Clock Synchronization

527950-1
Fall 2019
11/21/2019
Kyoung Shin Park
Applied Computer Engineering
Dankook University

Physical Clocks vs Logical Clocks

- **Physical clocks keep time of day**
  - Consistent across systems
- **Logical clocks keeps track of event ordering**
  - Among related (casual) events

Global Clocks

- **Distributed systems have no global clock**
- Each processor in the system is autonomous
- Each processor has its own clock
- Impossible to have the processes across the system synchronized exactly
- Cannot know the true time order of any two events

Ordering of Events

- It is impossible to know **which of two events happens first**
- This has an **impact on scheduling**
- This makes the distributed system **harder to debug**

Global Time via Shared Memory?

- Distributed systems have **no shared memory**
- Thus it is hard (impossible) to get an up-to-date state of the entire system
- **A global state** would give us
  - A view of all local states
  - The contents of all messages currently in transit

Definitions

- **Drifting**:
  - "the gradual misalignment of once synchronized clocks caused by the slight inaccuracies of the time-keeping mechanisms"
  - Clock tick at different rates; create ever-widening gap in perceived time
- Drift rate:
  - "the change in offset (difference in reading) between the clock and a nominal perfect reference clock per unit time measured by the reference clock."
  - For clocks based on a quartz crystal, this is about 10^{-6}, giving a difference of one second every 1,000,000 seconds, or 11.6 days.
- **Clock Skew**:
  - "the difference in time between two clocks due to drifting"

Global Time via Physical Clocks?

- Problem: Sometimes we simply need the exact time
  - Solution: Universal coordinated time (UTC)
- **Universal Time Coordinator (UTC)**
  - Based on the number of 9,192,631,770 transitions per second of the cesium 133 atom (pretty accurate)
  - Accurate to +/- 1 second per 20,000,000 years
    - about 1 part in 10^{12}
  - Sources:
    - **Geostationary Operational Environmental Satellites (GEOS)**
    - **Global Positioning System (GPS) devices**
    - **WWV: a Fort Collins radio station**
    - **MSF: a British radio station**

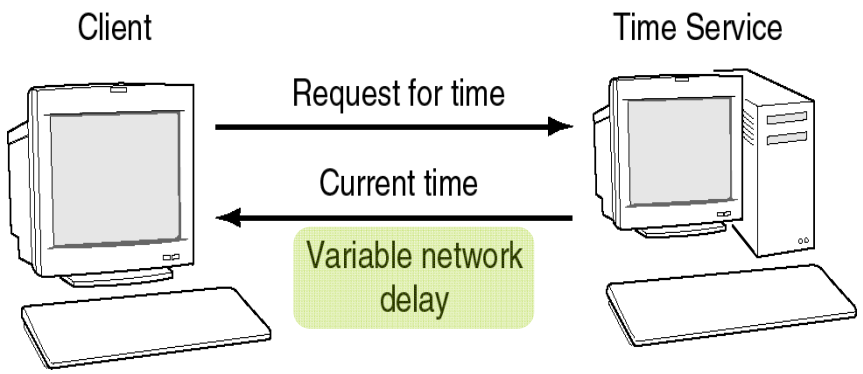
Global Time via WWV

- ***A Fort Collins shortwave radio station***
  - ***Transmits UTC signal***
  - *Low-frequency => less atmospheric disturbance*
  - *2000 mile radius*
  - *Sends signals once a day to clocks/watches*
  - *Transmission delay is 24000 microseconds at the extreme range*
    - ***Less than 0.1 second***
    - *Can be corrected for*

UTC Time Providers

- **Time Provider**:
  - "a commercial device that is capable of directly **receiving information from a UTC server** and making appropriate **adjustments** due to **communication delays**"
- Such devices are currently installed in watches, clocks, and computers

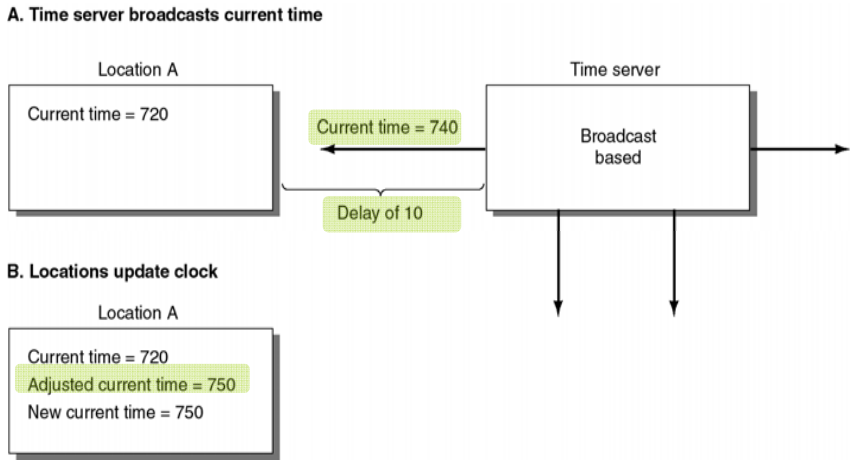
Network Delays when Communicating Time



Correcting for Transmitted Time

- A UTC signal is sent out
- Transmit time varies depending on
  - Atmospheric conditions
  - Humidity
- **Receiving clock** must make **compensation** for transmit time
- However, **once reset**, clock **will start drifting again**

