Technische Universität Berlin

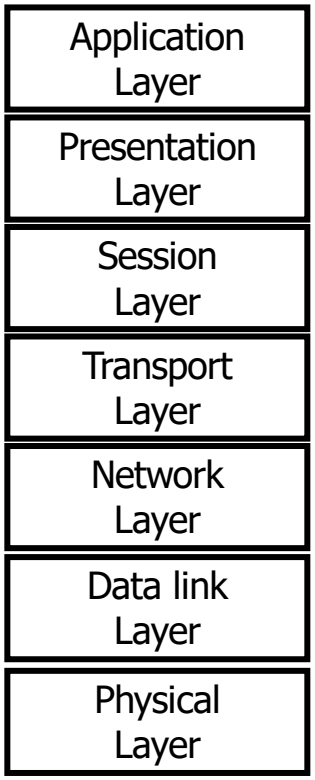
Computer Networks

Time Synchronization

Chapter

1. Introduction
2. Protocols
3. Application layer
4. Web services
5. Distributed hash tables
6. **Time synchronization**
   - **Physical clocks**
   - **Synchronization protocols**
   - **NTP**
   - **Logical clocks**
7. Transport layer
8. UDP and TCP
9. TCP performance
10. Network layer
11. Internet protocol
12. Data link layer

Top-Down-Approach



Time Synchronization

Communication & Interaction in Distributed Systems

- So far, we have looked at processes and communication between processes,
- But communication is just half of the story
  - Interaction is a more general issue!
  - Mastering (inter) action is a fundamental issue in (distributed) systems (DS),
  - **Doing the right thing "at the right time" is essential**
- Today: we will look at how processes cooperate and synchronize with one another

Interaction in Distributed Systems

- Examples:
  - Multiple processes should **not simultaneously access** a shared resource, e.g. printer → cooperate in granting each other temporary exclusive access
  - Multiple processes need to agree on the **ordering of events**, e.g. message *m*1 from process P was sent before or after message *m*2 from process Q

Synchronization in Distributed Systems

- But synchronization in distributed systems is generally much harder to achieve compared to synchronization in uni-/multiprocessor systems

- Two approaches will be discussed:
  - Synchronization based on **actual time**,
  - Synchronization in which only **relative ordering** matters → no need for absolute time

Time in Distributed System

- Synchronisation
  - Is there a notion of time in a distributed system?
  - Is there a notion of **global time** in a distributed system?
  - If not, what can we do about this?
  - How can we **synchronize activities** within a distributed system?

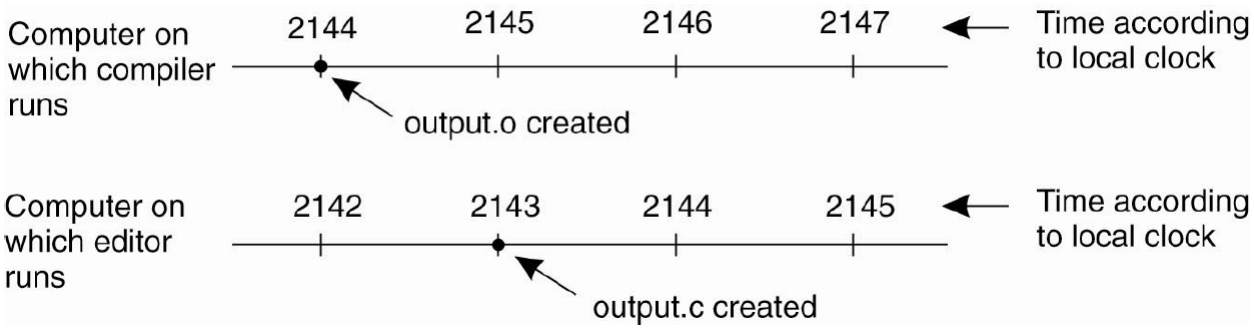
The Issue of Time

- Time in distributed systems (DS)
  - In centralized systems, time is unambiguous,
    - E.g. two processes asking for the time
  - In a distributed system, there is not a **natural notion** of time
  - Is it **possible** to build up a global notion of time in any distributed system?
  - Is it **useful** to build up a global notion of time in any distributed system?

Physical Time

Clock Synchronization

- Implications of the **lack of global time** in a DS on UNIX *make* program:
  - When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time!
    - E.g., *output.o* has time 2144, and shortly thereafter *output.c* is modified but assigned time 2143 because the clock on its machine is slightly behind → *make* will not call compiler



Physical Clocks

- Question: Is it possible to synchronize all the clocks in a distributed system?

oscillating quartz

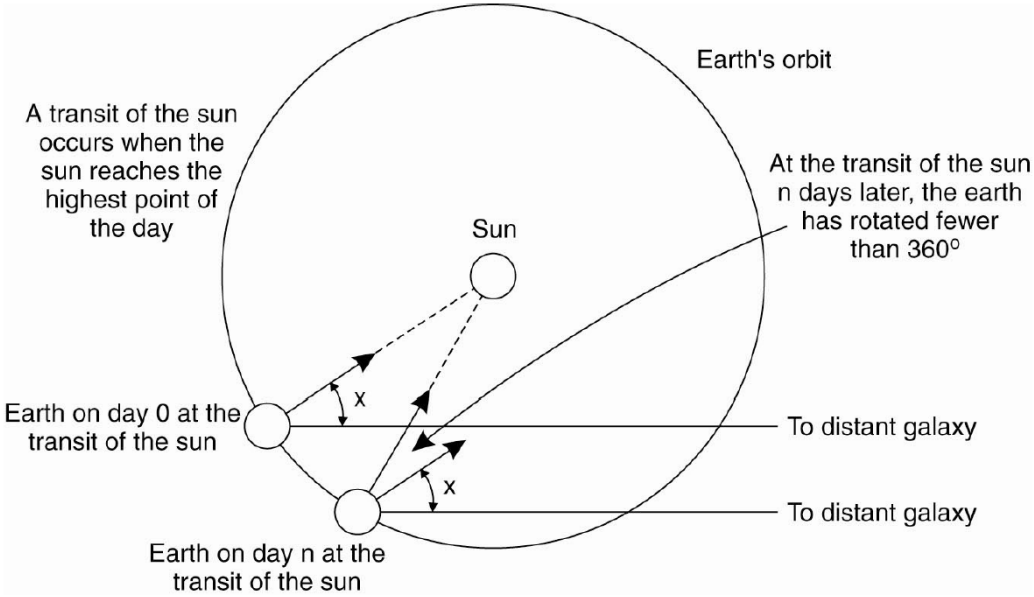
- Timers
  - A **clock** in a computer is actually a **timer**, i.e., typically, a precisely oscillating **quartz** with a counter and a holding register,
  - When the counter gets to zero, an interrupt is generated, and the counter is reloaded from the holding register,
  - Each interrupt is a clock tick,
  - At every clock tick the interrupt service procedure adds one to the time stored in memory → software clock is kept up-to-date

Physical Clocks (II)

- Multiple CPUs (each with its own clock)
  - No way to ensure two different crystals oscillate exactly at the same frequency
  - Different clocks gradually get out of synch – **clock skew** is the difference in time
  - Need for **synchronizing algorithms**!
  - Two approaches
    - global absolute time → actual clock time is important (e.g. real-time systems) → synchronize with real-world clocks
    - global relative time

Global Absolute Time

- How time is actually measured?
  - Not trivial if high accuracy is needed
- <17th century: time is measured **astronomically**
- Computation of the mean **solar day**
  - Solar second = 1/86400th of a solar day



Global Absolute Time (II)

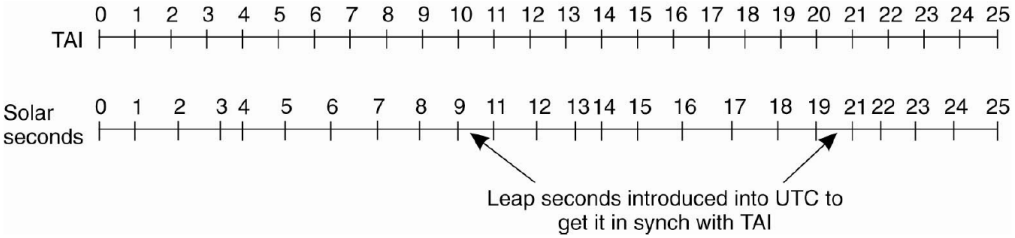
- 1948: invention of atomic clock
  - … counting the transitions of the cesium 133 atom,
  - 1 sec = 9192631770 periods of transition for cesium 133
- Absolute time is handled by BIH (Bureau International de l'Heure) :
  - International Atomic Time (TAI),
  - TAI = mean number of ticks of cesium 133 clocks since midnight of 1.1.1958
- **Problem**: TAI seconds are of constant length, unlike solar seconds
- **Solution**: leap seconds are introduced when necessary to keep in phase with the sun

src: wikipedia

Leap second as it would appear on a digital clock.

