Distributed Systems

(3rd Edition)

Maarten van Steen   Andrew S. Tanenbaum

Chapter 06: Coordination

Edited by: Hicham G. Elmongui

Physical clocks

Problem

Sometimes we simply need the exact time, not just an ordering.

Solution: Universal Coordinated Time (UTC)

- Based on the number of transitions per second of the cesium 133 atom (pretty accurate).
- At present, the real time is taken as the average of some 50 cesium clocks around the world.
- Introduces a leap second from time to time to compensate that days are getting longer.

Note

UTC is broadcast through short-wave radio and satellite. Satellites can give an accuracy of about ± 0.5 ms.

Clock synchronization

Precision

The goal is to keep the deviation between two clocks on any two machines within a specified bound, known as the precision π:

$$\forall t, \forall p, q : |C_p(t) - C_q(t)| \leq \pi$$

with $C_p(t)$ the computed clock time of machine p at UTC time t.

Accuracy

In the case of accuracy, we aim to keep the clock bound to a value α:

$$\forall t, \forall p : |C_p(t) - t| \leq \alpha$$

Synchronization

- Internal synchronization: keep clocks precise
- External synchronization: keep clocks accurate

Clock drift

Clock specifications

- A clock comes specified with its maximum clock drift rate ρ.
- $F(t)$ denotes oscillator frequency of the hardware clock at time t
- F is the clock's ideal (constant) frequency $\Rightarrow$ living up to specifications:

$$\forall t : (1-\rho) \leq \frac{F(t)}{F} \leq (1+\rho)$$

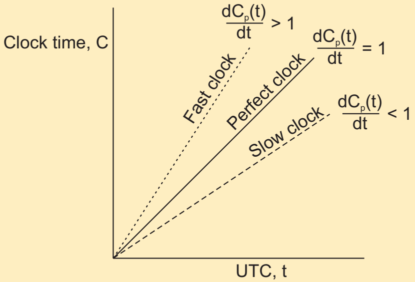
Observation

By using hardware interrupts we couple a software clock to the hardware clock, and thus also its clock drift rate:

$$C_p(t) = \frac{1}{F}\int_0^t F(t)dt \Rightarrow \frac{dC_p(t)}{dt} = \frac{F(t)}{F}$$

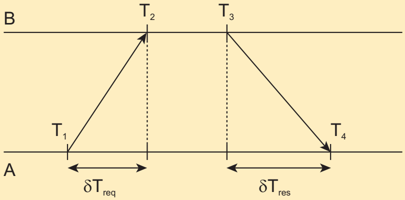
$$\Rightarrow \forall t : 1-\rho \leq \frac{dC_p(t)}{dt} \leq 1+\rho$$

Fast, perfect, slow clocks



Detecting and adjusting incorrect times

Getting the current time from a time server



Computing the relative offset θ and delay δ

Assumption: $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$

$$\theta = T_3 + \big((T_2 - T_1) + (T_4 - T_3)\big)/2 - T_4 = \big((T_2 - T_1) + (T_3 - T_4)\big)/2$$

$$\delta = \big((T_4 - T_1) - (T_3 - T_2)\big)/2$$

Network Time Protocol

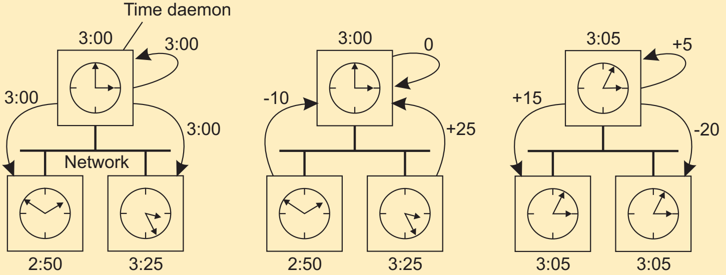
Collect eight (θ, δ) pairs and choose θ for which associated delay δ was minimal.

Keeping time without UTC

Principle

Let the time server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time relative to its present time.

Using a time server



Fundamental

You'll have to take into account that setting the time back is never allowed ⇒ smooth adjustments (i.e., run faster or slower).

The Happened-before relationship

Issue

What usually matters is not that all processes agree on exactly what time it is, but that they agree on the order in which events occur. Requires a notion of ordering.

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 clocks

Problem

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

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

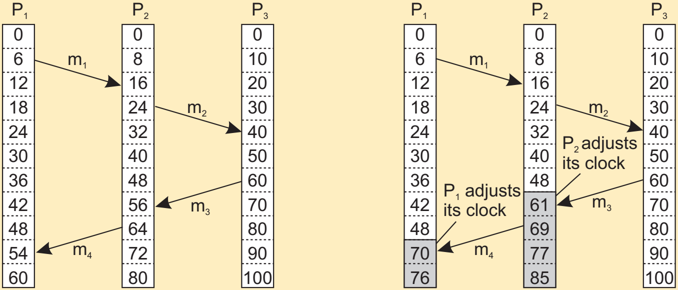
- **P1** If a and b are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.
- **P2** If a corresponds to sending a message m, and b to the receipt of that message, then also $C(a) < C(b)$.

