Distributed Systems: Time

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

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

Agenda

- Clocks & time in computers
- Synchronizing physical clocks
- Logical clocks
  - Lamport clocks
  - Vector clocks

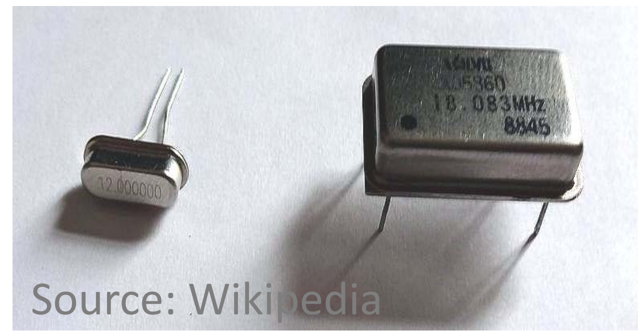
Why is time important?

- In distributed systems, we require…
  - **Cooperation** between nodes: must **agree** on certain things
  - **High degree of parallelism**: nodes should work independently to achieve the most progress
- Time gives us…
  - **A point of reference** every machine knows how to keep track of...
  - Without explicit communication!
- However…
  - Time-keeping is not perfect ☹
  - How do we efficiently synchronize time to achieve our goals?

Computer “clocks”

Computer clocks count oscillations of a crystal at a defined frequency

Crystal oscillator (quartz crystal)



Counter register

N

N-1

N-2

....

0

Decrements counter for each oscillation.

Interrupt

(one clock tick)

Holding register (N)

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

Reload upon interrupt

Real time clock (RTC, CMOSC, HWC)

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

- Referred to as “**wall clock**” time
  - Synchronizes the **system clock** when computer on
  - Should not be confused with **real-time computing**

- IRQ 8

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

Source: Wikipedia

In English: UTC - Coordinated Universal Time

Universal Time Coordinated (UTC)

Temps Universel Coordonné

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

Caesium-133 fountain atomic clock in Switzerland

Uncertainty of one second in 30 million Years!

(Probably the ``Rolex`` of atomic clocks ☺)

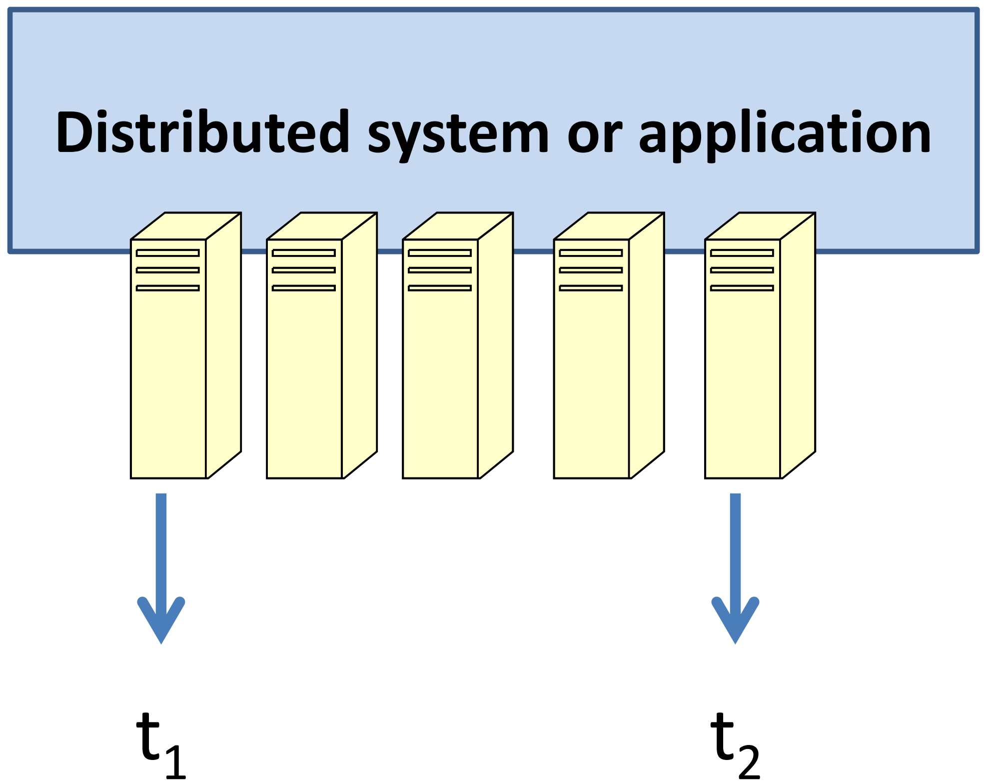
ROLEX

Atomic clocks

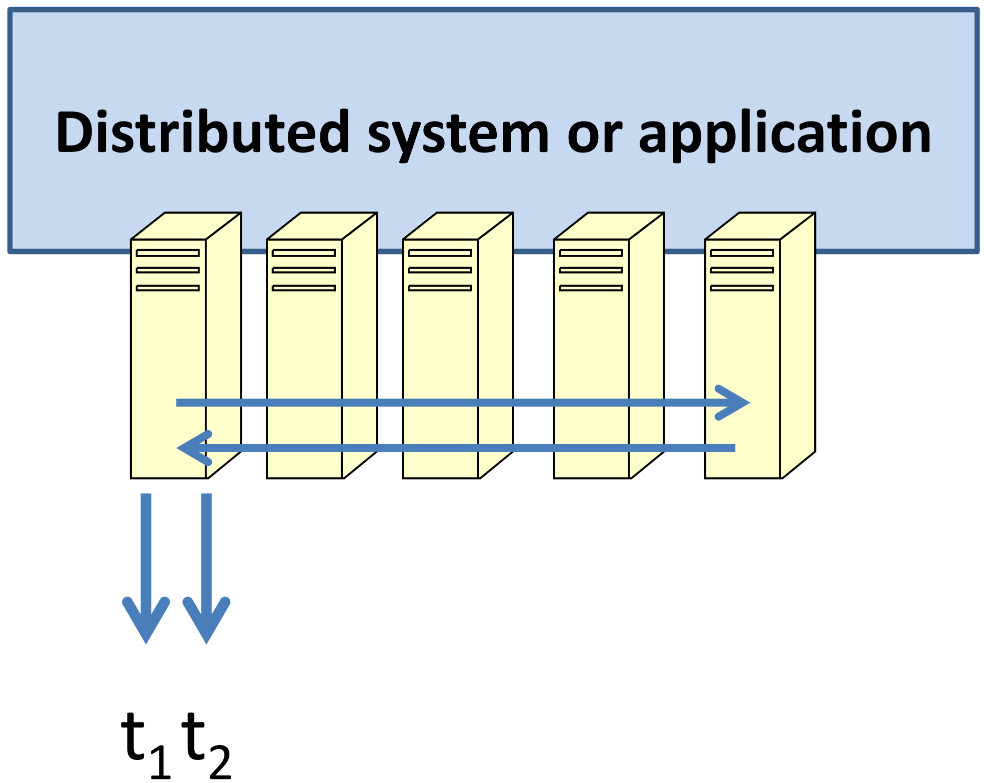
Atomic clock on the market May 11, 2011. Quoted $1500 with **an accuracy of less than 0.5 micro seconds per day.**

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

Measuring latency in distributed systems experiments



Measuring latency in distributed systems experiments



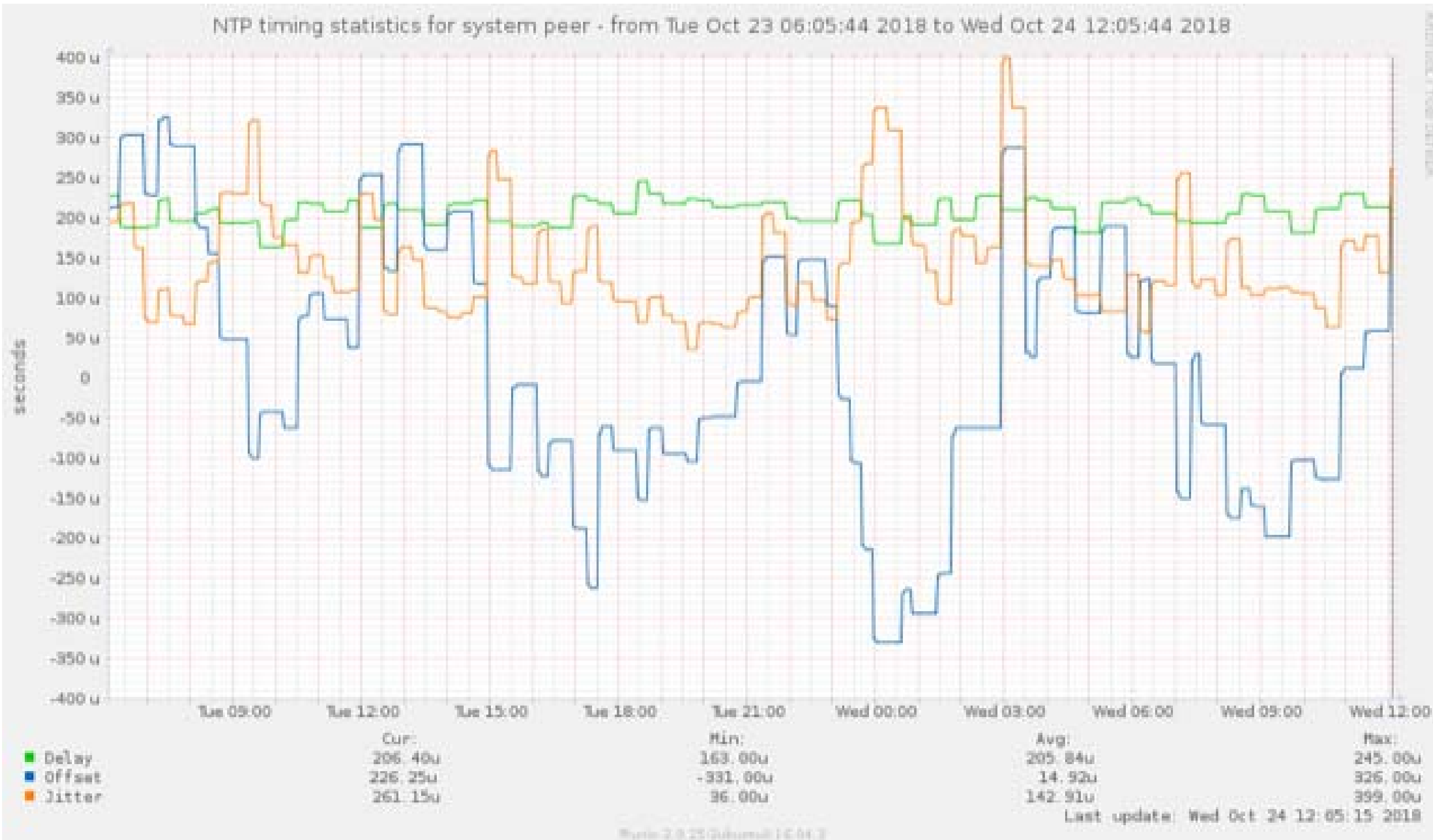
host=node-1 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-2 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-3 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-4 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-5 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-6 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-7 rtt=562(280)ms/0ms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

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

Software-based clock sync times with 1 millisecond precision

NTP timing statistic for single node (in µs)

NTP timing statistic in microseconds for delay, offset and jitter



Clock skew & drift

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

- **Clock drift**: Rate at which clock skew increases between a clock and some reference clock

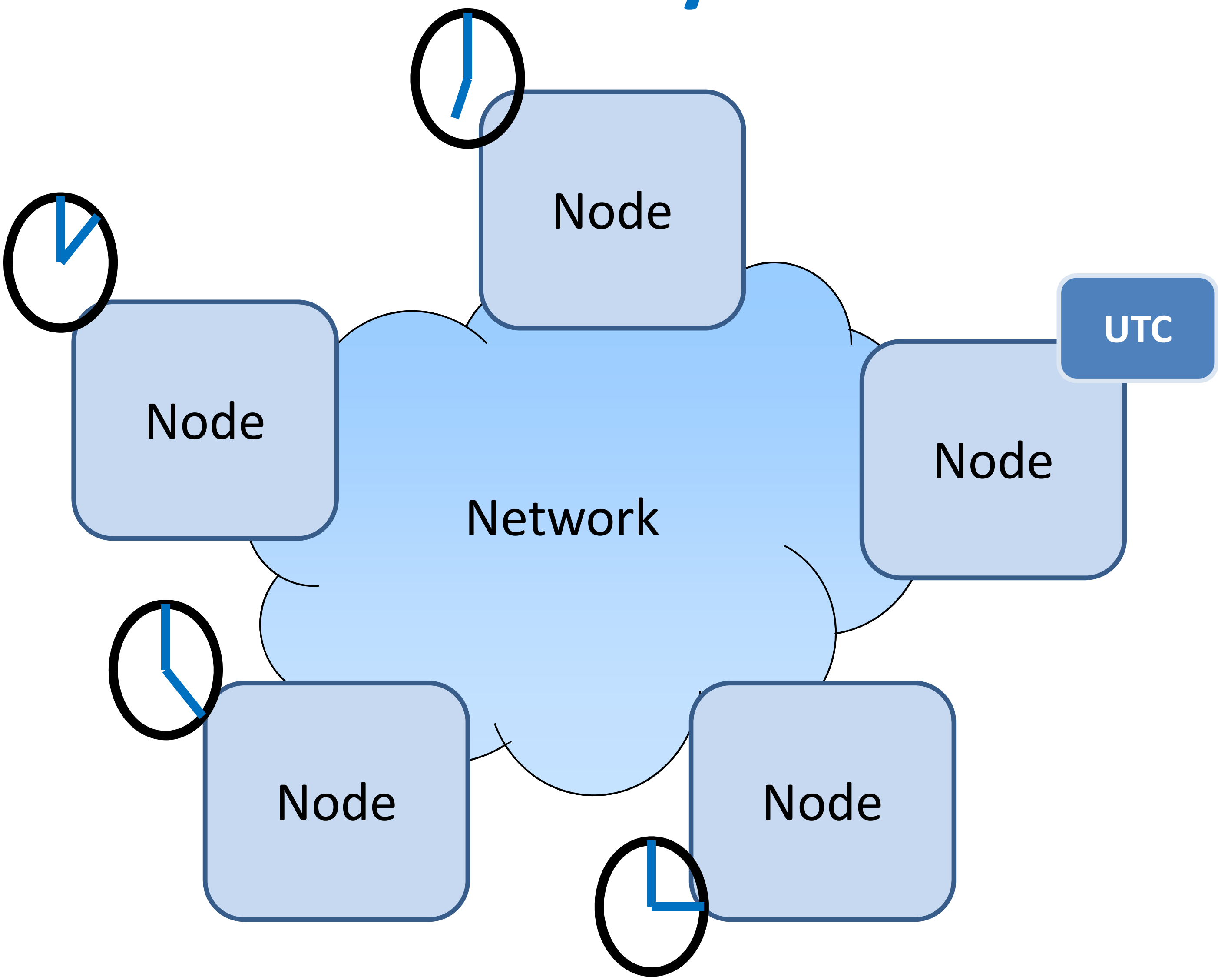
Synchronization mechanisms

- Hardware support: Radio receivers, GPS receivers, atomic clocks, shared backplane signals
  - Tight synchronization
    (+/- 10ns for GPS)
  - Negligible overhead
  - Costly
    Quartz is *good enough*