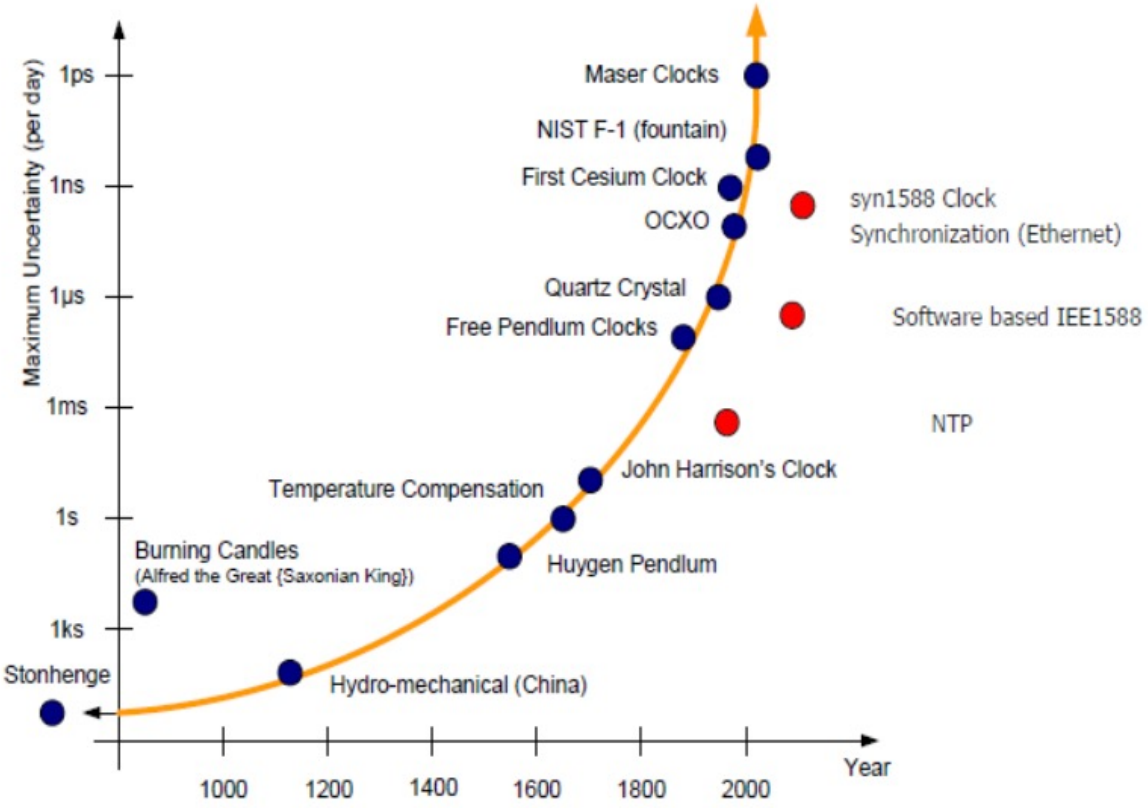
Global Absolute Time (III)

- Expressed in terms of **Universal Coordinated Time (UTC)**
  - Leap seconds are occasionally inserted into UTC to get it in synch with TAI & keep in step with astronomical time



- **UTC** is **provided** to people who need precise time
  - Time signal is broadcasted as a short radio pulse (WWV) by NIST (National Institute of Standard Time) every UTC second, and by satellites providing UTC service
- If one machine in the system has access to an UTC service, an algorithm can be used that synchronizes all machines based on this

How precisely can we measure time?



[Oregano systems, op. cit.]

Clock Synchronization Algorithms

- If one machine in the system has access to an UTC service, the goal becomes keeping all the other machines synchronized to it,
- If no machine has access to an UTC service, each machine keeps track of its own time → goal is to keep all the machines together as well as possible
- Many algorithm for synchronization proposed:
  - All have the same underlying model of the system → next slide

Model of the System

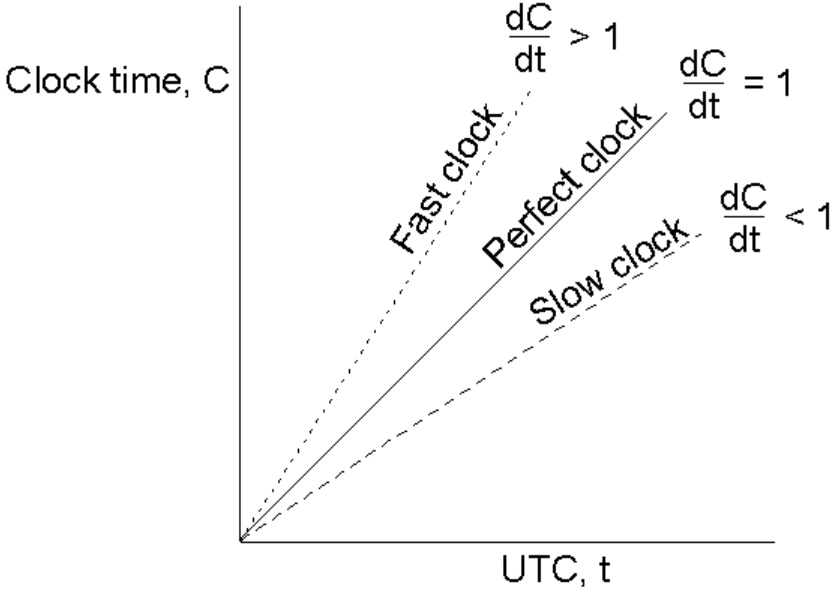
- Each machine *i* has a local clock
- $H_i(t)$: hardware clock value (by oscillator, discontinuous)
- $C_i(t)$: software clock value (generated by operating system)
  - $C_i(t) = r\,H_i(t) + A$, a **tick** occurs every so many quartz oscillations!
  - t is the UTC time,
  - Clock resolution: period between updates of $C_i(t)$ → limit on determining order of events
  - Perfect world: $C_i(t) = t$ for all i and all t,
- $C_i(t)$: **approximation of the UTC** - # nsec's elapsed at time t since a reference time

Model of the System (II)

- 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 ordinary (inexpensive) quartz → 1 second per 11.6 days

Clock Drift

- The relation between clock time and UTC when clocks tick at different rates,
- Drift must be bounded by parameter ρ
  - More precisely: $1-\rho \leq dC(t) / dt \leq 1+\rho$
- Parameter ρ is known to be the **maximum drift rate** being specified by manufacturer



The Problem → The Approaches

- The notion of accurate time is very problematic …
  - We are limited in our ability to timestamp events at different nodes (=machines) 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** (later)

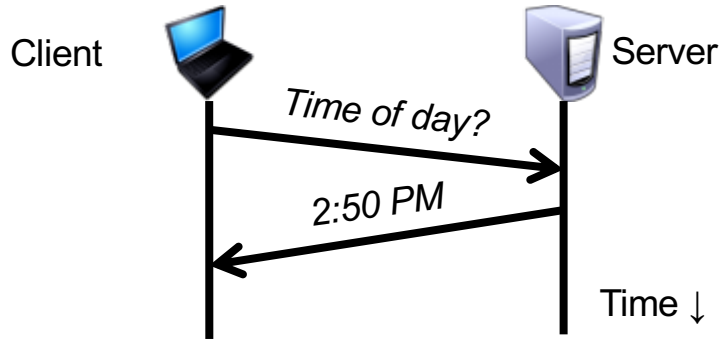
The story of the twin brothers

- On November 6, 2016, two twin brothers were born in the US
- Samuel was born first at 1:39am in the morning
- His brother Ronan was born 31 minutes later
- Exactly at 2am, all clocks were corrected to 1am (daylight savings time); thus, Ronan was officially born at 1:10am
- **Ronan is therefore 29 minutes older than his first-born brother**

Synchronization protocols

Synchronization to a Time Server

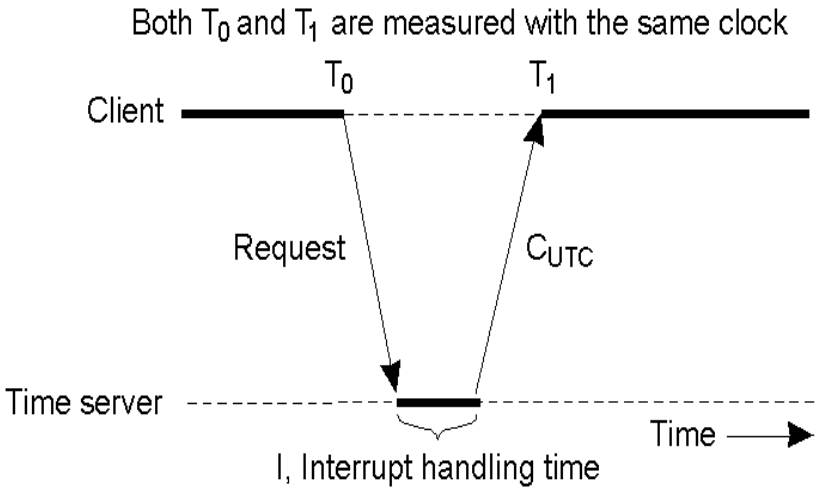
- Suppose a server with an accurate clock (e.g., GPS-receiver)
  - Could simply issue an RPC to obtain the time:



- But this doesn't account for network latency
  - Message delays will have **outdated** server's answer

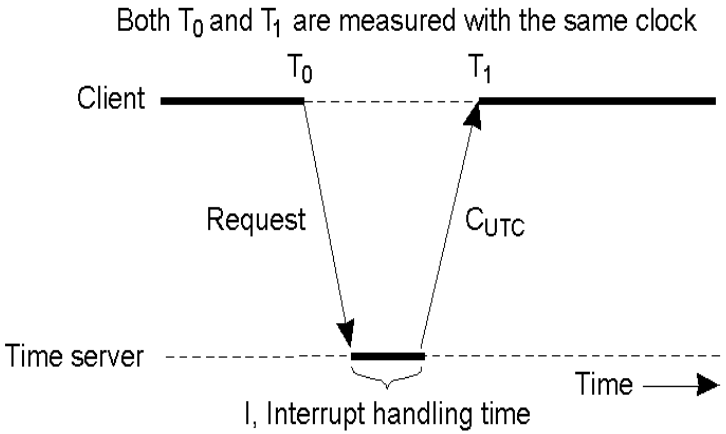
Cristian's Algorithm

- Well suited to systems in which one machine has access to an UTC service → **time server**,
- Goal is to have all the other machines **stay synchronized** with it,
- Periodically each machine sends a message to the time server asking for the current time,
- Time server responds as fast as it can with a message containing its current time, C_{UTC}



Cristian's Algorithm (II)

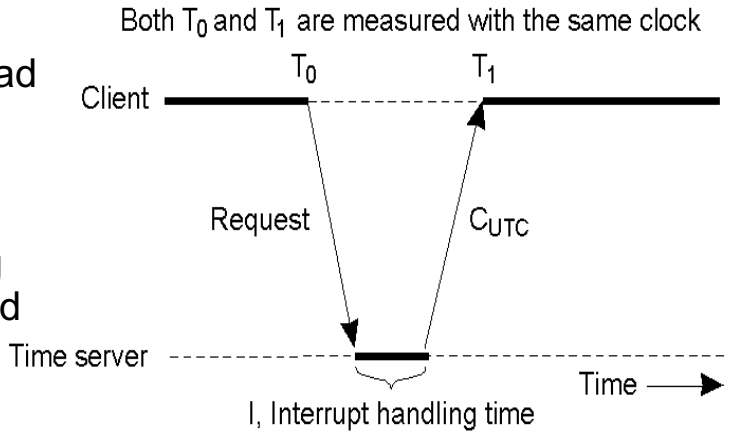
- First approximation:
  - When sender gets reply: just set its clock to C_{UTC}
  - **Problem I**:
    - Time must never run backward,
    - If sender's clock is fast, C_{UTC} will be smaller than its current value of C,
    - **Change** in C must be introduced **gradually**!
    - Possible approach: interrupt routine adds less amount of time,
    - Avoid also jumping in case of slower clock



Cristian's Algorithm (III)

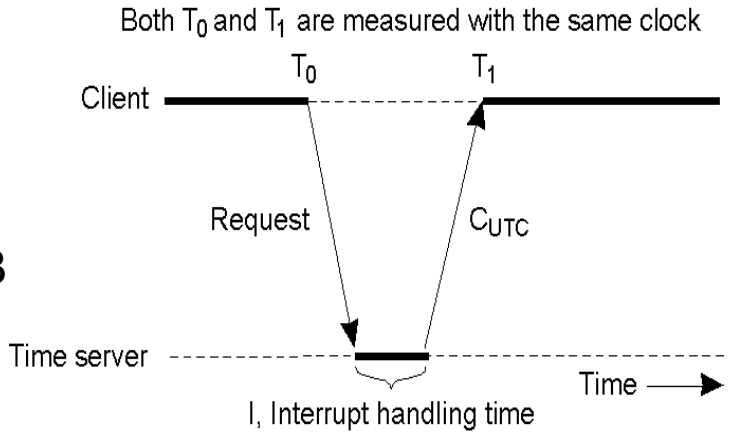
- First approximation:
  - **Problem II**:
    - It takes a **nonzero amount of time** for the time server's reply to get back to sender → network!
      - Delay may be large & vary with network load
    - Algorithm tries to **measure delay**:
      - Sender records interval between sending request and arrival of reply → both starting time, T_0, and ending time, T_1, are measured using **same clock**
    - In absence of any other information: best estimate of message propagation time is $(T_1 - T_0)/2$
    - → Estimate of the server's current time

= value in reply message + $(T_1 - T_0)/2$



Cristian's Algorithm (IV)

- **Improving time estimate**:
  - Possible if we know approx. how long it takes the time server to handle the interrupt and process the incoming message:
    - Let *I* be the interrupt handling time,
    - Best estimate of one-way propagation time: $(T_1 - T_0 - I)/2$
  - **Note**: assumption that messages from A to B take same route as those from B to A,
  - **Filtering**:
    - Collect multiple measurements → discard outliers (victims of network congestion) → use average value or take message arriving fastest

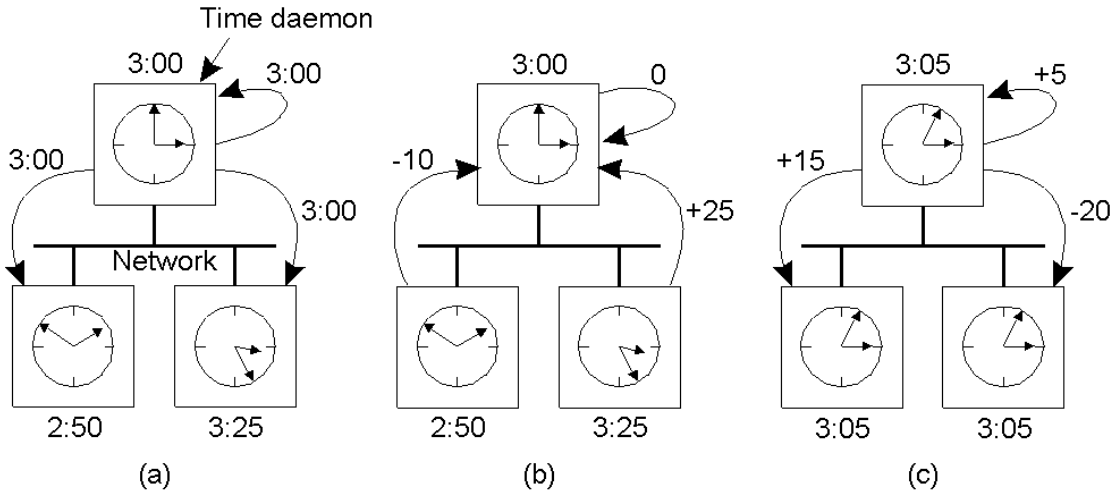


The Berkeley Algorithm

- In Cristian's algorithm the time server is passive, i.e., all other machines periodically ask it for the time,
- In Berkeley algorithm the opposite approach is taken:
  - **Time server** (daemon) is active, **polling** every machine from time to time to ask what time it is there,
  - Based on answers it computes the **average time**,
  - Tells all the other machines to **advance** their clocks to new time or **slow** their clocks down until some specified reduction has been achieved,
- Suitable for a system in which no machine has access to an UTC service

The Berkeley Algorithm (II) - Example

- E.g., synchronization in LAN



step (c): 3:00 → 180, 2:50 → 170, 3:25 → 205

average clock = (180+170+205) / 3 = 185 → **3:05**

The Berkeley Algorithm (III)

- Coordinator (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 → elimination of the outliers
- Election of new master, in case of failure → robustness

NTP

What is Network Time Protocol (NTP)?

