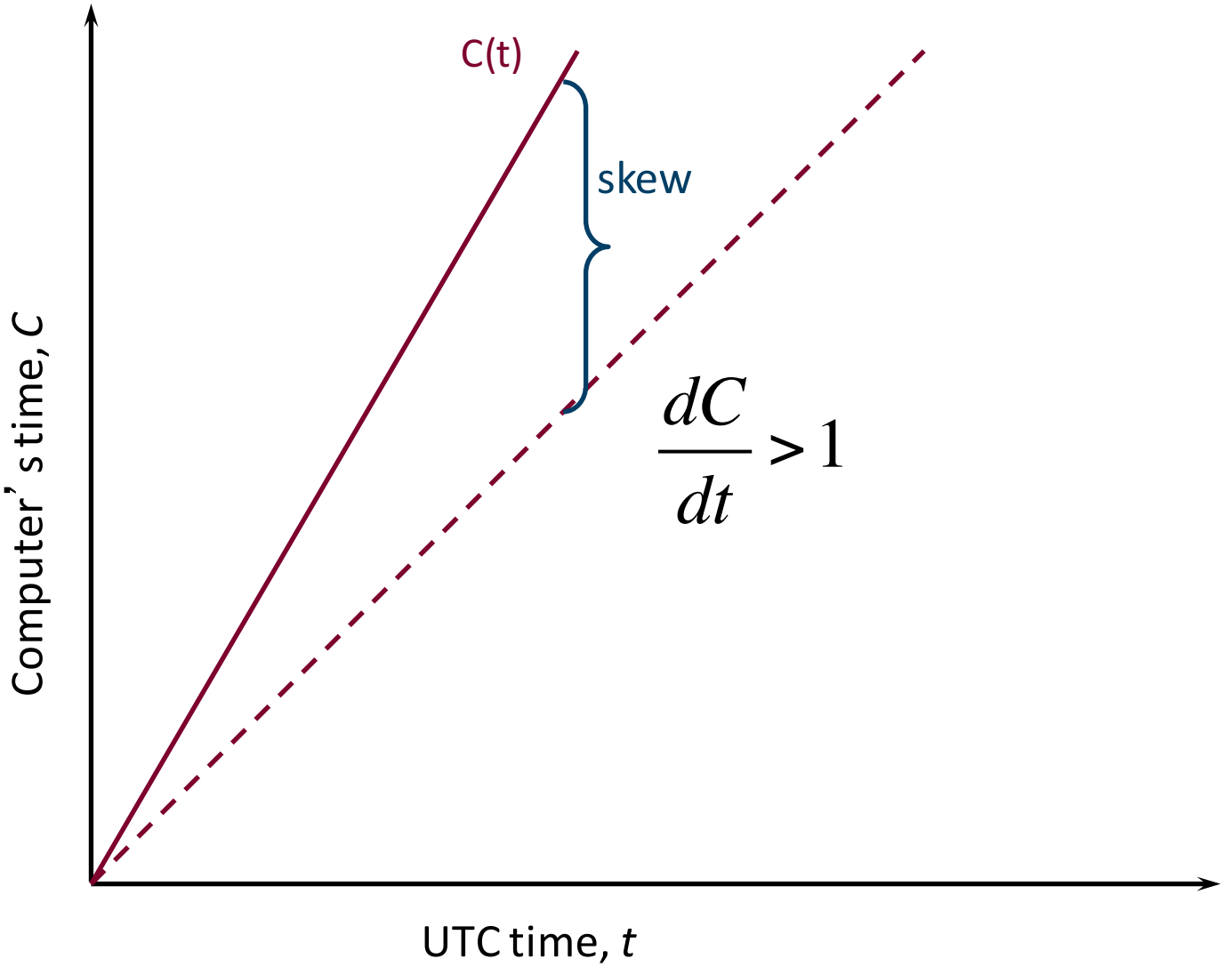
# Perfect clock



# Drift with slow clock



# Drift with fast clock



# How to adjust clocks?

**Be careful with jumping in time (especially going back)**

Illusion of time moving backwards can confuse message ordering

Y2K Bug

GNU Make

**Solution: gradual clock correction**

$$S(t) = AC(t) + B$$

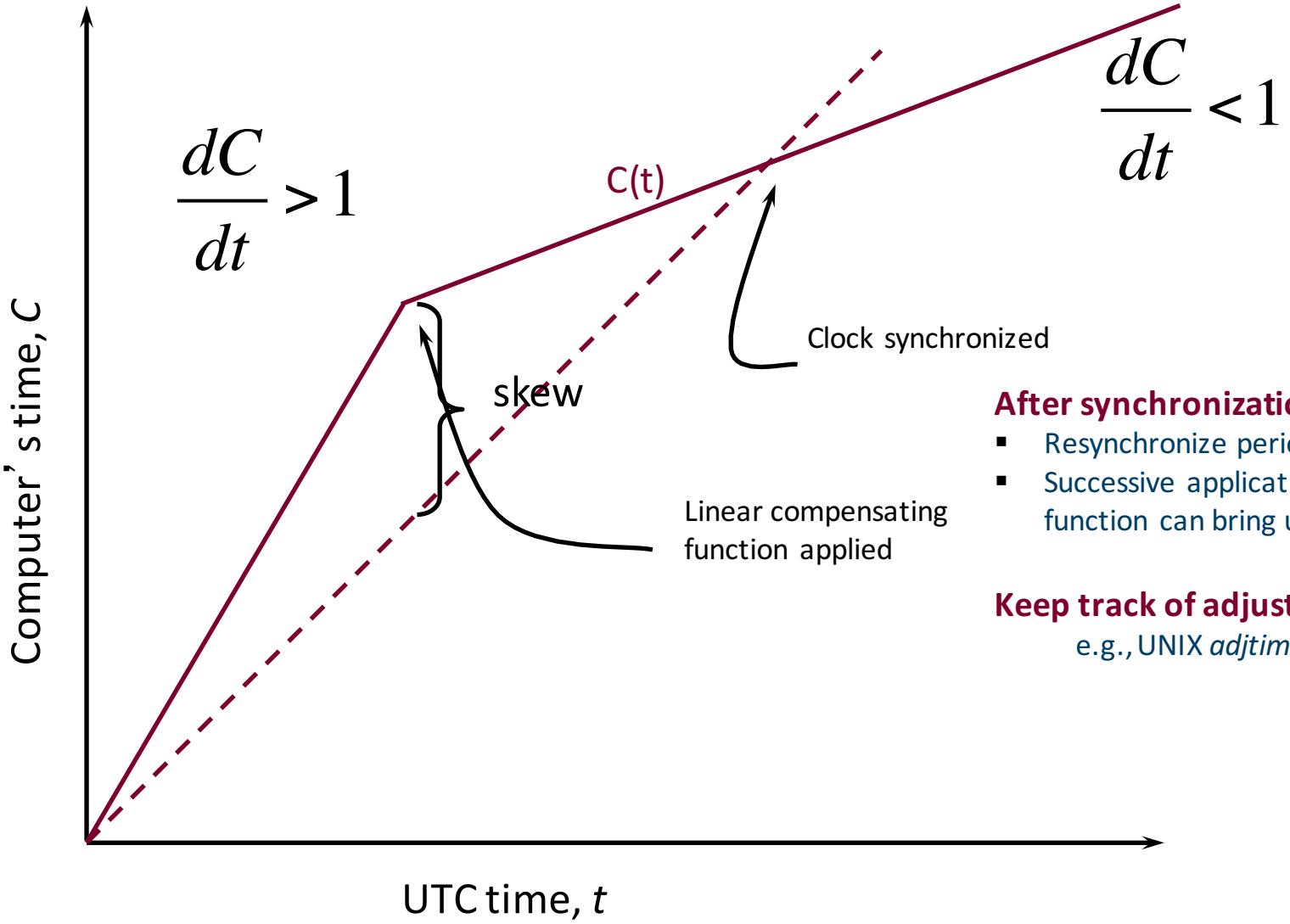
Linear compensating function, changing slope of system time
Choosing A and B such that S(t) is continuous and represents the correct time after some delay.

**OS can do this: Change rate at which it requests interrupts**

e.g.: if system requests interrupts every 17 msec but clock is too slow:

→ request interrupts at (e.g.) 15 msec

# Compensating for a fast clock




**After synchronization period is reached**

- Resynchronize periodically
- Successive application of a second linear compensating function can bring us closer to true slope

**Keep track of adjustments and apply continuously**

e.g., UNIX *adjtime* system call

# What is a correct clock?

**Different possible correctness conditions**

(i) Bounded drift rate

(ii) Monotonicity condition

(iii) Hybrid condition


**Faulty clock, either**

(i) stopped ticking (crashed)

(ii) still ticks, but does not obey correctness condition

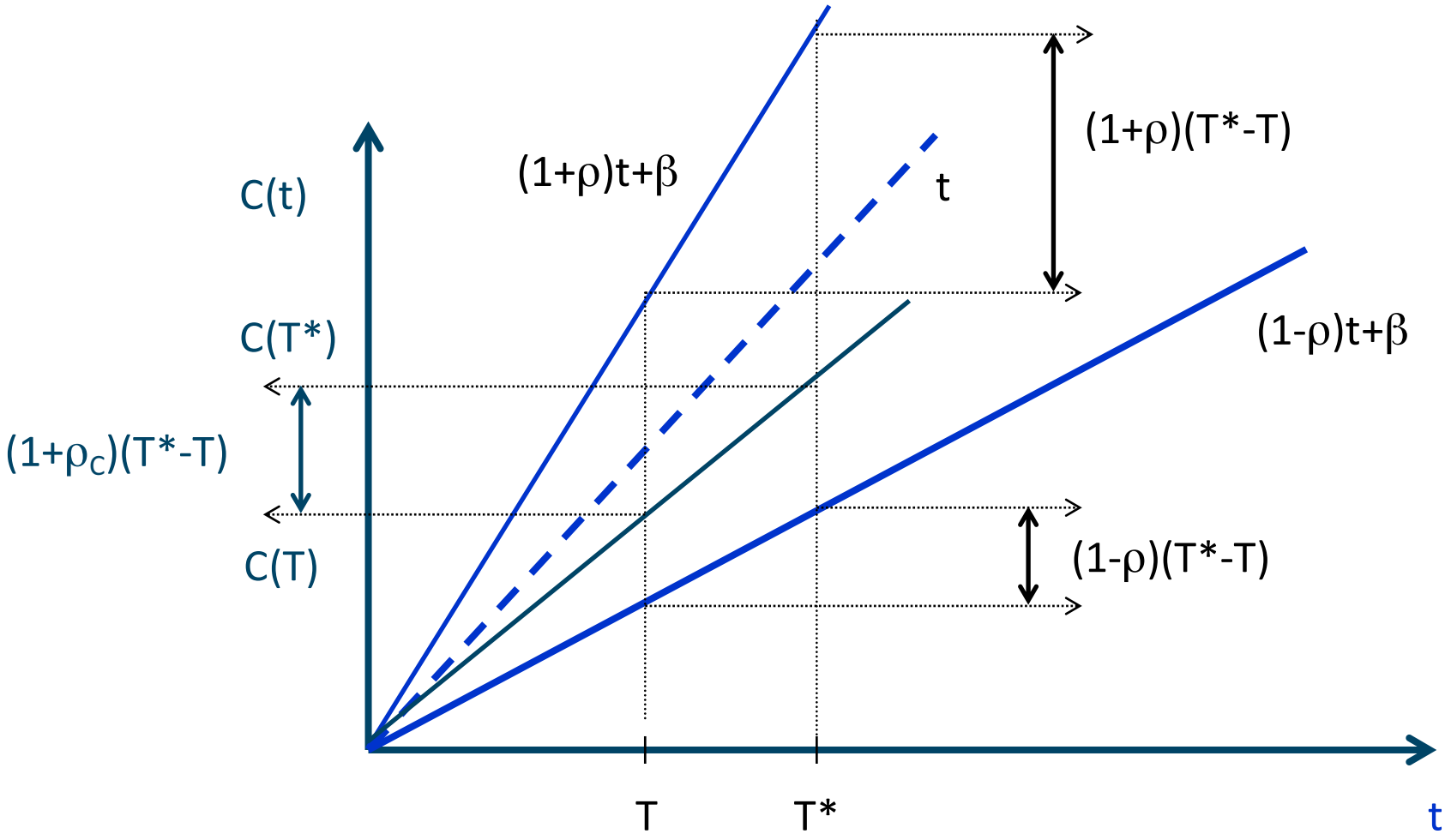
# Bounded drift rate

**max drift rate = ρ**
**actual drift rate = $\rho_c$**

$$C(t) = (1 + \rho_C)t + \beta$$

$$-\rho \le \rho_C \le \rho$$

**Error made when measuring intervals ?**



# Correct clock

**(i) Correctness condition : bounded drift rate**

$$(1-\rho)(T^*-T) \le C(T^*)-C(T) \le (1+\rho)(T^*-T)$$

➔ NO jumps in clock

**(ii) Monotonicity condition**

$$T < T^* \Rightarrow C(T) < C(T^*)$$

➔ do NOT go backward in time !
weaker than bounded drift rate condition

**(iii) Hybrid condition**

= monotonicity + forward jumps at sync. points

# The origin of time

**Historically : time used to organize everday's life**

- time related to sun apparent movement
- 24h = time elapsed between 2 solar culmination points
- timekeeping by astronomers (**Greenwich Mean Time**)



Ruth Belville

# Accuracy is all that matters…

## Taken over by physicists

Notion of time often redefined up till the nineties!

