IF3230 – Sistem Terdistribusi
Clock Synchronization

Achmad Imam Kistijantoro (imam@informatika.org)
Judhi Santoso (judhi@informatika.org)
Anggrahita Bayu Sasmita (bayu.anggrahita@informatika.org)

Clock Synchronization

- physical clock
- logical clock
- vector clock

Physical clock

- Koordinasi antar proses yang berjalan konkuren sering memerlukan order (keterurutan) antar event
- Misal:
  - untuk menentukan urutan update terhadap data yang terreplikasi
  - Menentukan urutan pesan yang akan diproses/ditampilkan
  - Scheduler, timeout, failure detectors, performance measurements, cache validity
- Clock pada komputer berbasis quartz crystal clock, untuk yang standar dapat memiliki akurasi 6 ppm (sekitar ½ detik/hari)
- Clock yang bagus dapat mencapai akurasi 1 detik dalam 10 tahun, namun sensitif thd perubahan suhu, dan freq dapat berubah sesuai dengan usia quartz crystal

Physical clock

- Atomic clock:
- Waktu referensi didefinisikan sebagai 9,192,631,770 periode radiasi yang berkorespondensi dengan 2 hyperfine level dari cesium-133
- Akurasi 1 detik dalam 6 juta tahun
- Standar NIST sejak 1960
- Pewaktuan standar berbasis atomic clock: UTC (Coordinated Universal Time)

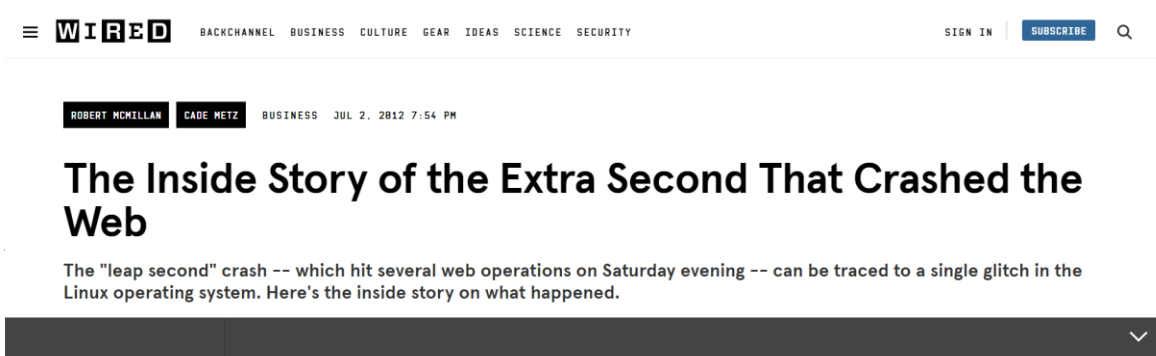
Leap second

- UTC menggunakan atomic clock, yang tidak persis sama dengan GMT (solar time) yang menggunakan rotasi bumi dan matahari => perputaran rotasi bumi tidak selalu konstan
- Kadang perlu dilakukan koreksi detik (leap second), dan dilakukan pada 30 juni dan 31 desember setiap tahun
  - Dimajukan 1 detik
  - Tetap
  - Mundur 1 detik

Leap second - problem

- Penanganan software/komputer terhadap leap second?
  - Diabaikan
- OS dan system terdistribusi sering bergantung pada timing dengan akurasi < 1 s
- 30 Juni 2012: bug pada linux mengakibatkan livelock pada leap second, menyebabkan banyak layanan Internet yang down

https://www.wired.com/2012/07/leap-second-glitch-explained/


WIRED BACKCHANNEL BUSINESS CULTURE GEAR IDEAS SCIENCE SECURITY SIGN IN SUBSCRIBE

ROBERT MCMILLAN CADE METZ BUSINESS JUL 2, 2012 7:54 PM

The Inside Story of the Extra Second That Crashed the Web

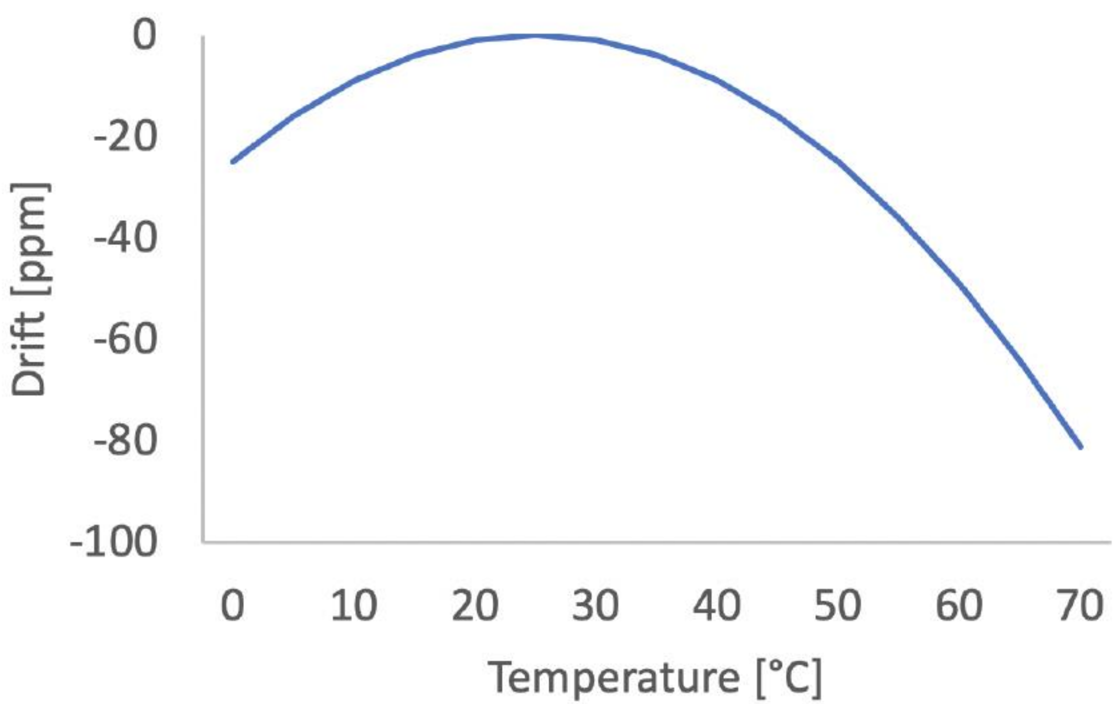
The "leap second" crash -- which hit several web operations on Saturday evening -- can be traced to a single glitch in the Linux operating system. Here's the inside story on what happened.