Clock synchronization

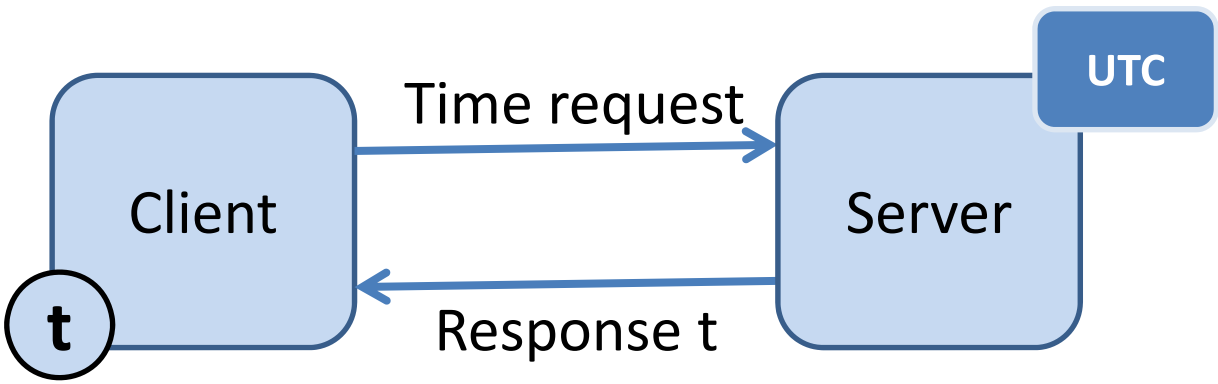
- External synchronization
- Internal synchronization

Problem: External synchronization



Synchronization request and reply

- Request to reference clock source for time
  - Request involves network round trip time (RTT)
  - Response is wrong by the time client receives it



- Client must adjust response based on knowledge of network RTT

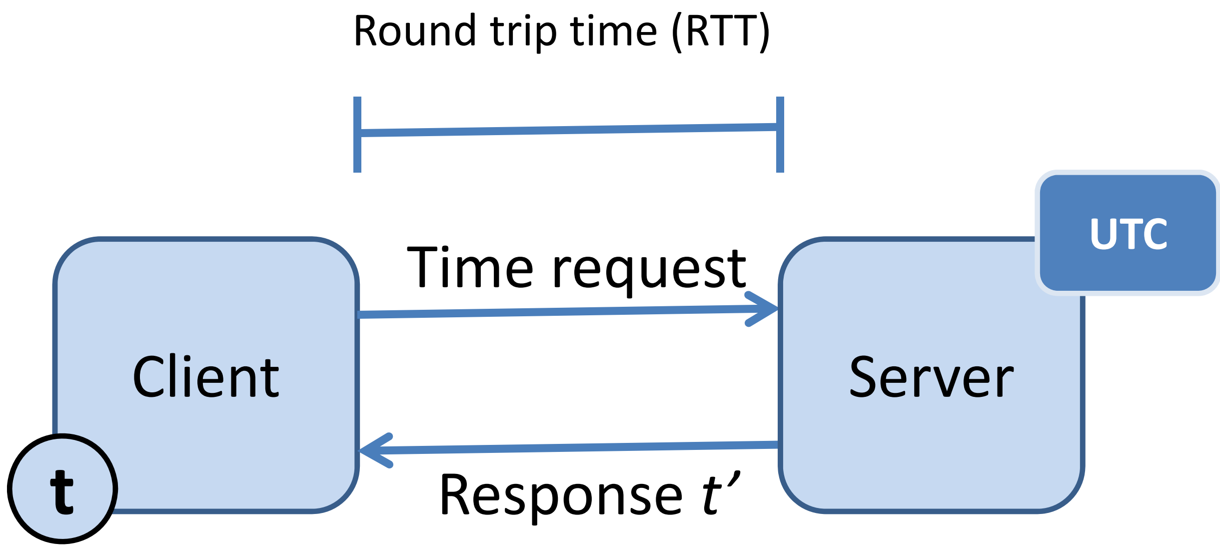
Probabilistic clock synchronization

Proposed by IBM, 1989

- **External clock synchronization**
- **Time server** connected to a time **reference source** (UTC)
- **Transmission time** is **unbounded**, but usually reasonably **short**
- Synchronization is achieved to the degree that network delays are short compared to desired accuracy
- Primarily intended for operation on **LANs**

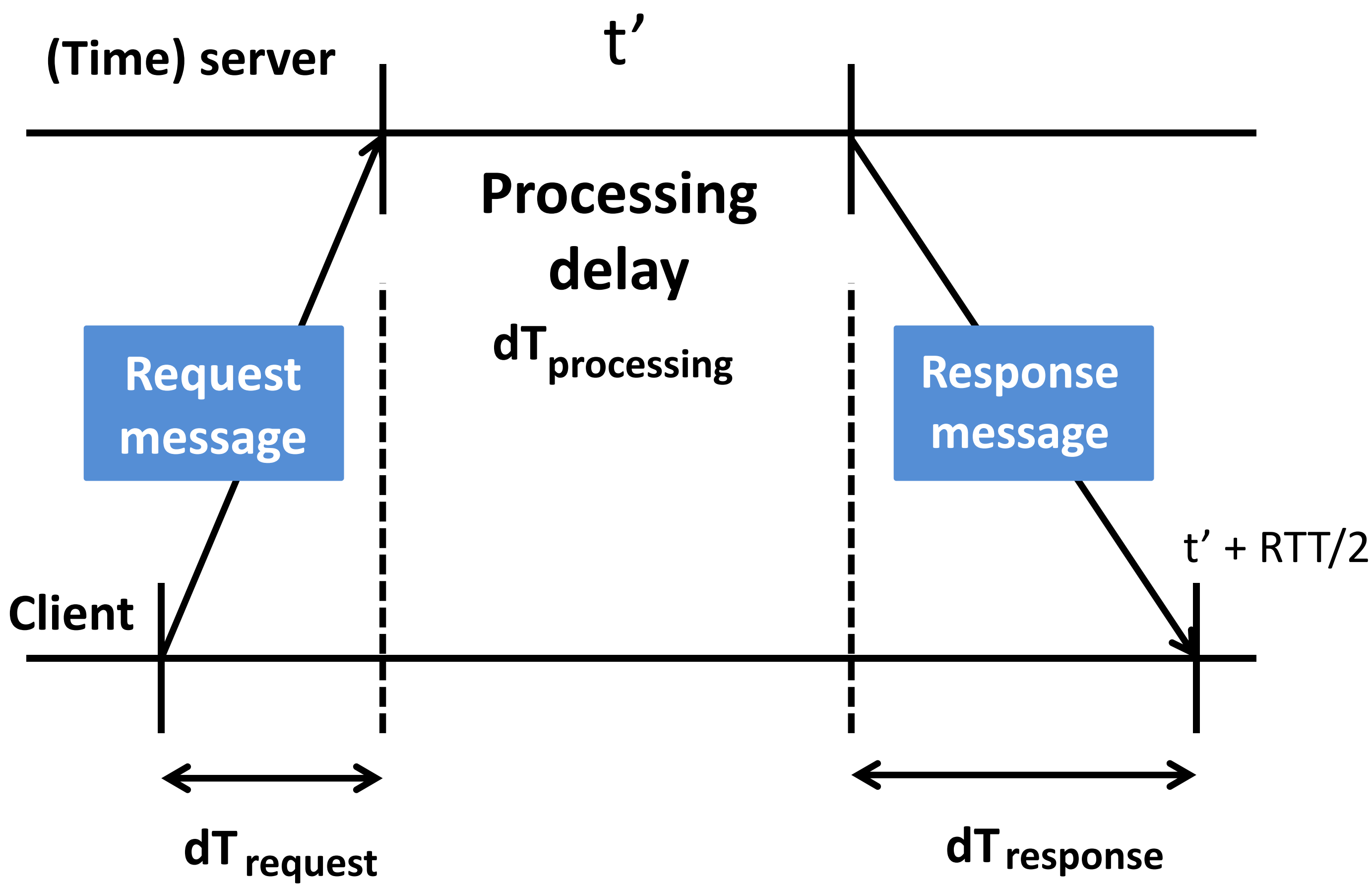
Cristian's algorithm (1989)

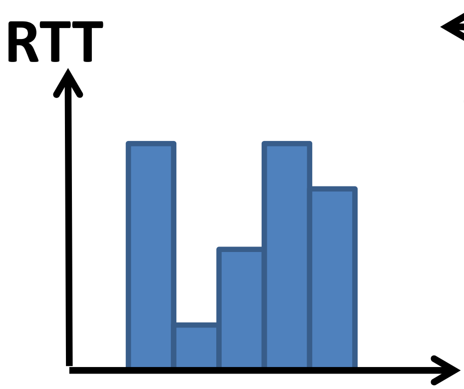
- Client measures **round trip time** for request to server
- Server responds with time value
- Client assumes transmission **delays split equally**
- Could factor in time to process request at server



$$t = t' + \mathrm{RTT}/2$$

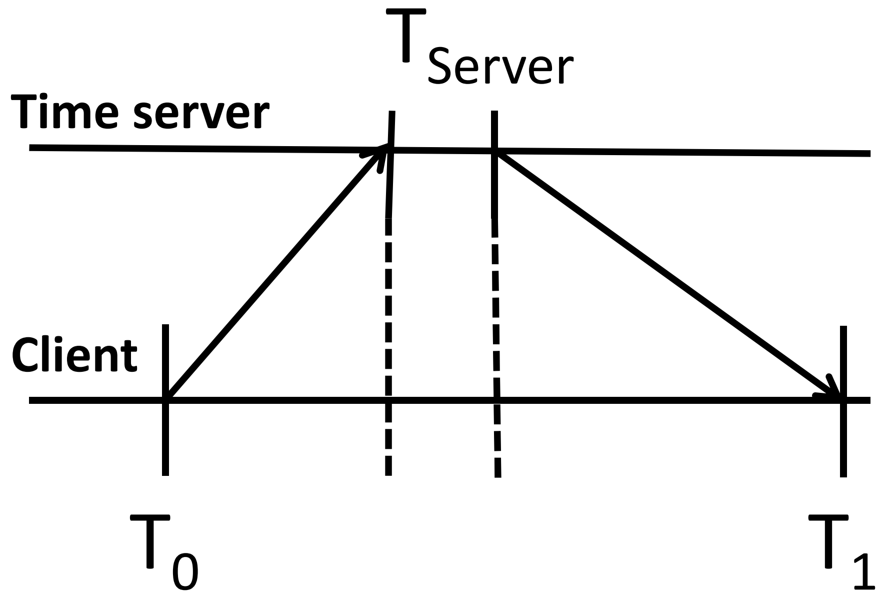
RTT: Accuracy estimation





- Make multiple requests
- Use what estimate?
- Accuracy is +/- RTT/2

New time



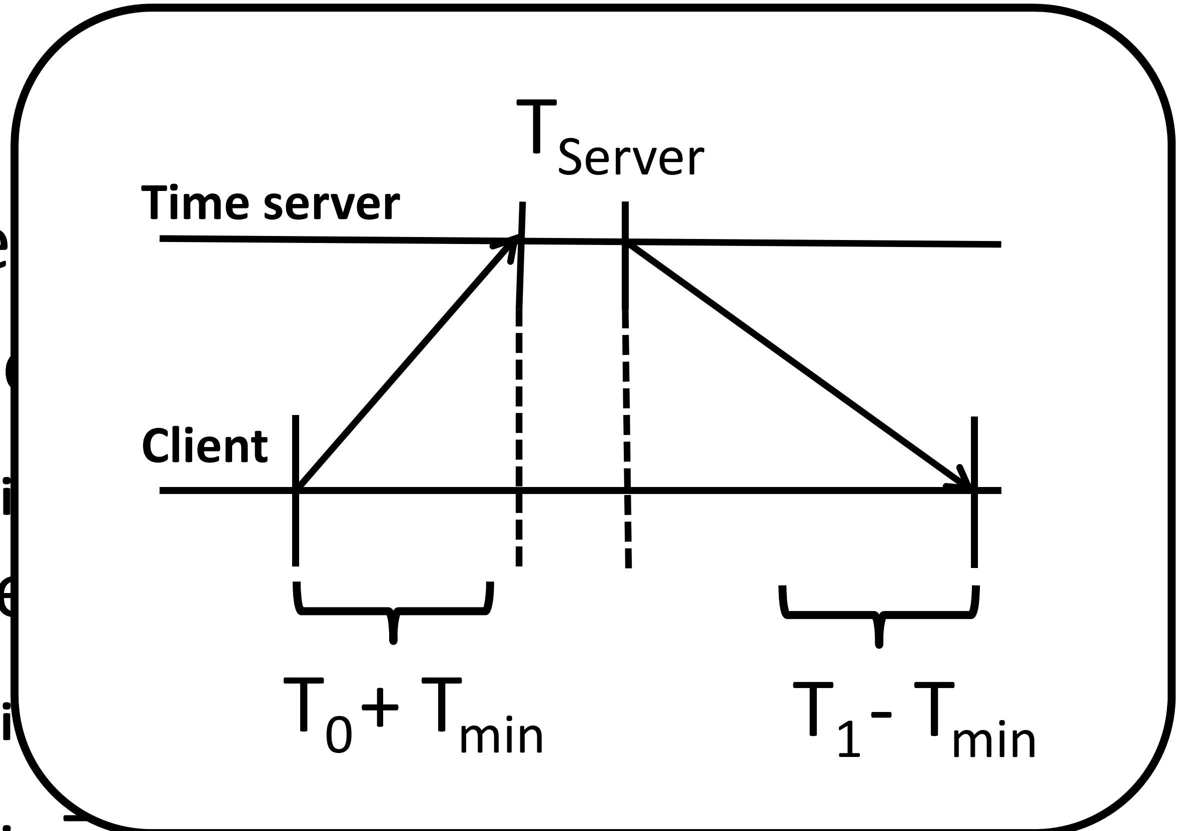
- Time request at T_0
- Time response at T_1
- Assume network delays are symmetric (RTT/2)
- Estimated overhead due to network delay is $(T_1 - T_0)/2$
- $\mathbf{T_{new} = T_{server} + (T_1 - T_0)/2}$

