Distributed Systems: Time

"Time has been invented in the universe so that everything would not happen at once."

“There is no change without the concept of time, and there is no movement without time.”

Why is time important for us?

- In distributed systems, we require…
  - **Coordination** between nodes: must **agree** on certain things
  - **High degree of parallelism**: nodes should work independently to make progress
- Time gives us…
  - **Point of reference** every machine knows how to keep track of...
  - Without need for explicit communication!
- However…
  - Time-keeping is not perfect ☹
  - ***How do we efficiently synchronize clocks?***

Why is time important for us?

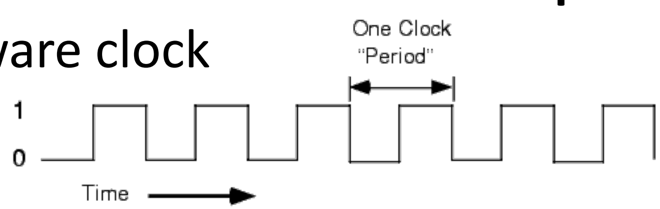
More practically speaking

- Distributed gaming – who grabbed an object first?
- Markets, auctions, trading – who issued order first?
- Multimedia synchronization for real-time teleconferencing
- Target tracking, air traffic control, location positioning

Real time clock (RTC, CMOSC, HWC)

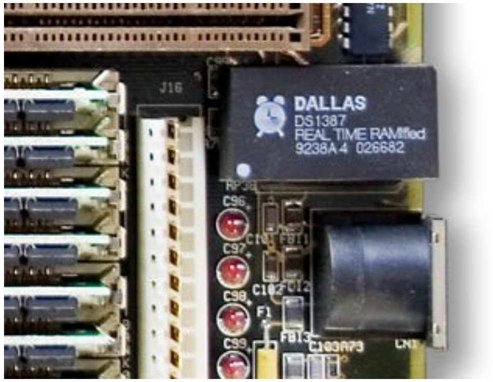
- RTC is used even when the PC is hibernated or switched off
  - Based on alternative low power source
  - Cheap quartz crystal (<$1), inaccurate (+/- 1-15 secs/day)

- Referred to as "**wall clock**" time
  - Synchronizes the **system clock** when computer on
  - Should not be confused with **real-time computing**
  - Neither with hardware clock



- IRQ 8

- sysfs interface /**sys/class/rtc/rtc0 … n**
  UNIX: cat /sys/class/rtc/rtc0/since_epoch
  /sys/class/rtc/rtc0/wakealarm



Source: Wikipedia

Computer "clocks"

Computer clocks count oscillations of a crystal at a defined frequency

Crystal oscillator (quartz crystal)

Counter register

N

N-1

N-2

....

0

Decrements counter for each oscillation.

Interrupt

(one clock tick)

Holding register (N)

frequency often 32.768 kHz (2^{15} cycles per sec.)

Reload upon interrupt

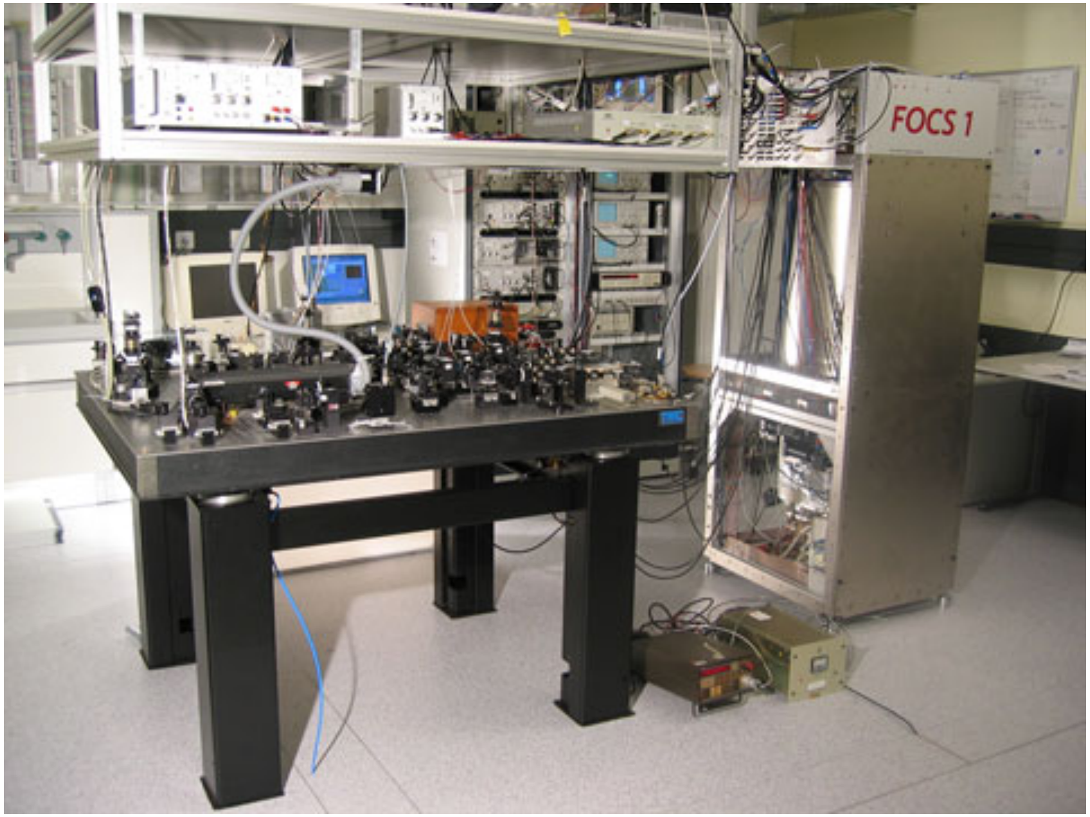
Universal Time Coordinated (UTC)

Temps Universel Coordonné

- **Universal**
  - Standard used around world & Internet (e.g., NTP)
  - Independent from time zones (UTC 0)
  - Converted to local time by adding/subtracting local time zone (EST: UTC-5; CET: UTC+2)
- **Coordinated**
  - 400 institutions contribute their estimates of current time (using **atomic clocks**)
  - UTC is built by combining these estimates

Caesium-133 fountain atomic clock in Switzerland

Uncertainty of one second in 30 million years!



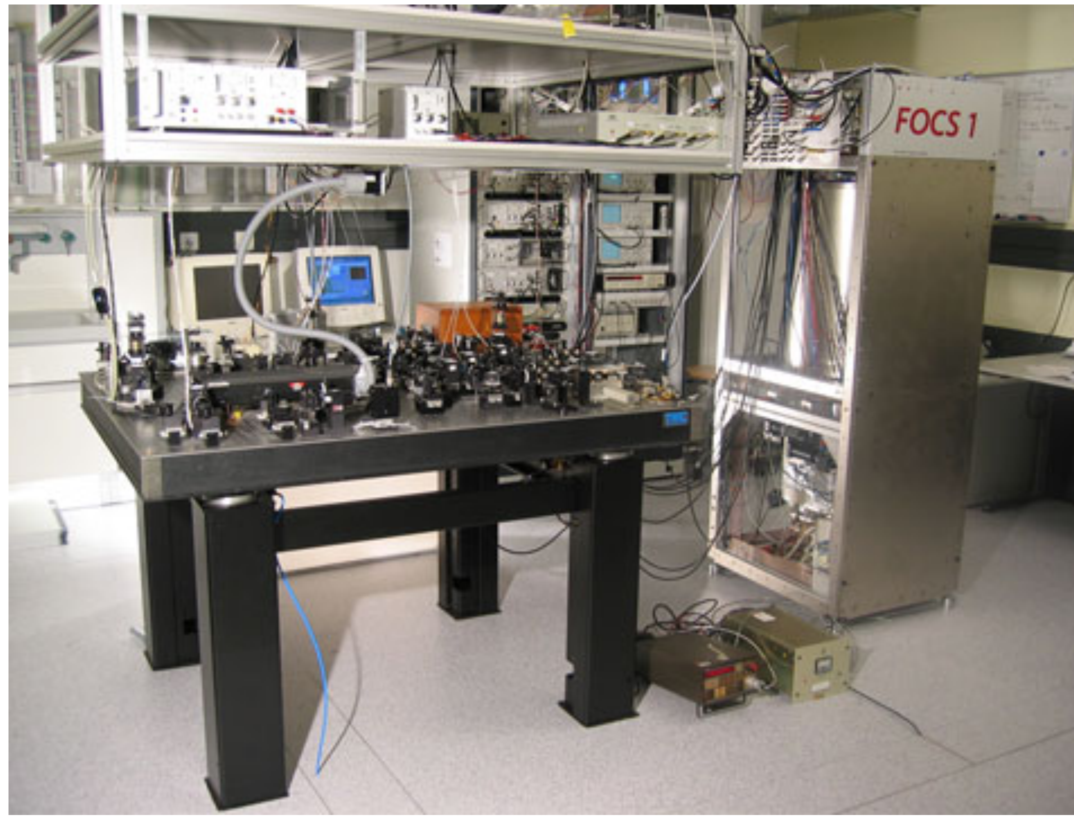
https://en.wikipedia.org/wiki/Atomic_clock

Caesium-133 fountain atomic clock in Switzerland

Uncertainty of one second in 30 million years!

Uncertainty of one second in 15 billion years

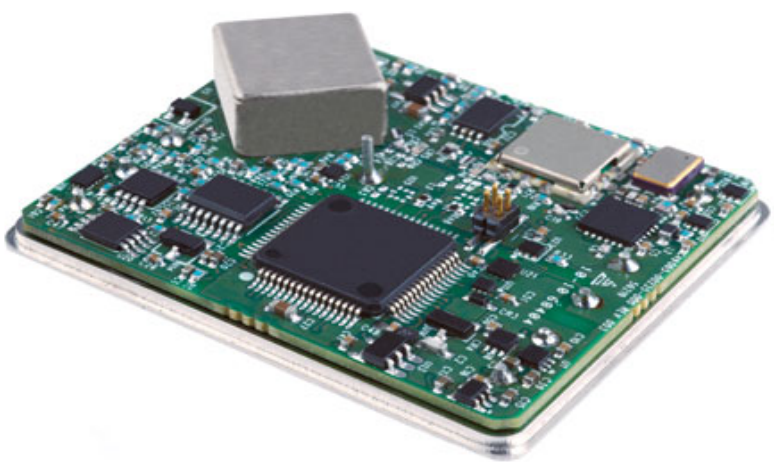
Probably the ``Rolex`` of atomic clocks ☺.



https://en.wikipedia.org/wiki/Atomic_clock

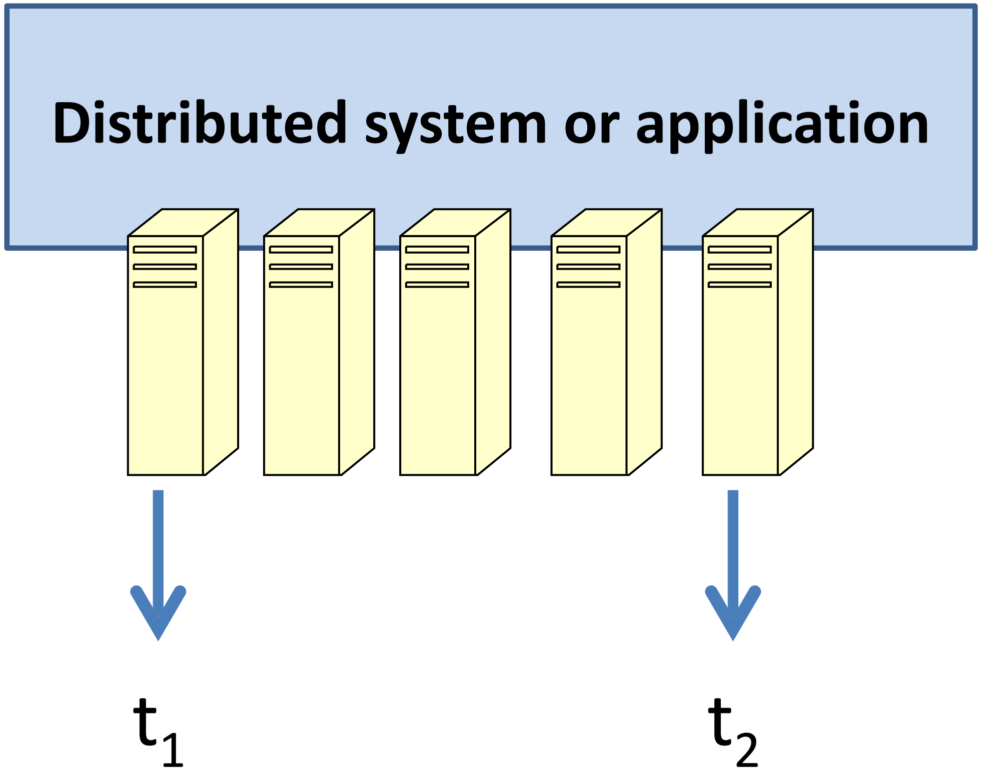
Atomic clocks

Atomic clock on the market May 11, 2011.
Quoted $1500 with **an accuracy of less than 0.5 micro seconds per day.**



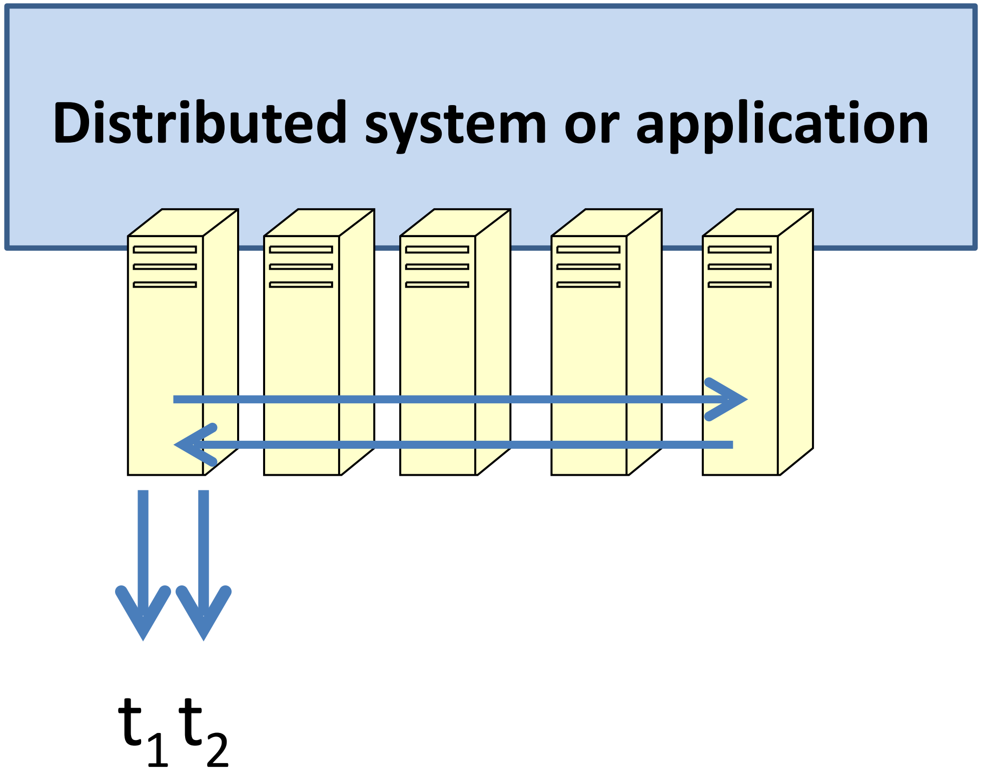
Chip-Scale Atomic Clock. The ultimate in precision--the caesium clock--has been miniaturized By Willie D. Jones Posted 16 Mar 2011.

Measuring latency in distributed systems experiments



$t_2 - t_1 < 0$?

Measuring latency in distributed systems experiments



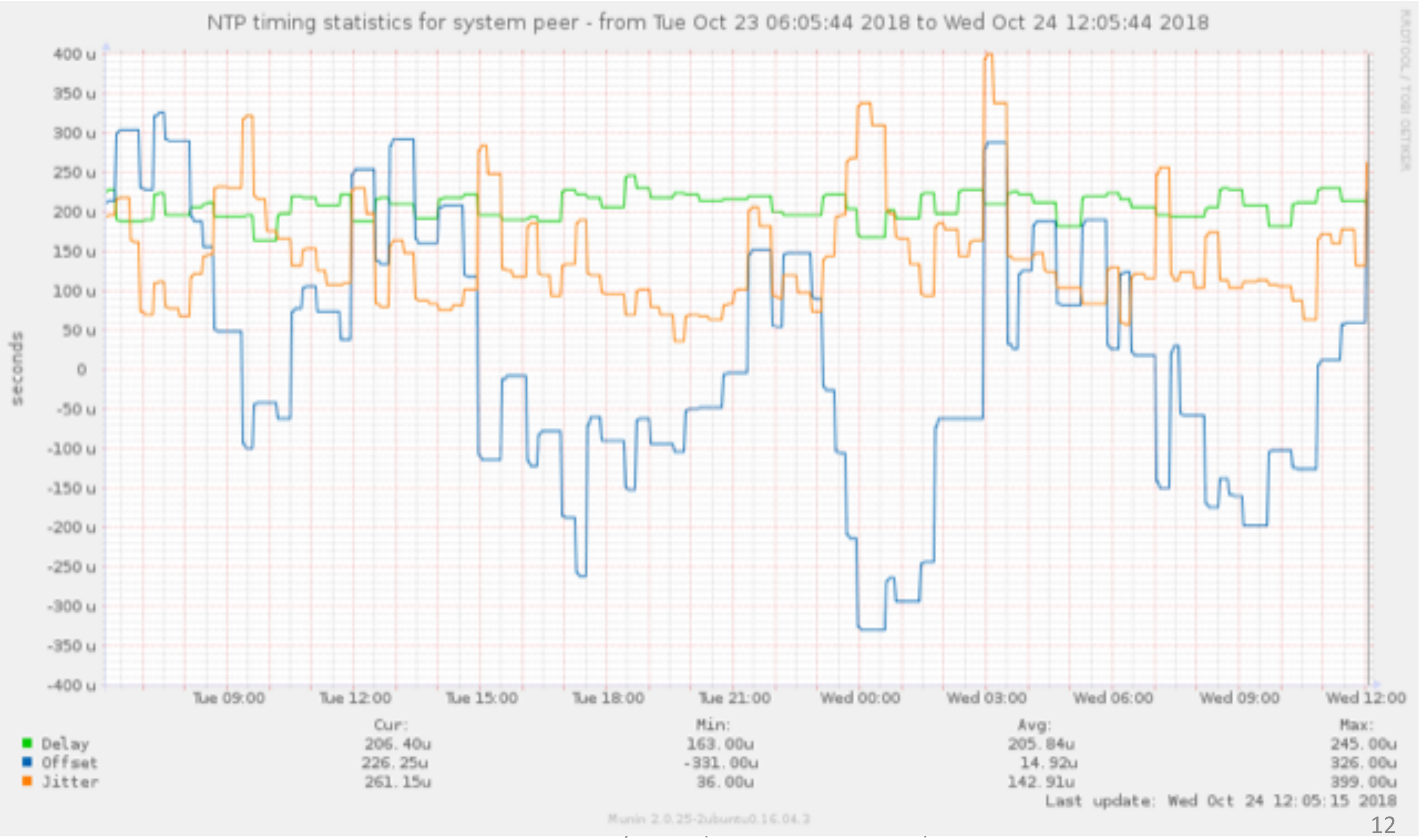
host=node-1 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-2 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-3 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-4 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-5 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-6 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-7 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-8 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-9 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-10 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-11 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-12 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-13 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-14 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-15 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-16 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-17 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-18 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-19 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-20 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-21 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-22 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-23 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-24 rtt=750(187)ms/0ms delta=0ms/0ms

host=node-25 rtt=750(187)ms/0ms delta=1ms/1ms
host=node-26 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-27 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-28 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-29 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-30 rtt=750(187)ms/0ms delta=-1ms/-1ms
host=node-31 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-32 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-33 rtt=750(187)ms/0ms delta=-1ms/-1ms
host=node-34 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-35 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-36 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-37 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-38 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-39 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-40 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-41 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-42 rtt=562(280)ms/0ms delta=290ms/290ms
host=storage-1 rtt=562(280)ms/0ms delta=0ms/0ms
host=storage-2 rtt=750(187)ms/0ms delta=0ms/0ms
host=storage-3 rtt=750(187)ms/0ms delta=0ms/0ms
host=storage-4 rtt=562(280)ms/0ms delta=0ms/0ms

Software-based clock sync times with 1 millisecond precision

NTP timing statistic for single node (in µs)

NTP timing statistic in microseconds for delay, offset and jitter



Clock skew & drift

- **Clock skew**: Instantaneous difference between readings of two clocks

- **Clock drift**: Variation in frequency over time

Summary

- Bad news: Clocks drifts
- Time keeping is not perfect

Self-study questions

- Bring all clock accuracies reported in this unit to the same reference frame, e.g., seconds per day
- Find typical clock accuracies and submit a detailed table with references to the instructor
  - Best submission will be aired in a future lecture

Distributed Computing (1989) 3:146–158

DISTRIBUTED COMPUTING
© Springer-Verlag 1989

Probabilistic clock synchronization

Flaviu Cristian
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA

Flaviu Cristian is a computer scientist at the IBM Almaden Research Center in San Jose, California. He received his PhD from the University of Grenoble, France, in 1979. After carrying out research in operating systems and programming methodology in France, and working on the specification, design, and verification of fault-tolerant programs in England, he joined IBM in 1982. Since then he has worked in the area of fault-tolerant distributed protocols and systems. He has participated in the design and implementation of a highly available system prototype at the Almaden Research Center and has reviewed and consulted for several fault-tolerant distributed system designs, both in Europe and in the American divisions of IBM. He is now a technical leader in the design of a new U.S. Air Traffic Control System which must satisfy very stringent availability requirements.

Abstract. A probabilistic method is proposed for reading remote clocks in distributed systems subject to unbounded random communication delays. The method can achieve clock synchronization precisions superior to those attainable by previously published clock synchronization algorithms. Its use is illustrated by presenting a time service which maintains externally (and hence, internally) synchronized clocks in the presence of process, communication and clock failures.

Key words: Communication – Distributed system – Fault-tolerance – Time service – Clock synchronization

Introduction

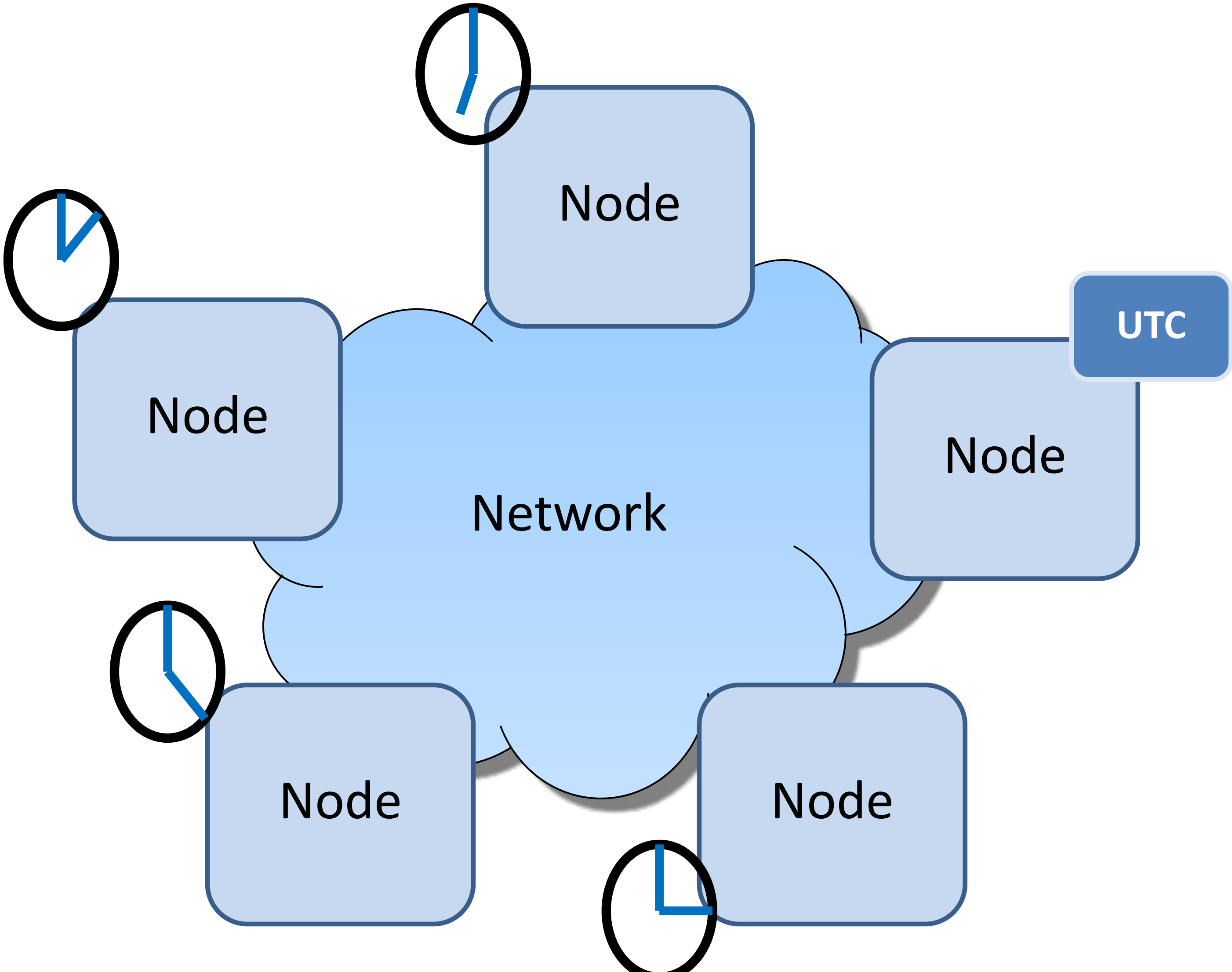
In a distributed system, *external* clock synchronization consists of maintaining processor clocks within some given maximum derivation from a time reference external to the system. *Internal* clock synchronization keeps processor clocks within some maximum relative deviation of each other. Externally synchronized clocks are also internally synchronized. The converse is not true: as time passes internally synchronized clocks can drift arbitrarily far from external time.

Clock synchronization is needed in many distributed systems. Internal clock synchronization enables one to measure the duration of distributed activities that start on one processor and terminate on another processor and to totally order distributed events in a manner that closely approximates their real time precedence. To allow exchange of information about the timing of events with other systems and users, many systems require external clock synchronization. For example external time can be used to record the occurrence of events for later analysis by humans, to instruct a system to take certain actions when certain specified (external) time deadlines occur, and to order the occurrence of related events observed by distinct systems.

This paper proposes a new approach for reading remote clocks in networks subject to unbounded random message delays. The method can be used to improve the precision of both internal and external synchronization algorithms. Our approach is *probabilistic* because it does not guarantee that a processor can always read a remote clock with an a priori specified precision (such a guarantee cannot be provided when there is no bound on message delays). However, by retrying a sufficient number of times, a process can read the clock of another process with a given precision with a probability as close to one a desired. An important characteristic of our method is that when a process succeeds in reading a remote clock, it *knows* the actual reading precision achieved.