Forward Adjustment of a Clock



Clock Skew

8:00:00 8:00:00

Sept 18 8:00:00

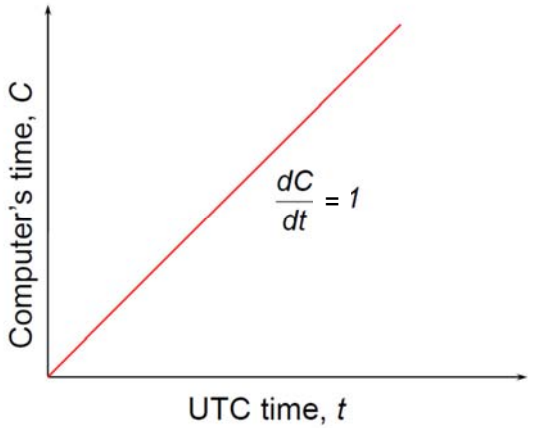
Clock Skew

8:01:24
Skew = +84 seconds
+84 seconds/35 days
Drift = +2.4 sec/day

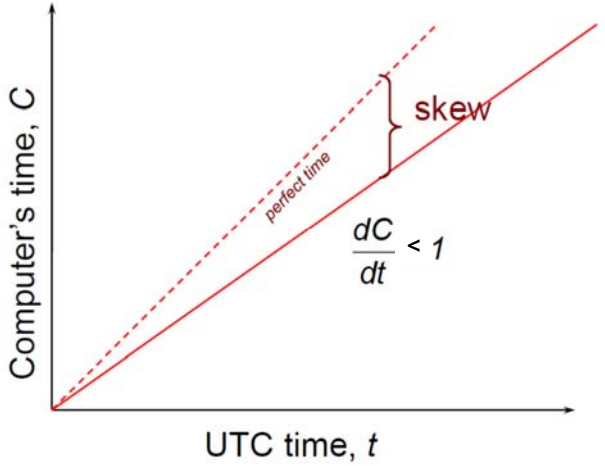
Oct 23 8:00:00

8:01:48
Skew = +108 seconds
+108 seconds/35 days
Drift = +3.1 sec/day

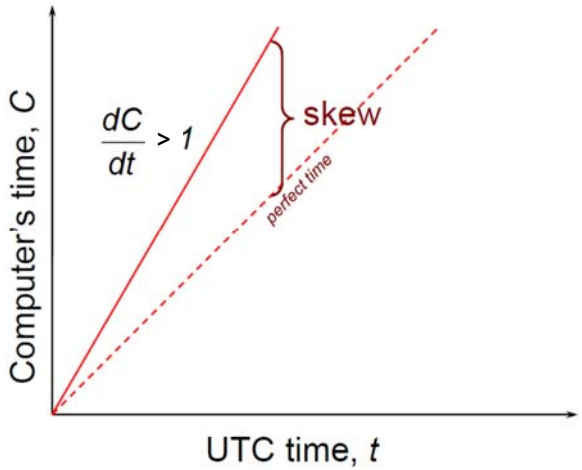
Perfect Clock



Drift with Slow Clock



Drift with Fast Clock



Problem with Clock Skew

- Problem:
  - Suppose we have a distributed system with a UTC-receiver somewhere in it => we still have to distribute its time to each machine
- Basic principle
  - Every machine has a timer that generates an interrupt H times per second.
  - There is a **clock** in machine p that **ticks** on each timer interrupt. Denote the value of that clock by $C_p(t)$, where t is UTC time.
  - **Ideally**, we have that for each machine p, $C_p(t)=t$, in other words, $dC/dt = 1$
    - UTC clock t=3.0 second, Clock started at 0 second.
    - For machine 1, $C_1(t)=C_1(3.0\ s) = 3.3$ s. $dC_1/dt = 1.1$ – **Fast** clock
    - For machine 2, $C_2(t)=C_2(3.0\ s) = 3.0$ s. $dC_2/dt = 1.0$ – **Exact** clock
    - For machine 3, $C_3(t)=C_3(3.0\ s) = 2.7$ s. $dC_3/dt = 0.9$ – **Slow** clock

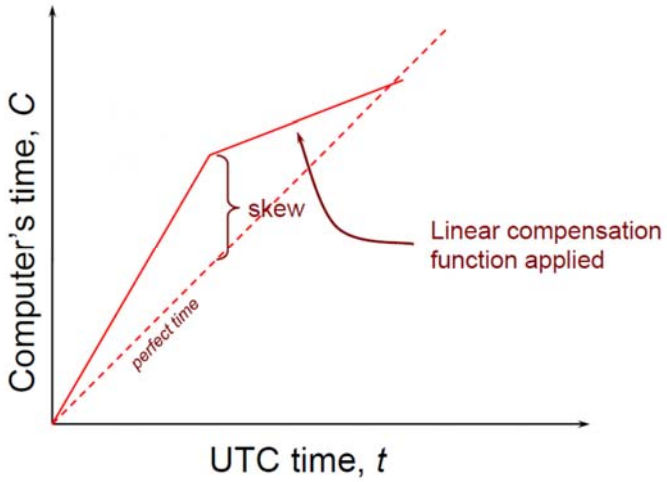
Dealing with Clock Skew

- Go for gradual clock correction
  - If fast:
    - *Make the clock run slower until it synchronizes*
  - If slow:
    - *Make the clock run faster until it synchronizes*

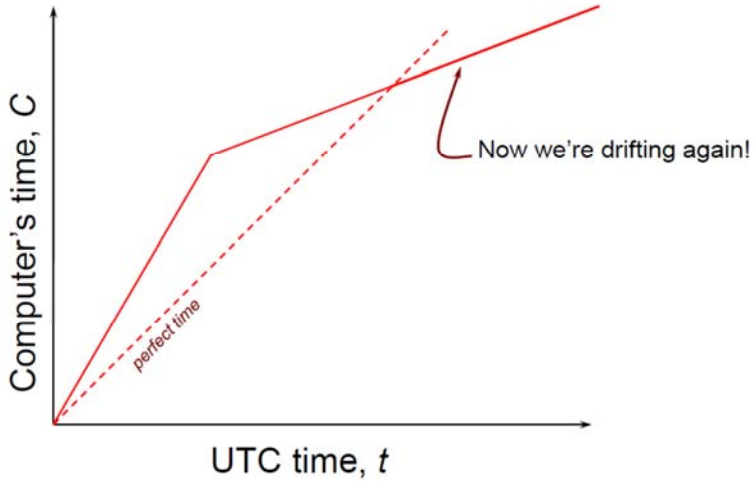
Dealing with Clock Skew

- The OS can do this:
  - **Change the rate** at which it requests interrupts:
    - **E.g. if system requests interrupts every 17 ms but clock is too slow, then request interrupts at (e.g.) 15 ms**
  - Not practical: we may not have enough precision
- Easier (software-only) solutions
  1. Redefine the rate at which system time is advanced with each interrupt
  2. Read the counter but compensate for drift
- Adjustment changes slope of system time:
  - **Linear compensation function**

Compensating for a Fast Clock



Compensating for a Fast Clock



Resynchronizing

- After synchronization period is reached
  - **Resynchronize periodically**
  - Successive application of a second **linear compensating function** can bring us closer to true slope
  - **Long-term stability is not guaranteed** – the system clock can still drift based on changes in temperature, pressure, humidity, and age of the crystal
- Keep track of adjustments and apply continuously
  - E.g. POSIX ***adjtime*** system call and ***hwclock*** command