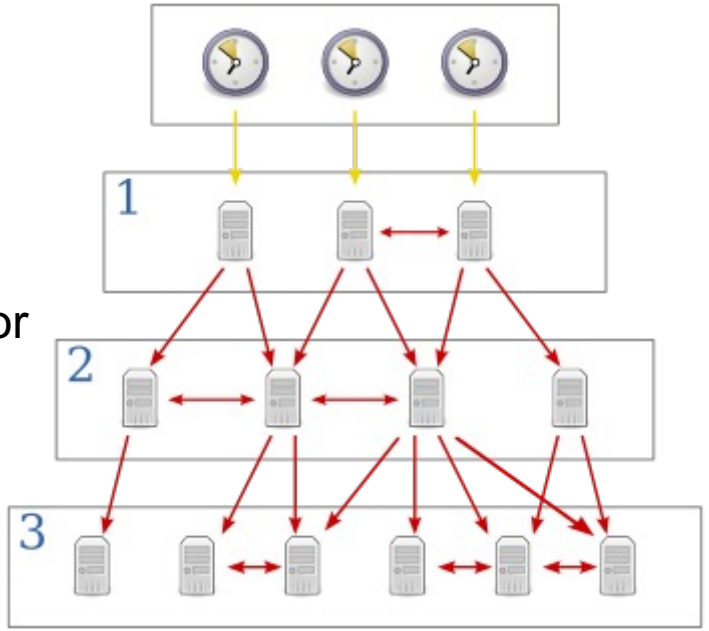
- Time synchronization system for computer clocks through the Internet – most widely used,
- Enables clients to be accurately synchronized to UTC despite message delays … with (worldwide) accuracy of 1-50 ms,
- Internet standard protocol
  - Version 3 (RFC-1305), simple NTP version 4 (RFC-2030)
  - Application Layer (OSI-layer 7) using UDP (port 123)
  - Clients exist for almost all platforms (Windows, Linux, etc.)
- Provides mechanisms to
  - Synchronize clocks to some reference time
  - Coordinate time distribution in a large, diverse Internet

Network Time Protocol - 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** → leverages heterogeneous accuracy in clocks
  - Multiple candidates can be combined to minimize the accumulated error
  - Detects and avoids insane time sources
  - Survives lengthy losses of connectivity
- 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 - Clock Strata

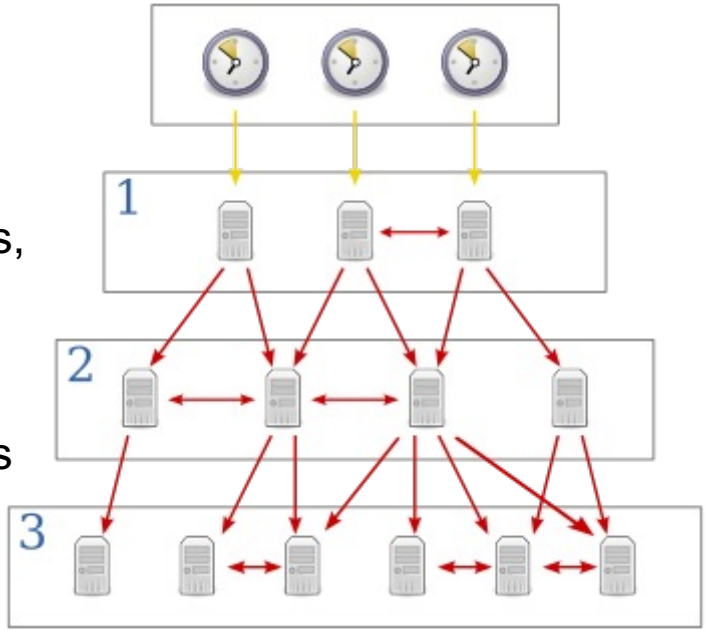
- NTP uses a **hierarchical system** of time sources:
  - Each level is termed a **stratum** and is assigned a number starting with zero for the reference clock at the top,
  - A server synchronized to a stratum *n* server runs at stratum *n+1*,
  - The number represents the distance from the reference clock,
  - Stratum is not always an indication of quality or reliability



source: wikipedia

Network Time Protocol - Clock Strata (II)

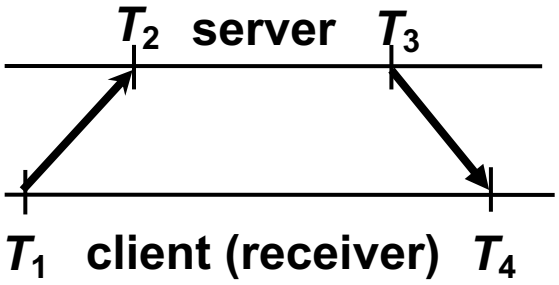
- **Stratum 0** = reference clocks
  - high-precision time sources themselves, e.g,. atomic clocks, GPS, …
- **Stratum 1** = primary time servers
  - are computers directly (no network!) connected to stratum 0 devices
- **Stratum 2**
  - are computers that are synchronized over a network to stratum 1 servers,
  - stratum 2 servers are clients of stratum 1 servers,
- **Stratum 3**
  - NTP servers that synchronize with stratum 2
  - stratum 3 servers are clients of stratum 2 servers
- Users' computers synchronize with stratum 3 servers



source: wikipedia

How does NTP work? - Time Exchange

- NTP client will regularly poll one or more NTP servers
  - Exchange of several packet pairs (request, reply)
  - Containing originate resp. receive timestamp
- To synchronize its clock, the client must compute its **time offset** and **round-trip delay** (travelling time)



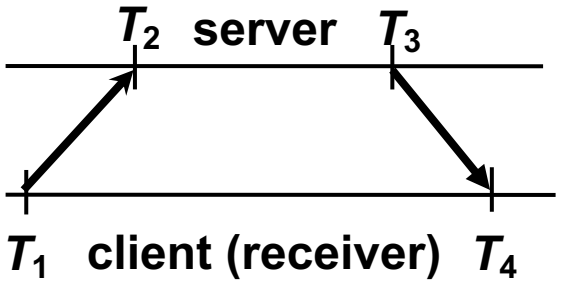
[Note : Tanenbaum's delay $= \delta / 2$]

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

Delay $\delta = (T_4 - T_1) - (T_3 - T_2)$

NTP - 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



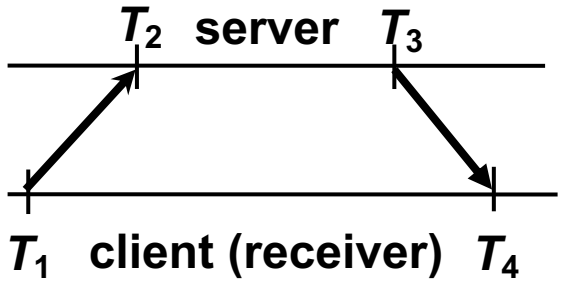
Eq.:
Delay δ = [(T4-T1) -(T3-T2)]

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

NTP - Time Exchange Example (II)

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

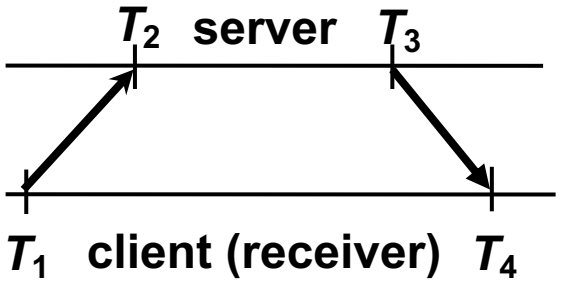


Eq.:
Offset Θ = 0.5[(T2-T1) +(T3-T4)]

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

NTP - Time Exchange Example (III)



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 delay 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 Operation: Server Selection

- Examine values from several peers (=servers) 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 dispersion** → next slide
- May modify local clock update frequency w.r.t. observed drift rate
- Accuracy:

  ~ 10s of milliseconds over Internet paths

  ~ 1 millisecond on LANs

NTP Operation: Server Selection (II)

- Messages between an NTP client and server are exchanged in pairs: request and response (like Cristian's algorithm)
- For i^{th} message exchange with a particular server, calculate:
  1. **Clock offset** θ_i from client to server
  2. **Round trip time** δ_i between client and server
- Over last eight exchanges with server k, the client computes its **dispersion** $\sigma_k = \max_i \delta_i - \min_i \delta_i$
  - Client uses the **server** with **minimum dispersion**
- Then uses a best estimate of clock offset

Clock Synchronization: Take-away points

- Clocks on different systems will always behave differently
  - Disagreement between machines can result in undesirable behavior
- NTP clock synchronization
  - Rely on timestamps to estimate network delays
  - 100s μs−ms accuracy
  - Clocks never exactly synchronized
- Often inadequate for distributed systems
  - Often need to reason about the order of events
  - Might need precision on the order of ns

Logical Time: Lamport Clocks

Motivation: Multi-site Database Replication

- A New York-based bank wants to make its transaction ledger database resilient to whole-site failures
- **Replicate** the database, keep one copy in SF, one in NYC

[Lloyd, op. cit.]