CRISTIAN'S CLOCK SYNCHRONIZATION ALGORITHM

External synchronization



Probabilistic clock synchronization

Proposed by F. Cristian (IBM), 1989

- External clock synchronization (therefore, also internal)
- Transmission delay is unbounded but usually reasonably short, especially in LANs
- Thus, no guarantee to achieve an a priori specified clock precision
- For sufficient number of attempts, a desired clock precision can be achieved with an as high a probability as desired
- Primarily intended for operation in LANs

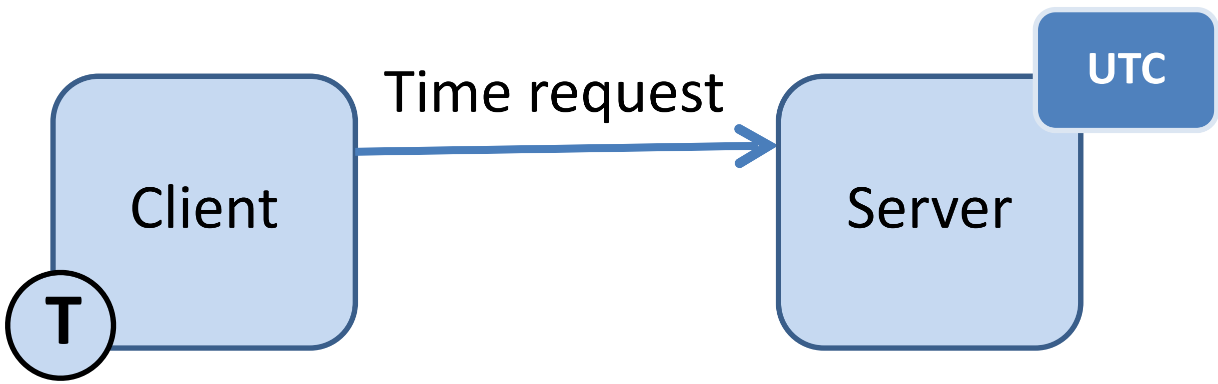
Synchronization request and reply

- Request to reference time source for time
  – Request involves network round trip time (RTT)
  – Response no longer current by time client receives it

- Client must adjust response based on knowledge of network RTT

Synchronization request and reply

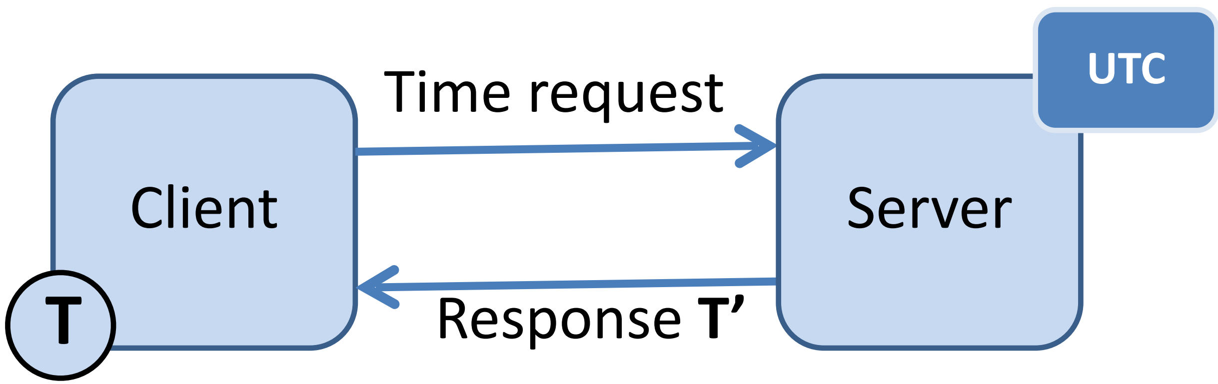
- Request to reference time source for time
  - Request involves network round trip time (RTT)
  - Response no longer current by time client receives it



- Client must adjust response based on knowledge of network RTT

Synchronization request and reply

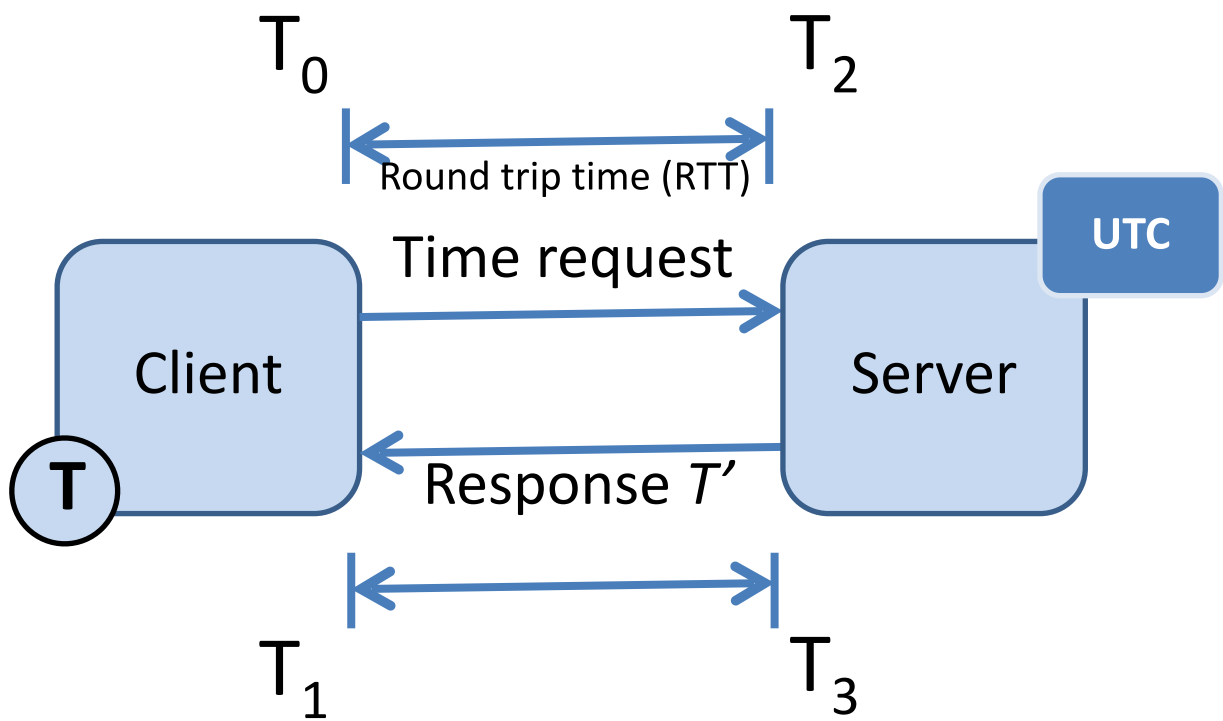
- Request to reference time source for time
  - Request involves network round trip time (RTT)
  - Response no longer current by time client receives it



- Client must adjust response based on knowledge of network RTT

Cristian's algorithm (1989)

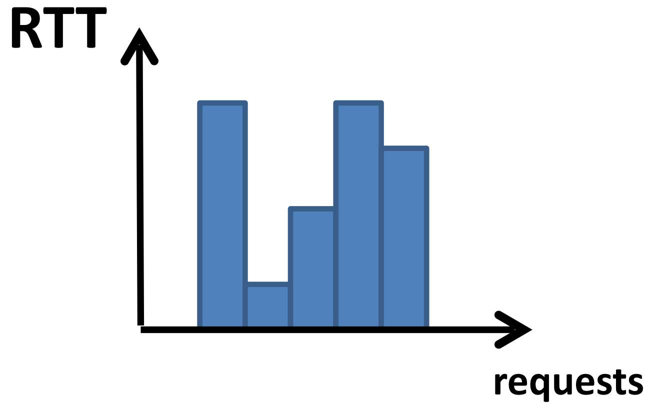
- Client measures **round trip time** for request to server
- Server responds with time value
- Client ***assumes*** transmission **delays split equally**
- Could factor in time to process request at server



$$T = T' + \mathbf{RTT/2}$$

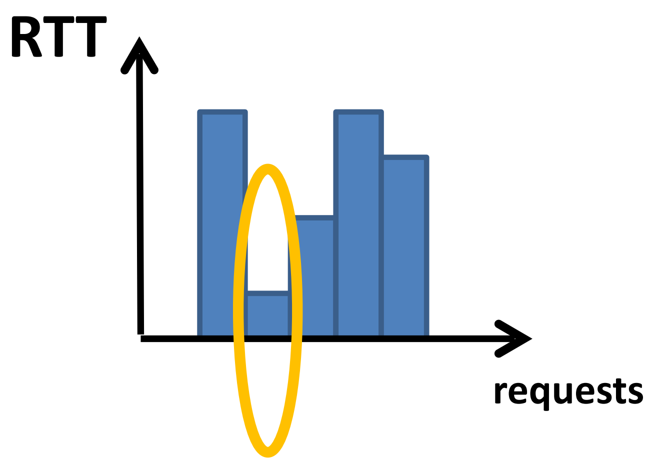
Small improvement

- Make multiple timing requests
- *Which one to use*?
- Accuracy is +/- RTT/2



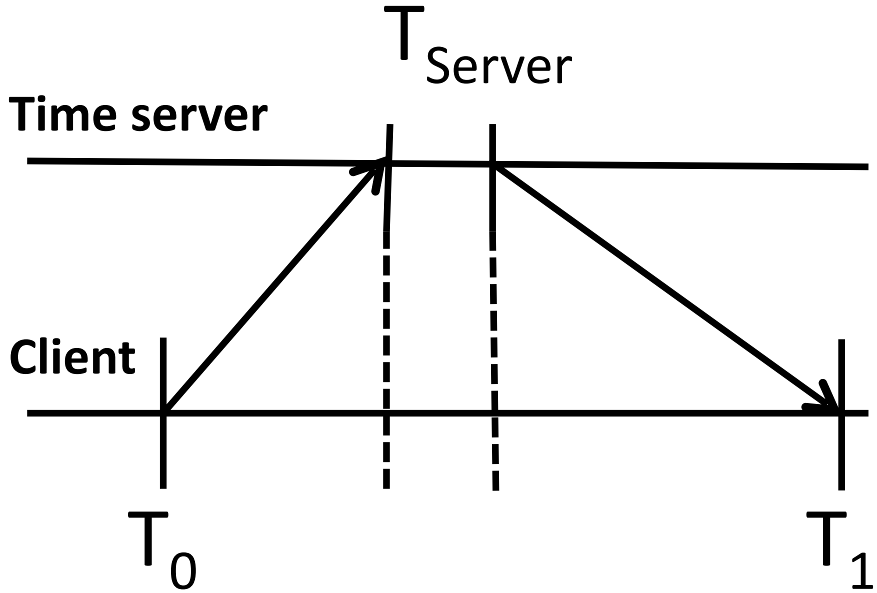
Small improvement

- Make multiple timing requests
- *Which one to use*?
- Accuracy is +/- RTT/2



RTT/2

Accuracy bound



- Time request at T_0
- Time response at T_1
- Assume transmission delays are symmetric (RTT/2)
- Estimate for transmission delay is $(T_1 - T_0)/2$
- New time:

$$\mathbf{T_{new} = T_{server} + (T_1 - T_0)/2}$$

Accuracy bound: +/- $|RTT/2 - T_{min}|$

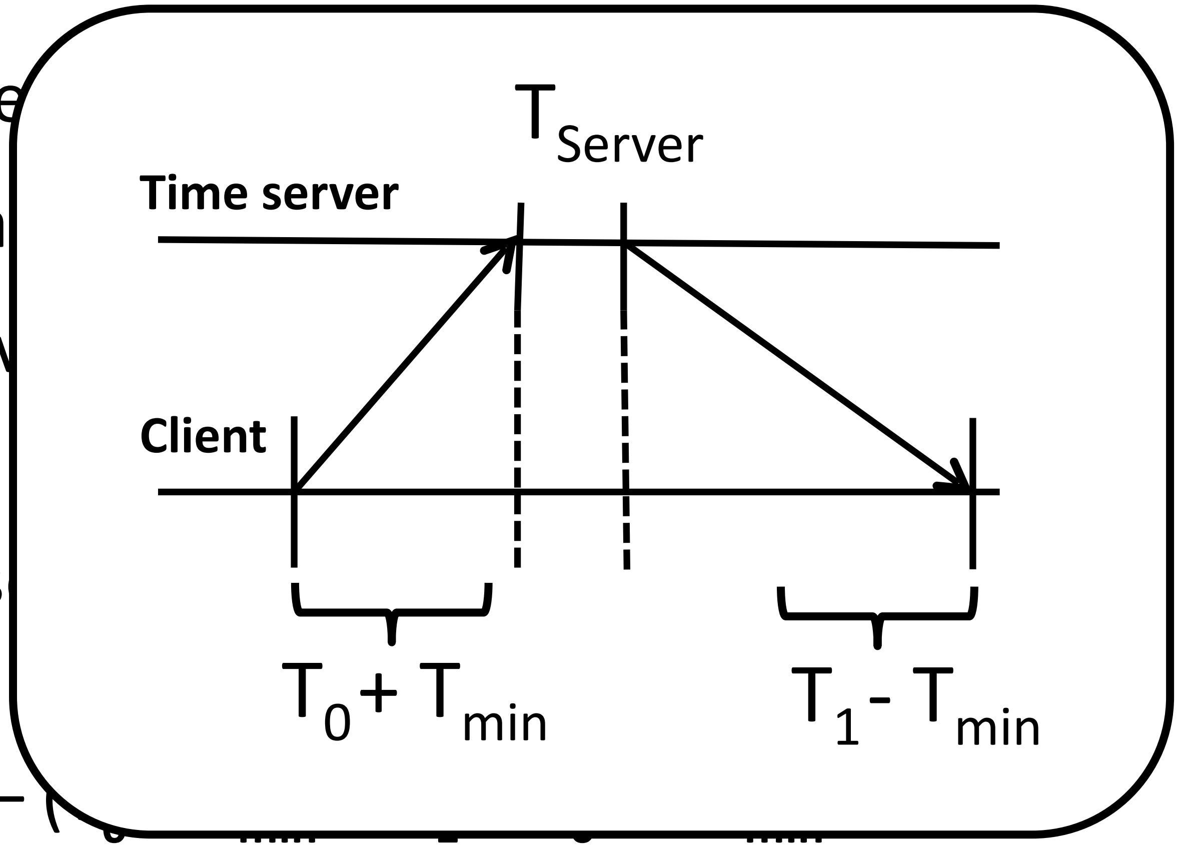
- T_{min} minimum transmission delay (unknown)
- T_0, T_1 as above, ***assume*** T_{min} for propagation
- **Earliest time** server could generate time stamp: $\mathbf{T_0 + T_{min}}$
- **Latest time** server could generate time stamp: $\mathbf{T_1 - T_{min}}$
- Range: $T_1 - T_{min} - (T_0 + T_{min}) = T_1 - T_0 - 2T_{min}$
- **Accuracy: +/-** $|(T_1 - T_0)/2 - T_{min})|$

Accuracy bound: +/- |RTT/2 - T_{min}|

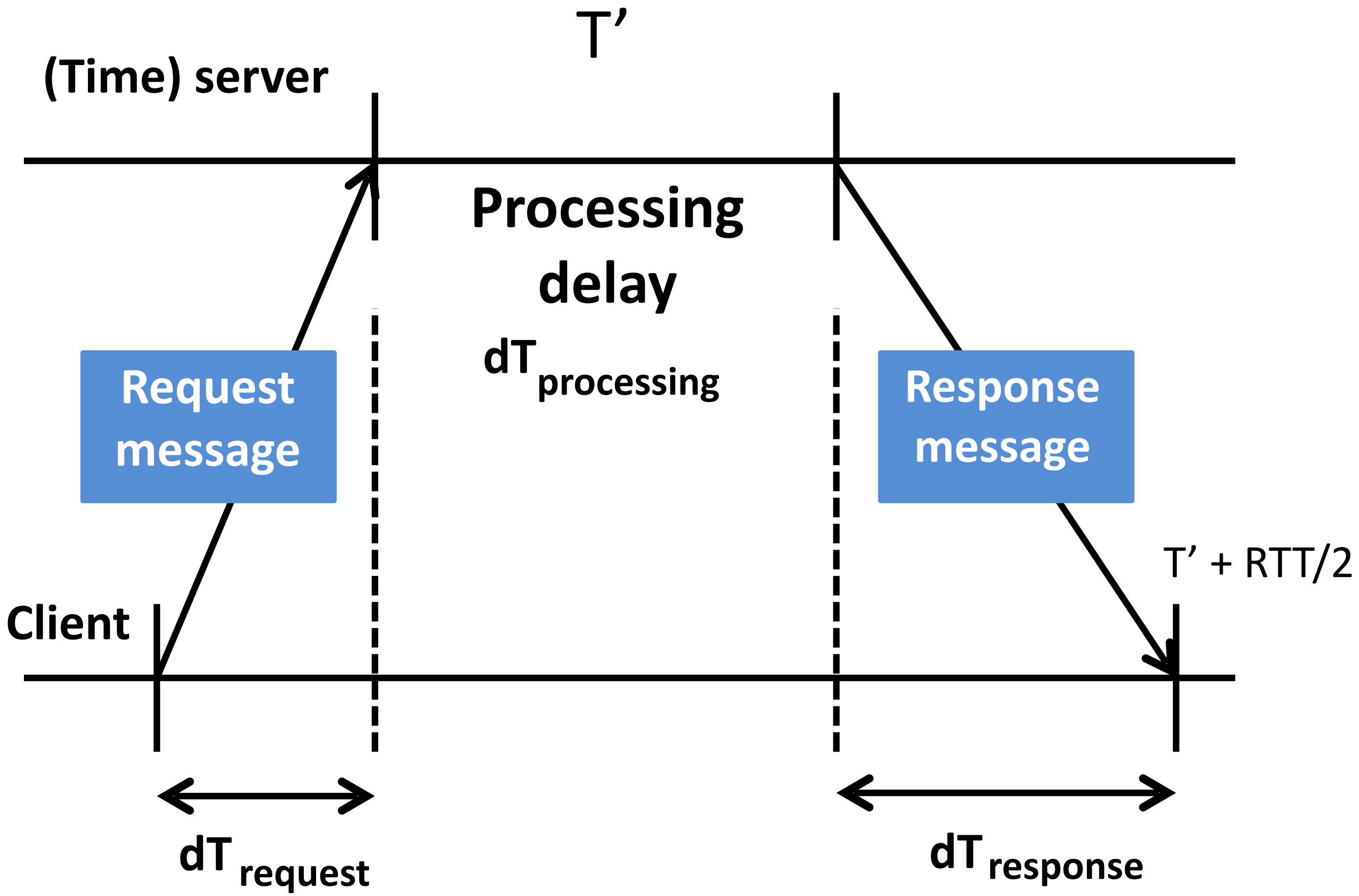
- T_{min} minimum ne
- T_0, T_1 as above, a
- **Earliest time** serv

  $\mathbf{T_{min}}$
- **Latest time** the s

  stamp: $\mathbf{T_1 - T_{min}}$
- Range: $\mathbf{T_1 - T_{min}} - ($
- **Accuracy: +/- | (**$\mathbf{T_1 - T_0}$**)**/2 – $\mathbf{T_{min}}$**) |**



Nota Bene: Factors contributing to RTT



Nota Bene: Errors are cumulative

For time requests over a series of hops

- Say Node B synchronizes time with Node C with an accuracy of +/- 5 *ms*

- Then, Node A synchronizes its time with Node B with an accuracy of +/- 7 *ms*

- Then, the net accuracy at Node A is +/-(7+5) *ms* = +/- 12 *ms*

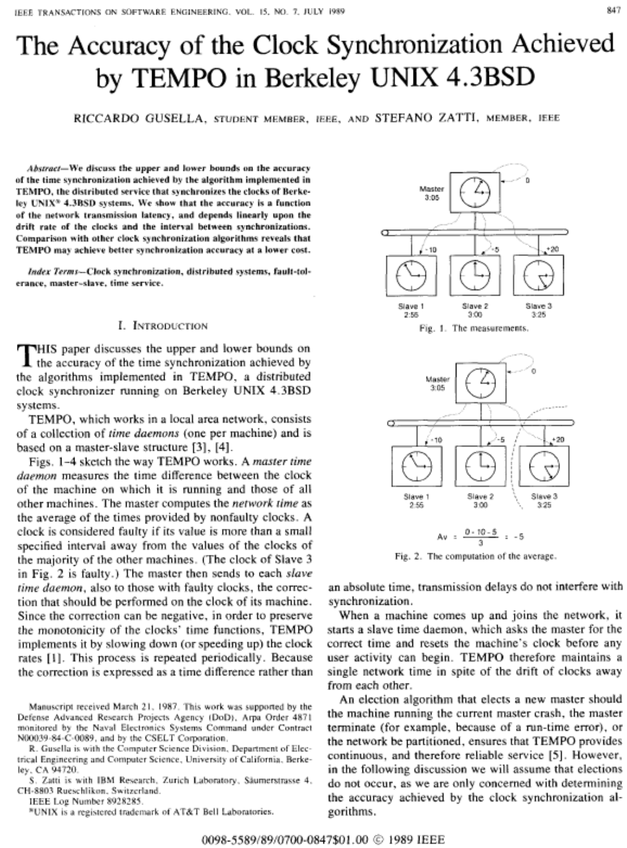
Summarizing observations

- Reliance on centralized time server
  - Distribution possible via broadcast to many servers
  - Communication with single server is preferred
    - Simpler approach
    - Better estimates based on series of requests

- Time server is trusted and single point of failure
  - Malicious, failed server would wreak havoc

Self-study questions

- What are some use case scenarios of external clock synchronization?
- Would resetting a fast clock cause problems?
- Would advancing a slow clock cause problems?
- List all factors that may impact transmission delay of timing requests and provide rough estimates for LANs vs. WANs?
- Update the accuracy bound calculation by taking processing delay into account, what changes?
- Experimentally determine the transmission delay distribution for two nodes on a LAN vs. a WAN.



IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 15, NO. 7, JULY 1989 847

The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD

RICCARDO GUSELLA, STUDENT MEMBER, IEEE, AND STEFANO ZATTI, MEMBER, IEEE

Abstract—We discuss the upper and lower bounds on the accuracy of the time synchronization achieved by the algorithm implemented in TEMPO, the distributed service that synchronizes the clocks of Berkeley UNIX* 4.3BSD systems. We show that the accuracy is a function of the network transmission latency, and depends linearly upon the drift rate of the clocks and the interval between synchronizations. Comparison with other clock synchronization algorithms reveals that TEMPO may achieve better synchronization accuracy at a lower cost.

Index Terms—Clock synchronization, distributed systems, fault-tolerance, master-slave, time service.

I. Introduction

THIS paper discusses the upper and lower bounds on the accuracy of the time synchronization achieved by the algorithms implemented in TEMPO, a distributed clock synchronizer running on Berkeley UNIX 4.3BSD systems.

TEMPO, which works in a local area network, consists of a collection of *time daemons* (one per machine) and is based on a master-slave structure [3], [4].

Figs. 1–4 sketch the way TEMPO works. A *master time daemon* measures the time difference between the clock of the machine on which it is running and those of all other machines. The master computes the *network time* as the average of the times provided by nonfaulty clocks. A clock is considered faulty if its value is more than a small specified interval away from the values of the clocks of the majority of the other machines. (The clock of Slave 3 in Fig. 2 is faulty.) The master then sends to each *slave time daemon*, also to those with faulty clocks, the correction that should be performed on the clock of its machine. Since the correction can be negative, in order to preserve the monotonicity of the clocks' time functions, TEMPO implements it by slowing down (or speeding up) the clock rates [1]. This process is repeated periodically. Because the correction is expressed as a time difference rather than an absolute time, transmission delays do not interfere with synchronization.

Master 3:05 0 -10 -5 +20 Slave 1 2:55 Slave 2 3:00 Slave 3 3:25

Fig. 1. The measurements.

Master 3:05 0 -10 -5 +20 Slave 1 2:55 Slave 2 3:00 Slave 3 3:25

$$\text{Av} = \frac{0 - 10 - 5}{3} = -5$$

Fig. 2. The computation of the average.

When a machine comes up and joins the network, it starts a slave time daemon, which asks the master for the correct time and resets the machine's clock before any user activity can begin. TEMPO therefore maintains a single network time in spite of the drift of clocks away from each other.

An election algorithm that elects a new master should the machine running the current master crash, the master terminate (for example, because of a run-time error), or the network be partitioned, ensures that TEMPO provides continuous, and therefore reliable service [5]. However, in the following discussion we will assume that elections do not occur, as we are only concerned with determining the accuracy achieved by the clock synchronization algorithms.

Manuscript received March 21, 1987. This work was supported by the Defense Advanced Research Projects Agency (DoD), Arpa Order 4871 monitored by the Naval Electronics Systems Command under Contract N00039-84-C-0089, and by the CSELT Corporation.

R. Gusella is with the Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

S. Zatti is with IBM Research, Zurich Laboratory, Säumerstrasse 4, CH-8803 Rueschlikon, Switzerland.

IEEE Log Number 8928285.

*UNIX is a registered trademark of AT&T Bell Laboratories.

0098-5589/89/0700-0847$01.00 © 1989 IEEE

BERKELEY CLOCK SYNCHRONIZATION ALGORITHM

Berkeley algorithm overview

- Physical clock synchronization algorithm developed in 1989 as part of TEMPO in BSD 4.3
- Internal clock synchronization: No node has accurate time source
- Performs clock synchronization to set clocks of all nodes to within a bound of each other
- Intended for use in intranets (LANs)
- TEMPO synchronized clocks to within 20-25 ms in LAN of 15 DEC VAX machines (1989)

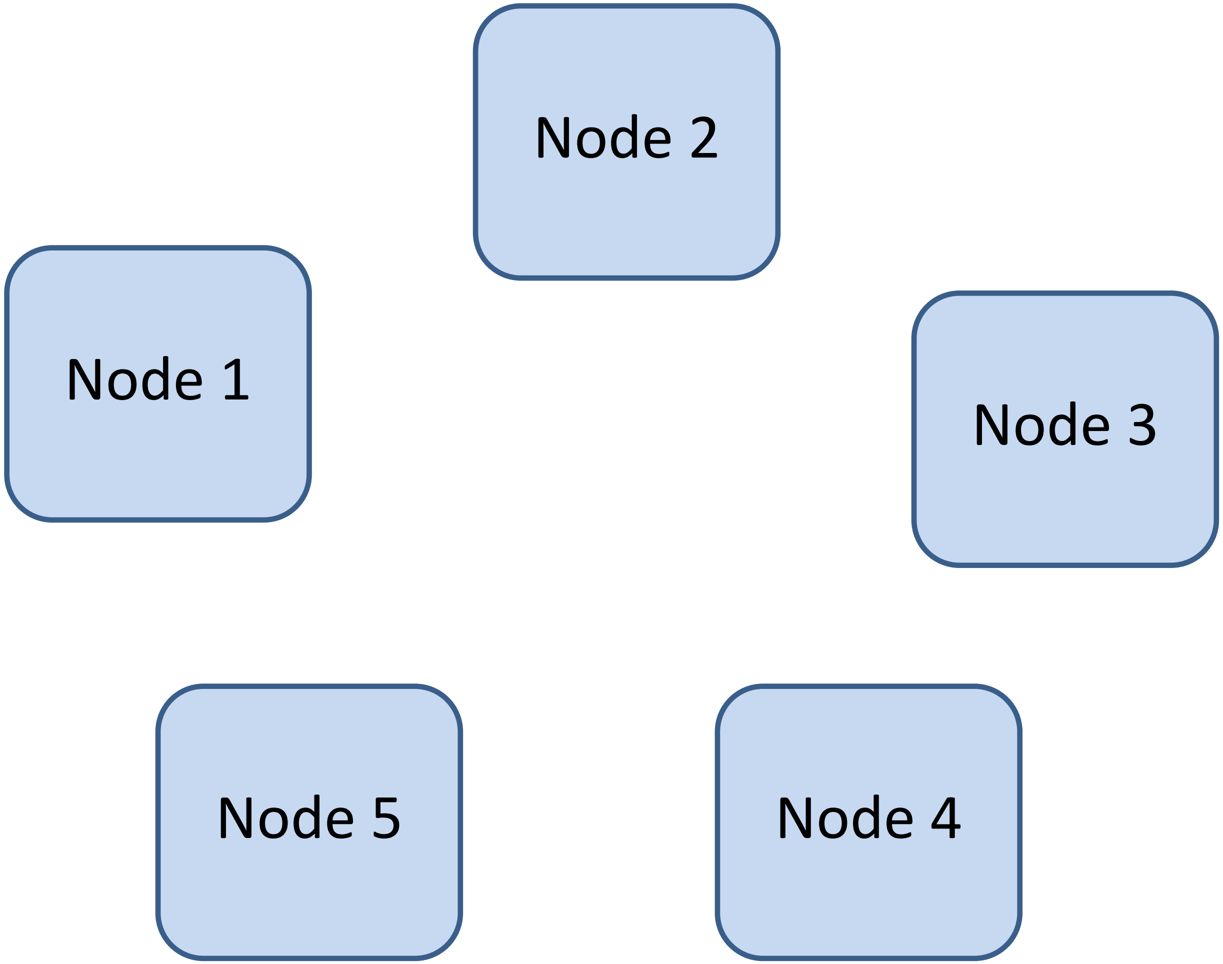
Berkeley algorithm

(a.k.a. TEMPO algorithm in original literature)

- Has a time daemon running on all nodes
- Assumes nodes may be faulty (crash failure model)
- Key idea – runs periodically at designated node
  - Measures time difference between its clock and clock of all nodes
  - Rejects outliers (based on threshold)
  - Averages measurements
  - Requests all nodes to adjust their clocks
- Clock adjustments at each done to respect clocks' monotonicity

