Lecture 08
Synchronization Issues in Distributed Systems

Dr. Ehtesham Zahoor

- These slides are not ***always*** prepared by me and they do use and present content from different sources
  - *CS 417 – Distributed Systems, Paul Krzyzanowski at Rutgers University*
  - *Distributed systems: Principles and Paradigms, Andrew S. Tanenbaum*
  - *NIST A Walk through Time - The Evolution of Time Measurement through the Ages*

- *Please do not share them ☺*

Synchronization - Centralized Systems

- Which synchronization primitives we have learned so far?
- Our solutions mainly relied on hardware support.

- What are the limitations?

Some synchronization aspects

- Mutual exclusion and scheduling constraints
  - Locks and semaphores can help
- Ordering constraints
  - Event A happens before Event B, the order should be preserved.
  - FileA is modified after FileB, some transaction in DB is after some other transaction and so on.
  - How this can be done?

Example

- A bank has a database.
  - **Update 1:** Assume a customer wants to add $100 to his account, which currently contains $1,000.
  - **Update 2:** The customer's account is then to be increased with 10 percent interest.
- How difficult this can be?

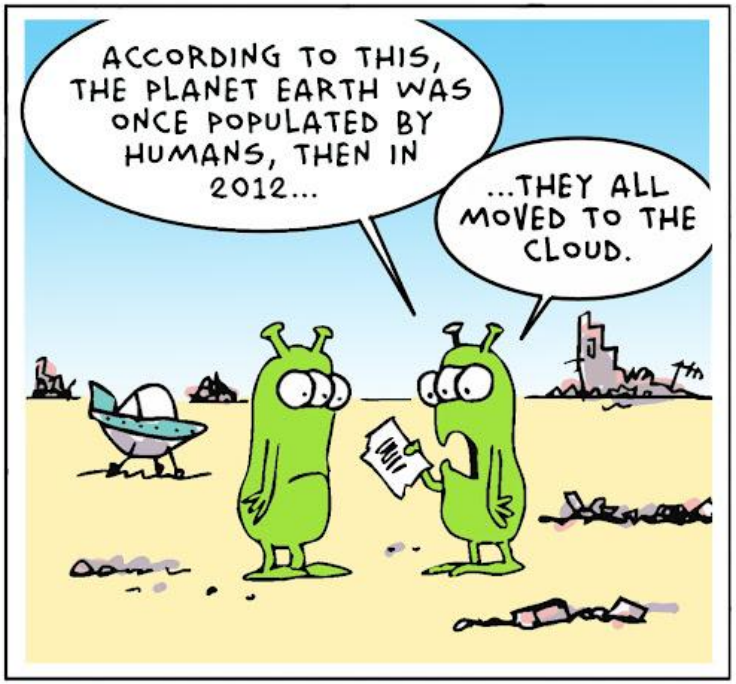
Introduction

- In a centralized system, time is unambiguous.
  - When a process wants to know the time, it makes a system call and the kernel tells it.
  - If process *A* asks for the time. and then a little later process *B* asks for the time, the value that *B* gets will be higher than the value *A* got
  - It will certainly not be lower.

Distributed Systems

- *"A distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility. "*

Distributed systems – any example?



Cloud Computing

- Definition from ***NIST*** ***(National Institute of Standards and Technology)***
  - Cloud computing is a model for enabling convenient, **on-demand network access** to a **shared pool** of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be **rapidly provisioned and released** with minimal management effort or service provider interaction.

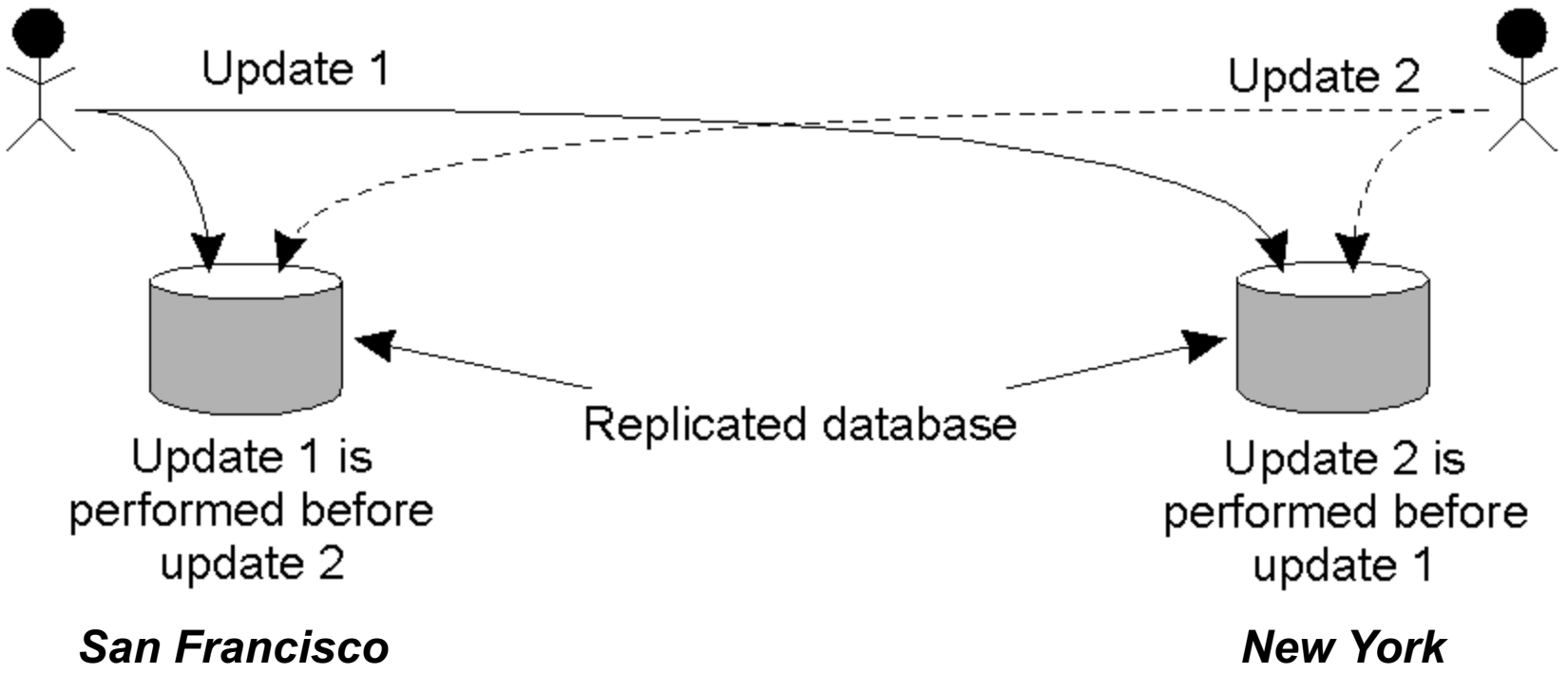
What can go wrong?

- How to acquire locks, how to reach consensus, how to preserve ordering, how to manage clock synchronization, given that …
  - Nodes may crash
  - Nodes may be malicious
  - Network is imperfect
  - Not all pairs of nodes connected
  - Latency

But why it is so important?

- A bank has distributed replicated copies of a database.
  - **Update 1:** Assume a customer in San Francisco wants to add $100 to his account, which currently contains $1,000.
  - **Update 2:** At the same time, a bank employee in New York initiates an update by which the customer's account is to be increased with 10 percent interest.
- Both updates should be carried out at both copies of the database. What can go wrong?

Time in Distributed Systems



Another example

- Small C/C++ applications with a couple of modules are easy to manage.
  - Developers can recompile them easily by calling the compiler directly, passing source files as arguments.

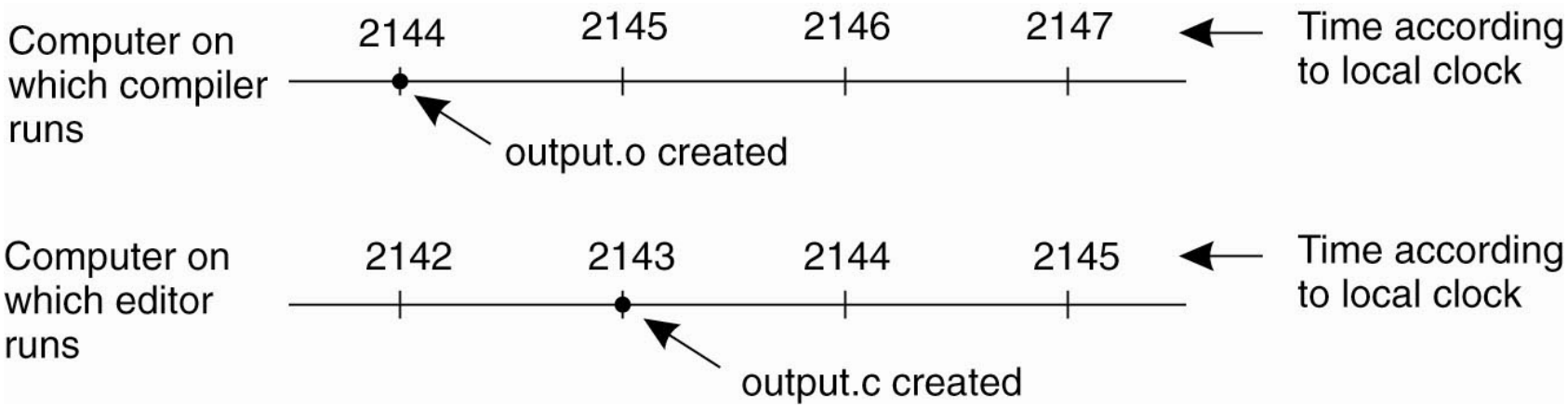
- If a program consists of 100 files; ideadlly, there should be no need to recompile everything because one file has been changed

The UNIX *make* tool

- The way *make* normally works is simple.

- After some modifications to the source files, programmer runs *make command*:
  - It examines the timestamps
  - If the source file *input.* c has time 2151 and the corresponding object file *input.o* has time 2150, *make* knows that *input.c* has been changed since *input.o* was created, and thus *input.c* must be re-compiled.
  - On the other hand, if *output.c* has time 2144 and *output.o* has time 2145, no compilation is needed.

What can go wrong in a DS?



Suppose that *output.o* has time 2144 as above, and shortly thereafter *output.c* is modified but is assigned time 2143 because the clock on its machine is slightly behind, *Make* will not call the compiler.

Convinced about the need?

- Yes, but why cant we just synchronize all the clocks in a distributed system? how hard can it be?

- The answer is surprisingly complicated !!

Clock synchronization

- Physical clocks keep time of day – Ideally should be consistent across systems

- Logical clock keeps track of event ordering among related (causal) events

Physical clocks

- We need to consider two aspects
  - How and why we cant keep the clocks accurate and consistent?
  - What can be done to handle the clock drift?
  - First a brief walk through time!

A walk through time …

- Ancient civilizations relied upon the apparent motion of Celestial bodies, the Sun, Moon, planets, and stars, through the sky to determine seasons, months, and years.