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.

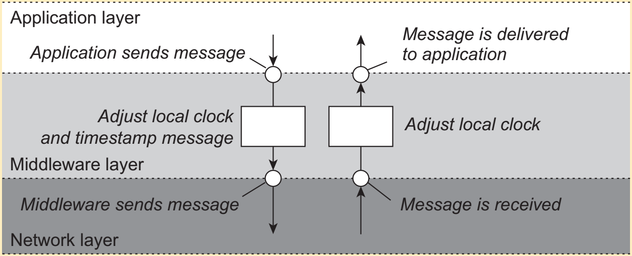
Logical clocks: example

Consider three processes with event counters operating at different rates



Logical clocks: where implemented

Adjustments implemented in middleware



Logical clocks: solution

Each process P_i maintains a local counter C_i and adjusts this counter

1. For each new event that takes 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.

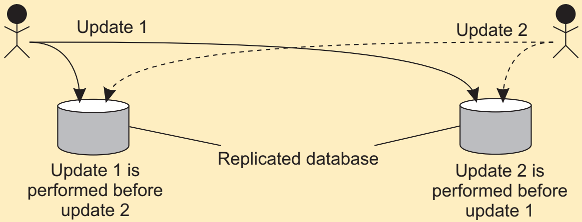
Notes

- 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.

Example: Total-ordered multicast

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: Total-ordered multicast

Solution

- Process P_i sends timestamped message m_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 m_i to its application if:

(1) m_i is at the head of $queue_j$

(2) for each process P_k, there is a message m_k in $queue_j$ with a larger timestamp.

Note

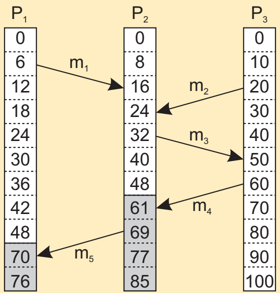
We are assuming that communication is reliable and FIFO ordered.

Vector clocks

Observation

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

Concurrent message transmission using logical clocks



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.

Causal dependency

Precedence vs. dependency

- We say that *a* causally precedes *b*.
- *b* may causally depend on *a*, as there may be information from *a* that is propagated into *b*.

Capturing causality

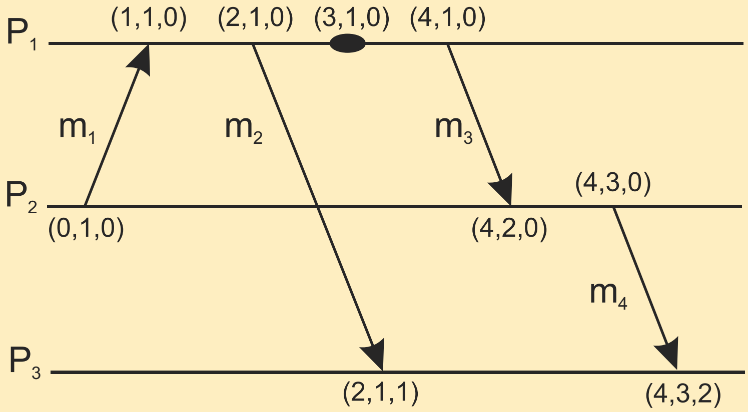
Solution: each P_i maintains a vector VC_i

- $VC_i[i]$ is the local logical clock at process P_i.
- If $VC_i[j] = k$ then P_i knows that k events have occurred at P_j.

Maintaining vector clocks

1. Before executing an event P_i executes $VC_i[i] \leftarrow VC_i[i] + 1$.
2. When process P_i sends a message m to P_j, it sets m's (vector) timestamp $ts(m)$ equal to VC_i after having executed step 1.
3. Upon the receipt of a message m, process P_j sets $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ for each k, after which it executes step 1 and then delivers the message to the application.