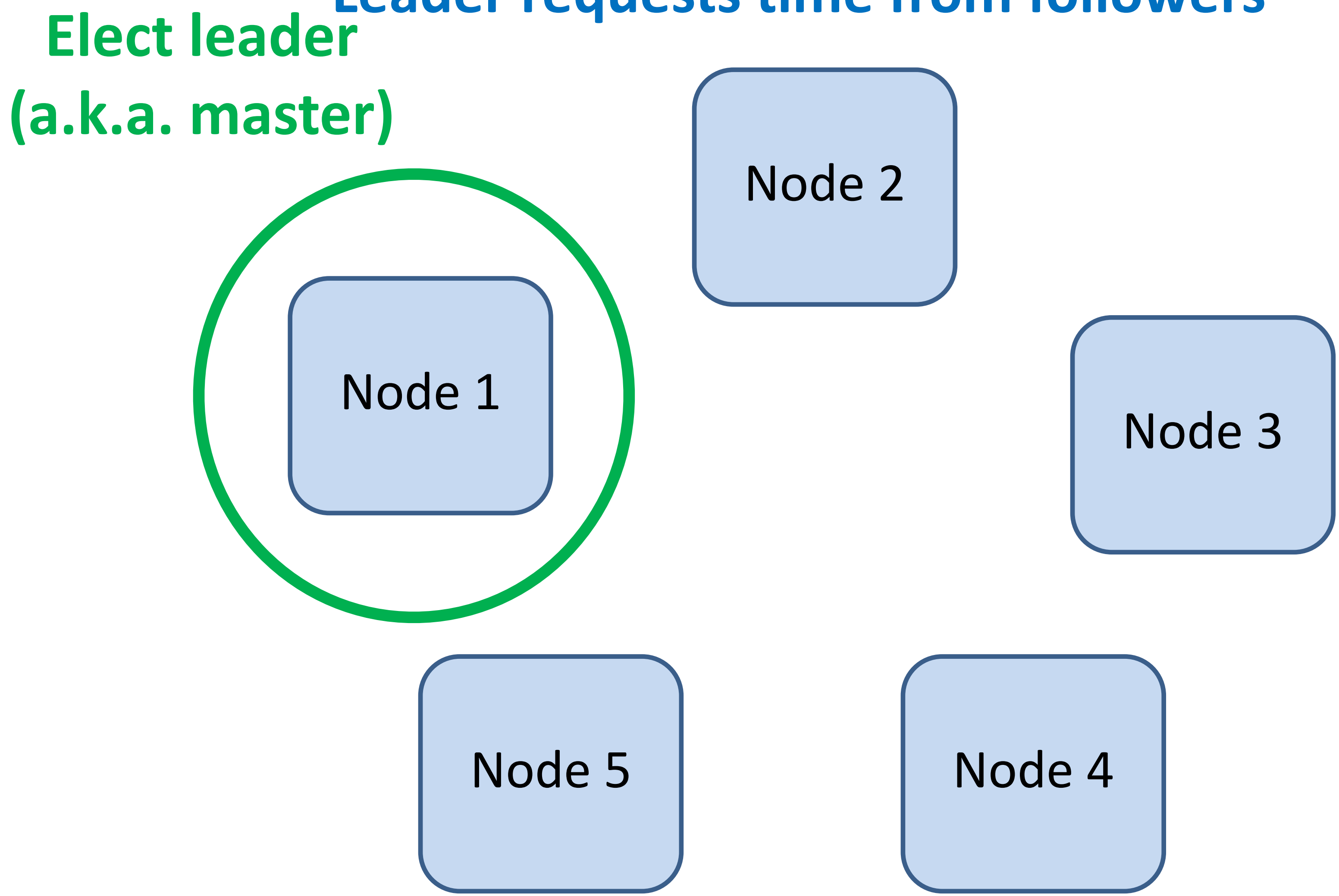
Determine clock differences

Leader requests time from followers

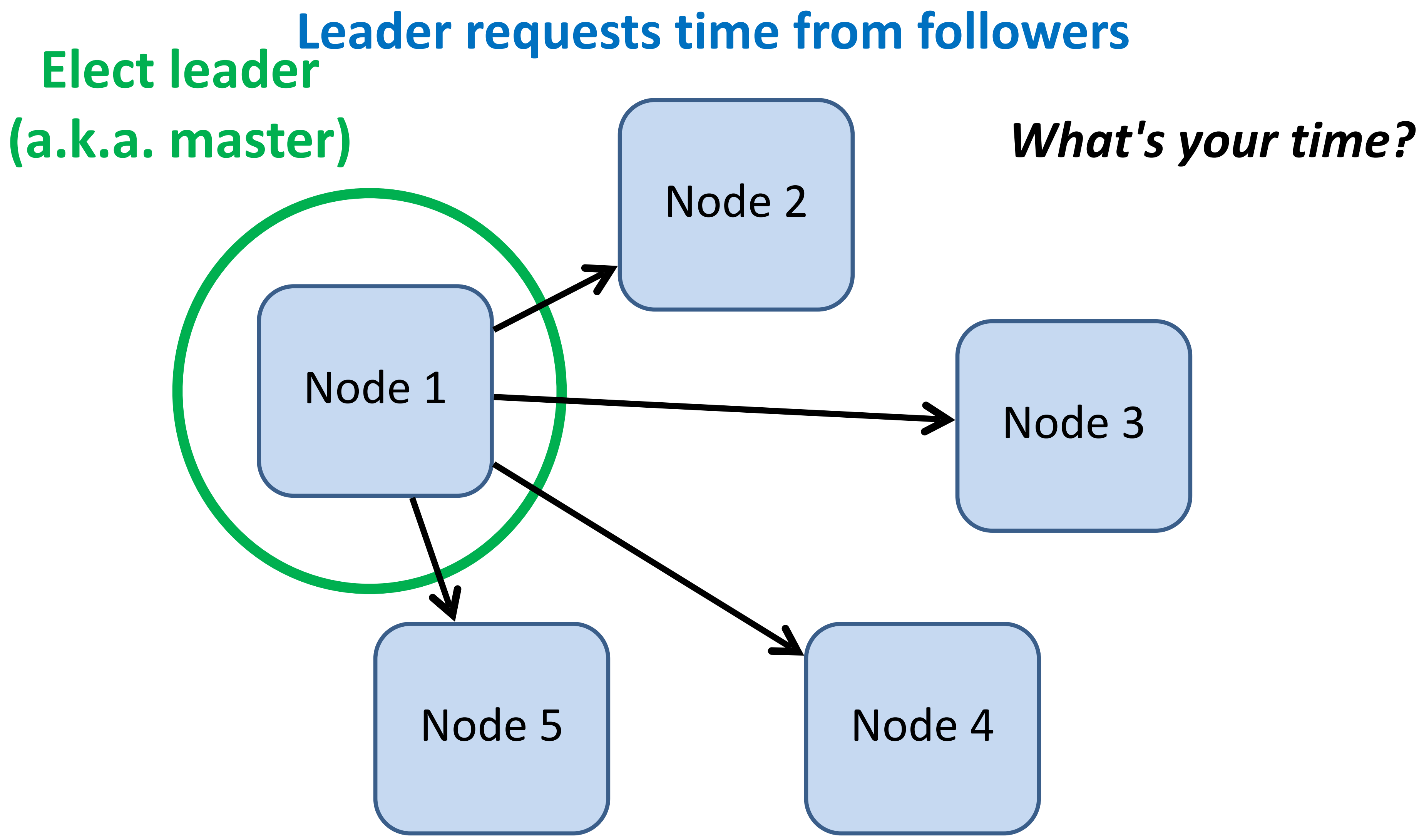


Determine clock differences

Leader requests time from followers



Determine clock differences

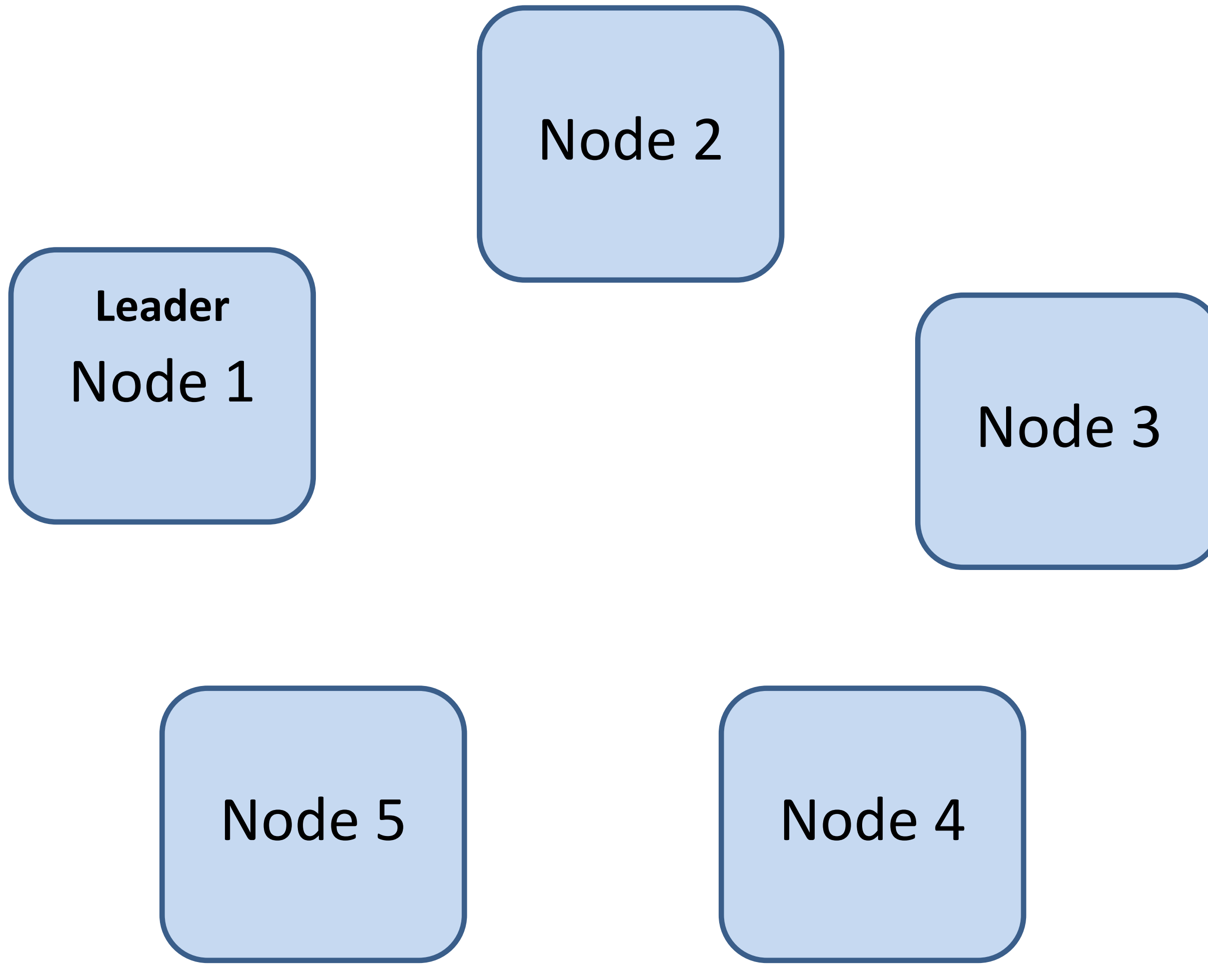


Determine network time

Nodes reply with their time

Nodes reply with their time and leader determines difference to its time

Leader computes **network time**:

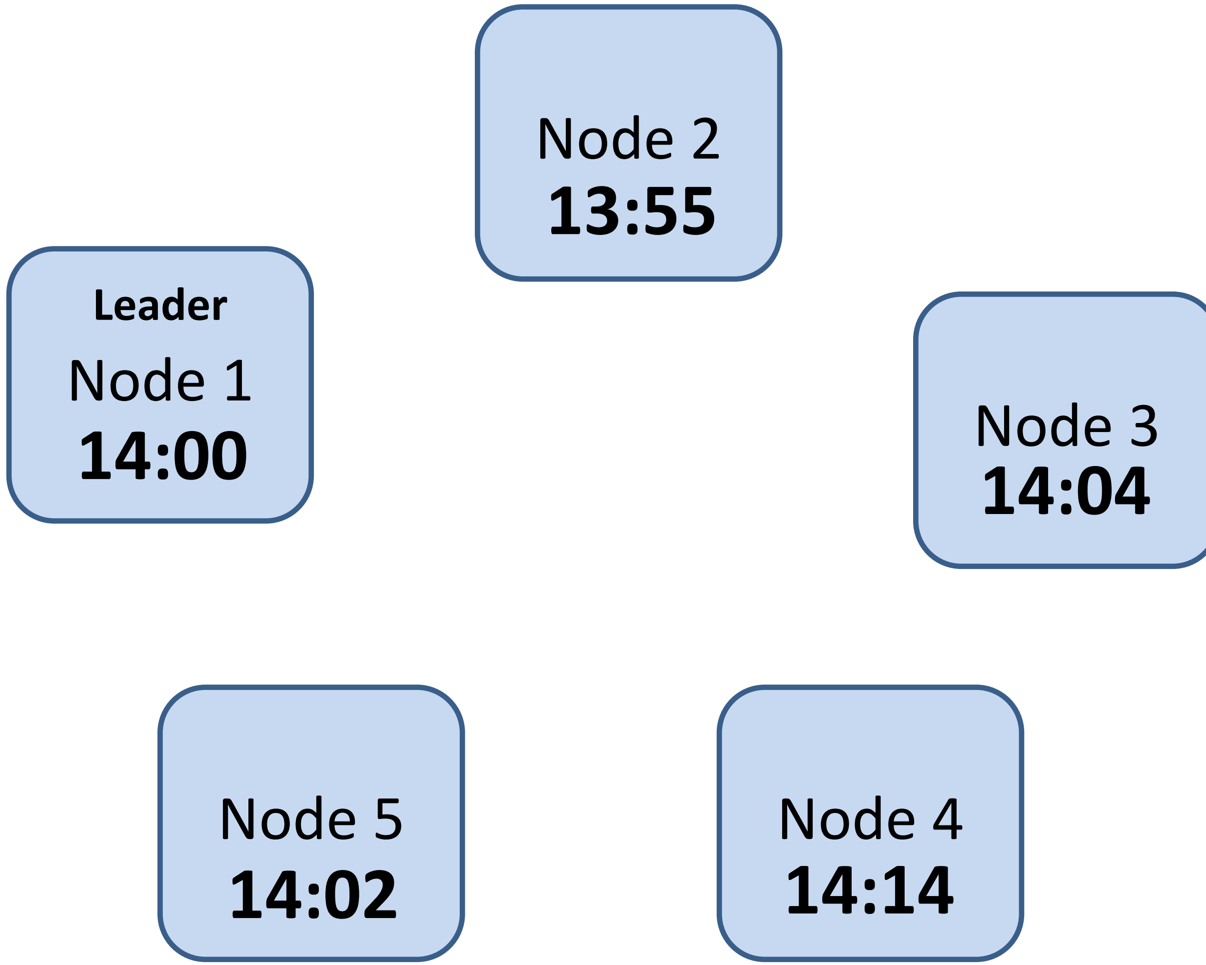


Determine network time

Nodes reply with their time

Nodes reply with their time and leader determines difference to its time

Leader computes **network time**:

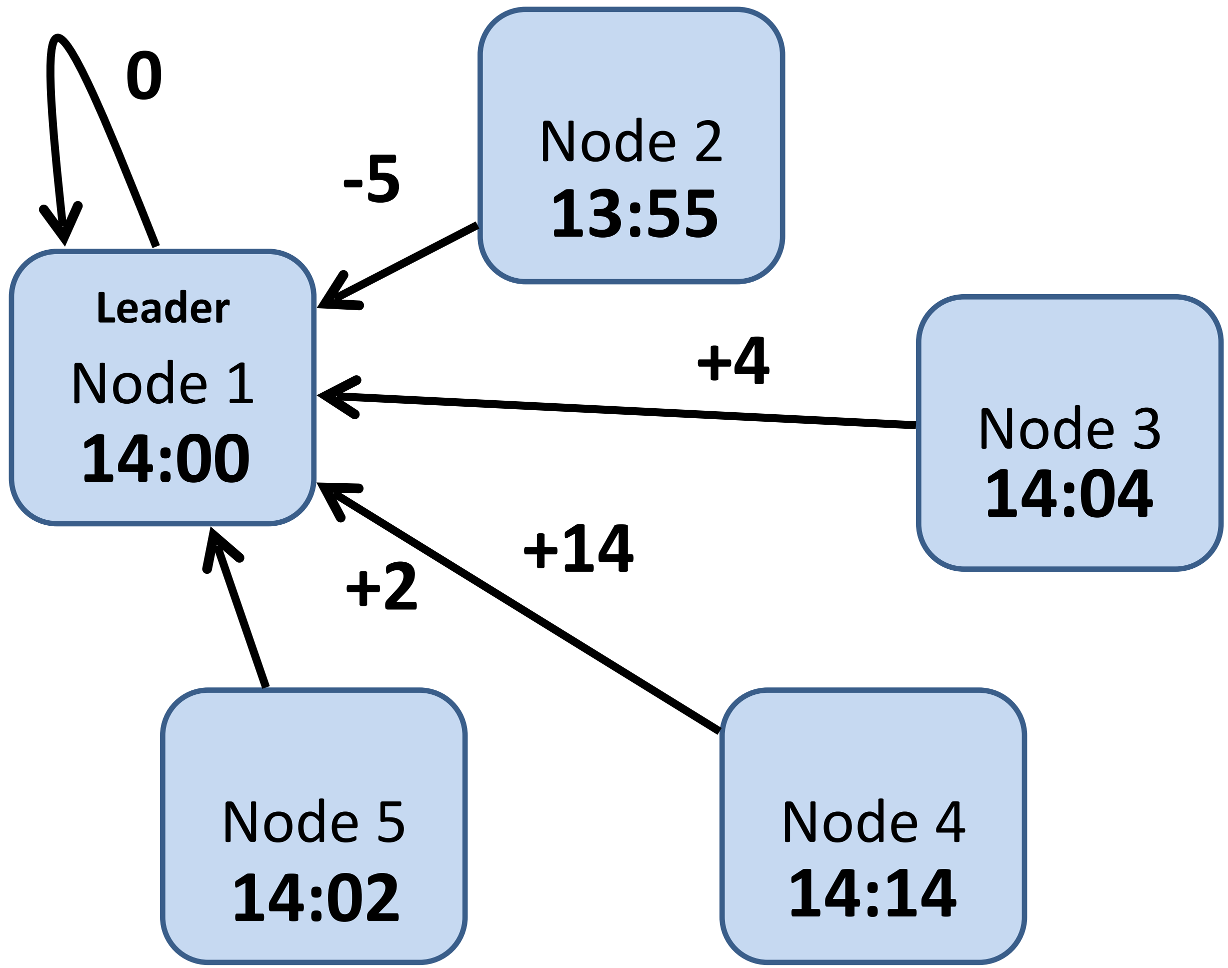


Determine network time

Nodes reply with their time

Nodes reply with their time and leader determines difference to its time

Leader computes **network time**:



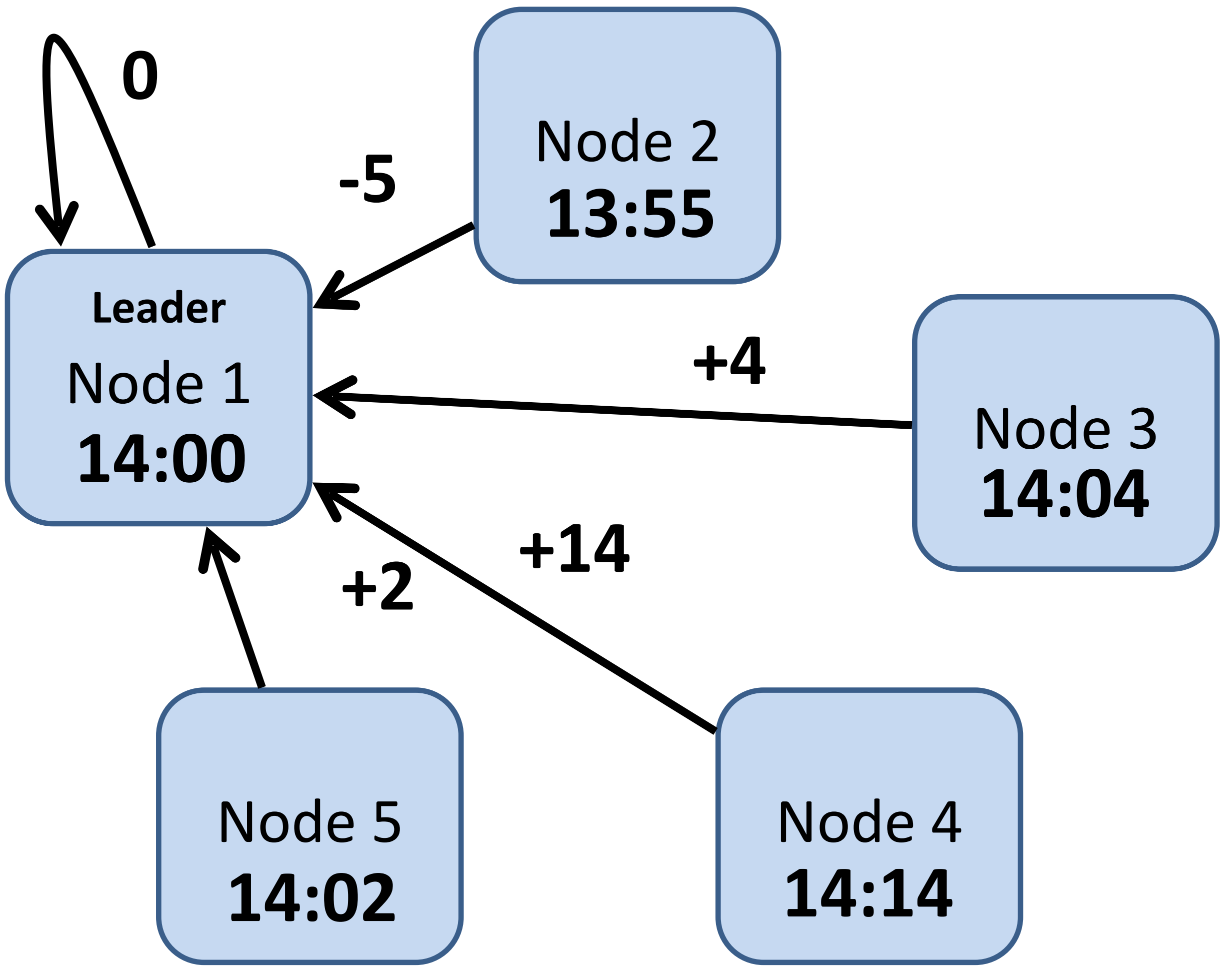
Determine network time

Nodes reply with their time

Nodes reply with their time and leader determines difference to its time

Leader computes **network time**:

$$\frac{\mathbf{0 - 5 + 4 + 14 + 2}}{\mathbf{5}}$$



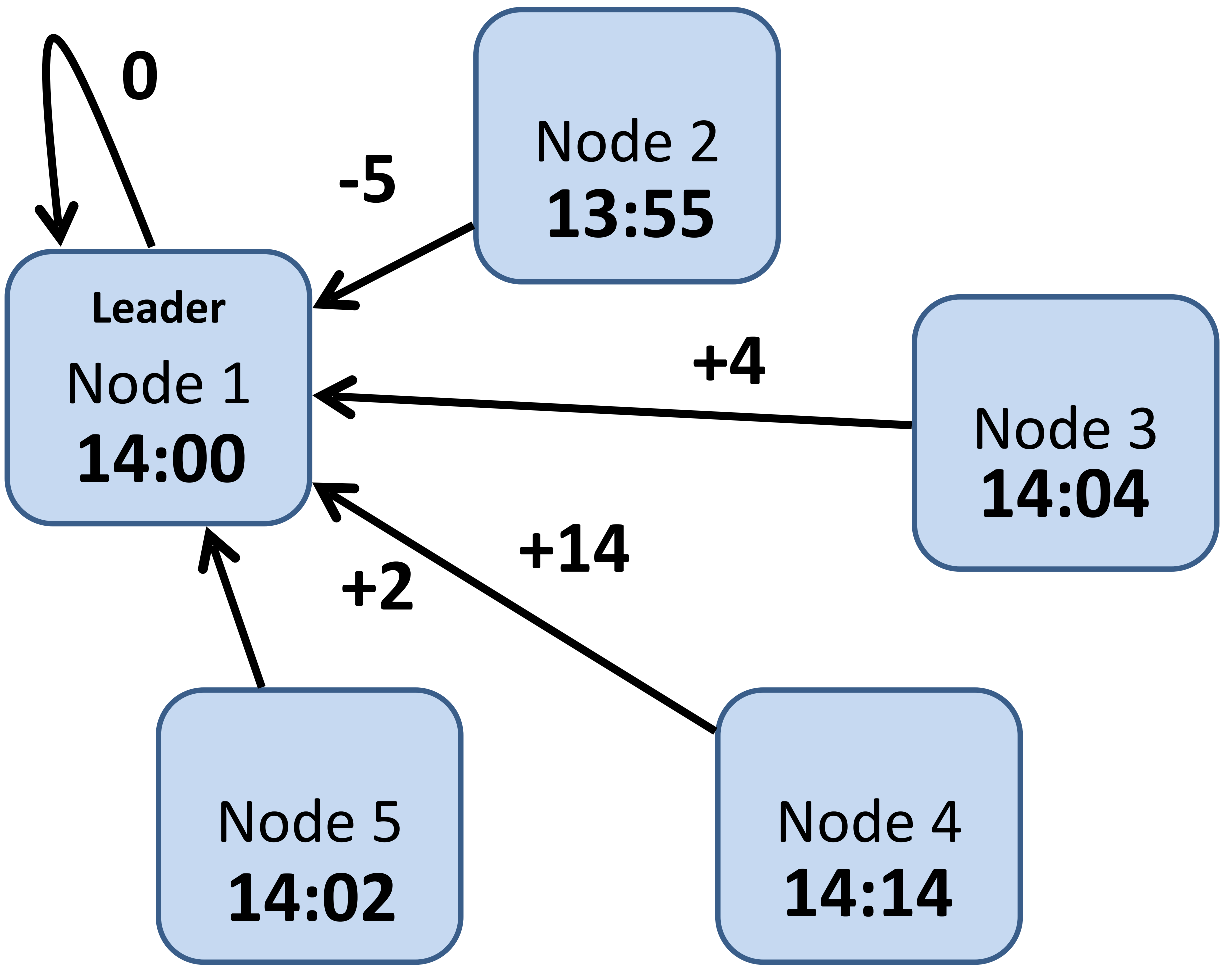
Determine network time

Nodes reply with their time

Nodes reply with their time and leader determines difference to its time

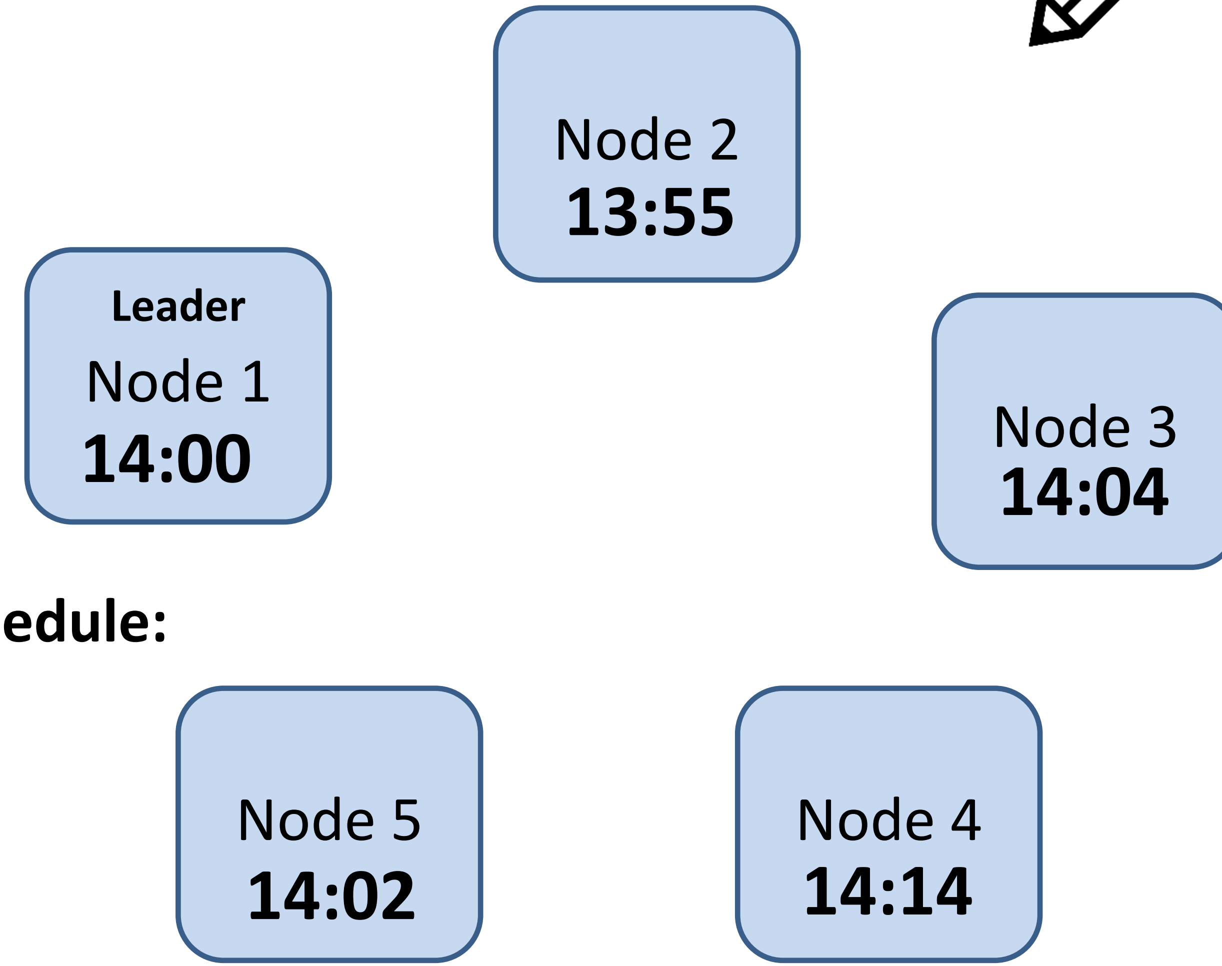
Leader computes **network time**:

$$\frac{0 - 5 + 4 + 14 + 2}{5} = \frac{15}{5} = 3$$



Compute clock adjustments

Network time is
14:03



Leader computes
clock adjustment schedule:

Node 1

Node 2

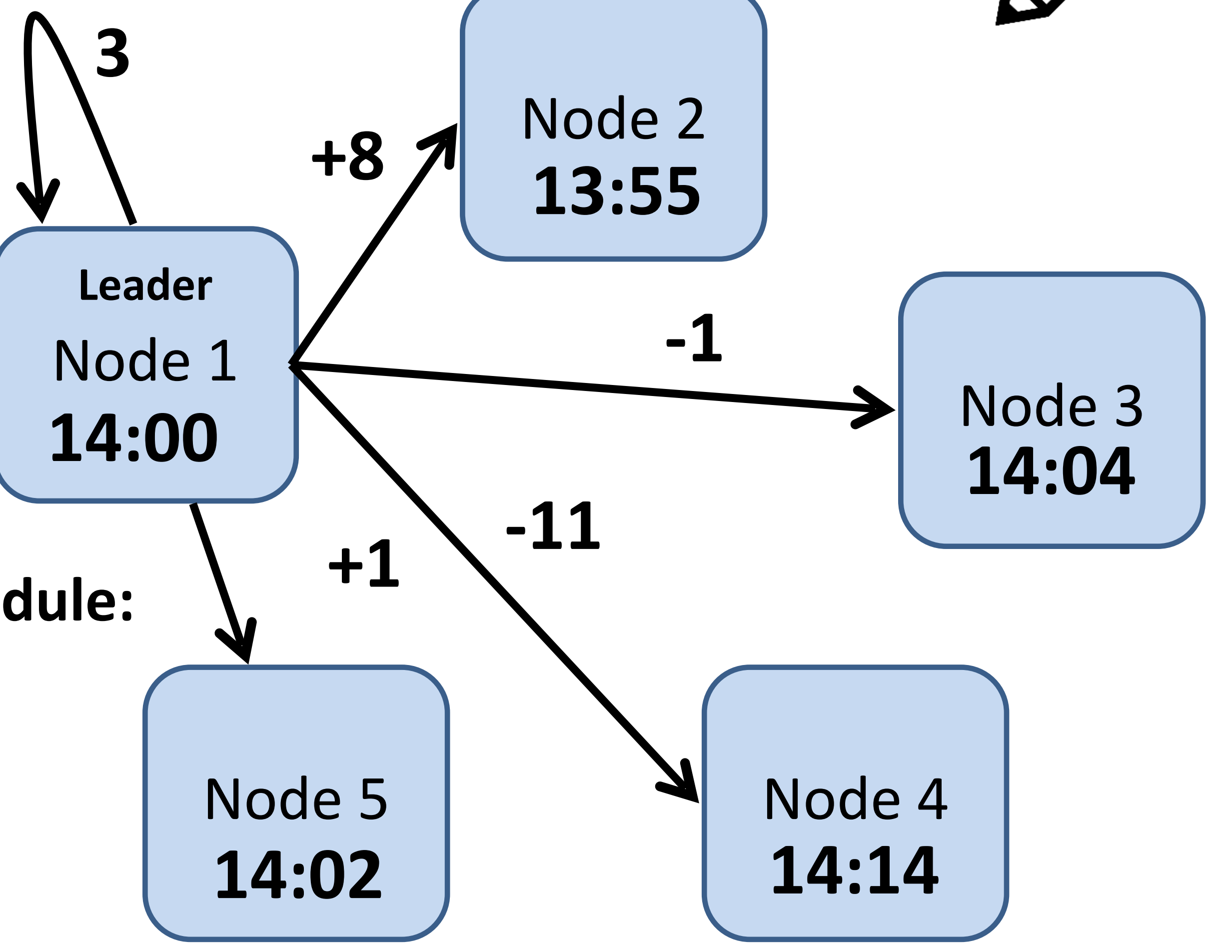
Node 3

Node 4

Node 5

Compute clock adjustments

Network time is
14:03



Leader computes
clock adjustment schedule:

Node 1	**+**	**3**
Node 2	**+**	**8**
Node 3	**-**	**1**
Node 4	**-**	**11**
Node 5	**+**	**1**

Clocks are back in-sync !

Network time is
14:03

Node 2
14:03

Leader
Node 1
14:03

Node 3
14:03

Leader computes
clock adjustment schedule:

Node 1	**+**	**3**
Node 2	**+**	**8**
Node 3	**-**	**1**
Node 4	**-**	**11**
Node 5	**+**	**1**

Node 5
14:03

Node 4
14:03

Determining clock outliers

Avoid adversely effecting "healthy" clocks

- Outliers represent faulty clocks/nodes (malfunctioning, fast drift)
- Leader determines outliers among clock values based on a threshold γ
- Outliers are not used in averaging, i.e., in computing network time
- Leader's clock may represent an outlier itself ***!***
- Faulty clocks are adjusted as well
- Clock considered faulty if its value is more than γ away from the majority of clocks in system
- γ is system-specific configuration parameter

γ

Summarizing observations

- Leader election is separate algorithm
- Local clock adjustment must respect clock's monotonicity requirement – separate algorithm
- Nodes' clocks can be fast or slow
- Leader's request for nodes' times is impacted by transmission delay – needs to be adjusted for
- Clock adjustments (corrections) are expressed as time differences as opposed to absolute times – no impact from transmission delay

Self-study questions

- What are some use case scenarios of internal clock synchronization?
- Would resetting a fast clock cause problems?
- Would advancing a slow clock cause problems?
- Think of an efficient way to determine outliers.
- What factors may impact transmission delay of timing requests?
- What is the impact of transmission delay on absolute timing measures vs. time differences?
- What is a good choice for γ?