Going to Sleep

- **RTC keeps on ticking** when the system is off (or sleeping)
- OS cannot apply correction continually
- **Estimate drift on wake-up** and apply a correct factor

Getting Accurate Time

- Attach GPS receiver to each computer
  - +/- 100 nanosecond to 1 microsecond of UTC
- Attach WWV radio receiver
  - Obtain time broadcasts from Boulder or DC
  - +/- 3 millisecond of UTC (depending on distance)
- Not practical solution for every machine
  - Cost, power, convenience, environment

Getting Accurate Time

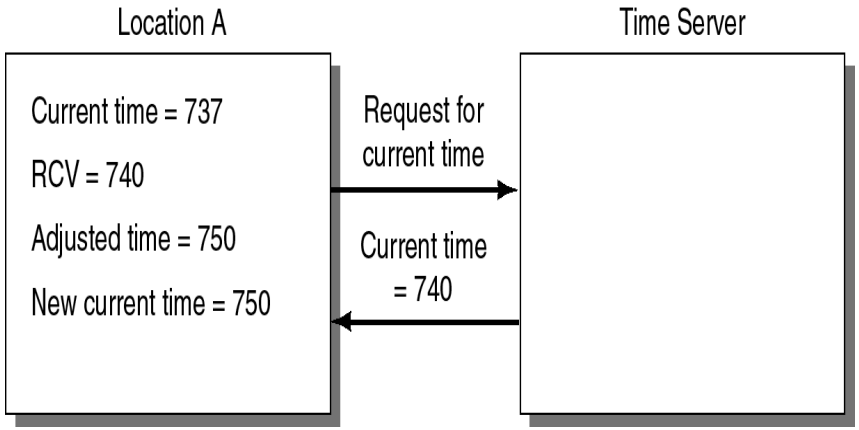
- **Synchronize from another machine**
  - One with a more accurate clock
- **Time server**:
  - Machine/service that provides time information

Physical Time Services

- Centralized
  - Broadcast-based
    - UTC (Universal Time Coordinator)
    - Berkeley Unix Algorithm by Gusella & Zatti (1989)
  - Request-driven
    - Cristian (1989)
- Distributed

- **Notice**: Clocks cannot be moved backward. Why?
  - Because illusion of time moving backwards can confuse message ordering and software development environments

Request-Driven Physical Clock Synchronization

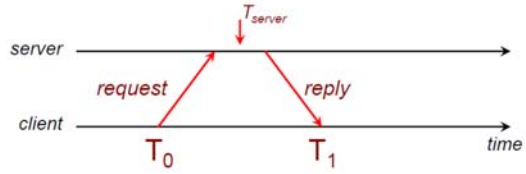


Cristian's Algorithm

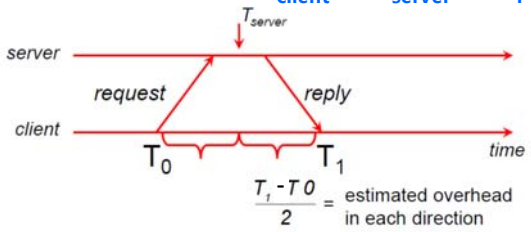
- UTC compensate for network delay, by Cristian (1989)
- Centralized time server has access to UTC
- **A process may request the current time**
- The processor receives the time T_C and sets its time to $T_C + RTT/2$ to **adjust for transmission time**
- Uses a threshold to remove bad times caused by slow/faulty message transmission
- Threshold matched against difference of times in current processor and received from server
- Considers transmit time and interrupt time

Cristian's Algorithm

- Request sent T_0 and Reply received T_1
  - Assume network delays are symmetric

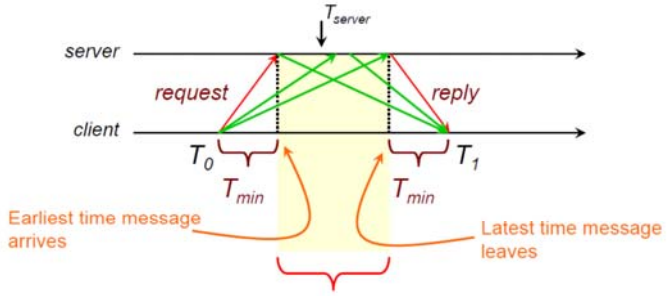


- Client sets time to: $T_{client} = T_{server} + (T_1 - T_0)/2$



Cristian's Algorithm

- If the minimum message transit time T_{min} is known:



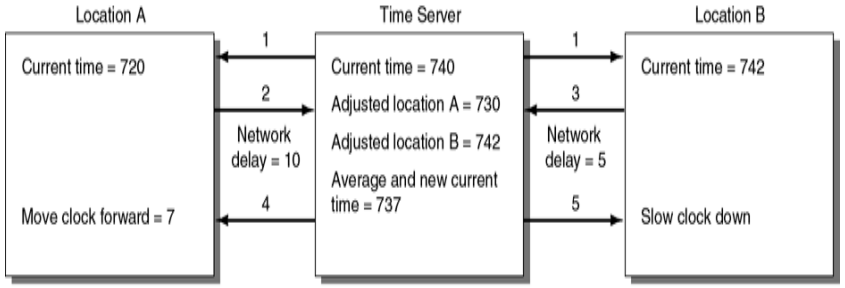
range = $T_1 - T_0 - 2T_{min}$

accuracy of result = $\pm \frac{T_1 - T_0}{2} - T_{min}$

Cristian's Algorithm

- Sent request at **5:08:15.100 (T_0)**
- Received response at **5:08:15.900 (T_1)**
  - Response contains **5:09:25.300 (T_{server})**
- Elapsed time is **$T_1 - T_0$**
  - **5:08:15.900 (T_1) - 5:08:15.100 (T_0) = 800 ms**
- Best guess:
  - Timestamp was generated 400 ms ago
- Set time to **T_{server} + elapsed time**
  - **5:09:25.300 + 400 ms = 5:09:25.700**
- If best-base message time = **200 ms (T_{min}=200)**
  - $Error = \pm \frac{900-100}{2} - 200 = \pm 200$