Result accuracy bound

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

Result accuracy bound

- T_{min} minimum
- T_0, T_1 as above
- **Earliest time s**
  stamp: $\mathbf{T_0 + T_{mi}}$
- **Latest time** the
  stamp: $\mathbf{T_1 - T_{mi}}$
- Range: $\mathbf{T_1 - T_{min}}$ –
- **Accuracy: +/- | (**$\mathbf{T_1 - T_0}$**)**$/2 -$ $\mathbf{T_{min}}$**) |**



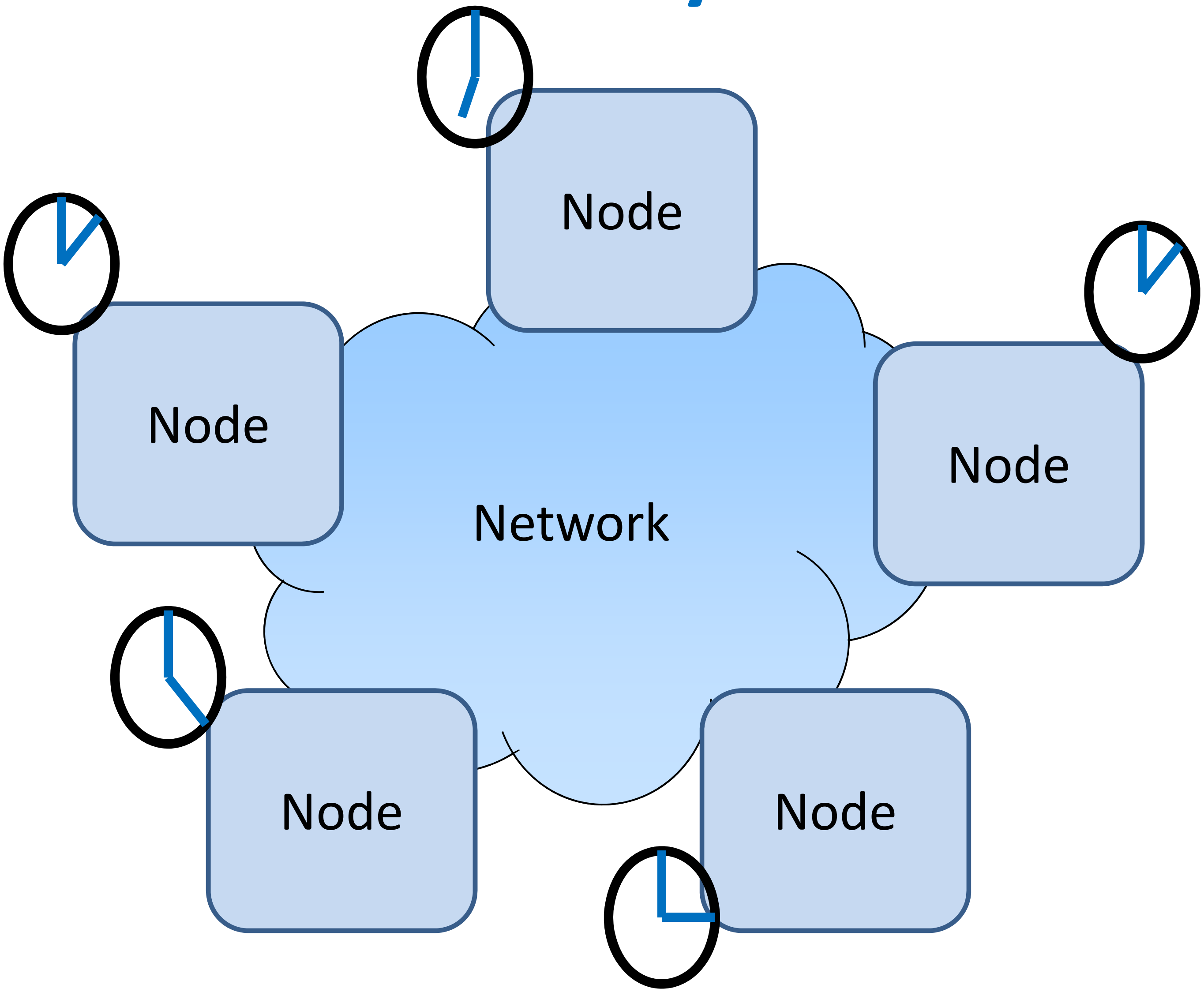
Errors are cumulative

- Say Node A synchronizes time with Node B with an accuracy of +/- 5 *ms*
- Node B synchronizes its time with Node C with an accuracy of +/- 7 *ms*
- Then, the net accuracy at Node A is +/-(5+7) *ms* = +/- 12 *ms*

Problems with Cristian's algorithm

- Centralized time server
  - Distribution possible via broadcast to many servers
  - Communication with single server is preferred
    - Simpler approach
    - Better estimates based on series of requests
- Time server is trusted & single point of failure
  - Malicious, failed server would wreak havoc

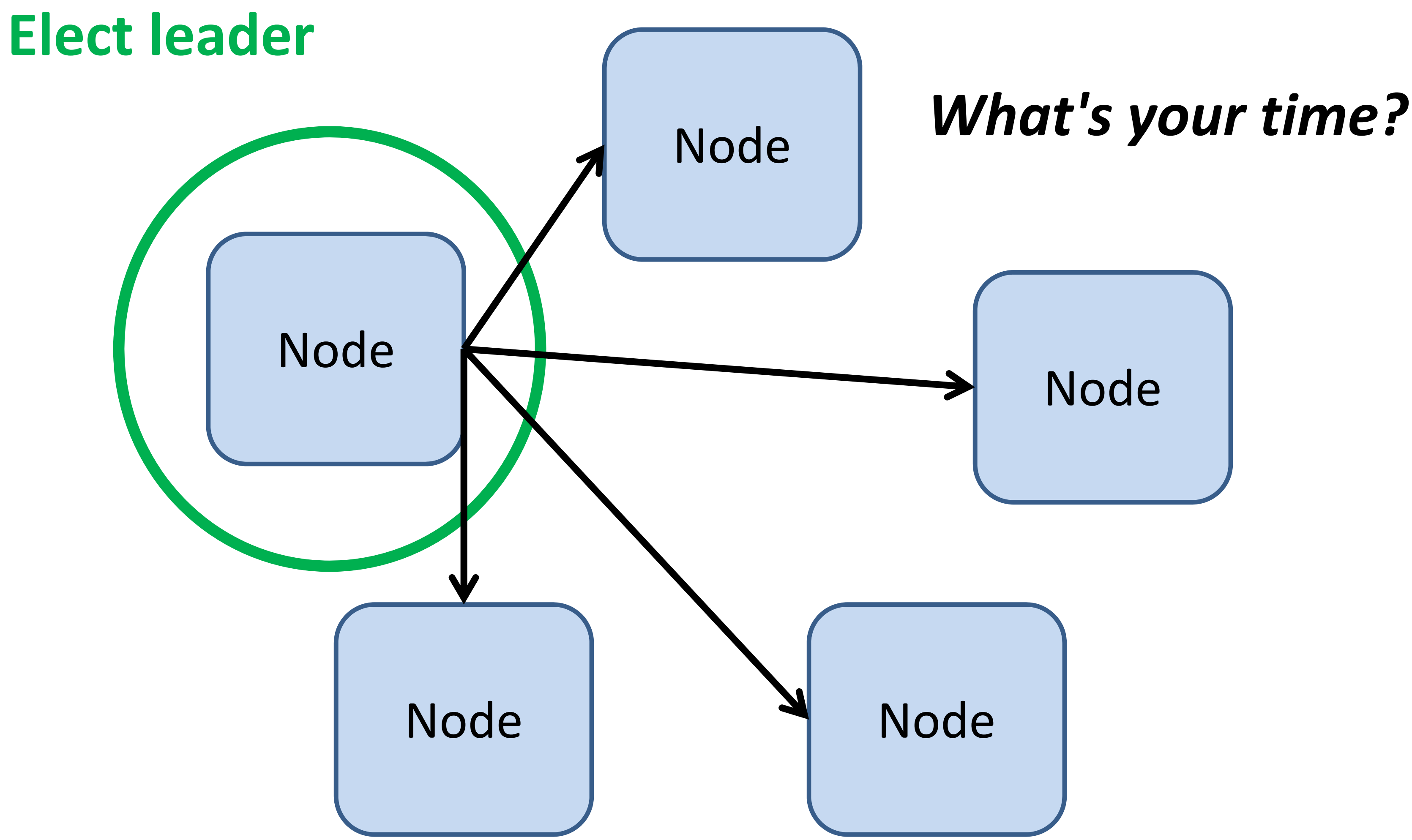
Problem: Internal synchronization



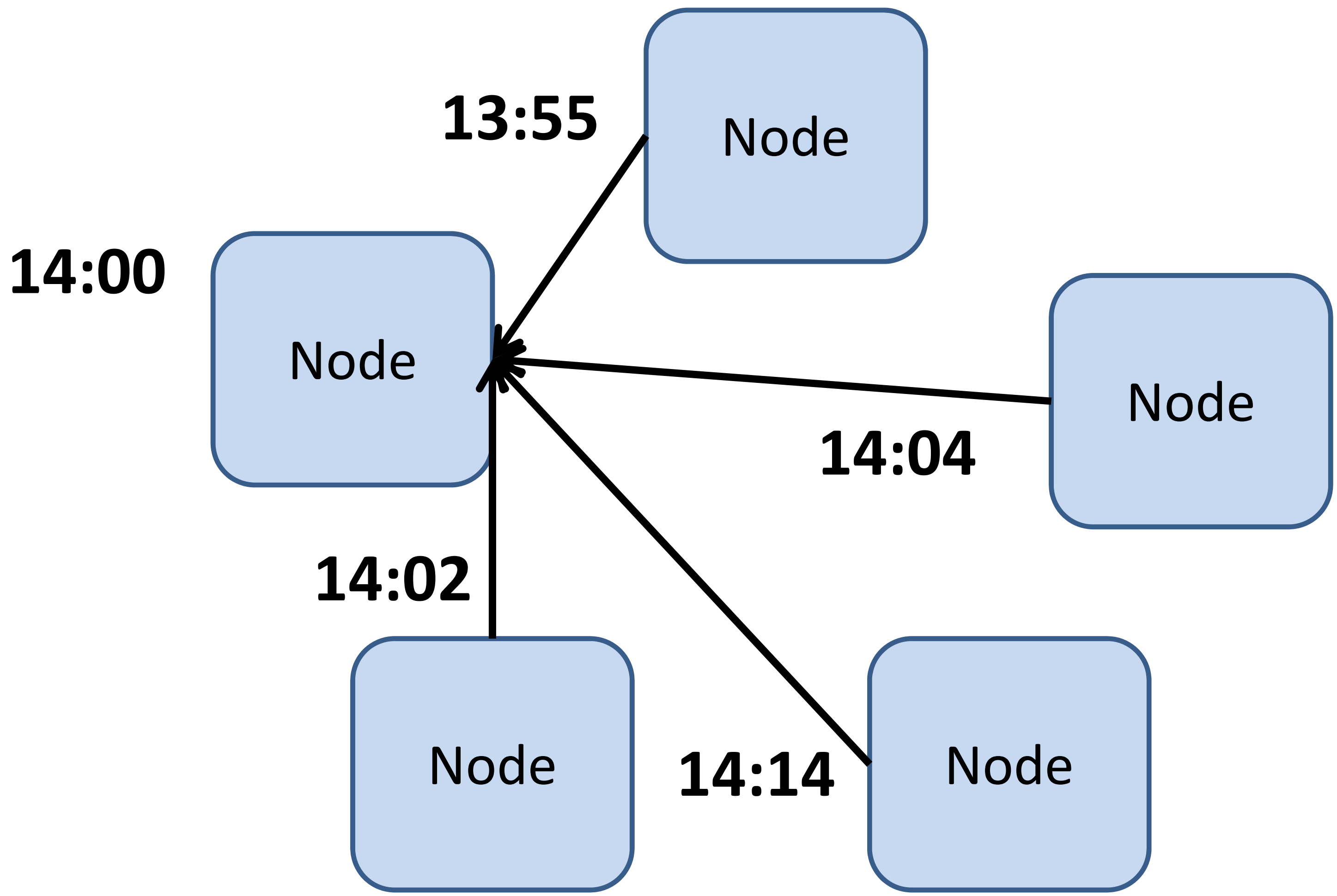
Berkeley algorithm overview

- **Clock synchronization** algorithm developed in 1989 as part of Berkeley Unix efforts
- **Internal synchronization**
- Assumes no machine has accurate time source
- Performs internal synchronization to **set clocks** of all machines **to within a bound**
- Intended for use in intranets / LANs
- Experimentally: Synchronization of clocks to within 20-25 *ms* on LAN