“Forward to the Past.”

“Back to the Future.”

```
hwclock –hctosys
```

```
adjtimex --tick 10002 --freq
4000000
```

CLOCK ADJUSTMENT

```
hwclock –systohc
```

```
date -s "13 Jun 2001
10:10:00"
```

Clock adjustment – *the wrong way*

- Adjusting the clock is not straight forward
  - ***T* = *T'* + RTT/2** *(strict no-no - ☹ !)*
  - Time must be **continuous** and **increase monotonically**
- Cannot **go back to the past**
  - Timestamps are important, can't repeat them
- Cannot **jump into future – show sudden jumps**
  - Lose time and miss deadlines

- *So, what are we to do?*

Clock adjustment – *the wrong way*

- Adjusting the clock is not straight forward
  - ***T* = *T'* + RTT/2** *(strict no-no - ☹ !)*
  - Time must be **continuous** and **increase monotonically**
- Cannot **go back to the past**
  - Timestamps are important, can't repeat them
- Cannot **jump into future – show sudden jumps**
  - Lose time and miss deadlines

The time must go on!

- *So, what are we to do?*

Hardware vs. software clock

- At real time t, $H(t)$ represents time on node's ***hardware clock***
- $C(t)$ is time calculated on computer's ***software clock***:

$$C(t) \quad = \quad a * H(t) + b$$

- **Clock monotonicity requirement**:

$$t' > t: \quad C(t') \; > \; C(t)$$

- Achieve monotonicity by adjusting a and b in $C(t) = a * H(t) + b$

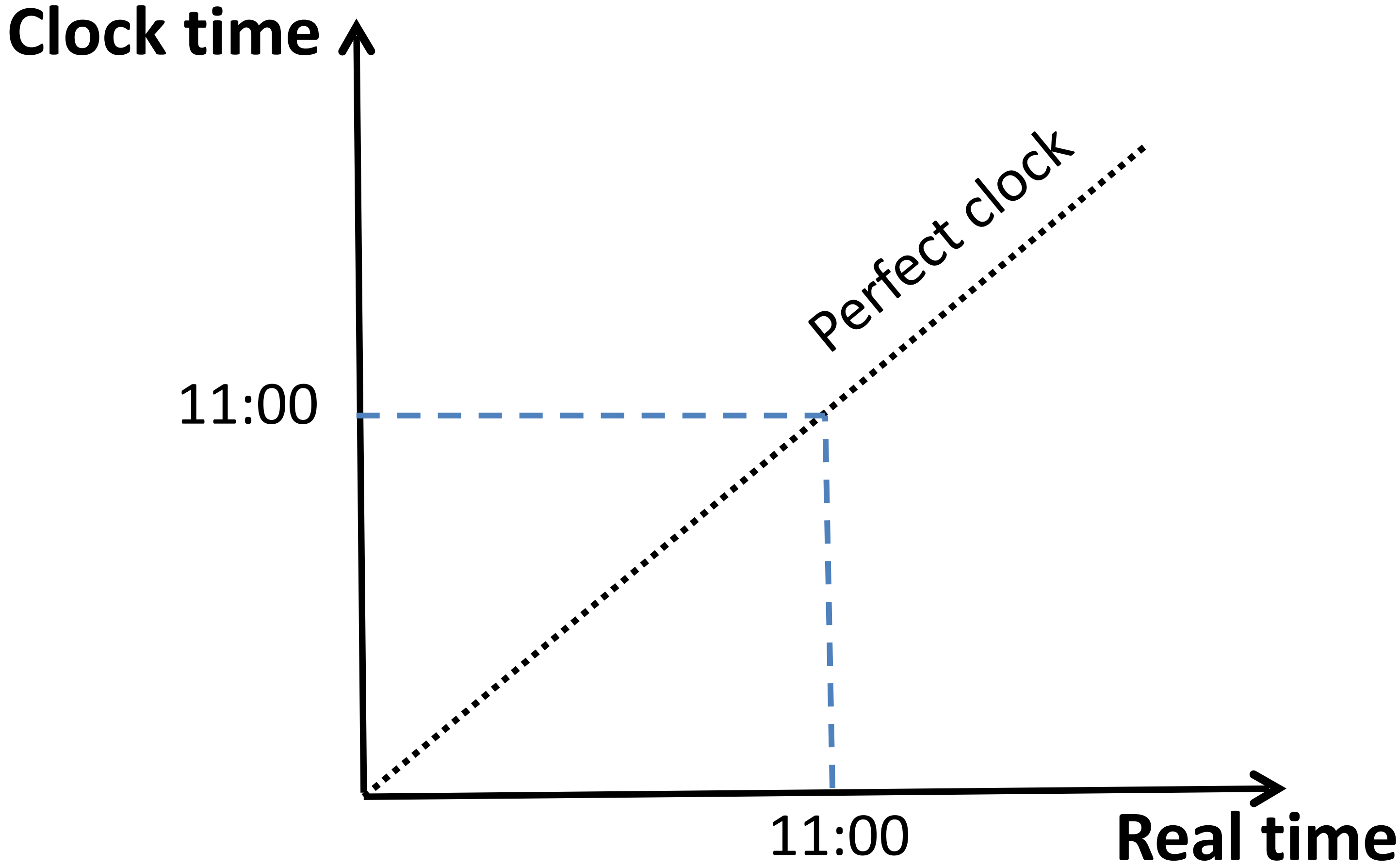
Clock adjustment – *the right way*

- Two parameters are constant **offset** and **slope**:

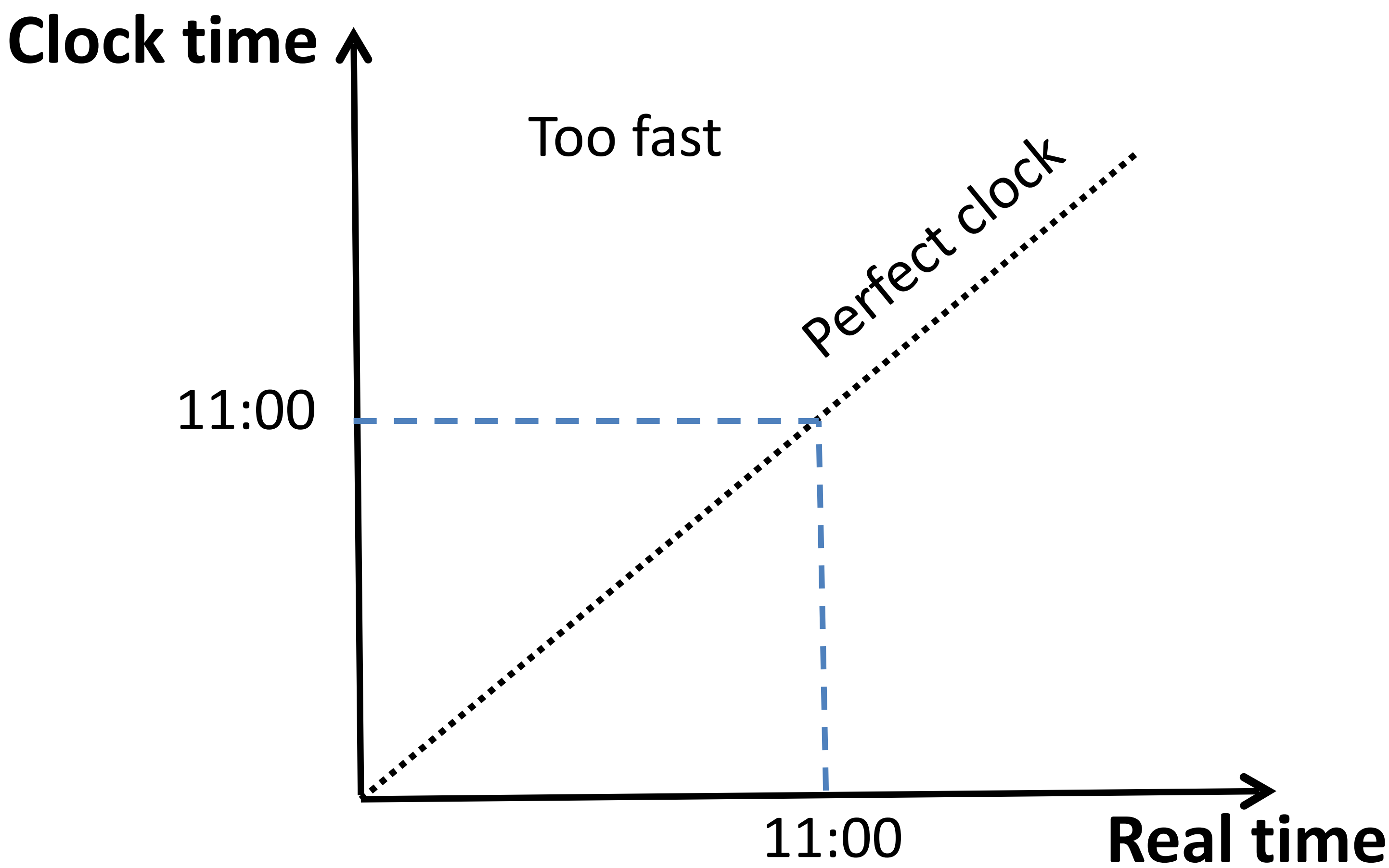
$$C(t) = \boldsymbol{a} * H(t) + \boldsymbol{b}$$

- Determine "catch up" value for scaling time ***down*** or ***up*** in a linear fashion
  - Run software clock slower vs. faster
  - Until it attains the real time (i.e., time measured) or
  - Until another synchronization is taking place

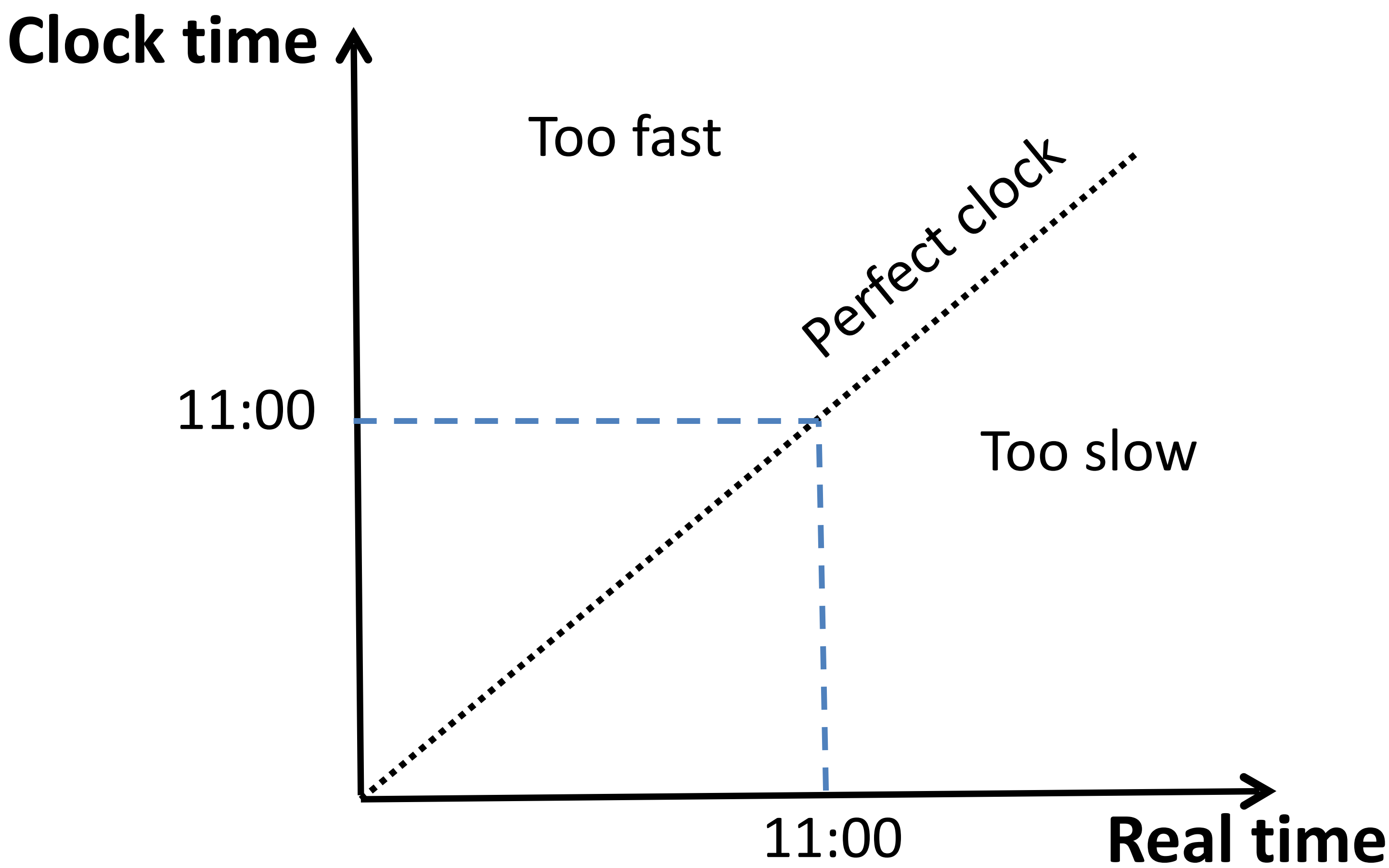
Perfect clock



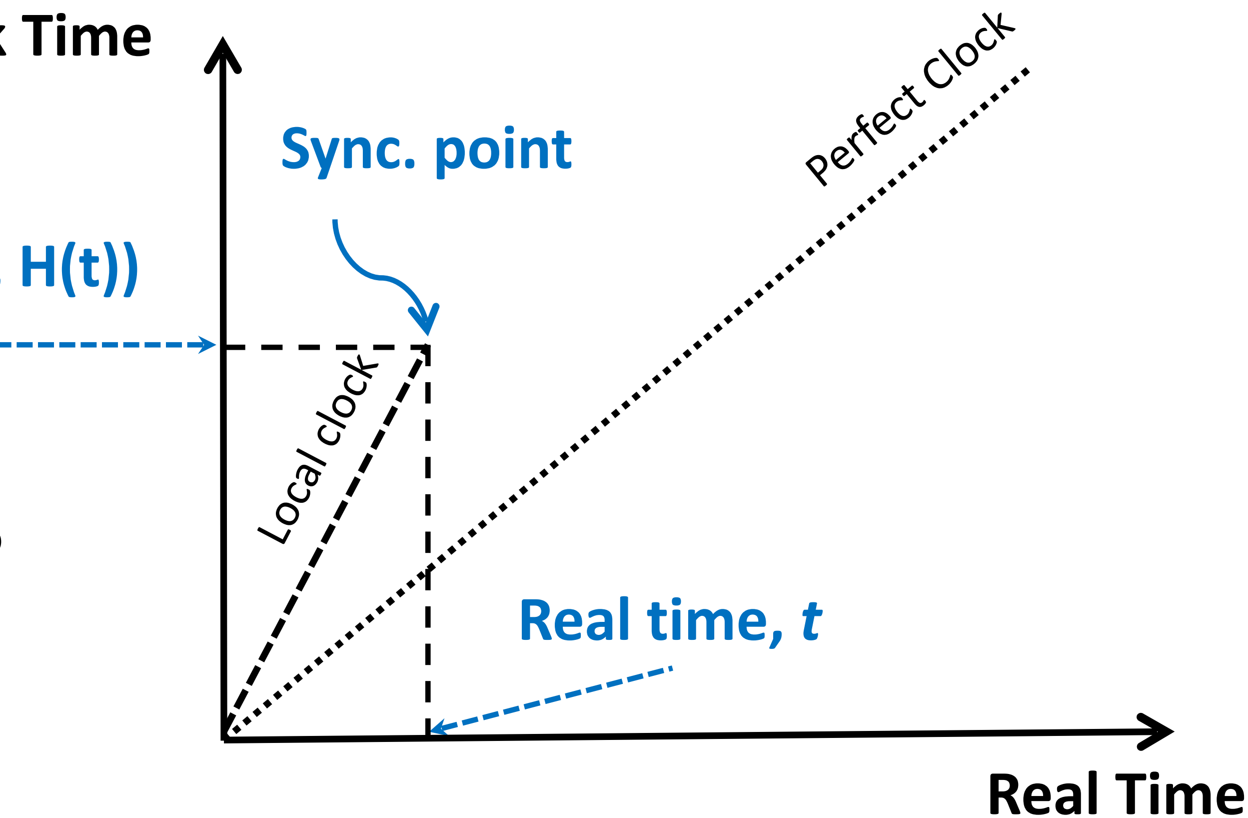
Perfect clock



Perfect clock



Clock too fast

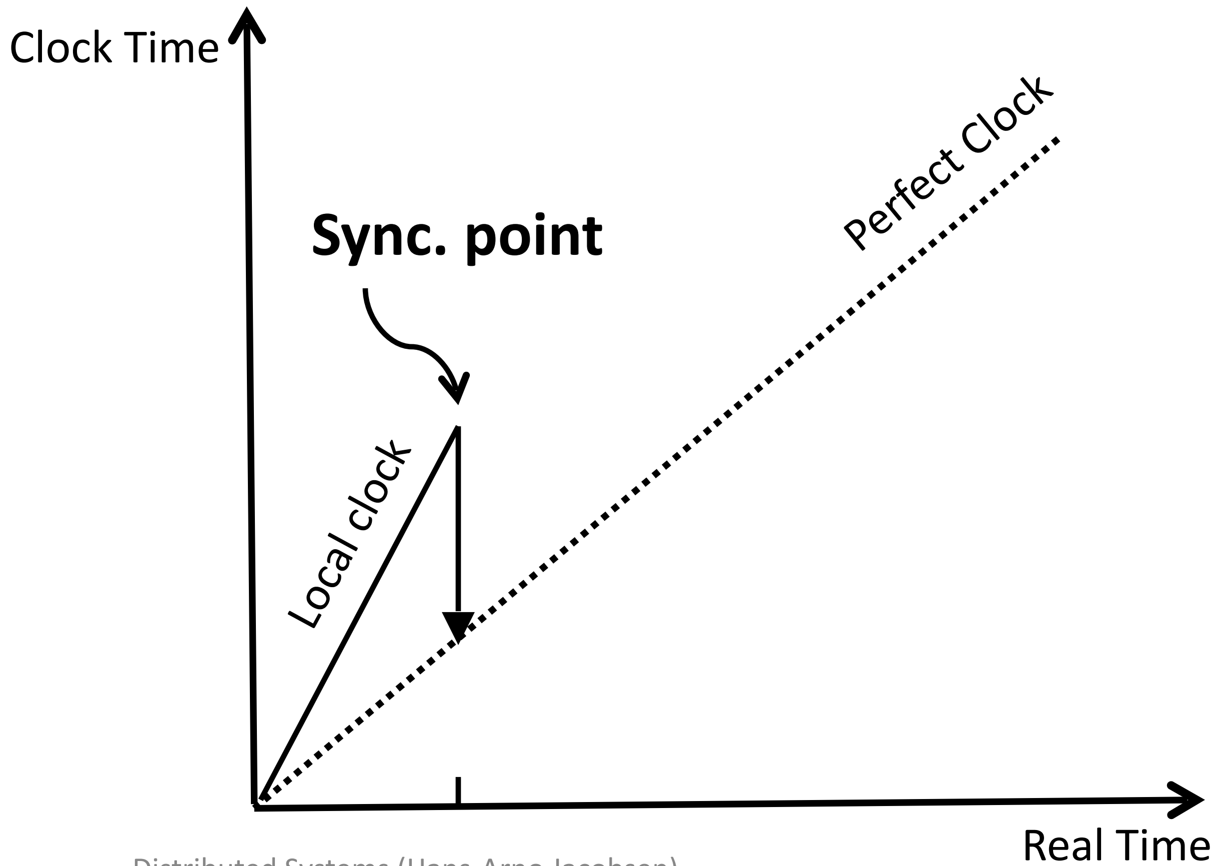


- At some synchronisation point, **clock is** discovered to be **running fast**

- Real time according to a UTC source, for example

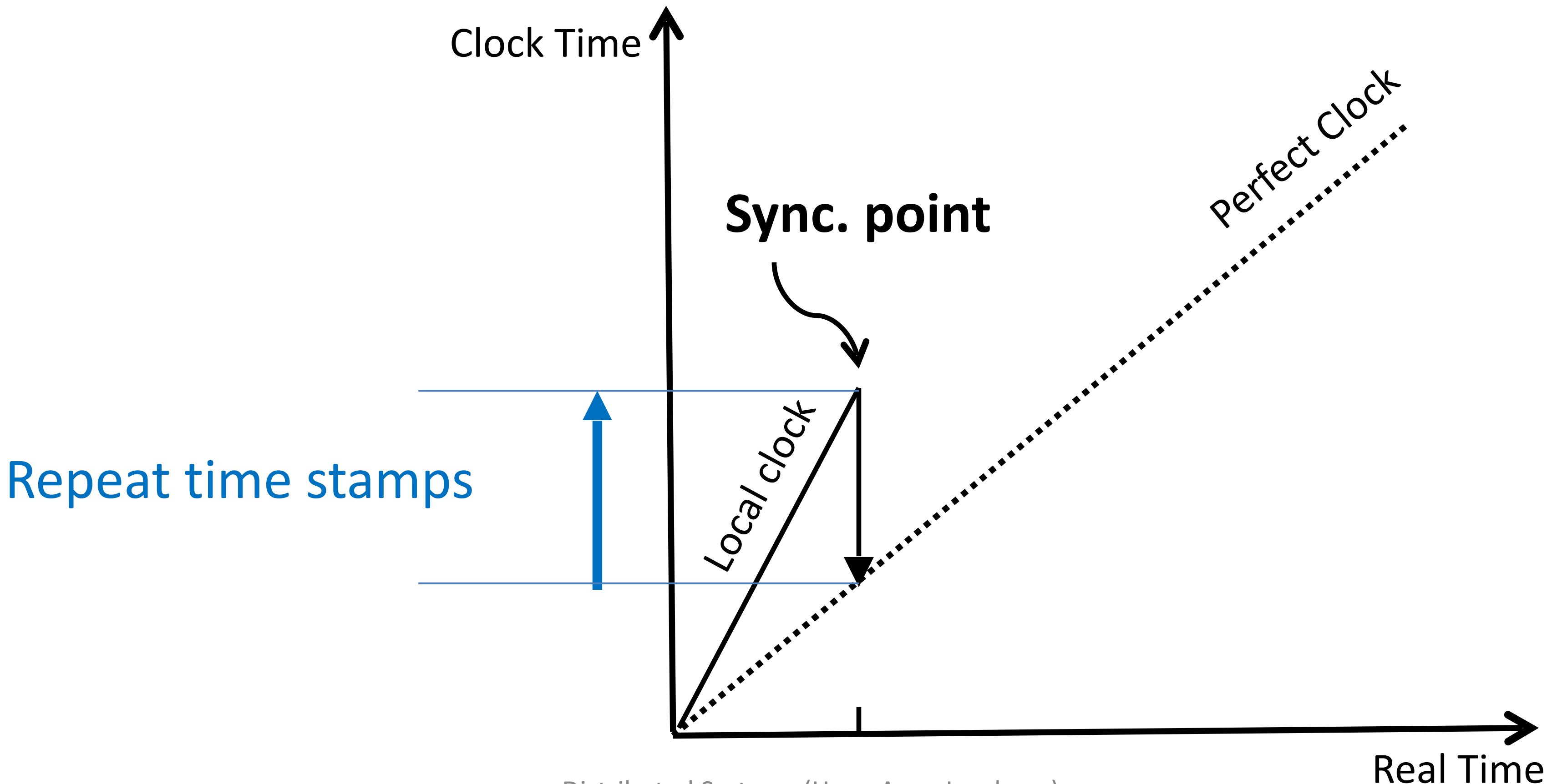
“Catch up” Example: Reset clock

- Can’t just set clock to be equal to real time



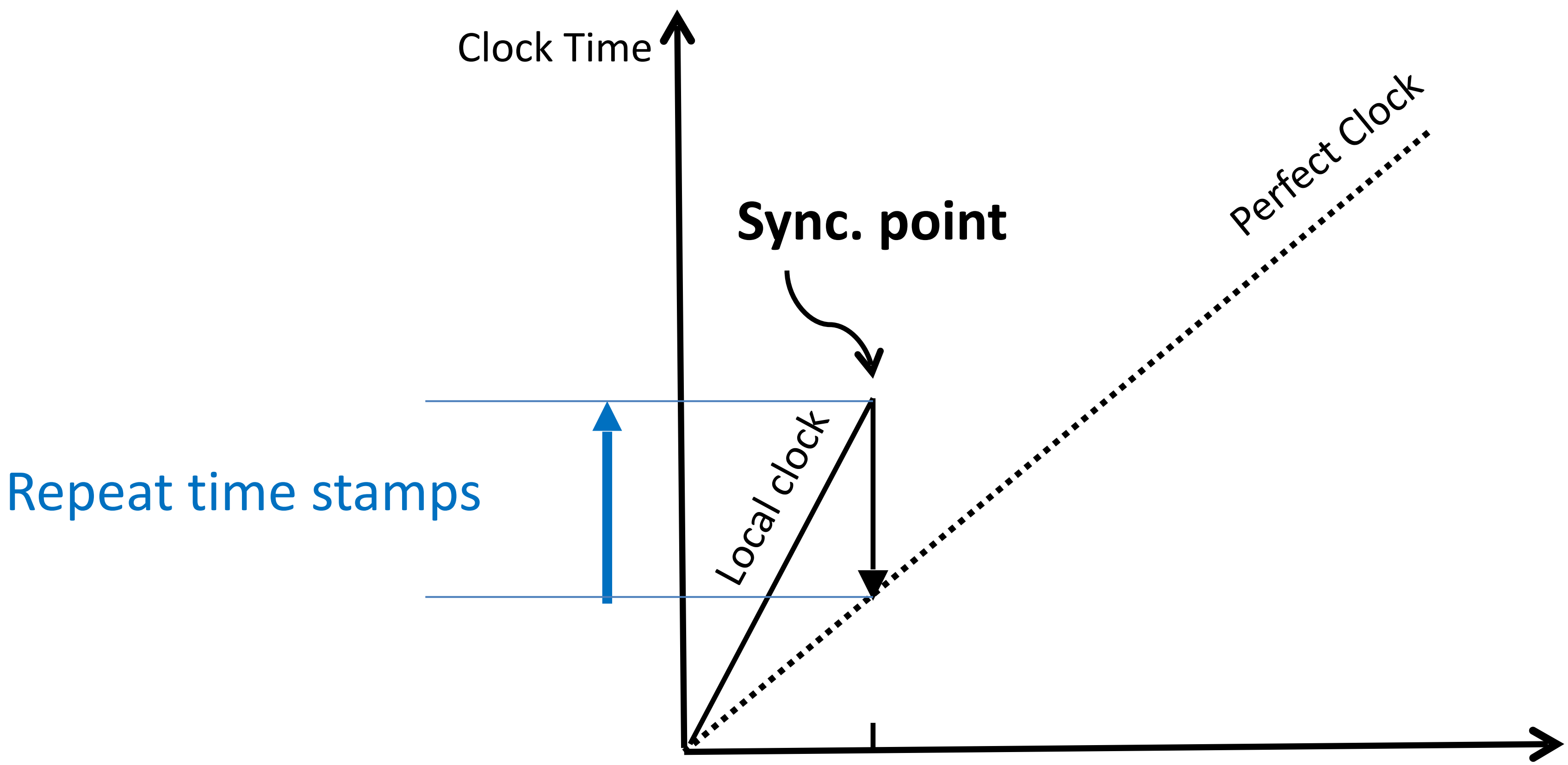
“Catch up” Example: Reset clock

- Can’t just set clock to be equal to real time



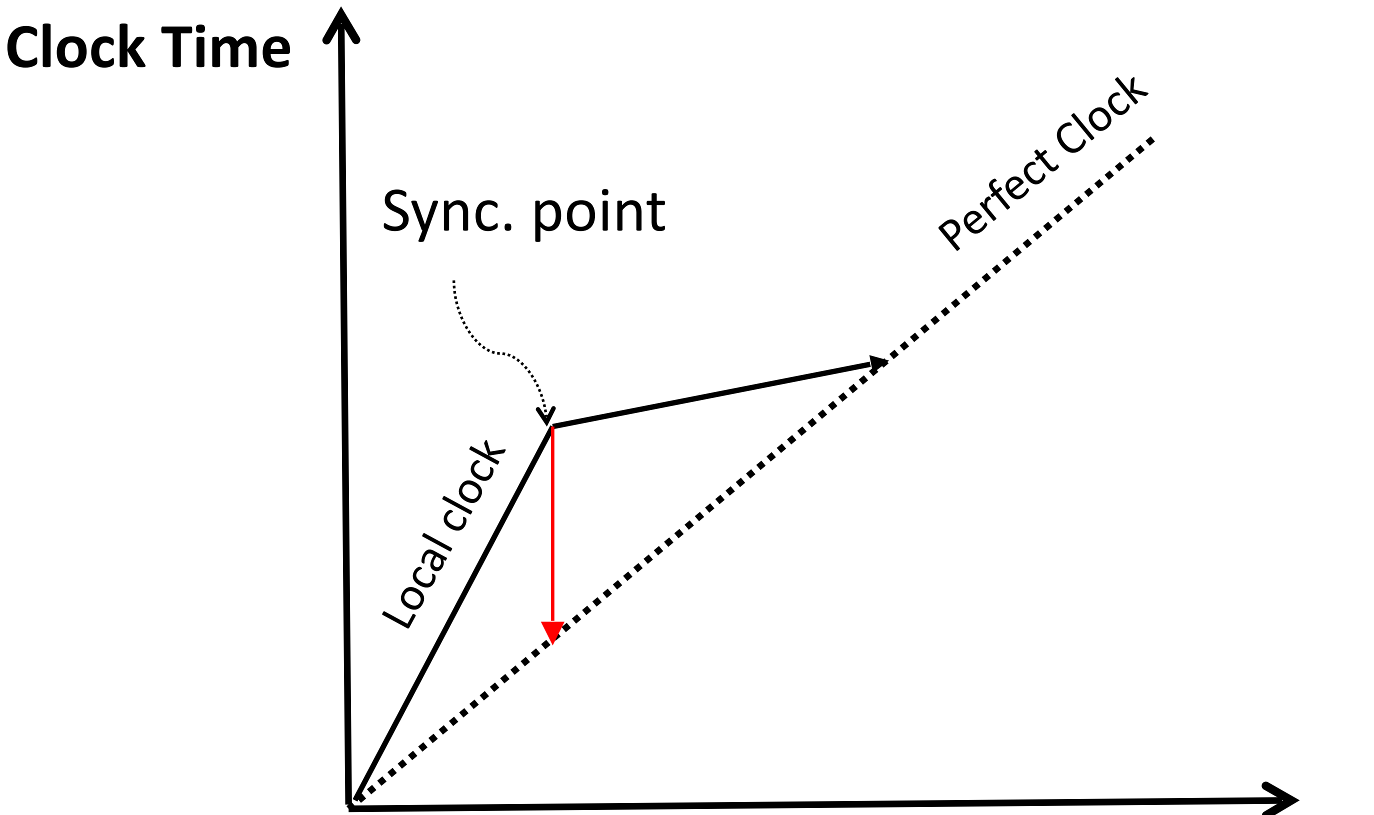
“Catch up” Example: ~~Reset Clock~~

- Can’t just set clock to be equal to real time



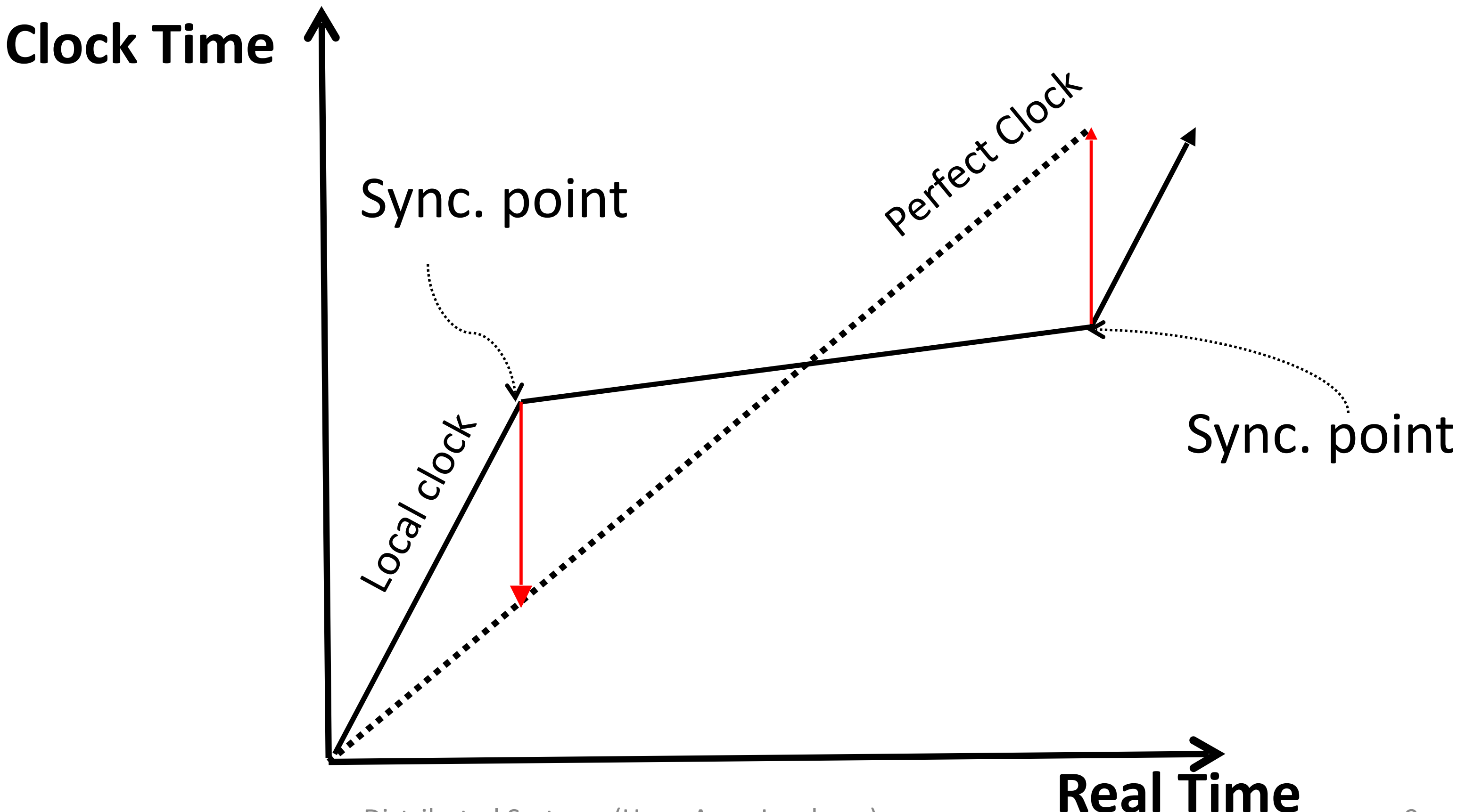
“Catch up” Example: Slow down clock

- Instead, **slow down** the clock by not updating the full amount at each clock interrupt