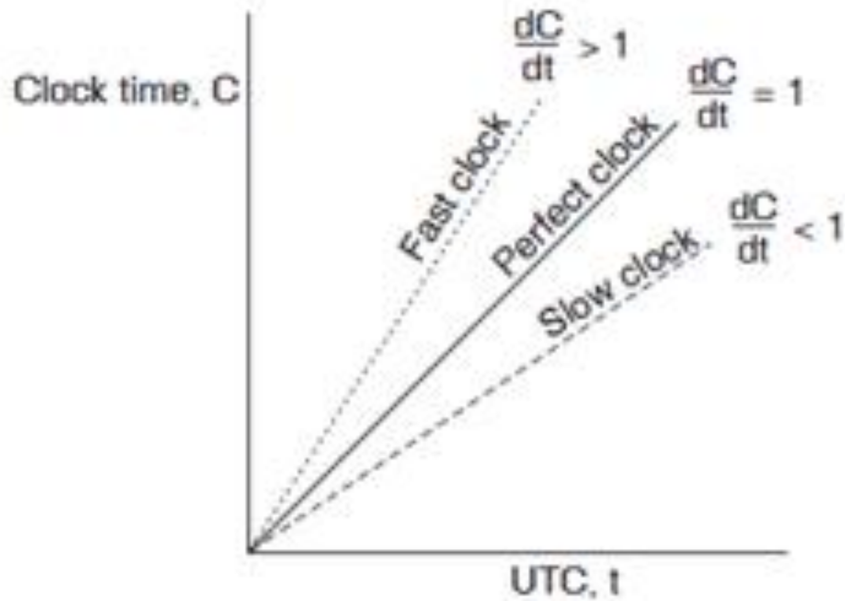
Physical clock

- Problem: 2 komputer tidak pernah memiliki physical clock yang sinkron
- Setiap quartz crystal memiliki frekuensi yang sedikit berbeda
  - Antar clock memiliki gap yang membesar dengan rate tertentu, yang disebut sebagai **time drift**
  - Selisih waktu antar 2 clock disebut sebagai **time skew**

Quartz clock error: drift

- Dipengaruhi lingkungan, e.g. suhu
- 1 ppm = 1 microsecond/second = 86 ms/day=32 s /year
- Typical computer: 50 ppm





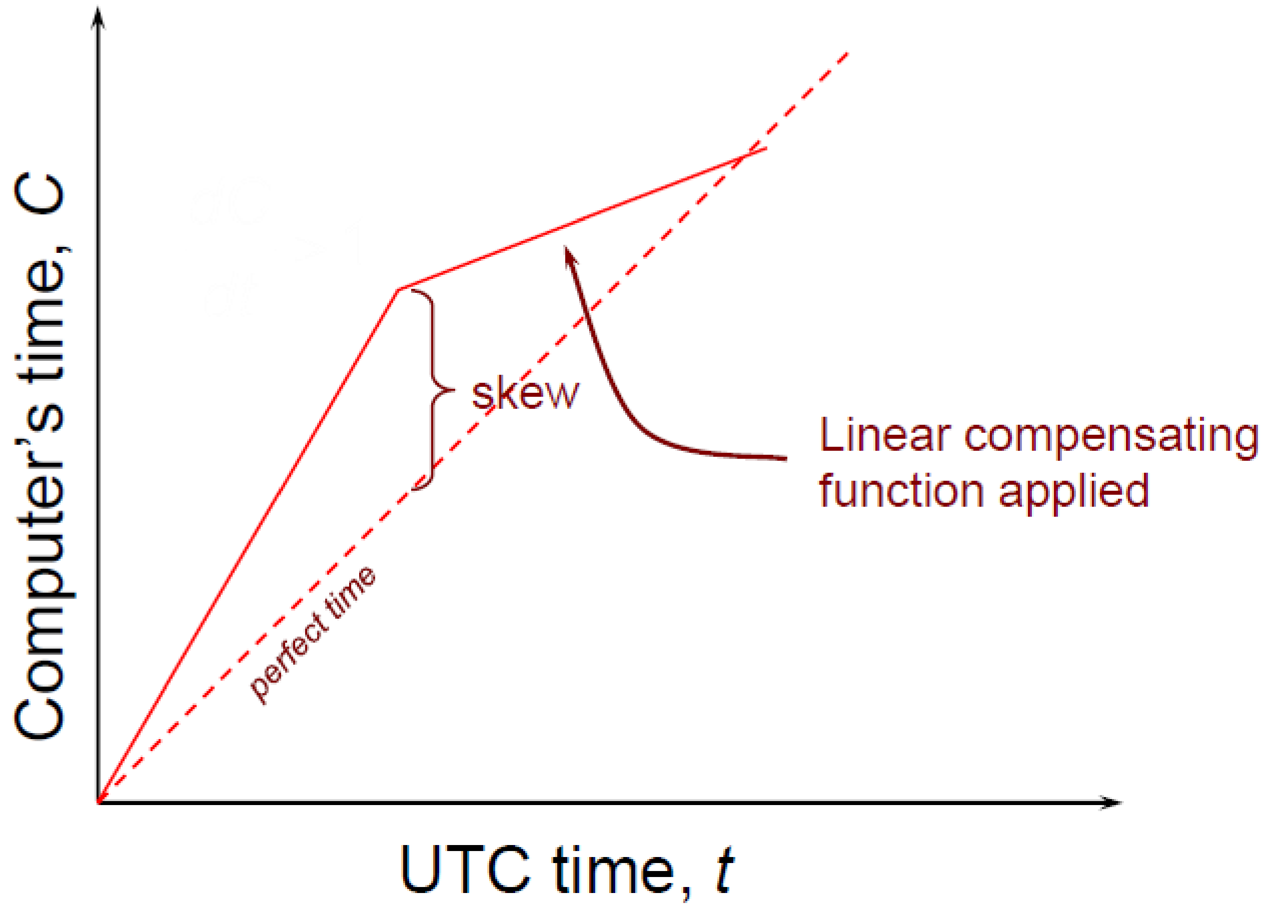
In practice: $1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho.$

