

# Distributed Systems

## Principles and Paradigms

Maarten van Steen

VU Amsterdam, Dept. Computer Science  
steen@cs.vu.nl

## Chapter 06: Synchronization

Version: November 19, 2012



# Clock Synchronization

- Physical clocks
- Logical clocks
- Vector clocks

# 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  $\pm 0.5$  ms.

# Physical clocks

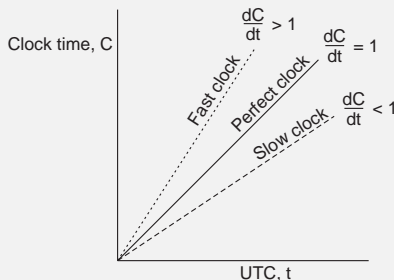
## Problem

Suppose we have a distributed system with a UTC-receiver somewhere in it  $\Rightarrow$  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  $p$ ,  $C_p(t) = t$ , or, in other words,  $dC/dt = 1$ .

# Physical clocks



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

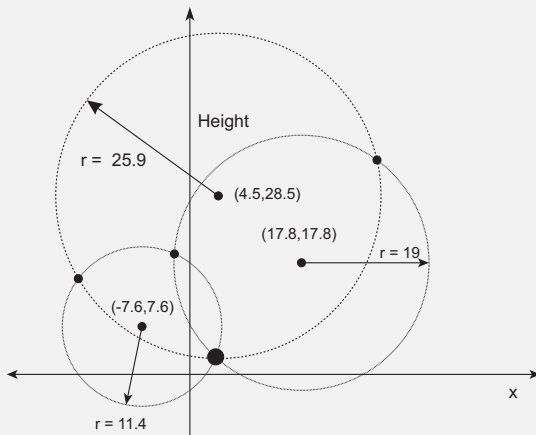
## Goal

Never let two clocks in any system differ by more than  $\delta$  time units  $\Rightarrow$  synchronize at least every  $\delta/(2\rho)$  seconds.

# Global positioning system

## Basic idea

You can get an accurate account of time as a side-effect of GPS.



# Global positioning system

## Problem

Assuming that the clocks of the satellites are accurate and synchronized:

- It takes a while before a signal reaches the receiver
- The receiver's clock is definitely out of synch with the satellite

# Global positioning system

## Principal operation

- $\Delta_r$ : unknown deviation of the receiver's clock.
- $x_r, y_r, z_r$ : unknown coordinates of the receiver.
- $T_i$ : timestamp on a message from satellite  $i$
- $\Delta_i = (T_{now} - T_i) + \Delta_r$ : measured delay of the message sent by satellite  $i$ .
- Measured distance to satellite  $i$ :  $c \times \Delta_i$   
( $c$  is speed of light)
- Real distance is

$$d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

## Observation

4 satellites  $\Rightarrow$  4 equations in 4 unknowns (with  $\Delta_r$  as one of them)



# Global positioning system

## Principal operation

- $\Delta_r$ : **unknown deviation** of the receiver's clock.
- $x_r, y_r, z_r$ : **unknown coordinates** of the receiver.
- $T_i$ : timestamp on a message from satellite  $i$
- $\Delta_i = (T_{now} - T_i) + \Delta_r$ : **measured delay** of the message sent by satellite  $i$ .
- **Measured distance** to satellite  $i$ :  $c \times \Delta_i$   
( $c$  is speed of light)
- Real distance is

$$d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

## Observation

4 satellites  $\Rightarrow$  4 equations in 4 unknowns (with  $\Delta_r$  as one of them)

# Global positioning system

## Principal operation

- $\Delta_r$ : unknown deviation of the receiver's clock.
- $x_r, y_r, z_r$ : unknown coordinates of the receiver.
- $T_i$ : timestamp on a message from satellite  $i$
- $\Delta_i = (T_{now} - T_i) + \Delta_r$ : measured delay of the message sent by satellite  $i$ .
- Measured distance to satellite  $i$ :  $c \times \Delta_i$   
( $c$  is speed of light)
- Real distance is

$$d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

## Observation

4 satellites  $\Rightarrow$  4 equations in 4 unknowns (with  $\Delta_r$  as one of them)