“Catch up” Example: Implications

- Imperfect timing of synchronization points and clock drift may lead to **saw tooth behaviour** (too slow, then too fast)



Summary

- Clock is a continuous function, cannot show sudden jumps (discontinuities)
- Clock must increase monotonically
- Either slow clock down or speed it up

Self-study questions

- Identify a few scenarios where resetting the clock would be detrimental.
- Identify a few scenarios where setting the clock forward would be detrimental.
- Lookup how to read the hardware clock on your computer from the command line.
- How are time zones and daylight savings time accounted for?

Operating Systems — R. Stockton Gaines, Editor

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

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system.

The Partial Ordering

Most people would probably say that an event a happened before an event b if a happened at an earlier time than b.

Communications of the ACM, July 1978, Volume 21, Number 7

LAMPORT CLOCK

(A.K.A. LOGICAL CLOCK, LINEAR CLOCK)

Events

- **Event** is an abstraction for anything we'd like to track (timestamp) in a (distributed) system
- E.g., event may represent instruction execution, method call, order entry, access request, exception…
- Events represent **message send** and **receive**

- Often sufficient to know **order of events** instead of events' exact physical time

Events within and across nodes

- Within a single node **event order** determined by execution sequence (e.g., relative to a physical clock)

- Between two different nodes **event order** cannot be determined using local physical clocks, since those **clocks cannot be perfectly synchronized**

- *How then are we to represent time?*

Logical clocks

- Key insight is to **abandon idea of physical time**
- Only care about **order of events**, **not *when*** *exactly* they happened (or ***how*** much time between events)
- Lamport introduced **logical time** and method to synchronize logical clocks in 1978

Leslie Lamport, "***Time, clocks, and the ordering of events in a distributed system***", Communications of the ACM, Vol. 27, No. 7, July 1978, pp. 558-565


Operating Systems — R. Stockton Gaines, Editor

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

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F 30602-76-C-0094.

Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025.

© 1978 ACM 0001-0782/78/0700-0558 $00.75

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

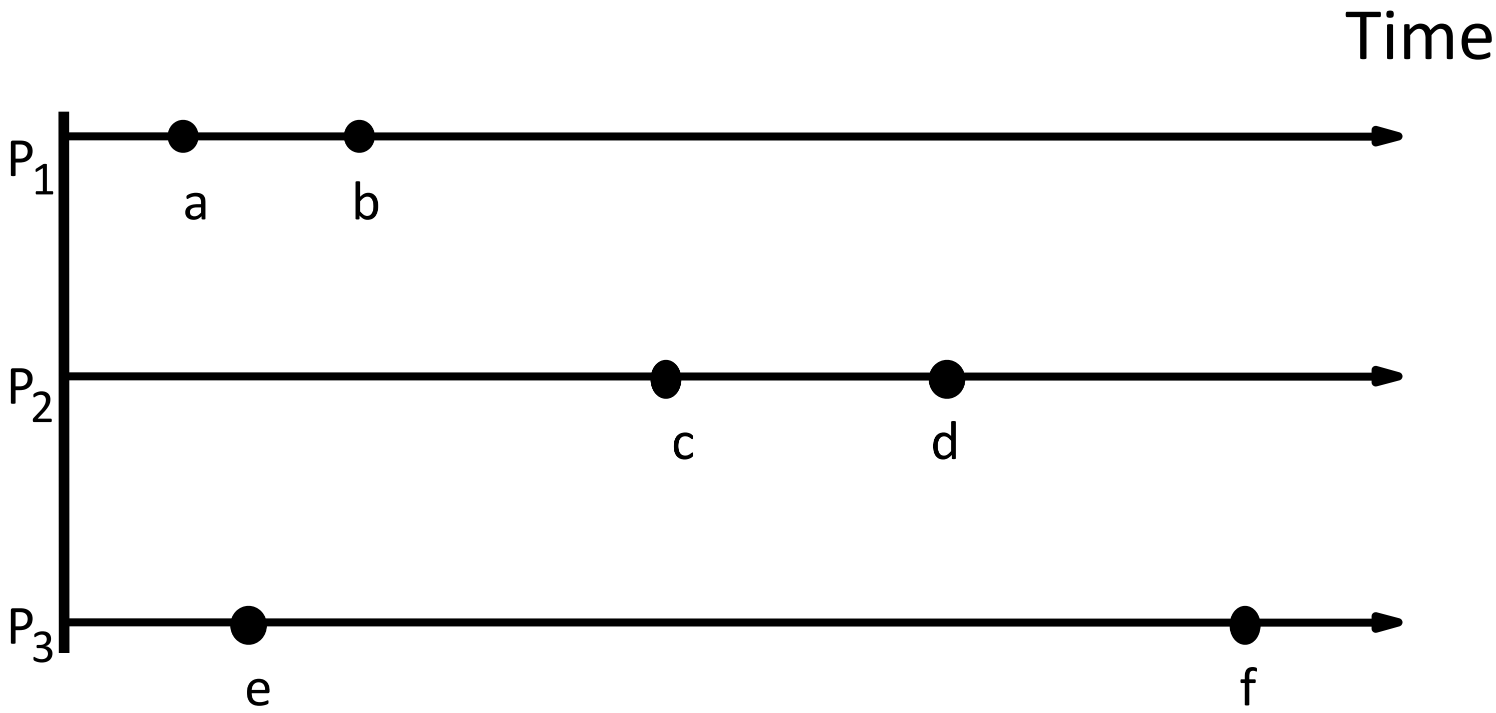
The Partial Ordering

Most people would probably say that an event *a* happened before an event *b* if *a* happened at an earlier time than *b*. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the "happened before" relation without using physical clocks.

We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events. Depending upon the application, the execution of a subprogram on a computer could be one event, or the execution of a single machine instruction could be one

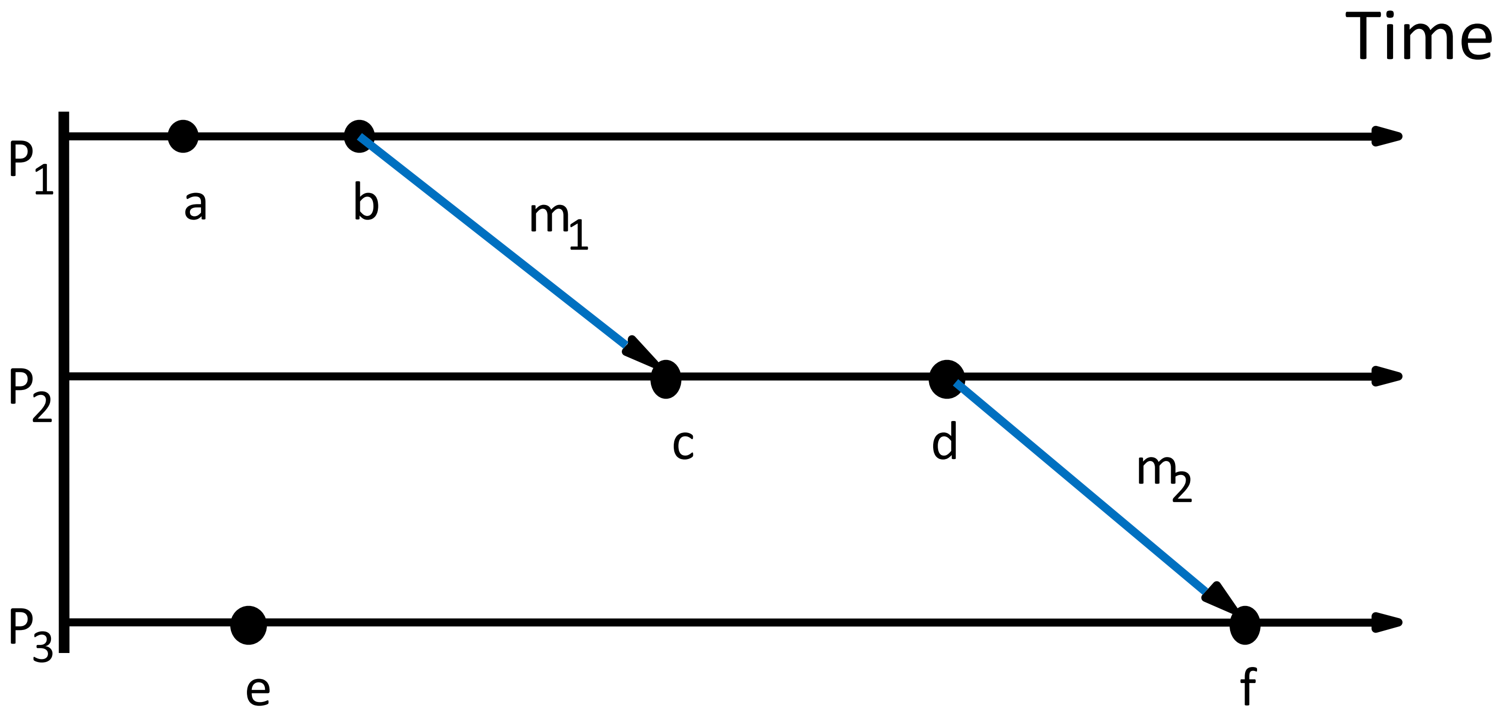
Events in a distributed system

Space-time diagram



Events in a distributed system

Space-time diagram



The happed-before relation

Denoted by "→"

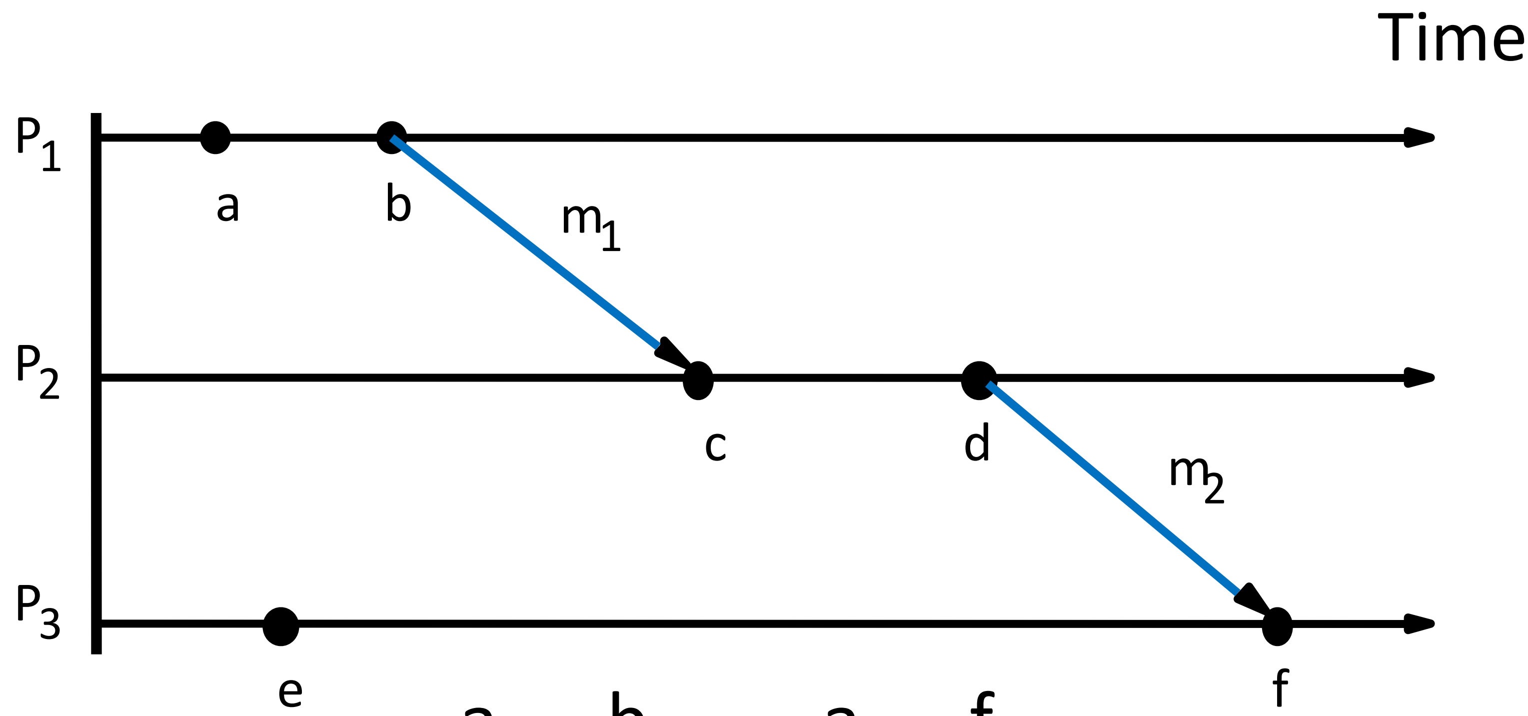
- Describes *causal* **order of events** in a system
- Definition "→" :
  - If ***a* and *b* are events** in the same node and ***a* occurred before *b*** then ***a* → *b***
  - If ***a*** is the event of **sending a message *m*** in one node and ***b*** is the event of **receiving *m*** in another node then ***a* → *b***
  - Relation "→" is **transitive:** If $a \rightarrow b$ and $b \rightarrow$ c then $a \rightarrow c$
- If neither $a \rightarrow b$ nor $b \rightarrow a$ then ***a* and *b* are concurrent,** denoted by ***a* || *b***
- For any two events *a* and *b,*

  either $a \rightarrow b$, $b \rightarrow a$ or $a \mid\mid b$

Causality of "→"-relation
(a.k.a. causality relation)

- Intuitively, past events influence future events
- Influence among causally related events is referred to as causal effects
- If $a \rightarrow b$, assume event *a* causally effects event *b*
- Concurrent events **do not causally effect** each other (e.g., neither $a \rightarrow b$ nor $b \rightarrow a$)