Ancient Calendars

The Calendar of Eclipses

Eclipses of the Sun

Solar Eclipse - main directory for NASA's Solar Eclipse Page (some popular links below)

Eclipses During:
- | 2001 | 2002 | 2003 | 2004 | 2005 | 2006 | 2007 | 2008 | 2009 | 2010 |
- | 2011 | 2012 | 2013 | 2014 | 2015 |

Decade Solar Eclipse Tables:
- | 1951 - 1960 | 1961 - 1970 | 1971 - 1980 | 1981 - 1990 | 1991 - 2000 |
- | 2001 - 2010 | 2011 - 2020 | 2021 - 2030 | 2031 - 2040 | 2041 - 2050 |

Solar Eclipses on Google Maps:
- | 1901 - 1920 | 1921 - 1940 | 1941 - 1960 | 1961 - 1980 | 1981 - 2000 |
- | 2001 - 2020 | 2021 - 2040 | 2041 - 2060 | 2061 - 2080 | 2081 - 2100 |

World Atlas of Solar Eclipse Maps: **Index Page**
- | 1901 - 1920 | 1921 - 1940 | 1941 - 1960 | 1961 - 1980 | 1981 - 2000 |
- | 2001 - 2020 | 2021 - 2040 | 2041 - 2060 | 2061 - 2080 | 2081 - 2100 |

North America Solar Eclipse Maps: 1851-2100 - **Index Page**

Five Millennium Catalog of Solar Eclipses: -1999 to +3000
Five Millennium Solar Eclipse Search Engine - search for solar eclipses and plot on Google maps
Javascript Solar Eclipse Explorer - calculate all solar eclipses visible from a city

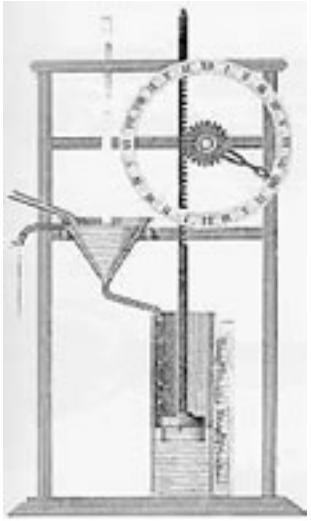
Eclipses of the Moon

Lunar Eclipse - main directory for NASA's Lunar Eclipse Page (some popular links below)

Eclipses During:
- | 2001 | 2002 | 2003 | 2004 | 2005 | 2006 | 2007 | 2008 | 2009 | 2010 |
- | 2011 | 2012 | 2013 | 2014 | 2015 |

Decade Lunar Eclipse Tables:

Early Clocks



A candle marked
for use as a timer

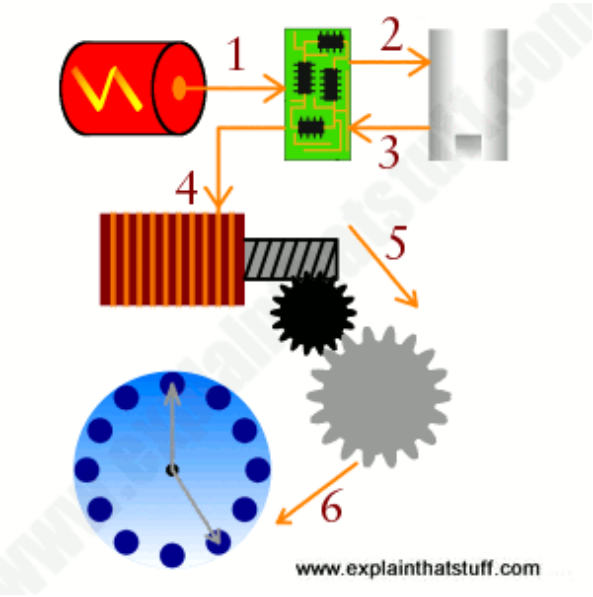
Quartz Clocks

- Quartz sounds exotic but it's actually one of the most common minerals on Earth.

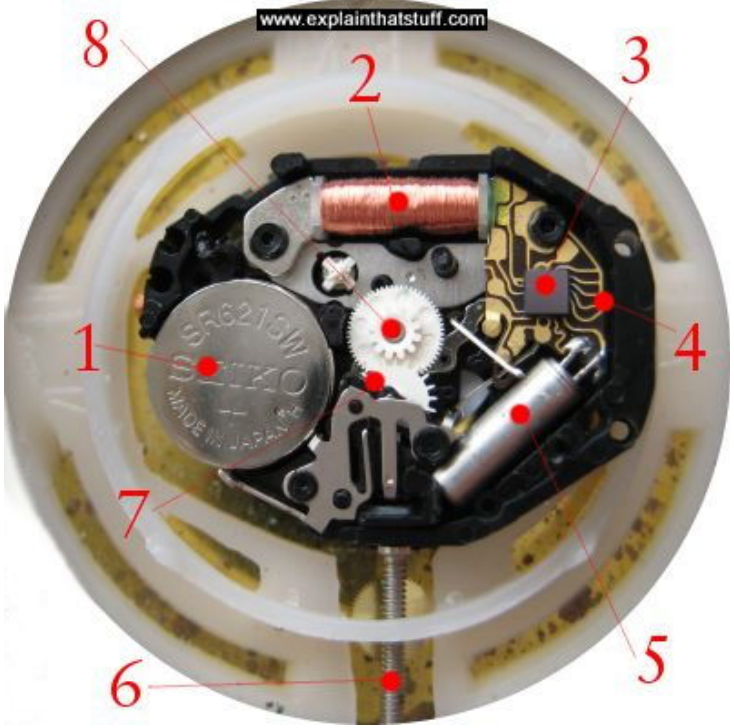
- When put in a suitable electronic circuit, the Quartz crystal vibrates and generate an electric signal of relatively constant frequency that can be used to operate an electronic clock display.

How Quartz Clocks Work?

- In theory, it works like this:
  1. Battery provides current to microchip circuit
  2. Microchip circuit makes quartz crystal (precisely cut and shaped like a tuning fork) oscillate (vibrate) 32768 times per second.
  3. Microchip circuit detects the crystal's oscillations and turns them into regular electric pulses, one per second.
  4. Electric pulses drive miniature electric stepping motor. This converts electrical energy into mechanical power.
  5. Electric stepping motor turns gears.
  6. Gears sweep hands around the clockface to keep time.



How Quartz Clocks Work?



1. *Battery.*
2. *Electric stepping motor.*
3. *Microchip.*
4. *Circuit connects microchip to other components.*
5. *Quartz crystal oscillator.*
6. *Crown screw for setting time.*
7. *Gears turn hour, minute, and second hands at different speeds.*
8. *Tiny central shaft holds hands in place.*

Quartz Clocks

- Quartz vibration frequency depends critically on the crystal's size, shape and temperature.

- Thus, no two crystals can be exactly alike, with just the same frequency !!

Atomic Clocks

- Each chemical element and compound absorbs and emits electromagnetic radiation at its own frequencies.
- The frequencies are highly stable over time and space.

- Thus atoms constitute a potential "pendulum" with a reproducible rate that can form the basis for more accurate clocks.

Atomic Clocks

- The cesium atom's natural frequency was formally recognized as the new international unit of time in 1967: the second was defined as exactly 9,192,631,770 oscillations or cycles of the cesium atom's resonant frequency

- NIST's cesium standard is capable of keeping time to about 30 billionths of a second per year.

Physical clocks in computers

- Real-time Clock: CMOS clock (counter) circuit driven by a quartz oscillator
  - Battery backup to continue measuring time when power is off

Problem

- Getting two systems to agree on time
  – Two clocks hardly ever agree
  – Quartz oscillators oscillate at slightly different frequencies

- Clocks tick at different rates
  – Create ever-widening gap in perceived time – Clock Drift

- Difference between two clocks at one point in time
  – Clock Skew

How it can be fixed?

- One easy? solution is to simply update the system time to the *true time*.
- On servers with actively running processes, a spontaneous change in time, possibly a jump back in time, can be disasterous.

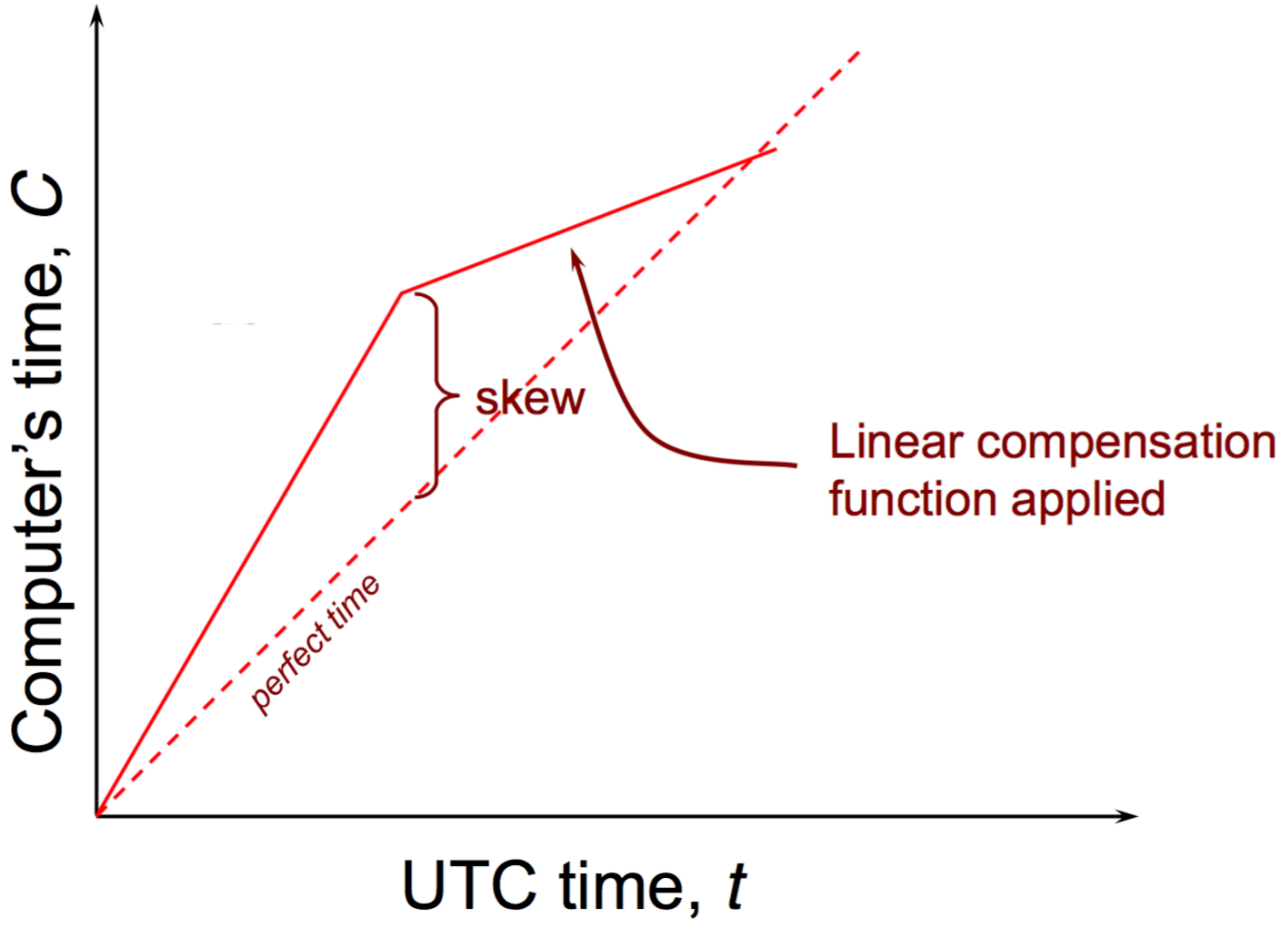
- Constraint: it is not a good idea to set the clock back!!

