Time in Distributed Systems

- Overview

  1 Notion of time in distributes systems

  2 Physical time synchronization

  3 Logical time

The notions of time...

- Examples
  - Selling tickets: Who did book the last seat on the plane first?
  - Updating variables: which value is the most current one?
- Astronomical Time: years, seconds, ...
- International Atomic Time: 1 standard sec = 9,192,631,770 periods of transition for Cs133
- UTC (Universal time coordinated): 1 leap sec is occasionally inserted, or more rarely deleted, to keep in step with Astronomical Time
  - Time signals broadcasted from land-based radio stations and satellites (GPS)
  - Accuracy: 0.1-10 millisec (land-based), up to 1 microsec (GPS)

The reality

- Each machine has a local clock
- $H_i(t)$: hardware clock value (by oscillator, discontinuous)
- How accurate are hardware clocks?
  - ➢ Ordinary quartz clocks: 10^{-6} drift-seconds/second
  - ➢ High-precision quartz clocks: 10^{-8}
- $C_i(t)$: software clock value (generated by OS)
  - ➢ $C_i(t) = r\, H_i(t) + A$, a **tick** occurs every so many quartz oscillations!
  - ➢ Clock resolution: period between updates of $C_i(t)$

    $\Rightarrow$limit on determining order of events
- $C_i(t)$: Approximation of the UTC

  # nsec's elapsed at time t since a reference time

The Reality – Local clock

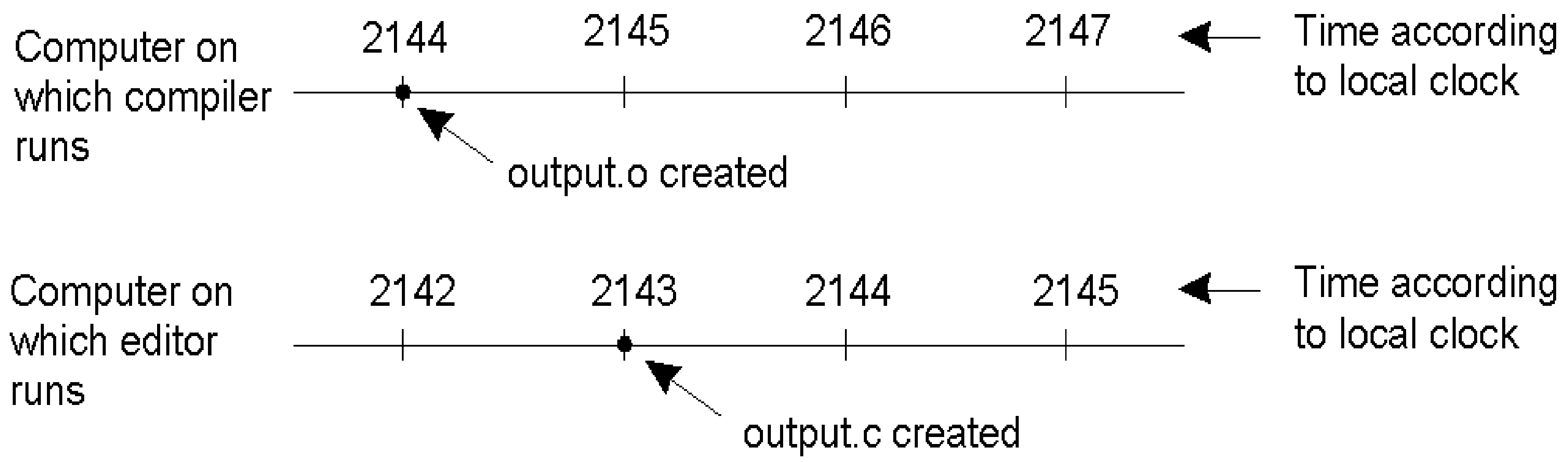
- Local clock value C(t), where t is UTC

  $C_i(t) = r * t + A$ where: A = offset; r-1 = skew

- No guarantee of accuracy, but **never runs backwards** !!!
- Drift: different rates of counting time
  - Physical variations of underlying oscillators, variance with temperature
  - Drift rate: difference in reading btw. a clock and a nominal "perfect clock" per unit of time measured by the reference clock
    - 10^{-6} seconds/sec for quartz crystals
    - 10^{-7} - 10^{-8} seconds/sec for high precision quartz crystals
- This accumulates over time …
  - Clocks on different machines will eventually differ substantially
  - With common quartz ➔ 1 second per 11.6 days

Clocks are not precise ...

- File edition and File compilation run on different machines


Computer on which compiler runs
2144
2145
2146
2147
Time according to local clock
output.o created
Computer on which editor runs
2142
2143
2144
2145
Time according to local clock
output.c created

- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time
  - The file has been compiled before its creation ???

The problem → The approaches

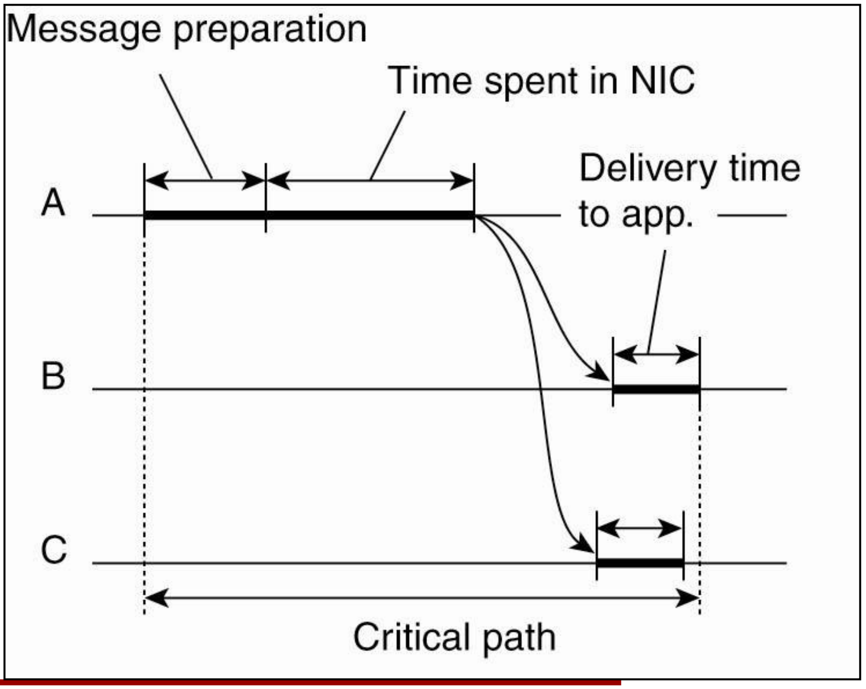
- The notion of accurate time is very problematic …

  We are limited in our ability to timestamp events at different nodes sufficiently accurately to know the order in which any pair of events occurred, or whether they occurred simultaneously.