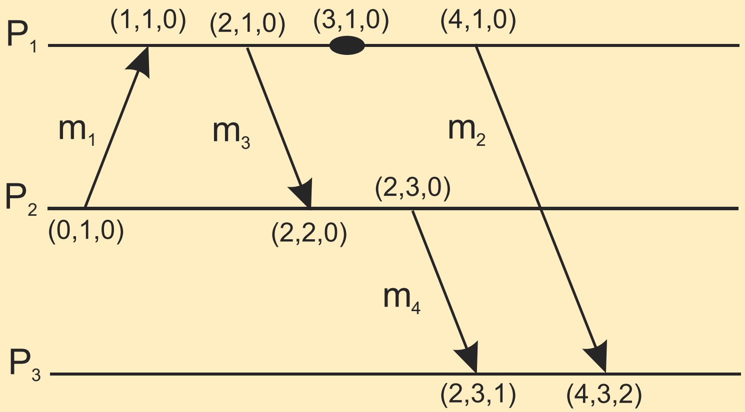
Vector clocks: Example



Potential Causal Precedence

$$ts(m_2) < ts(m_4)$$

Vector clocks: Example



Concurrent Events

$$ts(m_2) \not< ts(m_4) \quad \& \quad ts(m_4) \not< ts(m_2)$$

Mutual exclusion

Problem

A number of processes in a distributed system want exclusive access to some resource.

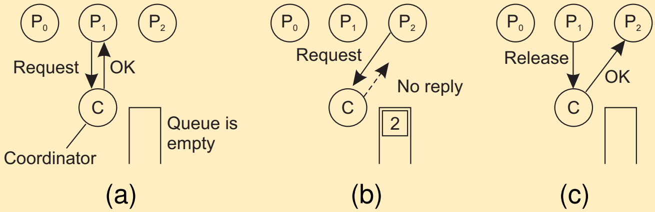
Basic solutions

Permission-based: A process wanting to enter its critical section, or access a resource, needs permission from other processes.

Token-based: A token is passed between processes. The one who has the token may proceed in its critical section, or pass it on when not interested.

Permission-based, centralized

Simply use a coordinator



(a) Process P_1 asks the coordinator for permission to access a shared resource. Permission is granted.

(b) Process P_2 then asks permission to access the same resource. The coordinator does not reply.

(c) When P_1 releases the resource, it tells the coordinator, which then replies to P_2.

Mutual exclusion Ricart & Agrawala

The same as Lamport except that acknowledgments are not sent

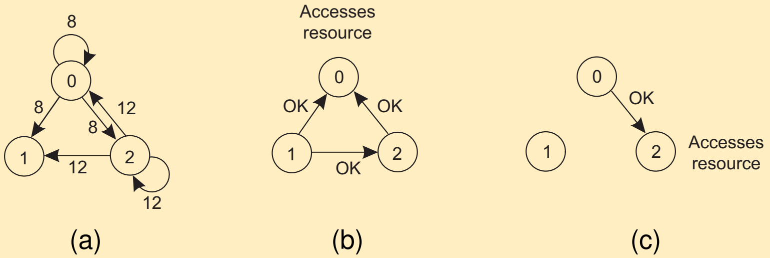
Return a response to a request only when:

- The receiving process has no interest in the shared resource; or
- The receiving process is waiting for the resource, but has lower priority (known through comparison of timestamps).

In all other cases, reply is deferred, implying some more local administration.

Mutual exclusion Ricart & Agrawala

Example with three processes



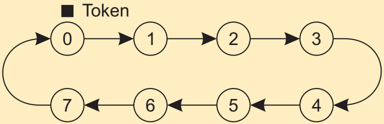
(a) Two processes want to access a shared resource at the same moment.
(b) P_0 has the lowest timestamp, so it wins.
(c) When process P_0 is done, it sends an *OK* also, so P_2 can now go ahead.

Mutual exclusion: Token ring algorithm

Essence

Organize processes in a logical ring, and let a token be passed between them. The one that holds the token is allowed to enter the critical region (if it wants to).

An overlay network constructed as a logical ring with a circulating token



Decentralized mutual exclusion

Principle

Assume every resource is replicated N times, with each replica having its own coordinator $\Rightarrow$ access requires a majority vote from $m > N/2$ coordinators. A coordinator always responds immediately to a request.

Assumption

When a coordinator crashes, it will recover quickly, but will have forgotten about permissions it had granted.

Decentralized mutual exclusion

How robust is this system?

- Let $p = \Delta t / T$ be the probability that a coordinator resets during a time interval Δt, while having a lifetime of T.
- The probability $\mathbb{P}[k]$ that k out of m coordinators reset during the same interval is
$$\mathbb{P}[k] = \binom{m}{k} p^k (1-p)^{m-k}$$
- f coordinators reset $\Rightarrow$ correctness is violated when there is only a minority of nonfaulty coordinators: when $m - f \leq N/2$, or, $f \geq m - N/2$.
- The probability of a violation is $\sum_{k=m-N/2}^{N} \mathbb{P}[k]$.