Adjustments deal with accuracy of a second

**1 second…**

**1940**: 1/86,400th of a mean solar day.

**1956**: 1/31,556,925.9747 of the tropical year for 1900 January 0 at 12 hours ephemeris time

**1967**: the duration of 9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium-133 atom

**1977**: … definition above but at sea level

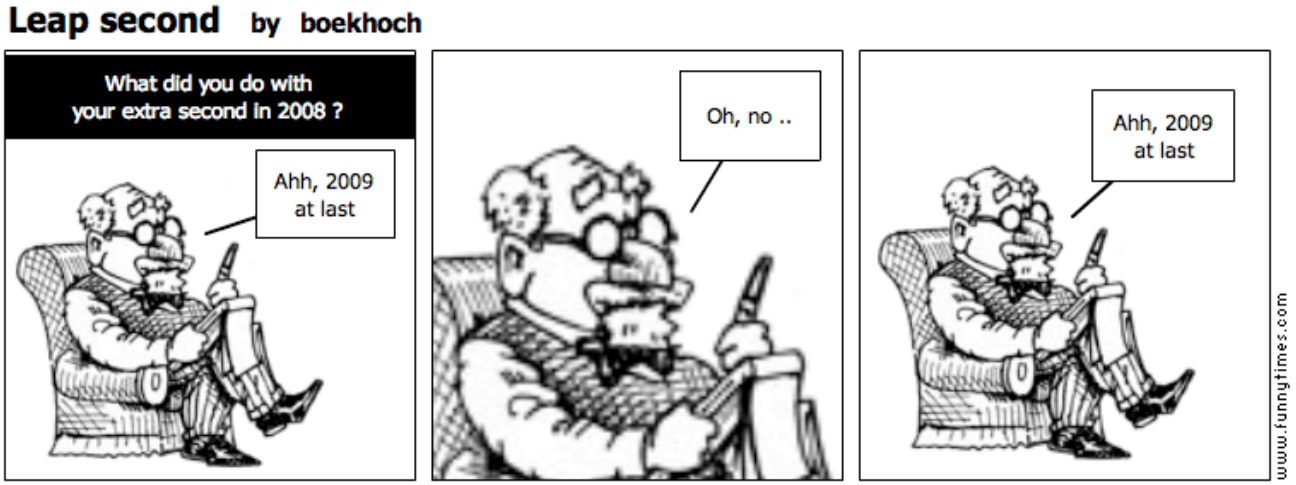
**1997**: …definition above but at a temperature of 0K

# Clock drift in standard time

**BUT: physicists' time gets out of pace with astronomers' time**

- irregularities in earth's orbit
- slow down of earth's rotation
- Need for small adjustments to keep sync. with "real" time
  - leap seconds introduced
  - when ? if skew(IAT,astronomical time) > 800 ms

**UTC** : Universal Coordinated Time




**35 seconds**

= Number of seconds that International Atomic Time (IAT) has diverged from Coordinated Universal Time (UTC) since IAT was established in 1972.

# Getting accurate time

1. **Attach GPS receiver to each computer**

   ± 1 msec of UTC

2. **Attach WWV radio receiver**

   ± 3 msec of UTC (depending on distance)

3. **Attach GOES receiver**

   ± 0.1 msec of UTC

**Not practical solution for every machine: cost, size, convenience, environment**

4. **Synchronize time from another machine**

   **→ Time server**

# 1 Physical clock synchronization

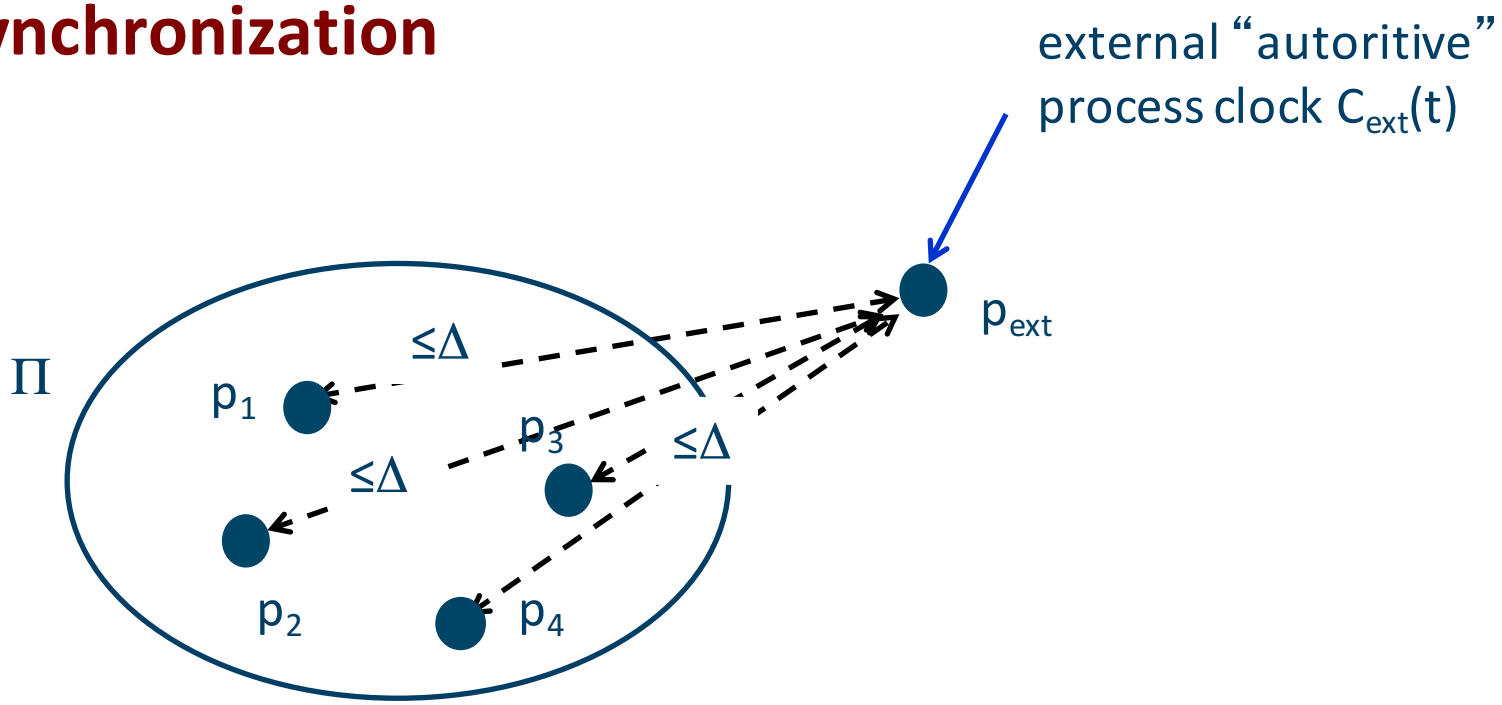
1. How do computers measure time?
2. Clock skew and clock drift
3. Time standards
4. **Clock synchronization**
   - A. External-Internal synchronization
   - B. Client-Server system
     - 1) Synchronous system
     - 2) Asynchronous system (Christian's algorithm)
   - C. Peering algorithms
     - A. Network Time Protocol
     - B. Berkeley algorithm

# External synchronization

Set of interacting processes $\Pi$

Each process p → clock $C_p(t)$

## External synchronization



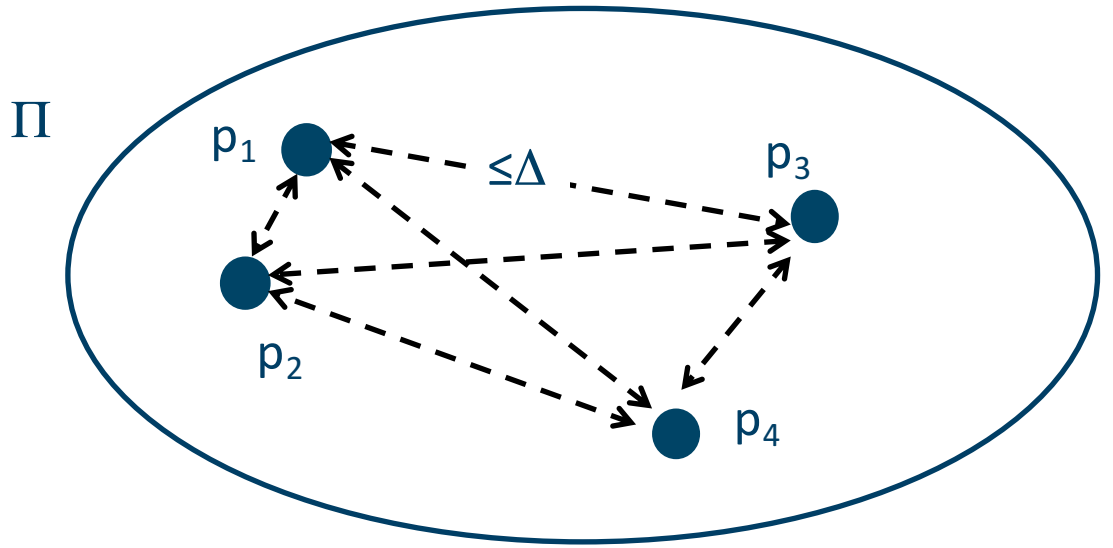

$\Pi$ is externally synchronized

$\leftrightarrow$ All $C_p(t)$ have bound skew $\Delta$ w.r.t. $C_{ext}(t)$

$$\left|C_p(t) - C_{ext}(t)\right| \leq \Delta, \forall p \in \Pi$$

# Internal synchronization

## Internal synchronization



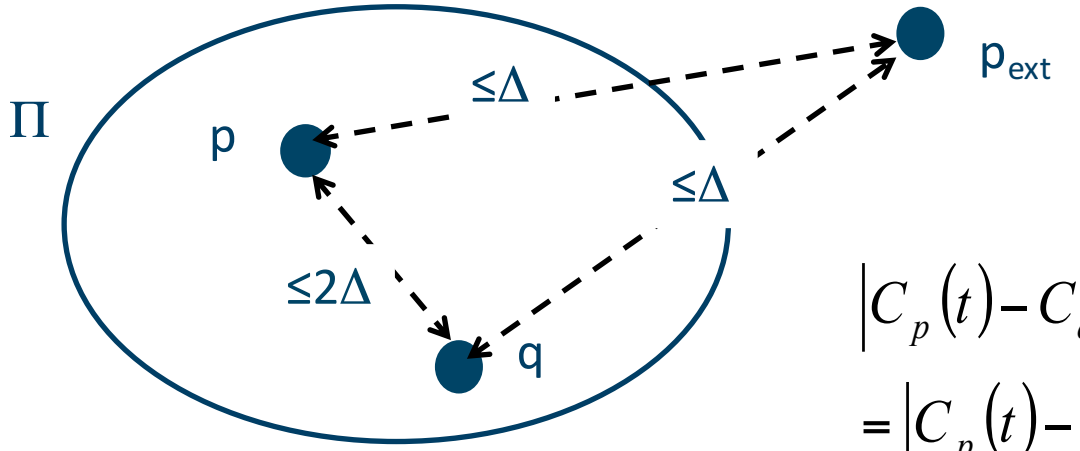

Π is internally synchronized

↔ Skew between any 2 clocks in Π is bounded

$$\left|C_p(t) - C_q(t)\right| \leq \Delta, \forall p, q \in \Pi$$

# External vs internal synchronization

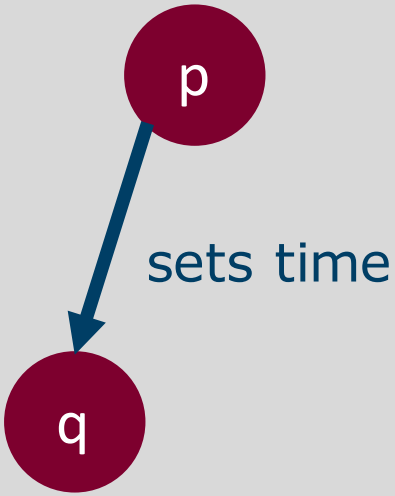
if $\Pi$ is externally synchronized with skew $\Delta$,
then it is internally synchronized with skew $2\Delta$