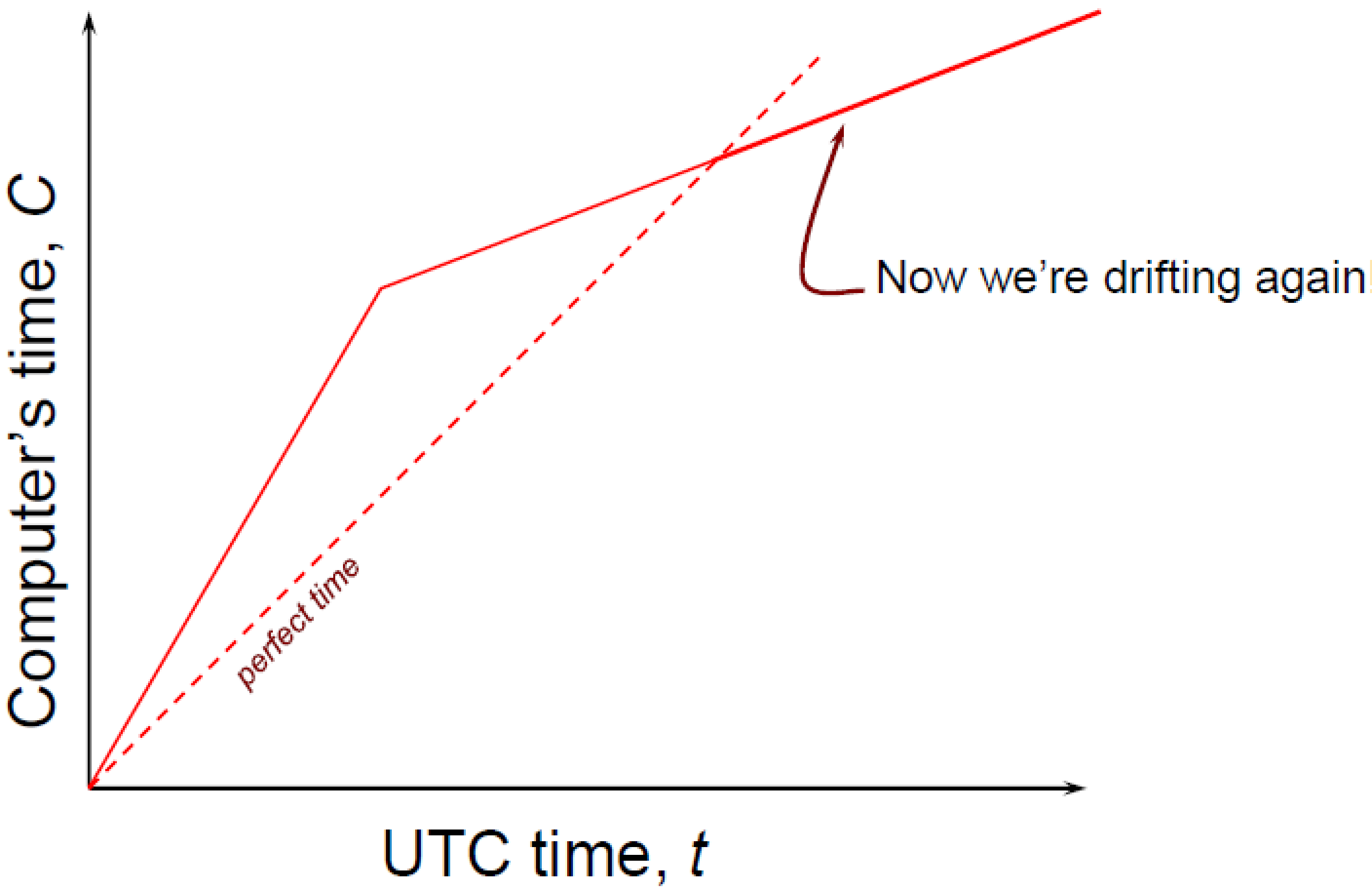
Penanganan drift

- Bagaimana mencocokkan waktu yang mengalami drift
- Clock sebaiknya tidak di-set mundur
  - Mengacaukan order message dan lingkungan pengembangan software

Penanganan drift

- Dengan koreksi gradual
  - Jika terlalu cepat, buat clock berjalan lebih lambat hingga sinkron
  - Jika terlalu lambat, buat clock berjalan lebih cepat hingga sinkron
  - Pada komputer, hal ini dapat dilakukan pada level OS, yaitu dengan mengubah frekuensi saat pembangkitan interrupt clock
  - Pada UNIX-based, disediakan oleh fungsi adjtime (lihat man pada Linux)


Computer's time, C
skew
Linear compensating
function applied
perfect time
UTC time, *t*

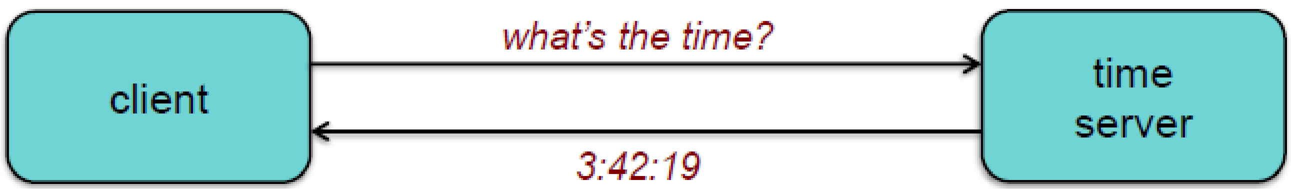

Computer's time, C
perfect time
Now we're drifting again!
UTC time, t

Mendapatkan waktu akurat

- Menggunakan GPS receiver yang terhubung ke komputer
  - Dapat memberikan waktu akurat hingga selisih 1 ms terhadap UTC
  - Tidak dapat digunakan in-door
- Di US, dapat menggunakan WWV radio receiver
  - Akurasi 3 -10 ms, tergantung lokasi
- Menggunakan GOES (Geostationary Operational Environment Satellites)
  - Akurasi 0.1 ms
- Solusi di atas tidak praktikal, sehingga umumnya sinkronisasi dilakukan dengan mencocokkan waktu dengan komputer lain yang lebih akurat => time server

Sinkronisasi

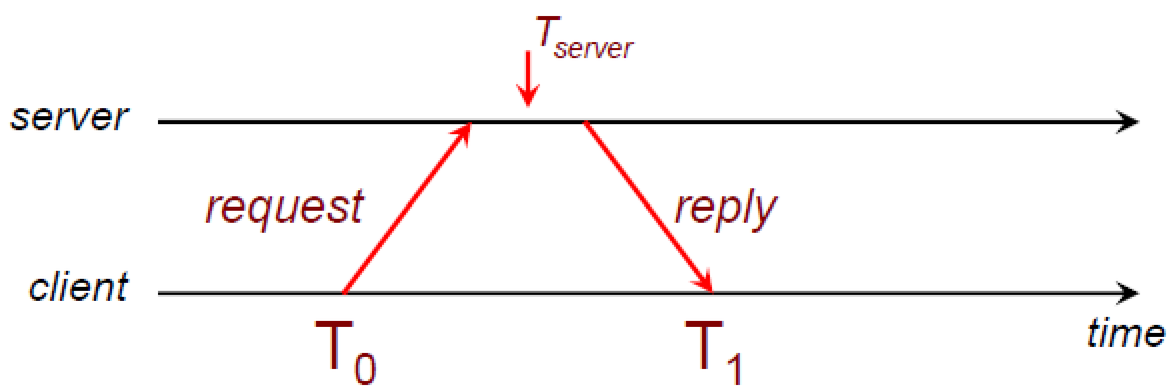
- Cara sederhana
  - Meminta waktu melalui jaringan
  - Set waktu sesuai dengan jawaban