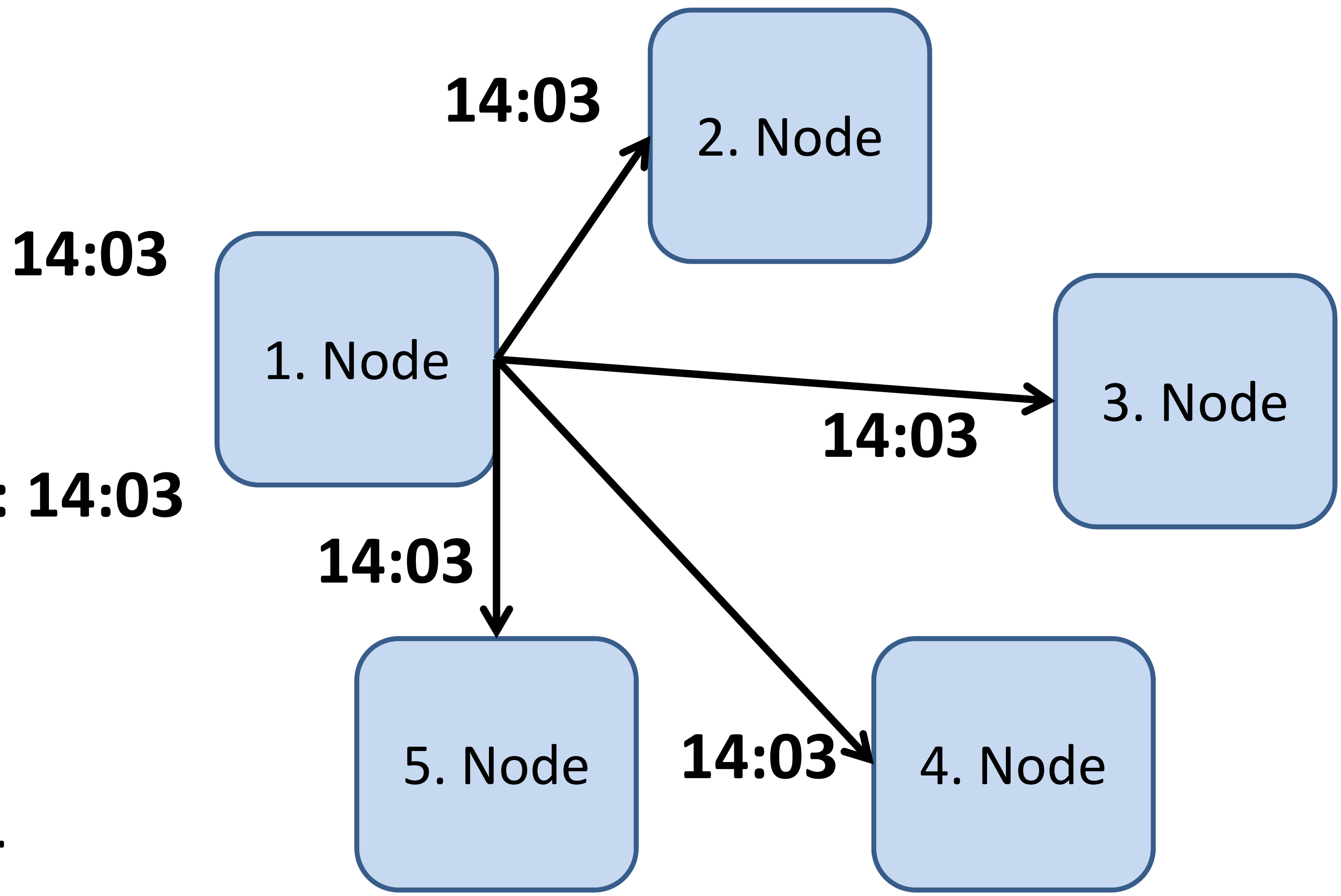
Berkeley algorithm: Request phase



Berkeley algorithm: Reply phase



Berkeley algorithm: Clock adjustment



Average: 14:03

1. + 3

2. + 8

3. - 1

4. - 11

5. + 1

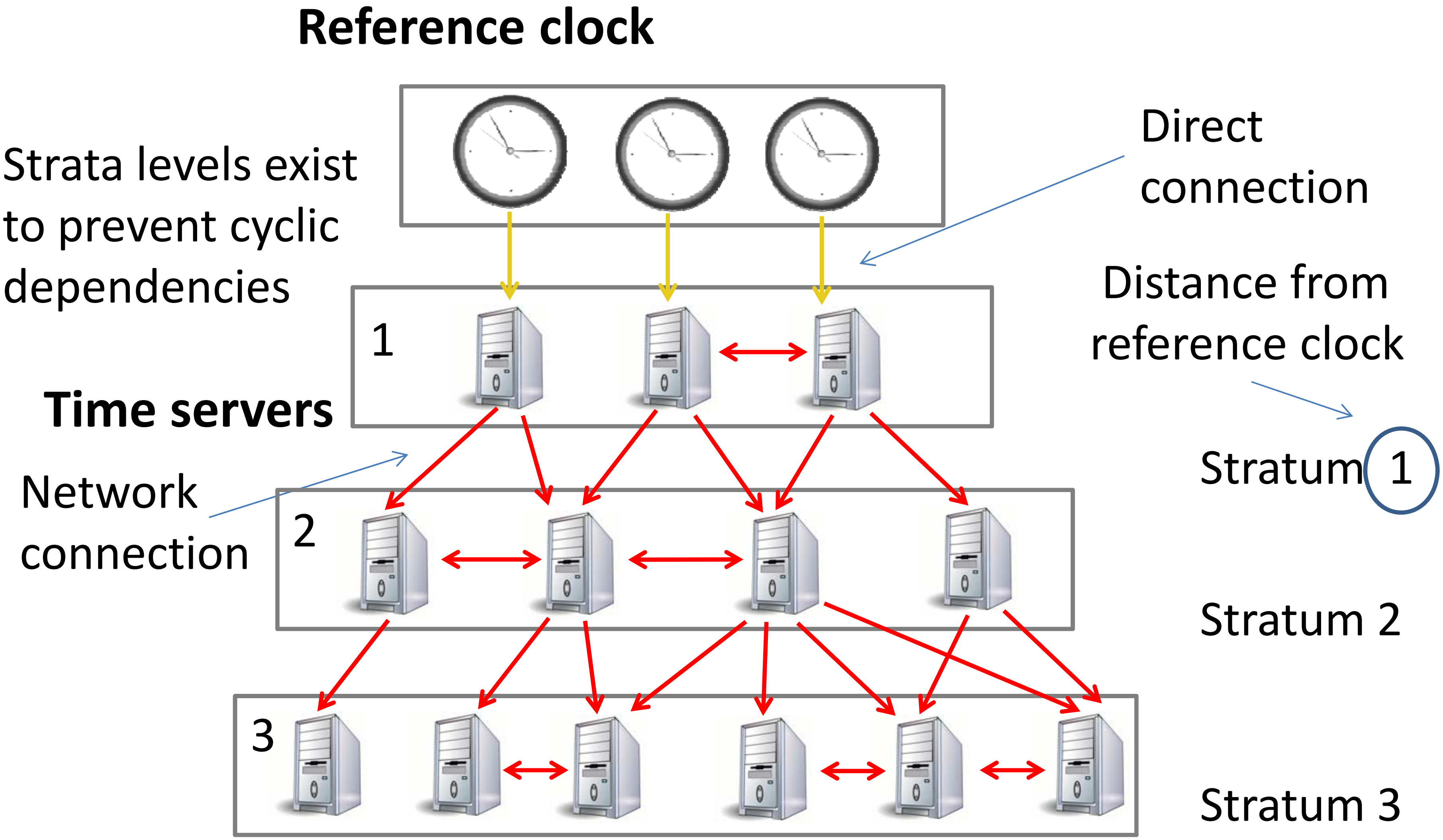
Berkeley algorithm: Summary

- Leader is chosen (via an **election process)**
- Leader polls nodes who reply with their time
- Leader **observes RTT** of messages and estimates time of each node and its own
- Leader **averages clock times**, **ignoring** any **values** it **receives far outside values of others** (including its own)
- Leader sends out **amount** (positive or negative) that each node must adjust its clock
- Accuracy originally reported was 20-25 ms (15 nodes)

Network time protocol (NTP) [D. Mills, 1994++]

- Service that provides UTC time over networks
- Hierarchical distributed system which provides scalability, fault-tolerance and security
- Time service and dissemination of time on Internet
- Maintains time to within **tens of milliseconds over Internet**
- Achieves **1 millisecond accuracy in LANs** under ideal conditions
- Supported by daemon (ntpd) in user space (kernel support) via UDP on port 123

NTP: Clock Strata



Up to 16 levels used, more possible (stratum 16 unsynchronized)

CLOCK ADJUSTMENT

Clock adjustment

- Adjusting the clock is not straight forward
  - Set *clock = new clock value* *(strict no-no)*
  - Time must **increase monotonically**
- Can't **go back to the past**
  - Timestamps are important, can't repeat them
- Time should not **show sudden jumps**
  - Cannot jump on your way to the future
  - Lose the present time and miss a deadline

Hardware and software clocks

- At real time, t, OS reads time on computer's **hardware clock** $H_i(t)$
- Calculates time on its *software clock*:

  $C_i(t) = a * H_i(t) + b$ (for process i)

- Monotonicity requirement:

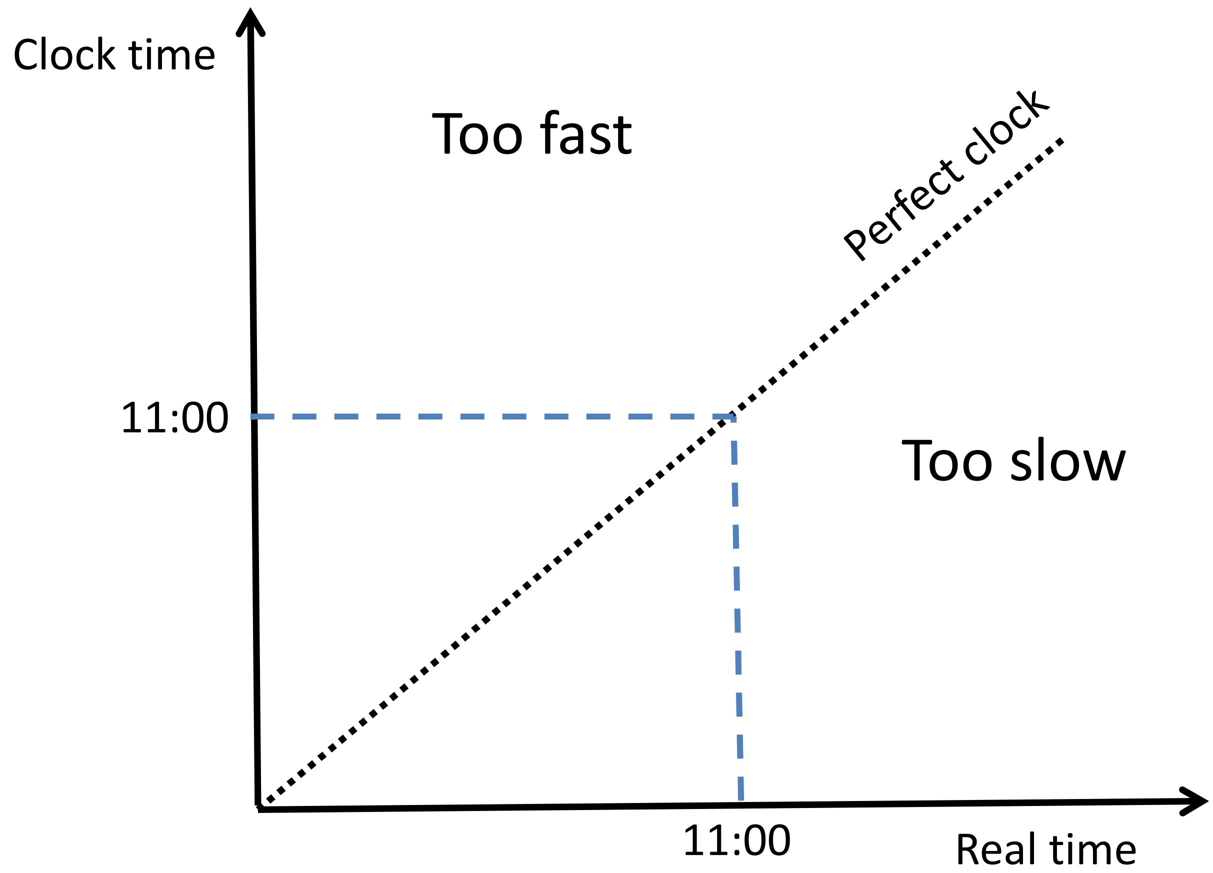
  $t' > t$: $C(t') > C(t)$

- Can achieve monotonicity by adjusting a and b in $C_i(t) = a * H_i(t) + b$

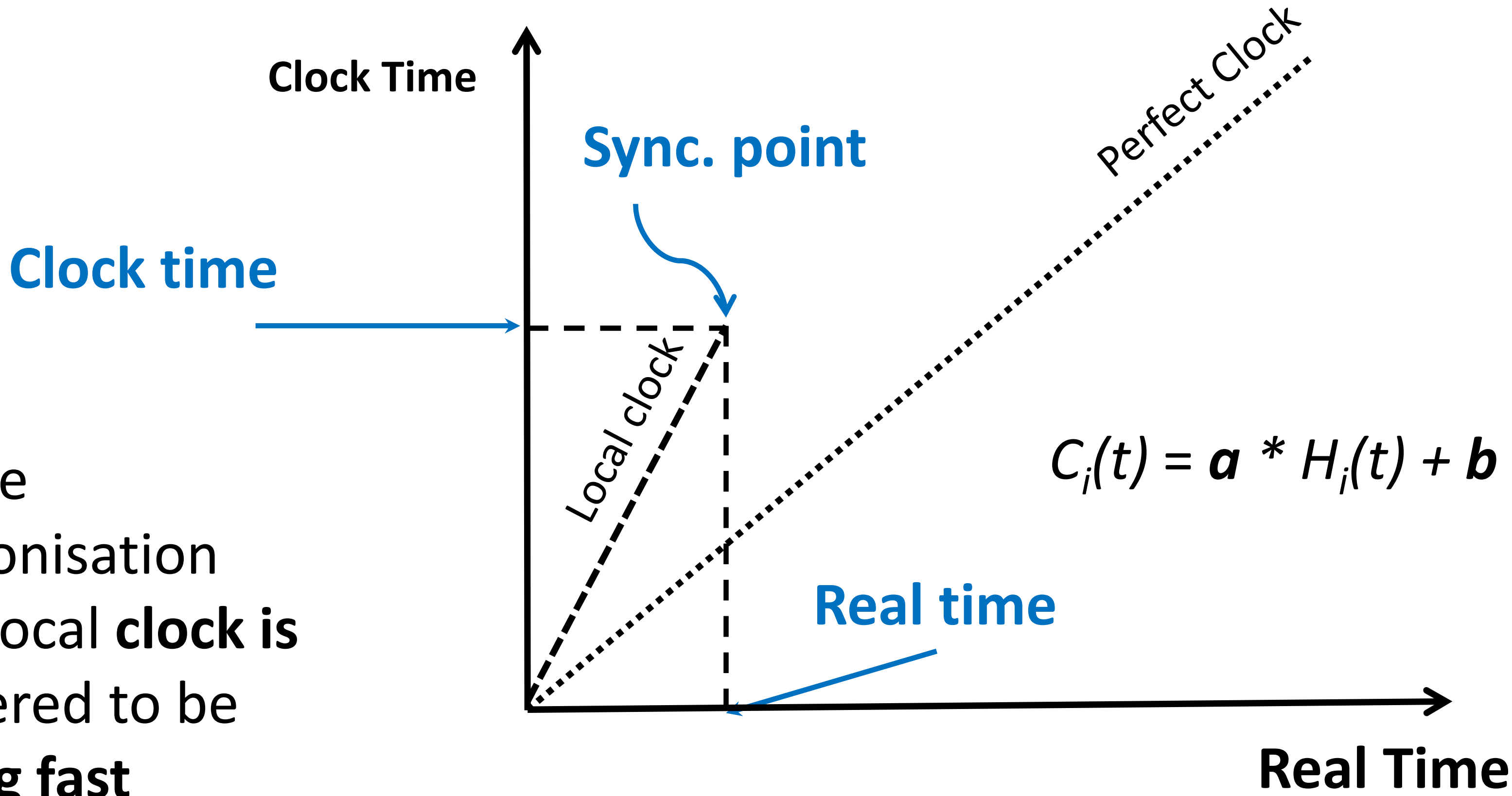
Clock adjustment

- Two parameters are a constant **offset** and a linear **slope**:
  - $C_i(t) = \boldsymbol{a} * H_i(t) + \boldsymbol{b}$
- The "catch up" value for scaling local time down or up in a linear fashion

Perfect time



Clock too fast

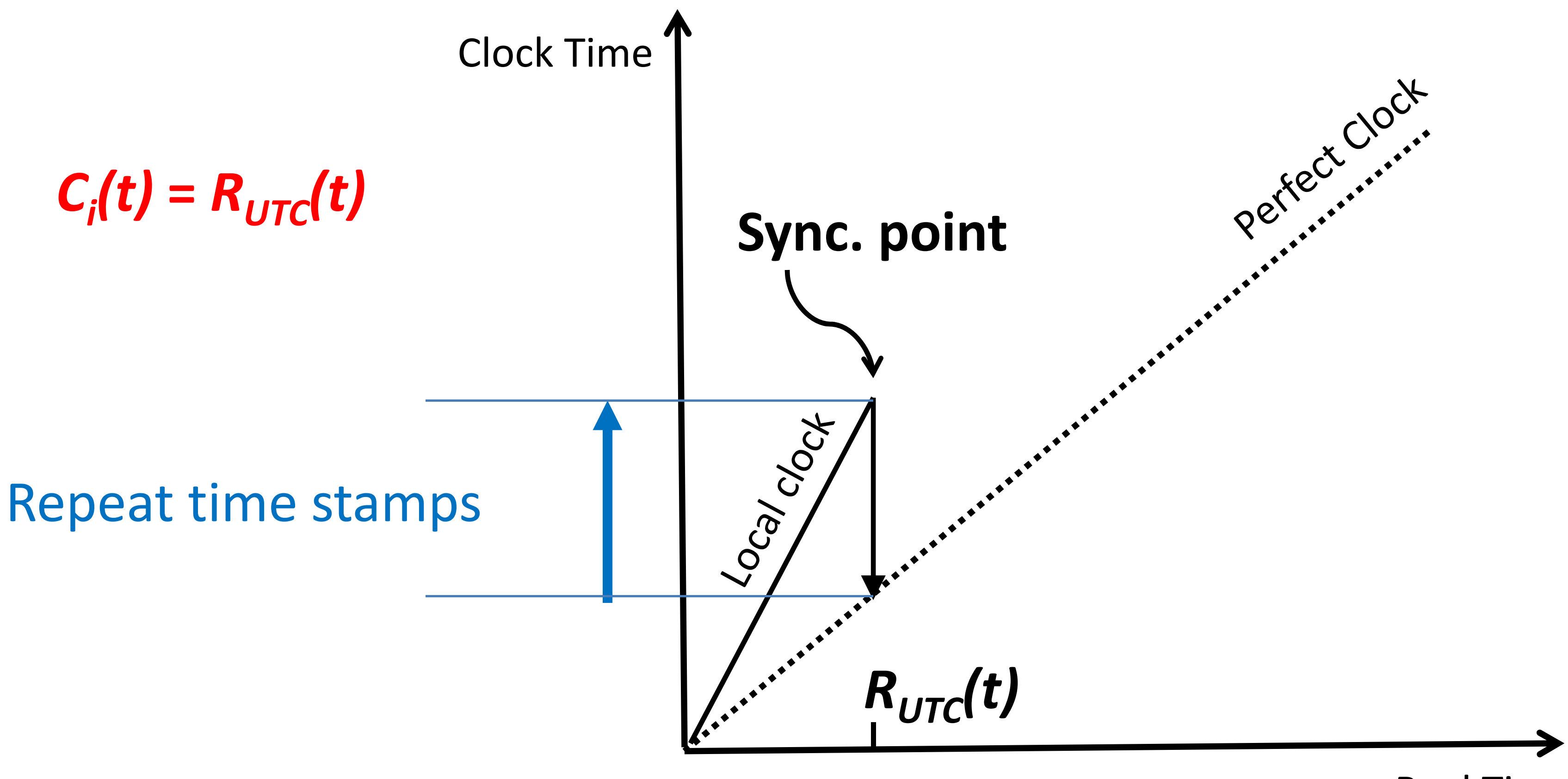


- At some synchronisation point, local **clock is** discovered to be **running fast**