- Two main approaches conceivable
  - ➢ Try to compensate for drift of real clocks
  - ⇒ Periodic Clock synchronization, BUT
    - ▪ Never make time go backwards! - Would mess up local orderings
    - ▪ If you want to adjust a clock backwards, just slow it down for a some time…
  - ➢ Try to do without information about the real, actual time – order of events is often sufficient ⇒ Logical time

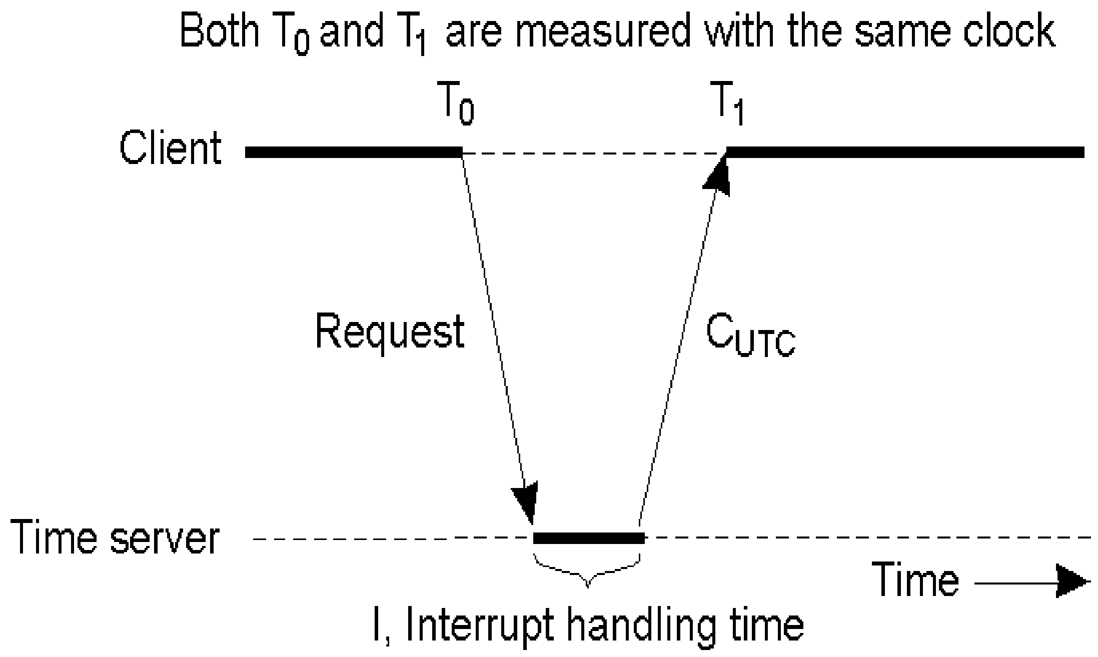
Clock Synchronization in Physical Time

- Synchronize all computer clocks (plural! distributed system) with external UTC source
  - ➢ Downside: receivers capable of accurate time signals are expensive, GPS does not work indoors
  - ➢ Or only mutually (internally) synchronize clocks of all nodes
- Set of clocks is synchronized, if clock difference of every pair of clocks within this set is below threshold!
- However, there is inaccuracy of Clock Adjustments

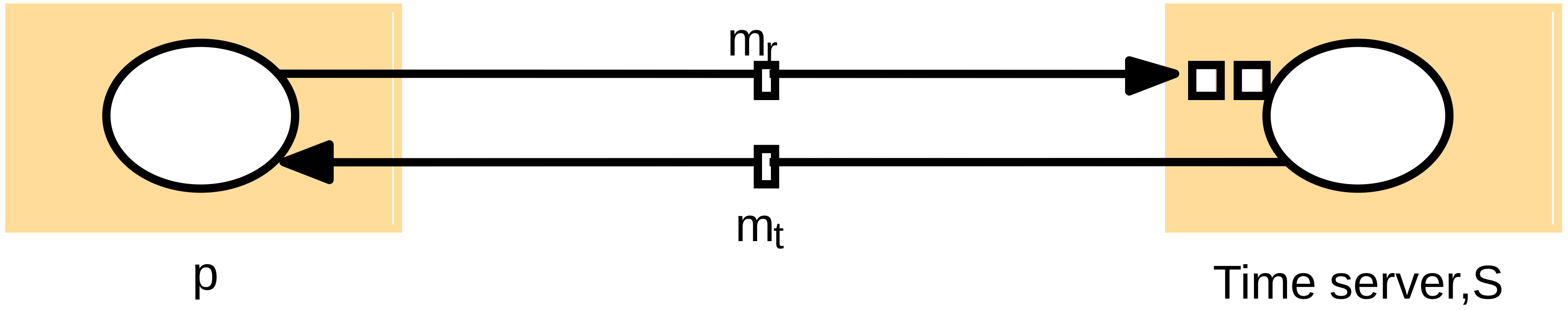


Cristian's Algorithm
[Marzakis]

- Getting the current time from a time server
- Round Trip time $T = T_1 - T_0$


Both T_0 and T_1 are measured with the same clock
T_0
T_1
Client
Request
C_{UTC}
Time server
Time
I, Interrupt handling time

Time servers: Cristian's algorithm [Marzakis]



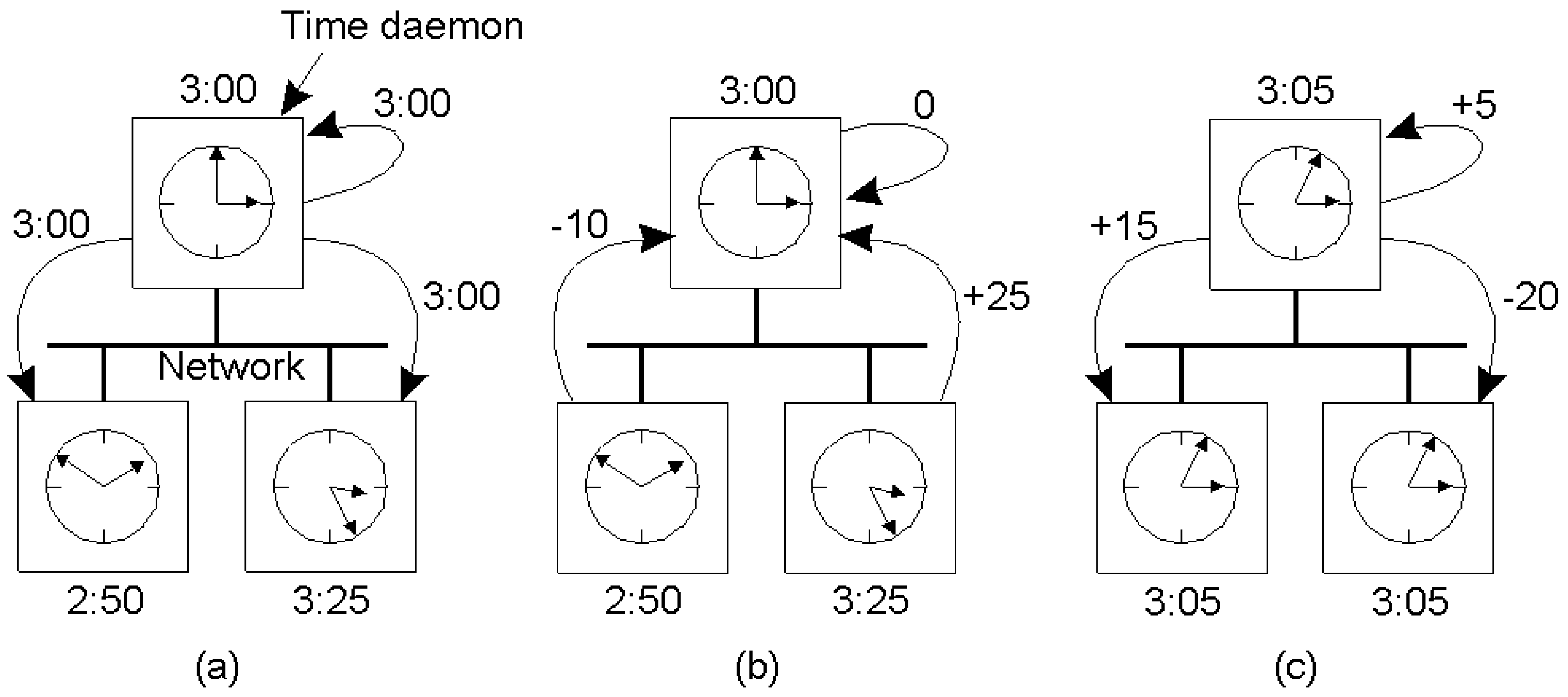
T_{round} := total round-trip time
t := time value in message m_t
estimate := (t + T_{round} /2)