How it can be fixed?

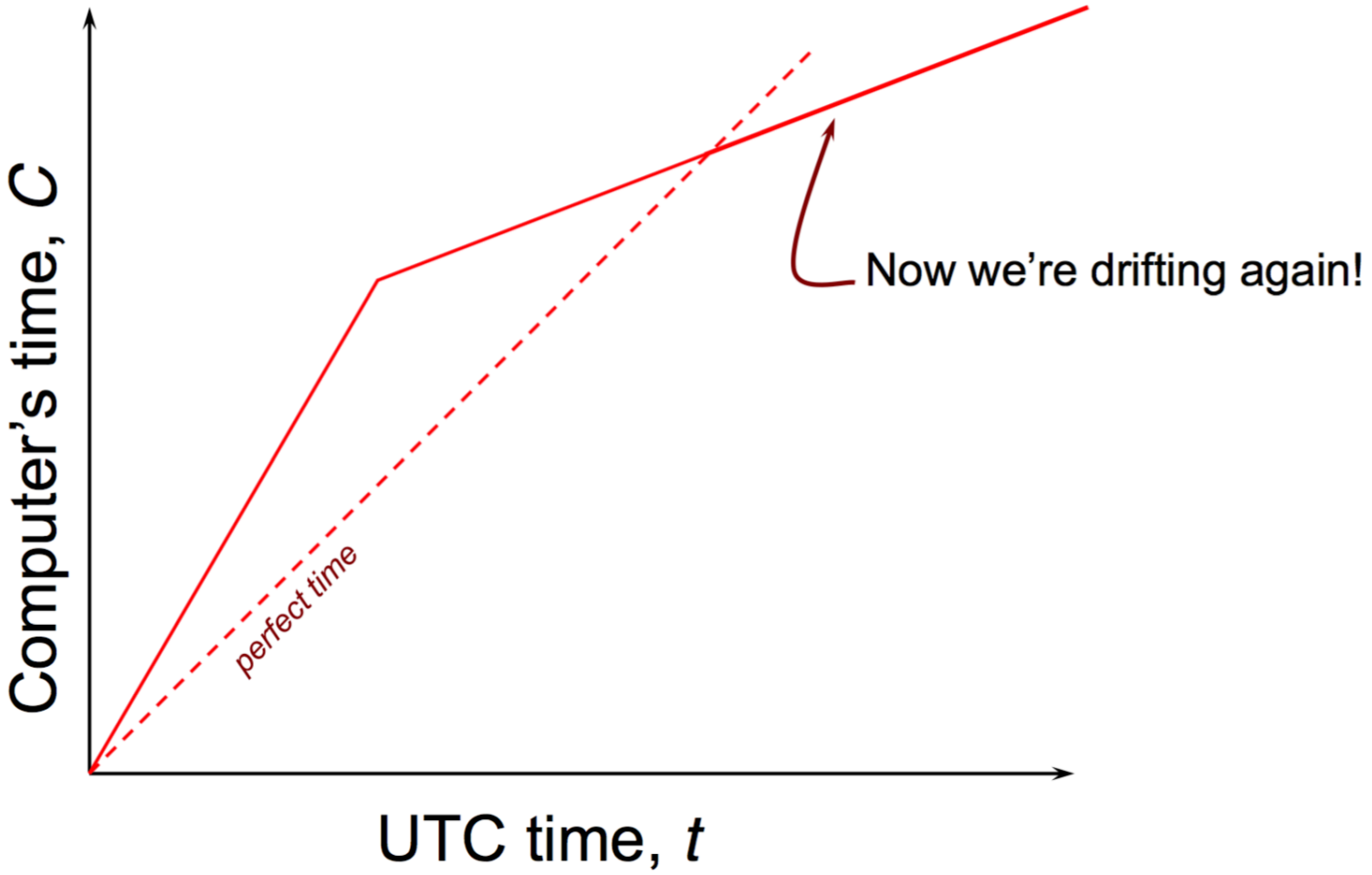
- Go for gradual clock correction
  - If fast, make the clock run slower until it synchronizes
  - If slow, make the clock run faster until it synchronizes

- The OS can do this but not with enough precision

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

Setting the time on physical clocks

- Our interest is not in advancing them just to ensure proper message ordering, but to have the system clock keep good time.

- We looked at methods for adjusting the clock to compensate for skew and drift, but it is essential that we get the time first so that we would know what to adjust.

Getting accurate time

- Attach GPS receiver to each computer, ±100ns to 1μs of UTC
- Attach WWV radio receiver
  – Obtain time broadcasts from Boulder or DC, ± 3 ms of UTC (depending on distance)

- Not practical solution for EVERY machine – Cost, power, convenience, environment

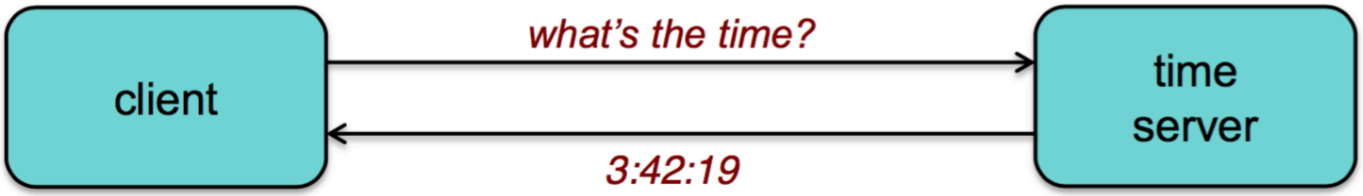
Getting accurate time

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

Time server

Remote Request/Response

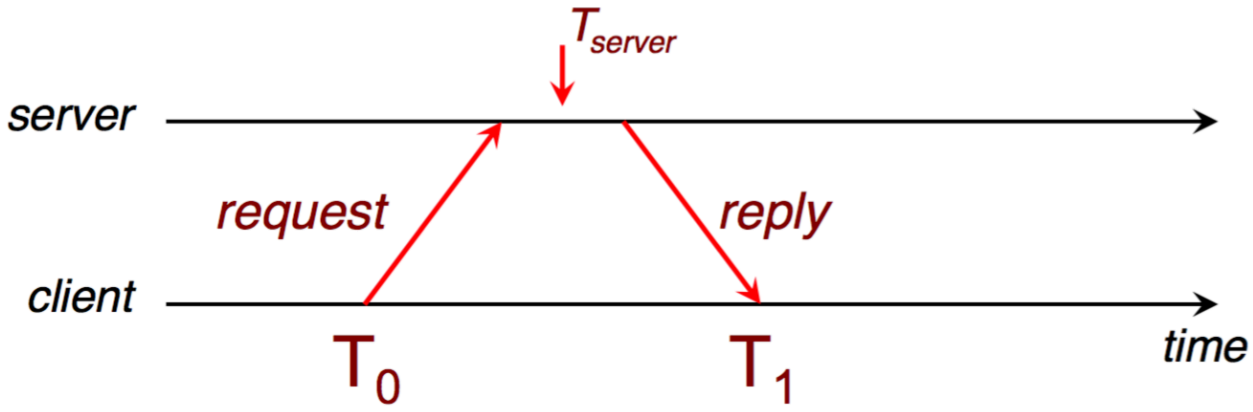
- Simplest synchronization technique
  - Send a network request to obtain the time
  - Set the time to the returned value



- Does not account for network or processing latency

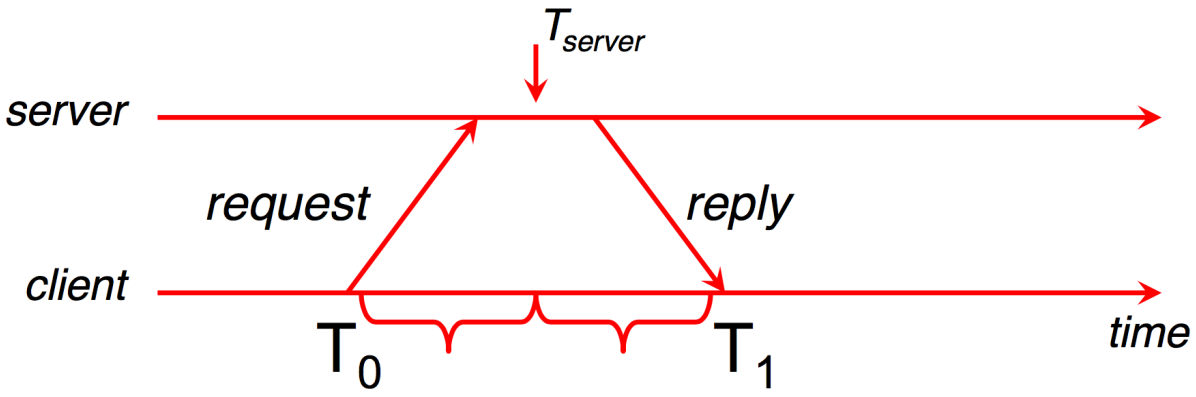
Cristian's algorithm

- Compensate for delays, note times:
  - request sent:T0
  - reply received:T1



- Assume network delays are symmetric

Cristian's algorithm



- Client sets time to: $T_{new} = T_{server} + \frac{T_1 - T_0}{2}$

Berkeley Algorithm

- Assumes no machine has an accurate time source
- Obtains average from participating computers

- Synchronizes all clocks to average

Berkeley Algorithm

- Machines run time dæmon
  - Process that implements protocol

- One machine is elected (or designated) as the server (master)
  - All others are slaves