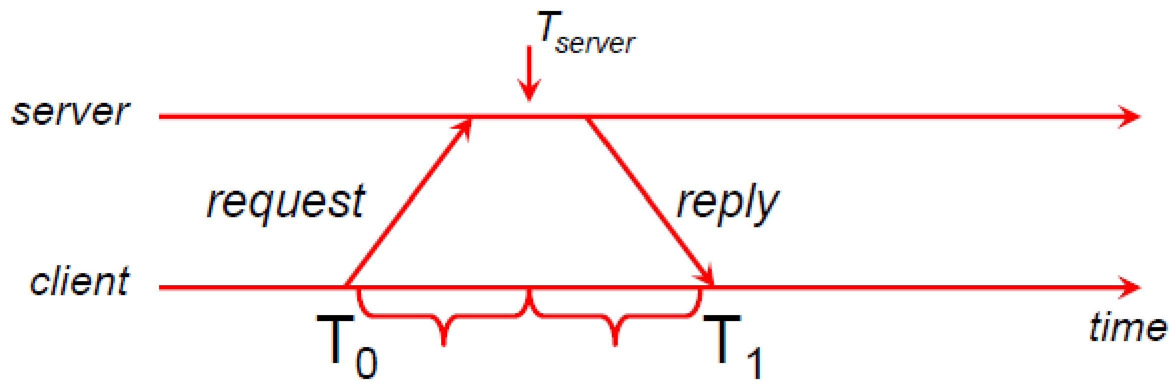
  - Belum mempertimbangkan network delay

Algoritma Cristian

- Kompensasi delay
  - T_0 : request dikirim
  - T_1 : reply diterima
- Asumsi network delay simetrik