Time by S's clock when reply msg arrives $\in$ [t+min, t+**T_{round}**-min]

(min: smallest seen T_{round}/2)

Accuracy: $\pm$ **(T_{round}/2 - min)**

The Berkeley Algorithm [Tanenbaum]



- Synchronisation in LAN (not necessarily corresponding to real time)
  - a) Time daemon asks all the other machines for the clock values
  - b) The machines answer
  - c) The time daemon averages and tells <u>everyone</u> to adjust their clock

The Berkeley algorithm (I)
[Marzakis]

- Co-ordinator (master) periodically polls slaves
  - ➢ estimates each slave's local clock (based on RTT)
  - ➢ averages the values obtained (incl. its own clock value)
  - ➢ ignores any occasional readings with RTT higher than max
- Slaves are notified of the adjustment required
  - ➢ This amount can be positive or negative
- Elimination of faulty clocks
  - ➢ averaging over clocks that do not differ from one another more than a specified amount
  - ➢ Intuition: Both to speedy and to slow "cancel" mutually…
- Election of new master, in case of failure

Averaging algorithms
[Marzakis]

- Divide time into fixed-length re-synchronization intervals
  - $[T_0 + iR, T_0 + (i+1)R]$
  - At the beginning of an interval, each machine broadcasts the current time according to its clock

    … and starts a local timer to collect all incoming broadcasts during a time interval S

  - New time value is computed, after all broadcasts received
    - Average
    - Average after discarding the m lowest and the m highest values

      $\Rightarrow$ Tolerate up to m faulty machines
    - May also correct each value based on estimate of propagation time from the source machine

What is NTP?

- Time synchronization system for computer clocks through the Internet

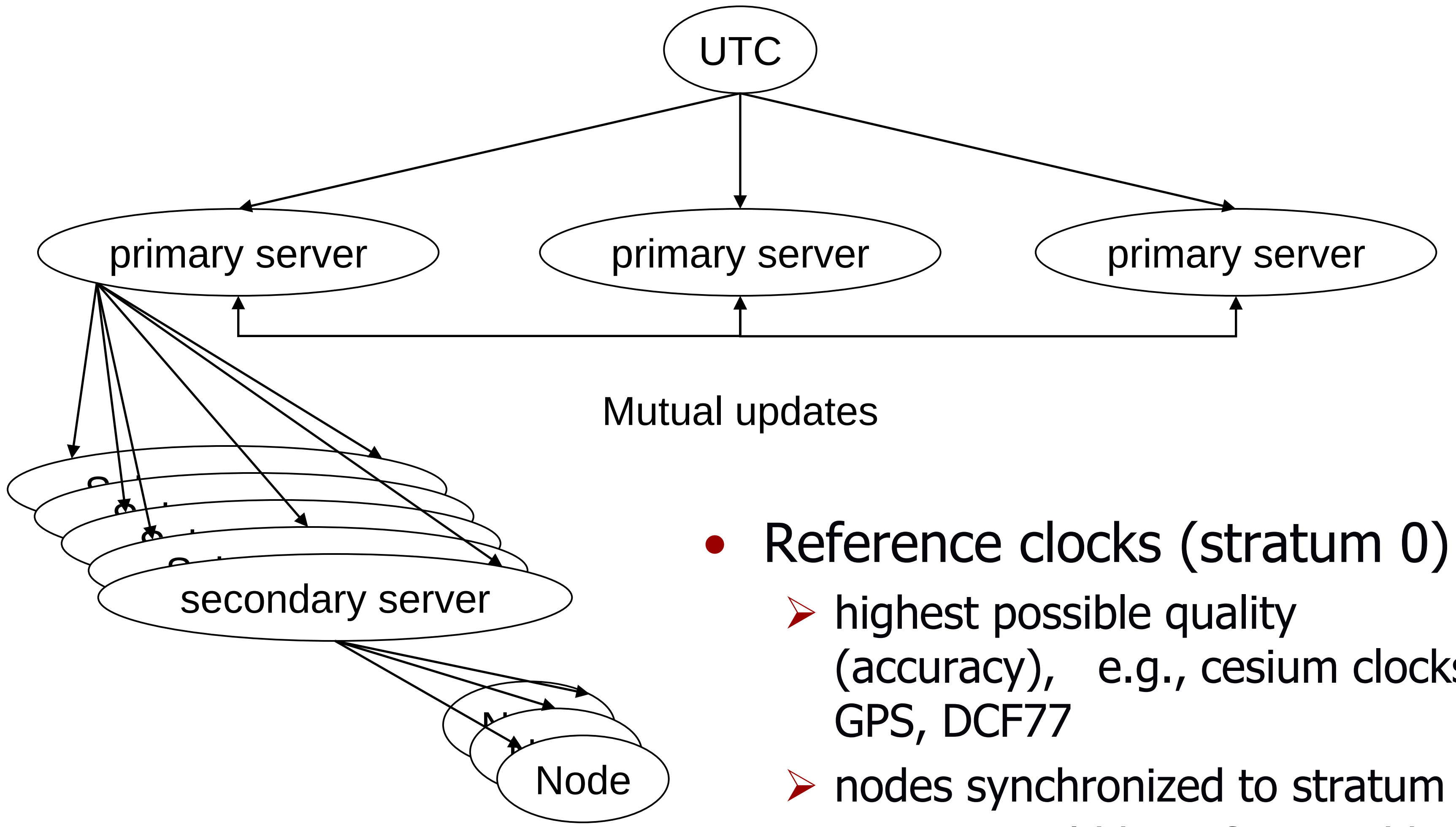
- Internet standard protocol
  - Version 3 (RFC-1305), Simple NTP Version 4 (RFC-2030)
  - Application Layer (OSI-layer 7) using UDP
  - Clients exist for almost all platforms