Berkeley Algorithm

- 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

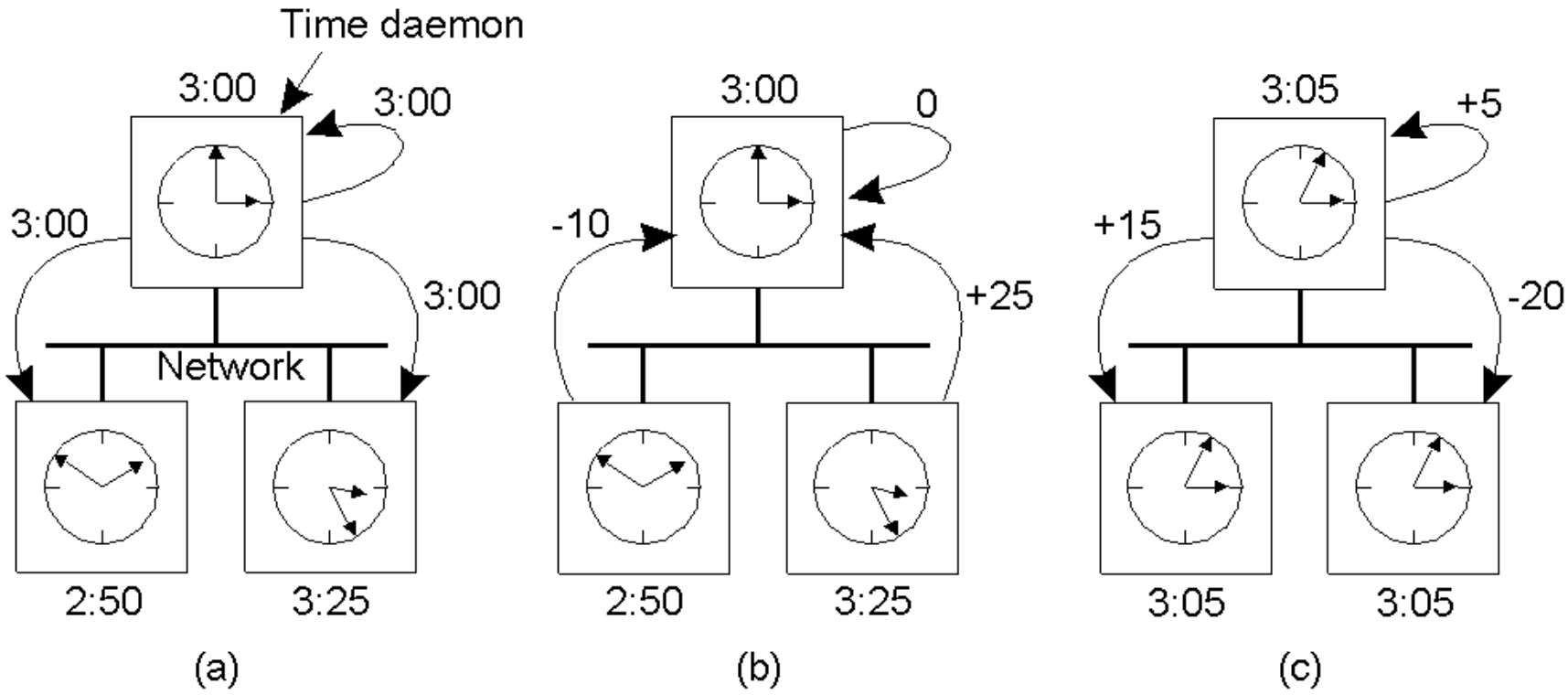
- We hope: an average cancels out individual clock's tendencies to run fast or slow
- Send offset by which each clock needs adjustment to each slave

Berkeley Algorithm

- Algorithm has provisions for ignoring readings from clocks whose skew is too great

- If master fails
- – Any slave can take over via an election algorithm

The Berkeley Algorithm



a) The time daemon asks all the other machines for their clock values

b) The machines answer

c) The time daemon tells everyone how to adjust their clock

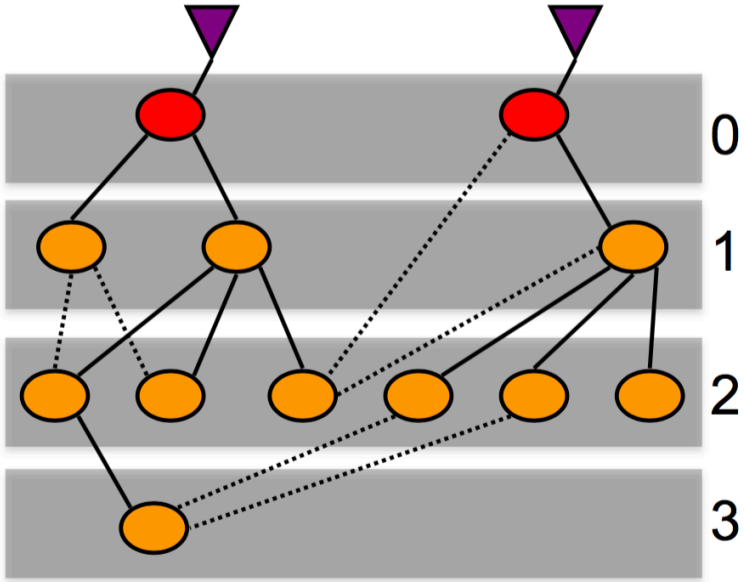
Network Time Protocol, NTP

- 1991, 1992
  – Internet Standard, version 3: RFC 1305
- June 2010
  – Internet Standard, version 4: – IPv6 support

- Improve accuracy to tens of microseconds

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



Simple Network Time Protocol

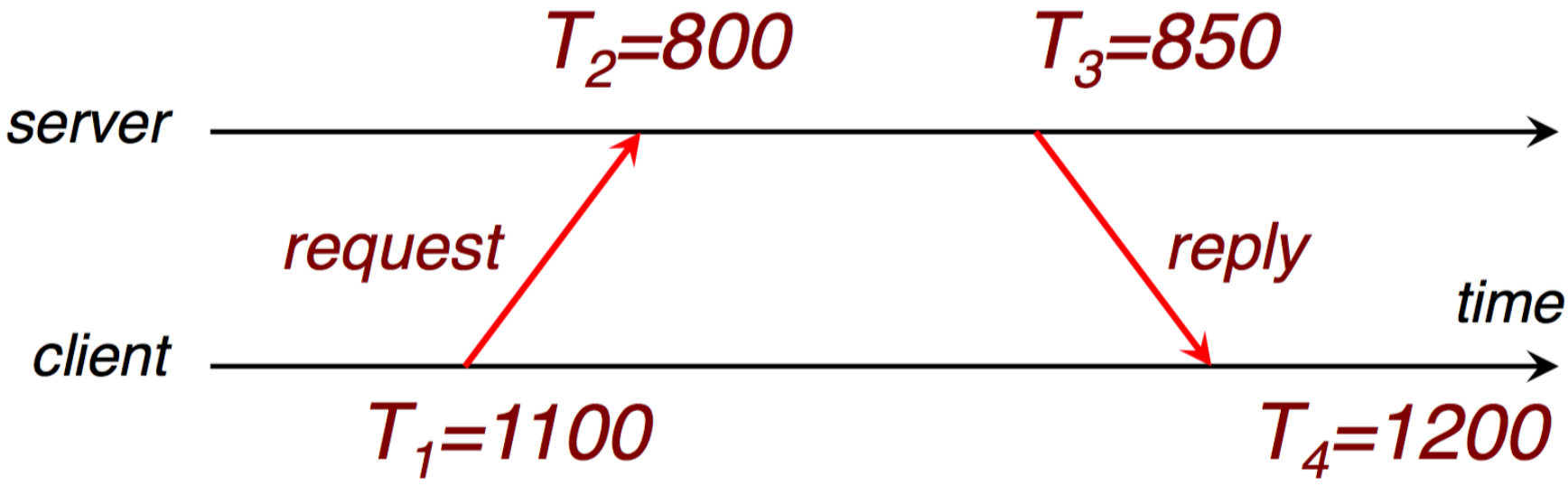
- A machine will often try to synchronize with several servers, using the best of all the results to set its time.

- The *best* result is a function of a number of qualities, including: round- trip delay, consistency of the delay, round-trip error, server's stratum, the accuracy of the server's clock, the last time the server's clock was synchronized, and the estimated drift on the server.

Simple Network Time Protocol

- The roundtrip delay and local offset are calculated as follows:
  - The client sets the transmit timestamp in the request to the time of day according to the client clock. (T1).
  - The server copies this field to the originate timestamp in the reply and sets the receive timestamp and transmit timestamps to the time of day according to the server clock (T2, T3).
  - When the server reply is received, the client determines a destination timestamp as the time of arrival according to its clock (T4).

SNTP Example



The local clock offset *t* is defined as: *t*=((T2 –T1)+(T3 –T4))/2

The client, after computing this offset, adds this amount to its clock.

Clock synchronization

- Physical clocks keep time of day – Ideally should be consistent across systems

- Logical clock keeps track of event ordering among related (causal) events

Logical clocks

- Our real-world view of clock synchronization is one of ensuring that multiple processes on multiple machines all see the same time of day.

- All computers have a time-of-day clock and synchronization becomes a matter of keeping these clocks set to the same value.

Logical clocks

- Having synchronized clocks is extremely useful.
- However, it is not always sufficient
  - First, the time setting has limited precision.
  - Second, just by looking at two timestamps, you cannot tell if one event may be the result of another event or if they are completely independent.

Logical clock is one where the clock does not have any bearing on the time of day but rather is a number that can be used for comparing sets of events, such as a messages, within a distributed system.

Logical clocks

- In some cases, What matters is not the time of day at which the event occurred but that all processes can agree on the *order* in which related events occur.

- Our interest is in getting event sequence numbers that make sense system-wide.

- If we can do this across all events in the system, we have something called ***total ordering***

Logical clocks

- If we can do this across all events in the system, we have something called ***total ordering***
  - every event is assigned a unique timestamp (number) and every such timestamp is unique.

- We don't always need total ordering.
  - If processes do not interact then we don't care when their events occur.
  - If we only care about assigning timestamps to potentially related (*causal*) events then we have something known as *partial ordering*.

Lamport Clocks

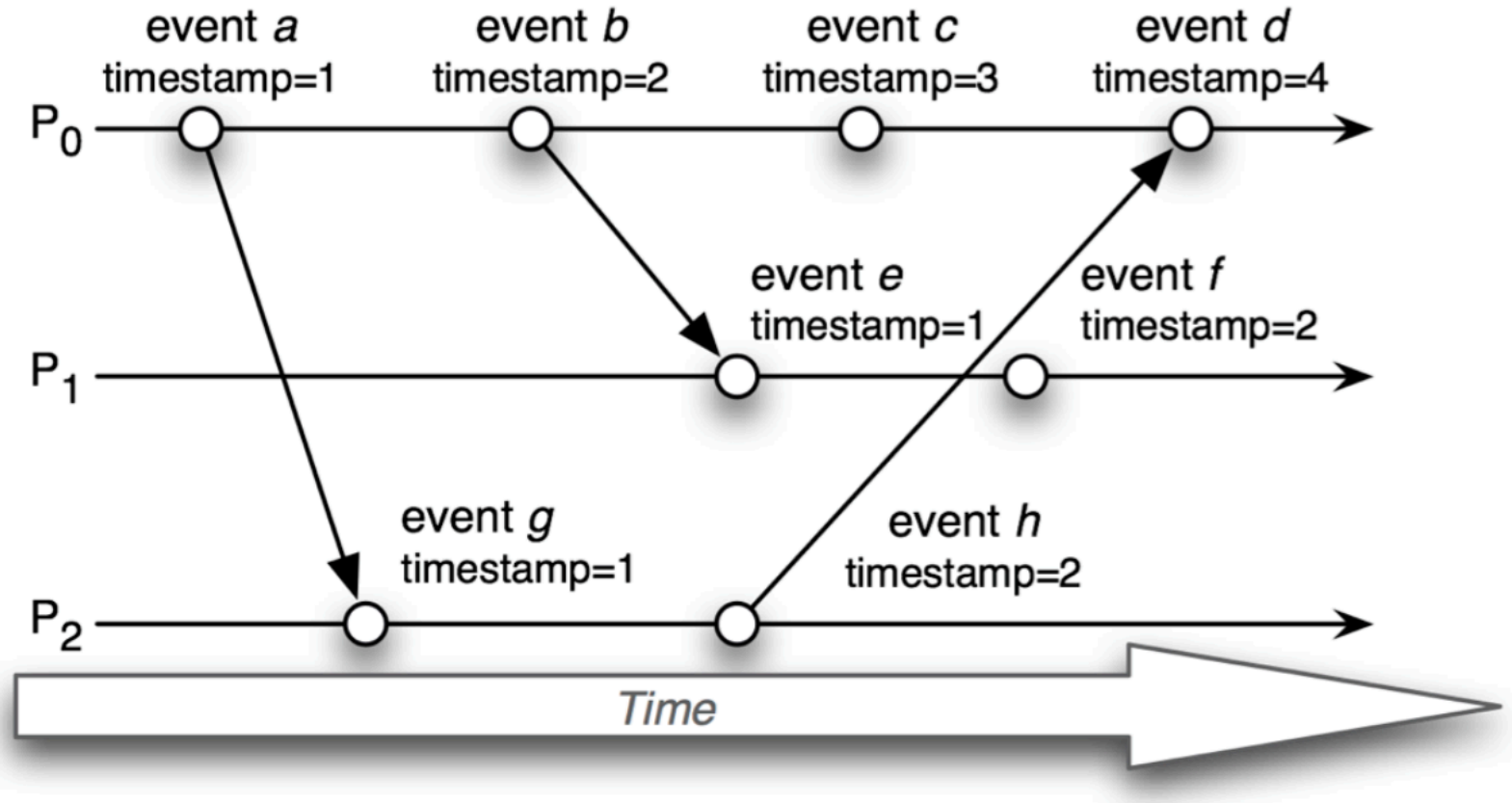
- Leslie Lamport defined a *happens before* notation to express the relationship between events:
  - $a \rightarrow b$ means that *a* happens before *b*.
  - This relationship is transitive. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Lamport Clocks