Example: Happened-before relation

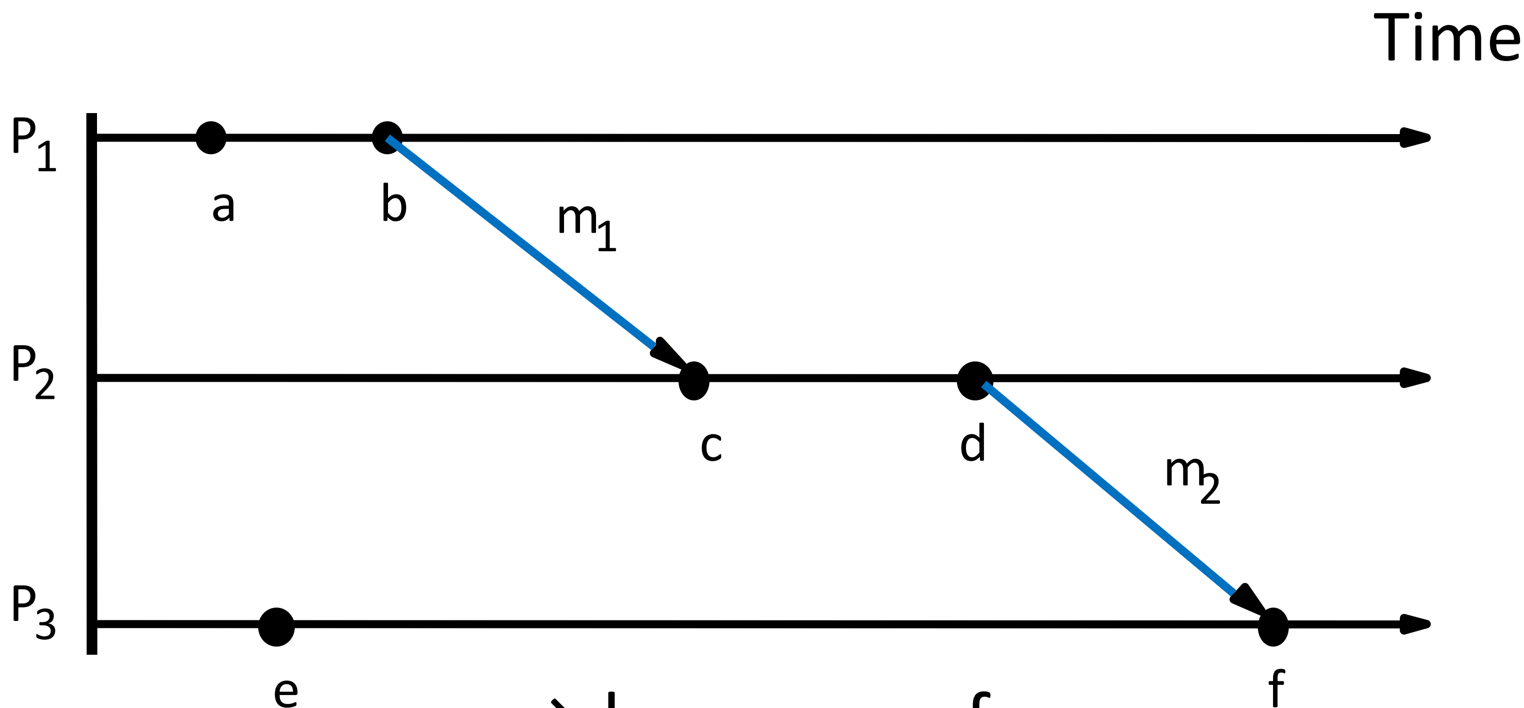


a b a f

b c e f

c d b e

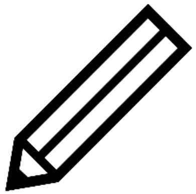
Example: Happened-before relation



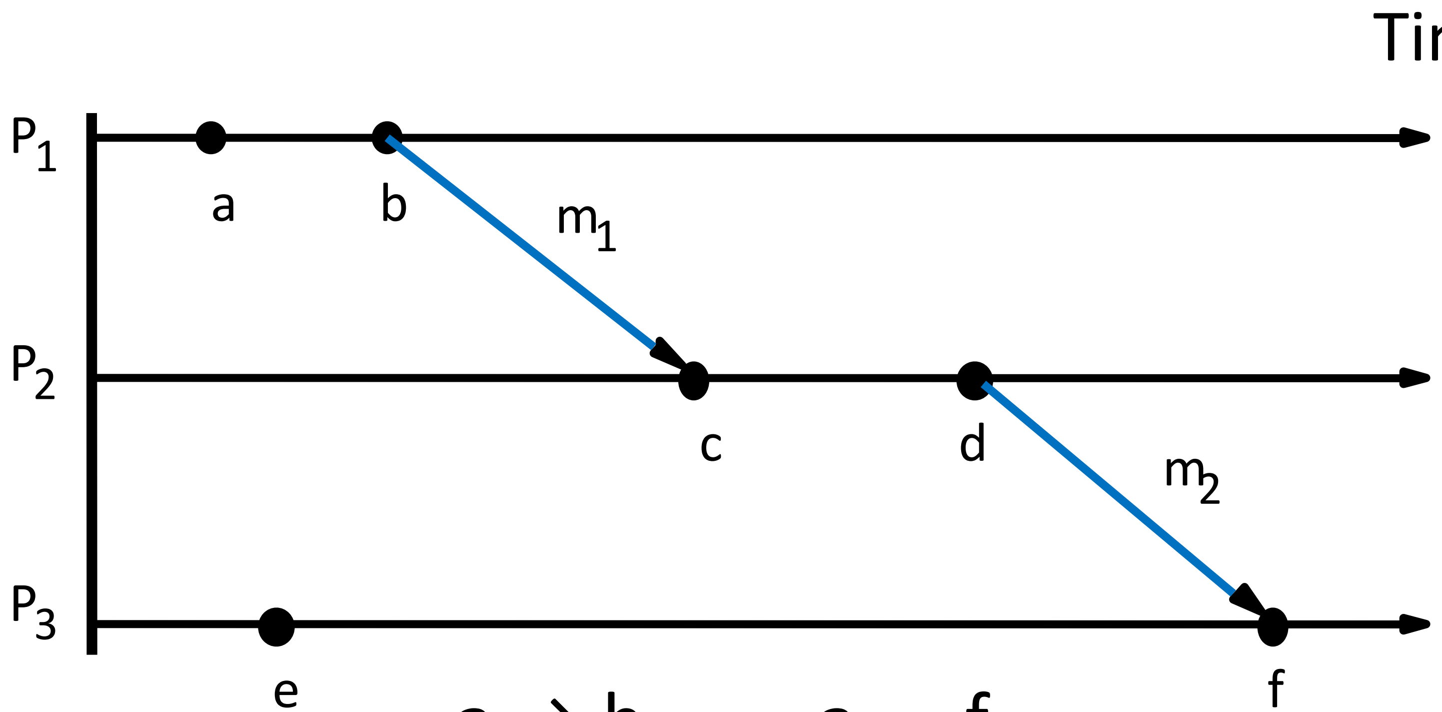
a → b   a   f

b   c   e   f

c   d   b   e



Example: Happened-before relation

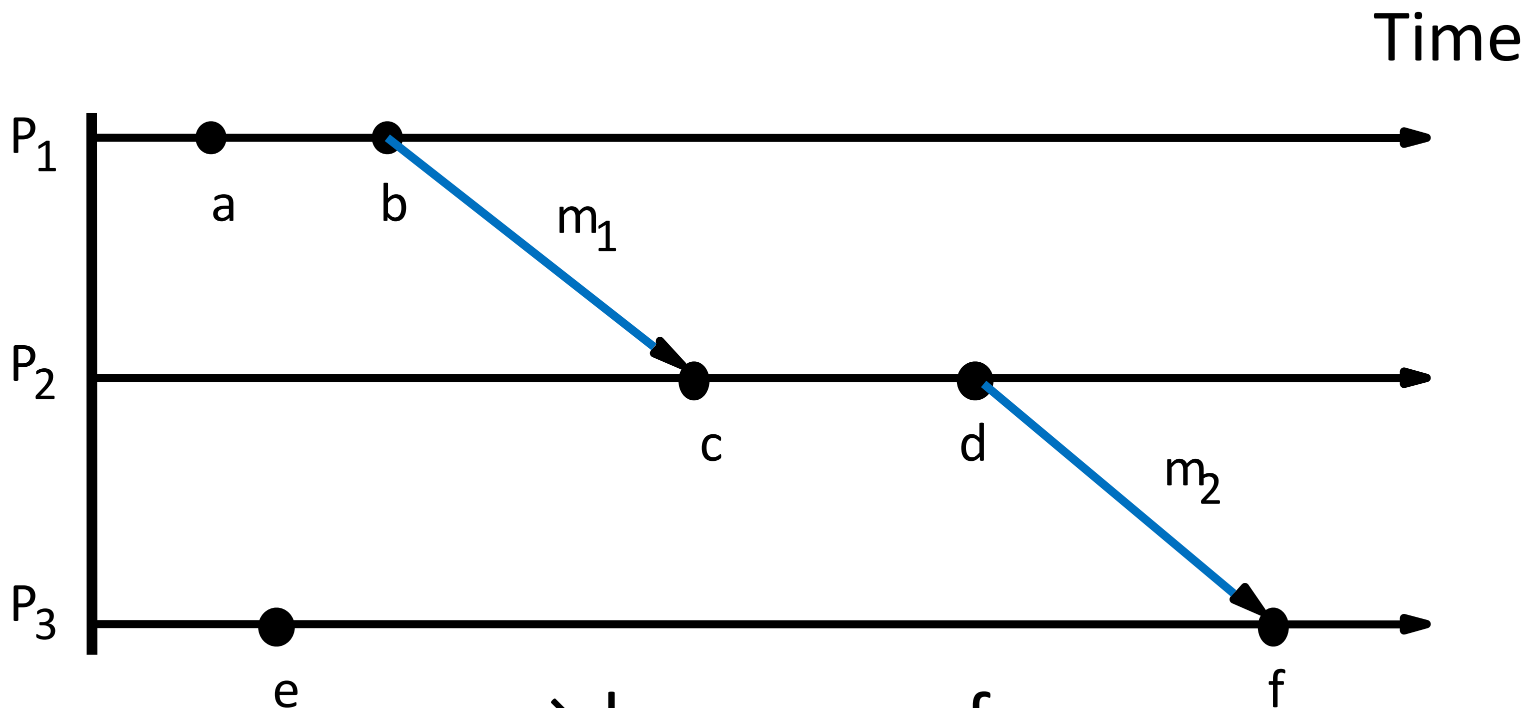


a → b   a   f

b → c   e   f

c   d   b   e

Example: Happened-before relation

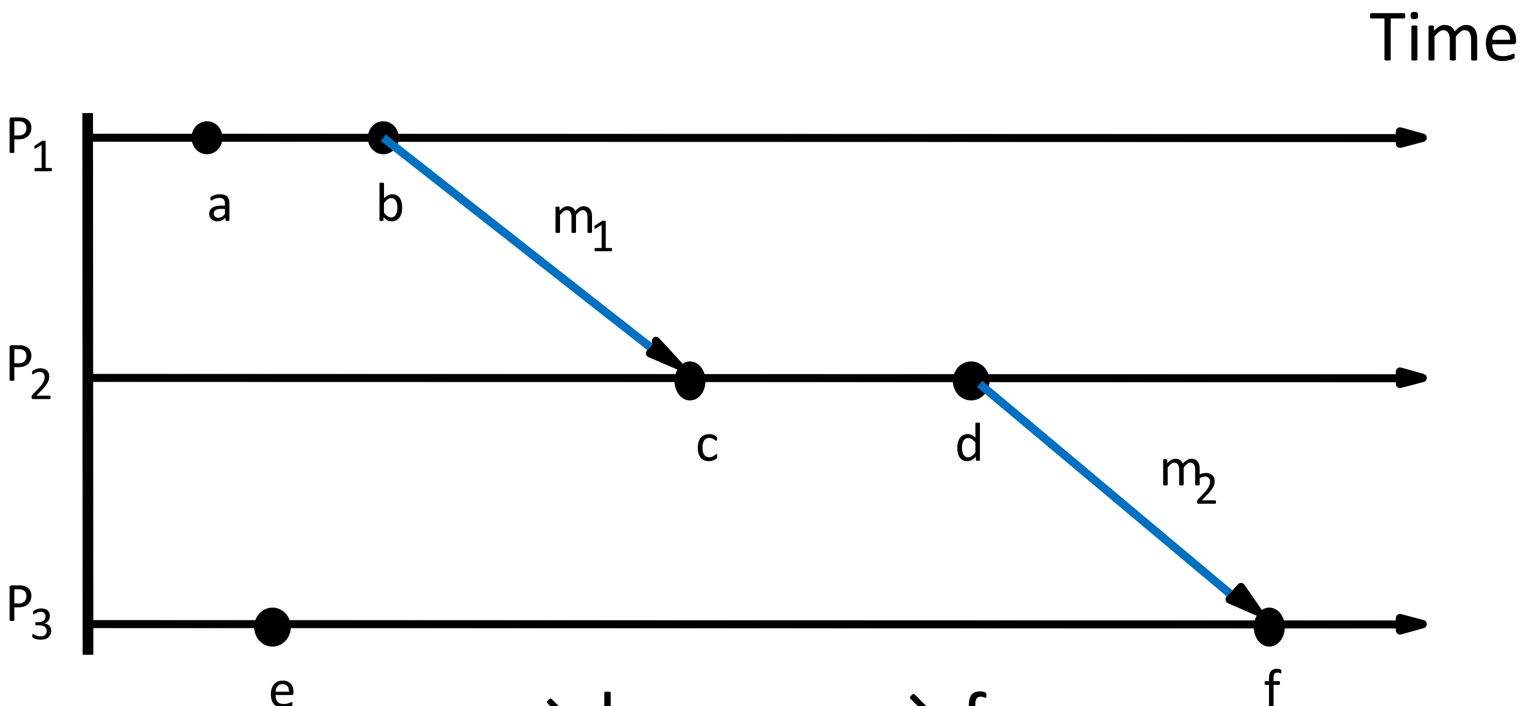


$a \rightarrow b$ $\quad$ a $\quad$ f

$b \rightarrow c$ $\quad$ e $\quad$ f

$c \rightarrow d$ $\quad$ b $\quad$ e

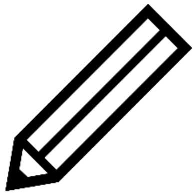
Example: Happened-before relation



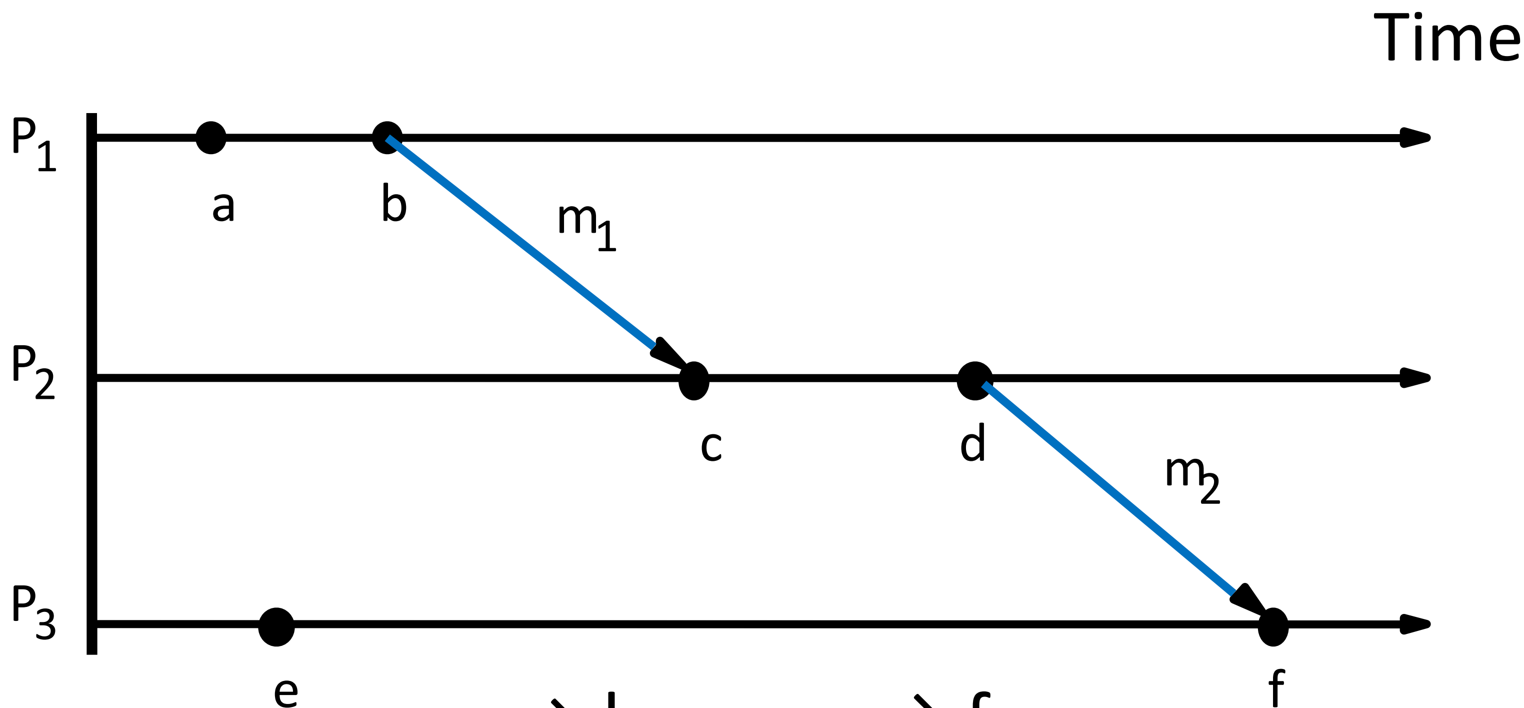
$a \rightarrow b$ $a \rightarrow f$

$b \rightarrow c$ e f

$c \rightarrow d$ b e



Example: Happened-before relation

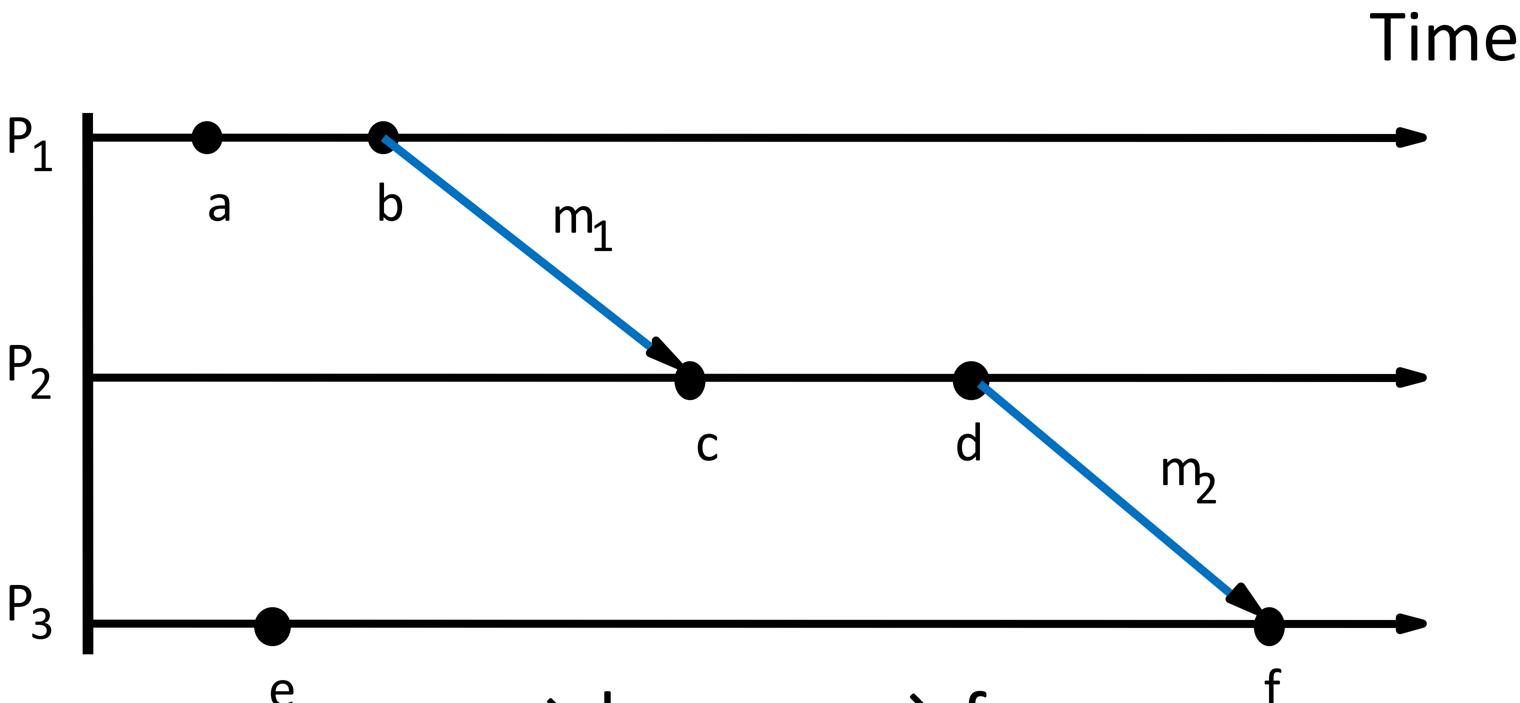


$a \rightarrow b$ $\quad$ $a \rightarrow f$

$b \rightarrow c$ $\quad$ $e \rightarrow f$

$c \rightarrow d$ $\quad$ b $\quad$ e

Example: Happened-before relation



$a \rightarrow b$ $\quad$ $a \rightarrow f$

$b \rightarrow c$ $\quad$ $e \rightarrow f$

$c \rightarrow d$ $\quad$ $b \,||\, e$

Lamport clock (logical clock)

- Clock that **tracks "→" numerically**
- Each node P_i has a **logical clock C_i** (a local variable)
- C_i assigns a **value $C_i(a)$** to any event *a* in P_i
- Value $C_i(a)$ is the **timestamp of event *a*** at node P_i
- Timestamps have **no relation to physical time**, which leads to the term **logical clock**
- Logical clocks **assign monotonically increasing timestamps**
- Can be implemented by an **integer counter**

Clock conditions

- Clock condition
  - **If a → b then C(a) < C(b)**

Clock conditions

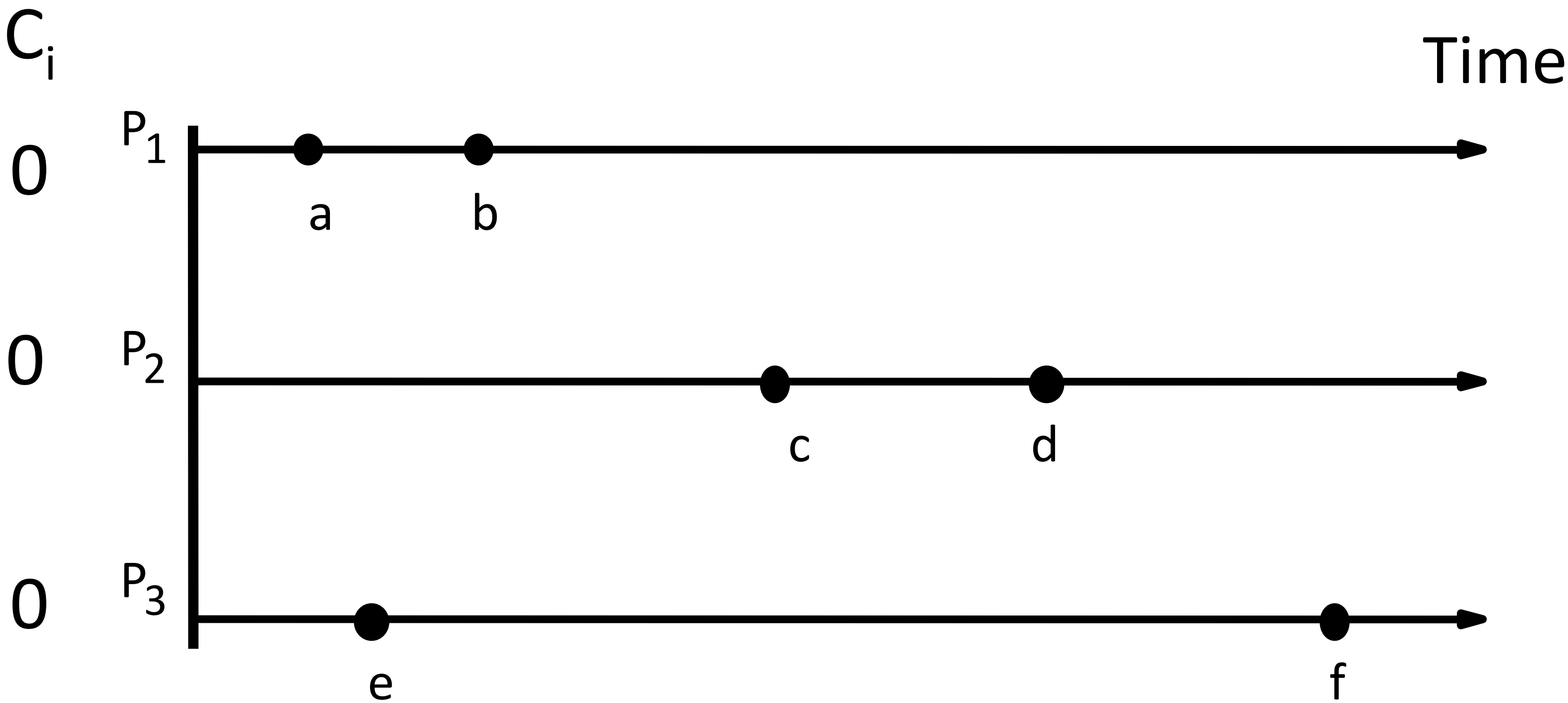
- Clock condition
  - **If a $\rightarrow$ b then C(a) < C(b)**

- Clock conditions
  - For any two events *a* and *b* at the same node P_i, **if** ***a*** **$\rightarrow$** ***b*** **then $C_i(a) < C_i(b)$**
  - If *a* is the event of **sending a message** at node P_i and *b* is the event of **receiving that same message** at a different node P_k then **$C_i(a) < C_k(b)$**

Logical clock implementation

- Clock C_i is **incremented** before event occurs at node P_i (before event is executed)
  - $\mathbf{C_i = C_i + d}$ ($d > 0$, e.g., $d = 1$)
- If *a* is event of **sending message m** at node P_i then **m is assigned a timestamp**
  - $T_m = C_i(a)$
- When **that same message m is received** by a different process P_k, $\mathbf{C_k}$ **is set to a value greater than its present value** (prior to message receipt) and greater than T_m
  - $\mathbf{C_k = max\{C_k , T_m\} + d}$ ($d > 0$, e.g., $d = 1$)

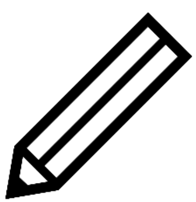
Example: Logical clock



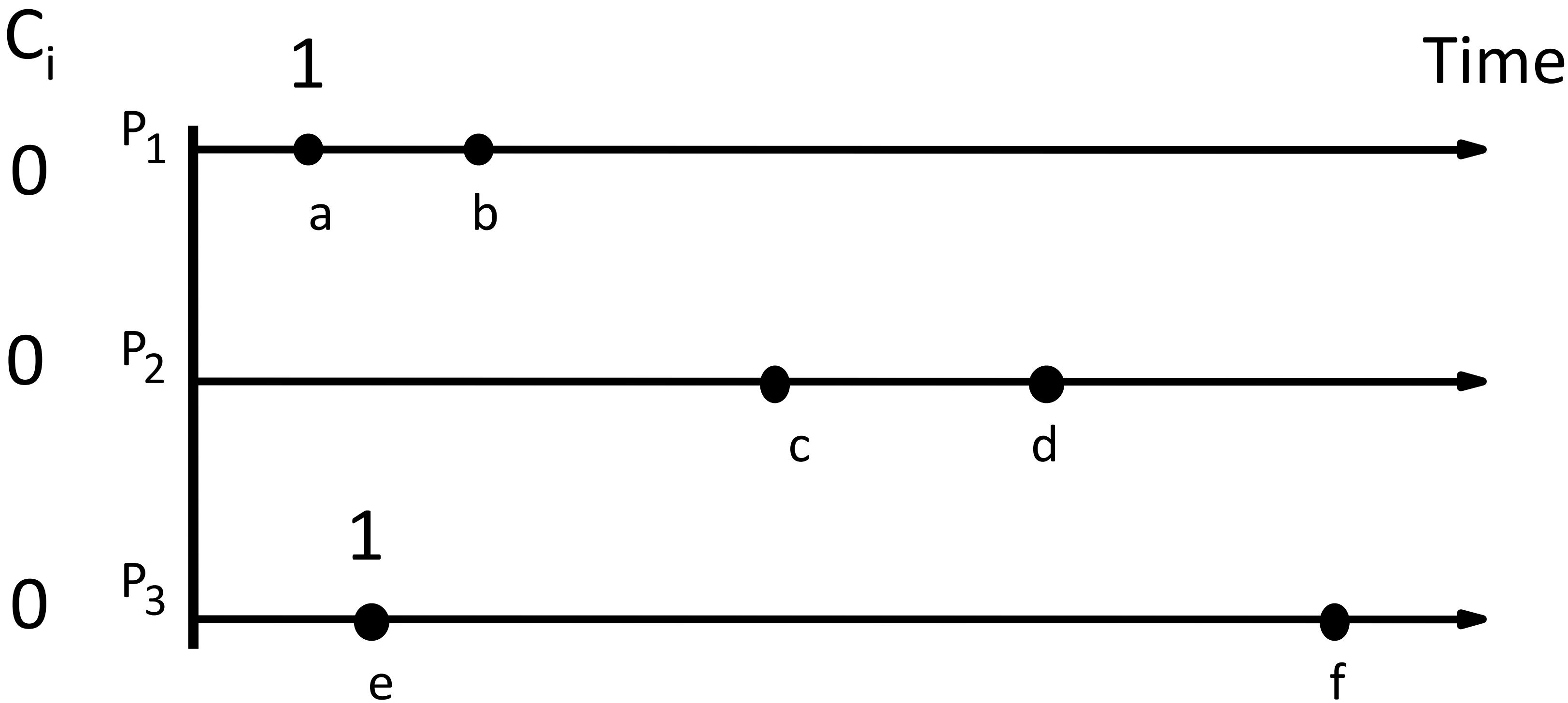
$a \rightarrow b$ $\quad$ $a \rightarrow f$