- True time according to a UTC source, for example

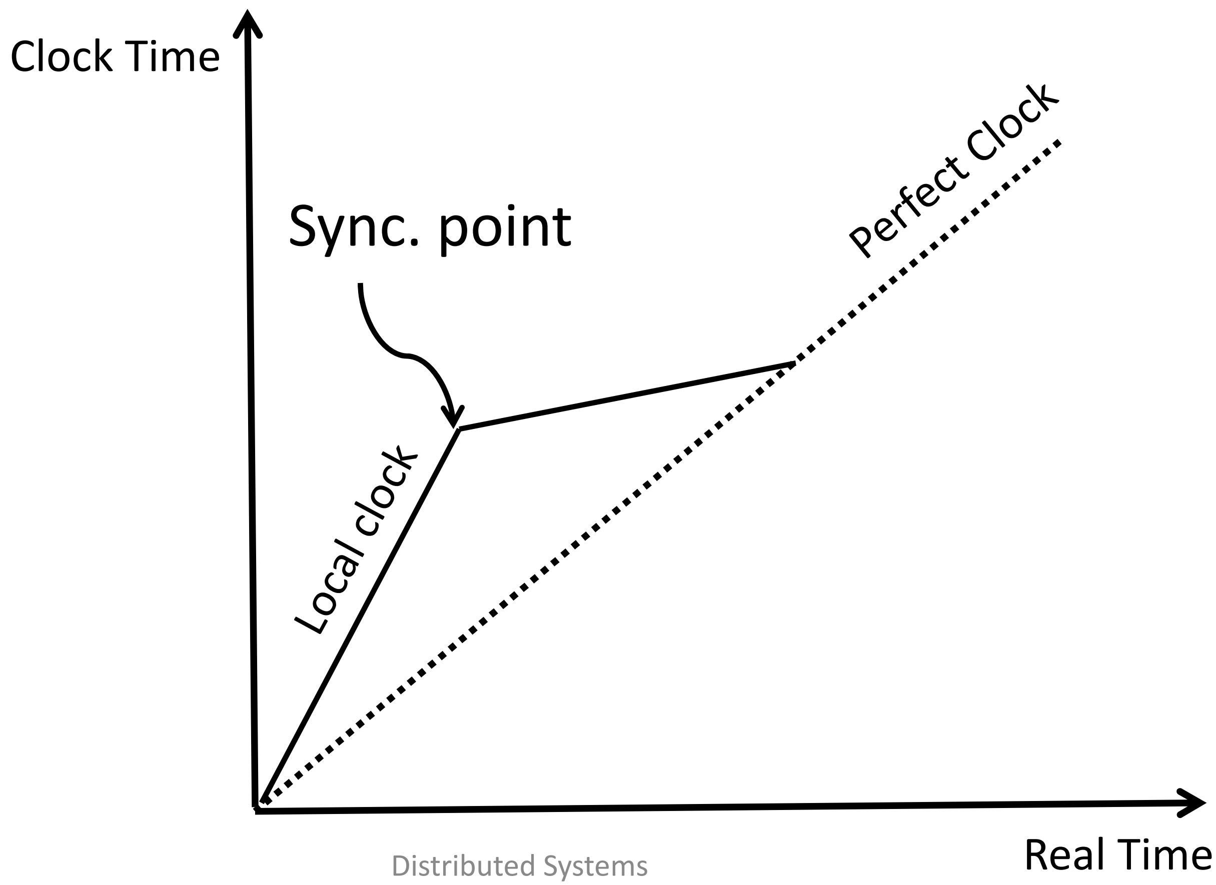
“Catch up” Example: Rese ock

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



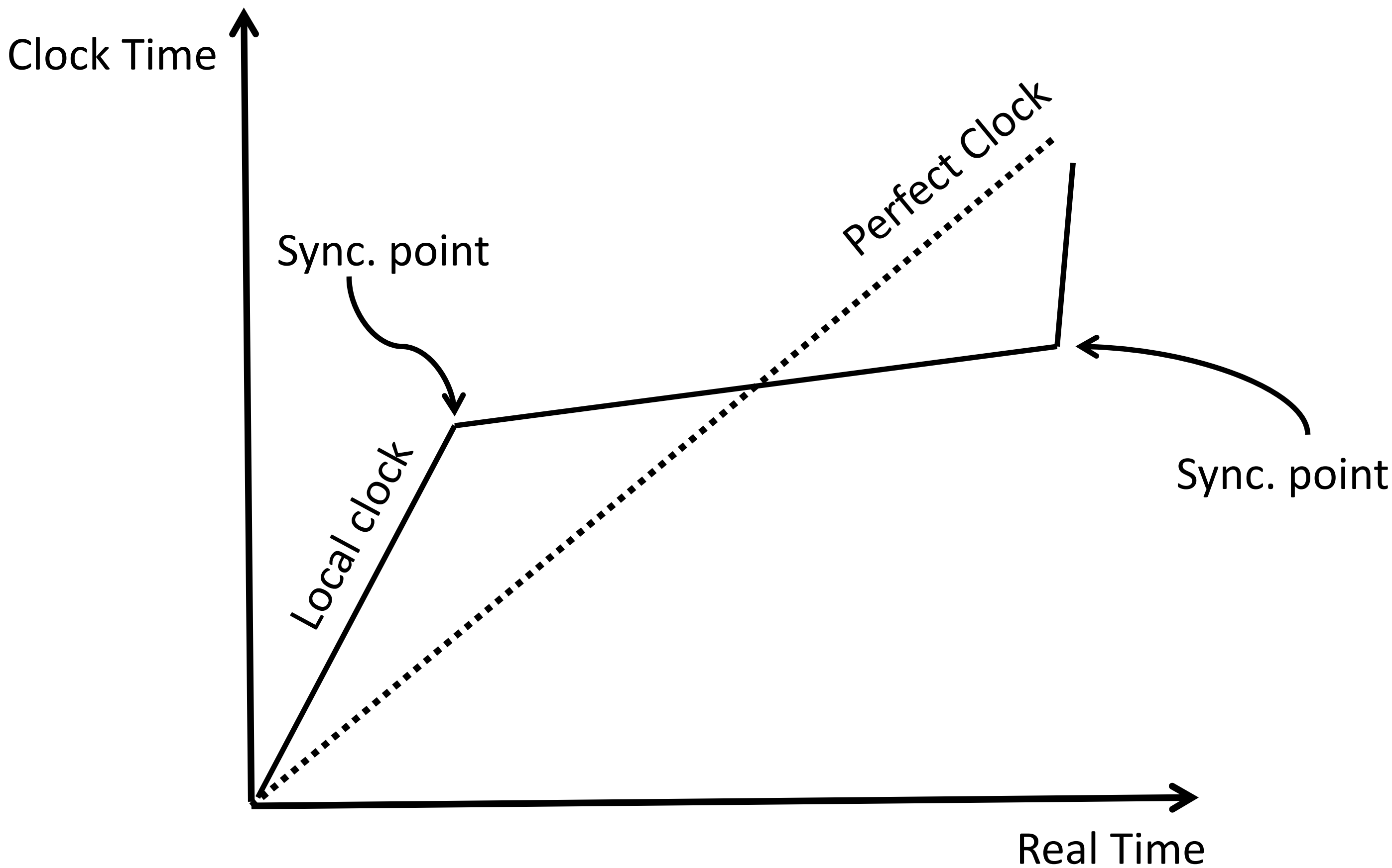
“Catch up” Example: Slow down clock

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



“Catch up” Example: Implications

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



LOGICAL CLOCKS

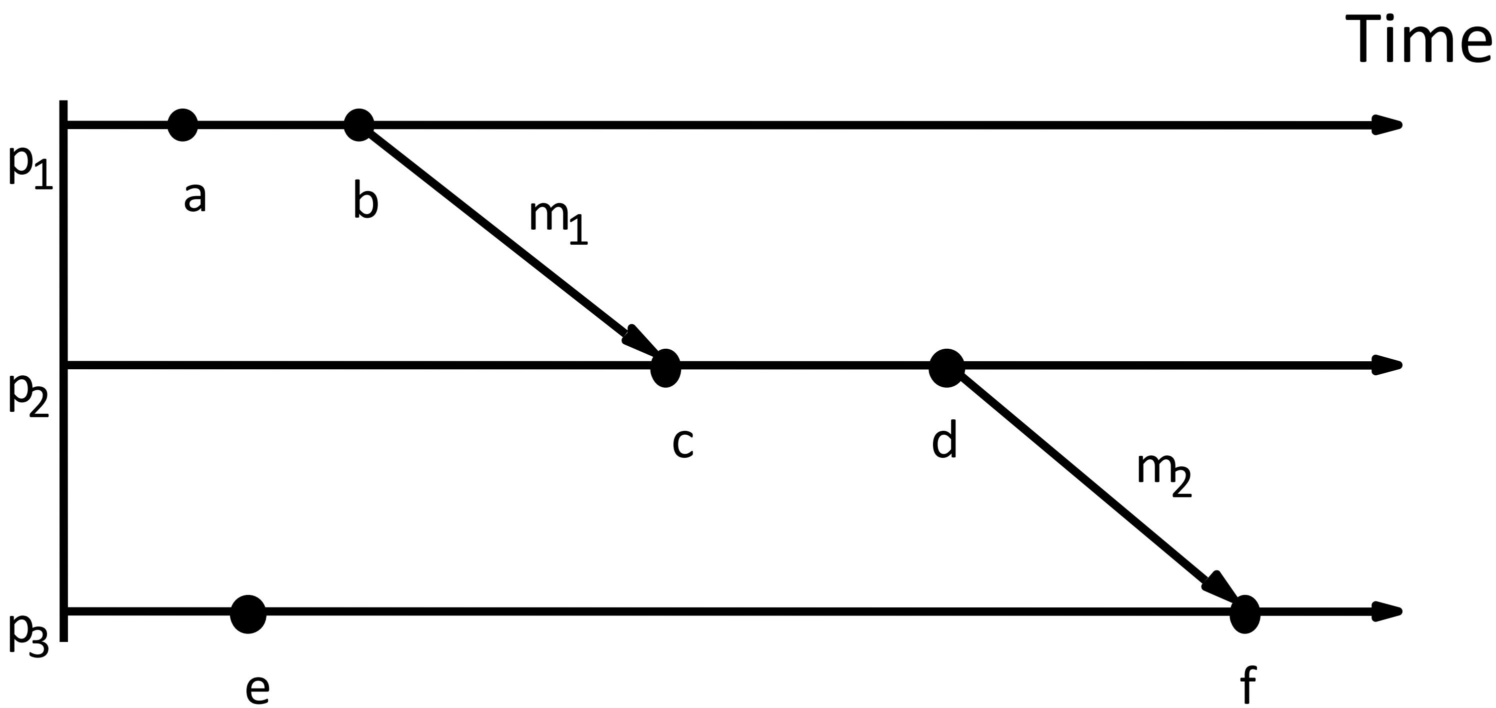
Events and event ordering

- Often sufficient to know the **order of events**
  - An **event** may be an instruction execution, method call, order entry, etc.
  - Events include **message send** & **receive**
- Within a **single process** order determined by **instruction execution** sequence
- Between **two processes** on same computer order determined using **local physical clock**
- Between **two different computers** order cannot be determined using **local physical clocks**, since those clocks **cannot be synchronized perfectly**

Logical clocks

- Key idea is to **abandon the idea of physical time**
- Only know **order of events**, not *when* they happened (or *how* much time between events)
- Lamport (1978) introduced **logical** (virtual) **time** and methods to synchronize logical clocks

Events in a distributed system



The happened-before relation

Denoted by "→"

- Describes a **(causal) ordering of events**
- If ***a* and *b* are events** in the same process and ***a* occurred before *b*** then ***a* → *b***
- If ***a*** is the event of **sending a message *m*** in one process and ***b*** is the event of **receiving *m*** in another process then ***a* → *b***
- Relation "→" is **transitive:** If *a* → *b* and *b* → c then *a* → *c*
- If neither *a* → *b* nor *b* → *a* then ***a* and *b* are concurrent,** denoted by a || b
- For any two events *a* and *b,* either *a* → *b, b* → *a,* or *a* || *b*

Causality of "→"-relation

(a.k.a. causality relation)

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

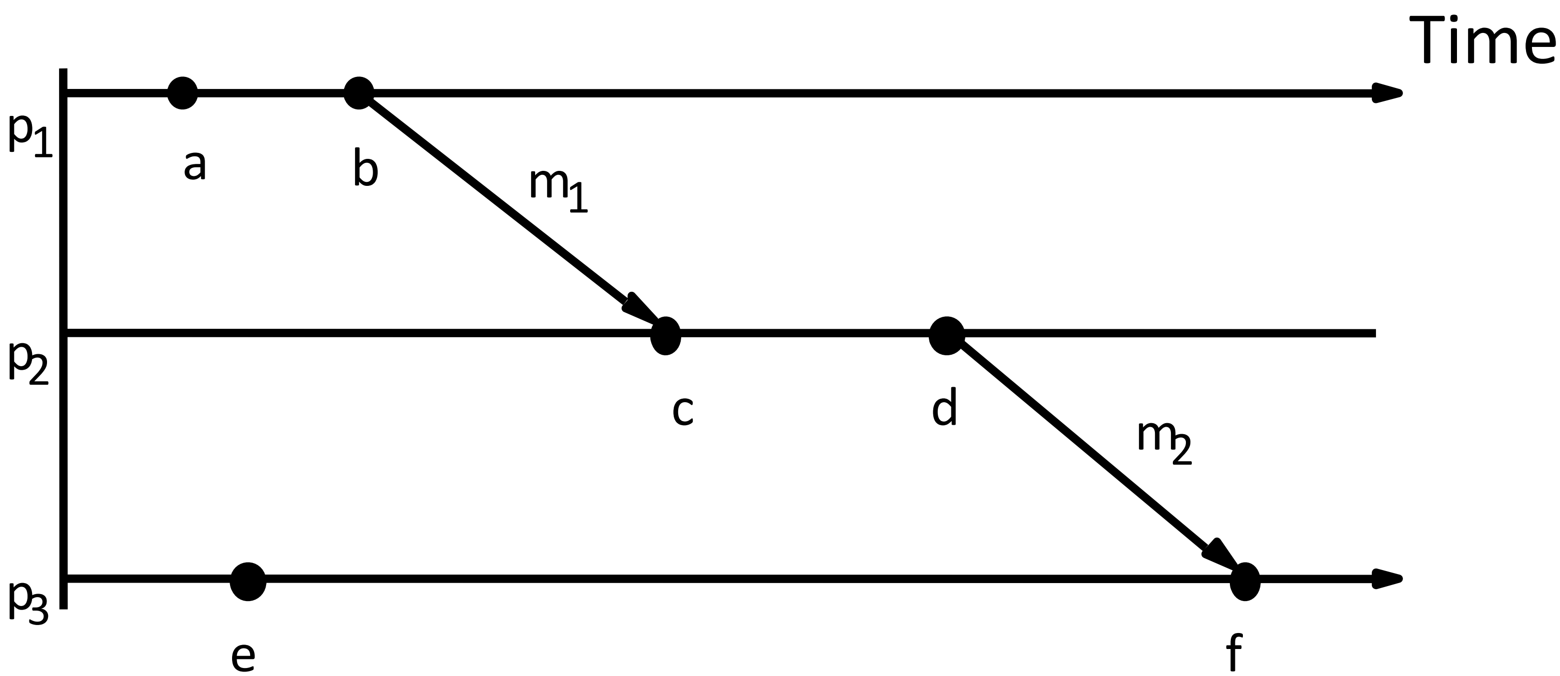
Potential causality of "→"-relation

- "→" captures potential flow of information between events

- Information may have flown in ways other than via message passing (not modeled by "→")

- In $a \rightarrow b$, *a* might or might not have caused *b* (relation assumes it has, but we don't know for sure)

Happened-before relation



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

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

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

Lamport clock (logical clock)

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

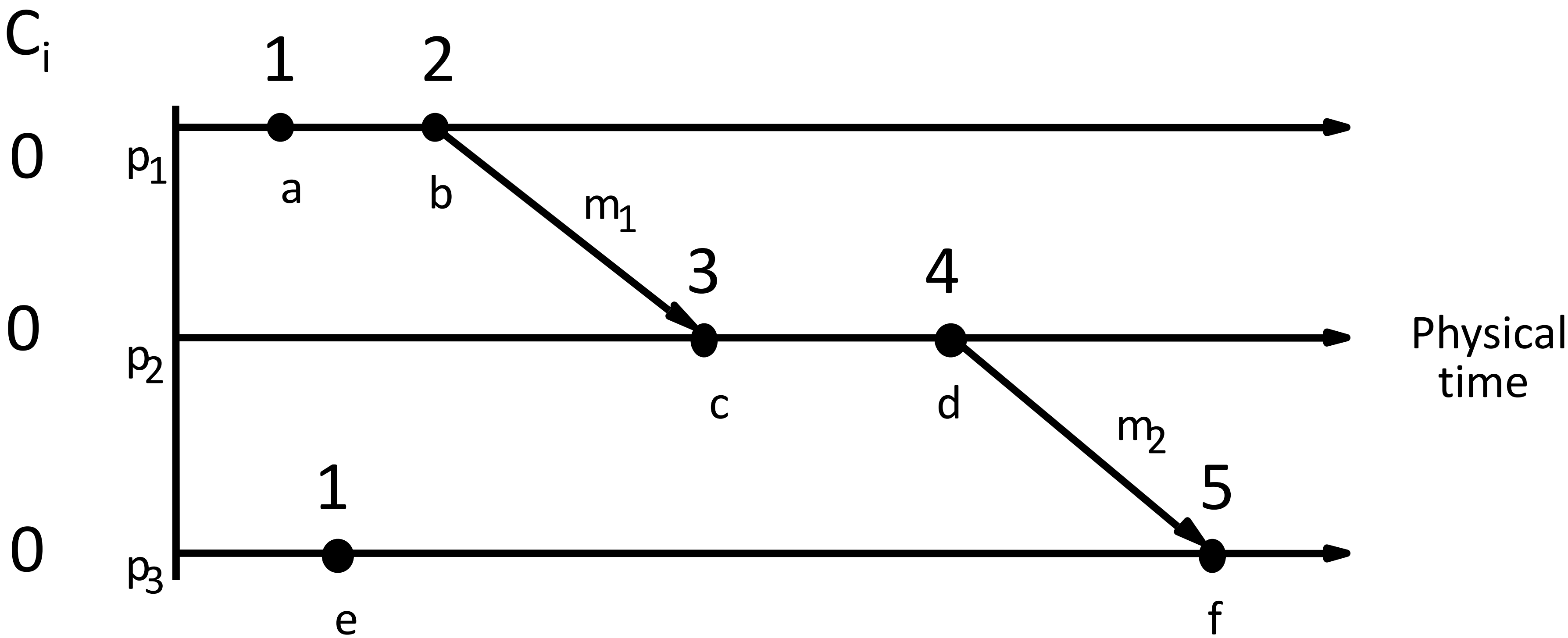
Correctness condition

- Clock condition
  - **If a → b then C(a) < C(b)**
  - **But not: If C(a) < C(b) then a → b**
- Correctness conditions
  - For any two events *a* and *b* in the same process P_i, **if *a* → *b* then $C_i(a) < C_i(b)$**
  - If *a* is the event of **sending a message** in process P_i and *b* is the event of **receiving that same message** in a different process P_j then $\mathbf{C_i(a) < C_j(b)}$

Implementation of logical clocks

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

Logical clock example



a → b   a → f

b → c   e → f

c → d   b || e - **C(b) > C(e) !**

Induced total order

- $C_1(a) = 1$ and $C_3(e) = 1$ can't be ordered according to happened-before relation!
- Happened-before is a **unique partial order** of events
- Induce a non-unique total order as follows
  - Use logical time stamps to order events
  - Break ties by using an arbitrary total ordering of processes, e.g., $P_1 < P_2$ (process identifiers)