The Berkeley Algorithm for Physical Clock Synchronization



1. Current time = 740
2. My current time is 720
3. My current time is 742
4. Adjust forward 7
5. Adjust slowdown to accommodate 5

Berkeley Algorithm

- Gusella & Zatti (1989)
- **Synchronizes clocks for processors running Berkeley Unix 4.3**
- **Does not require UTC**
- Centralized server broadcasts time periodically

Berkeley Algorithm

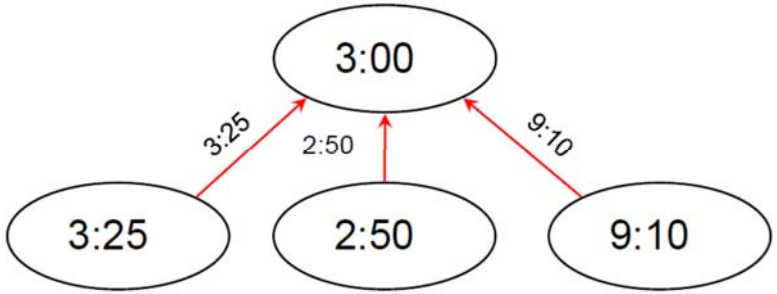
- Machines run time daemon
  - Process that implements protocol
- One machine is designated as the server (**master**)
  - Others are **slaves**
- **Master** polls each machine **periodically**
  - Ask each machine for **time** - Can use Cristian's algorithm to compensate for network latency
- When results are in, compute average
  - Including master's time
- **We hope: an average cancels out individual clock's tendencies to run fast or slow**
- Send offset by which each clock needs adjustment to each slave
  - Avoid problems with network delays if we send a timestamp

Berkeley Algorithm

- Algorithm has provisions for ignoring readings from clocks whose skew is too great
  - Compute a **fault-tolerant average**
- If master fails
  - Any slave can take over via an election algorithm

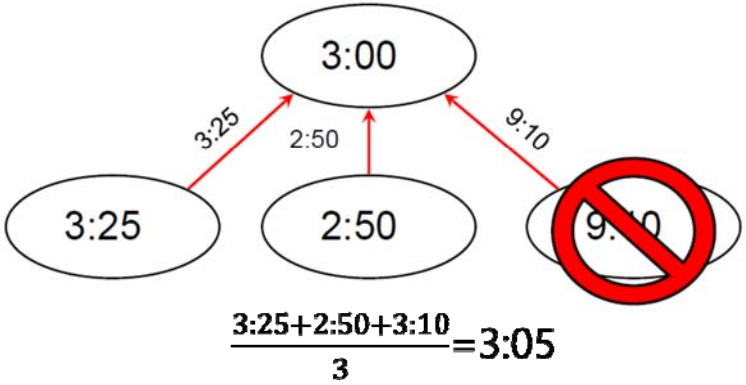
Berkeley Algorithm

- Request timestamps from all slaves



Berkeley Algorithm

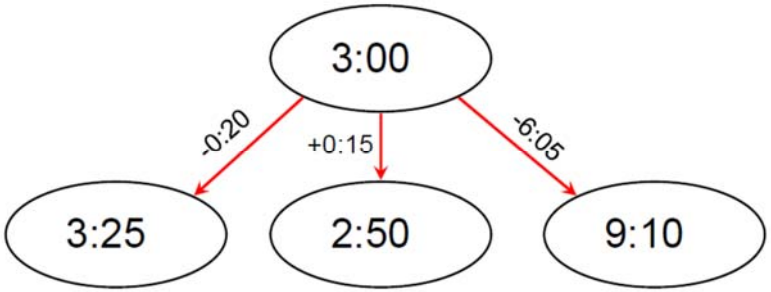
- Compute fault-tolerant average:
  - Suppose max δ = 0:45



$$\frac{3:25+2:50+3:10}{3}=3:05$$

Berkeley Algorithm

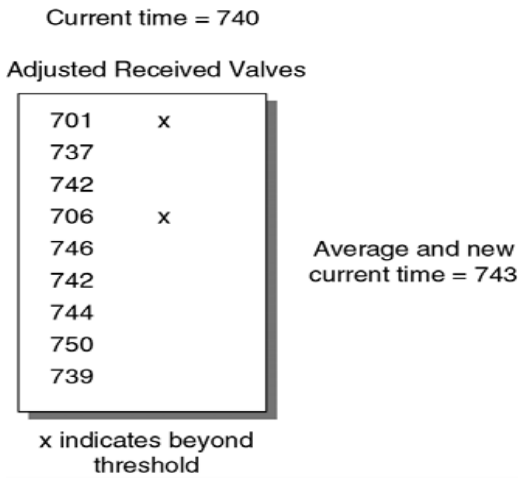
- Send offset to each client



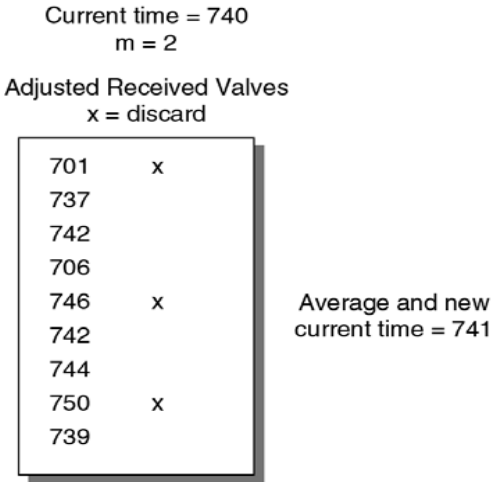
Distributed Physical Time Services

- Each processor **broadcasts** its current time at regular intervals
- Then starts a timer
- **Timestamps** each response
- Does so until timer runs out
- Then adjusts its own time accordingly