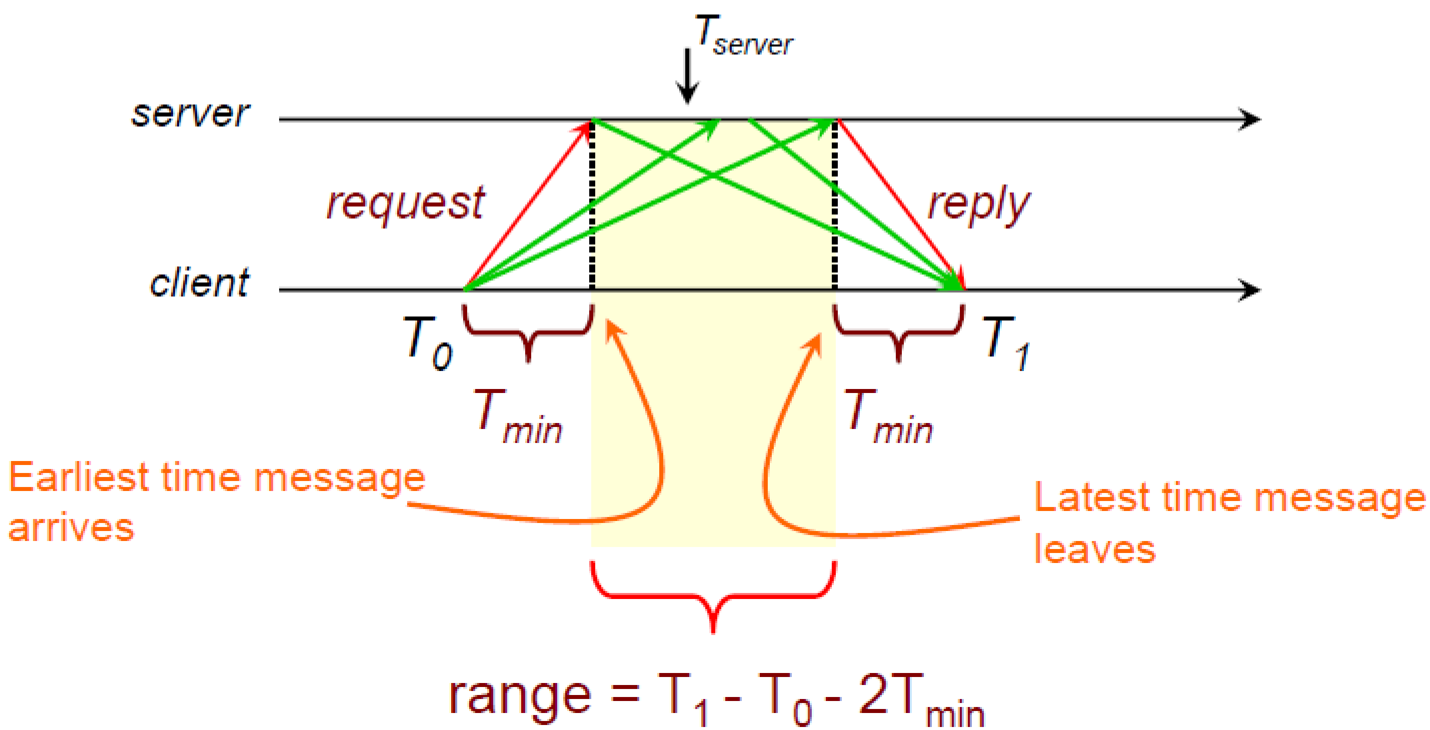
Algoritma Cristian



- $T_{new} = T_{server} + (T_1 - T_0)/2$

Algoritma Cristian

- Jika waktu pengiriman pesan minimum diketahui, dapat dihitung batasan akurasi



- Akurasi: $(T_1 - T_0)/2 - T_{min}$

Algoritma Berkeley

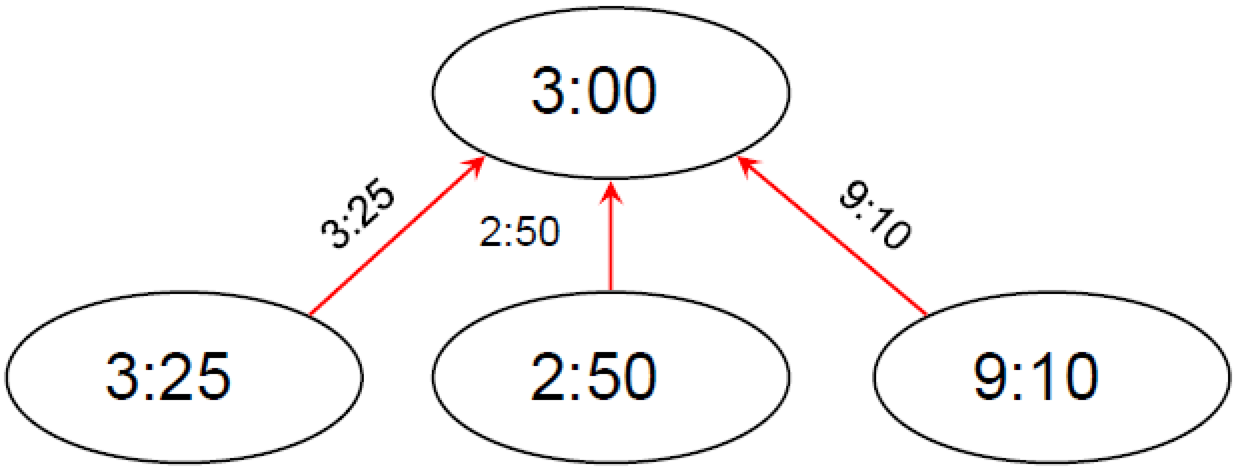
- Gusella & Zatti, 1989

- Asumsi: tidak ada mesin yang memiliki sumber waktu akurat
- Menghitung rata2 waktu dari semua komputer
- Sinkronisasi semua komputer dengan waktu rata2

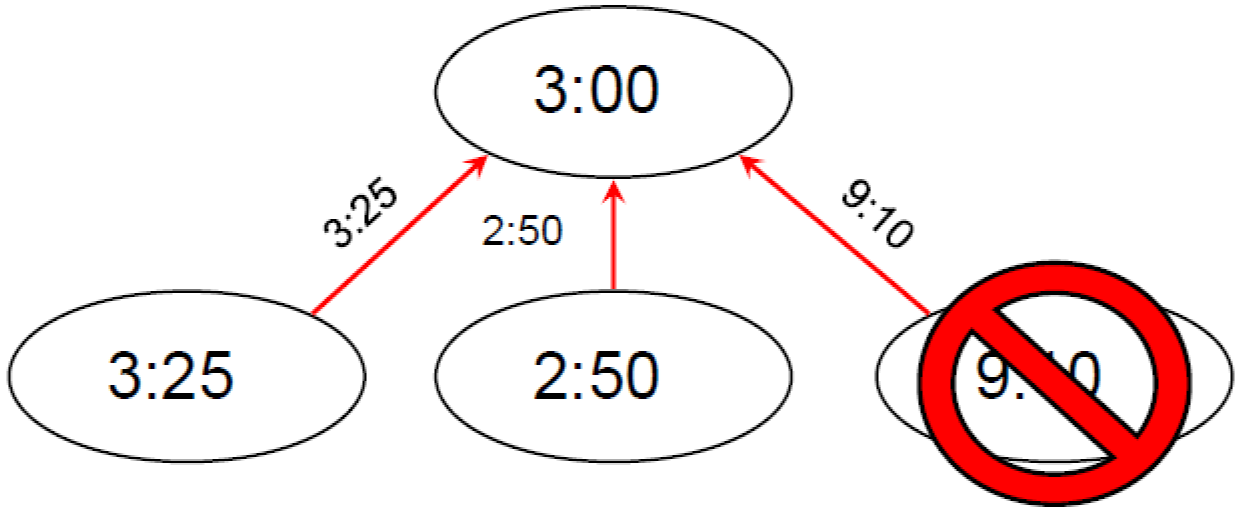
Algoritma Berkeley

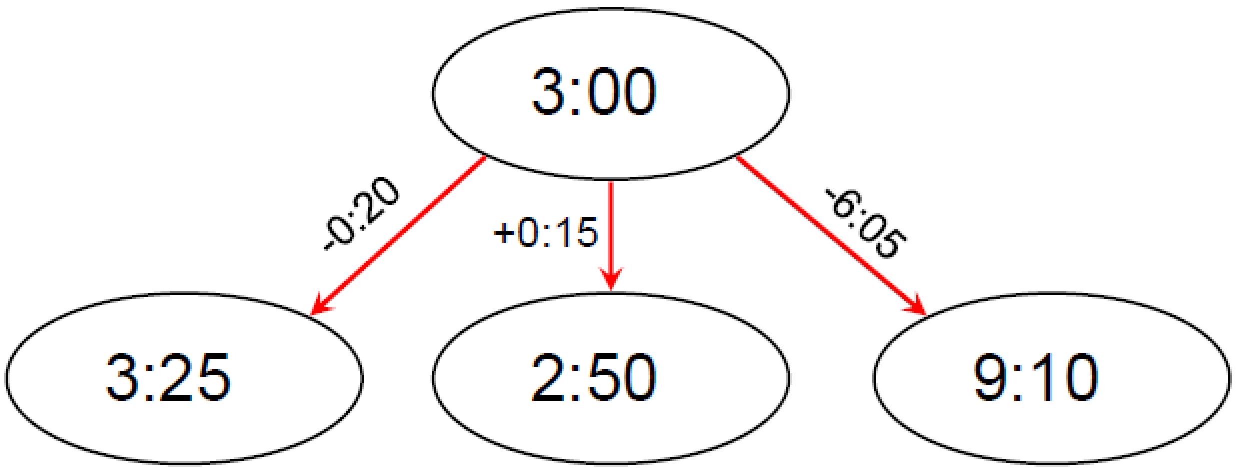
- Komputer menjalankan daemon yang mengimplementasikan protokol
- 1 mesin berfungsi sebagai master, lainnya slave
- Master poll setiap komputer periodik, menanyakan waktu
  - Dapat menggunakan algoritma cristian untuk kompensasi latency
- Saat reply diterima master, hitung waktu rata2
- Master mengirimkan informasi offset ke semua komputer
- Mesin yang memiliki beda waktu besar diabaikan

Berkeley Algorithm



- Waktu rata2: 3:25,2:50, 3:00 = 3:05




3:00
-0:20
+0:15
-6:05
3:25
2:50
9:10

Network Time Protocol

- 1991, 1992, Internet Standard v3: RFC 1305
- June 2010
  - Internet Standard v4: RFC 5905-5908
  - IPv6 support
  - Dynamic server discovery

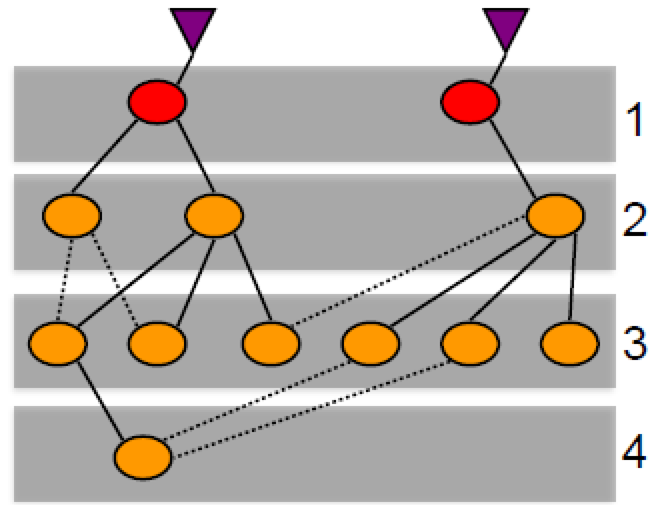
NTP Goals

- 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

NTP Servers

Arranged in strata

- 1st stratum: machines connected directly to accurate time source
- 2nd stratum: machines synchronized from 1st stratum machines
- ...