The Consequences of Concurrent Updates

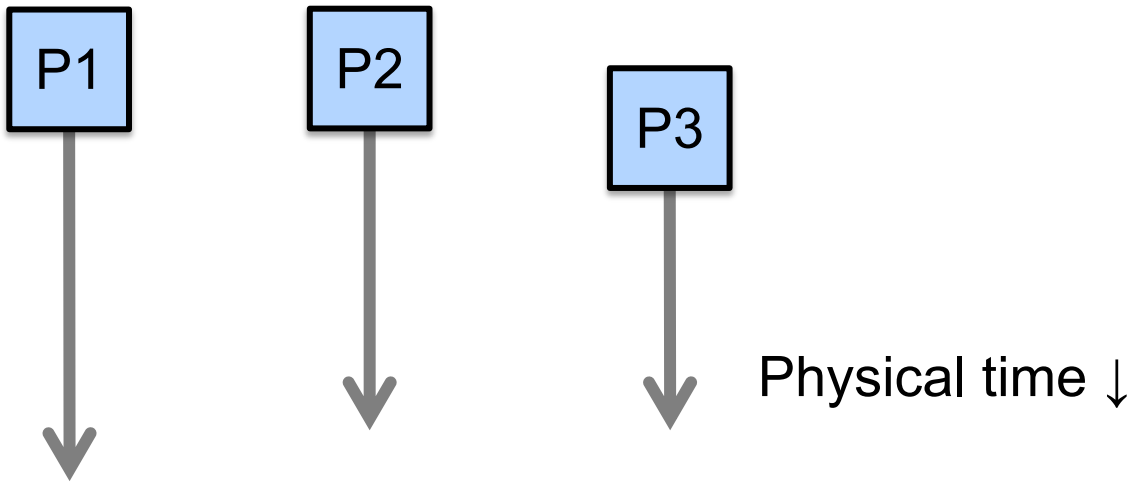
- **Replicate** the database, keep one copy in SF, one in NYC
  - Client sends query to the nearest copy
  - Client sends update to both copies

Logical Clocks

- For many purposes, it is sufficient that all machines **agree** on the **same time**; it does not need to be the real time,
  - Internal consistency of the clocks matters,
  - E.g., for make tool it is sufficient to know whether *input.c* is older or newer than *input.o*
- Logical clocks proposed by Leslie Lamport in 1978
- **Insight**: only the events themselves matter
- Basic **idea**:
  - Disregard the precise clock time,
  - Instead, capture just a **"happens before"** relationship between a pair of events

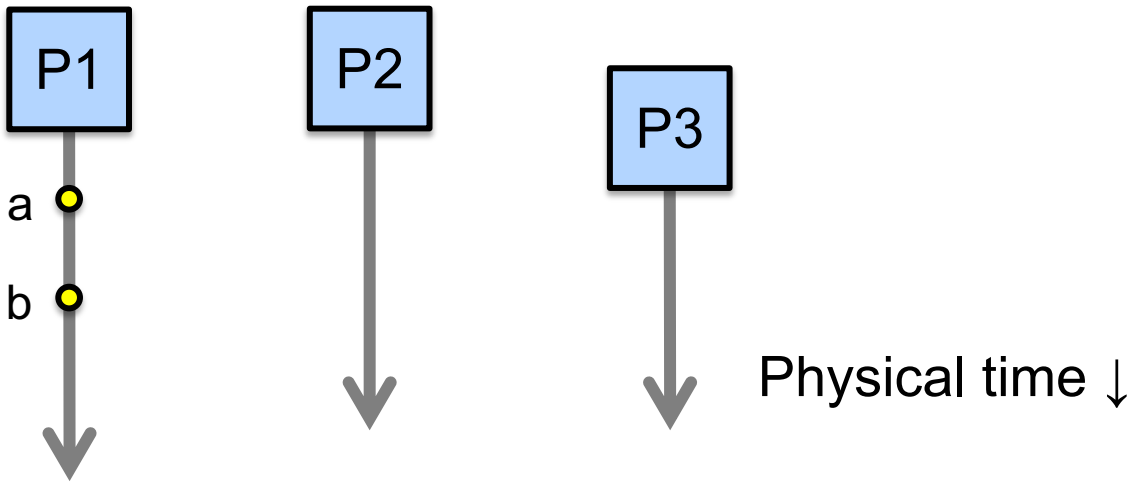
Defining “happens-before” (→)

- Consider three processes: P1, P2, and P3
- Notation: Event *a* happens before event *b* (*a* → *b*)



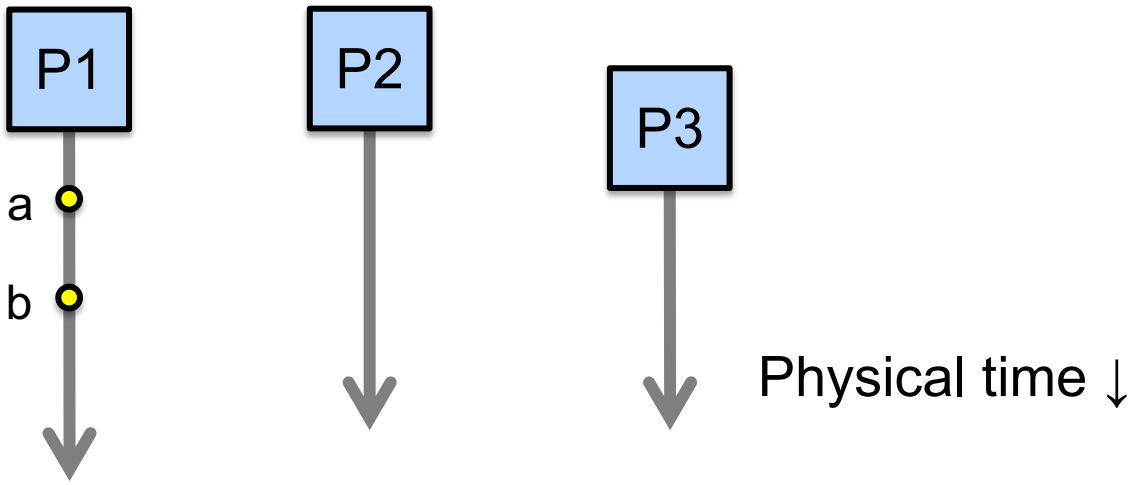
Defining “happens-before” (→)

- Can observe event order at a single process



Defining “happens-before” (→)

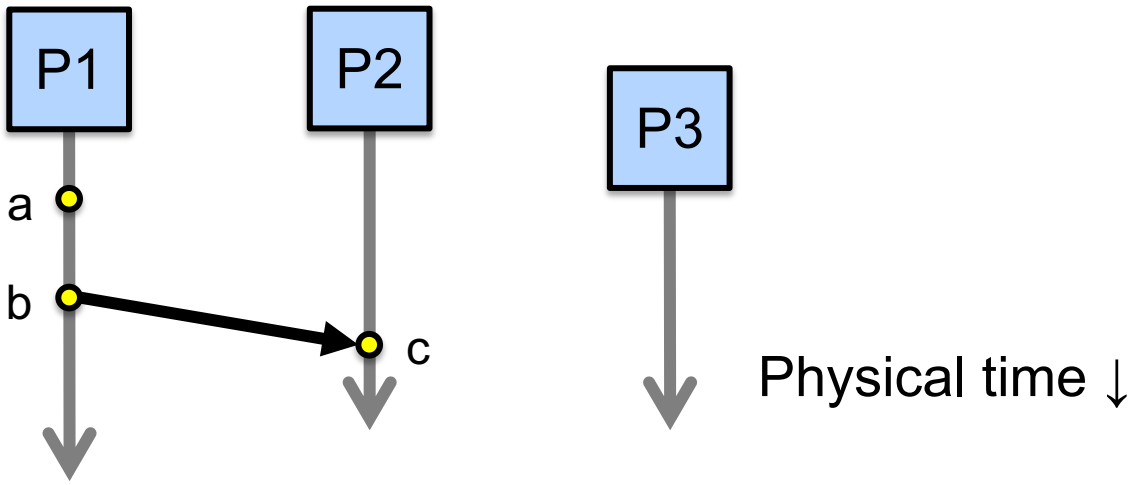
1. If **same process** and **a** occurs before **b**, then **a → b**



Defining “happens-before” (→)