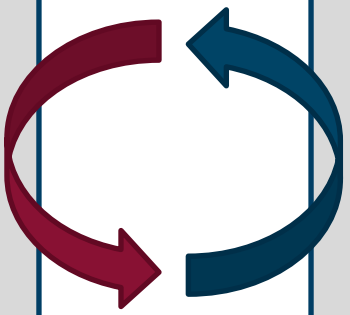

$$
\begin{aligned}
&\left|C_p(t) - C_q(t)\right| \\
&= \left|C_p(t) - C_{ext}(t) + C_{ext}(t) - C_q(t)\right| \\
&\le \left|C_p(t) - C_{ext}(t)\right| + \left|C_{ext}(t) - C_q(t)\right| \\
&\le 2\Delta
\end{aligned}
$$

# Clock synchronization

## Client-Server algorithms




- Asymmetric
- 2 processes p and q
- p has better time than q
- p is server, q is client



## Peering algorithms


agree on time

- Symmetric
- Mutual agreement between processes
- Exchange of messages

# Client-server algorithm

**Goal of synchronization**

minimize $|C_p - C_q|$

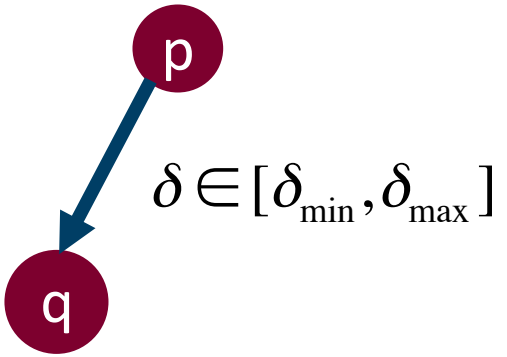
**Problem**

p sends its current time reading

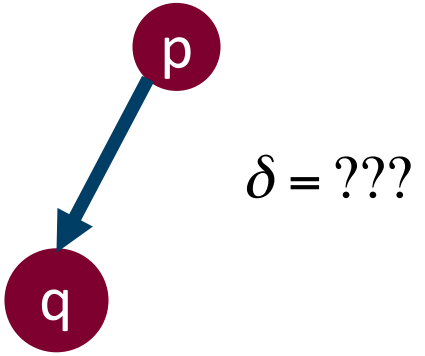
BUT: when arriving at q, unknown amount of time $\delta$ has elapsed

**2 Possibilities**

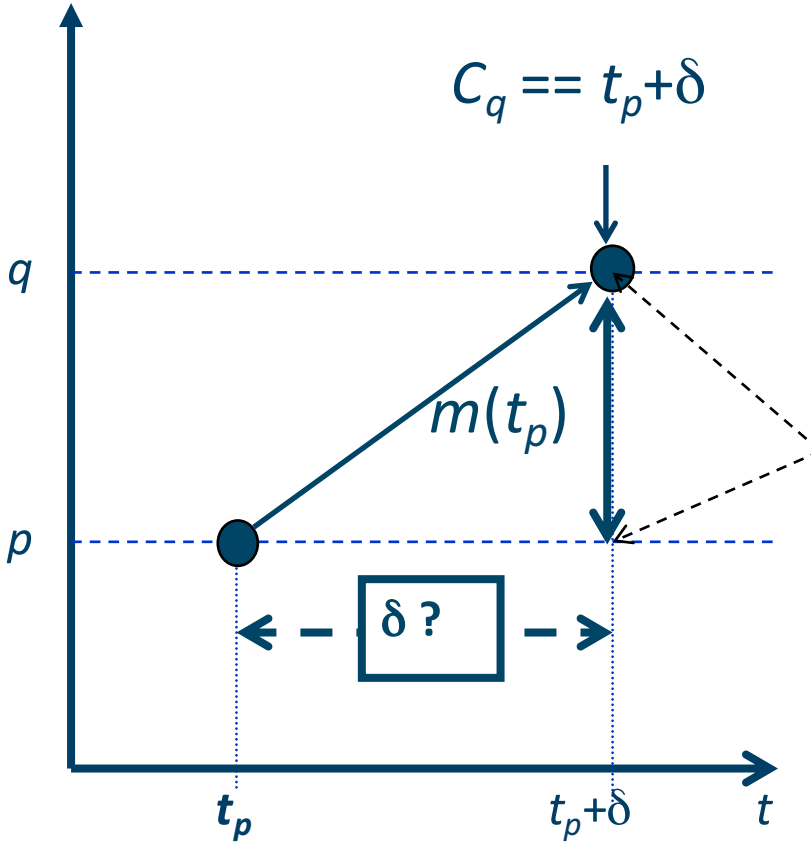
1. *Synchronous system*
   - Bounds of network delay known, i.e. $\delta_{max}$, $\delta_{min}$
   - Puts limit on uncertainty of p send time



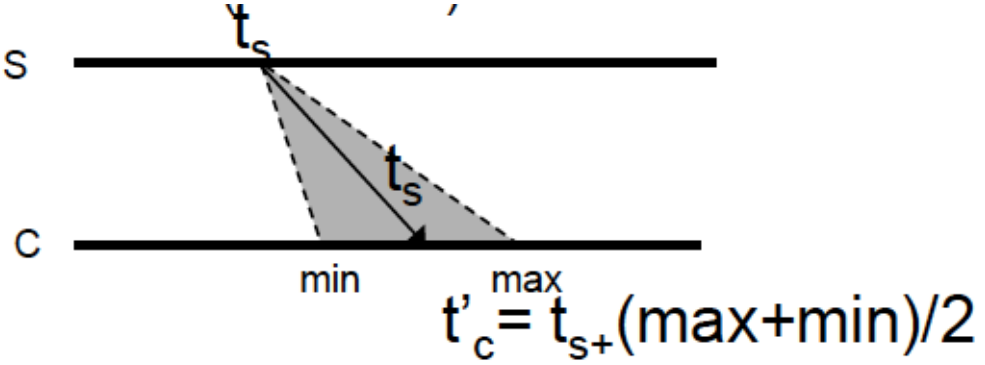

2. *Asynchronous system*
   - No info known on $\delta$, BUT, of course $\delta \leq$ RTT
   - Estimate $\delta \approx$ RTT/2



# Synchronous system




$C_p$ and $C_q$ should be synchronized.



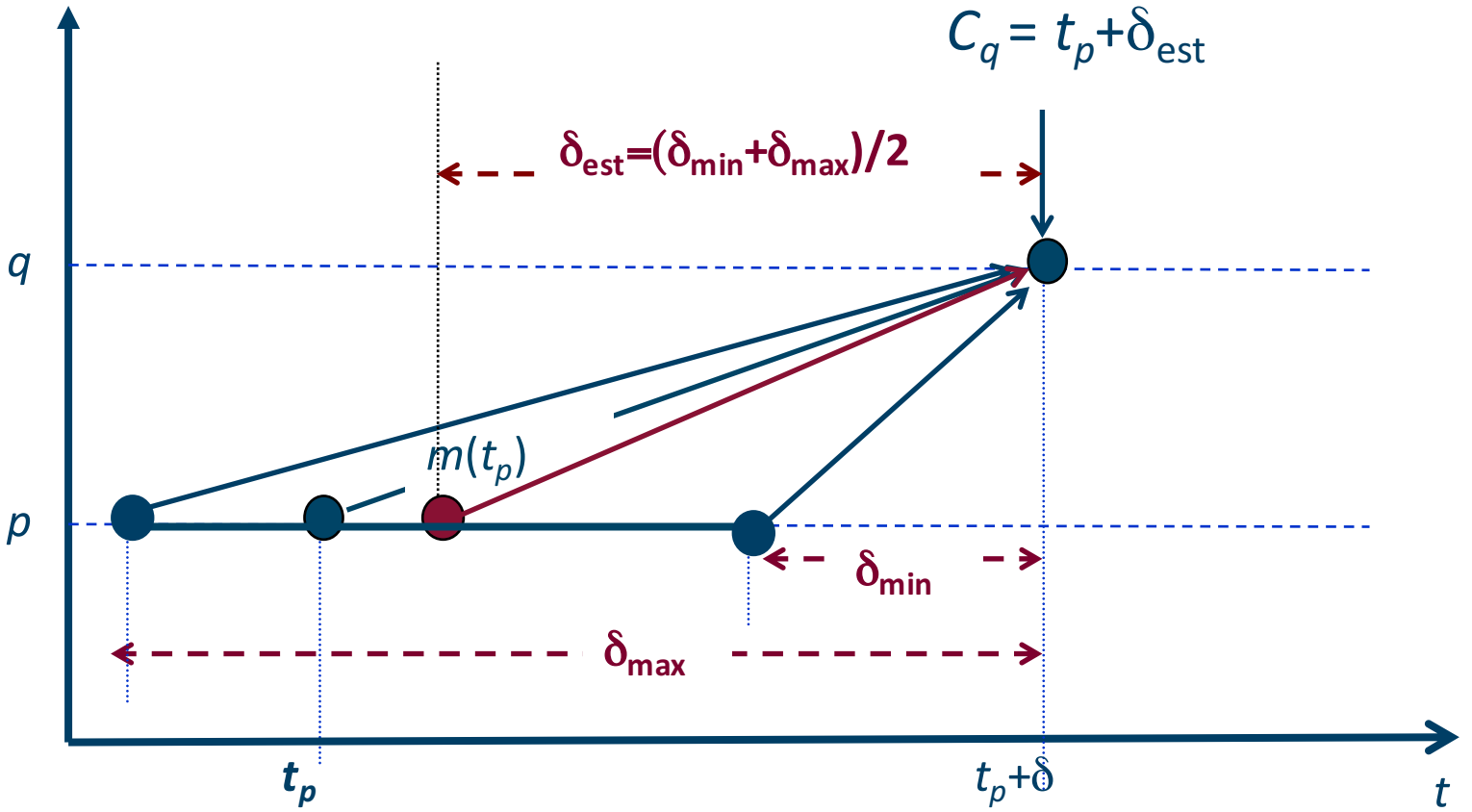

$\delta_{min} \le \delta \le \delta_{max}$

$\rightarrow t_p+\delta_{min} \le t_p+\delta \le t_p+\delta_{max}$

estimate : $t_p+\delta = t_p + (\delta_{min}+\delta_{max})/2$

$$C_q = t_p + \frac{\delta_{min} + \delta_{max}}{2}$$

# What is the error made?




skew($C_p$,$C_q$) $= |t_p+\delta_{est}-(t_p+\delta)|$

$= |\delta_{est} - \delta|$

worst case occurs at the edges, i.e. for

$\delta=\delta_{min}$

$\delta=\delta_{max}$

$$\max skew_{p,q} = \max|C_p - C_q| = \frac{\delta_{max} - \delta_{min}}{2}$$

# Asynchronous system: Cristian's algorithm

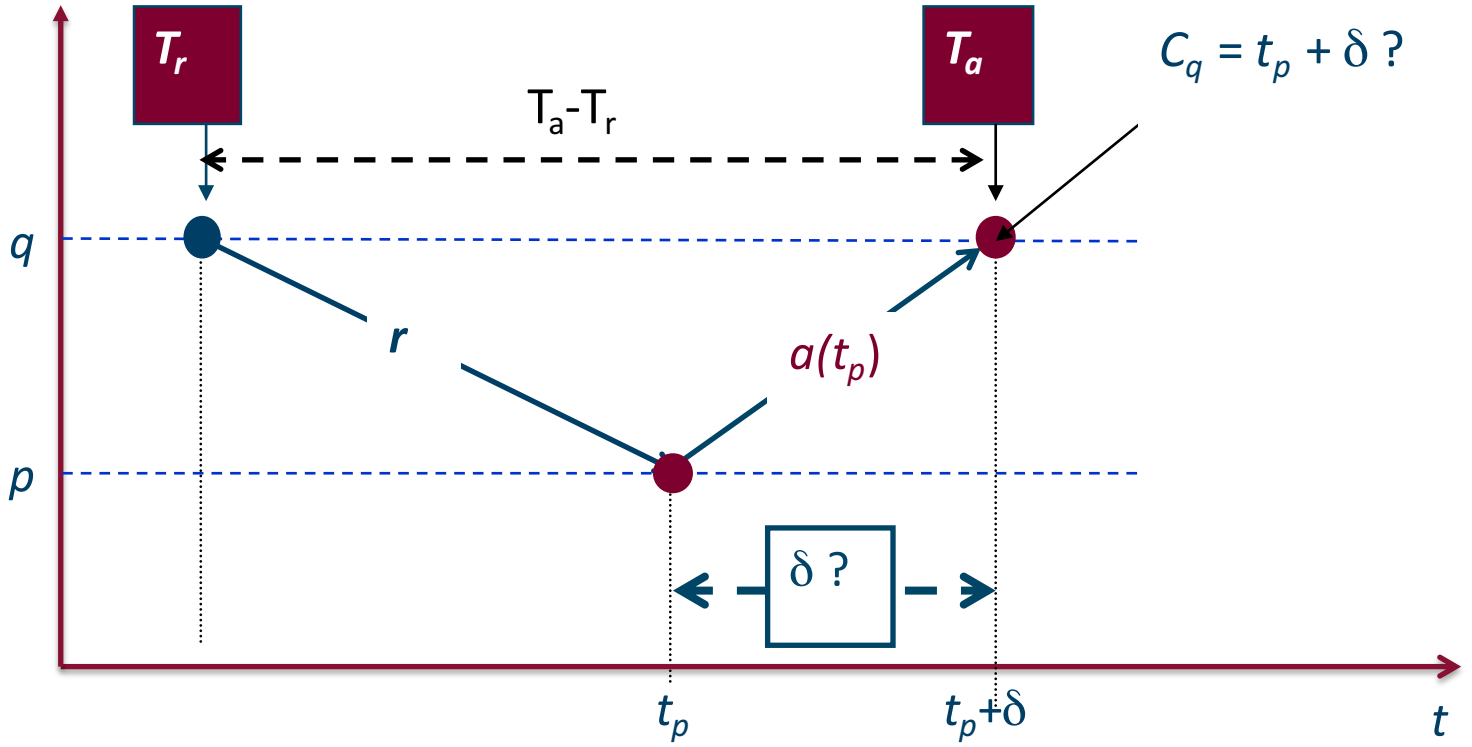
**No** upper bound for one way delay $\delta$