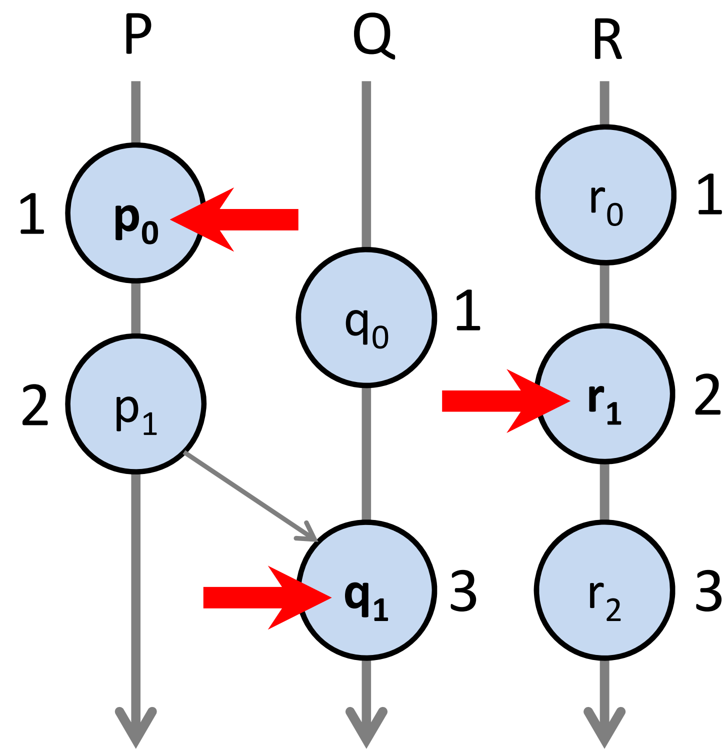
Total order for events

- Let $\boldsymbol{a}$ be an event in $\boldsymbol{P_i}$ and $\boldsymbol{b}$ an event in $\boldsymbol{P_j}$ *then* $\boldsymbol{a} \Rightarrow \mathbf{b}$ if and only if either
  - (i) $\mathbf{C_i(a) < C_j(b)}$ or
  - (ii) $\mathbf{C_i(a) = C_j(b)}$ and $\mathbf{P_i < P_j}$

- Results in total order of all events in system

Limitation of Logical Clocks

- If $C(a) < C(b)$ then
  a may or may not happen-before *b*
- Example illustrating this limitation
  - $C(\mathbf{p_0}) < C(\mathbf{q_1})$ and $p_0 \rightarrow q_1$ is true
  - $C(\mathbf{p_0}) < C(\mathbf{r_1})$ but $p_0 \rightarrow r_1$ **is false**



- **One cannot determine whether two events are causally related from timestamps alone**

VECTOR CLOCKS

Vector clocks I

- System with ***n*** processes
- Each process P_i has a **clock C_i,** which is **an integer vector of length n**:

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

- **C_i(a)** is the **timestamp** (clock value) of **event *a*** at process P_i (a vector)
- **C_i[i]**, entry i of C_i, is **P_i's logical time**
- C_i[i] represents the number of events that process P_i has timestamped

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

Vector clocks II

- $C_i[k]$, entry k of C_i (where $k \neq i$), is P_i's "guess" of the logical time at P_k

- $C_i[k]$ is the number of events that have occurred at P_k that P_i has potentially been affected by

$$\mathbf{P_i} \; : (C_i[1], \ldots, \mathbf{C_i[k]}, \ldots, C_i[n])$$

Implementation of vector clocks

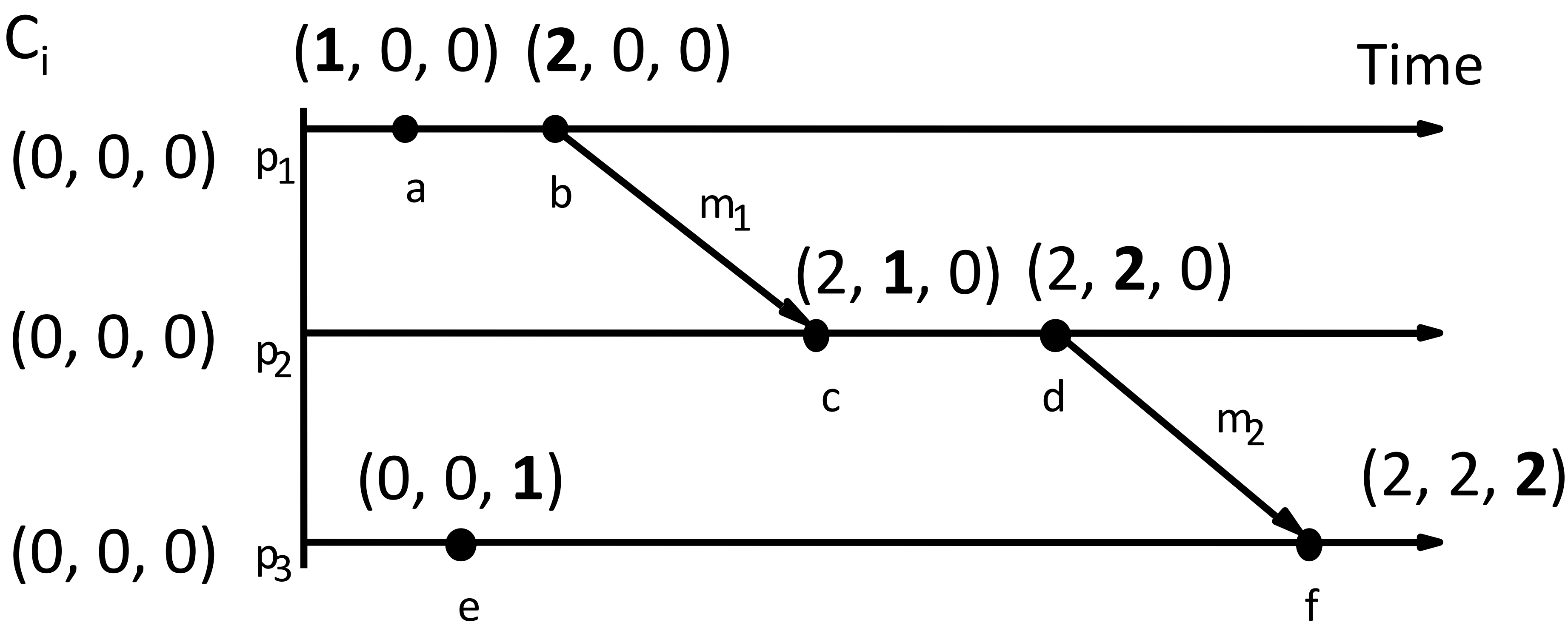
- Clock C_i is incremented before an event occurs in process P_i

  $C_i[i] = C_i[i] + d$ ($d > 0$, d usually 1)

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

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

Vector clock example



Relations for comparing vector clocks

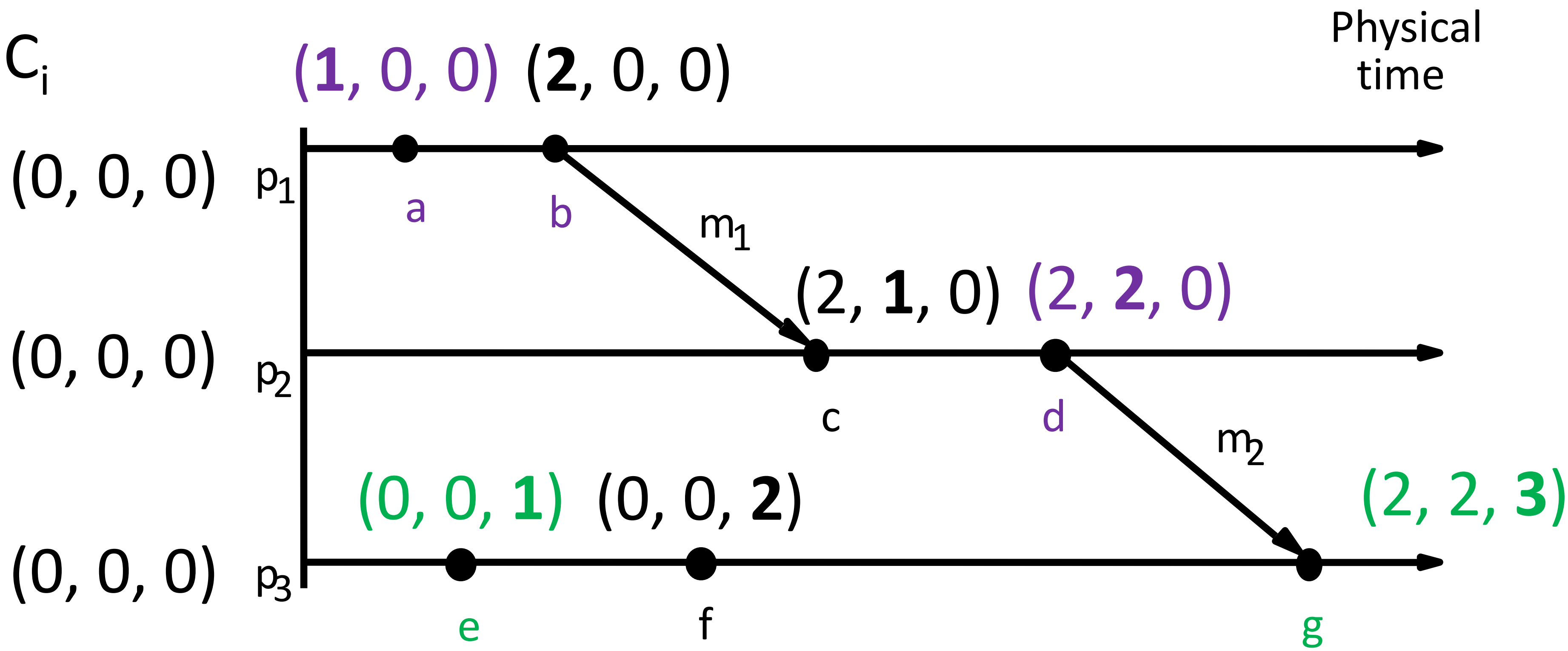
- C_a, C_b two vector timestamps

- $C_a = C_b$ iff for all *i*: $C_a[i] = C_b[i]$
- $C_a \leq C_b$ iff for all *i*: $C_a[i] \leq C_b[i]$
- $C_a < C_b$ iff $C_a \leq C_b$, $\exists\, i : C_a[i] < C_b[i]$
- $C_a \,||\, C_b$ iff $\neg(C_a[i] < C_b[i])$ and $\neg(C_b[i] < C_a[i])$

Example

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

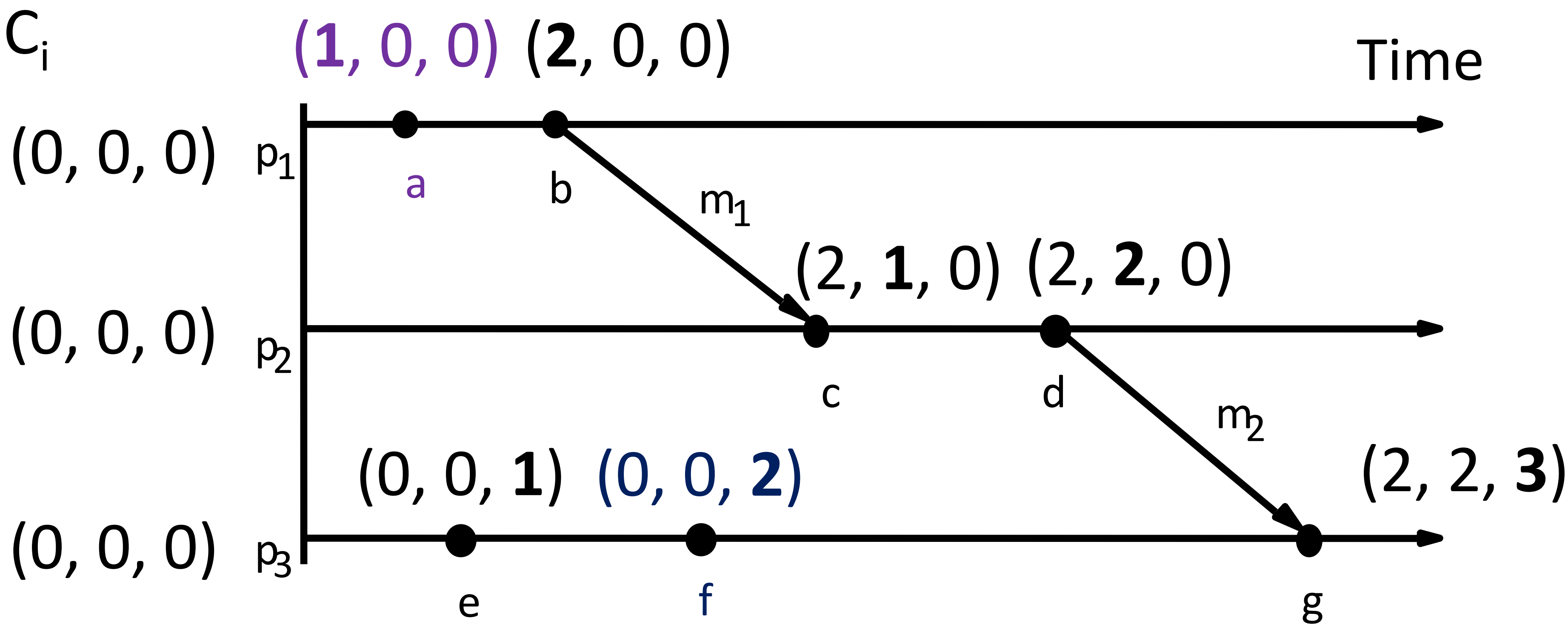
Vector clock example



(1, 0, 0) < (2, 2, 0) therefore a → d

(0, 0, 1) < (2, 2, 3) therefore e → g

Vector clock example



(1, 0, 0) || (0, 0, 2) therefore a || f

With Lamport: 1 < 2, but not a → f?

Properties of vector clocks

- Let *a*, *b* be two events with vector time stamps C_a, C_b then:
  - $a \rightarrow b$ *iff* $C_a < C_b$
  - $a \,||\, b$ *iff* $C_a \,||\, C_b$

Application of vector clocks in Dynamo

Versioning data objects

Assume three **servers** *X*, *Y*, *Z*, a data object *D* with object versions represented by a **vector clock** (*X*, *Y*, *Z*)

Client_1 → D1 (1, 0, 0) — Write handled by Server *X*

Client_1 → D2 (2, 0, 0) — Write handled by Server *X* *(produces a new version D)*

Dynamo can **syntactically reconcile** data (overwrite D1 with D2), D2 is a strictly newer version of D1.

Internal to Dynamo, **updates** to data **are replicated** asynchronously to other servers.

There may exist replicas of D1 that have not yet seen D2.