Decentralized mutual exclusion

Violation probabilities for various parameter values

N	m	p	Violation
8	5	3 sec/hour	$< 10^{-15}$
8	6	3 sec/hour	$< 10^{-18}$
16	9	3 sec/hour	$< 10^{-27}$
16	12	3 sec/hour	$< 10^{-36}$
32	17	3 sec/hour	$< 10^{-52}$
32	24	3 sec/hour	$< 10^{-73}$

N	m	p	Violation
8	5	30 sec/hour	$< 10^{-10}$
8	6	30 sec/hour	$< 10^{-11}$
16	9	30 sec/hour	$< 10^{-18}$
16	12	30 sec/hour	$< 10^{-24}$
32	17	30 sec/hour	$< 10^{-35}$
32	24	30 sec/hour	$< 10^{-49}$

What can we conclude?

In general, the probability of violating correctness can be so low that it can be neglected in comparison to other types of failure.

If a process is denied access to a resource (getting $< m$ votes), it will back off for some randomly chosen time, and make a next attempt later.

Election algorithms

Principle

An algorithm requires that some process acts as a coordinator. The question is how to select this special process dynamically.

Note

In many systems the coordinator is chosen by hand (e.g. file servers). This leads to centralized solutions $\Rightarrow$ single point of failure.

Teasers

1. If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?
2. Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?

Basic assumptions

- All processes have unique id's
- All processes know id's of all processes in the system (but not if they are up or down)
- Election means identifying the process with the highest id that is up

Election by bullying

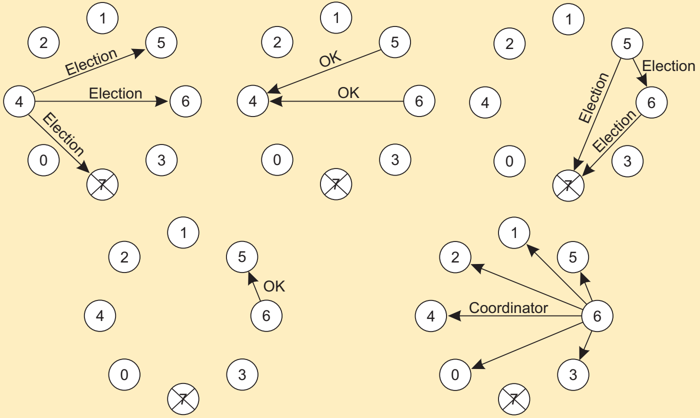
Principle

Consider N processes $\{P_0, \ldots, P_{N-1}\}$ and let $id(P_k) = k$. When a process P_k notices that the coordinator is no longer responding to requests, it initiates an election:

1. P_k sends an *ELECTION* message to all processes with higher identifiers: $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$.
2. If no one responds, P_k wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over and P_k's job is done.

Election by bullying

The bully election algorithm



Election in a ring

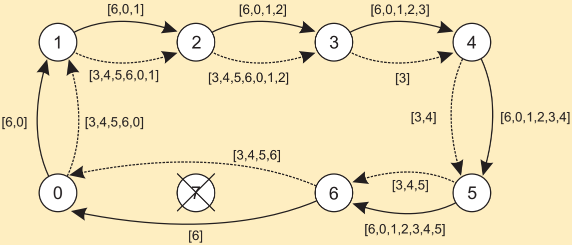
Principle

Process priority is obtained by organizing processes into a (logical) ring. Process with the highest priority should be elected as coordinator.

- Any process can start an election by sending an election message to its successor. If a successor is down, the message is passed on to the next successor.
- If a message is passed on, the sender adds itself to the list. When it gets back to the initiator, everyone had a chance to make its presence known.
- The initiator sends a coordinator message around the ring containing a list of all living processes. The one with the highest priority is elected as coordinator.

Election in a ring

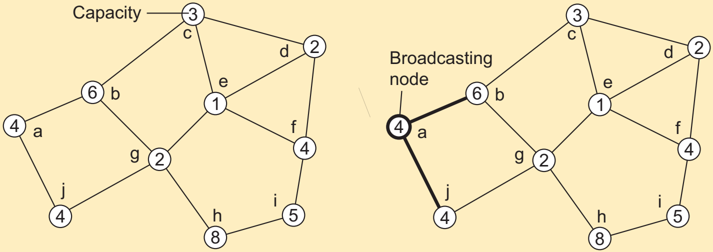
Election algorithm using a ring



- The solid line shows the election messages initiated by P_6
- The dashed one the messages by P_3

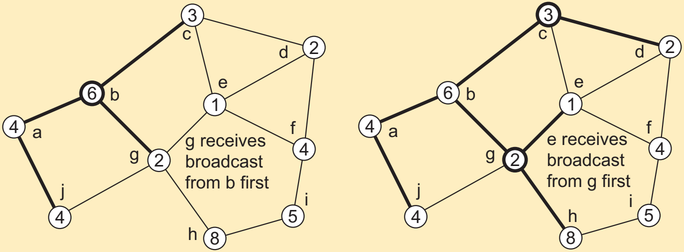
A solution for wireless networks

A sample network



A solution for wireless networks

A sample network



A solution for wireless networks

A sample network