Suppose we have a lower bound $\delta_{min}$ (possibly 0)

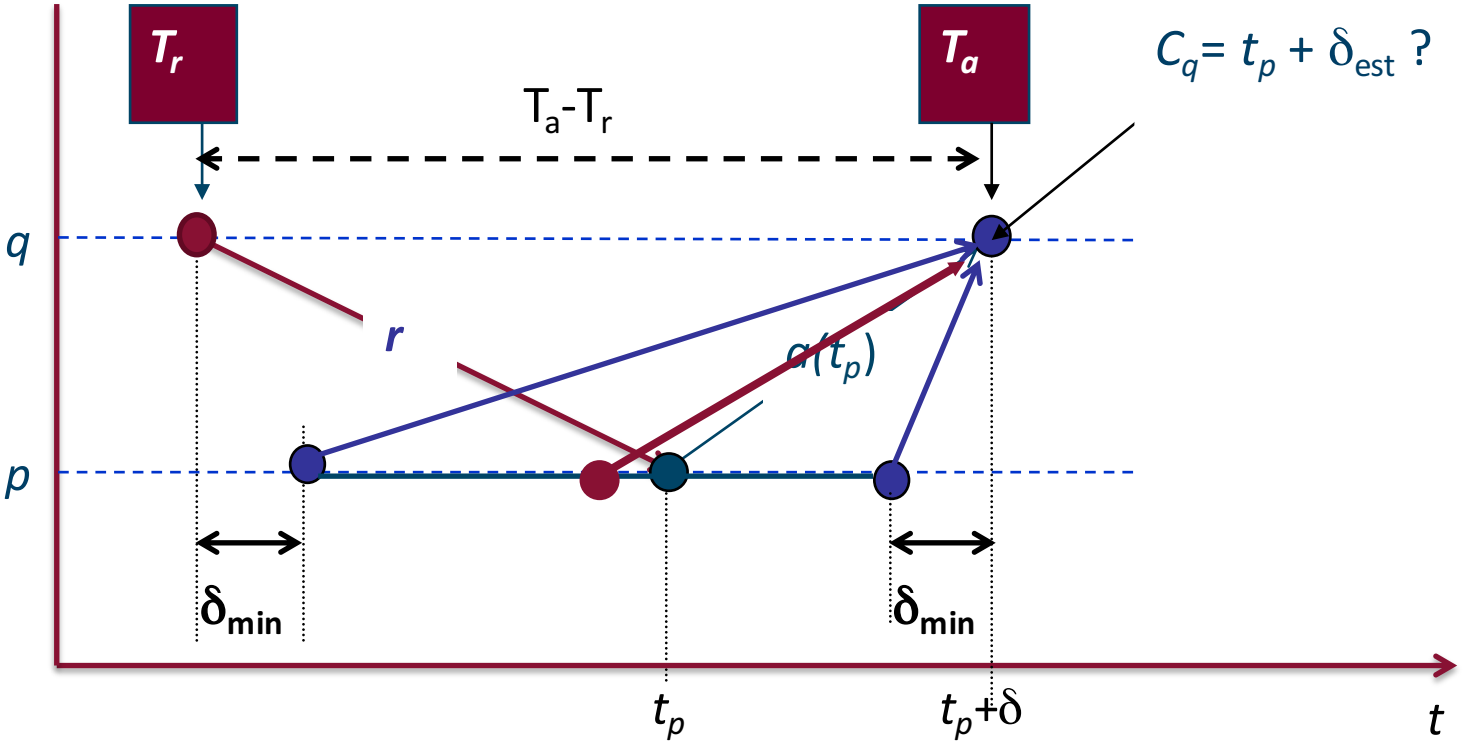
Two messages involved:

- Request (**r**)
- Reply (**a**) : contains timestamp by time server

(1951-1999)



# Asynchronous system: Cristian's algorithm




$\delta_{min} \le \delta \le T_a - T_r - \delta_{min}$

$\delta_{est} = (T_a - T_r)/2$

$$C_q = t_p + \frac{T_a - T_r}{2}$$

skew=$|C_q - C_p|$ = $|t_p + \delta_{est} - (t_p + \delta)|$
= $|\delta_{est} - \delta|$
max skew for extreme values of $\delta$

$$\max skew_{p,q} = \max|C_p - C_q| = \frac{T_a - T_r}{2} - \delta_{\min}$$

# Cristian's algorithm: example

Send request at 5:08:15.100 ($T^r$)

Receive response at 5:08:15.900 ($T_a$)

- Response contains 5:09:25.300 ($t_{pr}$)

Elapsed time is $T_a - T_r$

5:08:15.900 - 5:08:15.100 = 800 msec

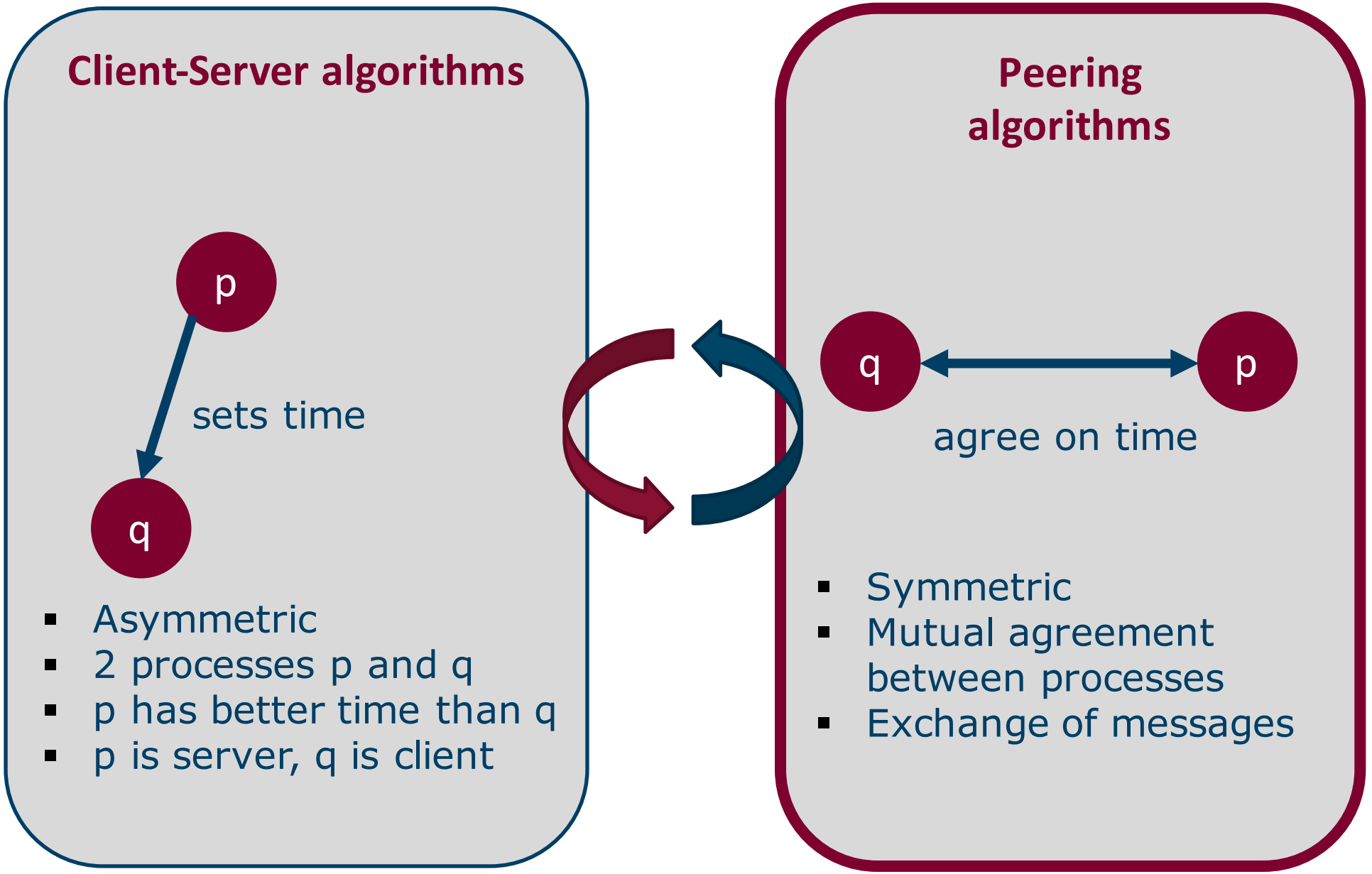
Best guess: timestamp was generated

400 msec ago

Set time to $t_p$ + *elapsed time*

5:09:25.300 + 400 = 5:09.25.700

# Clock synchronization



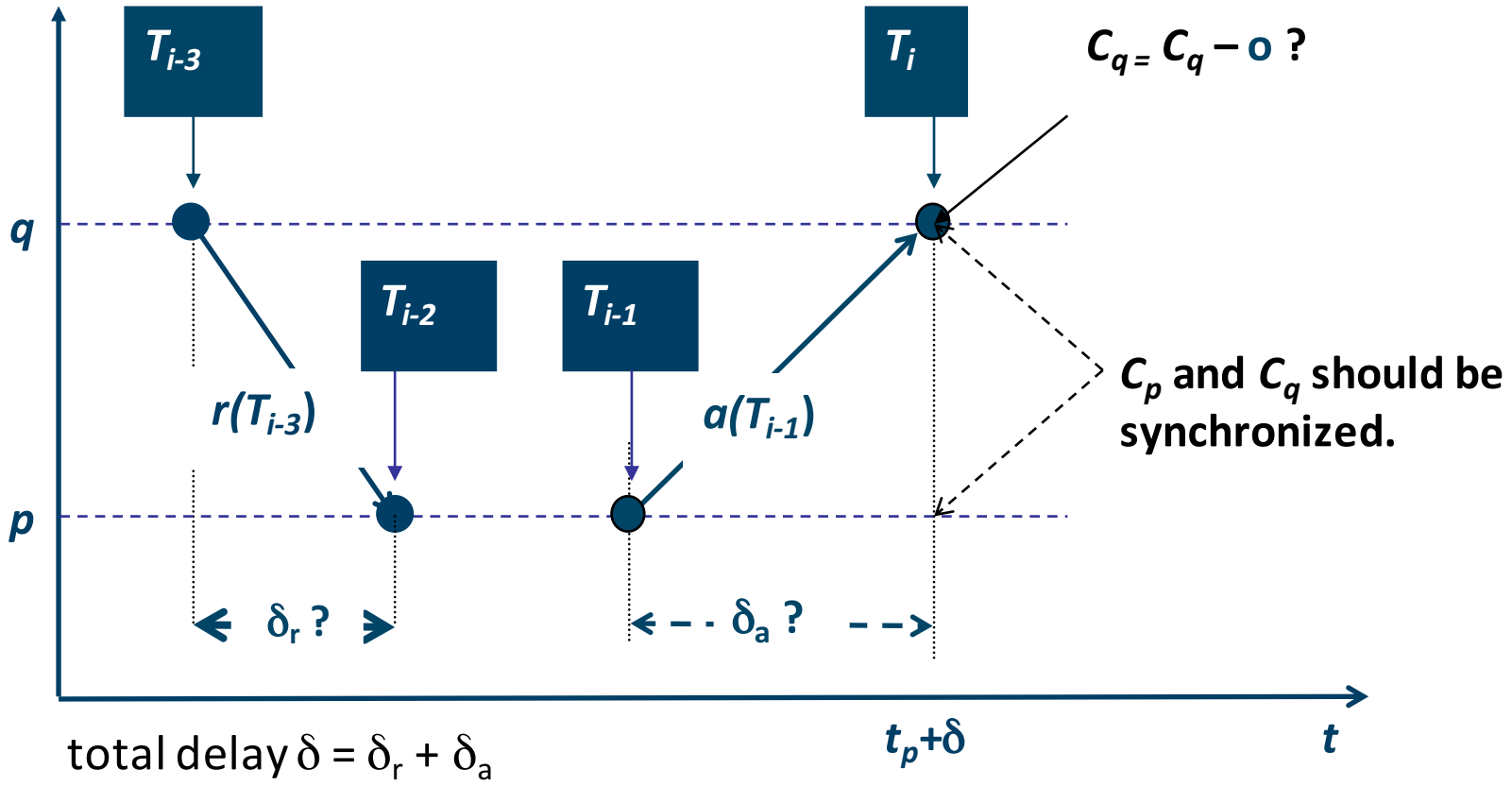
# Network Time Protocol (NTP)