# Global positioning system

## Principal operation

- $\Delta_r$ : unknown deviation of the receiver's clock.
- $x_r, y_r, z_r$ : unknown coordinates of the receiver.
- $T_i$ : timestamp on a message from satellite  $i$
- $\Delta_i = (T_{now} - T_i) + \Delta_r$ : measured delay of the message sent by satellite  $i$ .
- Measured distance to satellite  $i$ :  $c \times \Delta_i$   
( $c$  is speed of light)
- Real distance is

$$d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

## Observation

4 satellites  $\Rightarrow$  4 equations in 4 unknowns (with  $\Delta_r$  as one of them)

# Global positioning system

## Principal operation

- $\Delta_r$ : unknown deviation of the receiver's clock.
- $x_r, y_r, z_r$ : unknown coordinates of the receiver.
- $T_i$ : timestamp on a message from satellite  $i$
- $\Delta_i = (T_{now} - T_i) + \Delta_r$ : measured delay of the message sent by satellite  $i$ .
- Measured distance to satellite  $i$ :  $c \times \Delta_i$   
( $c$  is speed of light)
- Real distance is

$$d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

## Observation

4 satellites  $\Rightarrow$  4 equations in 4 unknowns (with  $\Delta_r$  as one of them)

# Global positioning system

## Principal operation

- $\Delta_r$ : unknown deviation of the receiver's clock.
- $x_r, y_r, z_r$ : unknown coordinates of the receiver.
- $T_i$ : timestamp on a message from satellite  $i$
- $\Delta_i = (T_{now} - T_i) + \Delta_r$ : measured delay of the message sent by satellite  $i$ .
- Measured distance to satellite  $i$ :  $c \times \Delta_i$   
( $c$  is speed of light)
- Real distance is

$$d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

## Observation

4 satellites  $\Rightarrow$  4 equations in 4 unknowns (with  $\Delta_r$  as one of them)

# Global positioning system

## Principal operation

- $\Delta_r$ : unknown deviation of the receiver's clock.
- $x_r, y_r, z_r$ : unknown coordinates of the receiver.
- $T_i$ : timestamp on a message from satellite  $i$
- $\Delta_i = (T_{now} - T_i) + \Delta_r$ : measured delay of the message sent by satellite  $i$ .
- Measured distance to satellite  $i$ :  $c \times \Delta_i$   
( $c$  is speed of light)
- Real distance is

$$d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

## Observation

4 satellites  $\Rightarrow$  4 equations in 4 unknowns (with  $\Delta_r$  as one of them)

# Global positioning system

## Principal operation

- $\Delta_r$ : unknown deviation of the receiver's clock.
- $x_r, y_r, z_r$ : unknown coordinates of the receiver.
- $T_i$ : timestamp on a message from satellite  $i$
- $\Delta_i = (T_{now} - T_i) + \Delta_r$ : measured delay of the message sent by satellite  $i$ .
- Measured distance to satellite  $i$ :  $c \times \Delta_i$   
( $c$  is speed of light)
- Real distance is

$$d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

## Observation

4 satellites  $\Rightarrow$  4 equations in 4 unknowns (with  $\Delta_r$  as one of them)

# Clock synchronization principles

## Principle I

Every machine asks a [time server](#) for the accurate time at least once every  $\delta/(2\rho)$  seconds ([Network Time Protocol](#)).

## Note

Okay, but you need an accurate measure of round trip delay, including interrupt handling and processing incoming messages.



# Clock synchronization principles

## Principle II

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**.

## Note

Okay, you'll probably get every machine in sync. You don't even need to propagate UTC time.

## Fundamental

You'll have to take into account that setting the time back is **never** allowed  $\Rightarrow$  smooth adjustments.

# 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.

# 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.

# 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 clocks

## 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 clocks

## 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 clocks

## 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 clocks

## Solution

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

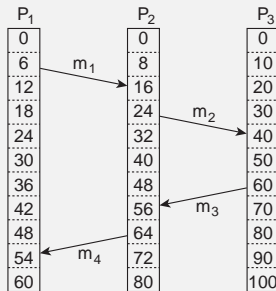
- 1: For any two **successive events** that take place within  $P_i$ ,  $C_i$  is incremented by 1.
- 2: Each time a message  $m$  is **sent** by process  $P_i$ , the message receives a timestamp  $ts(m) = C_i$ .
- 3: Whenever a message  $m$  is **received** by a process  $P_j$ ,  $P_j$  adjusts its local counter  $C_j$  to  **$\max\{C_j, ts(m)\}$** ; then executes step 1 before passing  $m$  to the application.