Fault-Tolerant Threshold Method



Discard *m* Highest and Lowest Values



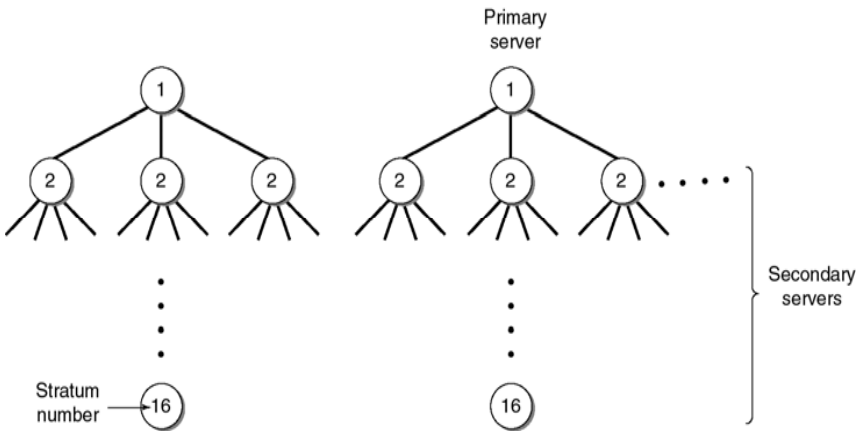
Network Time Protocol (NTP)

- 1991, 1992
  - Internet Standard, version 3, RFC 1305
- June 2010
  - Internet Standard, version 4, RFC 5905-5908
  - IPv6 support
  - Improve accuracy to tens of microseconds
  - Dynamic server discovery

NTP Goals

- Enable clients across Internet to be **accurately synchronized to UTC** despite message delays
  - Use statistical techniques to filter data and gauge quality of results
- Provide **reliable service**
  - Survive lengthy losses of connectivity
  - Redundant paths
  - Redundant servers
- Provide **scalable service**
  - Enable clients to **synchronize frequently**
  - Offset effects of clock drift
- Provide **protection against interference**
  - Authenticate source of data

Strata in the NTP Architecture



NTP Servers

- Arranged in strata
  - Stratum 0: machines connected directly to accurate time source
  - Stratum 1: machines synchronized from stratum-0 machines
  - Stratum 2: machines synchronized from stratum-1 machines
  - …

Synchronization Subnet

NTP Synchronization Modes

- Multicast mode
  - For high speed LANS
  - Lower accuracy but efficient
- Procedure call mode
  - Similar to Cristian's algorithm
- Symmetric mode
  - Intended for master servers
    - A probes B; B probes A -> A adjusts its clock only if A's stratum > B's
  - Peer servers can synchronize with each other to provide mutual backup
    - Pair of servers retain data to improve synchronization over time

All message are delivered unreliably with UDP

NTP Clock Quality

- Precision
  - Smallest increase of time that can be read from the clock
- Jitter
  - Difference in successive measurements
  - Due to network delays, OS delays, and wander – clock oscillator instability
- Accuracy
  - How close it the clock to UTC?

NTP Messages

- Procedure call and symmetric mode
  - Messages exchanged in pairs: request and response
- Time encoded as a 64 bit value
  - Divide by 2^{32} to get the number of seconds since Jan 1 1900 UTC
- NTP calculates
  - **Offset** for each pair of messages (θ)
    - Estimate of time offset between two clocks
  - **Delay** (δ)
    - Travel time : ½ of total delay minus remote processing time
  - **Jitter/Dispersion**
    - Maximum offset error
- Use this data to find preferred server
  - Probe multiple servers – each several times
  - Pick lowest total dispersion & lowest stratum

NTP Message Structure

- Leap second indicator
  - Last minute has 59, 60, 61 seconds
- Version number
- Mode (symmetric, unicast, broadcast)
- Stratum (1=primary reference, 2-15)
- Poll interval
  - Maximum interval between 2 successive messages, nearest power of 2
- Precision of local clock
  - Nearest power of 2

NTP Message Structure

- Root delay
  - Total roundtrip delay to primary source
  - 16 bits seconds, 16 bits decimal
- Root dispersion
  - Nominal error relative to primary source
- Reference clock ID
  - Atomic, NIST dial-up, radio, LORAN-C navigation system, GPS, ..
- Reference timestamp
  - Time at which clock was last set (64 bit)
- Authenticator (key ID, digest)
  - Signature (ignored in SNTP)

NTP Message Structure

- T_1: originate timestamp
  - Time request departed client (client's time)
- T_2: receive timestamp
  - Time request arrived at server (server's time)
- T_3: transmit timestamp
  - Time request left server (server's time)

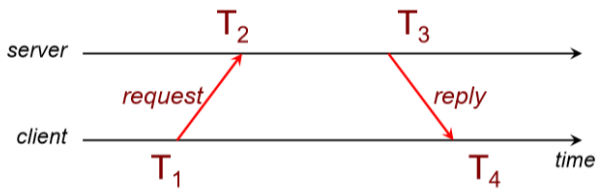
NTP Validation Tests

- Timestamp provided ≠ last timestamp received
  - Duplicate message?
- Originating timestamp in message consistent with sent data
  - Messages arriving in order?
- Timestamp within range?
- Originating and received timestamps ≠ 0?
- Authentication disabled? Else authenticate
- Peer clock is synchronized?
- Don't sync with clock of higher stratum #
- Reasonable data for delay & dispersion

Simple Network Time Protocol (SNTP)

- Ver3 RFC 2030, Oct 1996
- Ver4 RFC 5905, June 2010
- An adaptation of NTP
  - Subset of NTP, not new protocol
- Simplifies access to an NTP server
- Involves stateless remote computer calls
  - Operates in multicast or procedure call mode
- Clients located only at the highest strata
  - Recommended for environments where server is root node and client is leaf of synchronization subnet
- SNTP servers do not implement fault tolerance
  - Root delay, root dispersion, reference timestamp ignored

Simple Network Time Protocol (SNTP)



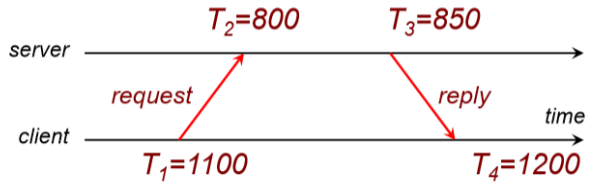
Round-trip delay:

$$d = (T_4 - T_1) - (T_2 - T_3)$$

Time offset:

$$t = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

Simple Network Time Protocol (SNTP)