1. If **same process** and **a** occurs before **b**, then **a** → **b**

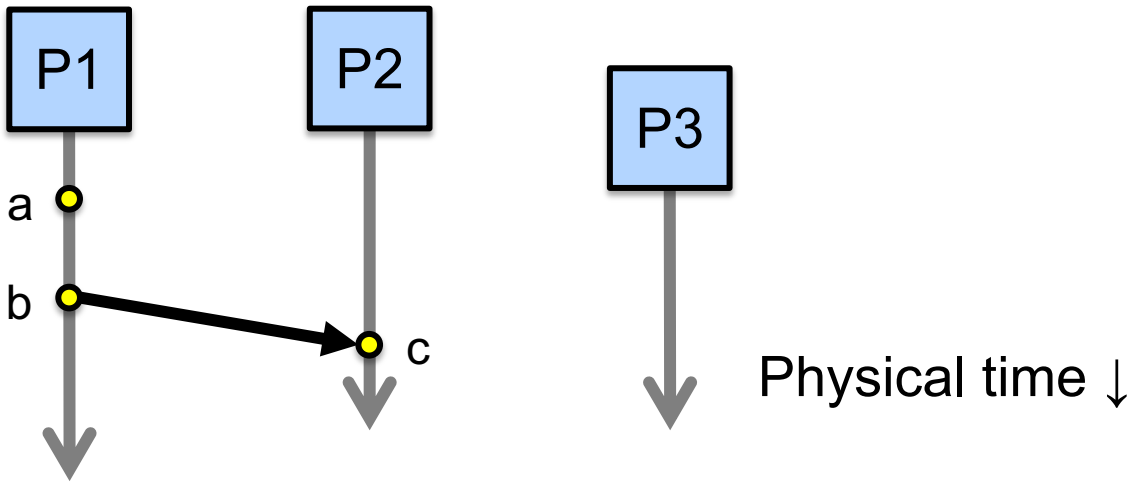
2. Can observe ordering when processes communicate



Defining “happens-before” (→)

1. If **same process** and **a** occurs before **b**, then **a** → **b**

2. If **c** is a message receipt of **b**, then **b** → **c**

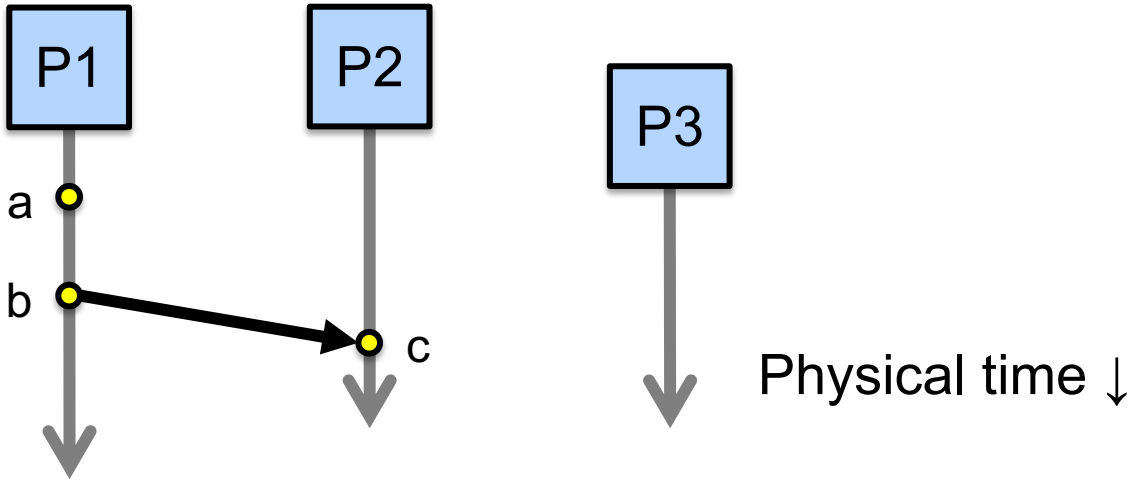


Defining “happens-before” (→)

1. If **same process** and **a** occurs before **b**, then **a** → **b**

2. If **c** is a message receipt of **b**, then **b** → **c**

3. Can observe ordering transitively

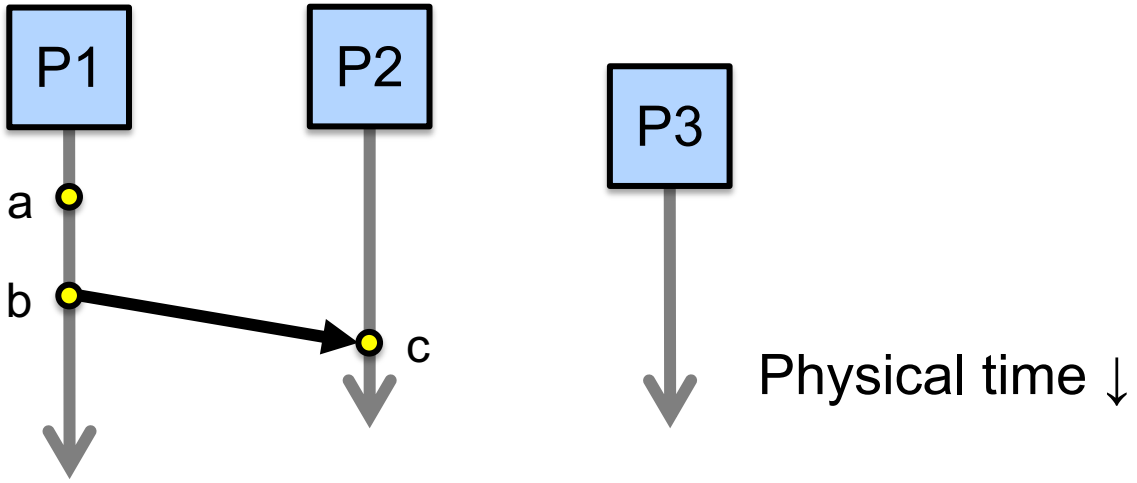


Defining “happens-before” (→)

1. If **same process** and **a** occurs before **b**, then **a** → **b**

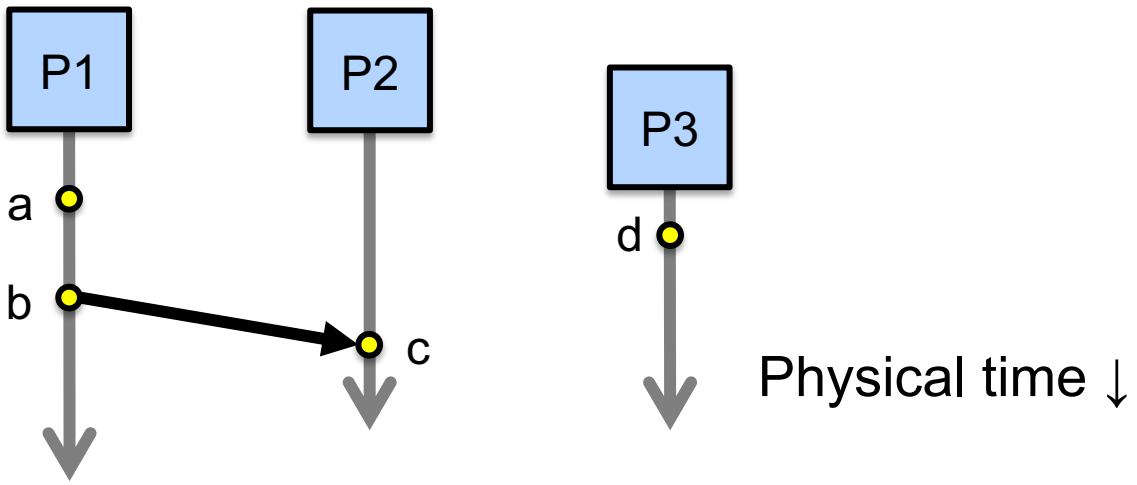
2. If **c** is a message receipt of **b**, then **b** → **c**

3. If **a** → **b** and **b** → **c**, then **a** → **c**



Concurrent events

- Not all events are related by →
- a, d not related by → so concurrent, written as a || d



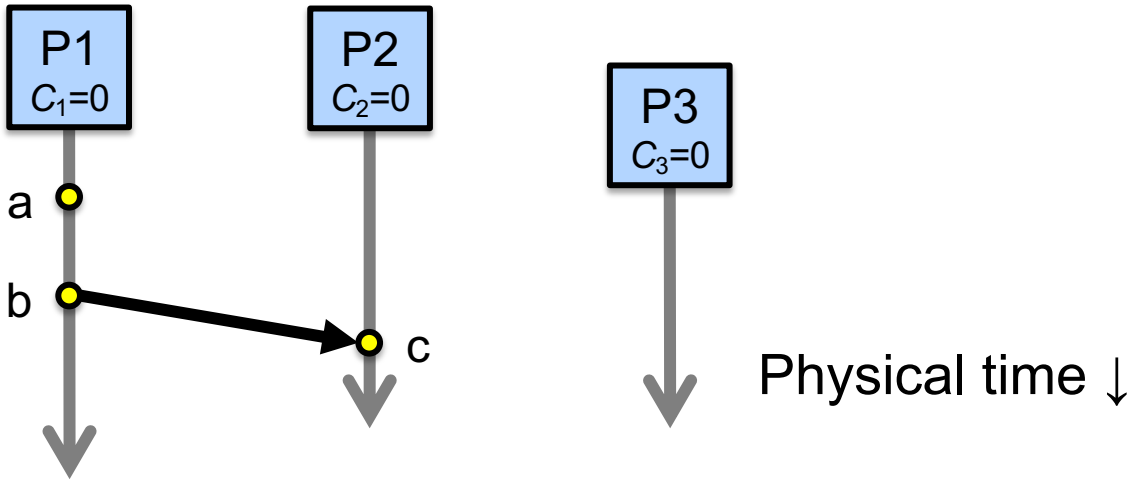
Lamport Clocks: Objective

- We seek a **clock time** *C(a)* for every event a on which all processes agree,
- Clock condition: If *a* → *b*, then *C(a)* < *C(b)*
- In addition: clock time, *C*, must always go forward!
  - Corrections to time: adding a positive value
- **Plan**: tag events with clock times; use clock times to make distributed system correct