- Assign “clock” value to each event
  - if $a \rightarrow b$ then clock(a) < clock(b)
  - since time cannot run backwards

- If a and b occur on different processes that do not exchange messages, neither $a \rightarrow b$ nor $b \rightarrow a$ is true
  - These events are concurrent there is no way that *a* could have influenced *b*

Event ordering example



Event ordering example

- we can observe a number of peculiarities.
  - Event *g*, the event representing the receipt of the message sent by event *a*, has the exact same timestamp as event *a* when it clearly had to take place *after* event *a*.
  - Event *e* has an earlier time stamp (1) than the event that sent the message (*b)*, with a timestamp of 2).

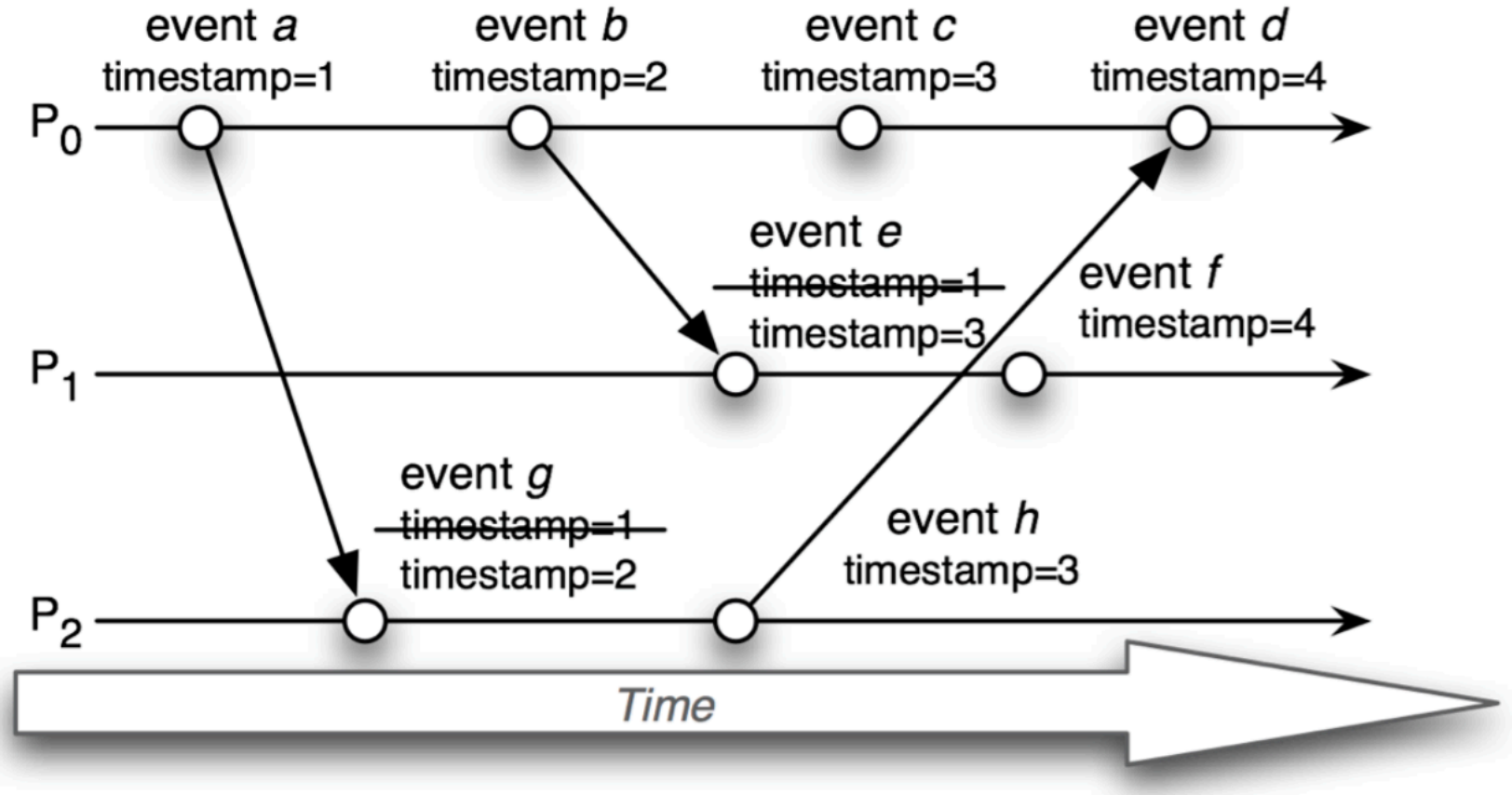
Lamport's algorithm

- Each process has its own clock, which can be a simple counter that is incremented prior to each event.
- Each message carries a timestamp of the sender's clock

- When a message arrives:
  - if receiver's clock < message timestamp
    set system clock to (message timestamp + 1)
  - else do nothing

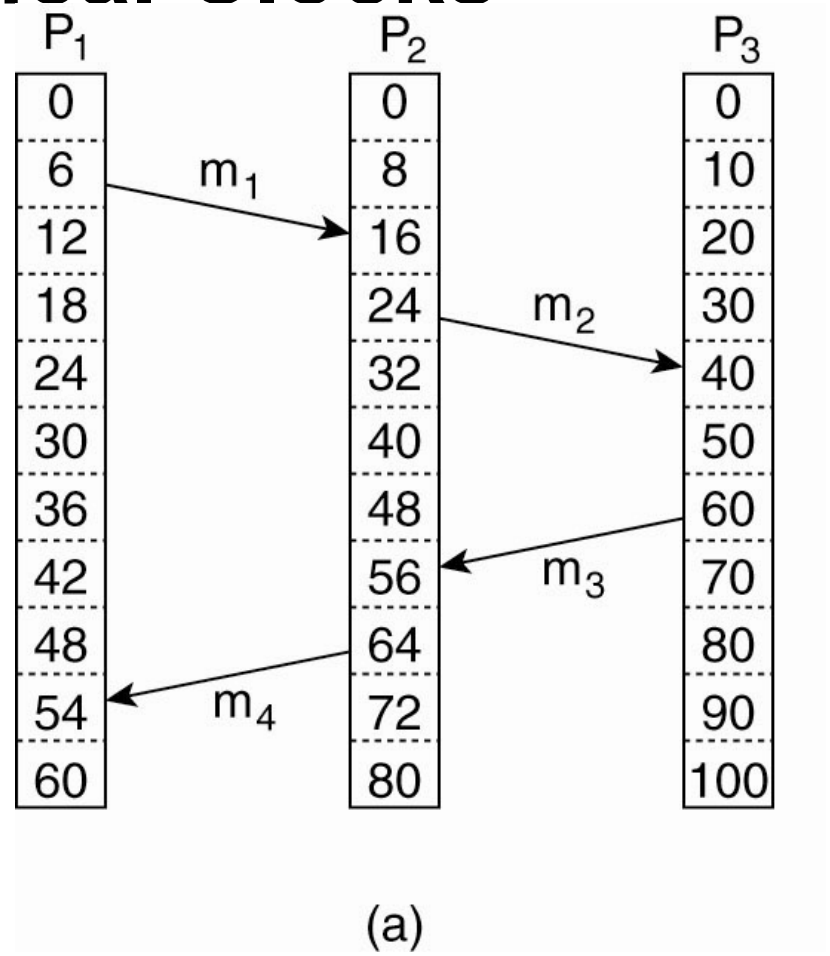
- Clock must be advanced between any two events in the same process

Lamport's algorithm



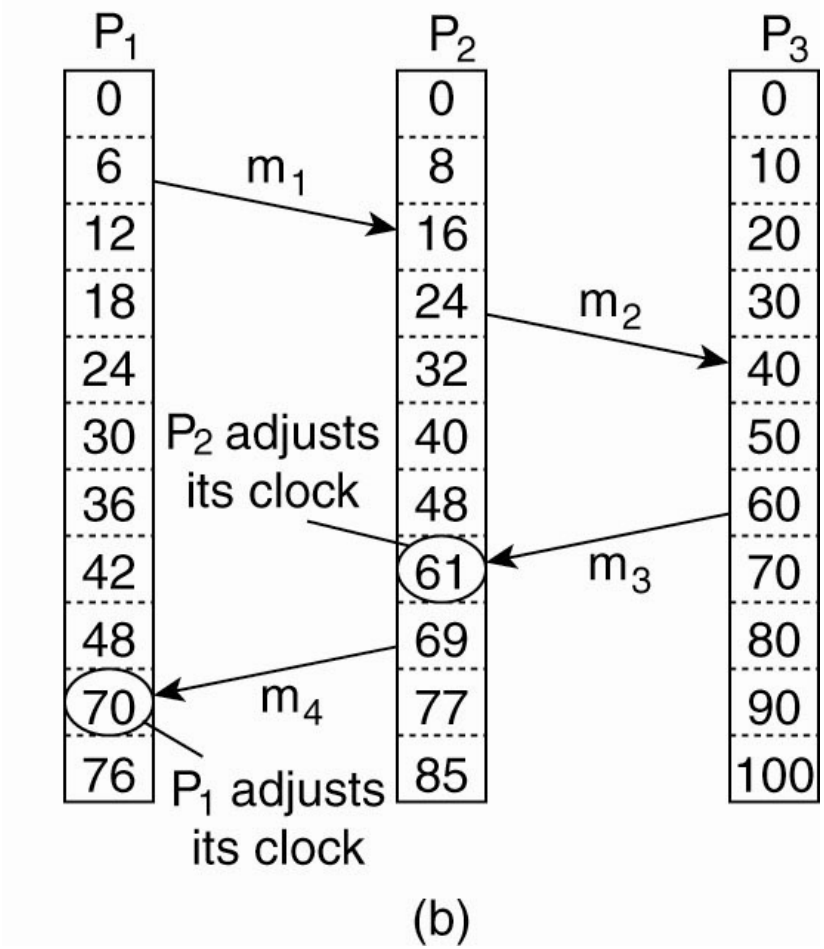
Lamport's Logical Clocks

Event a: P1 sends m1 to P2 at t = 6,
Event b: P2 receives m1 at t = 16.
If C(a) is the time m1 was sent, and C(b) is the time m1 is received, do C(a) and C(b) satisfy the correctness conditions ?