Offset =

((800 - 1100) + (850 - 1200)) / 2

= ((-300) + (-350)) / 2

= -650 / 2 = -325

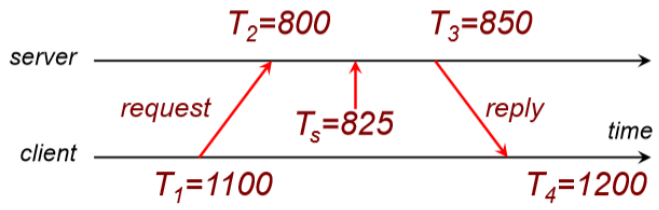
Time offset:

$$t = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

Set time to $T_4 + t$

= 1200 - 325 = 875

Cristian's Algorithm



Offset = (1200 - 1100) / 2 = 50

Set time to T_s + *offset* = 825 + 50 = 875

Key Points: Physical Clocks

- Cristian's algorithm & SNTP
  - Set clock from server
  - But account for network delays
  - Error: uncertainty due to network/processor latency
    - Errors are additive
    - Example: ±10 ms and ±20 ms = ±30 ms
- Adjust for local clock skew
  - Linear compensating function

Logical Time

- **Because of clock skew, physical clocks do not provide absolute time ordering of events**
- Instead we use the concept of virtual time to order certain events
- There are a great number of algorithms that attempt to provide logical time and some event ordering
  - E.g. **Lamport's logical clock**

Ordering Events

- What is an **event**?
  - Sending a message
  - Receiving a message
  - Execution within a process
- Most events happen asynchronously
  - Non-instantaneous communication
  - Interruptions
- **There is no global state**

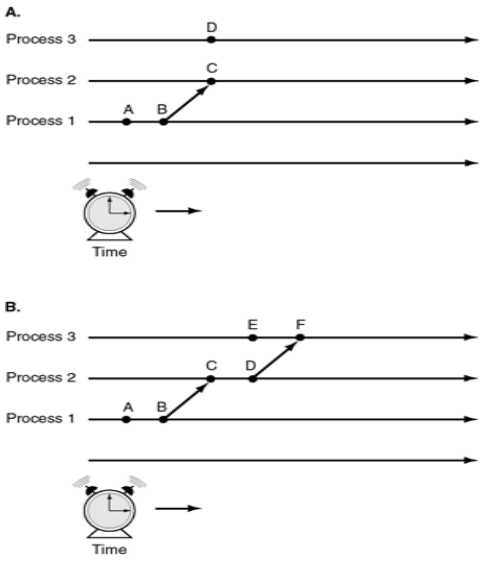
Assumptions

- Assume all processes are sequential
- Assume that the sending of a message always precedes the receiving of said message
- Need to define a relationship that combine this information
  - **Lamport's → *"happens before"* relation**

Properties of →

1. **a → b** is defined as
   i. If a and b are in the same process, then **a happens before b** happens
   ii. If **a is sending** a message, then **b is receiving** the same message
2. Transitive: If **a →b and b → c**, then **a → c**
3. If there is no ordering between a and b, **!(a → b) and !(b → a)**, then a and b are **concurrent** (disjoint)

Happen-Before(HR) Relationship Examples



Properties of a Logical Clock

- Let Ci be associated with the process Pi, for all processes Pi
- Clock condition:
  - if **a → b**, then **C(a) → C(b)**
- Subconditions:
  - If **a → b** in process Pi, then **Ci(a) < Ci(b)**
  - If **a sends** message m and **b receives** m, then **Ci(a) < Cj(b)**

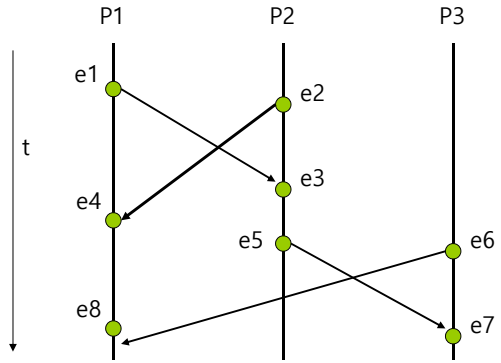
Logical Clock Conditions

- In order to achieve these conditions,
  - Pi increments Ci between any two events related to Pi
  - If a is sending message m from Pi, put a timestamp, Tm = Ci(a), on the message m
  - When m is received by b in Pj, Pj sets Cj to be the maximum value of Cj + d or Tm + d for some increment d

Definition of *precede*

- Definition: Event a in Pi ***precedes*** event b in Pj if and only if (system-wide)
  - 1. Ci(a) < Cj(b) OR
  - 2. Ci(a) = Cj(b) and Pi < Pj
- Assume that each process Pi is ordered by a unique value of i
- This relation is written as **a ➔ b**

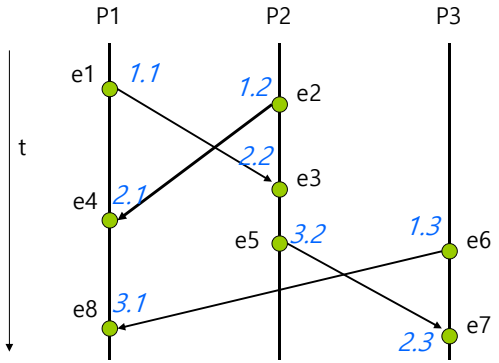
Example



e1 → e4 → e8 (P1)
e2 → e3 → e5 (P2)
e6 → e7 (P3)
e1 → e3 (m)
e2 → e4 (m)
e5 → e7 (m)
e6 → e8 (m)

A partial ordering

Example



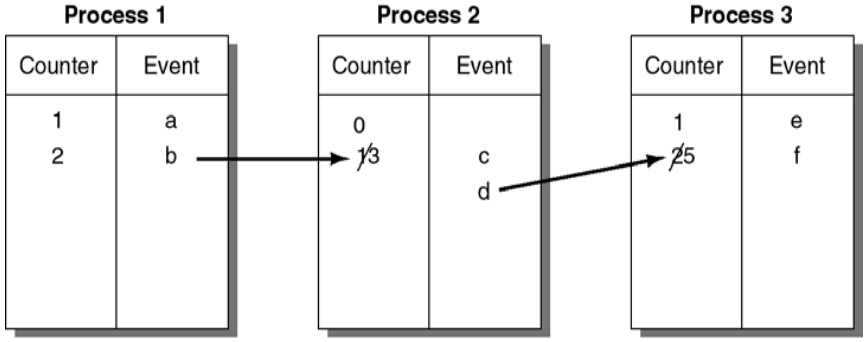
A DAG

Becomes a total ordering on events

Total Ordering of Events

- Any total ordering on events must be consistent with the existent partial order
- One solution: a topological sort on the partial order – after the fact
- **Lamport: Uses an event number and a timestamp on all events**
- Further, a timestamp is attached to all messages