## Notes

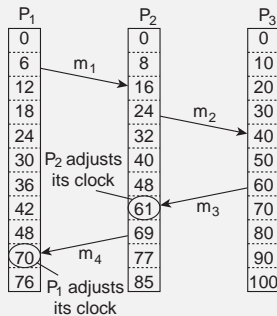
- Property **P1** is satisfied by (1); Property **P2** by (2) and (3).
- It can still occur that two events happen at the same time. Avoid this by **breaking ties through process IDs**.



# Logical clocks – example



(a)

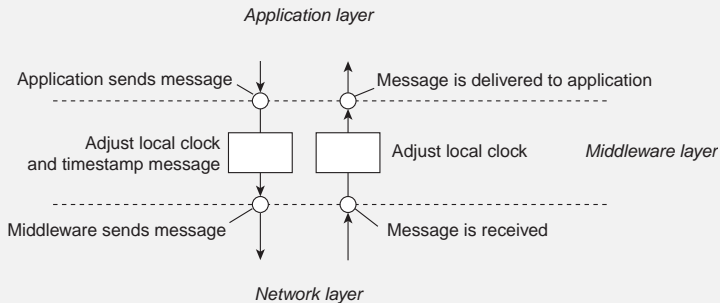


(b)

# Logical clocks – example

## Note

Adjustments take place in the middleware layer

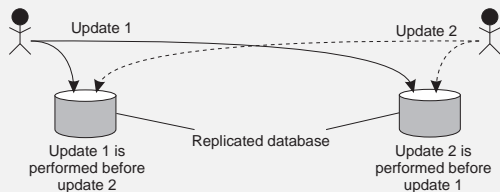


# Example: Totally ordered multicast

## 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**.

# 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**.

# 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.

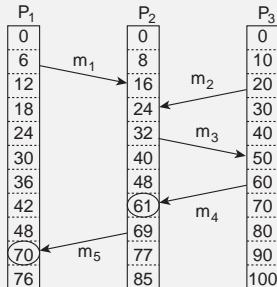
## Note

We are assuming that communication is **reliable** and **FIFO ordered**.

# Vector clocks

## 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 clocks

## 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?



# Vector clocks

## 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?

# Causally ordered multicasting

## Observation

We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.

## Adjustment

$P_i$  increments  $VC_i[i]$  only when sending a message, and  $P_j$  “adjusts”  $VC_j$  when receiving a message (i.e., effectively does not change  $VC_j[j]$ ).

$P_j$  postpones delivery of  $m$  until:

- $ts(m)[i] = VC_j[i] + 1$ .
- $ts(m)[k] \leq VC_j[k]$  for  $k \neq i$ .

# Causally ordered multicasting

## Observation

We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.

## Adjustment

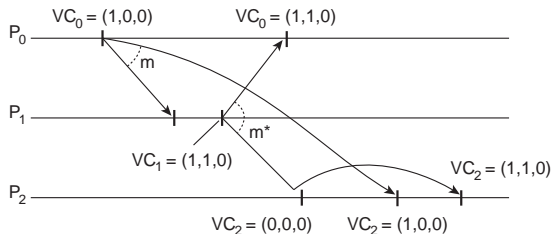
$P_i$  increments  $VC_i[i]$  only when sending a message, and  $P_j$  “adjusts”  $VC_j$  when receiving a message (i.e., effectively does not change  $VC_j[j]$ ).

$P_j$  postpones delivery of  $m$  until:

- $ts(m)[i] = VC_j[i] + 1$ .
- $ts(m)[k] \leq VC_j[k]$  for  $k \neq i$ .

# Causally ordered multicasting

## Example

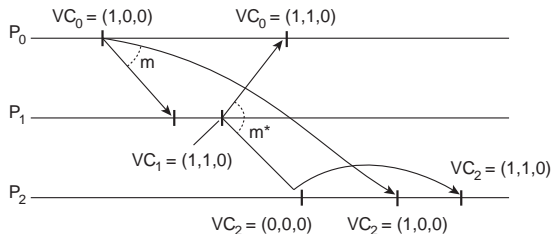


## Example

Take  $VC_2 = [0, 2, 2]$ ,  $ts(m) = [1, 3, 0]$  from  $P_0$ . What information does  $P_2$  have, and what will it do when receiving  $m$  (from  $P_0$ )?