$b \rightarrow c$ $\quad$ $e \rightarrow f$

$c \rightarrow d$



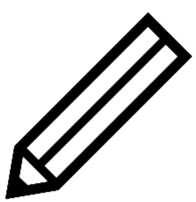
Example: Logical clock



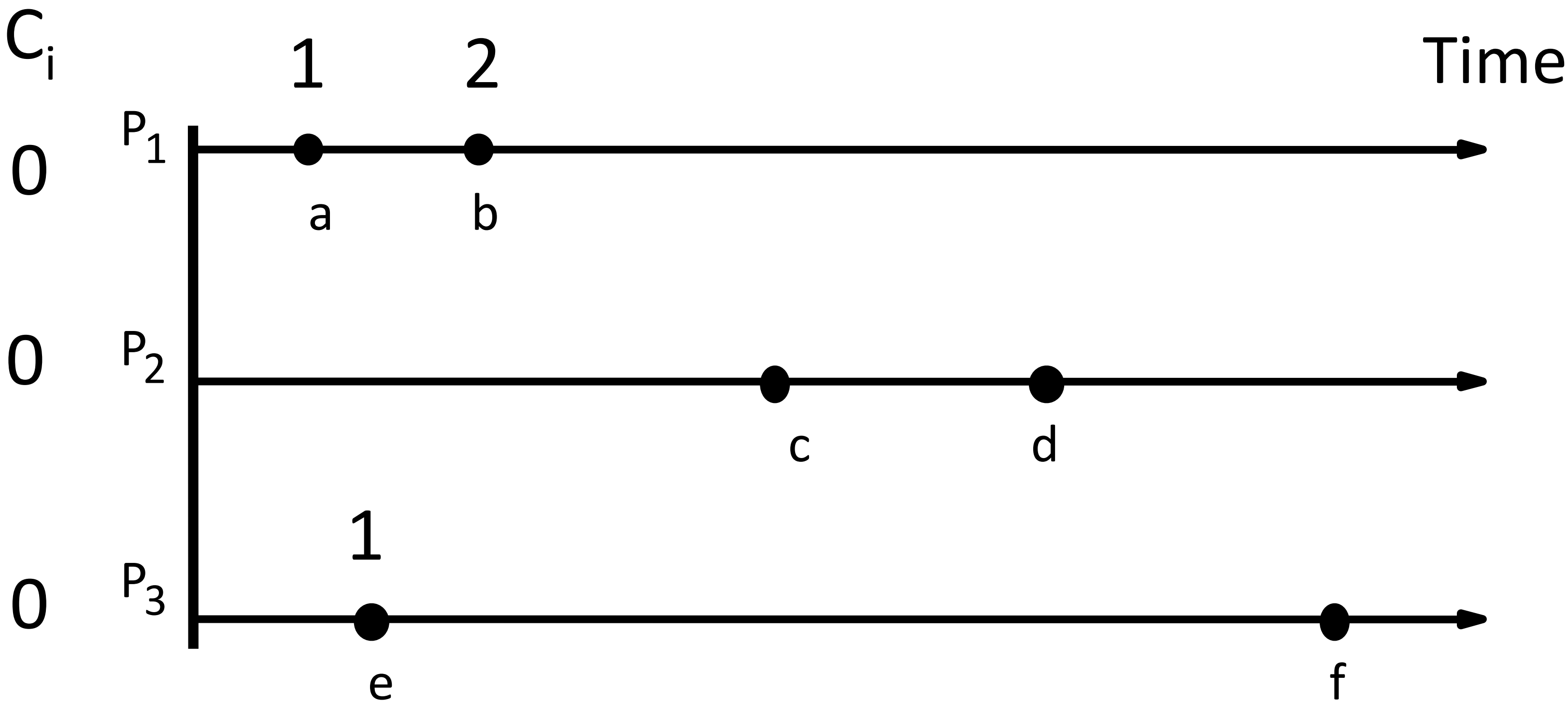
$a \rightarrow b$ $a \rightarrow f$

$b \rightarrow c$ $e \rightarrow f$

$c \rightarrow d$



Example: Logical clock

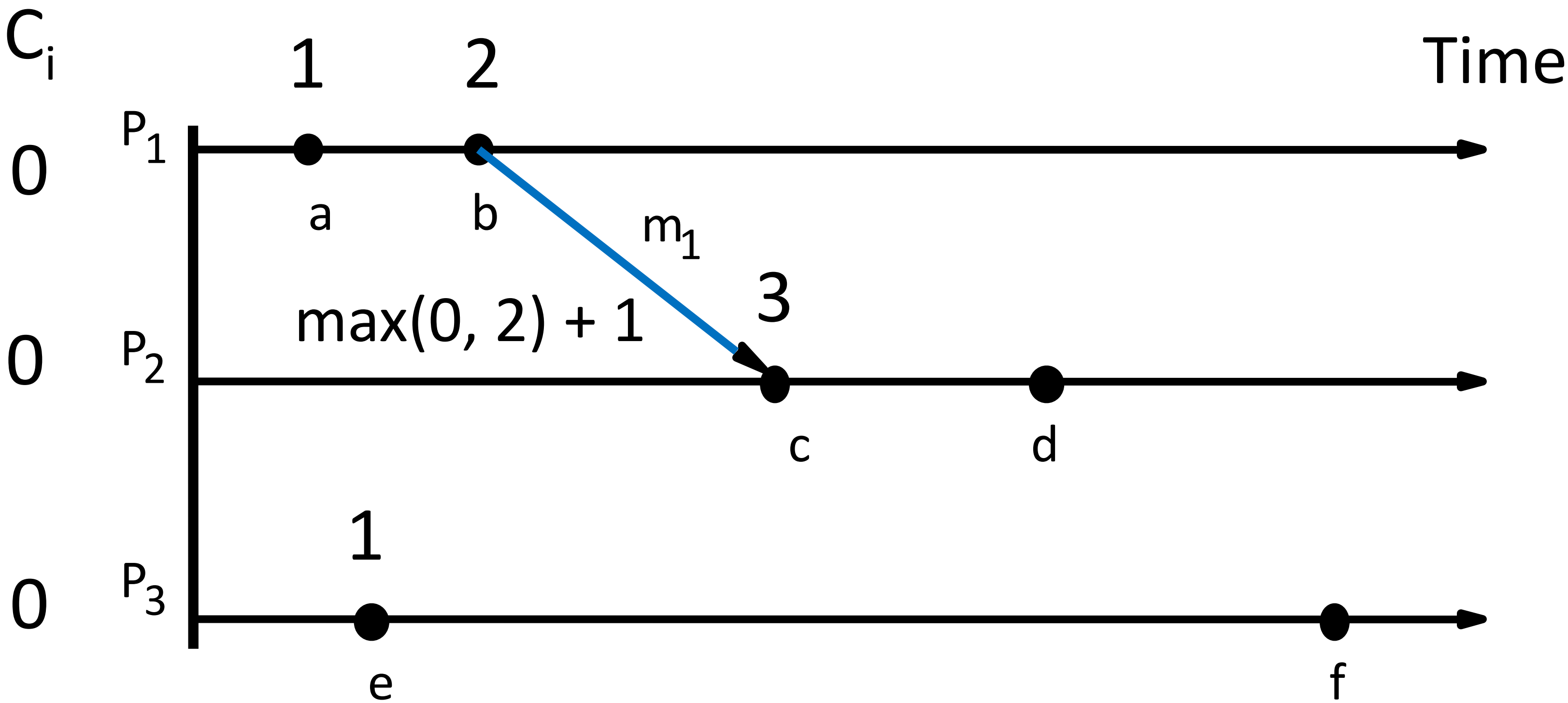


a → b   a → f

b → c   e → f

c → d

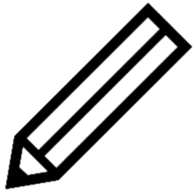
Example: Logical clock



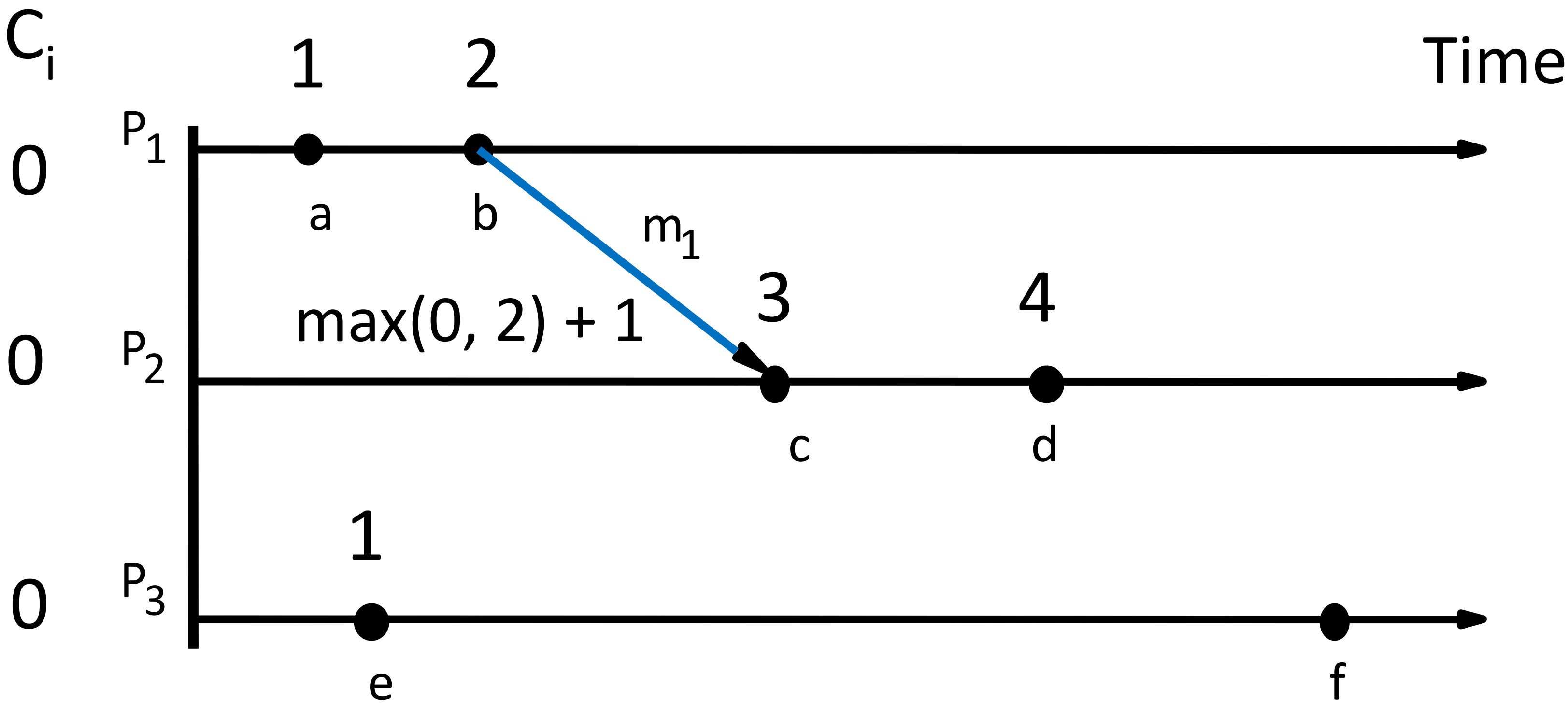
$a \rightarrow b$ $a \rightarrow f$

$b \rightarrow c$ $e \rightarrow f$

$c \rightarrow d$



Example: Logical clock

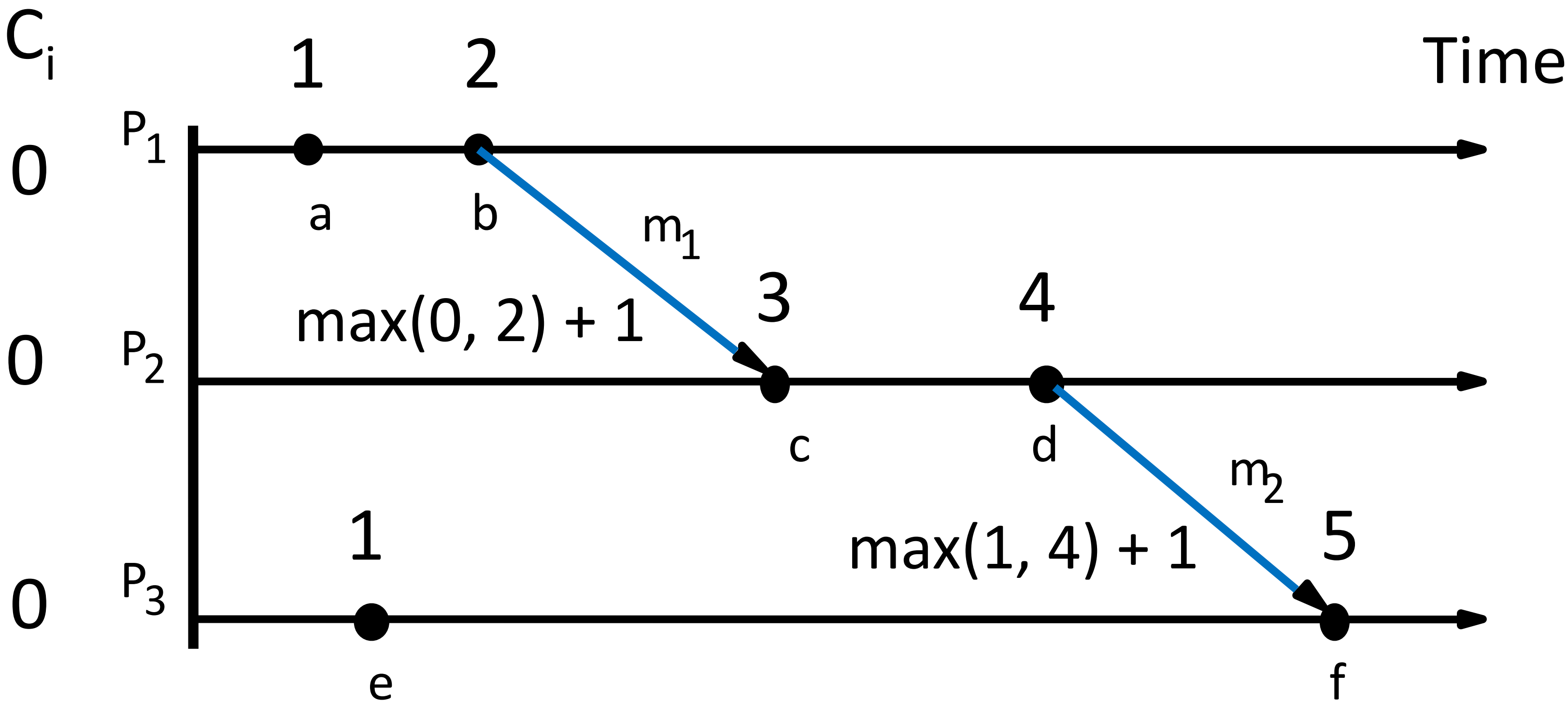


$a \rightarrow b$ $\qquad$ $a \rightarrow f$

$b \rightarrow c$ $\qquad$ $e \rightarrow f$

$c \rightarrow d$

Example: Logical clock

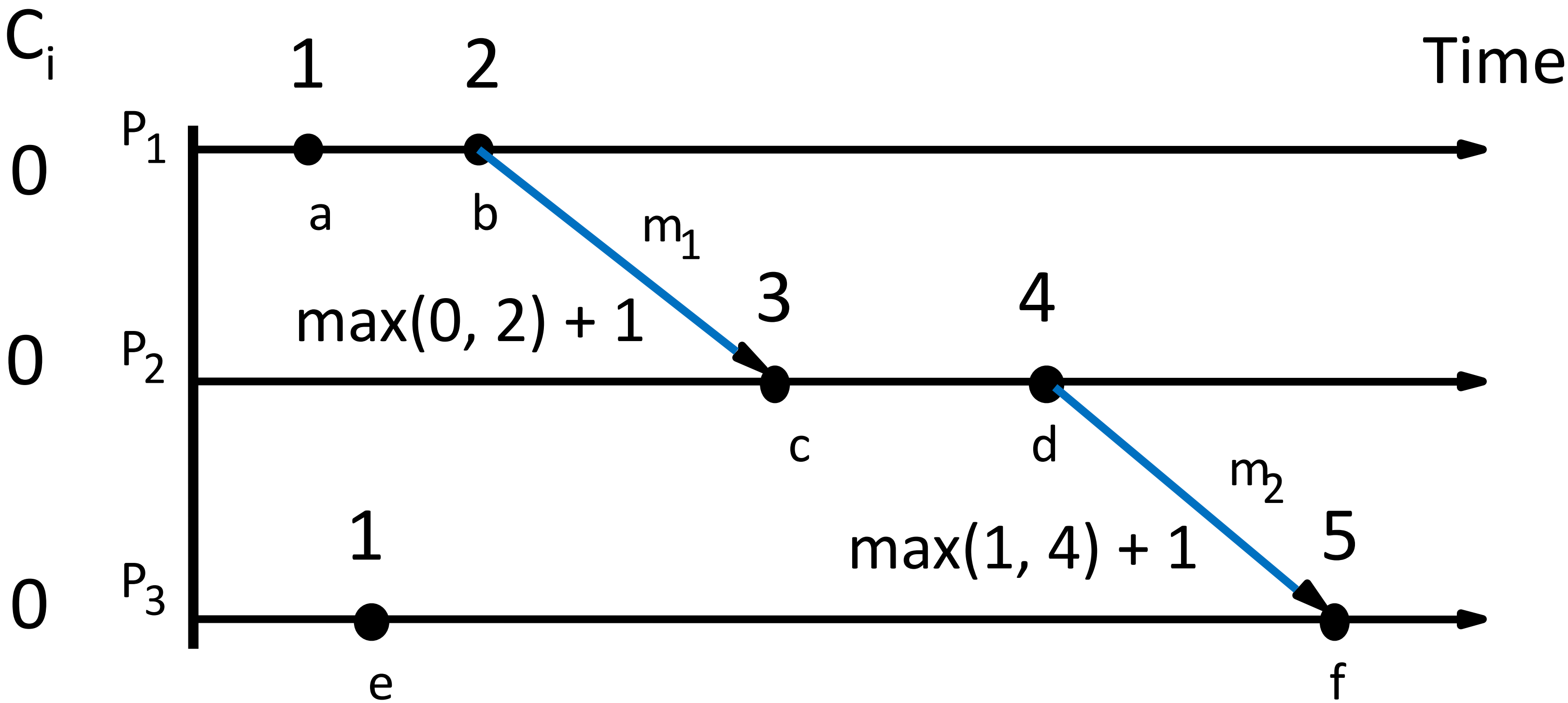


a → b   a → f

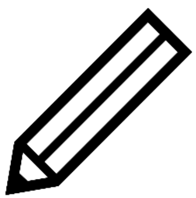
b → c   e → f

c → d

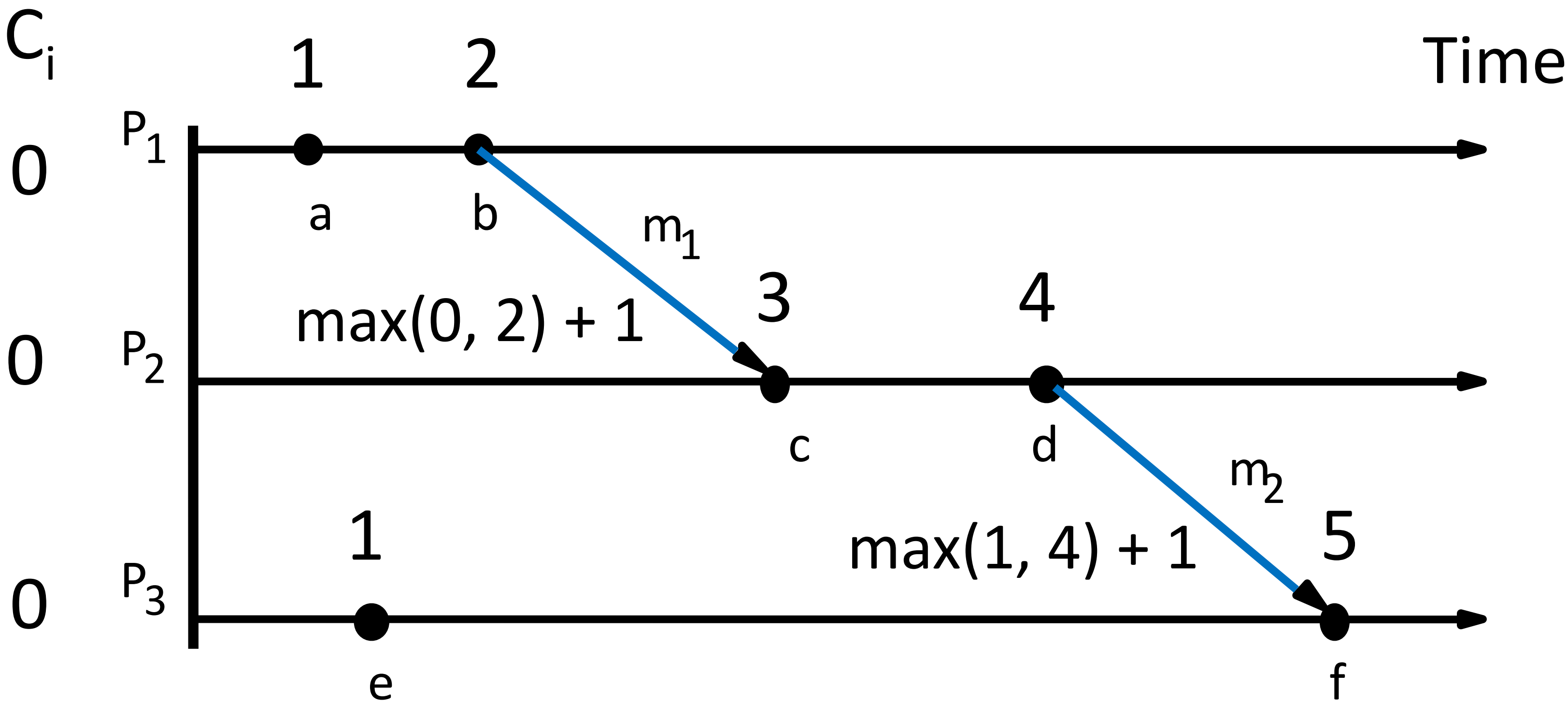
Example: Logical clock



$a \rightarrow b$	$a \rightarrow f$
$b \rightarrow c$	$e \rightarrow f$
$c \rightarrow d$	$a \parallel e$



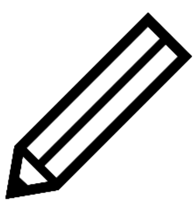
Example: Logical clock



$a \rightarrow b$ $\quad$ $a \rightarrow f$

$b \rightarrow c$ $\quad$ $e \rightarrow f$

$c \rightarrow d$ $\quad$ $a \,||\, e$ $\quad$ $b \,||\, e$



Clock condition

- Clock condition
  - **If a → b then C(a) < C(b)**

- Correctness conditions
  - For any two events *a* and *b* at the same node P_i, **if *a* → *b* then $C_i(a) < C_i(b)$**
  - If *a* is the event of **sending a message** at node P_i and *b* is the event of **receiving that same message** at a different node P_j then **$C_i(a) < C_j(b)$**

Clock condition

- Clock condition
  - **If a → b then C(a) < C(b)**

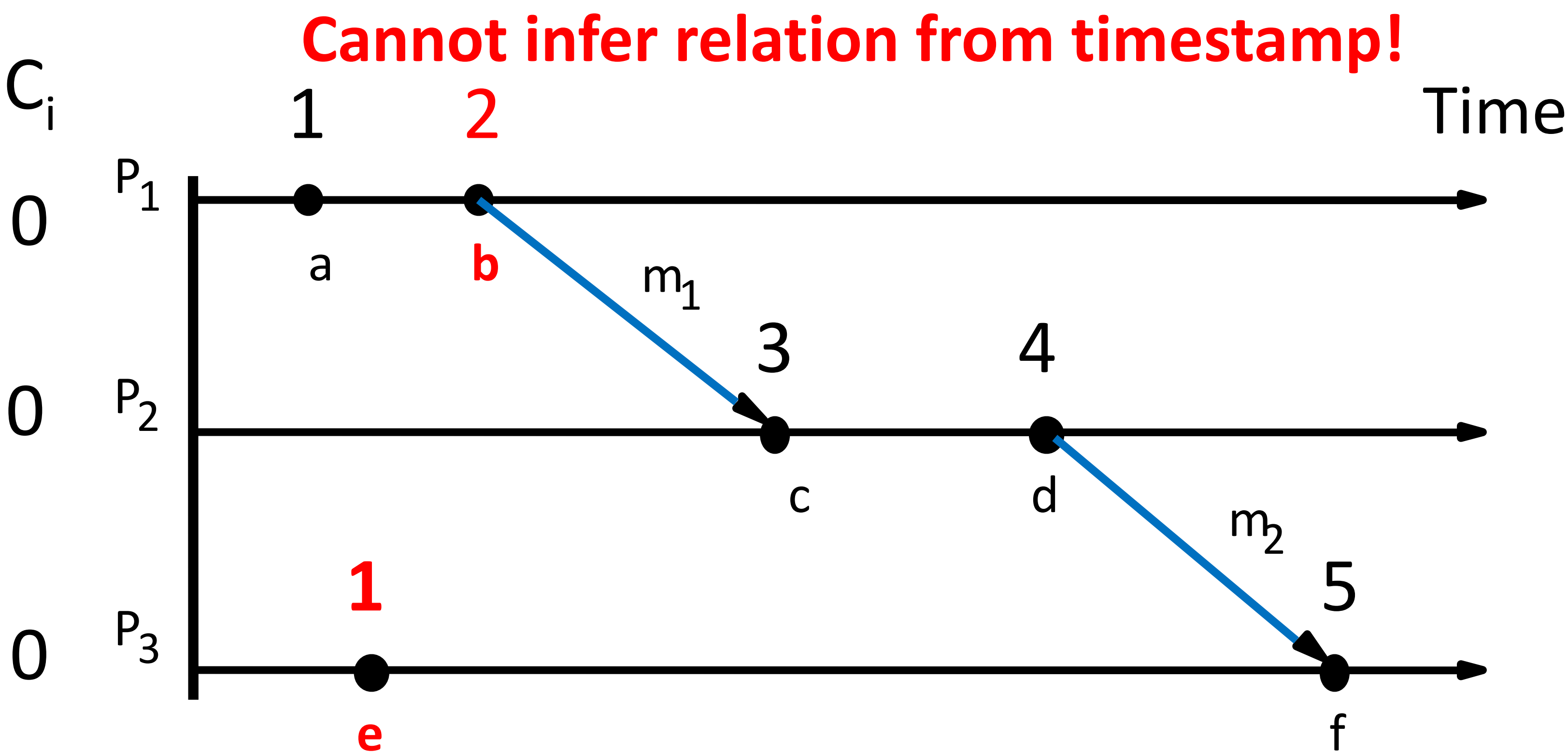
But not:

If C(e) < C(b)

then e → b

- Correctness conditions
  - For any two events *a* and *b* at the same node P_i, **if** ***a*** **→** ***b*** **then** $C_i(a) < C_i(b)$
  - If *a* is the event of **sending a message** at node P_i and *b* is the event of **receiving that same message** at a different node P_j then $C_i(a) < C_j(b)$

Example: Logical clock



a → b　　a → f

b → c　　e → f

c → d　　b || e - **C(b) > C(e) !**

"→" is a unique partial order of events

- For $C_1(a) = 1$ and $C_3(e) = 1$, event *a, e* cannot be ordered according to the happened-before relation

- Happened-before relation is a **unique partial order** of events in system

- *a*, *e* are concurrent events

Induced total order

- $C_1(a) = 1$ and $C_3(e) = 1$ can't be ordered according to happened-before relation!
- Happened-before relation is a **unique partial order** of events
- Induce a **non-unique total order** as follows
  - Use logical time stamps to order events
  - Break ties by using an **arbitrary total ordering of nodes**, e.g., $P_1 < P_2$ (numerically compare node identifiers)
  - Thus, timestamp is comprised of logical clock value and identifier, i.e., $(C_i(a), i)$

Total order of events in system

Denoted by "⇒"

- Timestamps are $(C_i(a), i)$ and $(C_k(b), k)$

- Let ***a*** be an event at $\boldsymbol{P_i}$ and ***b*** an event at $\boldsymbol{P_k}$ *then* ***a*** ⇒ **b** if either
  - (i) $\mathbf{C_i(a) < C_k(b)}$ or
  - (ii) $\mathbf{C_i(a) = C_k(b)}$ and $\mathbf{P_i < P_k}$ **(e.g., $\mathbf{i < k}$)**

- Results in **total order of all events** in system

Potential causality of "→"-relation

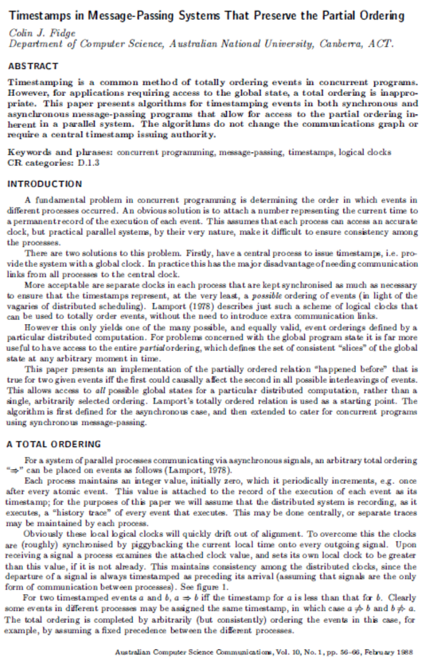
- "→" captures potential flow of information between events
- In $a \rightarrow b$, a might or might not have caused b (relation assumes it has, but we don't know for sure)
- Information may have flown in system in ways other than via message passing (not modeled by "→")

Summary

- Do away with physical time, focus on **order of events**
- **Happened-before relation** as unique partial order of events in system
- Relation tracks **potential causality** of events in system
- Can induce a **non-unique total order** by agreeing on an arbitrary order of nodes
- Implement happened-before relation with **integer variable** (essentially a counter) at each node and rules for updating it
- You cannot determine whether two events are causally related from timestamps alone*!*

Self-study questions

- Can all events in a distributed system be ordered?
- What is the difference between a partial order and a total order?
- Why is it important to totally order events?
- If event timestamps are equal, does it always follow that the associated events occurred at different nodes?
- If event timestamps are equal, does it always follow that the associated events are concurrent?
- If clocks are initialized to zero at the beginning of time and *d* is always one, what conclusions can we draw from looking at timestamps?
- What are applications of Lamport clocks?


Timestamps in Message-Passing Systems That Preserve the Partial Ordering

Colin J. Fidge
Department of Computer Science, Australian National University, Canberra, ACT.

ABSTRACT

Timestamping is a common method of totally ordering events in concurrent programs. However, for applications requiring access to the global state, a total ordering is inappropriate. This paper presents algorithms for timestamping events in both synchronous and asynchronous message-passing programs that allow for access to the partial ordering inherent in a parallel system. The algorithms do not change the communications graph or require a central timestamp issuing authority.

Keywords and phrases: concurrent programming, message-passing, timestamps, logical clocks
CR categories: D.1.3

INTRODUCTION

A fundamental problem in concurrent programming is determining the order in which events in different processes occurred. An obvious solution is to attach a number representing the current time to a permanent record of the execution of each event. This assumes that each process can access an accurate clock, but practical parallel systems, by their very nature, make it difficult to ensure consistency among the processes.

There are two solutions to this problem. Firstly, have a central process to issue timestamps, i.e. provide the system with a global clock. In practice this has the major disadvantage of needing communication links from all processes to the central clock.

More acceptable are separate clocks in each process that are kept synchronised as much as necessary to ensure that the timestamps represent, at the very least, a *possible* ordering of events (in light of the vagaries of distributed scheduling). Lamport (1978) describes just such a scheme of logical clocks that can be used to totally order events, without the need to introduce extra communication links.

However this only yields one of the many possible, and equally valid, event orderings defined by a particular distributed computation. For problems concerned with the global program state it is far more useful to have access to the entire *partial* ordering, which defines the set of consistent "slices" of the global state at any arbitrary moment in time.

This paper presents an implementation of the partially ordered relation "happened before" that is true for two given events iff the first could causally affect the second in all possible interleavings of events. This allows access to *all* possible global states for a particular distributed computation, rather than a single, arbitrarily selected ordering. Lamport's totally ordered relation is used as a starting point. The algorithm is first defined for the asynchronous case, and then extended to cater for concurrent programs using synchronous message-passing.

A TOTAL ORDERING

For a system of parallel processes communicating via asynchronous signals, an arbitrary total ordering "$\Rightarrow$" can be placed on events as follows (Lamport, 1978).

Each process maintains an integer value, initially zero, which it periodically increments, e.g. once after every atomic event. This value is attached to the record of the execution of each event as its timestamp; for the purposes of this paper we will assume that the distributed system is recording, as it executes, a "history trace" of every event that executes. This may be done centrally, or separate traces may be maintained by each process.

Obviously these local logical clocks will quickly drift out of alignment. To overcome this the clocks are (roughly) synchronised by piggybacking the current local time onto every outgoing signal. Upon receiving a signal a process examines the attached clock value, and sets its own local clock to be greater than this value, if it is not already. This maintains consistency among the distributed clocks, since the departure of a signal is always timestamped as preceding its arrival (assuming that signals are the only form of communication between processes). See figure 1.

For two timestamped events a and b, $a \Rightarrow b$ iff the timestamp for a is less than that for b. Clearly some events in different processes may be assigned the same timestamp, in which case $a \not\Rightarrow b$ and $b \not\Rightarrow a$. The total ordering is completed by arbitrarily (but consistently) ordering the events in this case, for example, by assuming a fixed precedence between the different processes.

Australian Computer Science Communications, Vol. 10, No. 1, pp. 56–66, February 1988


Virtual Time and Global States of Distributed Systems *

Friedemann Mattern †

Department of Computer Science, University of Kaiserslautern
D 6750 Kaiserslautern, Germany

Abstract

A distributed system can be characterized by the fact that the global state is distributed and that a common time base does not exist. However, the notion of time is an important concept in every day life of our decentralized "real world" and helps to solve problems like getting a consistent population census or determining the potential causality between events. We argue that a linearly ordered structure of time is not (always) adequate for distributed systems and propose a generalized non-standard model of time which consists of vectors of clocks. These clock-vectors are partially ordered and form a lattice. By using timestamps and a simple clock update mechanism the structure of causality is represented in an isomorphic way. The new model of time has a close analogy to Minkowski's relativistic space-time and leads among others to an interesting characterization of the global state problem. Finally, we present a new algorithm to compute a consistent global snapshot of a distributed system where messages may be received out of order.

1 Introduction

An *asynchronous distributed system* consists of several processes without common memory which communicate solely via messages with unpredictable (but non-zero) transmission delays. In such a system the notions of *global time* and *global state* play an important role but are hard to realize, at first sight even their definition is not all clear. Since in general no process in the system has an immediate and complete view of all process states, a process can only *approximate* the global view of an idealized external observer having immediate access to all processes.

The fact that *a priori* no process has a consistent view of the global state and a common time base does not exist is the cause for most typical problems of distributed systems. Control tasks of operating systems and database systems like *mutual exclusion, deadlock detection, and concurrency control* are much more difficult to solve in a distributed environment than in a classical centralized environment, and a rather large number of distributed control algorithms for those problems has found to be wrong. *New problems* which do not exist in centralized systems or in parallel systems with common memory also emerge in distributed systems. Among the most prominent of these problems are *distributed agreement, distributed termination detection*, and the *symmetry breaking or election problem*. The great diversity of the solutions to these problems—some of them being really beautiful and elegant—is truly amazing and exemplifies many principles of distributed computing to cope with the absence of global state and time.

Since the design, verification, and analysis of algorithms for asynchronous systems is difficult and error-prone, one can try to

1. simulate a *synchronous* distributed system on a given asynchronous systems,
2. simulate global time (i.e., a common clock),
3. simulate global state (i.e., common memory),

and use these simulated properties to design simpler algorithms. The first approach is realized by so-called *synchronizers* [1] which simulate clock pulses in such a way that a message is only generated at a clock pulse and will be received before the next pulse. A synchronizer is intended to be used as an additional layer of software, transparent to the user, on top of an asynchronous system so that it can execute synchronous algorithms. However, the message overhead of this mechanism is rather high. The second approach does not need additional messages and the system remains asynchronous in the sense that messages have unpredictable

*This is an edited version, with minor alterations, of the paper with same title which originally appeared in the *Proceedings of the International Workshop on Parallel and Distributed Algorithms* (Chateau de Bonas, France, October 1988), M. Cosnard et al. (ed.) copyright 1989 Elsevier Science Publishers B. V. (North-Holland), with author's permission.

†Current address: Department of Computer Science, University of Saarland, 6600 Saarbrucken, Germany. Email address: mattern@cs.uni-sb.de

120

VECTOR CLOCK

PREREQUISITE: LAMPORT CLOCK

Vector clocks I

At node *i* $(C_i[1], \ldots, \mathbf{C_i[i]}, \ldots, C_i[n])$

- System with ***n*** nodes
- Each node *i* has a **clock C_i** which is **an integer vector of length *n*** (its knowledge of "global" system time):

$$C_i = \mathbf{(C_i[1], C_i[2], \ldots, C_i[n])}$$

- **$C_i(a)$** is the **timestamp** (clock value) of **event *a*** at node *i* (a vector)
- **$C_i[i]$**, component *i* of C_i, is ***i*'s** "local" **logical time**
- $C_i[i]$ represents number of events that node *i* timestamped (assuming its clock increment is 1)

Vector clocks II

- $C_i[k]$, component k of vector C_i (where $k \neq i$), is node *i*'s knowledge of local logical clock at *k*

- $C_i[k]$ is number of events that occurred at *k* that *i* has potentially been causally affected by*

$$\text{At } i\ :\ (\mathbf{C_i[1]},\ \ldots,\ C_i[i],\ \ldots,\ \mathbf{C_i[n]})$$

* Assuming clock increment is 1

Vector clock implementation

- Clock C_i is **incremented** before an event occurs at node *i*

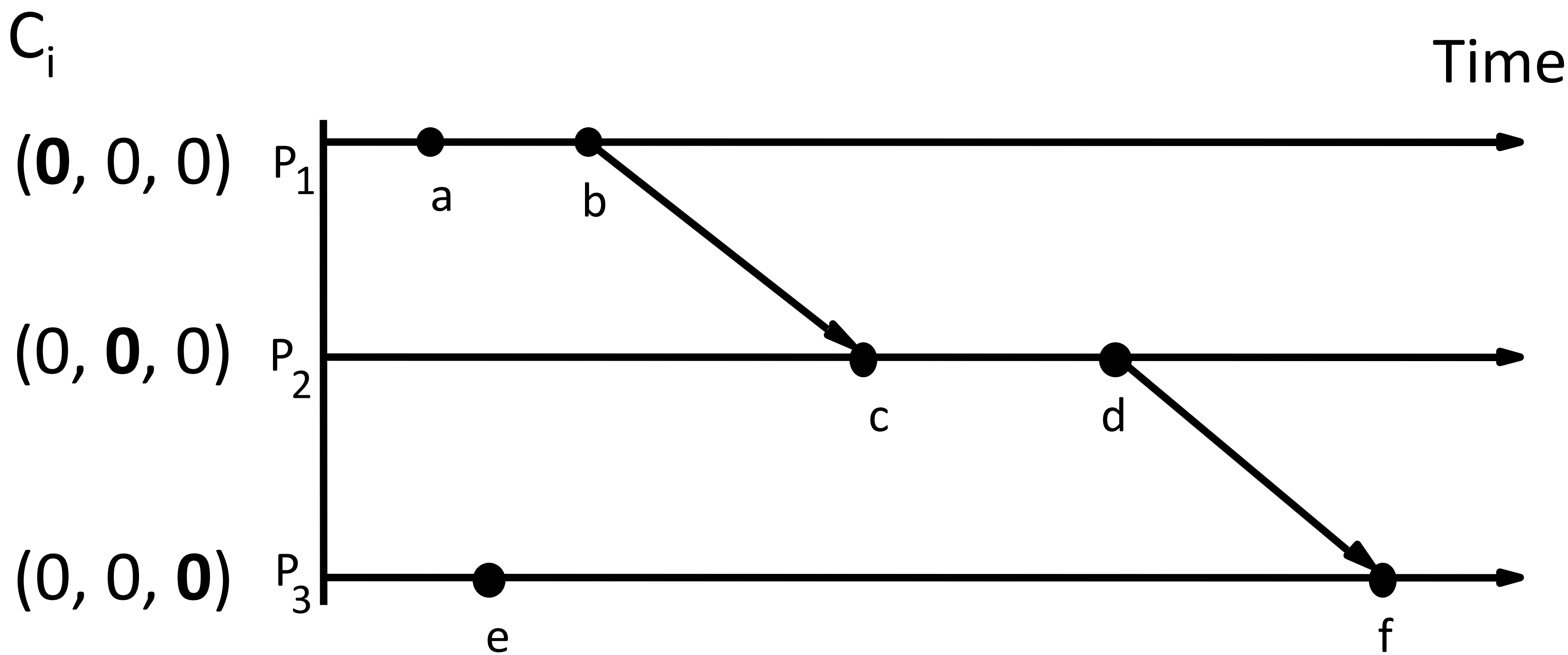
  $$C_i[i] = C_i[i] + d \ (d > 0, \text{e.g.}, d = 1)$$