Synchronization Subnet

NTP Synchronization Modes

Multicast mode

- for high speed LANS
- Lower accuracy but efficient

Procedure call mode

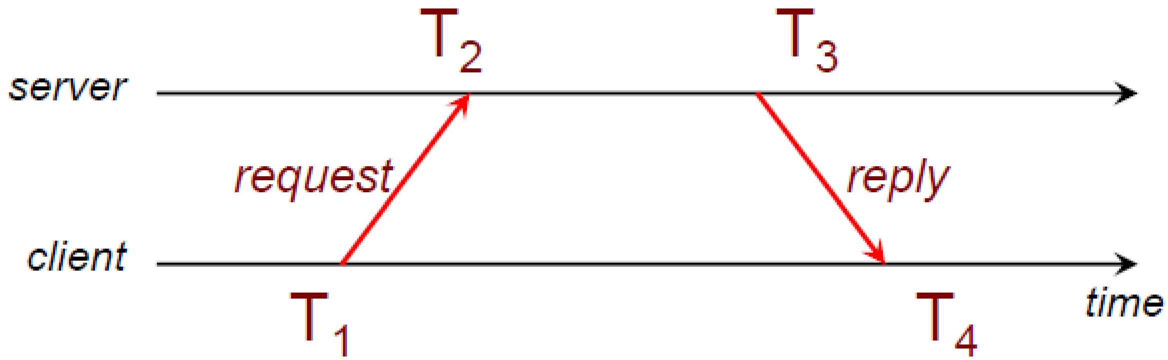
- Similar to Cristian's algorithm

Symmetric mode

- Intended for master servers
- Peer servers can synchronize with each other to provide mutual backup
  - Pair of servers retain data to improve synchronization over time

All messages delivered unreliably with UDP

Simple NTP



- $d = (T_4 - T_1) - (T_3 - T_2)$
- Offset: $((T_2 - T_1) + (T_3 - T_4))/2$

Logical Clock

the happened before relationship

Problem

We first need to introduce a notion of ordering before we can order anything.

The happened-before relation

- If a and b are two events in the same process, and a comes before b, then $a \rightarrow b$.
- If a is the sending of a message, and b is the receipt of that message, then $a \rightarrow b$
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

Note

This introduces a partial ordering of events in a system with concurrently operating processes.

logical clock

Problem

How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?

Solution

Attach a timestamp $C(e)$ to each event e, satisfying the following properties:

- P1 If a and b are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.
- P2 If a corresponds to sending a message m, and b to the receipt of that message, then also $C(a) < C(b)$.

Problem

How to attach a timestamp to an event when there's no global clock $\Rightarrow$ maintain a consistent set of logical clocks, one per process.

logical clock

Solution

Each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:

1: For any two successive events that take place within P_i, C_i is incremented by 1.

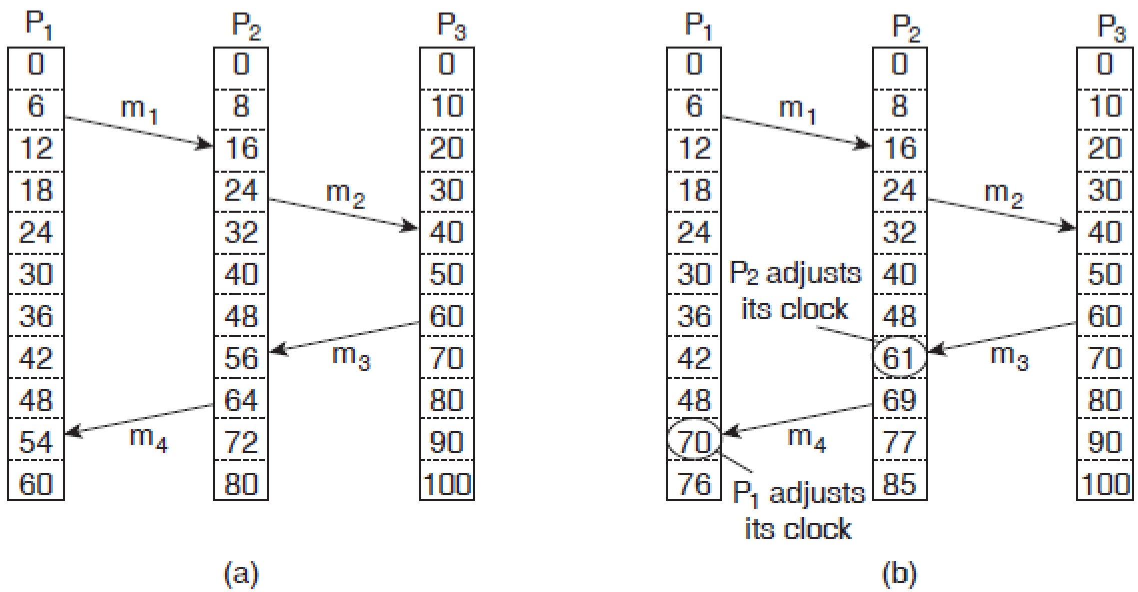
2: Each time a message m is sent by process P_i, the message receives a timestamp $ts(m) = C_i$.

3: Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j to $\max\{C_j, ts(m)\}$; then executes step 1 before passing m to the application.

Notes

- Property P1 is satisfied by (1); Property P2 by (2) and (3).
- It can still occur that two events happen at the same time. Avoid this by breaking ties through process IDs.