Clock skew between p and q : $C_q = C_p + o$

Two messages exchanged between p and q

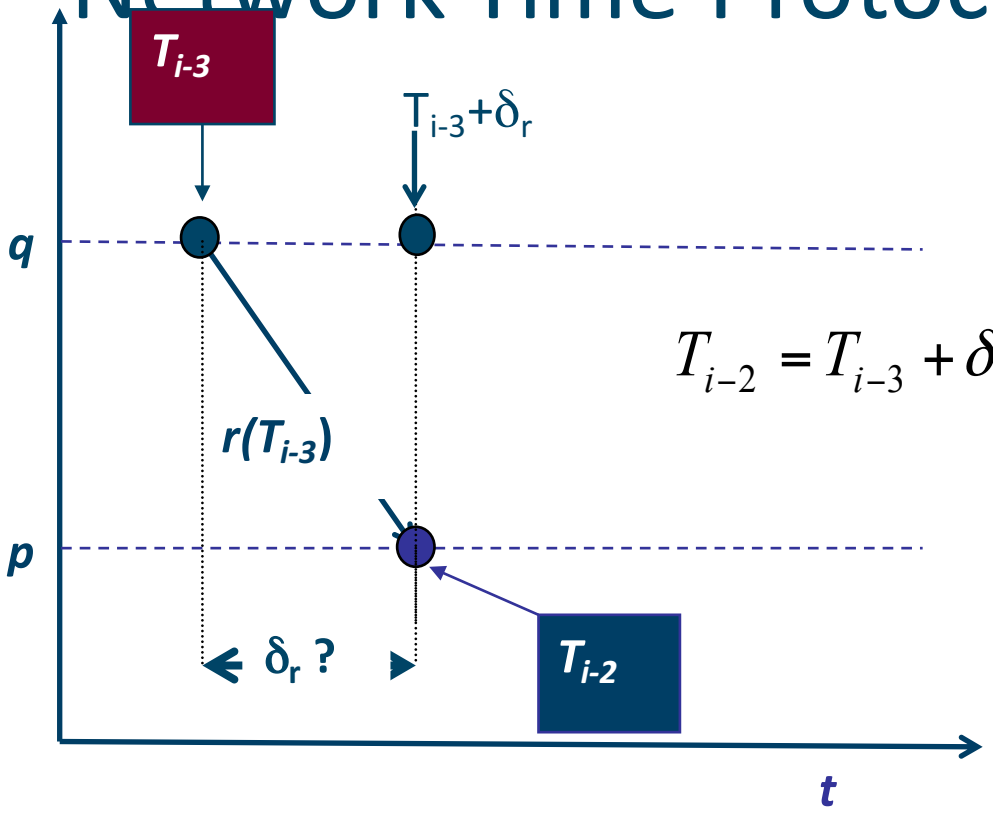
- request (**r**)
- reply (**a**)

Time stamp info embedded in **a** and **r**

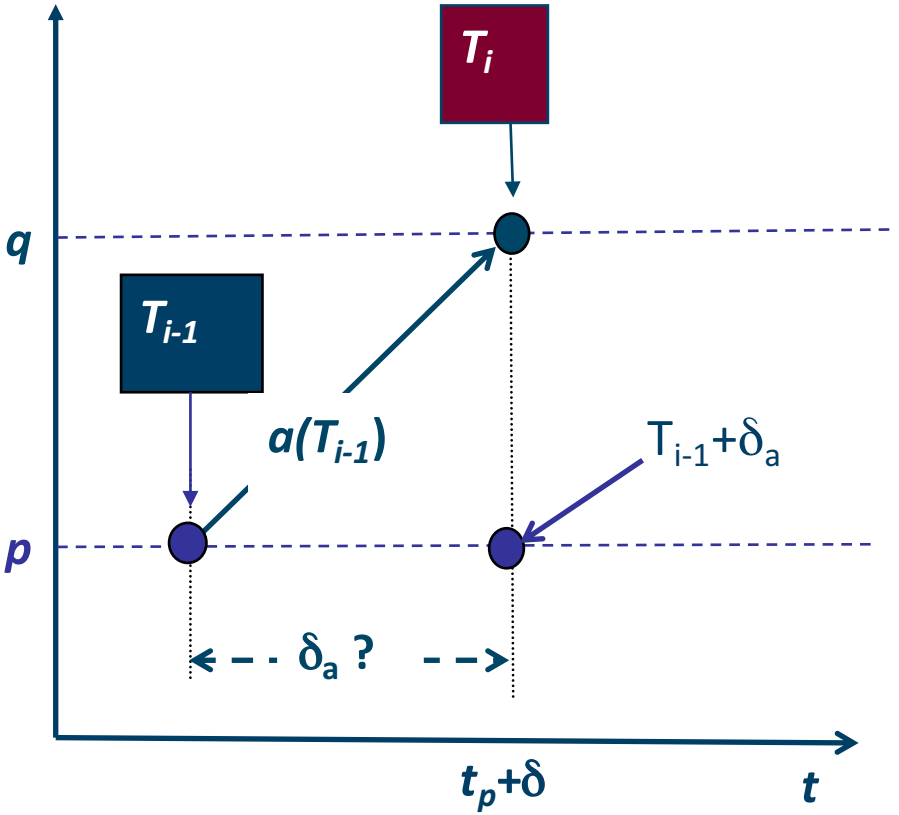


# Network Time Protocol (NTP)

$C_q = C_p + o$




$$T_{i-2} = T_{i-3} + \delta_r - o$$




$$T_i = T_{i-1} + \delta_a + o$$

# Network Time Protocol (NTP)

We now know the relationship between

- Ti-2 and Ti-3
- Ti-1 and Ti

Let's combine these two by subtracting them

$$T_{i-2} = T_{i-3} + \delta_r - o$$

$$- \quad T_i = T_{i-1} + \delta_a + o$$

$$T_{i-2} - T_i = T_{i-3} - T_{i-1} + \delta_r - \delta_a - 2o$$

or $$o = \frac{1}{2}\left(T_i - T_{i-2} + T_{i-3} - T_{i-1}\right) + \frac{1}{2}\left(\delta_r - \delta_a\right)$$

estimate o : suppose network is symmetric ($\delta_r = \delta_a$)

$$\tilde{o} = \frac{1}{2}\left(T_i - T_{i-2} + T_{i-3} - T_{i-1}\right)$$

# Network Time Protocol (NTP)

Now let's combine them by adding them

$$T_{i-2} = T_{i-3} + \delta_r - o$$

$$+ \quad T_i = T_{i-1} + \delta_a + o$$

$$T_{i-2} + T_i = T_{i-3} + T_{i-1} + \delta_r + \delta_a$$

or $$\boxed{\delta = \delta_r + \delta_a = T_{i-2} - T_{i-3} + T_i - T_{i-1}}$$

# Network Time Protocol (NTP)

$$o = \frac{1}{2}\left(T_i - T_{i-2} + T_{i-3} - T_{i-1}\right) + \frac{1}{2}\left(\delta_r - \delta_a\right)$$

$$\tilde{o} = \frac{1}{2}\left(T_i - T_{i-2} + T_{i-3} - T_{i-1}\right)$$

$$\delta = \delta_r + \delta_a = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

**synchronization error ?**

$$o = \tilde{o} + \frac{1}{2}\left(\delta_r - \delta_a\right) \le \tilde{o} + \frac{1}{2}\left(\delta_r - \delta_a\right) + \delta_a = \tilde{o} + \frac{\delta}{2}$$

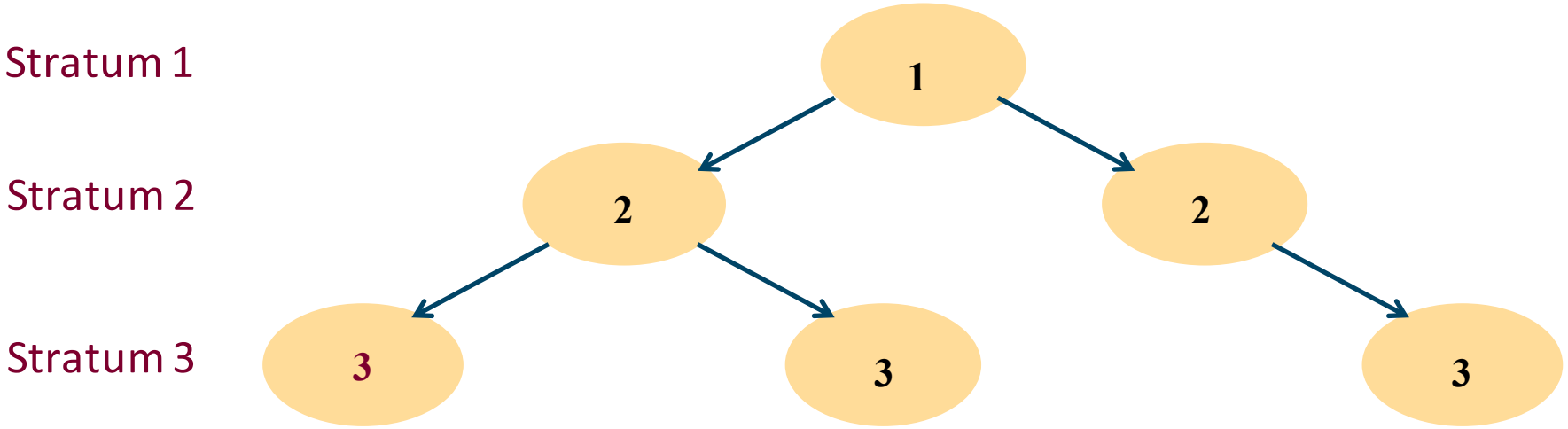
$$o = \tilde{o} + \frac{1}{2}\left(\delta_r - \delta_a\right) \ge \tilde{o} + \frac{1}{2}\left(\delta_r - \delta_a\right) - \delta_r = \tilde{o} - \frac{\delta}{2}$$

**maximum error = δ/2**

if <o,δ> pairs are stored, o-values with lowest δ's are most accurate

# Peering algorithms: NTP hierarchy

- Primary servers are connected to UTC sources
- Stratum n servers synchronized by stratum (n-1) servers (unless unreachable ...)
- Synchronization subnet
  - = lowest level servers in users' computers LAN




- Reliability through redundant paths
- Scalable
- Authenticates time sources

# Peering algorithms: NTP modes

**Multicast**

- used in high speed (low delay) LAN environment
- server multicasts time to clients
- clients assume some average delay

**Procedure call**

- cf. Christian's algorithm
- better accuracy obtained

**Symmetric**

- Peering servers execute P2P algorithm
  - collect 8 <o,δ> pairs
  - optimal one based on minimal δ
- Used to achieve high accuracy

NTP accuracy : 1 - 50 ms

# Peering algorithms: Berkeley algorithm

**Peering processes elect time server**
**Server (master) actively polls it's clients (slaves)**

1. Poll for slave time $\mathbf{t_{slave}}$,
   - using Christian's algorithm, measure $\mathbf{RTT_{slave}}$
2. Compute average time value, but
   - neglect slaves with $\mathbf{RTT_{slave} > RTT_{max}}$
     - Too large RTTs cause inaccurate values
   - if $\mathbf{|t_{avg} - t_{slave}| > \varepsilon}$ -> remove $\mathbf{t_{slave}}$ from set
     - Would influence the average value too much
   - send adjustment ($t_{slave} - t_{avg}$) to slave
     - removes uncertainty of sending new time message)
3. Elect new master if old one fails