- Provides mechanisms to
  - synchronize clocks to some reference time
  - coordinate time distribution in a large, diverse internet

Basic Features

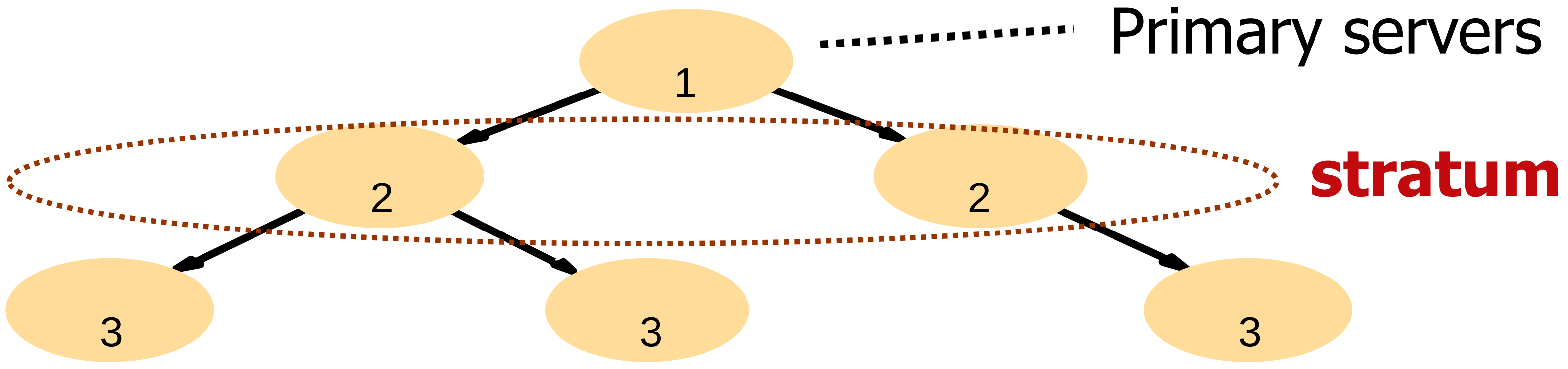
- Based on UTC (Universal Time Coordinated)
- independent from time zones
  - several institutions contribute their estimate of the current time
- Fault-tolerant
  - selects the best of several available time sources
  - multiple candidates can be combined to minimize the accumulated error
  - detects and avoids insane time sources
- Highly scalable
  - reference clock sources, sub-nodes and clients form hierarchical graph
  - each node can exchange time information either bidirectional or unidirectional

Network Time Protocol



- Reference clocks (stratum 0)
  - highest possible quality (accuracy), e.g., cesium clocks, GPS, DCF77
  - nodes synchronized to stratum 0
  - sources could be referenced by other nodes

NTP Synchronization Subnets



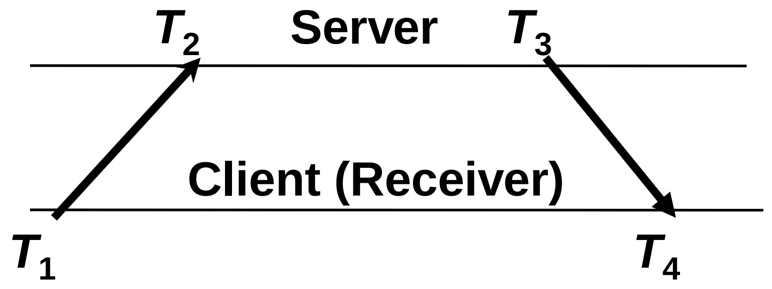
High stratum # → server more liable to be less accurate

Node → root RTT as a quality criterion

- 3 modes of synchronization:
  - multicast: acceptable for high-speed LAN
  - procedure-call: similar to Christian's algorithm
  - symmetric: between a pair of servers
- All modes rely on UDP messages

How does NTP work? Time exchange

- Exchange of several packet pairs (request, reply)
- Containing originate resp. receive timestamp
- Receiver estimates travelling time (delay) and jitter (offset/delay error)



[Note : Tanenbaum's delay $= \delta / 2$]

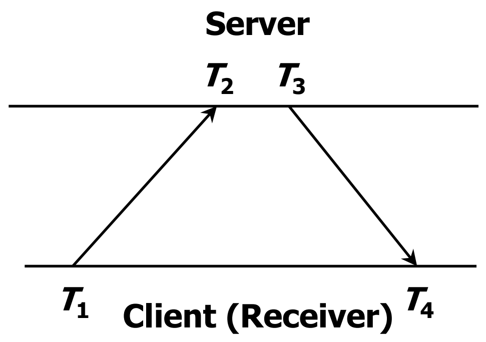
$$\text{Offset} \quad \theta = \tfrac{1}{2}[(T_2 - T_1) + (T_3 - T_4)]$$

$$\text{Delay} \quad \delta = (T_4 - T_1) - (T_3 - T_2)$$

- If sources disagree, largest set of agreeing sources is used to calculate combined reference time, set of disagreeing sources is declared invalid
- First Synchronization usually takes about 5 min.
- Estimation quality improves over time -> more sources could get invalid

Time Exchange – Example

Timestamp Name	ID	When Generated
Originate Timestamp	T1	time request sent by client
Receive Timestamp	T2	time request received at server
Transmit Timestamp	T3	time reply sent by server
Destination Timestamp	T4	time reply received at client



Delay d = [(T4-T1) -(T3-T2)]

Example:

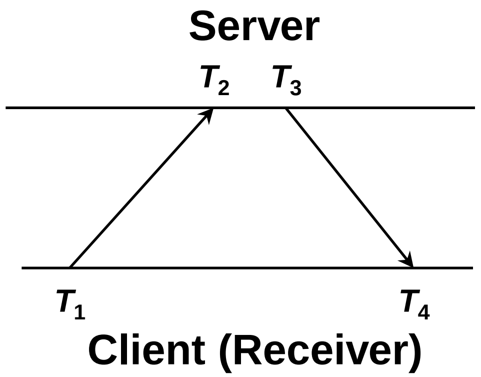
-Local time at client:	**T1 = 117**	->	packet is sent
-Local time at server:	**T2 = 115**	->	packet is received (at server)
-Local time at server:	**T3 = 115.5**	->	packet is sent back
-Local time at client:	**T4 = 125**	->	packet is received (at client)