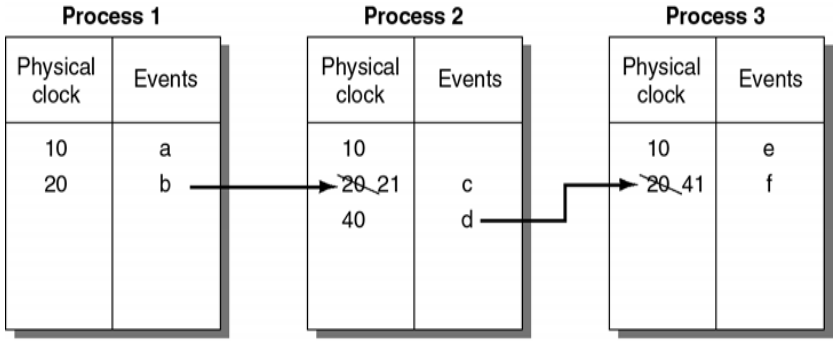
Logical Ordering of Events Using Counters



c is the event of receiving b
f is the event of receiving d

Each requires a counter adjustment to preserve the happens-before relationship.

Logical Ordering of Events Using Physical Clocks



c is the event of receiving b
f is the event of receiving d

Each required a clock adjustment to preserve the happens-before relationship.

Causal Events

- Causal:
  - "1. Expressing or indicting cause;
  - 2. Relating to or acting as cause"
    - *(Merriam-Webster)*
- Causal events:
  - If e1 → e2, then C(e1) → C(e2)
  - Two events may have the same timestamp
    - Just include the i of Pi as part of the timestamp

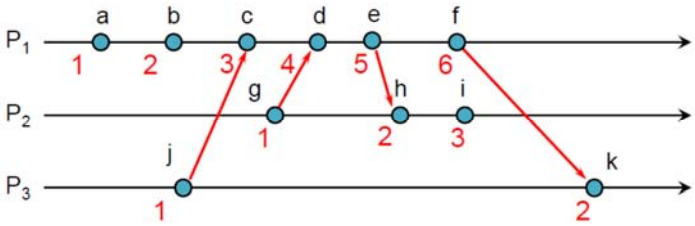
Lamport's Algorithm

- Each message carries a timestamp of the sender's clock
- When a message arrives:
  - If receiver's clock < message_timestamp,
  - Then set system clock to message_timestamp + 1
  - Else do nothing
- Clock must be advanced between any two events in the same process
- Lamport's algorithm allows us to maintain time ordering among related events – **Partial ordering**

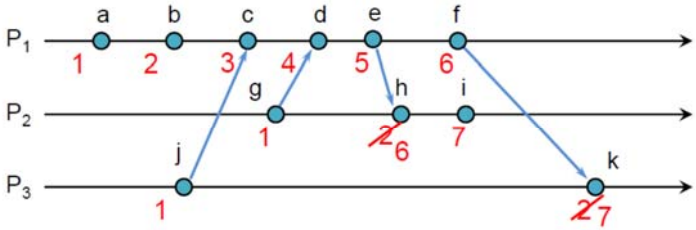
Lamport's Algorithm

- For each process p,
  - Initialize the timestamp, p.TS, to zero
  - On each event,
    - If e is receipt of message m
      - p.TS = max (m.TS, p.TS);
    - p.TS ++;
    - e.TS = p.TS;
    - If e is sending message m
      - m.TS = p.TS;

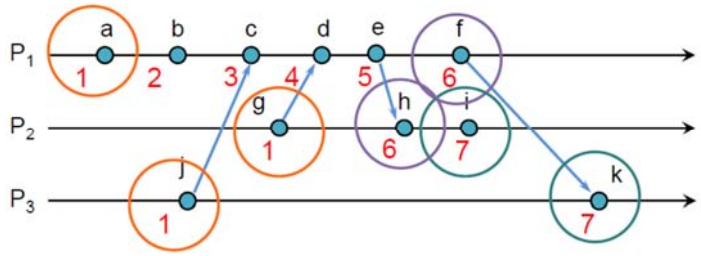
Lamport's Algorithm



- Applying Lamport's algorithm



Problem: Identical Timestamps

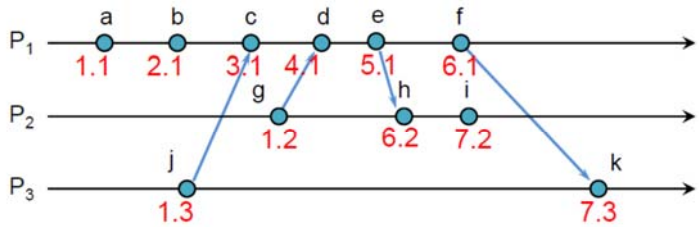


- a → b, b → c, c → d, … : local events sequenced
- j → c, g → d, e → h … : Lamport imposes a send->receive relationship
- Concurrent events (e.g., b & g; i& k) may have the same timestamp or not

Unique Timestamps (Total Ordering)

- We can force each timestamp to be unique
  - Define **global logical timestamp** ***(Ti, i)***
    - ***Ti*** represents **local Lamport timestamp**
    - ***i*** represents **process number (globally unique)**
      - e.g., (host address, process ID)
  - Compare timestamps:
    - ***(Ti, i) < (Tj, j)***
      if and only if
      ***Ti < Tj* or**
      Ti = Tj* and *i < j
- Does not necessarily relate to actual event ordering

Unique Timestamps (Total Ordering)