# What is time?

1. Physical clock synchronization
2. Logical clocks

1. Events and temporal ordering
2. Lamport clock
3. Vector clock

# Event ordering

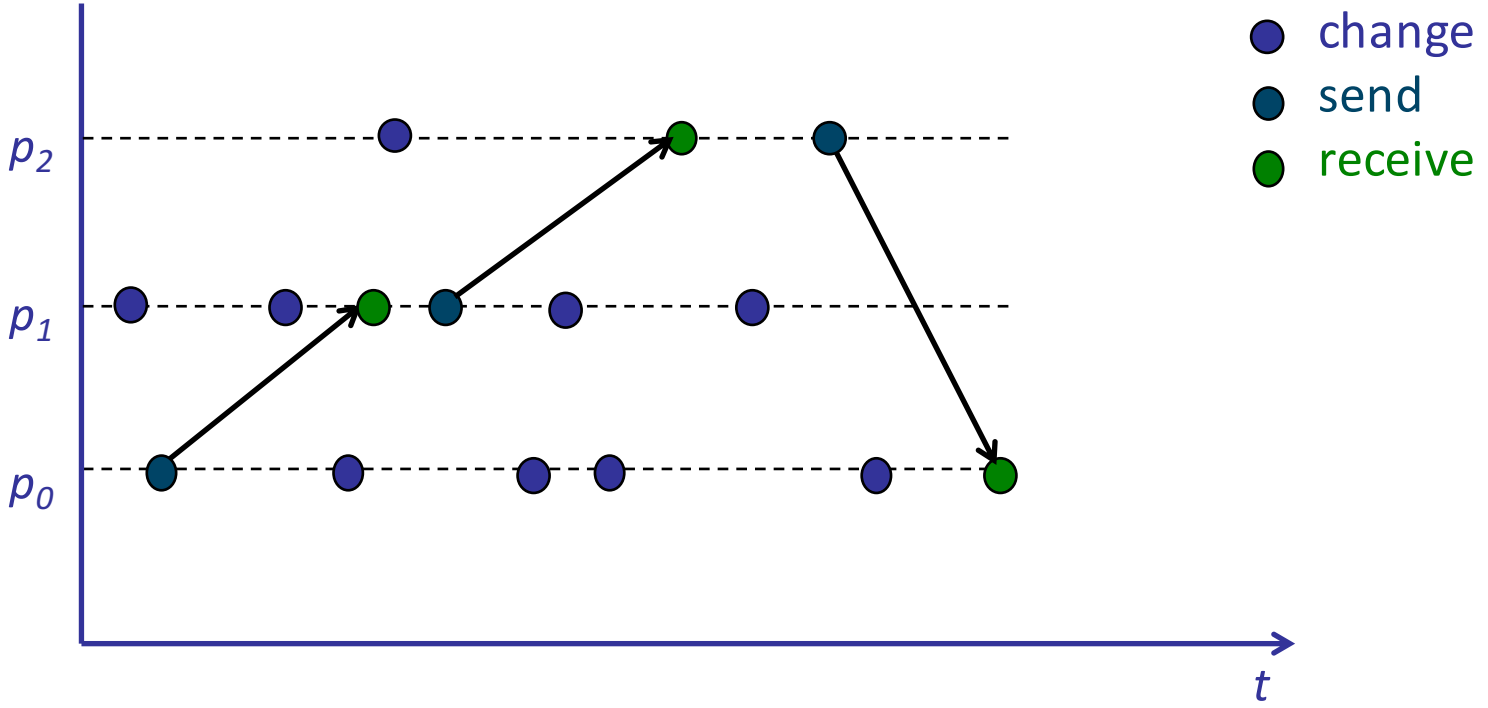
**Logical clock**

Goal:given two events p and q, happening in different processes which one happened first ?

**Distributed system $\Pi=\{p_1, \ldots, p_N\}$**

Event is

- **change** state in one of $\Pi$'s processes
- **send** message to process
- **receive** message from process



# Event ordering within one process

**… is easy !**

local clock can time stamp events

Define "$\rightarrow_i$" as process local happened-before relation

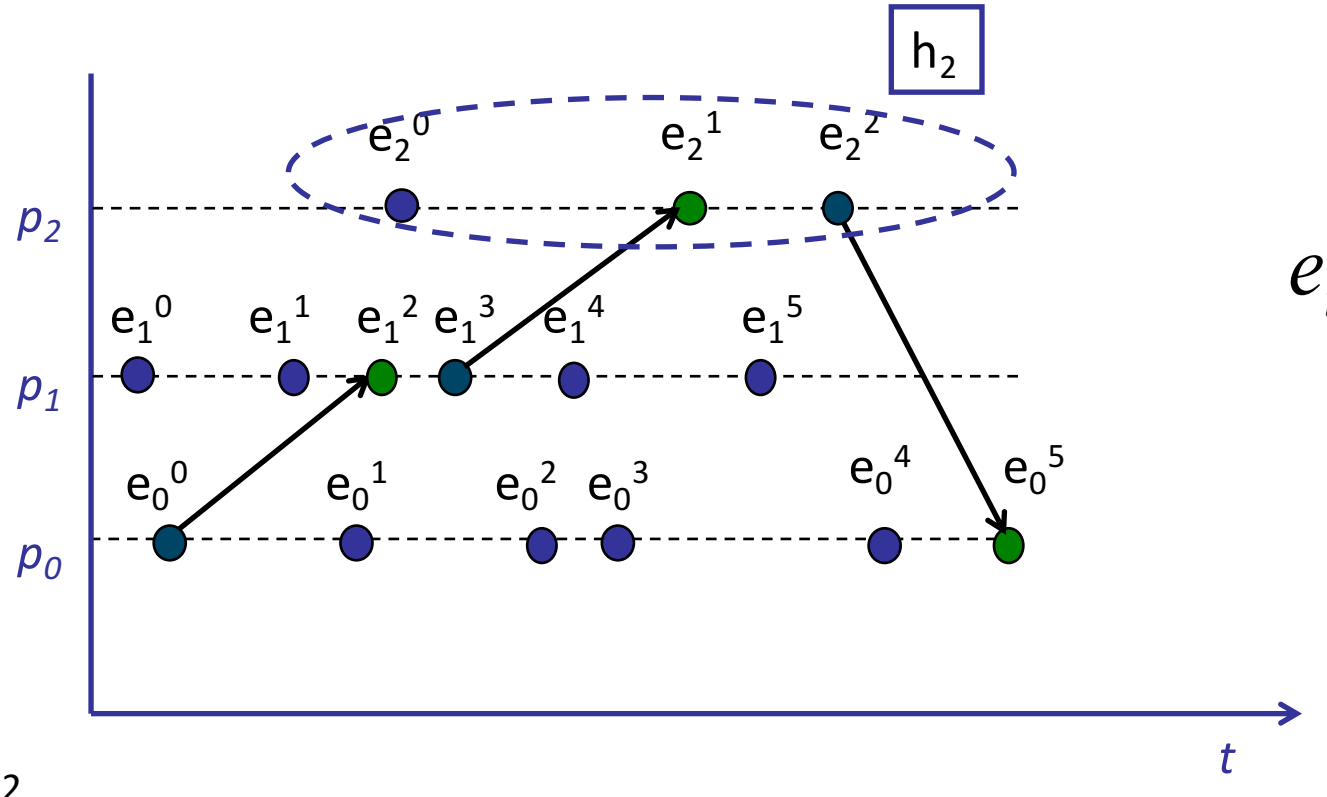
*for events happening in $p_i$*

Define $h_i$ event history of process $p_i$

$$h_i = \left\langle e_i^0, e_i^1, e_i^2, \ldots \right\rangle$$

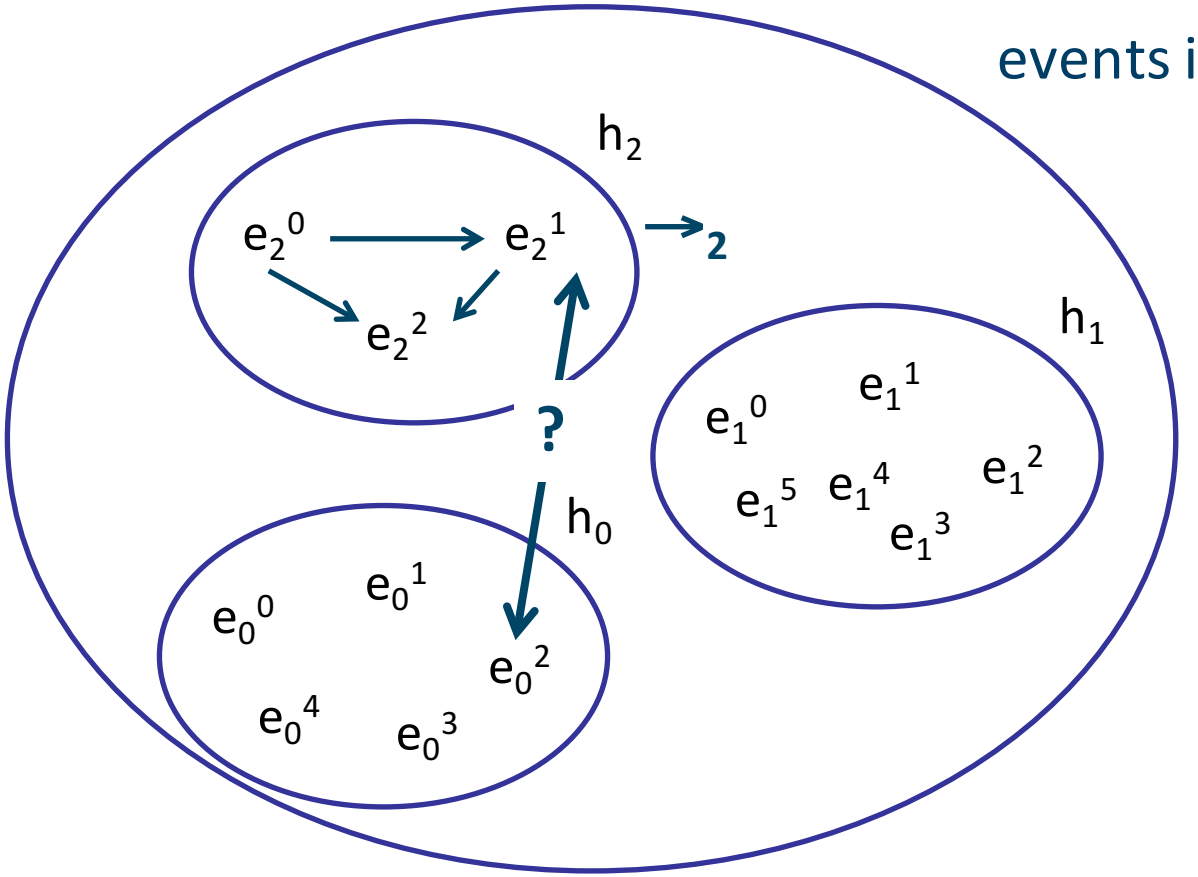
**process local logical clock**

$$e_i^k \rightarrow_i e_i^l \Leftrightarrow k < l$$



# Towards global event ordering

## events happened before real time **T**




**$\rightarrow_i$ only ordering in $h_i$**

**global ordering "$\rightarrow$" ?**

# Towards global event ordering

**How should → look like?**

Common sense for →

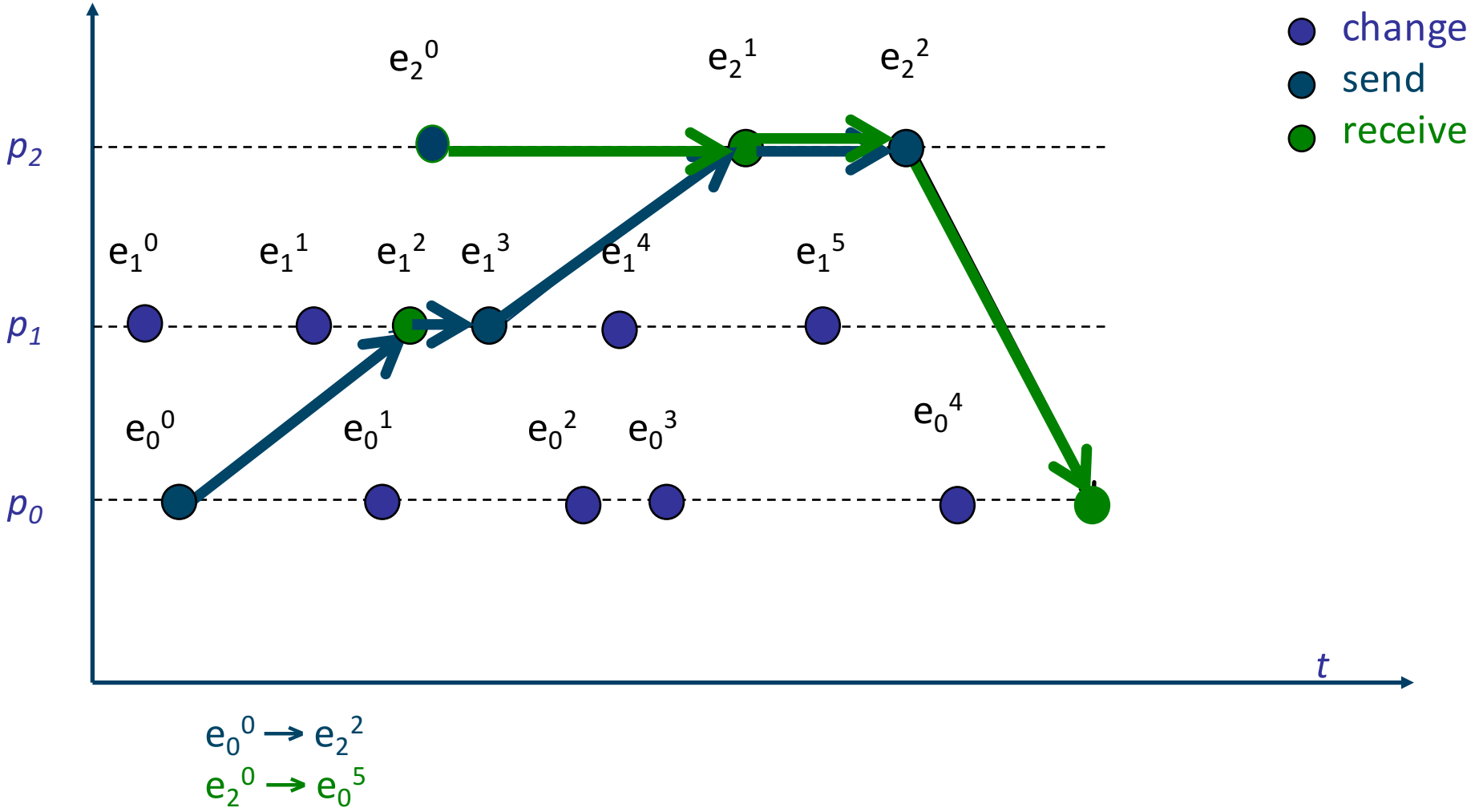
1. if both events belong to same process
   global ordering coincides with local ordering
   $a \rightarrow_i b \Rightarrow a \rightarrow b$