The Lamport Clock algorithm

- Each process P_i maintains a local clock C_i

1. Before executing an event, $C_i \leftarrow C_i + 1$

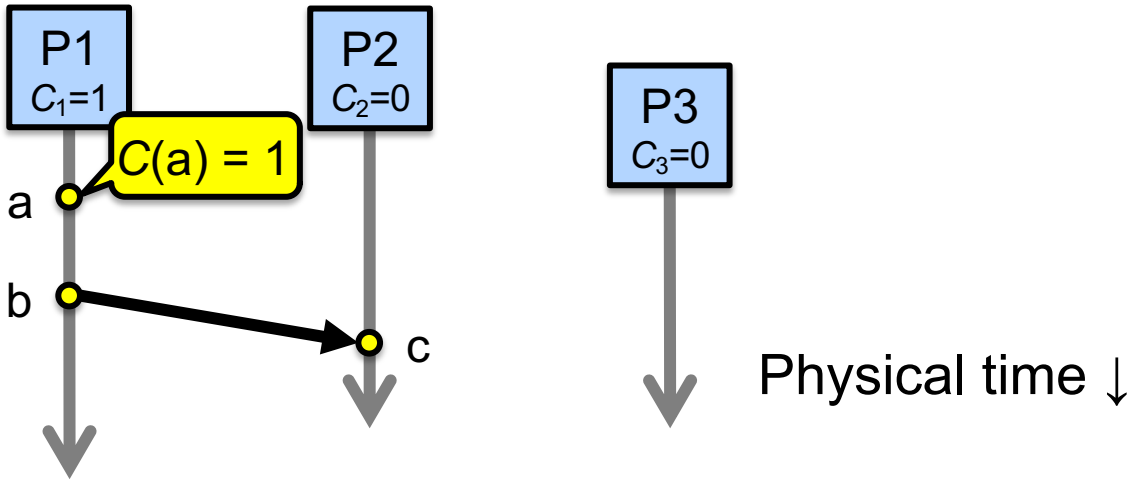


[Lloyd, op. cit.]

The Lamport Clock algorithm

1. Before executing an event a, $C_i \leftarrow C_i + 1$:

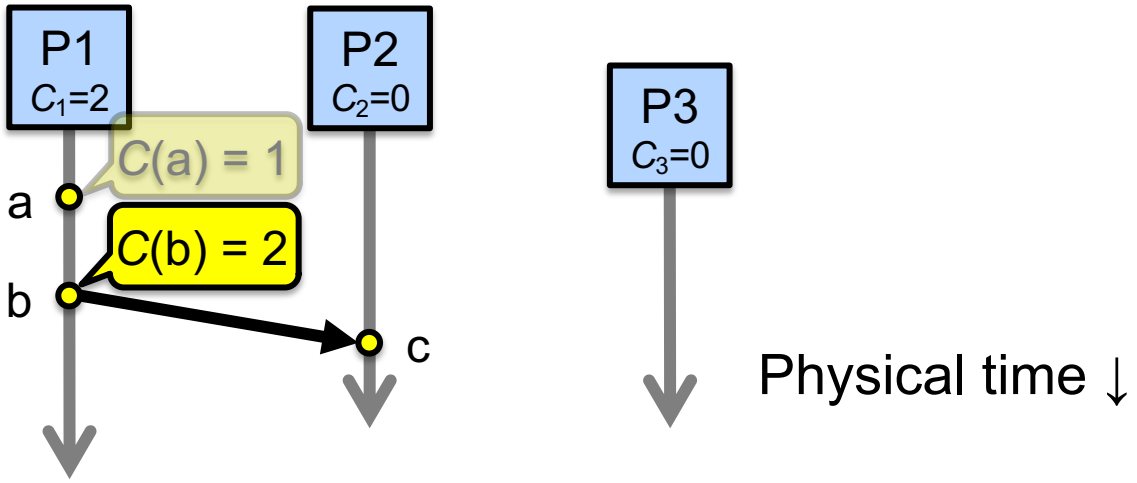
   - Set event time $C(a) \leftarrow C_i$



The Lamport Clock algorithm

1. Before executing an event b, $C_i \leftarrow C_i + 1$:

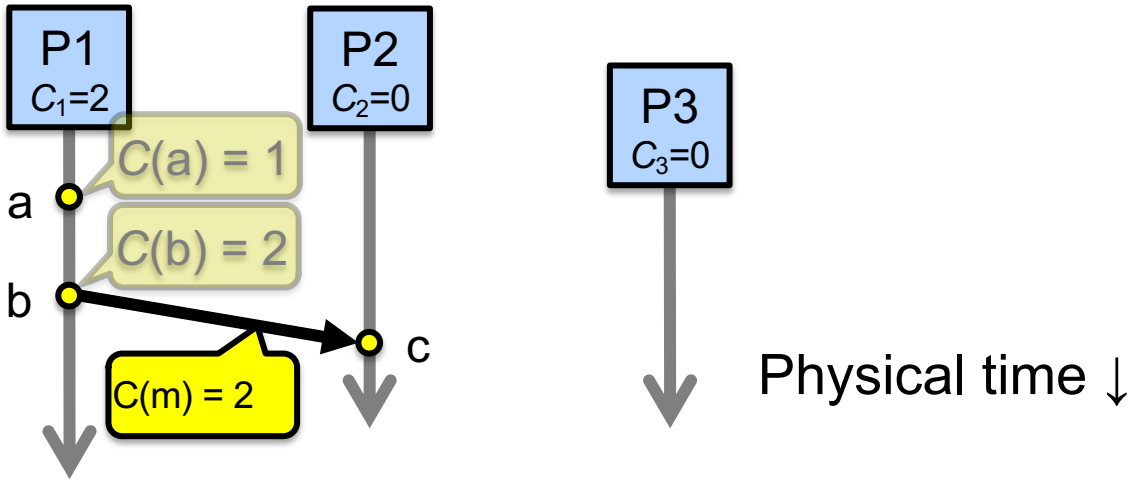
   - Set event time $C(b) \leftarrow C_i$



The Lamport Clock algorithm

1. Before executing an event b, $C_i \leftarrow C_i + 1$

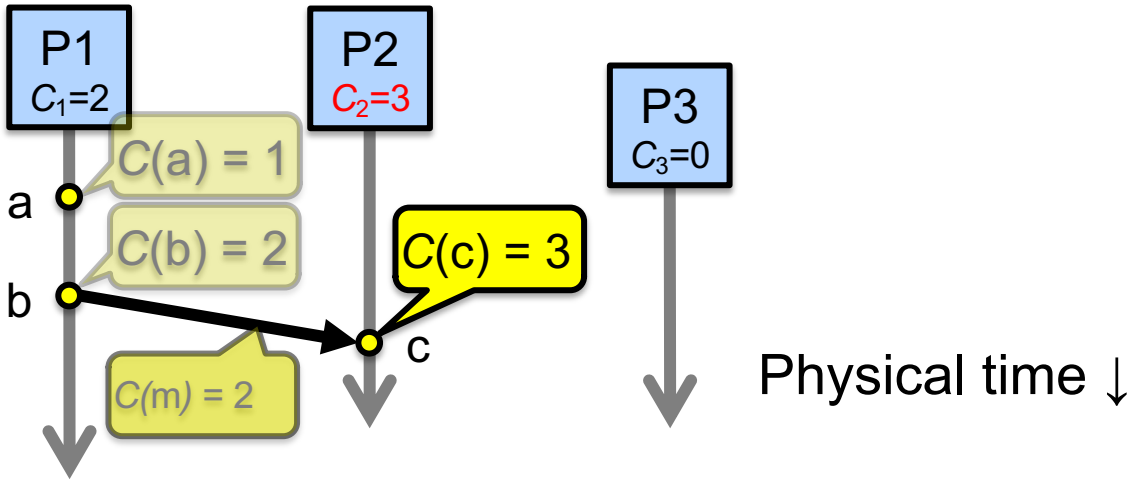
2. Send the local clock in the message m