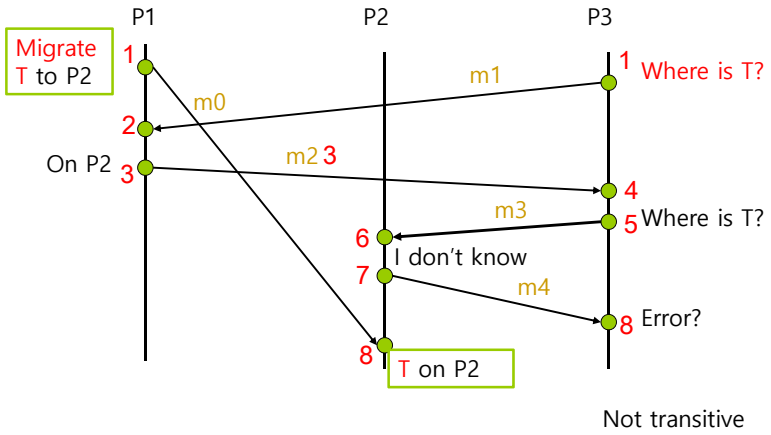
Limitation of Lamport's Clocks

- If a → b , then C(a) < C(b)
- But the reverse is not necessarily true – if the events occur in different processes
  - I.e., if C(a) < C(b), we cannot conclude that b → a
  - **We can't tell how a and b are related**
  - Each clock can independently advance based on its local events
  - We need messages exchanges to synchronize between a pair of processes

Concurrent Events

- There is an arbitrary ordering of concurrent events
- This can lead to a ***causality violation***:
  - When distributed objects are mobile, i.e. they can move freely among processes
  - This may happen when load balancing occurs

Casuality Violation



Not transitive

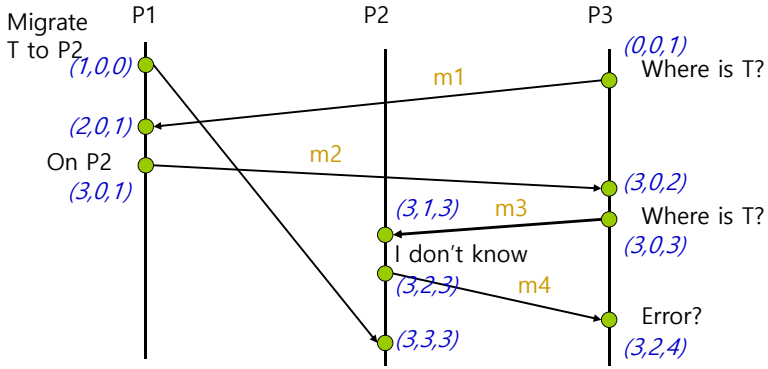
Causality Violation

- Message m0 arrives late to P2
- Message m3 arrives at P2 before P2 knows that T is migrating there
- To fulfill the transitivity condition, m3 should have arrived at P2 after m0 arrived at P2

Causality Violation

- s(m) = the event of sending message m
- r(m) = the event of receiving message m
- m1 <c m2 if s(m1) → s(m2)
- A causality violation happens if m1 <c m2, but r(m2) <p r(m1)
- Need a comparison function f such that
  - e → e' iff f(e) < f(e')
  - Idea – **vector timestamps**

Causality Violation, relabelled



Not transitive

Vector Clocks

- Each Pi keeps a clock vector Ci[k], k=1,..,n
- The kth entry is Pi's best guess of what process Pk has for its clock values
- A message carries a timestamp vector of the clock vector of the sender
- A receiver updates its clock vector using the timestamp vector from the message

Vector Clocks

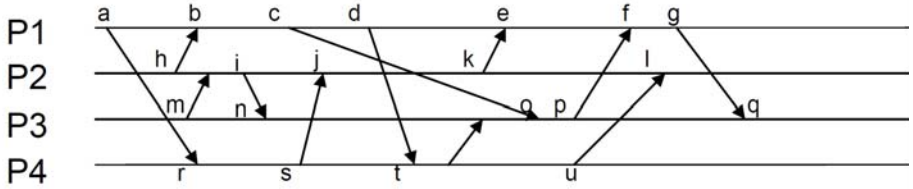
- The vector clocks provide a partial ordering of the timestamps
  - Using a vector comparison (all elements must be =, <, or > pairwise
  - If ta < tb or ta > tb , then a and b are **causally related**
    - **If a->b then C(a) < C(b)**
    - **If C(a) < C(b) then a->b**
  - Otherwise a and b must be **concurrent**
    - **C(a) < C(b) nor C(b) < C(a)**

Vector Clocks

- For M processes,
  - Initialize p.VT = [0, 0, … 0]
  - On event e,
    - If e is receipt of message m
      - For i=1 to M
        - P.VT[i] = max (p.VT[i], m.VT[i])
    - p.VT(self) ++ ;
    - e.VT = p.VT ;
    - If e is sending message m
      - m.VT = p.VT;

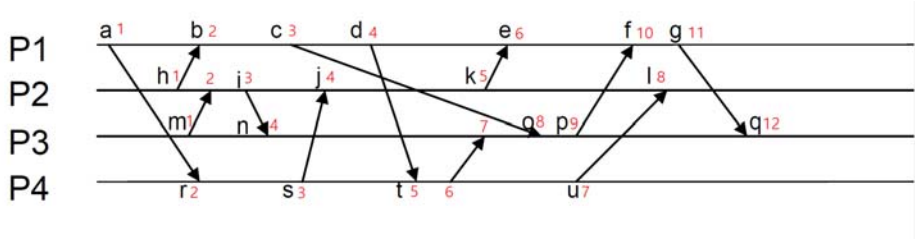
Lamport vs Vector Clock Timestamps

- 4 processes (p1, P2, P3, P4) with events a,b,c,d,e,f,g,…



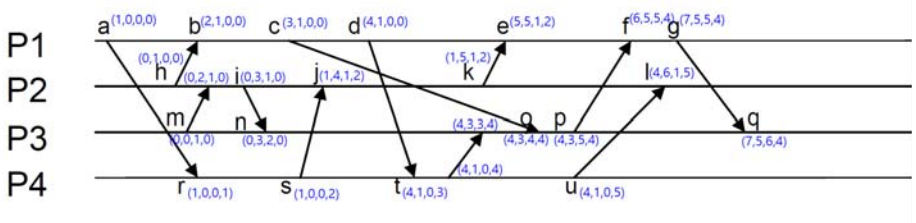
Lamport vs Vector Clock Timestamps

- Lamport Timestamps



Lamport vs Vector Clock Timestamps

- Vector Clock Timestamps



References

- http://www.cs.colostate.edu/~cs551/CourseNotes/Synchronization/SynchTOC.html
- https://www.cs.rutgers.edu/~pxk/417/notes/content/05-clock-synchronization-slides.pdf
- https://www.cs.rutgers.edu/~pxk/417/notes/content/06-logical-clocks-slides.pdf