# Causally ordered multicasting

## Example



## Example

Take  $VC_2 = [0, 2, 2]$ ,  $ts(m) = [1, 3, 0]$  from  $P_0$ . What information does  $P_2$  have, and what will it do when receiving  $m$  (from  $P_0$ )?

# Mutual exclusion

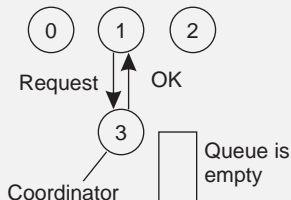
## Problem

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

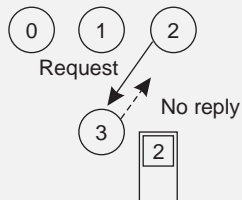
## Basic solutions

- Via a centralized server.
- Completely decentralized, using a peer-to-peer system.
- Completely distributed, with no topology imposed.
- Completely distributed along a (logical) ring.

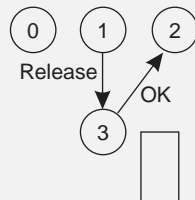
# Mutual exclusion: centralized



(a)



(b)



(c)

# 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

## Issue

How robust is this system? Let  $p = \Delta t / T$  denote the probability that a coordinator crashes and recovers in a period  $\Delta t$  while having an average lifetime  $T \Rightarrow$  probability that  $k$  out of  $m$  coordinators **reset**:

$$P[\text{violation}] = p_v = \sum_{k=2m-n}^n \binom{m}{k} p^k (1-p)^{m-k}$$

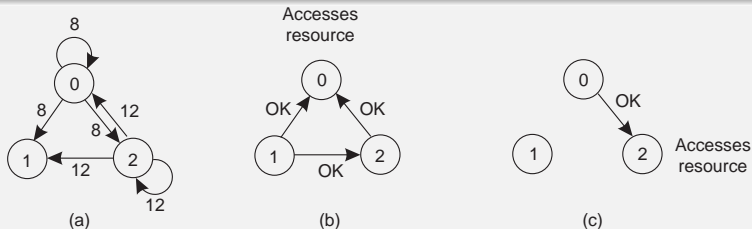
With  $p = 0.001$ ,  $n = 32$ ,  $m = 0.75n$ ,  $p_v < 10^{-40}$

# Mutual exclusion Ricart & Agrawala

## Principle

The same as Lamport except that acknowledgments aren't sent. Instead, replies (i.e. grants) are sent only when

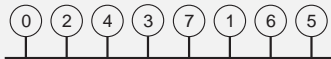
- The receiving process has no interest in the shared resource; or
- The receiving process is waiting for the resource, but has lower priority (known through comparison of timestamps).
- In all other cases, reply is **deferred**, implying some more local administration.



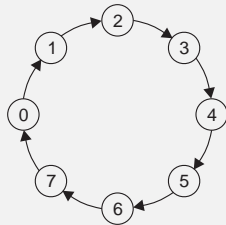
# Mutual exclusion: Token ring algorithm

## 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).



(a)



(b)

# Mutual exclusion: comparison

Algorithm	# msgs per entry/exit	Delay before entry (in msg times)	Problems
Centralized	3	2	Coordinator crash
Decentralized	$2mk + m$ , $k = 1, 2, \dots$	$2mk$	Starvation, low eff.
Distributed	$2(n - 1)$	$2(n - 1)$	Crash of any process
Token ring	1 to $\infty$	0 to $n - 1$	Lost token, proc. crash

# Global positioning of nodes

## Problem

How can a single node efficiently estimate the latency between any two other nodes in a distributed system?

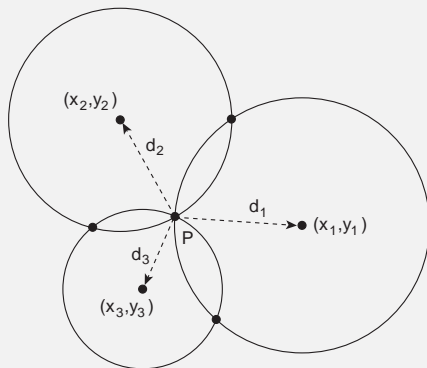
## Solution

Construct a geometric overlay network, in which the distance  $d(P, Q)$  reflects the actual latency between  $P$  and  $Q$ .