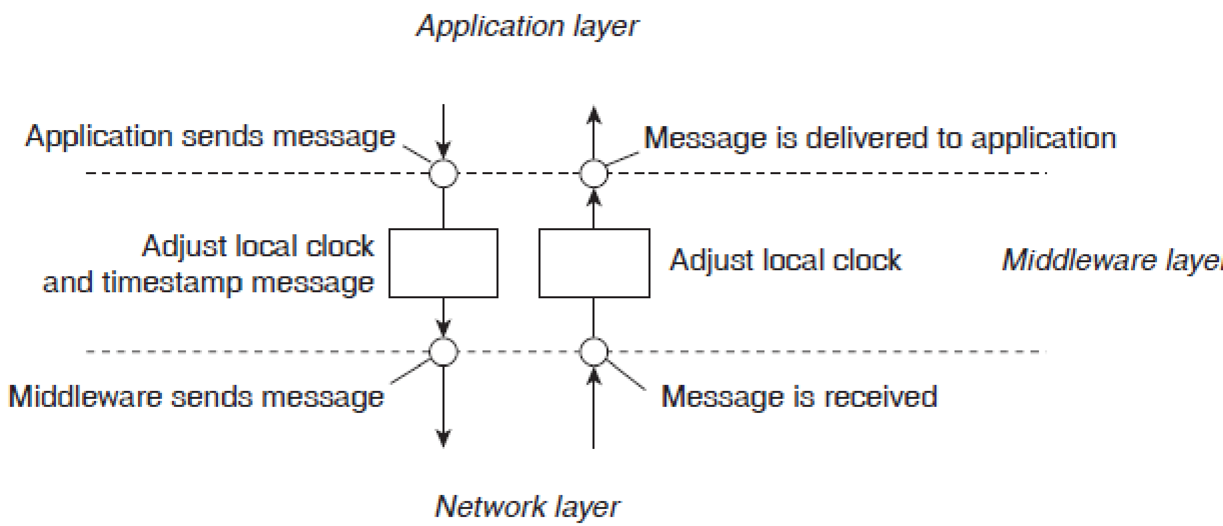
logical clock



logical clock - example

Note

Adjustments take place in the middleware layer

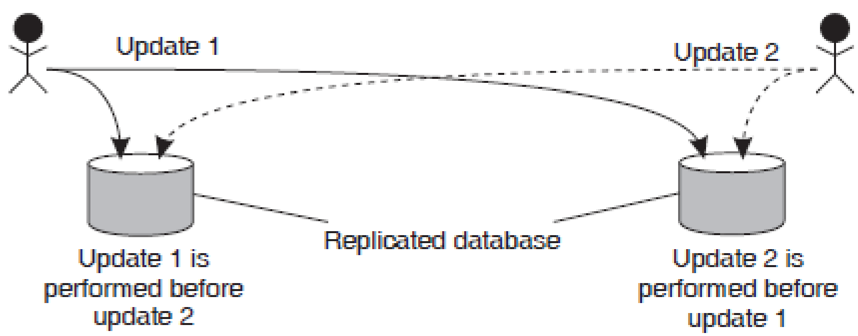


example – totally ordered multicast

Problem

We sometimes need to guarantee that concurrent updates on a replicated database are seen in the same order everywhere:

- P_1 adds \$100 to an account (initial value: \$1000)
- P_2 increments account by 1%
- There are two replicas



Result

In absence of proper synchronization:
replica #1 $\leftarrow$ \$1111, while replica #2 $\leftarrow$ \$1110.

example – totally ordered multicast

Solution

- Process P_i sends timestamped message msg_i to all others. The message itself is put in a local queue $queue_i$.
- Any incoming message at P_j is queued in $queue_j$, according to its timestamp, and acknowledged to every other process.

P_j passes a message msg_i to its application if:

(1) msg_i is at the head of $queue_j$
(2) for each process P_k, there is a message msg_k in $queue_j$ with a larger timestamp.

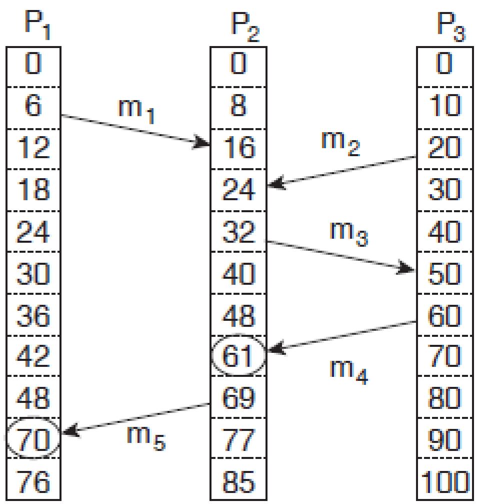
Note

We are assuming that communication is reliable and FIFO ordered.

vector clock

Observation

Lamport's clocks do not guarantee that if $C(a) < C(b)$ that a causally preceded b



Observation

Event a: m_1 is received at $T = 16$;
Event b: m_2 is sent at $T = 20$.

Note

We cannot conclude that a causally precedes b.

vector clock

Solution

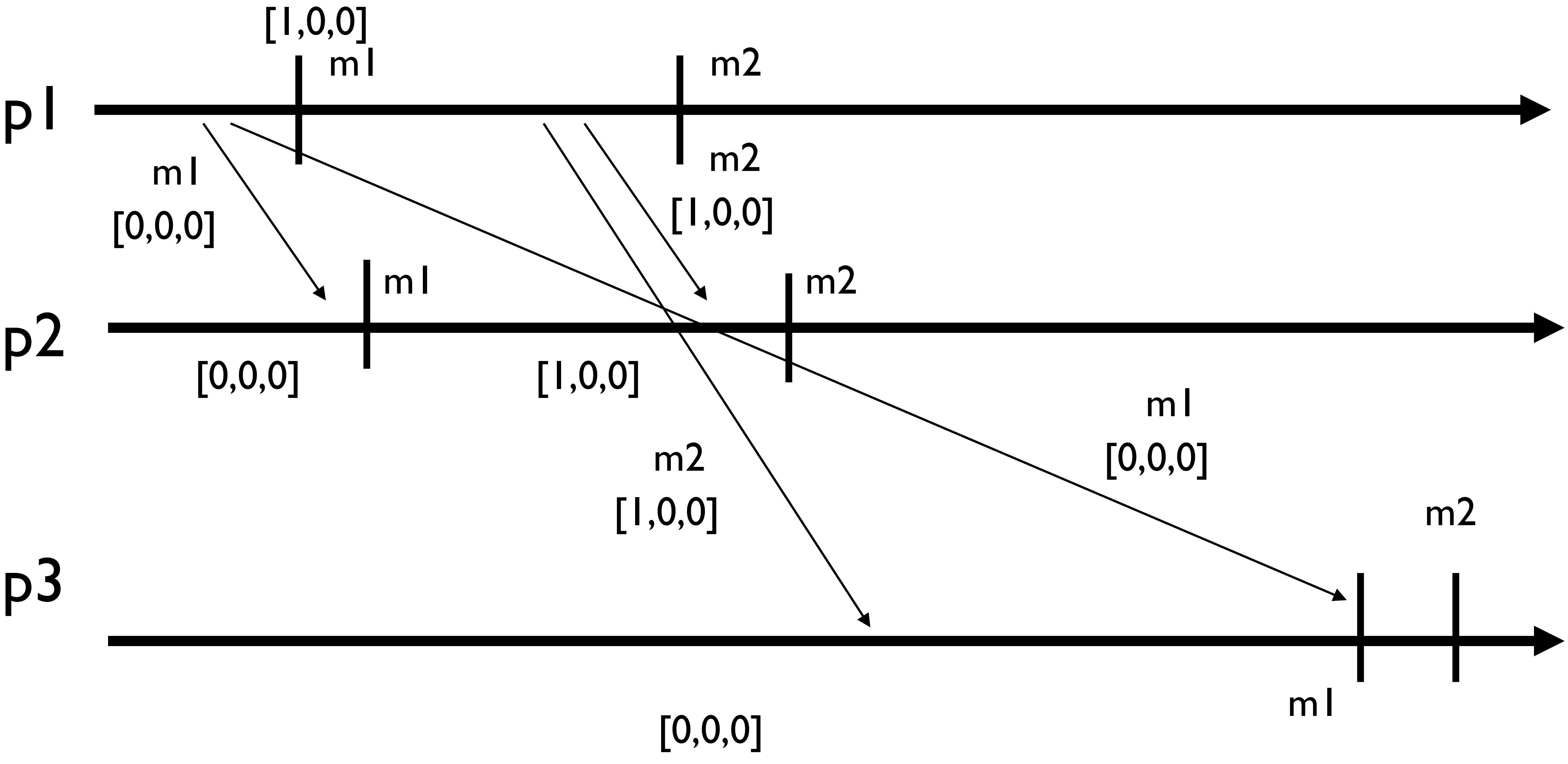
- Each process P_i has an array $VC_i[1..n]$, where $VC_i[j]$ denotes the number of events that process P_i knows have taken place at process P_j.
- When P_i sends a message m, it adds 1 to $VC_i[i]$, and sends VC_i along with m as vector timestamp $vt(m)$. Result: upon arrival, recipient knows P_i's timestamp.
- When a process P_j delivers a message m that it received from P_i with vector timestamp $ts(m)$, it