2. sending a message (s) happens before receiving (r)
   $s \rightarrow r$

3. → is transitive
   $(a \rightarrow b) \text{ AND } (b \rightarrow c) \Rightarrow (a \rightarrow c)$

**"p → q" if connected through a chain of events**

# Towards global event ordering



# Towards global event ordering

**Concurrent events**

if no chain of events links e and e'

e || e' ⇔ NOT(e → e') AND NOT(e' → e)

"e and e' happen concurrently"

**Causality ?**

e → e' ≠> e causes e'

e → e' => e' DOES NOT cause e

"→" : *potential* causal ordering

How to easily determine if a chain of events connects e and e' ?

**we need a clock …**

# A scalar clock: Lamport clock

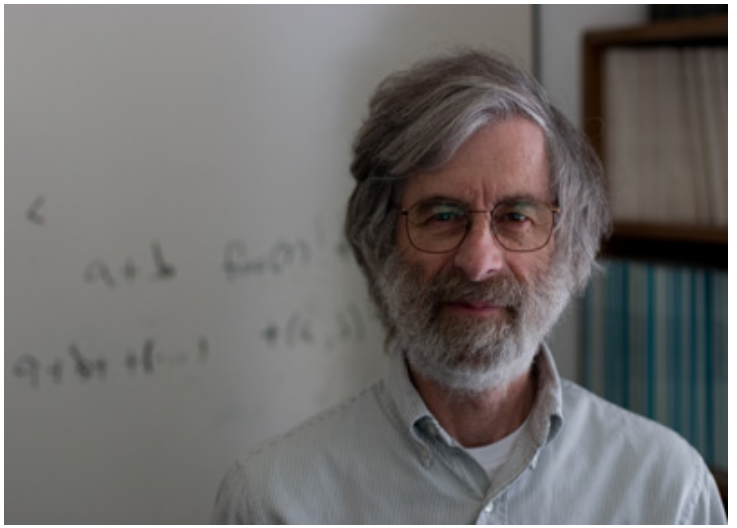
Each process $p_i$ maintains a scalar $L_i$, initial value 0