(a)

Lamport's Logical Clocks (3)



Event c: P3 sends m3 to P2 at t = 60
Event d: P2 receives m3 at t = 56
Do C(c) and C(d) satisfy the conditions?

Lamport's algorithm - Summary

- Algorithm needs increasing software counter
- Incremented at least when events that need to be timestamped occur
- Each event has a Lamport timestamp attached to it
- For any two events, where a ▫ b:

  L(a) < L(b)

Lamport's algorithm

- Total ordering vs Partial ordering?

Lamport's algorithm

- it is very possible for multiple non-causal (concurrent) events to share identical Lamport timestamps

Lamport's algorithm

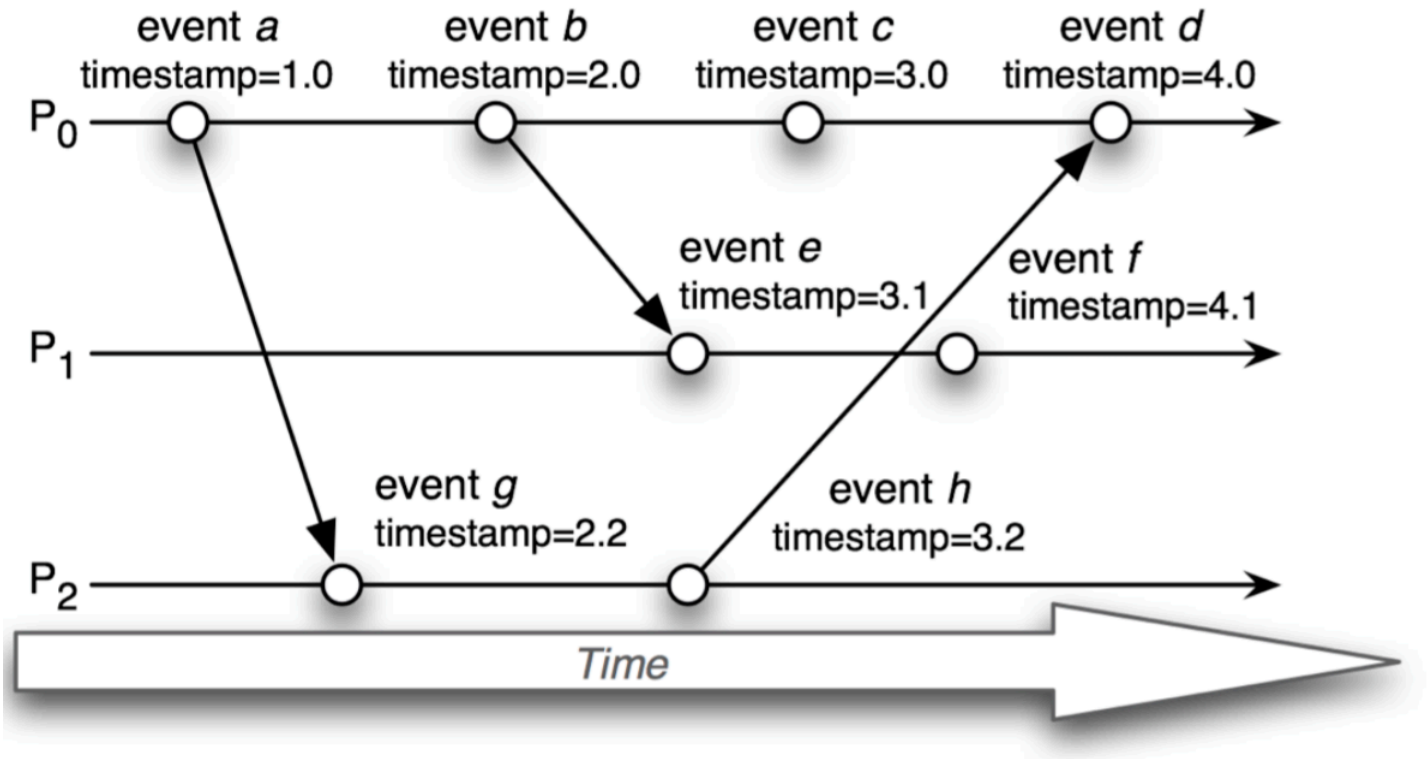
- This may cause confusion if multiple processes need to make a decision based on the received timestamps of two concurrent events.

- The selection of one event over the other may not matter if the events are concurrent but we want all processes to make the same decision.

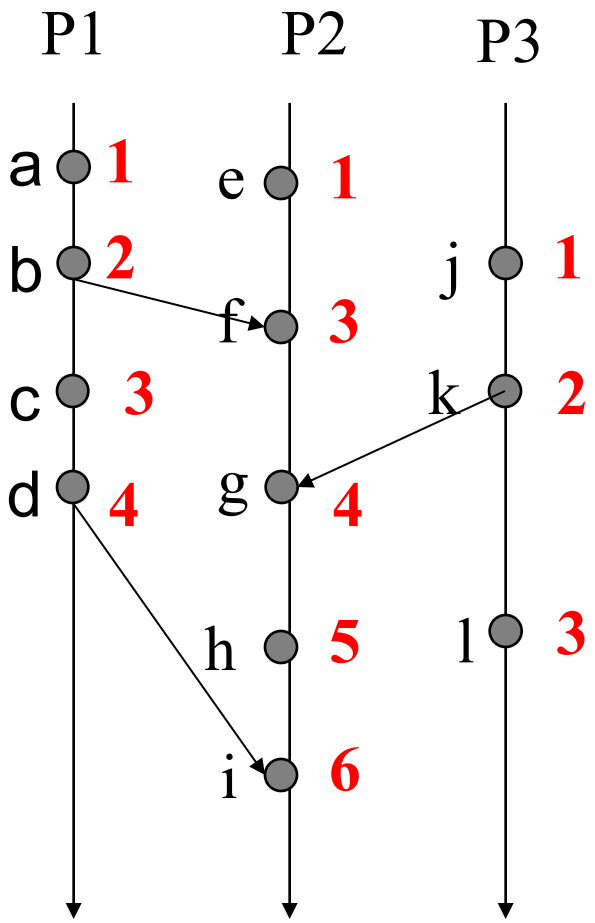
- This is difficult if the timestamps are identical.

Lamport's algorithm - Total ordering

- We can create a total order on events by further qualifying them with process ID numbers.



Example



Assume that each process's logical clock is set to 0

Example

- From the timing diagram on the previous slide, what can you say about the following events?
  - Between *a* and *b:* **$a \rightarrow b$**
  - Between *b* and *f*: **$b \rightarrow f$**
  - Between *e* and *k*: **concurrent**
  - Between *c* and *h*: **concurrent**
  - Between *k* and *h*: **$k \rightarrow h$**

Total Order

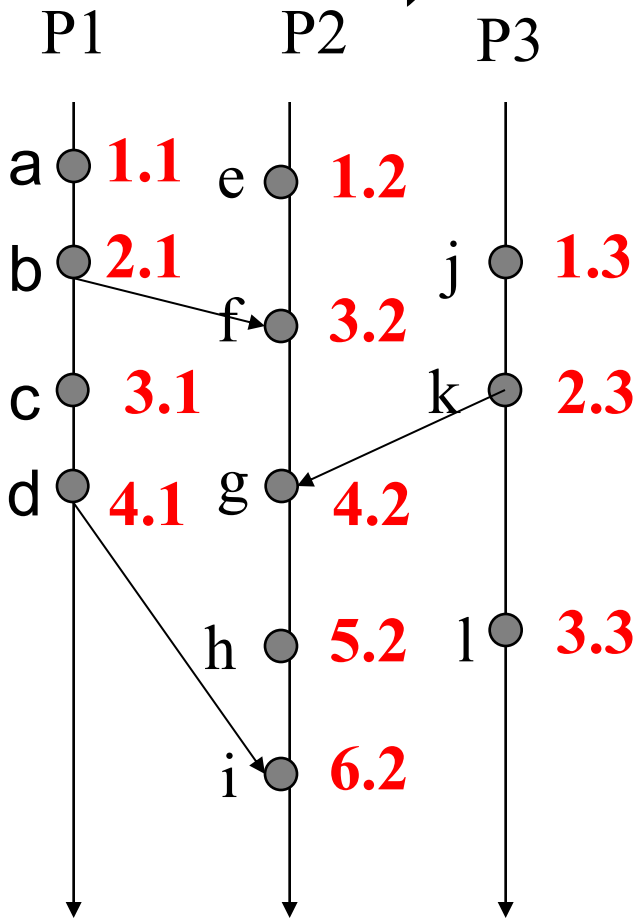
- A timestamp of 1 is associated with events *a*, *e*, *j* in processes P_1, P_2, P_3 respectively.
- A timestamp of 2 is associated with events *b*, *k* in processes P_1, P_3 respectively.

- The times may be the same but the events are distinct.
- We would like to create a total order of all events

Total Order

- Create total order by attaching a process number to an event.
- P_i timestamps event e with C_i (e).i
- We then say that C_i(a).i happens before C_j(b).j iff:
  - $C_i(a) < C_j(b)$; or
  - $C_i(a) = C_j(b)$ and $i < j$

Example (total order)



Assume that each process's logical clock is set to 0

Vector clocks

- If two events are causally related and event *e* happened before event *e'* then we know that $L(e) < L(e')$.

- However, the converse is not necessarily true. if $L(e) < L(e')$ we *cannot* conclude that $e \rightarrow e'$.