Application of vector clocks in Dynamo

Versioning data objects

Client_1

Write handled by Server *X*

D1 (1, 0, 0)

Client_1

Write handled by Server *X*
(produces a new version D)

D2 (2, 0, 0)

Client_1

Write handled by Server Y

D3 (2, 1, 0)

Application of vector clocks in Dynamo

Versioning data objects

Client$_1$

Write handled by Server *X*

D1 (1, 0, 0)

Client$_1$

Write handled by Server *X*
(produces a new version D)

D2 (2, 0, 0)

Client$_1$

Client$_2$ (Reads D2)

Write handled by Server Y

D3 (2, 1, 0)

D4 (2, 0, 1)

Write handled by Server Z

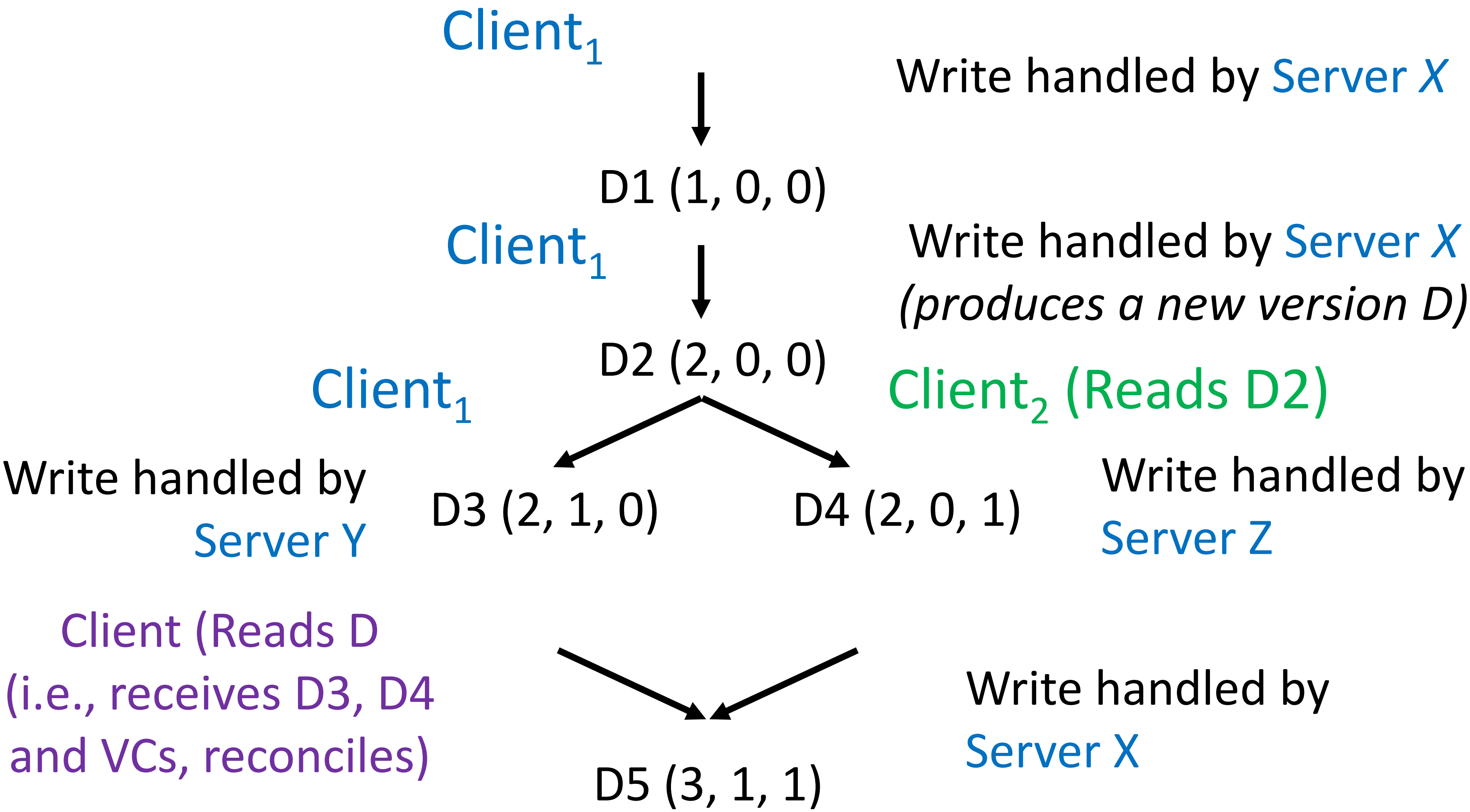
Application of vector clocks in Dynamo

Versioning data objects

- A node seeing D1 (1, 0, 0) and D2 (2, 0, 0)
  - (1, 0, 0) ≤ (2, 0, 0) (overwrite D1 with D2)
- A node seeing D1 (1, 0, 0), D2 (2, 0, 0) and D4 (2, 0, 1) (or D3 (2, 1, 0))
  - E.g., (2, 0, 0) ≤ (2, 0, 1) (overwrite D2 with D4)
- A node aware of D3 (2, 1, 0), receiving D4 (2, 0, 1)
  - (2, 1, 0) || (2, 0, 1) (no causal relation!)
  - Exist changes in D3, D4 that are not reflected in each others' version of the data
  - Both versions must be kept and presented to client for **semantic reconciliation**

Application of vector clocks in Dynamo

Versioning data objects



{}

Add beer

{beer}

Add chips

{beer, chips}

Delete beer

Add diapers

{<u>chips</u>, <u>diapers</u>}

Delete chips

Add sausages

{<u>beer</u>, *sausages*}

Reconciliation is to "merge" shopping carts.

{***beer***, **<u>chips</u>**, **<u>diapers</u>**, ***sausages***}

Didn't loose "Add," loose "Delete"!

Logical clock summary

- Clocks that are not based on real-time
- Logical time progresses using events
  - Local execution of some code
  - Send/receive messages
- Happened-before relationship
  - Tracks causality between events across processes
- Lamport clock
  - Single counter updated at every event
  - Introduces false positive causality
- Vector clock
  - Size of timestamp is relative to number of processes
  - Can determine causality more accurately than Lamport clock

JSON – FYI (OFFLINE READING)

JavaScript Object Notation (JSON)

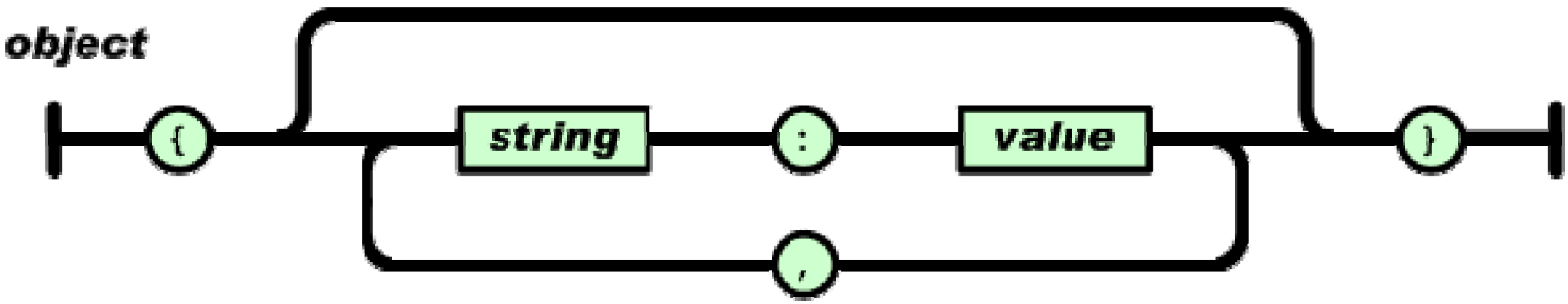
- Lightweight data-interchange format
- Easy for humans to read and write
- Easy for machines to parse and generate
- A language-independent text format
- Nowadays, widely supported in libraries for many programming languages

JSON built on two structures

- Collection of name/value pairs
  - E.g., realized as *object*, record, struct, dictionary, hash table, keyed list, or associative array in other programming languages
- Ordered list of values
  - E.g., realized as an *array*, vector, list, or sequence in other programming languages

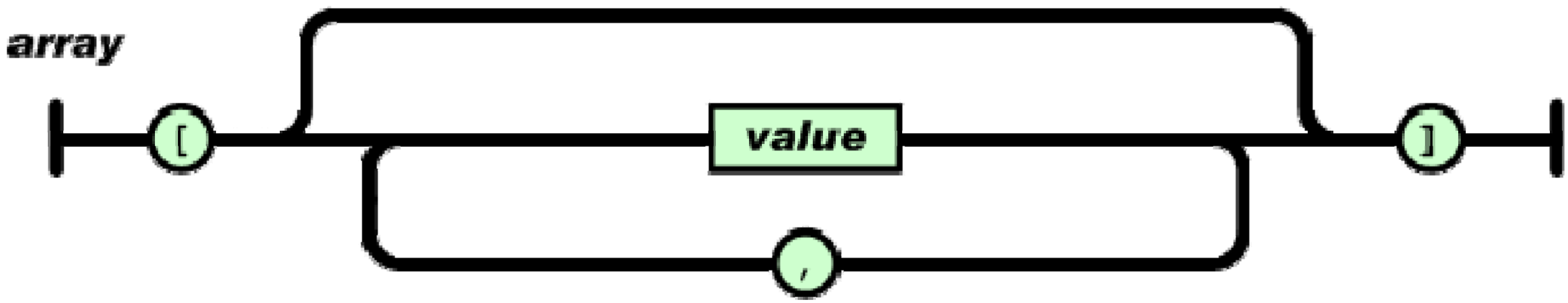
Object

An ***object*** is an unordered set of name/value pairs.



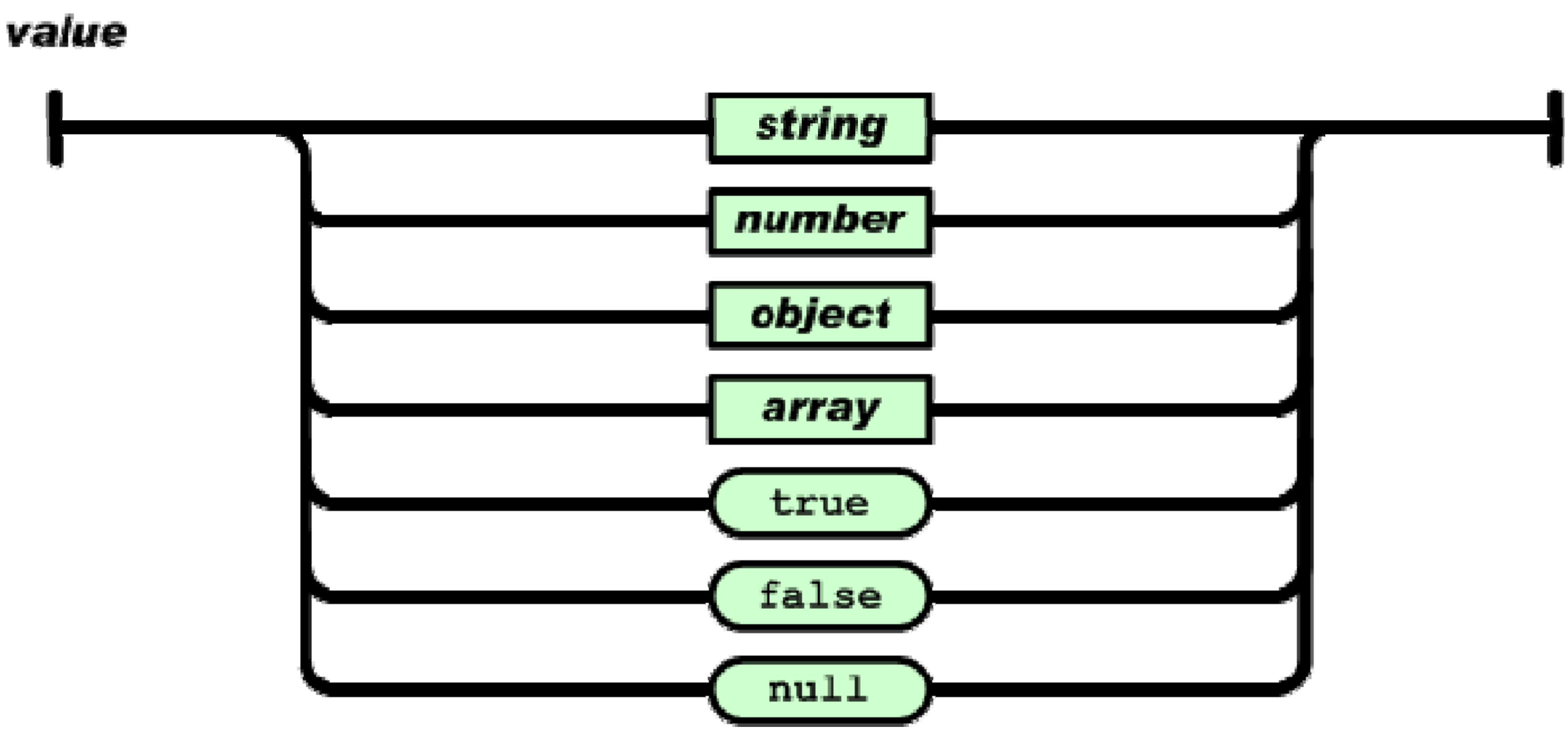
Array

An ***array*** is an ordered collection of values.



Value

A **value** can be a string, a number, true, false, null, an object, or an array.
These structures can be nested.



Examples

{name: "John", age: 31, city: "New York"}


```
{ "markers":
  [ { "name": "Rixos The Palm Dubai",
      "position": [25.1212, 55.1535] },
   { "name": "Shangri-La Hotel",
     "location": [25.2084, 55.2719] },
   { "name": "Grand Hyatt",
      "location": [25.2285, 55.3273] }
   ] }
```

Examples

```
{"menu": {
    "id": "file",
    "value": "File",
     "popup": {
          "menuitem": [
              {"value": "New", "onclick": "CreateNewDoc()"},
              {"value": "Open", "onclick": "OpenDoc()"},
              {"value": "Close", "onclick": "CloseDoc()"}
          ]
} }}
```