The Lamport Clock algorithm

3. On process P_j receiving a message m:

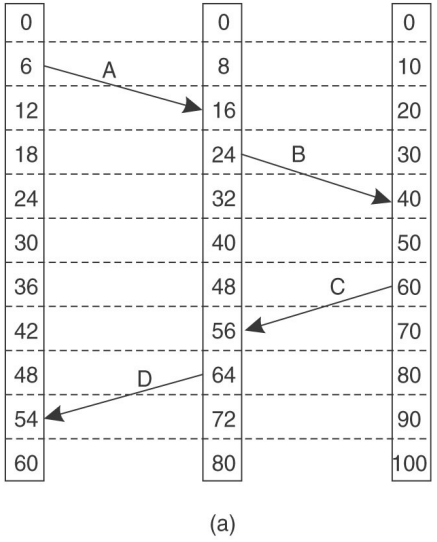
   - Set C_j and receive event time $C(c) \leftarrow 1 + \max\{ C_j, C(m) \}$



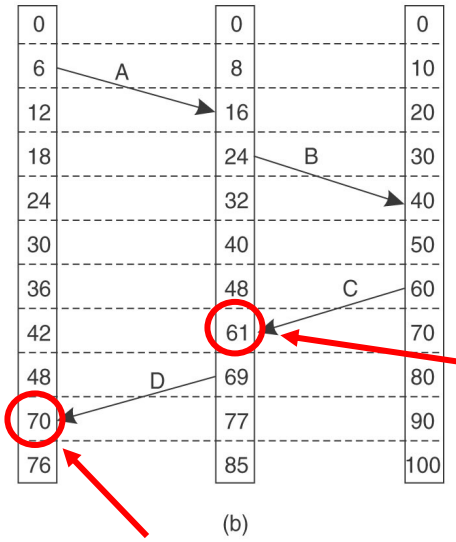
Lamport Clock algorithm – Another Example

(a) Three processes, each with **its own clock**; clocks run at different rates.

(b) Lamport's algorithm corrects the clocks.



(a)



(b)

P2 adjusts its clock

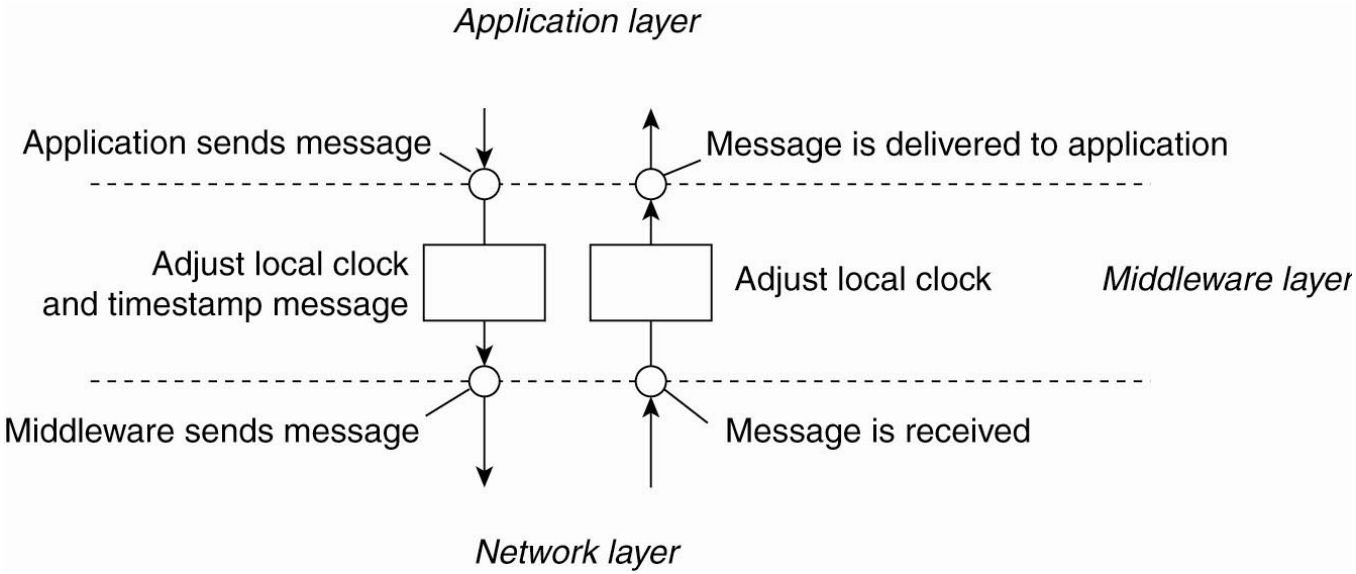
P1 adjusts its clock

Lamport Timestamps: Ordering all events

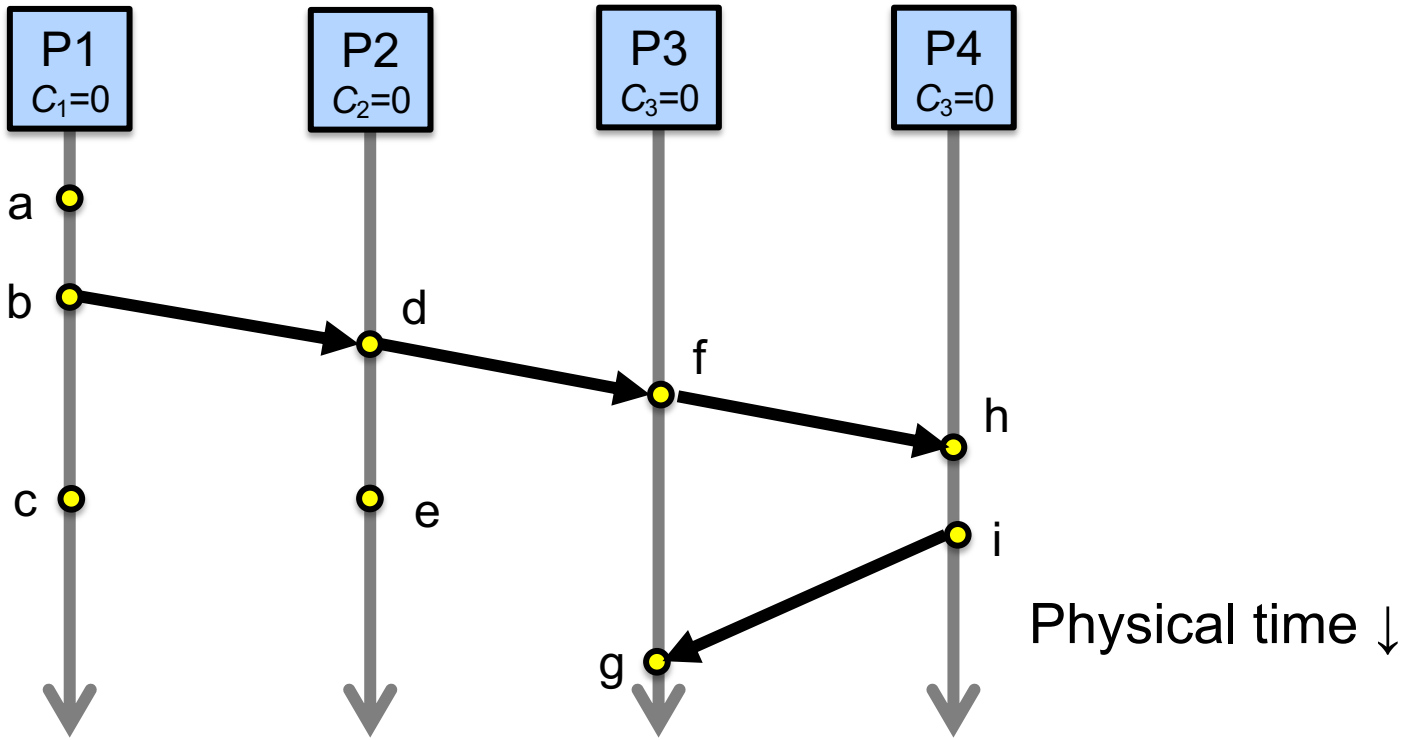
- Additional **requirement**:
  - No two events ever occur at exactly the same time
- **Solution**: appending the process number to each event:
  1. Process P_i timestamps event e with $C_i(e).i$
  2. $C(a).i < C(b).j$ when:
     - $C(a) < C(b)$, or $C(a) = C(b)$ and $i < j$
- Now, for any two events a and b, $C(a) < C(b)$ or $C(b) < C(a)$
  - This is called a **total ordering of events**
- **Example**:
  - Events happen in process 1 and 2, both with time 40, the former becomes 40.1 and the latter 40.2

Lamport Clocks as part of Middleware Layer

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



Homework: Order all these Events



Take-away points: Lamport Clocks

- Can totally-order events in a distributed system: that's useful!
  - We saw an application of Lamport clocks for totally-ordered multicast

- **But**: while by construction, a → b implies *C*(a) < *C*(b),
  - The converse is not necessarily true:
    - *C*(a) < *C*(b) does not imply a → b (possibly, a || b)

- We cannot use Lamport clock timestamps to infer causal **relationships between** events → see vector timestamps (Tanenbaum)