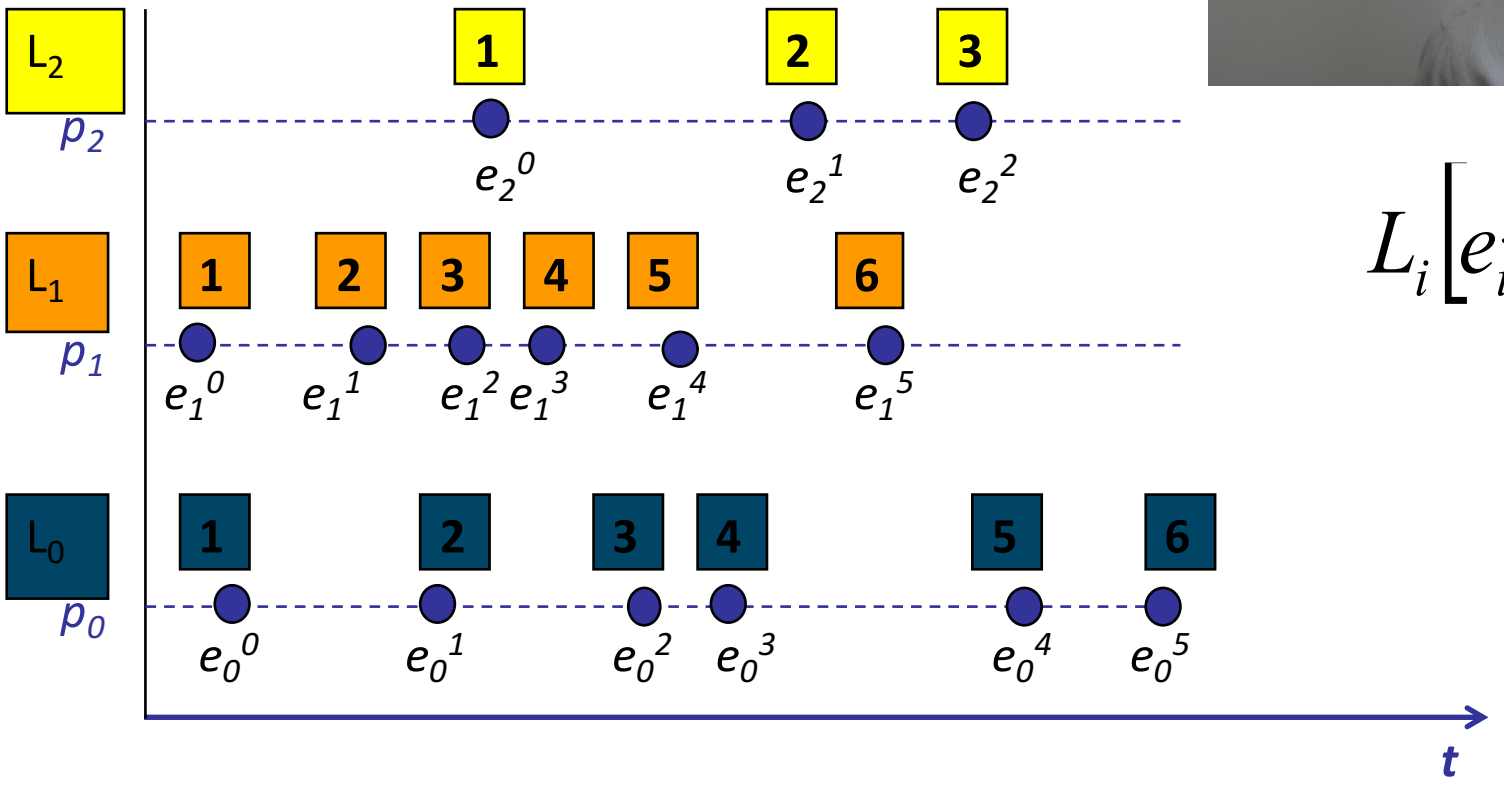
$L_i$ increases monotonically

$L_i$ updated just before event is timestamped

if **NO** communication between processes

$L_i$ is incremented at each event occurrence

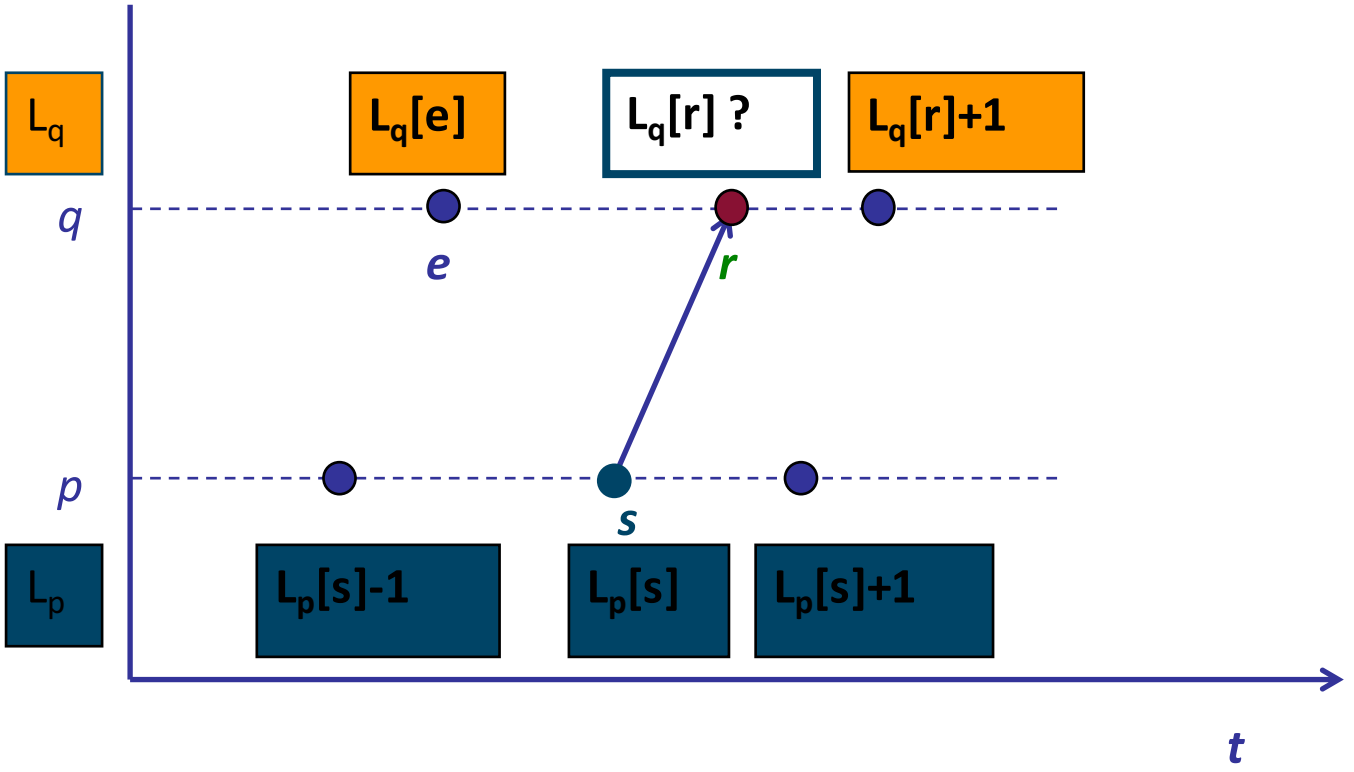





$$L_i\left[e_i^j\right] = j + 1$$

# Communicating processes

## In case of communication




e → r => $L_q[r] > L_q[e]$

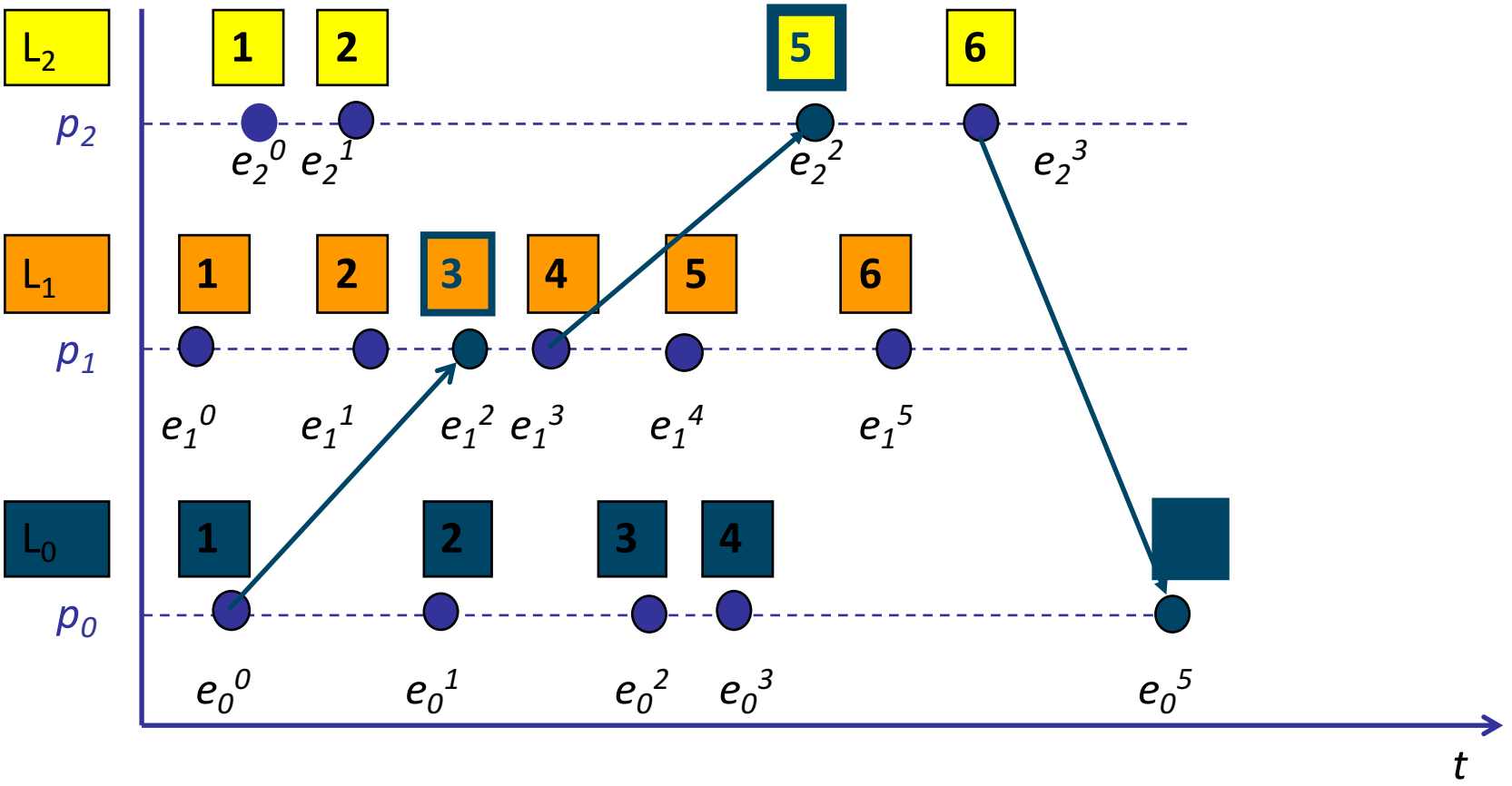
s → r => $L_q[r] > L_p[s]$

$$L_q[r] = \max\left(L_p[s], L_q[e]\right) + 1$$

# Lamport clock algorithm

1. Initialize all $L_p$ to **0**
2. Increment $L_p$ just before each event is handled in process ***p***
3. When message is sent from process ***p***,
   - send event is time stamped using 2.,
   - time stamp $L_p[s]$ is sent along with the message.
4. When a message is received at process ***q***:
   - new local clock is computed: $L_q \leftarrow \max(L_p[s], L_q)+1$
   - receive event is time stamped using this clock value

# Example


L2
1
2
5
6
p2
e2^0
e2^1
e2^2
e2^3
L1
1
2
3
4
5
6
p1
e1^0
e1^1
e1^2
e1^3
e1^4
e1^5
L0
1
2
3
4
p0
e0^0
e0^1
e0^2
e0^3
e0^5
t

# Interpretation

**No guarantee** for potential causal ordering !

But $$\neg\left[L[e] < L[e'] \Rightarrow (e \rightarrow e')\right]$$

$$(e \rightarrow e') \Rightarrow (L[e] < L[e'])$$

**No total ordering** on simple Lamport clock
(clock values can be equal for different events)

Relation R is **totally ordered** iff

i) R is antisymmetric: aRb AND bRa => a=b

ii) R is transitive : aRb AND bRc => aRc

iii) R is total : aRb OR bRa

Possible extension :

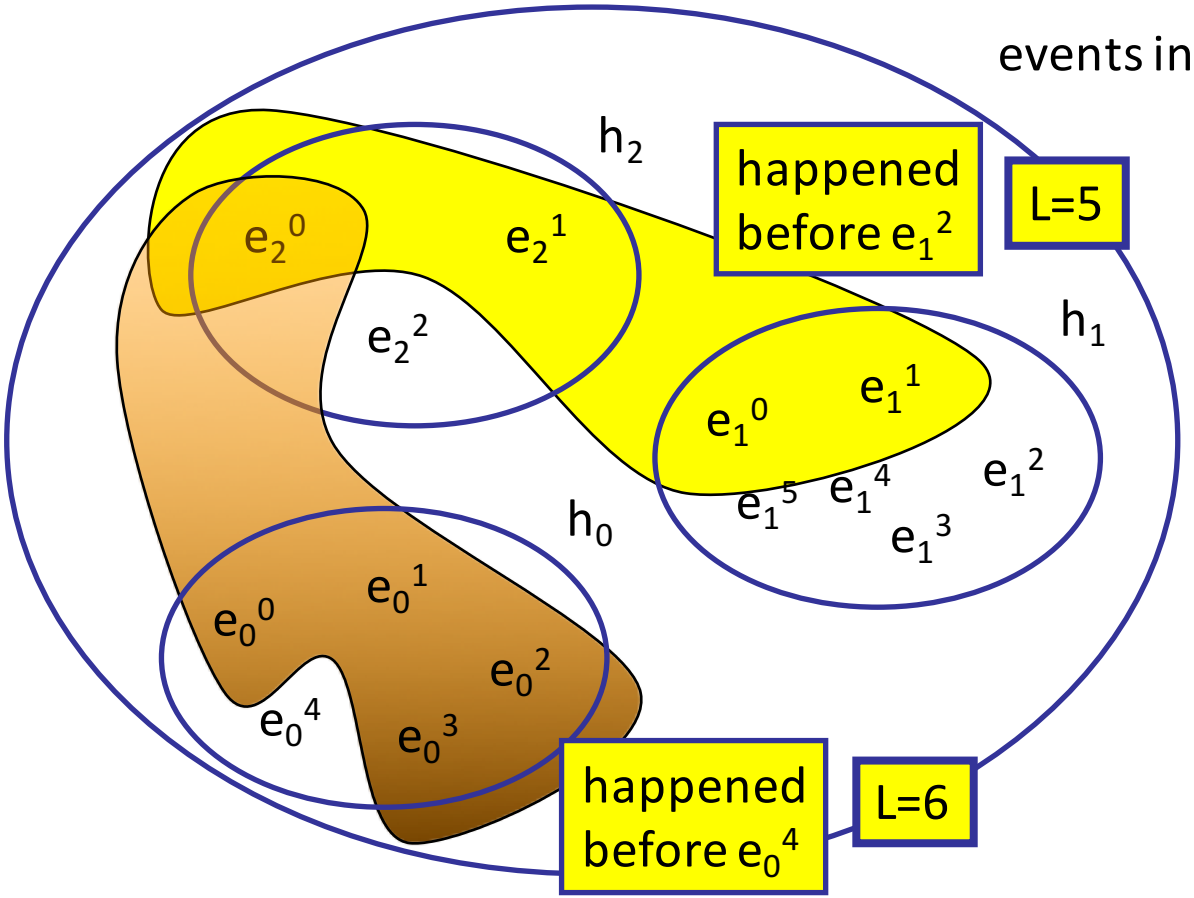
allocate (numeric) ID to each process

define ordering on <L,ID>

$$\langle L_p, ID_p \rangle < \langle L_q, ID_q \rangle \Leftrightarrow \begin{cases} L_p < L_q \\ (L_p = L_q) \wedge (ID_p < ID_q) \end{cases}$$

# Interpretation

Lamport clock just keeps track of the maximum number of events seen at a process from any process in $\Pi$




The events seen before $e_0^4$ are not necessarily a superset of those seen by $e_1^2$ !!!

$e_1^2 \mid\mid e_0^4$

# Vector clocks

**Goal**

provide guarantee for potential causal ordering based on clock value

**How?**

Keep track of the number of events seen from each process *individually*

→ Each process has a vector $\mathbf{V_p}$ of **N** elements (if N processes)

$\mathbf{V_p}$ defines how many events p has seen for each other process i

**Interpretation**

If an event (e') has a clock value, indicating that is has seen more events from every process than another event (e)

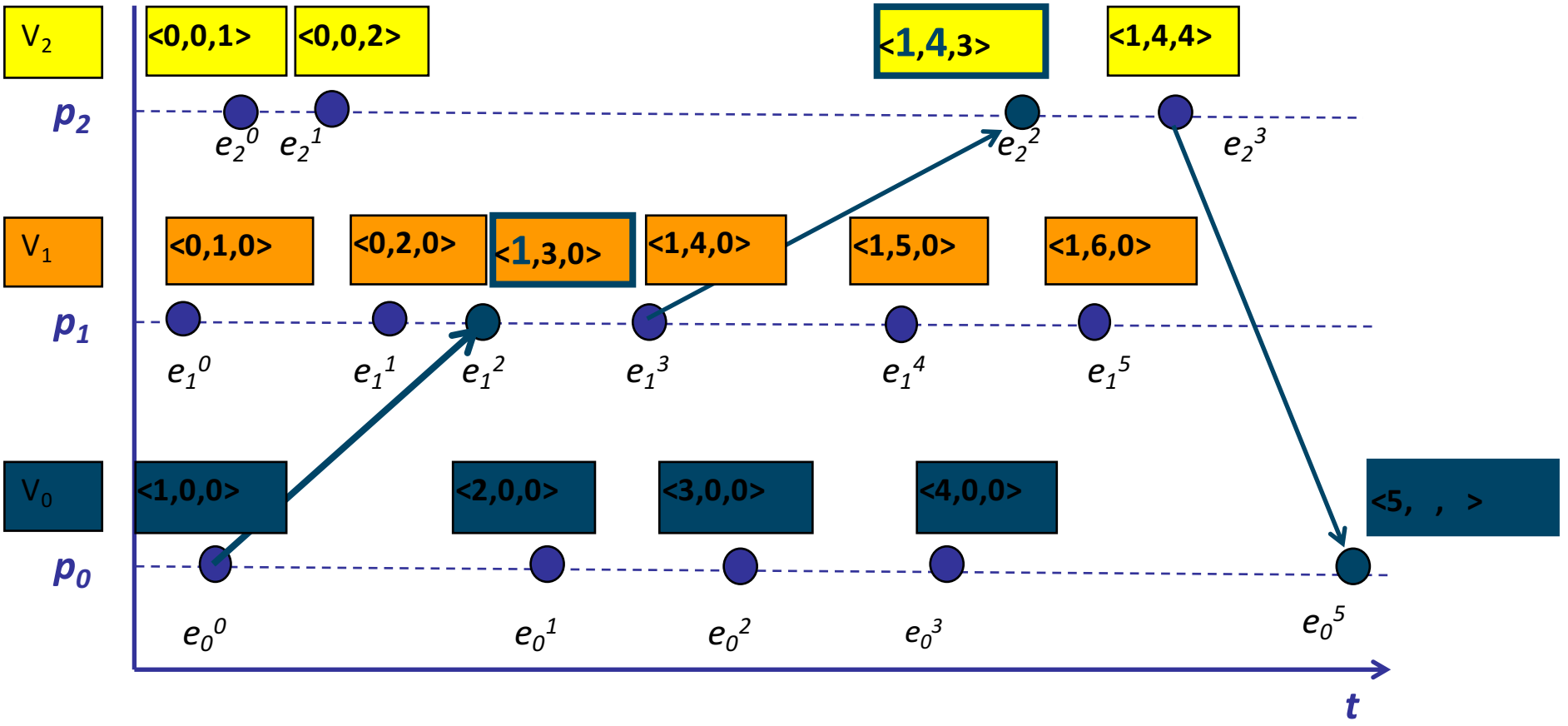
THEN we are sure that e → e'

$$(e \rightarrow e') \Leftrightarrow (V[e] < V[e'])$$

# Vector clock algorithm

1. Intialize $\mathbf{V_p=0}$
2. Increment $V_p[p]$ just before event is time stamped in process p:
   $V_p[p] \leftarrow V_p[p]+1$
3. When process p sends message, p sends complete vector
4. When process q receives a message
   $V_q[i] \leftarrow \max(V_q[i],V_p[i])$
   Increment $V_q$ according to 2.
   Timestamp the receive event with $V_q$

# Example


V2
<0,0,1>
<0,0,2>
<1,4,3>
<1,4,4>
p2
e2^0
e2^1
e2^2
e2^3
V1
<0,1,0>
<0,2,0>
<1,3,0>
<1,4,0>
<1,5,0>
<1,6,0>
p1
e1^0
e1^1
e1^2
e1^3
e1^4
e1^5
V0
<1,0,0>
<2,0,0>
<3,0,0>
<4,0,0>
<5, , >
p0
e0^0
e0^1
e0^2
e0^3
e0^5
t

# Remarks

**< ???**

$$V[e] = V[e'] \Leftrightarrow V_i[e] = V_i[e'], \forall i$$
$$V[e] \leq V[e'] \Leftrightarrow V_i[e] \leq V_i[e'], \forall i$$
$$V[e] < V[e'] \Leftrightarrow (V[e] \leq V[e']) \wedge (V[e] \neq V[e'])$$

**Drawbacks**

- more data exchanged
- number of processes needs to be known