- If event *a* is the event of sending a message *m* at node *i*, then message *m* is assigned a **vector timestamp** $T_m = C_i(a)$
- When that same message *m* is received by a different node *k*, C_k is updated as follows:

  $$\text{For all } j,\ C_k[j] = \max\{\ C_k[j], T_m[j]\ \}$$

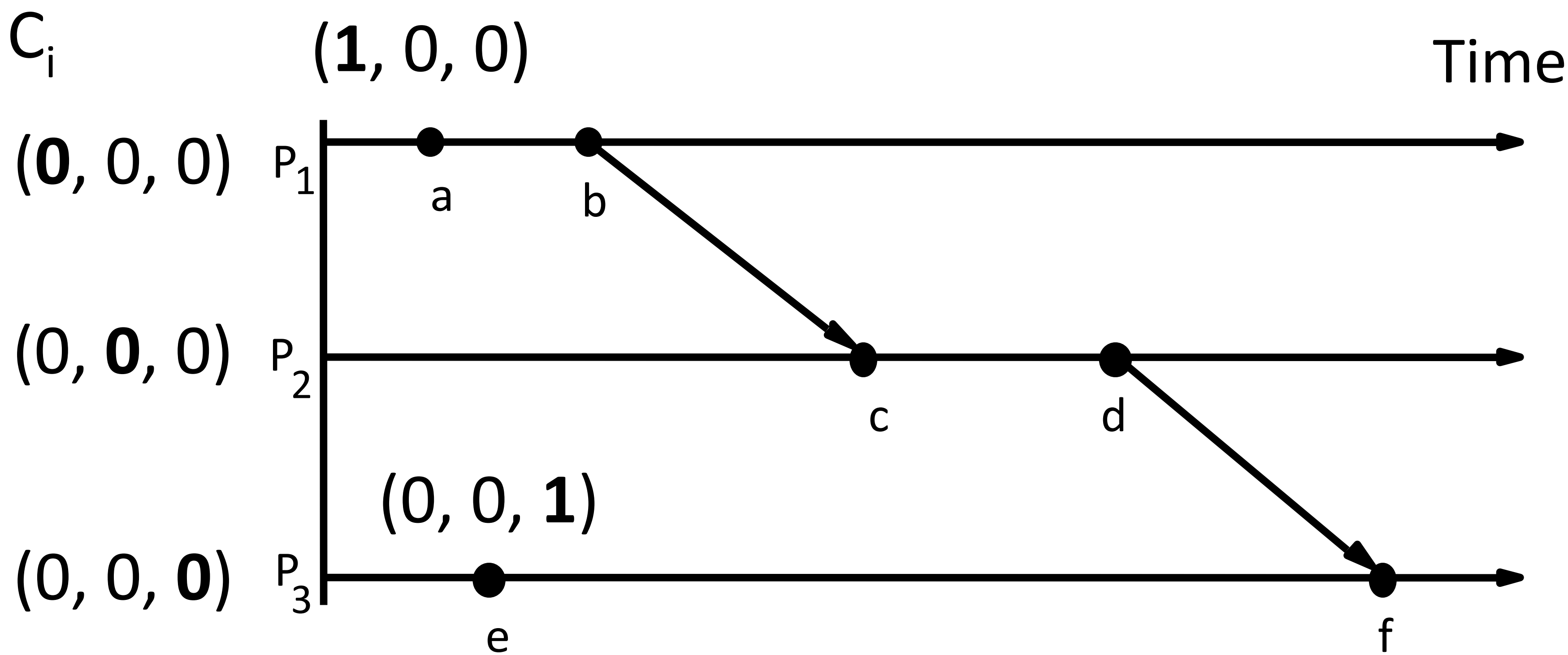
Example: Vector clock

Space-time diagram



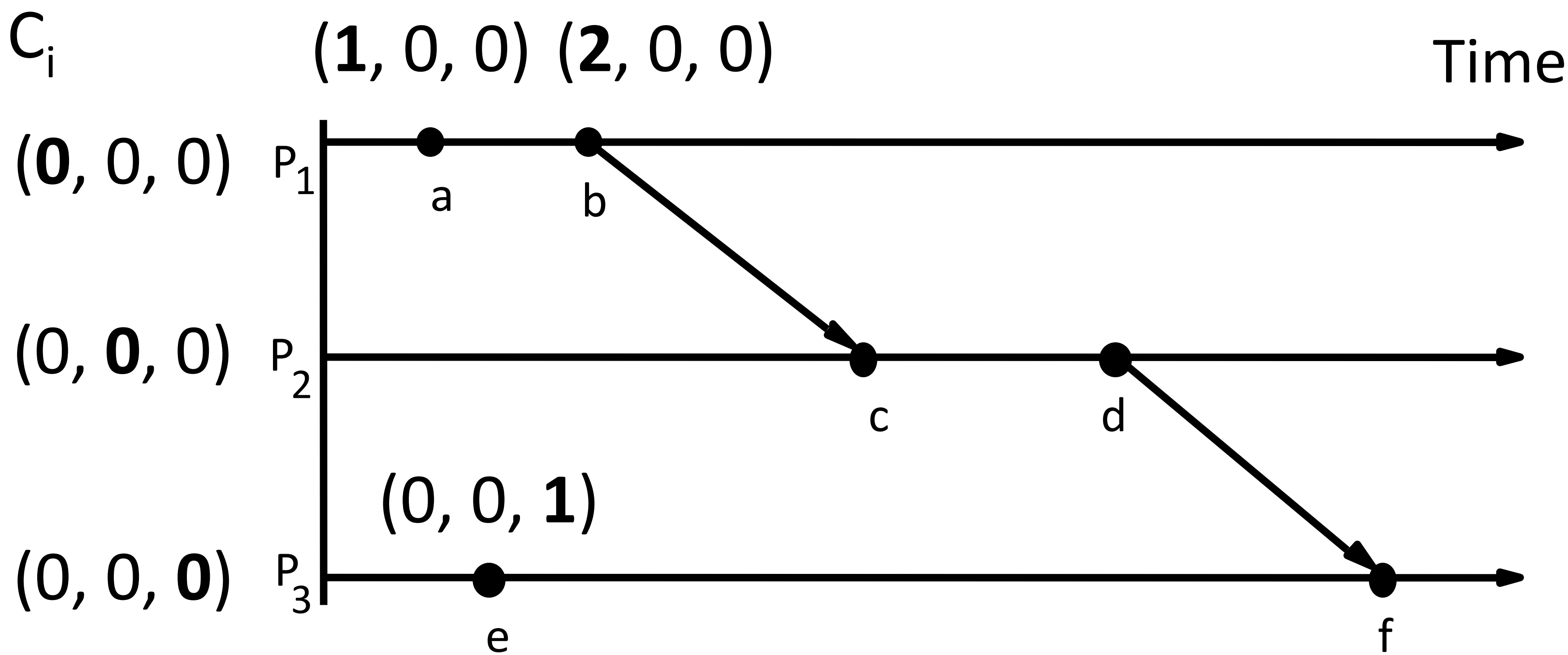
Example: Vector clock

Space-time diagram



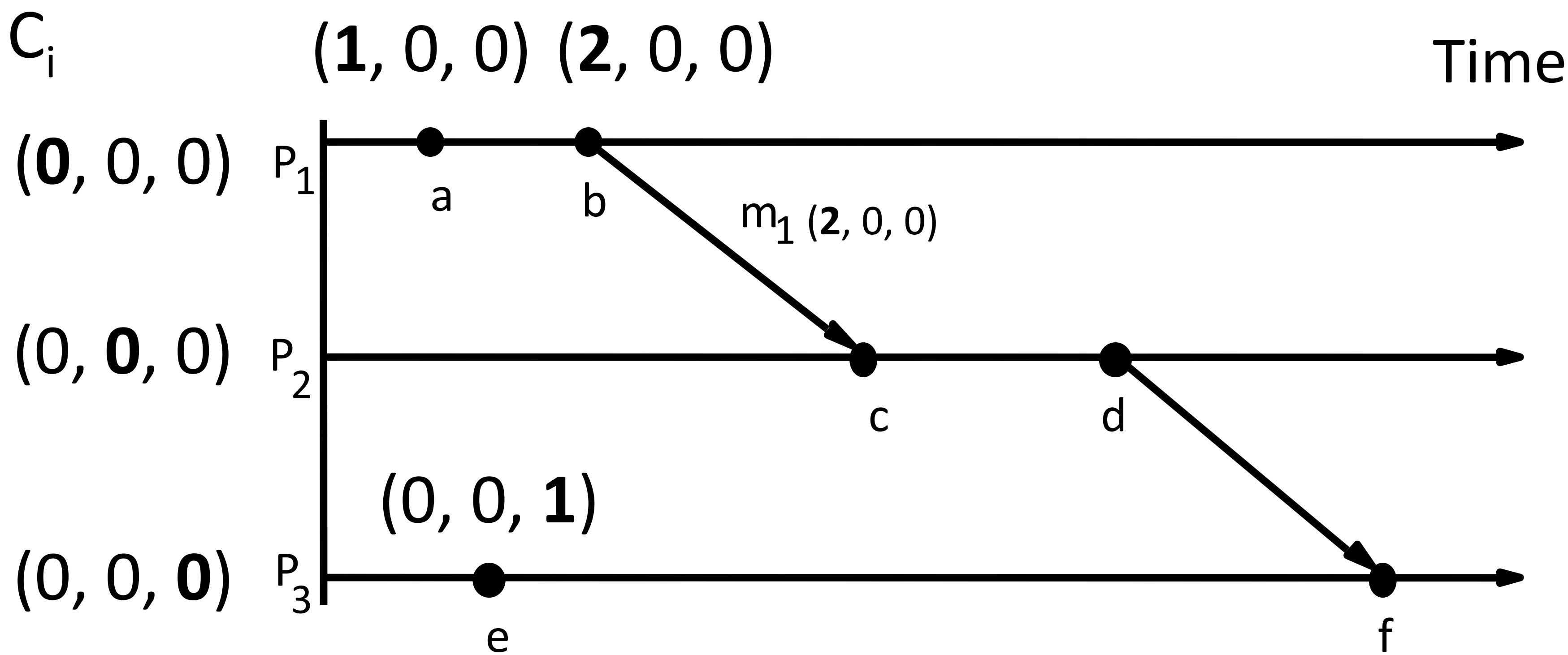
Example: Vector clock

Space-time diagram



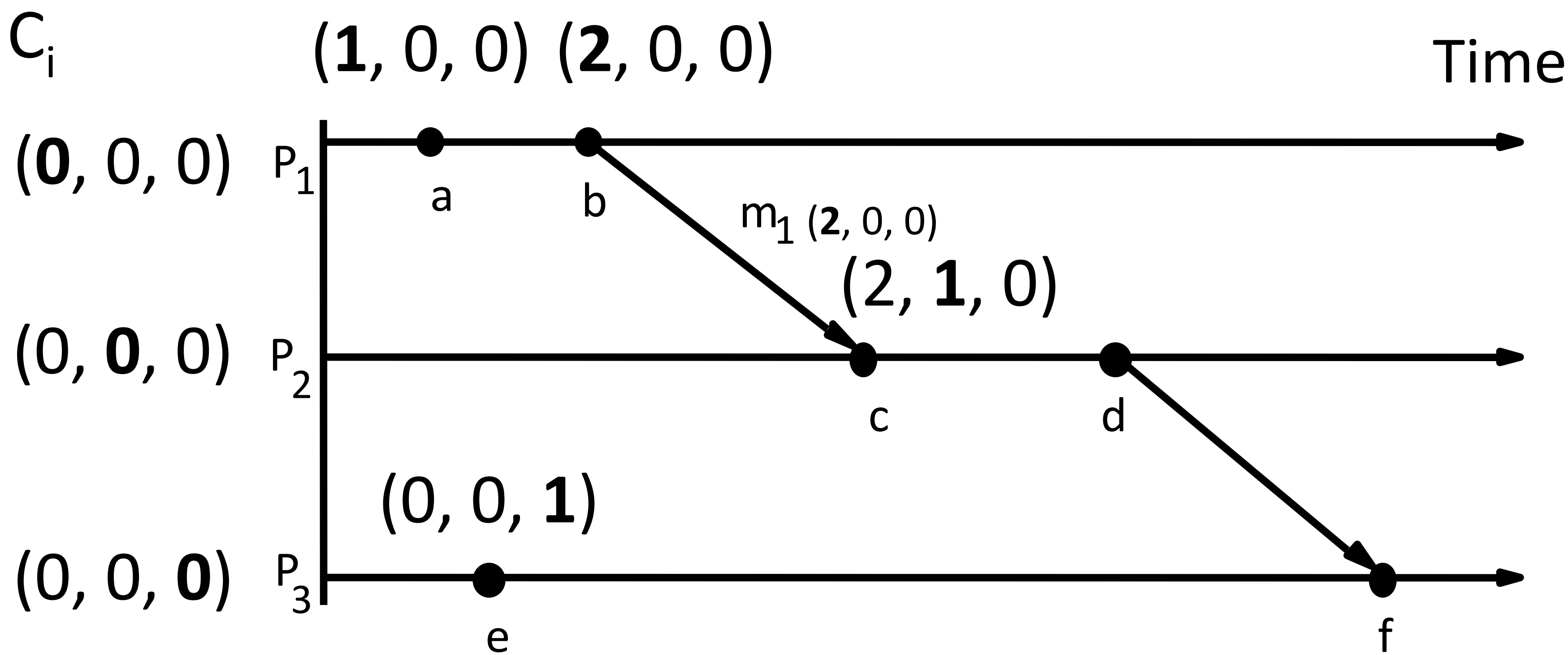
Example: Vector clock

Space-time diagram



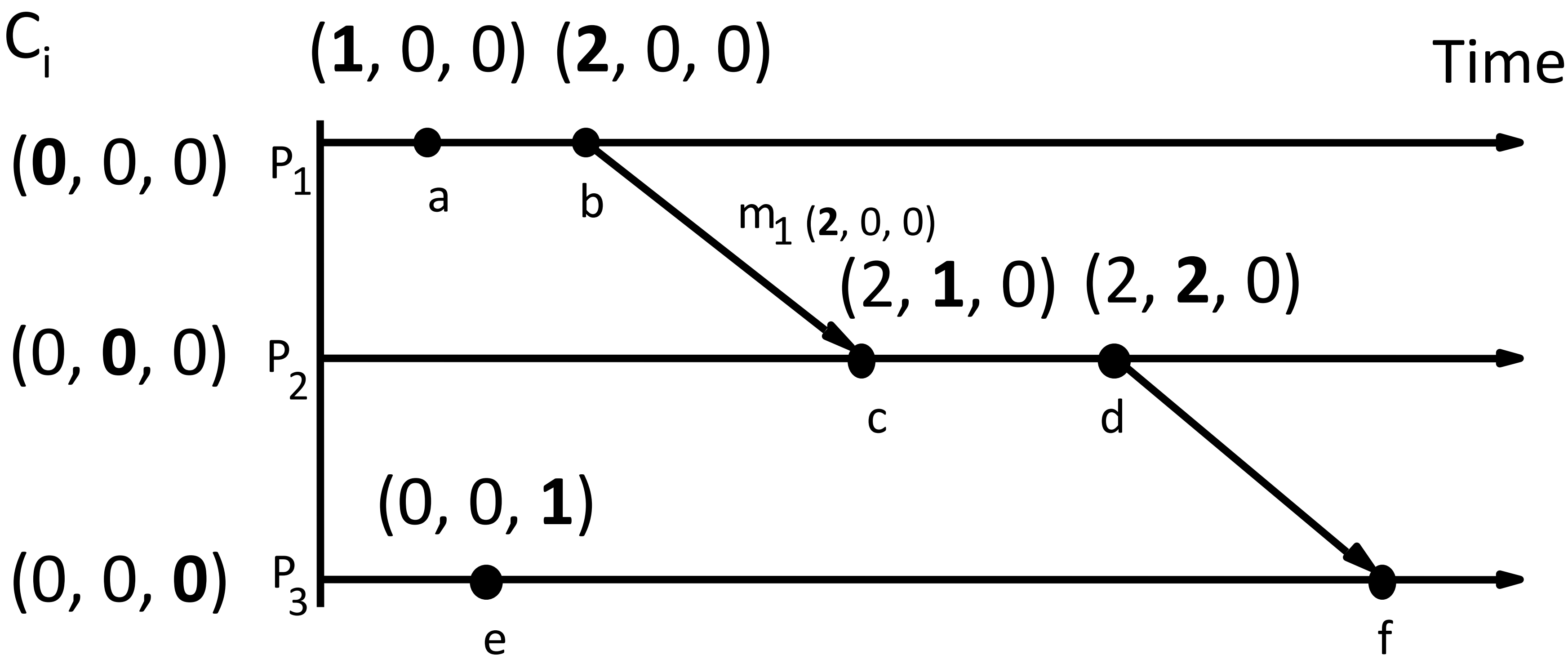
Example: Vector clock

Space-time diagram



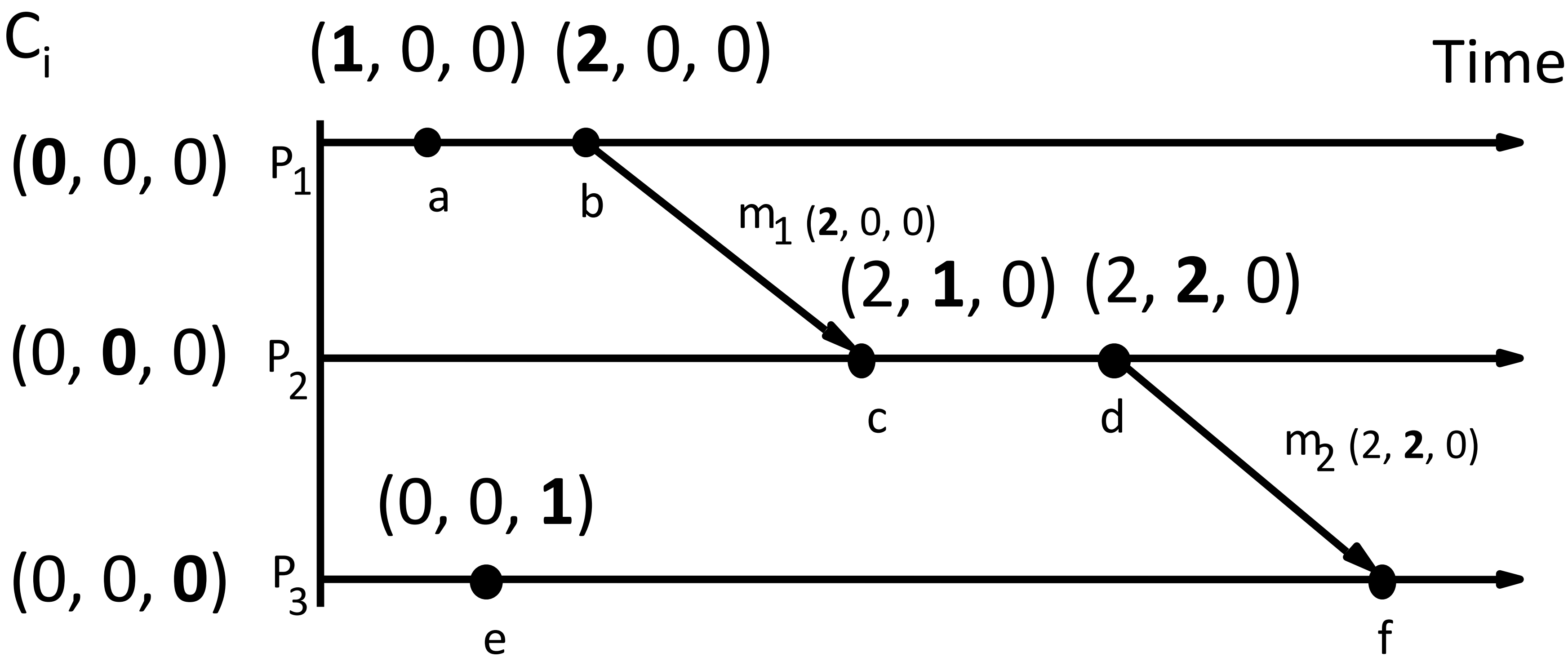
Example: Vector clock

Space-time diagram



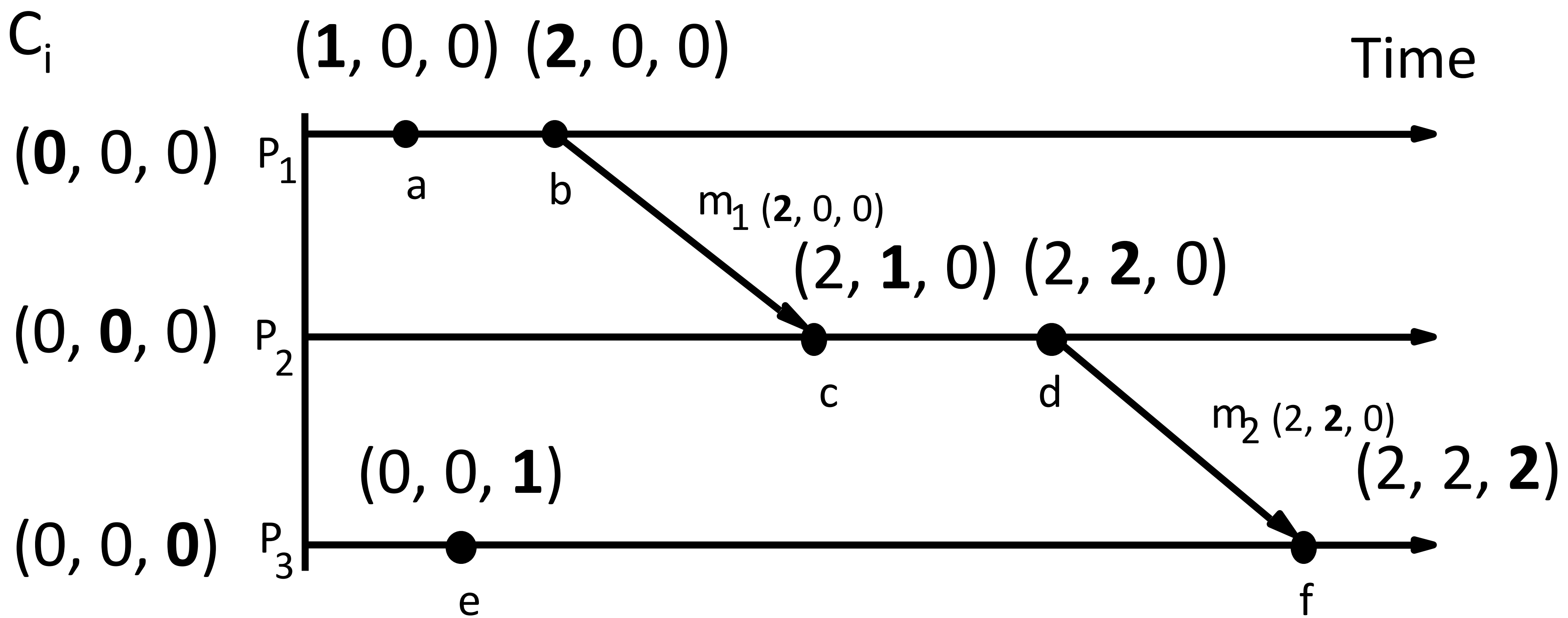
Example: Vector clock

Space-time diagram



Example: Vector clock

Space-time diagram



Comparing vectors

- C_a, C_b are two vectors (vector timestamps)

$C_a = C_b \quad \Leftrightarrow$ for all i: $\quad C_a[i] = C_b[i]$

$C_a \leq C_b \quad \Leftrightarrow$ for all i: $\quad C_a[i] \leq C_b[i]$

$C_a < C_b \quad \Leftrightarrow \quad C_a \leq C_b$ and $\exists\, i: \quad C_a[i] < C_b[i]$

$C_a \,||\, C_b \quad \Leftrightarrow \neg(C_a < C_b)$ and $\neg(C_b < C_a)$

- $\Leftrightarrow$ means "*if and only if*" (a.k.a. "*iff*")*

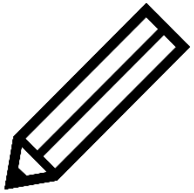
* Two directions

Example: Comparing vectors

- (1 1 2 3) (1 1 2 3)
- (1 1 2 3) (1 1 2 4) and (1 1 2 3) (1 1 2 3)
- (1 1 2 **3**) (1 1 2 **4**)
- (1 1 3 3) (1 1 2 4)

Example: Comparing vectors

- (1 1 2 3) = (1 1 2 3)
- (1 1 2 3)   (1 1 2 4) and (1 1 2 3)   (1 1 2 3)
- (1 1 2 **3**)   (1 1 2 **4**)
- (1 1 3 3)   (1 1 2 4)



Example: Comparing vectors

- (1 1 2 3) = (1 1 2 3)

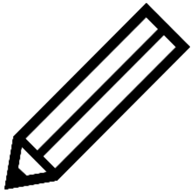
- (1 1 2 3) ≤(1 1 2 4) and (1 1 2 3) (1 1 2 3)

- (1 1 2 **3**) (1 1 2 **4**)

- (1 1 3 3) (1 1 2 4)

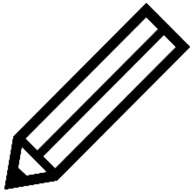
Example: Comparing vectors

- (1 1 2 3) = (1 1 2 3)
- (1 1 2 3) ≤ (1 1 2 4) and (1 1 2 3) ≤ (1 1 2 3)
- (1 1 2 **3**) (1 1 2 **4**)
- (1 1 3 3) (1 1 2 4)



Example: Comparing vectors

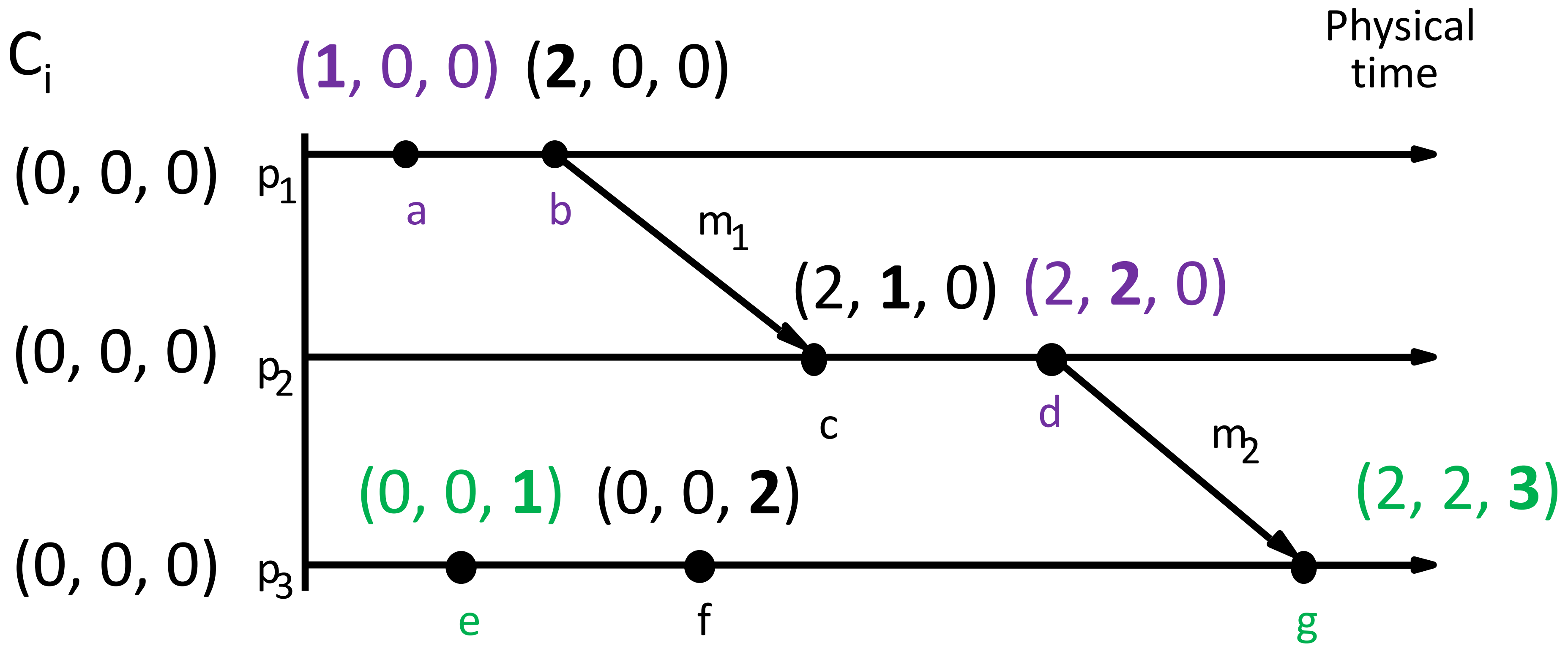
- (1 1 2 3) = (1 1 2 3)
- (1 1 2 3) ≤ (1 1 2 4) and (1 1 2 3) ≤ (1 1 2 3)
- (1 1 2 **3**) < (1 1 2 **4**)
- (1 1 3 3)   (1 1 2 4)



Example: Comparing vectors

- (1 1 2 3) $=$ (1 1 2 3)
- (1 1 2 3) $\leq$ (1 1 2 4) and (1 1 2 3) $\leq$ (1 1 2 3)
- (1 1 2 **3**) $<$ (1 1 2 **4**)
- (1 1 3 3) $||$ (1 1 2 4)

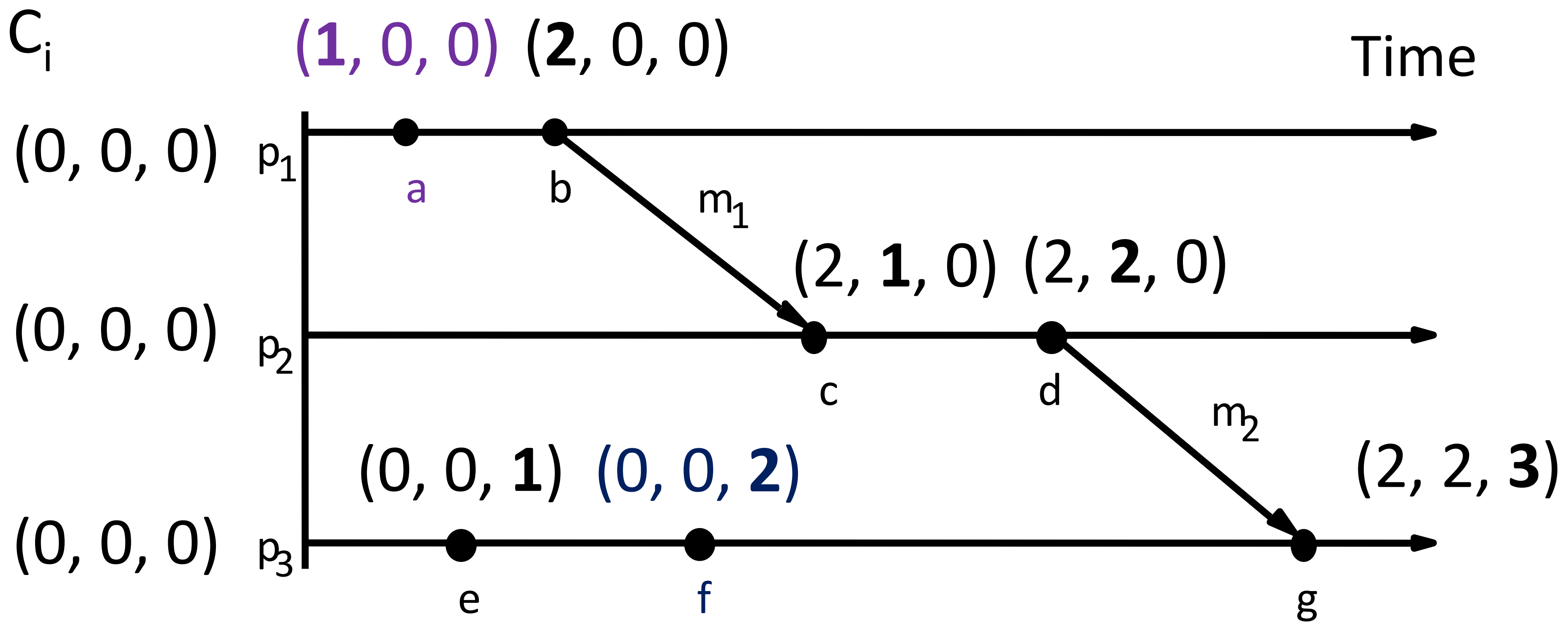
Example: Vector clock



$(1, 0, 0) < (2, 2, 0) \Leftrightarrow a \rightarrow d$

$(0, 0, 1) < (2, 2, 3) \Leftrightarrow e \rightarrow g$

Example: Vector clock



(1, 0, 0) || (0, 0, 2) ⇔ a || f

With Lamport Clock: 1 < 2, but not a → f !

Comparing vector clocks

- Let a, b be two events with vector timestamps C_a, C_b then:
  - $a \rightarrow b \quad \Leftrightarrow \quad C_a < C_b$ (both directions!)
  - $a \,||\, b \quad \Leftrightarrow \quad C_a \,||\, C_b$ (both directions!)

- E.g., "⇔" means
  - If $a \rightarrow b$ then $C_a < C_b$
  - If $C_a < C_b$ then $a \rightarrow b$

Summary

- Similar context as Lamport Clocks
- Represent knowledge about **logical time** with a **vector of size *n***
- Each vector component *i* represents node's knowledge of **logical clock** at node *i*
- Node's own vector component **tracks number of events it has timestamped** (assuming d = 1)
- Comparing timestamps based on comparing vectors
- **Isomorphic relationship between timestamps and event order** (i.e., inferences in both direction possible, fundamental difference to Lamport clock)

Self-study questions

- Can all events in a distributed system be ordered?
- Does a vector clock induce a partial or total order?
- Why is it important to totally order events?
- Can two events have equal vector clock timestamps?
- For two events to be concurrent, what does this imply for their timestamps?
- If clocks are initialized to zero at the beginning of time and *d* is always one, what conclusions can we draw from looking at timestamps?
- What are applications of vector clocks?