Problem with Lamport Logical Clock

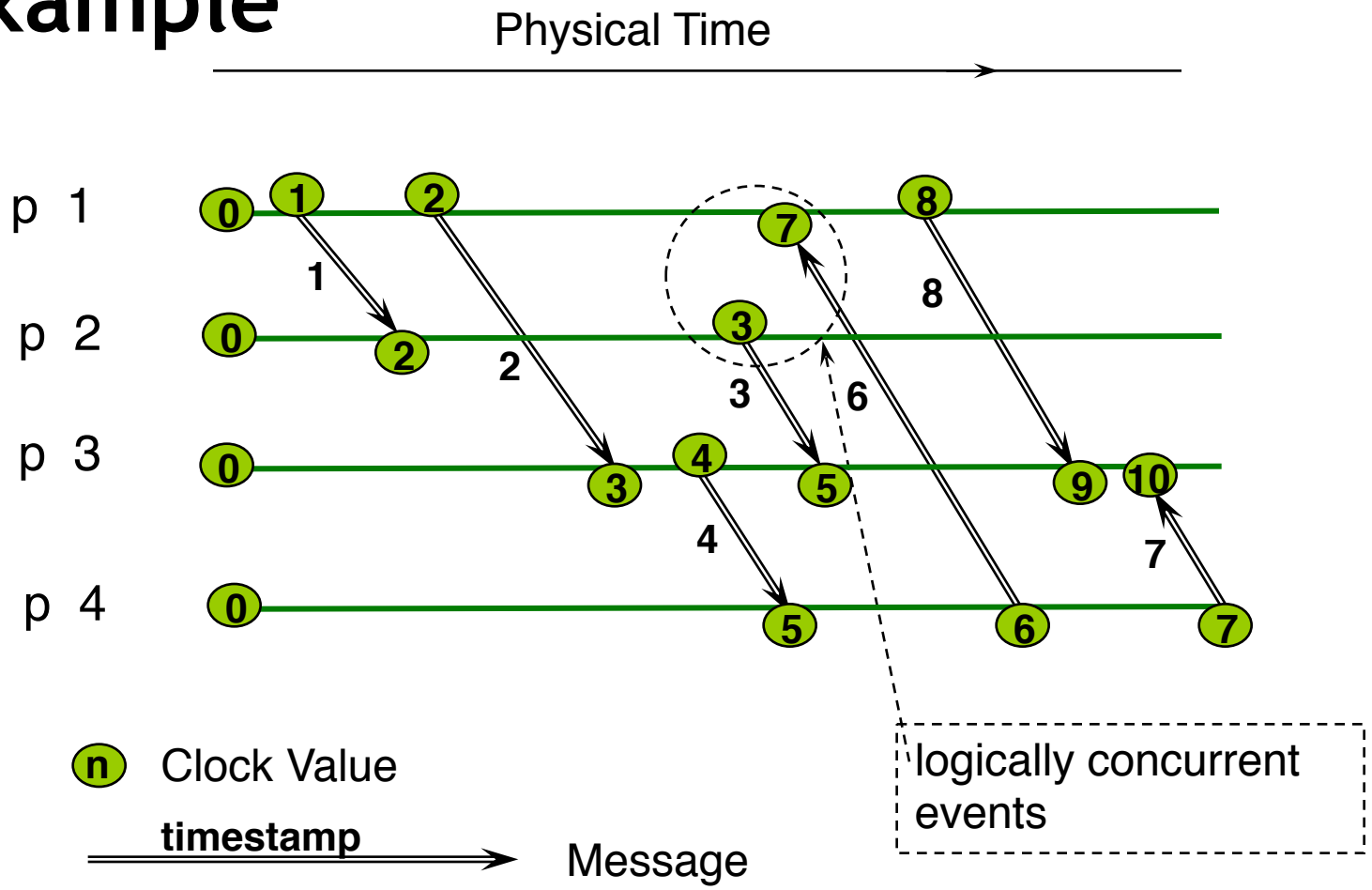
a → *b* => *timestamp(a)* < *timestamp(b)*

(if *a* happens before *b*, then *Lamport_timestamp(a)* < *Lamport_timestamp(b)*)

timestamp(a) < *timestamp(b)* ~~=>~~ *a* → *b*

(If *Lamport_timestamp(a)* < *Lamport_timestamp(b),* it does NOT imply that *a* happens before *b*

Example



Note: Lamport Timestamps: 3 < 7, but event with timestamp 3 is concurrent to event with timestamp 7, i.e., events are not in 'happen-before' relation.

Vector Logical Clocks

- A vector clock in a system of *N* processes is a vector of *N* integers.
- Each process maintains its own vector clock (V_i for a process P_i) to timestamp local events.

- Like Lamport timestamps, vector timestamps (the vector of *N* integers) are sent with each message.

Vector Logical Clocks

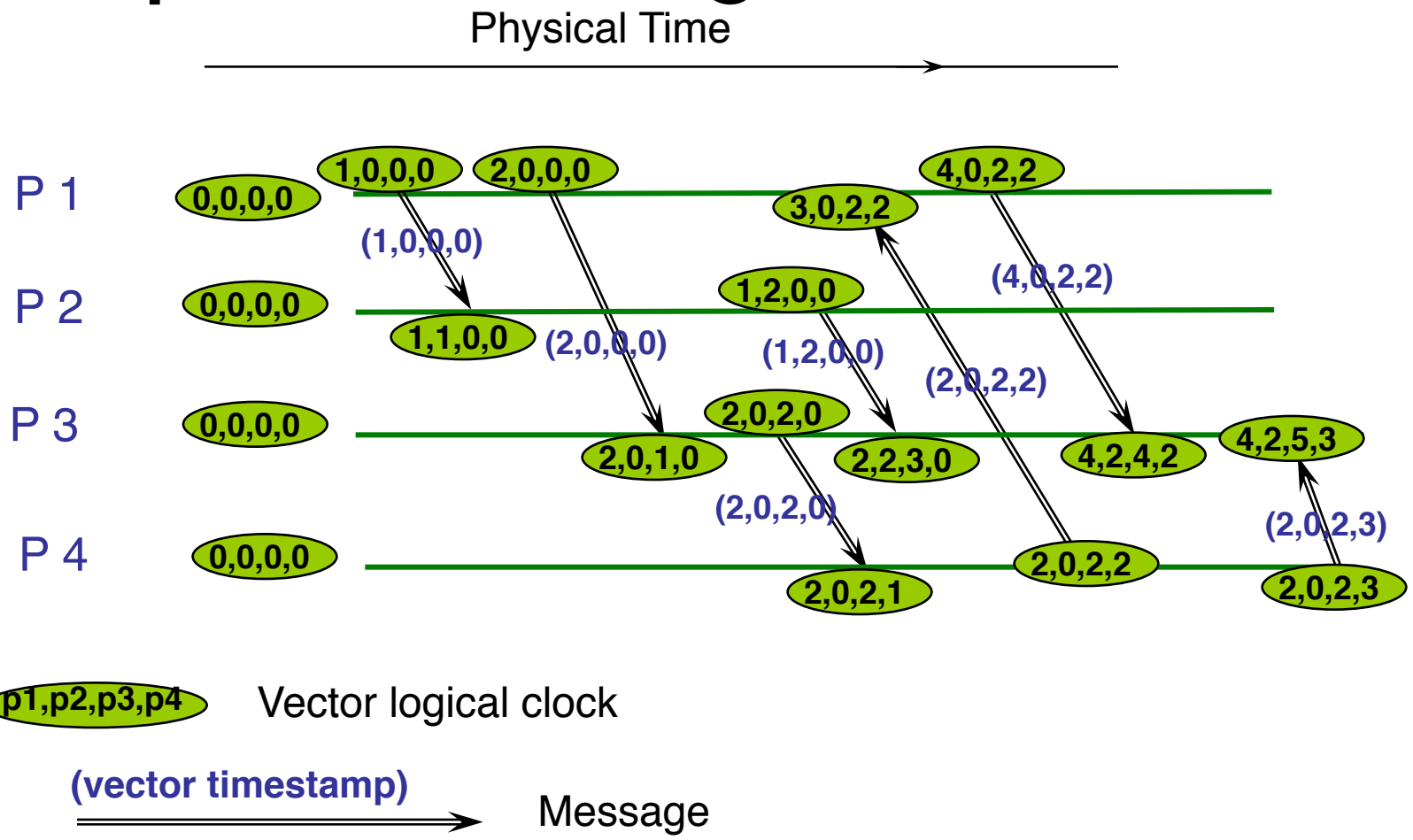
1. The vector is initialized to 0 at all processes:
   – $V_i[j]=0$ for $i,j=1,...,N$
2. Before a process P_i timestamps an event, it increments its element of the vector in its local vector:
   – $V_i[i] = V_i[i]+1$
3. A message is sent from process P_i with V_i attached to the message.

Vector Logical Clocks

4. When a process P_j receives a vector timestamp t, it compares two vectors, element by element, setting its local vector clock to the higher one:
   - $V_j[i] = max(V_j[i], t[i])$ for $i=1, ..., N$

Further the process executes the first step and delivers the message to the application.

Example: Vector Logical Time



Vector Logical Clocks

- For any two events *e*, *e'*, if $e \rightarrow e'$ then $V(e) < V(e')$. This is the same as we get from Lamport's algorithm.

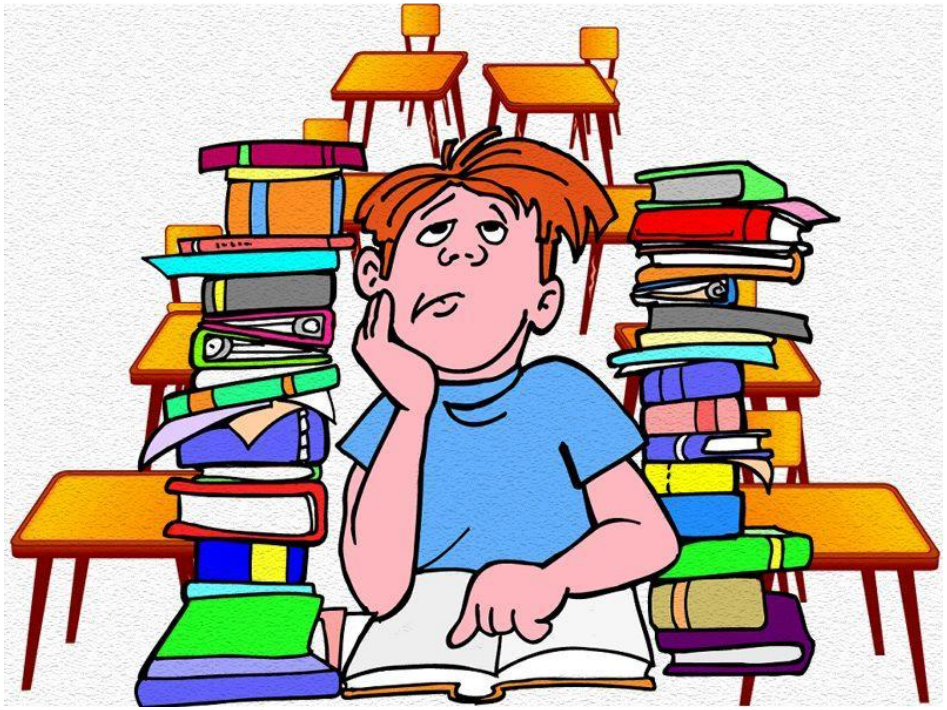
- With vector clocks, we now have the additional knowledge that if $V(e) < V(e')$ then $e \rightarrow e'$.
- Two events *e*, *e'* are concurrent if *neither* $V(e) \leq V(e')$ nor $V(e') \leq V(e)$.

Comparing Vector Timestamps

- **Less than**: ts(a) < ts(b) iff at least one component of ts(a) is strictly less than the corresponding component of ts(b) and all other components of ts(a) are either less than or equal to the corresponding component in ts(b).
- (3,3,5) ≤ (3,4,5), (3, 3, 3) = (3, 3, 3),
  (3,3,5) ≥ (3,2,4), (3, 3 ,5) | | (4,2,5).

Review

Vector Clocks?



The Consensus Problem

- The consensus problem in distributed systems is the problem of getting a set of nodes to agree on something

- Some examples include:
  - Which process is the leader of a group of processes?
  - What is a value in a datastore?
  - whether or not a distributed lock is held?
  - …

Mutual Exclusion

- n processes all competing to use some shared data / resource

- Problem: ensure that
  – Never two process are allowed to execute in their critical section at the same time

- For distributed systems, locking and many other related aspects are the different versions of the consensus algorithm.

Why consensus is difficult?