  (1) updates each $VC_j[k]$ to $\max\{VC_j[k], ts(m)[k]\}$
  (2) increments $VC_j[j]$ by 1.

Question

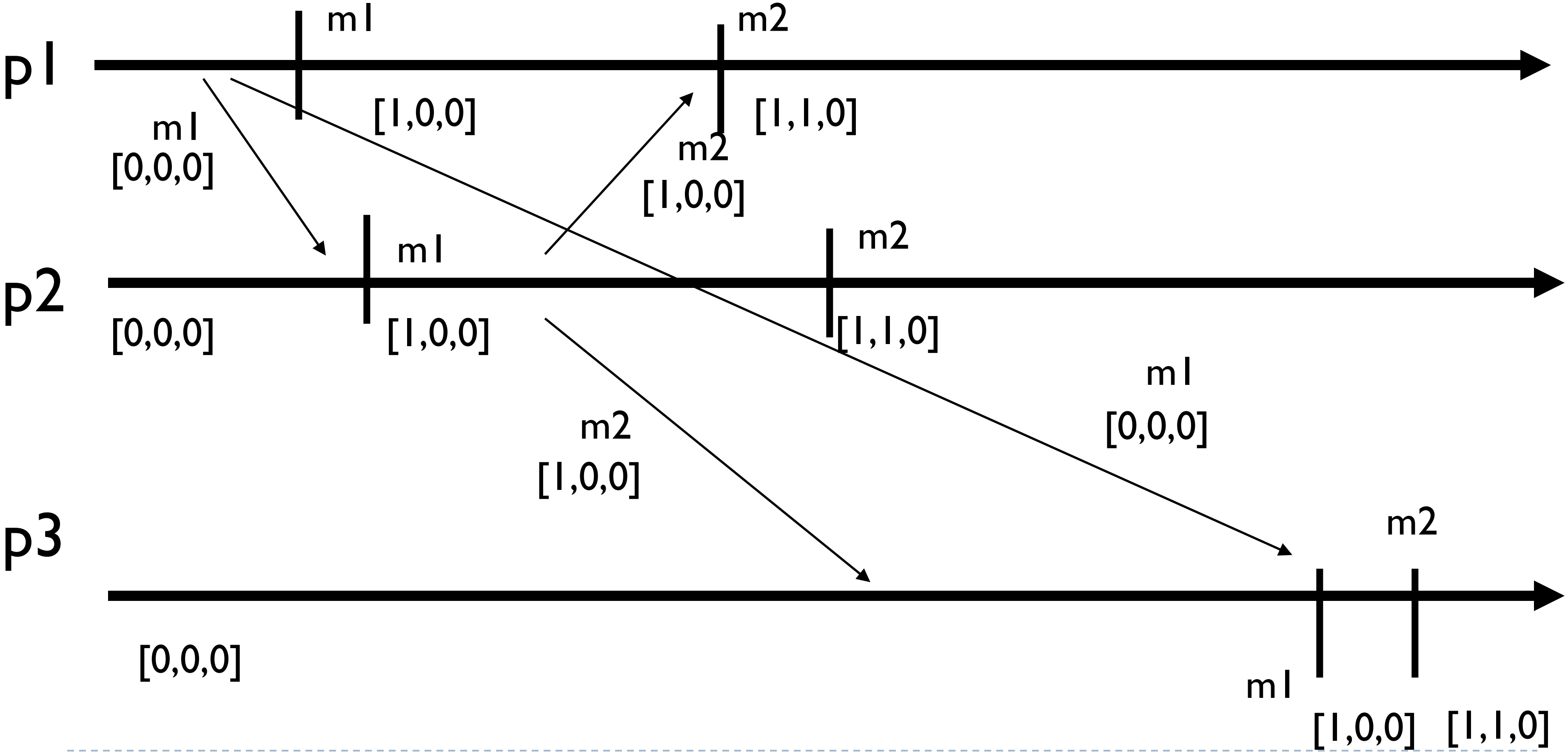
What does $VC_i[j] = k$ mean in terms of messages sent and received?

Contoh



[1,0,0]
m1
m2
p1
m2
m1
[0,0,0]
[1,0,0]
m1
m2
p2
[0,0,0]
[1,0,0]
m1
[0,0,0]
m2
[1,0,0]
m2
p3
m1
[0,0,0]

Contoh



p1
m1
m2
[1,0,0]
[1,1,0]
m1
[0,0,0]
m2
[1,0,0]
p2
m1
m2
[0,0,0]
[1,0,0]
[1,1,0]
m1
[0,0,0]
m2
[1,0,0]
p3
m2
[0,0,0]
m1
[1,0,0]
[1,1,0]

Sumber

- Paul Krzyzanowski, Clock Synchronization, Lectures on Distributed Systems, Rutgers University 2013
- Andrew S. Tanenbaum & Marten v. Steen, Distributed Systems Principles and Paradigms, 2nd edition, Chapter 6, Prentice Hall, 2007