Delay calculation (naive): T4 – T1 = 125 – 117 = 8
But: T3 – T2 = processing overhead (server) = 115.5 – 115 = 0.5
-> Delay = 8 – 0.5 = 7.5

Internet: Processing overhead small compared to round trip time

Time Exchange – Example

Timestamp Name	ID	When Generated
Originate Timestamp	T1	time request sent by client
Receive Timestamp	T2	time request received at server
Transmit Timestamp	T3	time reply sent by server
Destination Timestamp	T4	time reply received at client



$$\text{Offset } \Theta = 0.5[(T2-T1) + (T3-T4)]$$

Example:

-Local time at client:	**T1 = 117**	->	packet is sent
-Local time at server:	**T2 = 115**	->	packet is received (at server)
-Local time at server:	**T3 = 115.5**	->	packet is sent back
-Local time at client:	**T4 = 125**	->	packet is received (at client)

Offset calculation: **T2 - T1 = 115 – 117 = -2**

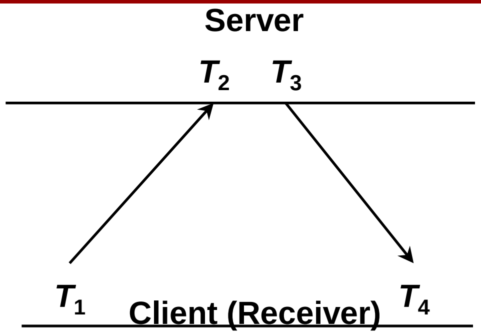
T3 - T4 = 115.5 – 125 = -9.5

-> -2 + -9.5 = -11.5

Offset: -> -11.5 / 2 = -5.75

Time Exchange – Example

Timestamp Name	ID	When Generated
Originate Timestamp	T1	time request sent by client
Receive Timestamp	T2	time request received at server
Transmit Timestamp	T3	time reply sent by server
Destination Timestamp	T4	time reply received at client



Offset $\Theta = 0.5[(T2-T1) + (T3-T4)]$

The time offset of the client to the server is -5.75 [ms].

Corrected time at the client:

T1 = 117 + -5.75 = 111.25

T4 = 125 + -5.75 = 119.25

new time distance T1 to T2 is 115 - 111.25 = 3.75

new time distance T3 to T4 is 119.25 - 115.5 = 3.75

This calculation uses the premise that transport is symmetric (same time in both directions) -> in principle the timeline must be an equilateral triangle

Many packet pairs and many sources are averaged over a long time -> accuracy increases

NTP data filtering & peer selection

- Retain 8 most recent <oi, di > pairs
  - compute “filter dispersion” metric
    - higher values → less reliable data
    - The estimate of offset with min. delay is chosen
- Examine values from several peers and look for relatively unreliable values
- May switch the peer used primarily for sync.
- Peers with low stratum # are more favored
  - “closer” to primary time sources
- Also favored are peers with lowest sync. dispersion:
  - sum of filter dispersions bet. peer & root of sync. subnet
- May modify local clock update frequency wrt observed drift rate
- Accuracy:

  ~10s of milliseconds over Internet paths

  ~ 1 millisecond on LANs

NTP Version 4 (1)

- Latest NTP Version (2010)

- Basic Principles remain unchanged
  - Same Time Exchange Mechanism
  - Compatible to Version 3 and Simple Network Time Protocol (SNTP)

- Reference Implementation Changes:
  - IPv6 ready
  - Dynamic Server Discovery over Multicast
  - Longer data representation
    - Preparation for Version 5: should solve “year 2036 bug”

NTP V. 4 _improved Algorithms

- Better statistical processing

- Reduced impact of network jitter and asymmetric delays

→ Better accuracy (ca. Factor 10) → 10 ms…

Precision Time Protocol (PTP / IEEE 1588)

„Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems"

- Gap: high-speed LAN, no GPS, NTP not precise enough
  - ➢ Example: automation systems, manufacturing robots

- Key features:
  - ➢ High Precision with Hardware support (<10^{-6}s)
  - ➢ Based on International Atomic Time (TAI)
    - ▪ No leap seconds, UTC offset (2013: 35s) given by Master Clock
  - ➢ Scalable:
    - ▪ Hierarchical Master-Slave Topology
    - ▪ UDP Multicast for all Messages (Unicast allowed in 2008 revision)
  - ➢ Fault-tolerant:
    - ▪ Best master clock algorithm chooses best master clock to synchronize with

Precision Time Protocol (PTP / IEEE 1588)

- NTP not precise enough for measurements, control, automation

  ➔ *„Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*"

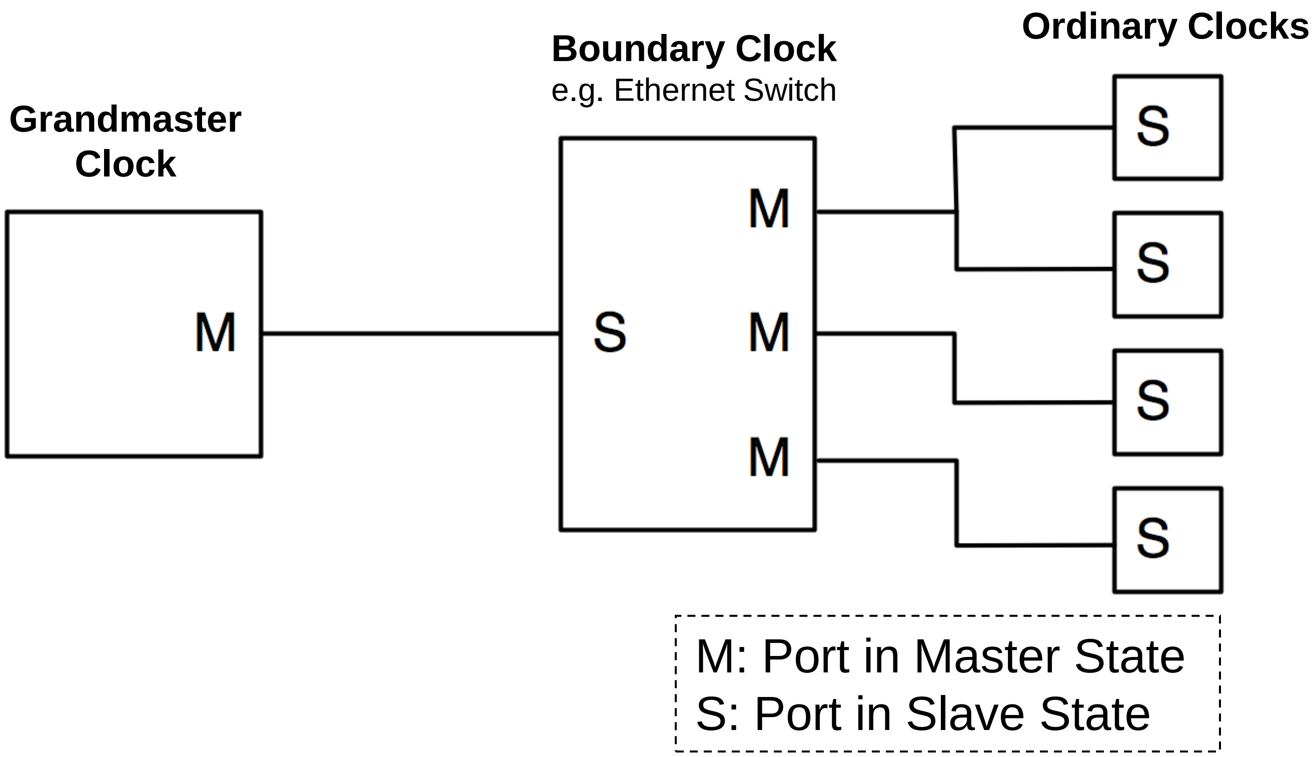
Features:

- Intended for SMALL networks (high-speed LANs)
- Hardware support specified
- UTC offset (2013: 35s) instead of leap seconds
- Hierarchical Master-Slave Topology
  - ➢ Best master clock chosen to synchronize with
  - ➢ UDP based message exchange (similar to NTP!)

➔ **Achieved accuracy (<10^{-6}s)**

Precision Time Protocol Principles of high Accuracy (2)

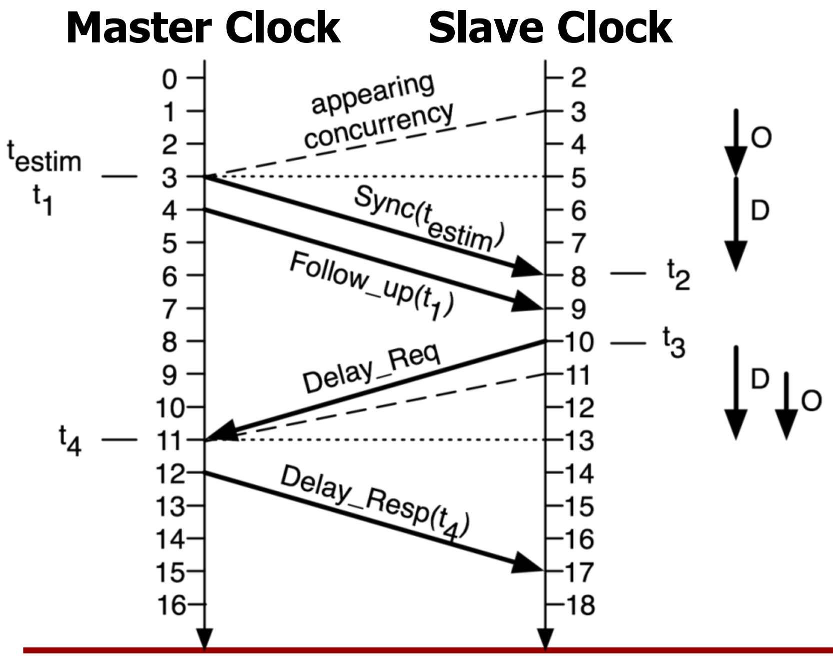
2. Master-Slave Topology:



- ➢ Minimize number of network segments to pass ➔ minimize jitter
- ➢ Accuracy gain negligible for most networks & applications

Precision Time Protocol Principles of high Accuracy (1)

1. Precise Timestamps ("preamble hardstamps")
   - Precise physical layer timestamp in follow_up packet
   - In Software (Driver Interrupt): 20x-100x accuracy compared to NTP (problems under heavy load)
   - In Network Interface Card (NIC) ➔ Hardware support
   - Hardware: further 25x-50x accuracy (depending on load)



O = Offset Θ
D = One-way delay δ/2

$$\delta = (t_4 - t_1) - (t_3 - t_2)$$

$$\Theta = \frac{1}{2}\left[(t_2 - t_1) + (t_3 - t_4)\right]$$

➔ Same Calculation as NTP

Logical time

- Time seen by external observer. A global clock of perfect accuracy ➔ not realistic, sorry!!!! ☹

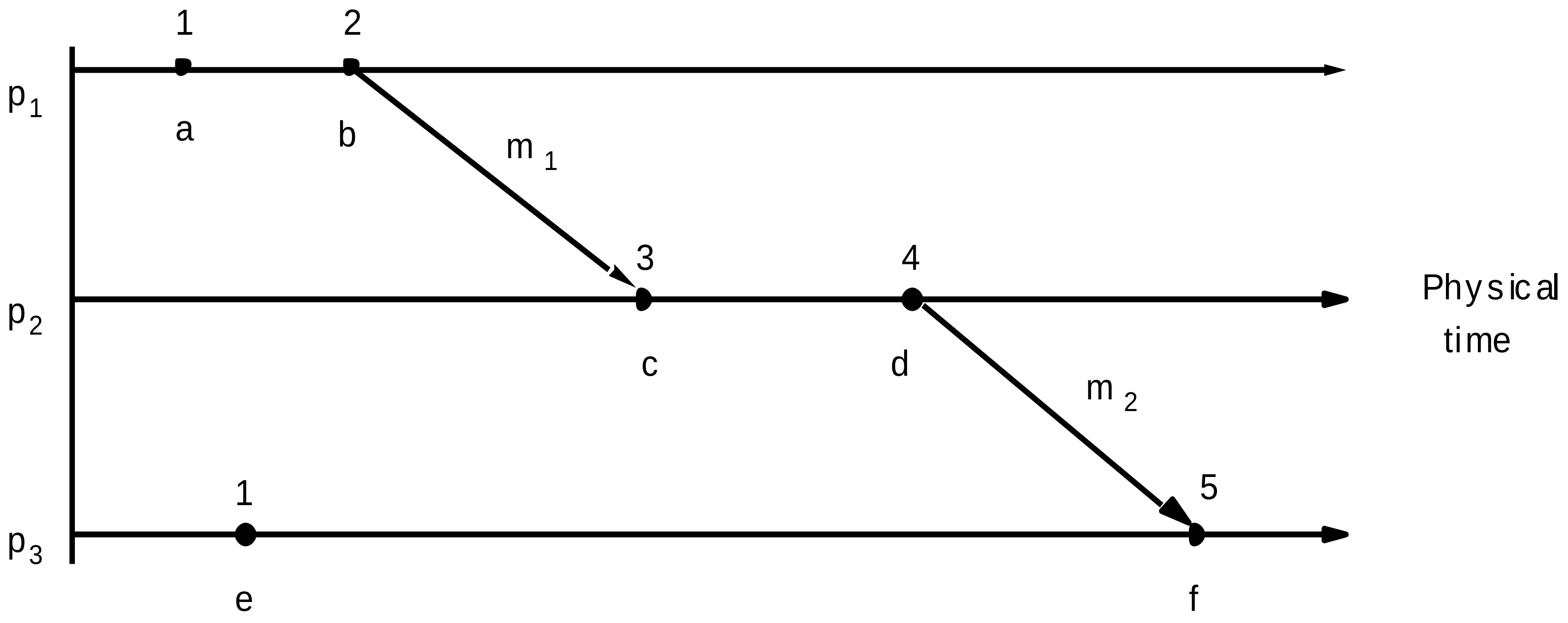
- Time seen on clocks of individual processes.
  Local clocks do drift out of sync ☹