- Nodes may crash, they may even be malicious
- Network is unreliable and latency can be problematic
- Network may have partitions and not all pairs connected
- …

The CAP Theorem

- A distributed system cannot simultaneously have all three of the following properties:
  - Consistent views of the data at each node
  - Availability of the data at each node
  - Partitions resistence

- Common misconception, 2/3 is possible!

Explain Bitcoin Like I'm Five Nik Custodio

- We're sitting on a park bench. It's a great day.
- I have one apple with me. I give it to you.
- You now have one apple and I have zero.

- That was simple, right?

Explain Bitcoin Like I'm Five Nik Custodio

- My apple was physically put into your hand.
- We didn't need a ***third person*** there to help us make the transfer.

- We didn't need to pull in Uncle Tommy (who's a famous judge) to sit with us on the bench and confirm that the apple went from me to you.

What about a Digital Apple?

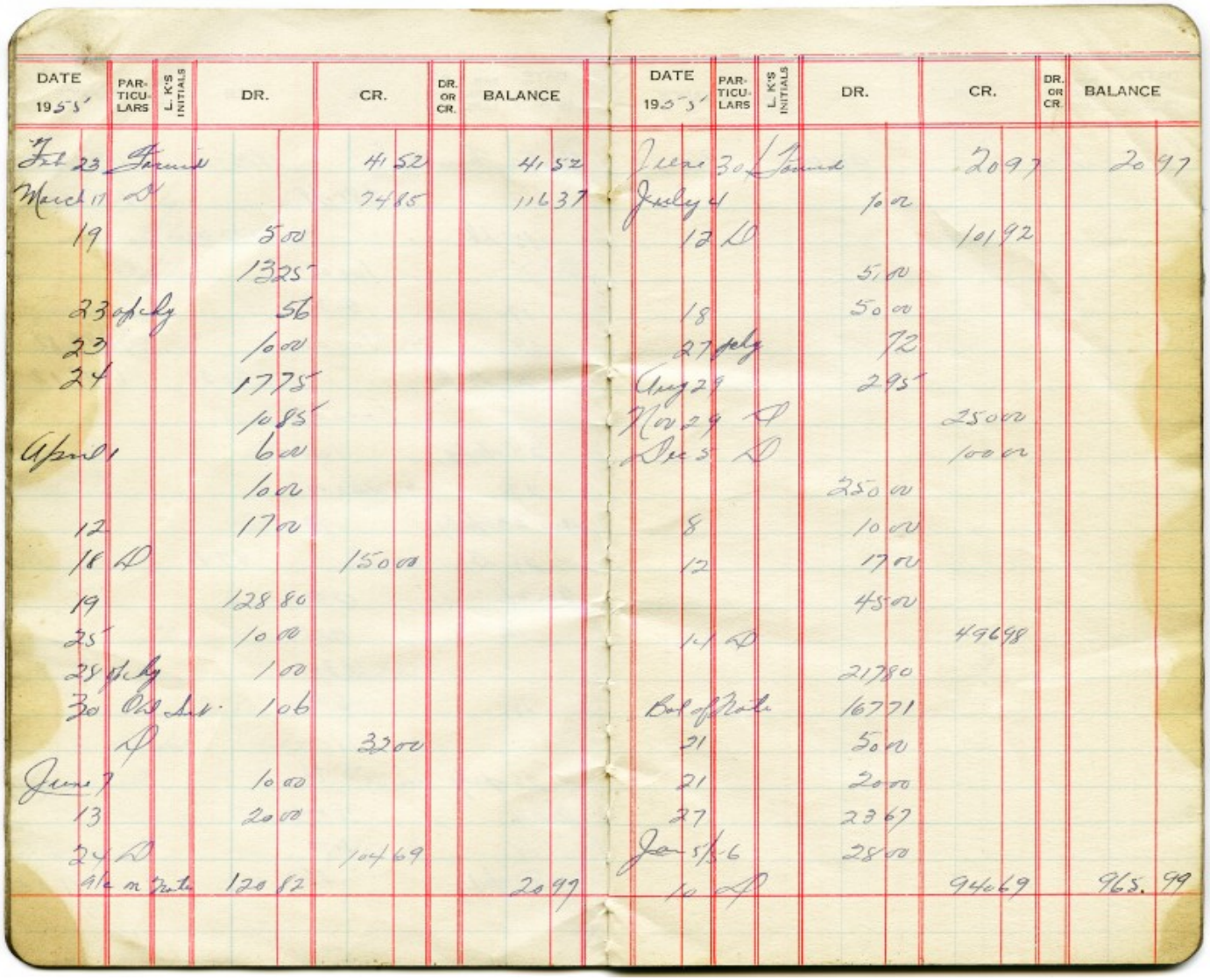
What about a Digital Apple?

- Now say, I have one digital apple. I can give you my digital apple by sending you the image over email or even Whatsapp.

- ***The apple's yours! I can't control it anymore. You have full control over that apple now. Really?***

- How do you know that that digital apple that used to be mine, is now yours, and only yours? Think about it for a second.

One possible solution



Ledgers

- Maybe these digital apples need to be tracked in a ledger. It's basically a book where you track all transactions — an accounting book.

- This ledger, since it's digital, needs to live in its own world and have ***someone*** in charge of it.

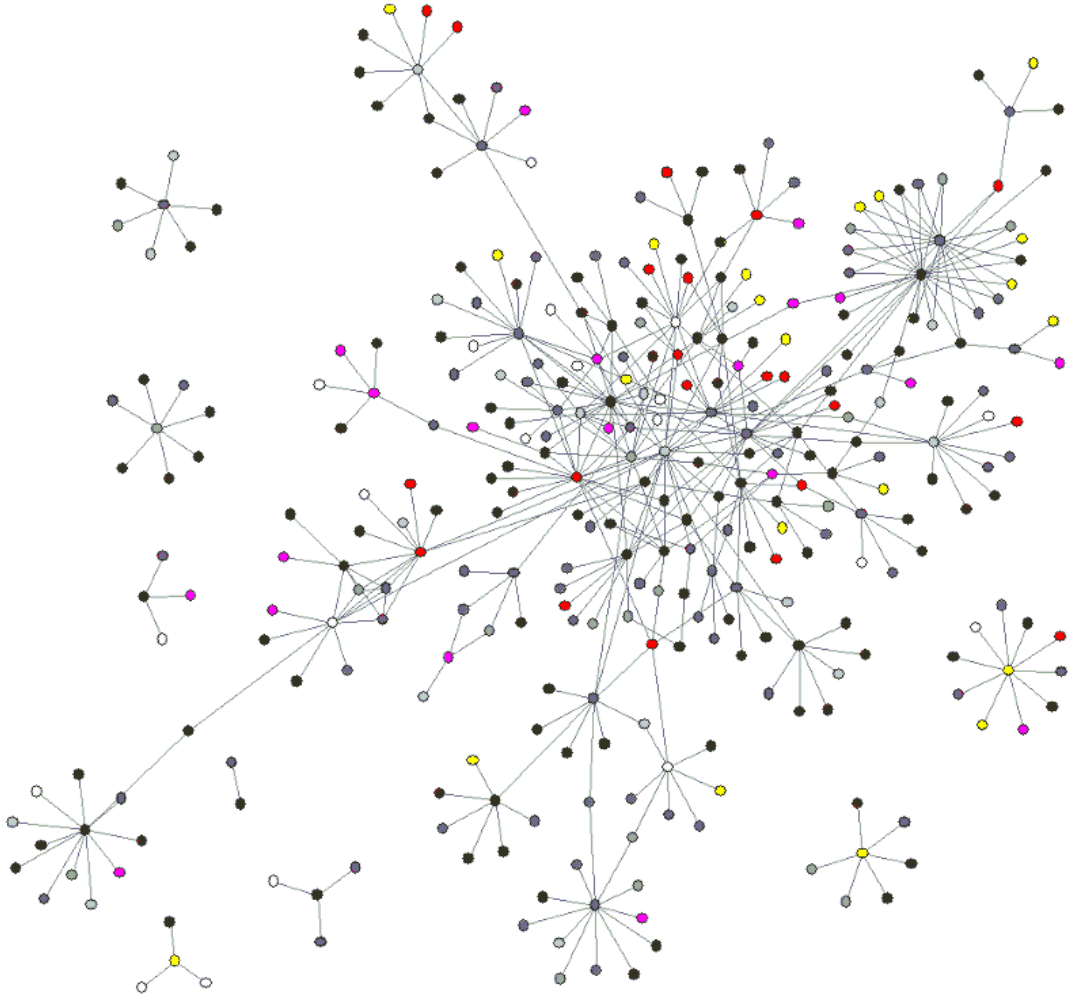
Centralized Ledgers

- What if that ***someone*** cheats? He could just add a couple of digital apples to his balance whenever he wants!

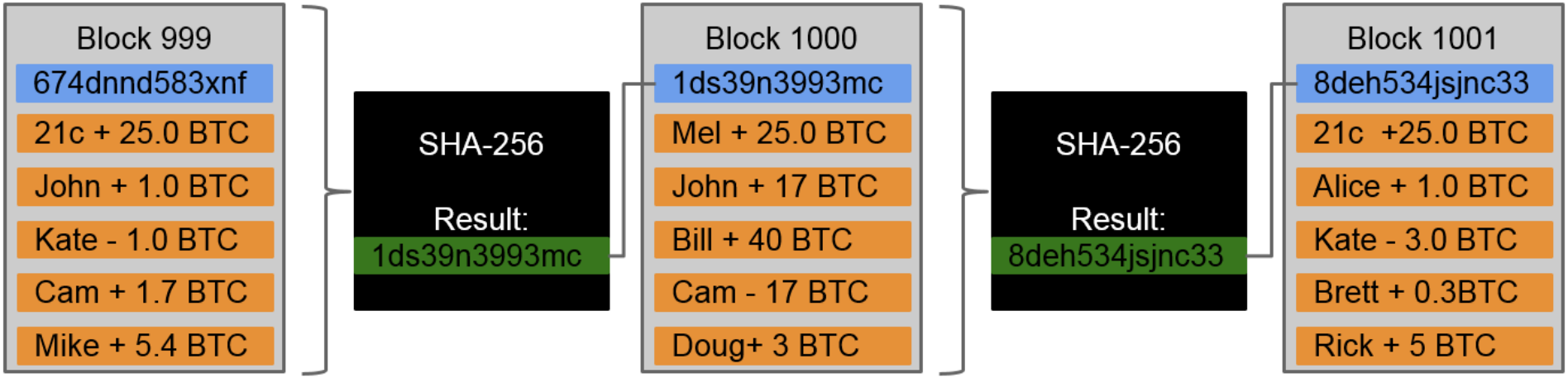
What else can be done?

- What if we gave this ledger — to everybody?
- Instead of the ledger living on someone's computer, it'll live in everybody's computers.
- All the transactions that have ever happened, from all time, in digital apples will be recorded in it.

Decentralized Ledgers



What does a ledger look like?



Who will add to this ledger?

1. *New transactions are broadcast to all nodes*
2. *Each node collects new transactions into a block*
3. *In each round a random node gets to broadcast its block*

Question: How to ensure randomness andhandle collisions?

… back to distributed consensus

- Some well-known algorithms include Paxos and RAFT.

http://thesecretlivesofdata.com/raft/