# Computing position

## Observation

A node  $P$  needs  $k + 1$  **landmarks** to compute its own position in a  $d$ -dimensional space. Consider two-dimensional case.



## Solution

$P$  needs to solve three equations in two unknowns  $(x_P, y_P)$ :

$$d_i = \sqrt{(x_i - x_P)^2 + (y_i - y_P)^2}$$

# Computing position

## Problems

- measured latencies to landmarks fluctuate
- computed distances will not even be consistent:



## Solution

Let the  $L$  landmarks measure their pairwise latencies  $d(b_i, b_j)$  and let each node  $P$  minimize

$$\sum_{i=1}^L \left[ \frac{d(b_i, P) - \hat{d}(b_i, P)}{d(b_i, P)} \right]^2$$

where  $\hat{d}(b_i, P)$  denotes the distance to landmark  $b_i$  given a **computed coordinate** for  $P$ .

# Computing position

## Problems

- measured latencies to landmarks fluctuate
- computed distances will not even be consistent:



## Solution

Let the  $L$  landmarks measure their pairwise latencies  $d(b_i, b_j)$  and let each node  $P$  minimize

$$\sum_{i=1}^L \left[ \frac{d(b_i, P) - \hat{d}(b_i, P)}{d(b_i, P)} \right]^2$$

where  $\hat{d}(b_i, P)$  denotes the distance to landmark  $b_i$  given a **computed coordinate** for  $P$ .



# 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.

## Question

If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?

## Question

Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?

# 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.

## Question

If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?

## Question

Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?

# 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.

## Question

If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?

## Question

Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?

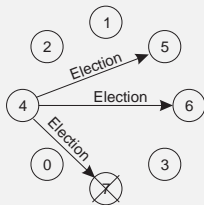
# Election by bullying

## Principle

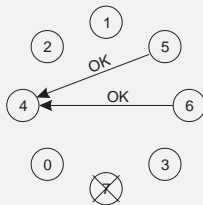
Each process has an associated priority (weight). The process with the highest priority should always be elected as the coordinator. **Issue**  
How do we find the heaviest process?

- Any process can just start an election by sending an election message to all other processes (assuming you don't know the weights of the others).
- If a process  $P_{heavy}$  receives an election message from a lighter process  $P_{light}$ , it sends a take-over message to  $P_{light}$ .  $P_{light}$  is out of the race.
- If a process doesn't get a take-over message back, it wins, and sends a victory message to all other processes.

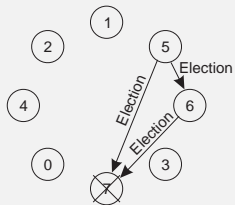
# Election by bullying



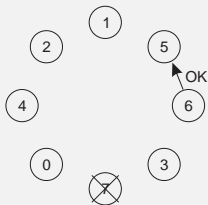
(a)



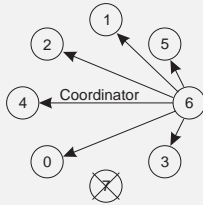
(b)



(c)



(d)



(e)

# 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

## Question

Does it matter if two processes initiate an election?

## Question

What happens if a process crashes *during* the election?

# Superpeer election

## Issue

How can we select **superpeers** such that:

- Normal nodes have low-latency access to superpeers
- Superpeers are evenly distributed across the overlay network
- There is be a predefined fraction of superpeers
- Each superpeer should not need to serve more than a fixed number of normal nodes



# Superpeer election

## DHTs

Reserve a fixed part of the ID space for superpeers. **Example:** if  $S$  superpeers are needed for a system that uses  $m$ -bit identifiers, simply reserve the  $k = \lceil \log_2 S \rceil$  leftmost bits for superpeers. With  $N$  nodes, we'll have, on average,  $2^{k-m}N$  superpeers.

## Routing to superpeer

Send message for key  $p$  to node responsible for

$p$  AND  $\underbrace{11 \dots 11}_k \underbrace{00 \dots 00}_{m-k}$