- But: Do we REALLY ALWAYS need the precise time ?
  - ➢ Ordering is usually enough
  - ➢ Logical notion of time: event a occurs before event b

Lamport "Happens Before"

- A $\rightarrow$ B means A "happens before"
  - A and B are in same process, and B has a later timestamp
  - A is the sending of a message and B is the receipt
- Transitive relationship
  - A $\rightarrow$ B and B $\rightarrow$ C implies A $\rightarrow$ C
- If neither A $\rightarrow$ B nor B $\rightarrow$ A are true, then A and B are "***concurrent***" (this does not mean *simultaneous*)

Happened-before relationship, concurrent events



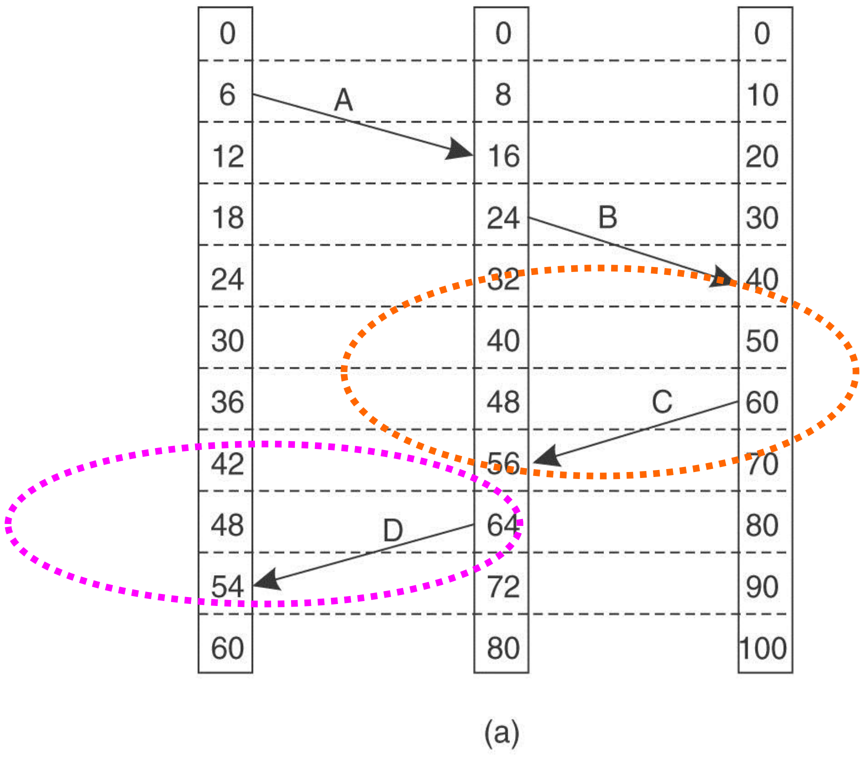
- (a→f) (via b, c, d): also (independently) e → f
- But: b and e are not ordered; b and e are concurrent
- Happened-before relation represents only potential causality
  - All messages that possibly had influence on a given event contribute to this partial order

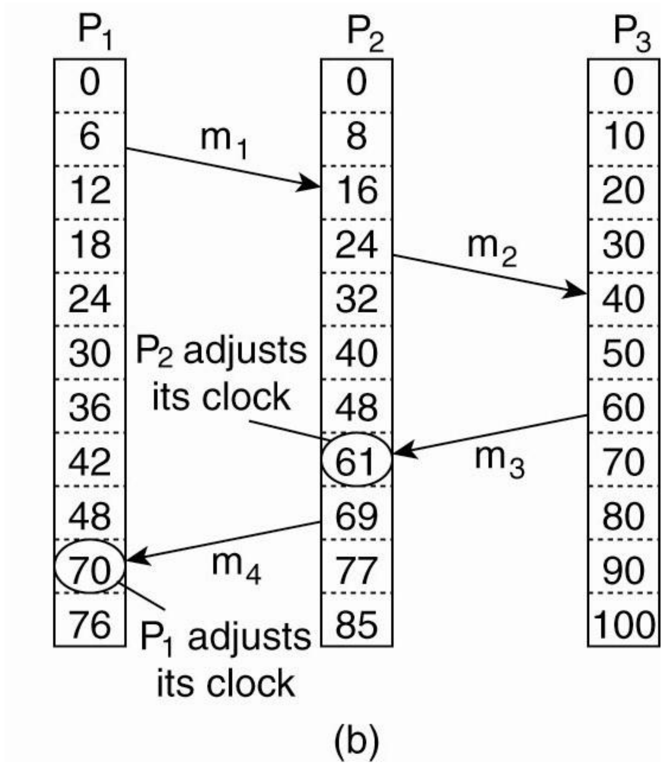
Lamport Timestamps

- When message arrives, if process time is less than timestamp s, then jump process time to s+1
- Clock must tick once between every two events
  - If $A \rightarrow B$ then must have $L(A) < L(B)$
  - If $L(A) < L(B)$, it does NOT follow that $A \rightarrow B$

Lamport Timestamps – example

[Tanenbaum]

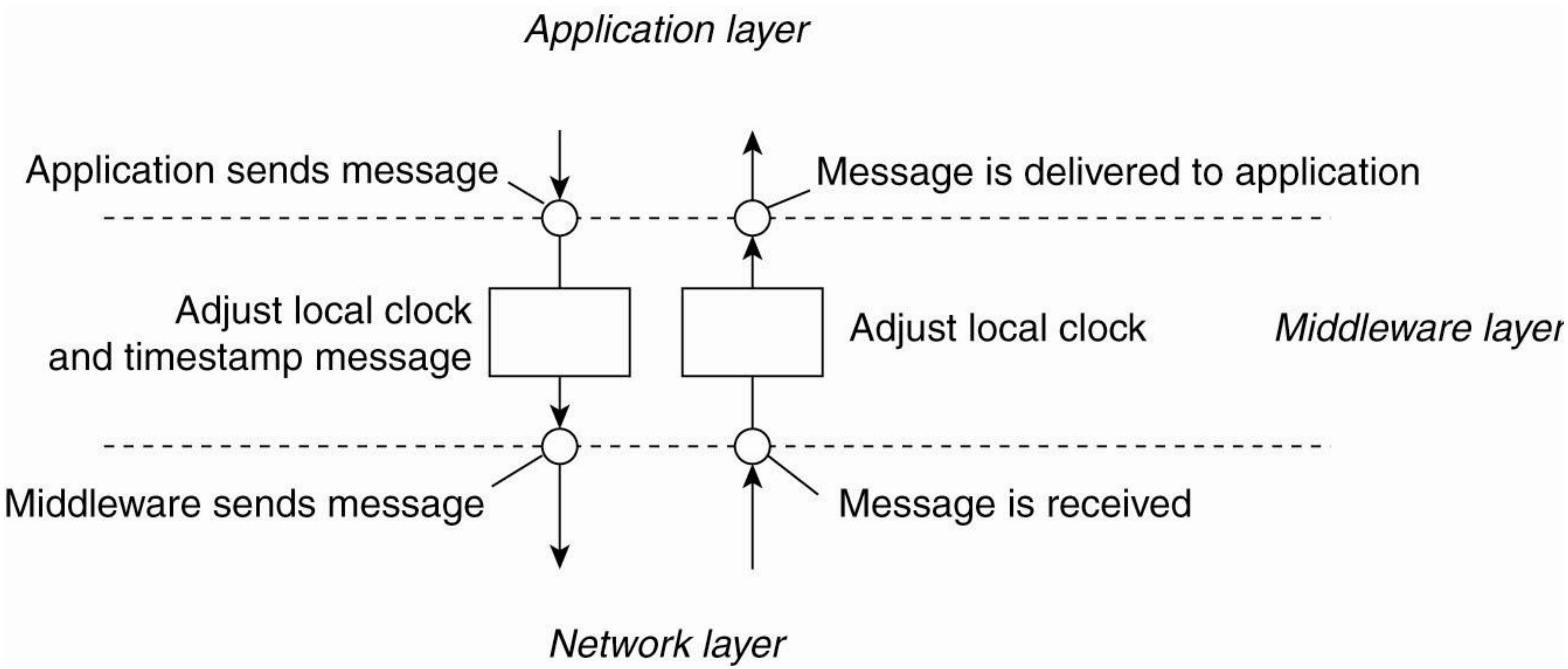




- 3 processes, each with its own clock
- The clocks run at different rates
- Lamport's algorithm corrects the clocks

Where does it happen?

- The positioning of Lamport's logical clocks in distributed systems



Lamport Timestamp limitation…

- L(A) < L(B) doesn't tell you that A came before B
- Only incorporates intrinsic causality, ignores any relationship with external clocks or external events

Vector Clocks

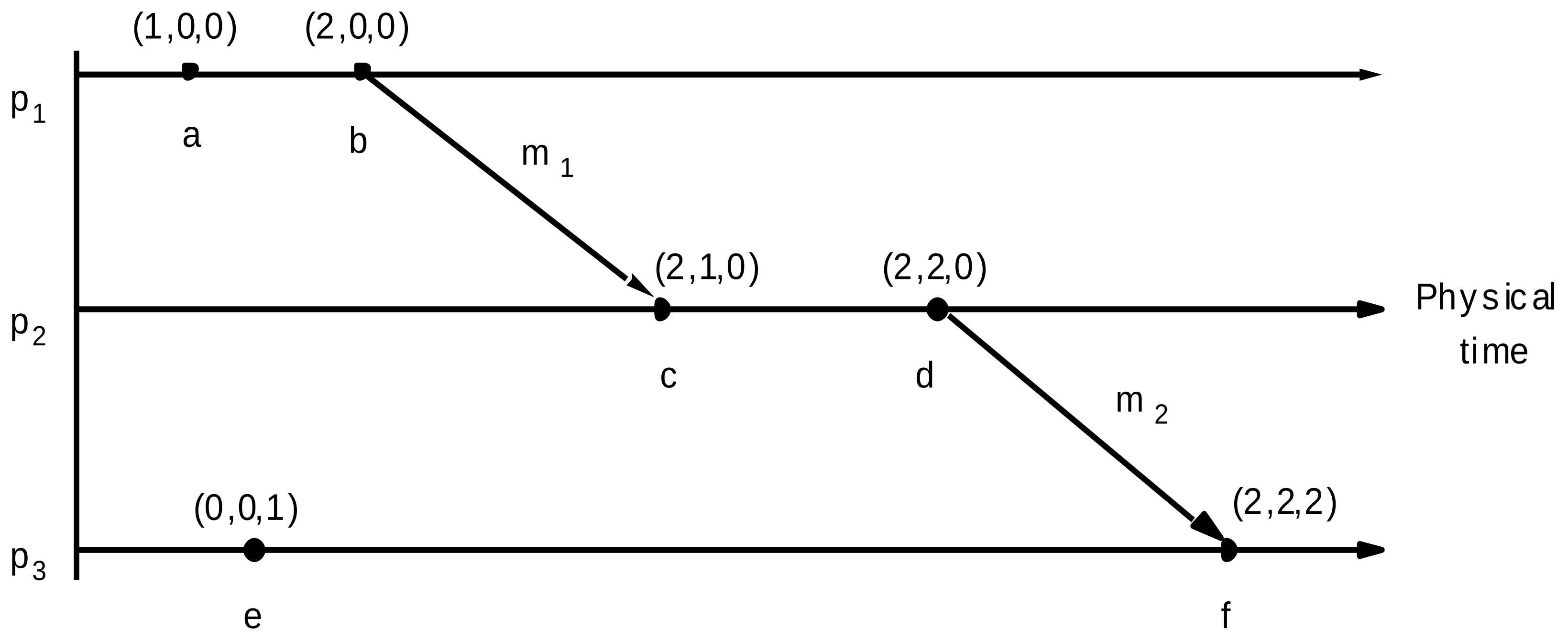
- For *n* processes, every process *i* contains a vector VC with *n* entries, initially all 0
- To every event *e* a *vector time* VC(*e*) is attached
- Idea: each element *VC(e)*[*j*] represents the number of events that preceded *e* on processor *j*
- Rules for change of VC on processor *i*:
  - ➢ If *e* is an internal or send event it applies

    VC[*i*] := VC[*i*] +1, all other entries remain equal
  - ➢ If *e* is a receive event it applies

    VC := max{VC, send vector time of the message}

    VC[i] := VC[i]+1
- **Claim:** Vector clocks allow to reconstruct the entire causal structure of an execution (e.g. concurrency of events)
  - ➢ Formally: VC(e) < VC (e') **if and only if** e $\rightarrow$ e' !

Vector Timestamps



Comparing Vector Clocks

- Two vector clocks are identical if they are identical in all components:

$$VC(e) = VC(e') \text{ iff}$$

$$\forall k \in 1, \ldots, n : VC(e)[k] = VC(e')[k]$$

- One vector clock is smaller than or equal to another if all components are smaller or equal:

$$VC(e) \leq VC(e') \text{ iff}$$

$$\forall k \in 1, \ldots, n : VC(e)[k] \leq VC(e')[k]$$

- Vectors clocks are smaller if they are (smaller or equal) and unequal:

$$VC(e) < VC(e') \text{ iff } VC(e) \leq VC(e') \wedge VC(e) \neq VC(e')$$

- Two events are independent (or concurrent) if neither
  VC(e) < VC(e') nor VC(e') < VC(e)